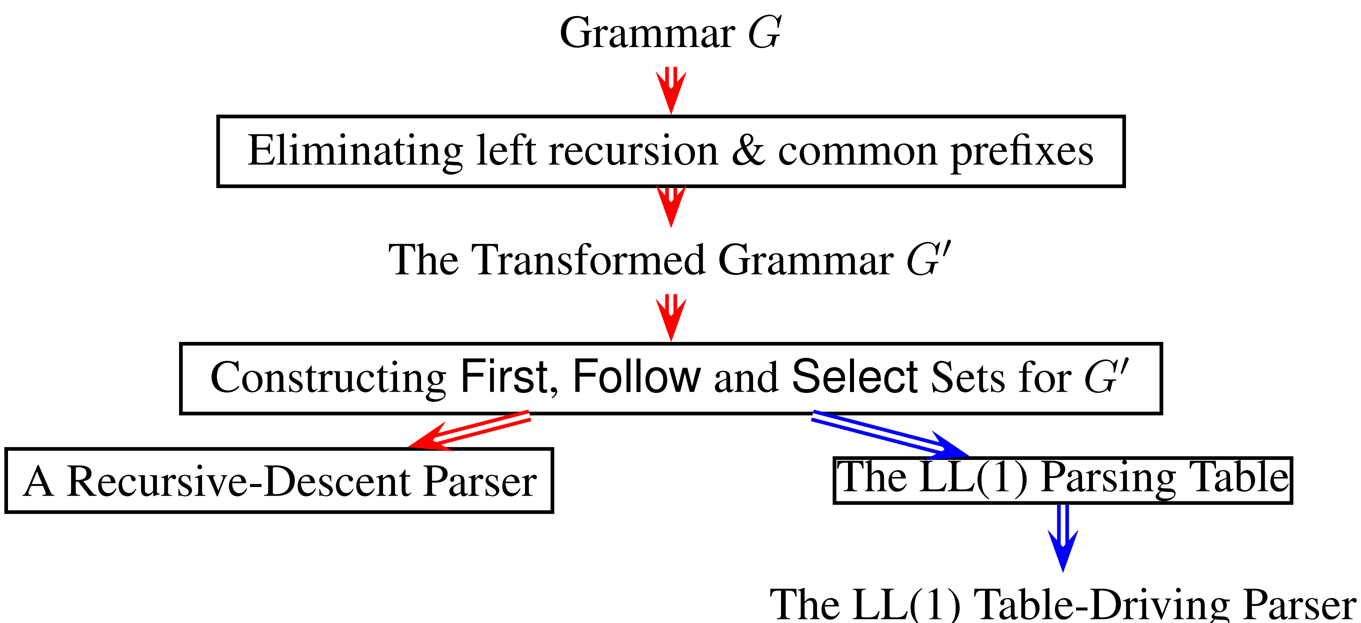
COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
`http://www.cse.unsw.edu.au/~cs3131`
`http://www.cse.unsw.edu.au/~cs9102`

Week 9 (2nd Lecture): Table-Driven Top-Down Parsing

1. LL(1) table-driven parsing
2. Parser generators
3. Error recovery



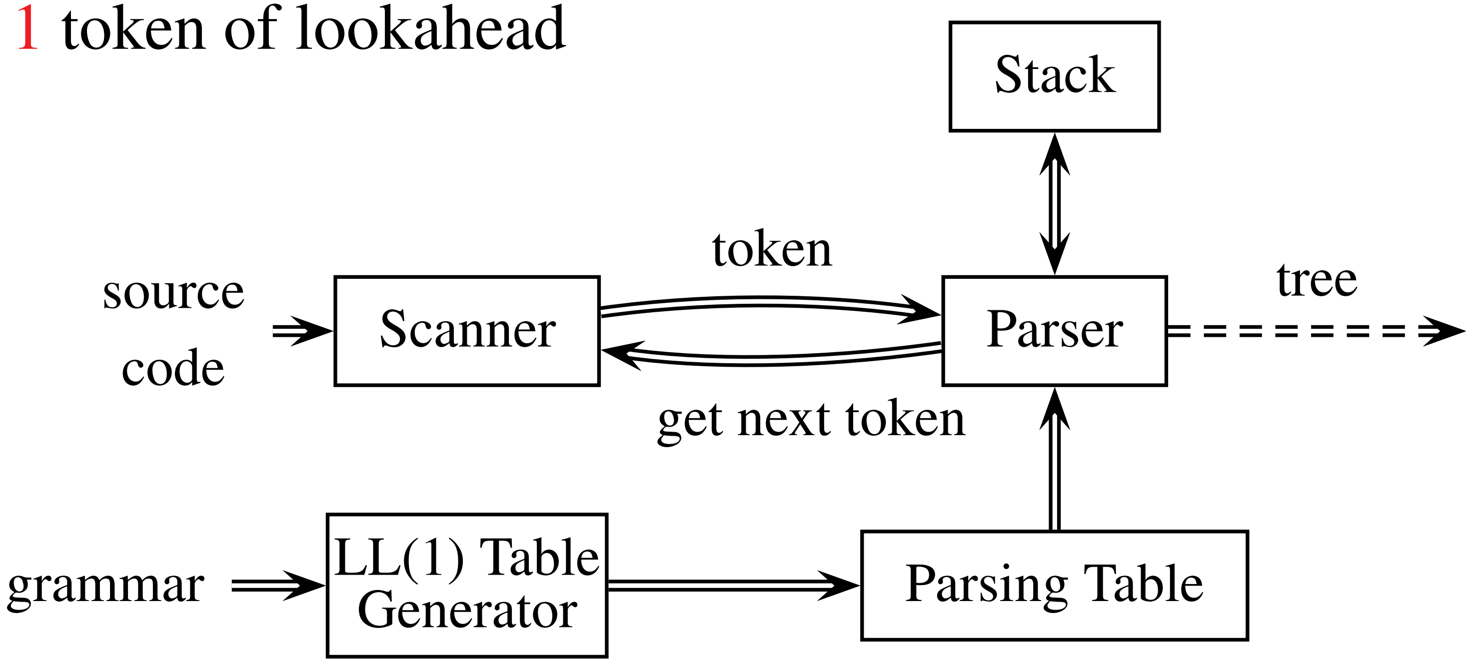
- The red parts done last week
- The the blue parts today

Predictive Non-Recursive Top-Down Parsers

- Recursion = Iteration + Stack
- Recursive calls in a recursive-descent parser can be implemented using
  - an explicit stack, and
  - a parsing table
- Understanding one helps your understanding the other

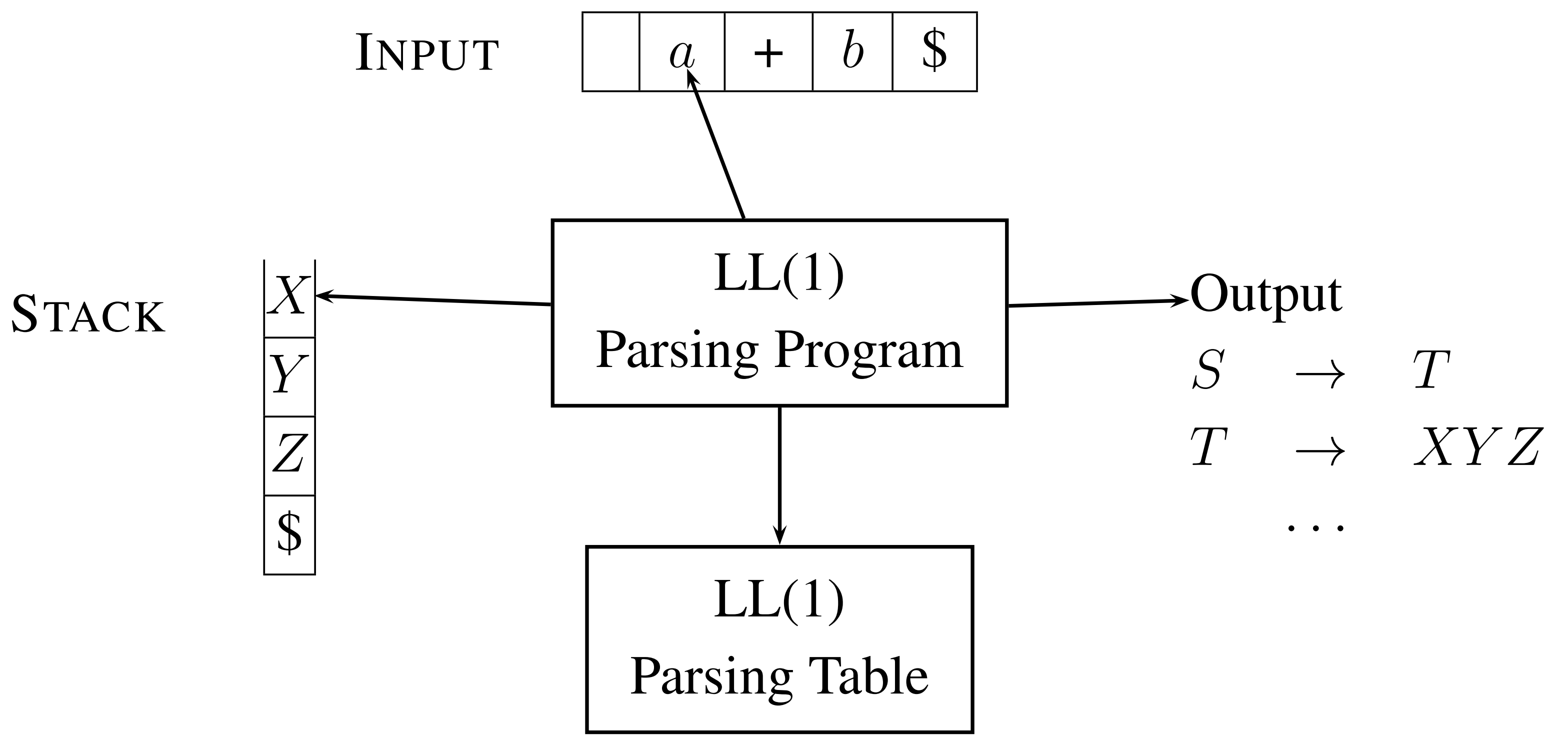
The Structure of a Table-Driven LL(1) Parser

- Input parsed from left to right
- Leftmost derivation
- 1 token of lookahead



- LR(1) parsers (almost always table-driven) also built this way

The Model of an LL(1) Table-Driven Parser



Output:

- The productions used (representing the leftmost derivation), or
- An AST (built by a parser (Assignment 3))

The LL(1) Parsing Program

Push \$ onto the stack

Push the start symbol onto the stack

WHILE (stack not empty) **DO**

  BEGIN

    Let X be the top stack symbol and a be the lookahead symbol in the input

    IF X is a terminal **THEN**

      IF $X = a$ then pop X and get the next token /* match */

      ELSE error

    ELSE /$*$ X is a nonterminal $*$/

      IF $Table[X, a]$ nonblank **THEN**

        Pop X

        Push $Table[X, a]$ onto stack in the reverse order

      ELSE error

  END

The parsing is successful when the stack is empty and no errors reported

Building a Table-Driving Parser from a Grammar

Grammar G

$\Downarrow$

Eliminating left recursion & common prefixes (Slide 207)

$\Downarrow$

The Transformed Grammar G'

$\Downarrow$

Constructing First, Follow and Select Sets for G'

$\Downarrow$

Constructing the LL(1) Parsing Table

$\Downarrow$

The LL(1) Table-Driving Parser

- Follow Slide 207 to eliminate left recursion to get a BNF grammar
- Can build a table-driven parser for EBNF as well (but not considered)

The Expression Grammar

- The grammar with left recursion:

$$
\text{Grammar 1:} \quad
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow \textbf{INT} \mid (E)
\end{aligned}
$$

- The transformed grammar without left recursion:

$$
\text{Grammar 2:} \quad
\begin{aligned}
E &\rightarrow TQ \\
Q &\rightarrow +TQ \mid -TQ \mid \epsilon \\
T &\rightarrow FR \\
R &\rightarrow *FR \mid /FR \mid \epsilon \\
F &\rightarrow \textbf{INT} \mid (E)
\end{aligned}
$$

First and Follow Sets for Grammar 2

- First sets:

$$
\begin{array}{rcl}
\text{First}(TQ) = \text{First}(FR) & = & \{(, \mathbf{i}\} \\
\text{First}(Q) & = & \{(+, -, \epsilon\} \\
\text{First}(R) & = & \{(*, /, \epsilon\} \\
\text{First}(+TQ) & = & \{+\} \\
\text{First}(-TQ) & = & \{-\} \\
\text{First}(*FR) & = & \{*\} \\
\text{First}(/FR) & = & \{/\} \\
\text{First}((E)) & = & \{(\} \\
\text{First}(\mathbf{i}) & = & \{\mathbf{i}\}
\end{array}
$$

- Follow sets:

$$
\begin{array}{rcl}
\text{Follow}(E) & = & \{\$,)\} \\
\text{Follow}(Q) & = & \{\$,)\} \\
\text{Follow}(T) & = & \{+, -, \$,)\} \\
\text{Follow}(R) & = & \{+, -, \$,)\} \\
\text{Follow}(F) & = & \{+, -, *, /, \$,)\}
\end{array}
$$

Select Sets for Grammar 2

$$
\begin{array}{lll}
\text{Select}(E\to TQ) & = \text{First}(TQ) & = \{(, \textbf{INT}\} \\
\text{Select}(Q\to +TQ) & = \text{First}(+TQ) & = \{+\} \\
\text{Select}(Q\to -TQ) & = \text{First}(-TQ) & = \{-\} \\
\text{Select}(Q\to\epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(Q) & = \{), \$\} \\
\text{Select}(T\to FR) & = \text{First}(FR) & = \{(, \textbf{INT}\} \\
\text{Select}(R\to *FR) & = \text{First}(+FR) & = \{*\} \\
\text{Select}(R\to /FR) & = \text{First}(/FR) & = \{/\} \\
\text{Select}(R\to\epsilon) & = (\text{First}(\epsilon) - \{\epsilon\}) \cup \text{Follow}(T) & = \{+, -,), \$\} \\
\text{Select}(F\to\textbf{INT}) & = \text{First}(\textbf{INT}) & = \{\textbf{INT}\} \\
\text{Select}(F\to(E)) & = \text{First}((E)) & = \{(\}
\end{array}
$$

The Rules for Constructing an LL(1) Parsing Table

For every production of the $A \rightarrow \alpha$ in the grammar, do:

for all a in Select$(A \rightarrow \alpha)$, set $Table[A, a] = \alpha$

LL(1) Parsing Table for Grammar 2

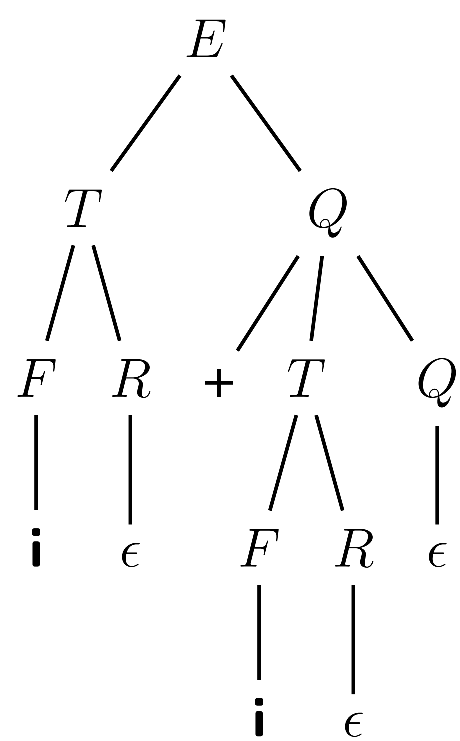
	INT	+	$-$	$*$	/	(	)	\$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ
F	**INT**					(E)		

The blanks are errors.

An LL(1) Parse on Input **i+i**: **INT** $\iff$ **i**

STACK	INPUT	PRODUCTION	DERIVATION
$\$E$	**i+i**$	$E \to TQ$	$E \Longrightarrow_{\text{lm}} TQ$
$\$QT$	**i+i**$	$T \to FR$	$\Longrightarrow_{\text{lm}} FRQ$
$\$QRF$	**i+i**$	$F \to \mathbf{i}$	$\Longrightarrow_{\text{lm}} \mathbf{i}RQ$
$\$QR\mathbf{i}$	**i+i**$	pop and go to next token	
$\$QR$	**+i**$	$R \to \epsilon$	$\Longrightarrow_{\text{lm}} \mathbf{i}Q$
$\$Q$	**+i**$	$Q \to +TQ$	$\Longrightarrow_{\text{lm}} \mathbf{i}+TQ$
$\$QT+$	**+i**$	pop and go to next token	
$\$QT$	**i**$	$T \to FR$	$\Longrightarrow_{\text{lm}} \mathbf{i}+FRQ$
$\$QRF$	**i**$	$F \to \mathbf{i}$	$\Longrightarrow_{\text{lm}} \mathbf{i}+\mathbf{i}RQ$
$\$QR\mathbf{i}$	**i**$	pop and go to next token	
$\$QR$	$	$R \to \epsilon$	$\Longrightarrow_{\text{lm}} \mathbf{i}+\mathbf{i}RQ$
$\$Q$	$	$Q \to \epsilon$	$\Longrightarrow_{\text{lm}} \mathbf{i}+\mathbf{i}Q$
$	$		

PARSE TREE



An LL(1) Parse on an Erroneous Input "()"

STACK	INPUT	PRODUCTION	DERIVATION
$\$E$	()\$	$E \to TQ$	$E \Longrightarrow_{\text{lm}} TQ$
$\$QT$	()\$	$T \to FR$	$E \Longrightarrow_{\text{lm}} FRQ$
$\$QRF$	()\$	$F \to (E)$	$E \Longrightarrow_{\text{lm}} (E)RQ$
$\$QR)E($	()\$	pop and go to next token	
$\$QR)E$	)\$	$***$ Error: no table entry for $[E,)]$	

A better error message: expression missing inside ( )

LL(1) Grammars and Table-Driven LL(1) Parsers

- Like recursive descent, table-driven LL(1) parsers can only parse LL(1) grammars. Conversely, only LL(1) grammars can be parsed by the table-driven LL(1) parsers.
- Definition of LL(1) grammar given in Slide 188
- Definition of LL(1) grammar – using the parsing table:
  A grammar is LL(1) if every table entry contains at most one production.

Why Table-Driven LL(1) Parsers Cannot Handle Left Recursions?

- A grammar with left recursion:

$$\langle \text{expr} \rangle \quad \to \quad \langle \text{expr} \rangle + \textbf{id} \;\mid\; \textbf{id}$$

- Select Sets:

$$\begin{array}{lcl} \text{Select}(\langle \text{expr} \rangle + \textbf{id}) & = & \{\textbf{id}\} \\ \text{Select}(\textbf{id}) & = & \{\textbf{id}\} \end{array}$$

- The parsing table:

	id	$
⟨expr⟩	⟨expr⟩ + **id** **id**	

$Table[\langle \text{expr} \rangle, \textbf{id}]$ contains two entries!

- Any grammar with left recursions is not LL(1)

Why Table-Driven LL(1) Parsers Cannot Handle Left Recursions (Cont'd)?

- Eliminating the left recursion yields an LL(1) grammar:

$$\begin{array}{lcl} \langle \text{expr} \rangle & \rightarrow & \textbf{id}\ \langle \text{expr-tail} \rangle \\ \langle \text{expr-tail} \rangle & \rightarrow & \epsilon \mid + \textbf{id}\ \langle \text{expr-tail} \rangle \end{array}$$

- Select Sets:

$$\begin{array}{lcl} \mathsf{Select}\langle \text{expr} \rangle \rightarrow \textbf{id}\ \langle \text{expr-tail} \rangle) & = & \{\textbf{id}\} \\ \mathsf{Select}(\langle \text{expr-tail} \rangle \rightarrow \epsilon) = & = & \{\$\} \\ \mathsf{Select}(\langle \text{expr-tail} \rangle \rightarrow + \textbf{id}\ \langle \text{expr-tail} \rangle) & = & \{+\} \end{array}$$

- The parsing table for the transformed grammar:

	id	+	$
⟨expr⟩	**id** ⟨expr-tail⟩		
⟨expr-tail⟩		+ **id** ⟨expr-tail⟩	ϵ

Why LL(1) Table-Driven Parsers Cannot Handle Common Prefixes?

- A grammar with a common prefix:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if}\ (E)\ S\ \mid \textbf{if}\ (E)\ S\ \textbf{else}\ S\ \mid \textbf{s} \\ E & \rightarrow & e \end{array}$$

- Select sets:

$$\begin{array}{lcl} \text{Select}(S{\rightarrow}\textbf{if}\ (E)\ \ S) & = & \{\textbf{if}\} \\ \text{Select}(S{\rightarrow}\textbf{if}\ (E)\ \ S\ \textbf{else}\ S) & = & \{\textbf{if}\} \end{array}$$

- Any grammar with common prefixes is not LL(1)
- Eliminating the common prefix does not yield an LL(1) grammar:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if}\ (E)\ SQ\ \mid s \\ Q & \rightarrow & \textbf{else}\ S\ \mid \epsilon \\ E & \rightarrow & e \end{array}$$

Why LL(1) Table-Driven Parsers Cannot Handle Common Prefixes (Cont'd)?

- Select sets:

$$
\begin{array}{lcl}
\mathsf{Select}(S \to \textbf{if}(E)SQ) & = & \{\textbf{if}\} \\
\mathsf{Select}(S \to s) & = & \{s\} \\
\mathsf{Select}(Q \to \textbf{else}\,S) & = & \{\textbf{else}\} \\
\mathsf{Select}(Q \to \epsilon) & = & \{\textbf{else}, \epsilon\} \\
\mathsf{Select}(E \to e) & = & \{e\}
\end{array}
$$

- The parsing table:

	if	(	e	)	s	**else**	\$
S	**if** E **then** SQ				s		
E			e				
Q						**else** S ϵ	ϵ

- This modified grammar, although having no common prefixes, is still ambiguous. You are referred to Week 6 Tutorial. To resolve the ambiguity in the grammar, we make the convention to select **else** S as the table entry. This effectively implements the following rule:

  Match an **else** to the most recent unmatched **then**

Recognise Palindromes Easily

- Grammar:

$$S \rightarrow (S) \mid \epsilon$$

- Parsing Table:

	(	)	\$
S	(S)	ϵ	ϵ

- Try to parse the following three inputs:

```
a. (())
b. (()
c. ()))
```

- Cannot design a DFA/NFA to recognise the language $L(S)$

Week 9 (2nd Lecture): Table-Driven Top-Down Parsing

1. LL(1) table-driven parsing $\surd$
2. Parser generators

The Expression Grammar

- The grammar with left recursion:

$$\text{Grammar 1:}\quad \begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to \textsf{\textbf{INT}} \mid (E) \end{aligned}$$

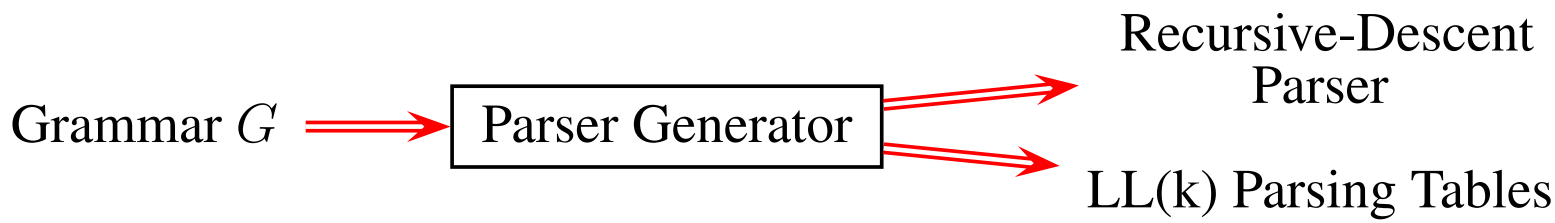
- Eliminating left recursion using the Kleene Closure

$$\text{Grammar 3:}\quad \begin{aligned} E &\to T\ (\text{"+"}\ T \mid \text{"-"}\ T)^* \\ T &\to F\ (\text{"*"}\ F \mid \text{"/"}\ F)^* \\ F &\to \textsf{\textbf{INT}} \mid \text{"("}\ E\ \text{")"} \end{aligned}$$

  All tokens are enclosed in double quotes to distinguish them for the regular operators: (,) and *

- Compare with Slide **??**

Parser Generators (Generating Top-Down Parsers)



Tool	Grammar Accepted	Parsers and Their Implementation Languages
JavaCC	EBNF	Recursive-Descent LL(1) (with some LL(k) portions) in Java
COCO/R	EBNF	Recursive-Descent LL(1) in Pascal, C, C++,, Java, etc.
ANTLR	Predicated LL(k)	Recursive-Descent LL(k) in C, C++,, Java

- These and other tools can be found on the internet
- Predicated: a conditional evaluated by the parser at run time to determine which of the two conflicting productions to use

$$Q \quad \rightarrow \boxed{\text{if (lookahead is "else")}}\ \textbf{else}\ S \mid \epsilon$$

where the condition inside the box resolves the dangling-else problem.

Parser Generators (Generating Bottom-Up Parsers)

Grammar G $\Rightarrow$ $\boxed{\text{Parser Generator}}$ $\Rightarrow$ LALR(1) Parsing Tables

Tool	Grammar Accepted	Parsers and Their Implementation Languages
Yacc	BNF	LALR(1) table-driven in C
JavaCUP	BNF	LALR(1) table-driven in Java

- These and other tools can be found on the internet
- Will not deal with LR parsing in this course

The JavaCC Spec for Grammar 3

```
/*
 * Parser.jj
 *
 * The scanner and parser for Grammar 3
 *
 * Install JavaCC from https://javacc.dev.java.net/
 *
 * 1. javacc Parser.jj
 * 2. javac Parser.java
 * 3. java Parser
 */

options {
LOOKAHEAD=1;
}

PARSER_BEGIN(Parser)

public class Parser {

    public static void main(String args[]) throws ParseException {
        Parser parser = new Parser (System.in);
        while (true) {
            System.out.print("Enter Expression: ");
            System.out.flush();
            try {
                switch (parser.one_line()) {
```

```
                    case -1:
                        System.exit(0);
                    default:
                        System.out.println("Compilation was successful.");
                        break;
                }
            } catch (ParseException x) {
                System.out.println("Exiting.");
                throw x;
            }
        }
    }
}
PARSER_END(Parser)

SKIP :
{
        " "
|       "\r"
|       "\t"
}

TOKEN :
{
        < EOL: "\n" >
}

TOKEN : /* OPERATORS */
{
```

```
        < PLUS: "+" >
|       < MINUS: "-" >
|       < MULTIPLY: "*" >
|       < DIVIDE: "/" >
}

TOKEN :
{
    < CONSTANT: ( <DIGIT> )+ >
|   < #DIGIT: ["0" - "9"] >
}

int one_line() :
{}
{
    Expr() <EOL>
        { return 1; }
  | <EOL>
        { return 0; }
  | <EOF>
        { return -1; }
}

void Expr() :
{ }
{
    Term() (( <PLUS> | <MINUS> ) Term())*
}
```

```
void Term() :
{ }
{
    Factor() (( <MULTIPLY> | <DIVIDE> ) Factor())*
}

void Factor() :
{}
{
    <CONSTANT>
|   "(" Term() ")"
}
```

Lecture 5: Top-Down Parsing: Table-Driven

1. LL(1) table-driven parsing $\surd$
2. Parser generators $\surd$

Reading

- Table-driven parsing: pages 186 – 192 of Red Dragon or §4.4.4 of Purple Dragon
- JavaCC, ANTLR and COCO/R: available on the Web
- JavaCUP also available on the Web

  `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

- Error recovery for LL parsers:
  - Using acceptance sets:
    - Pages 192 – 195 of Red Dragon / Pages §4.4.5 of Purple Dragon
    - `http://teaching.idallen.com/cst8152/98w/panic_mode.html`
  - Using continuation (pages 136 – 142 of Grune et al's 2000 compiler book. ISBN: 0-471-97697-0)

Next Week: Easter Holiday (1st Lecture) and JVM + Revision (2nd Lecture)