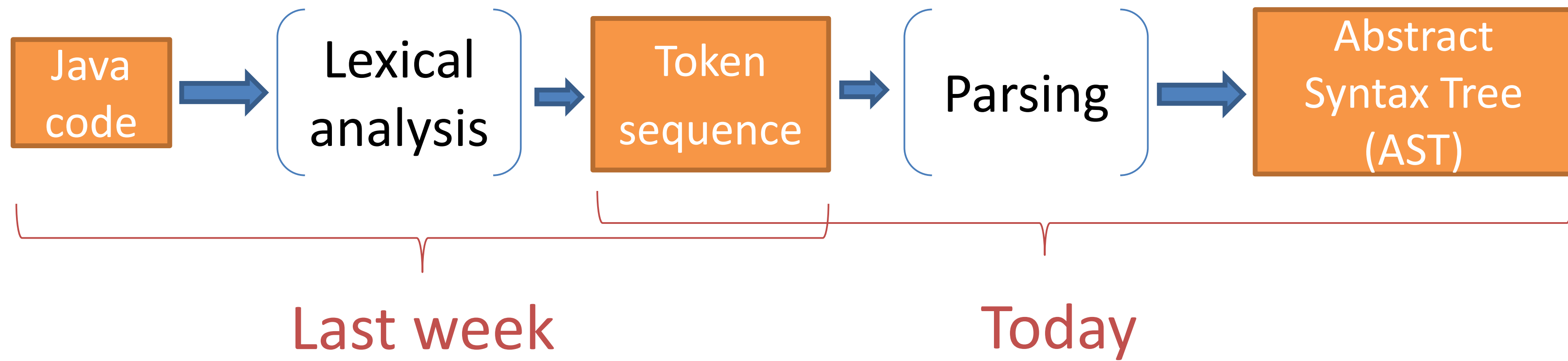
Compiler Construction
Winter 2020

Recitation 10:
Parsing

Yotam Feldman

Based on slides by Guy Golan-Gueta
and the Technion compilers class' staff

Lexing & Parsing



MiniJava Grammar

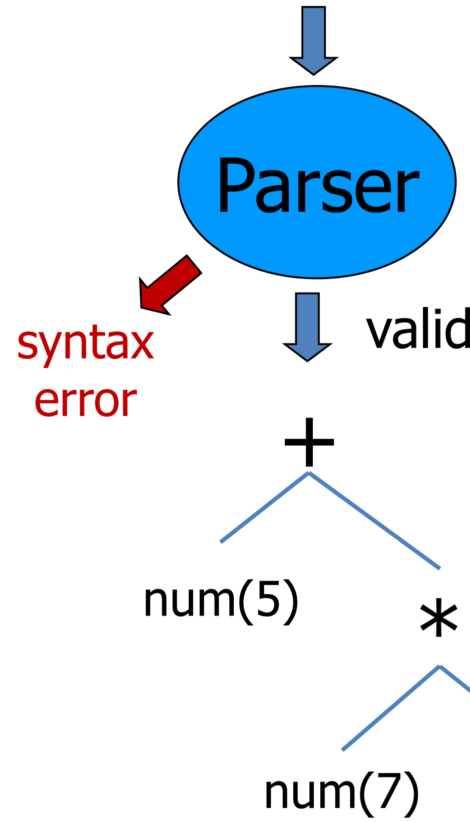
```
Goal ::= MainClass ( ClassDeclaration )* <EOF>
MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")" "{" Statement "}" "}"
ClassDeclaration ::= "class" Identifier ( "extends" Identifier )? "{" ( VarDeclaration )* ( MethodDeclaration )* "}"
VarDeclaration ::= Type Identifier ";"
MethodDeclaration ::= "public" Type Identifier "(" ( Type Identifier ( "," Type Identifier )* )? ")" "{" ( VarDeclaration )* ( Statement )* "return" Expression ";" "}"
Type ::= "int" "[" "]"
       | "boolean"
       | "int"
       | Identifier
Statement ::= "{" ( Statement )* "}"
       | "if" "(" Expression ")" Statement "else" Statement
       | "while" "(" Expression ")" Statement
       | "System.out.println" "(" Expression ")" ";"
       | Identifier "=" Expression ";"
       | Identifier "[" Expression "]" "=" Expression ";"
Expression ::= Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression
       | Expression "[" Expression "]"
       | Expression "." "length"
       | Expression "." Identifier "(" ( Expression ( "," Expression )* )? ")"
       | <INTEGER_LITERAL>
       | "true"
```

…

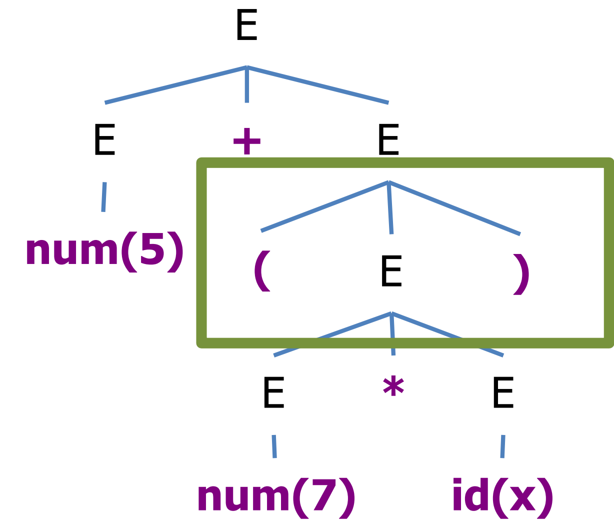
Parsing

token stream

Grammar:
$E \rightarrow$ **id**
$E \rightarrow$ **num**
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow - E$
$E \rightarrow (E)$



Abstract syntax tree



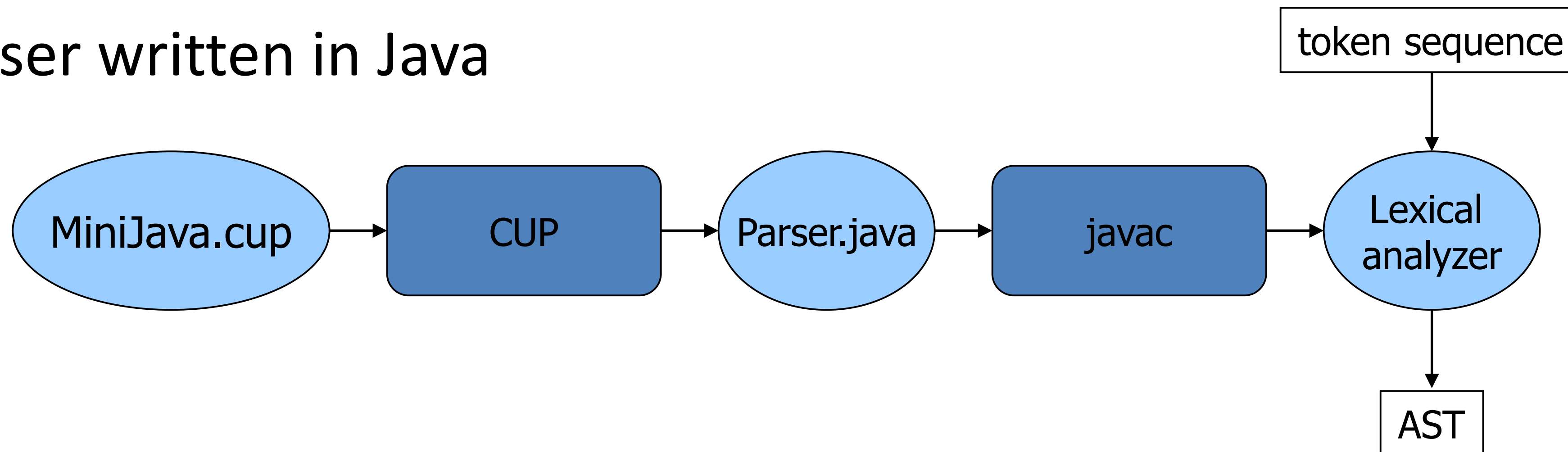
parse tree

Manual Solution

- Usually using recursive descent
- Not always easy to get right
- Not always easy to get efficient
  - Avoiding backtracking: predictive parsing, LL
- Alternative: table-based parsers

CUP

- **C**onstructor of **U**seful **P**arsers
- Automatic LALR(1) parser generator
- Input
  - Parser specification file
- Output
  - Parser written in Java



CUP Spec File

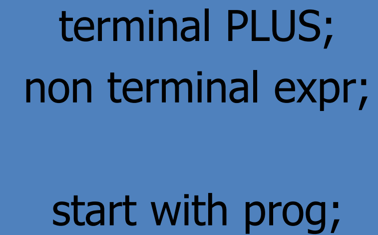
User code

Copied directly to Java file & customization

Terminal and non-terminals

over which the grammar is defined

tokens come from the lexer



Grammar

Parsing rules

Action when rules applied

Parser Code

```
import java_cup.runtime.*;
import ast.*;
import java.util.List;
import java.util.LinkedList;

parser code
{:
        public Lexer lexer;

        public Parser(Lexer lexer)
        {
                super(lexer);
                this.lexer = lexer;
        }
        public void report_error(String message, Object info)
        {
            System.err.print("Syntax error at line " + lexer.getLine() + " of input\n");
                System.exit(1);
        }
:}
```

```
scan with
{:
        Symbol s;
        s = lexer.next_token();
        return s;
:};
```

Lexer’s Interface with the Parser

```
public class sym {
    public static final int EOF = 0;
    public static final int ID = 1;
     ...
}
```

- Defines symbol constant ids
- Actual values don’t matter
  - Unique value for each token
- Auto-generated by CUP
- JFlex’s returns tokens using these constants (in JFlex actions):
  - `return sym.EOF;`

Grammar in CUP

```
terminal int NUMBER;
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;

non terminal expr;
```

```
start with expr;
```

```
expr ::= expr PLUS expr
       | expr MINUS expr
       | expr MULT expr
       | expr DIV expr
       | MINUS expr
       | LPAREN expr RPAREN
       | NUMBER;
```

Actions: On-the-Fly Calculation

```
terminal int NUMBER;
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;

non terminal int expr;
```

```
expr ::= expr:e1 PLUS expr:e2       {: RESULT = e1 + e2; :}
       | expr:e1 MINUS expr:e2      {: RESULT = e1 - e2; :}
       | expr:e1 MULT expr:e2       {: RESULT = e1 * e2; :}
       | expr:e1 DIV expr:e2        {: RESULT = e1 / e2; :}
       | MINUS:e expr               {: RESULT = (-e); :}
       | LPAREN expr:e RPAREN       {: RESULT = e; :}
       | NUMBER:n                   {: RESULT = n; :}
```

Actions: Building an AST

```
terminal int NUMBER;
terminal PLUS, MINUS, MULT, DIV;
terminal LPAREN, RPAREN;

non terminal Expr expr;
```

```
expr ::= expr:e1 PLUS expr:e2      {: RESULT = new AddExpr(e1, e2); :}
       | expr:e1 MINUS expr:e2     {: RESULT = new SubtractExpr(e1, e2); :}
       | expr:e1 MULT expr:e2      {: RESULT = new MultExpr(e1, e2); :}
       | expr:e1 DIV expr:e2       {: RESULT = new DivExpr(e1, e2); :}
       | MINUS:e expr              {: RESULT = new Minus(e); :}
       | LPAREN expr:e RPAREN      {: RESULT = e; :}
       | NUMBER:n                  {: RESULT = new IntLiteral(n); :}
```

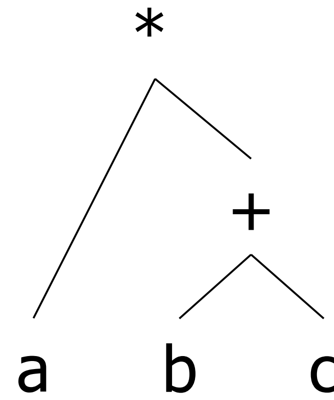
Lo and Behold

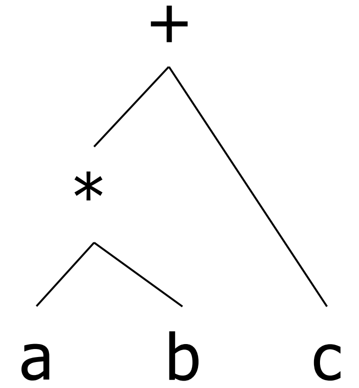
- See how it so majestically compiles!
- But…

Ambiguity

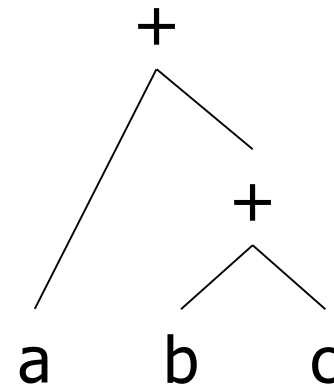
- A grammar is *ambiguous* if there exists a string that has two different derivations
- Solutions:
  - Changing the grammar
  - Specifying precedence and associativity

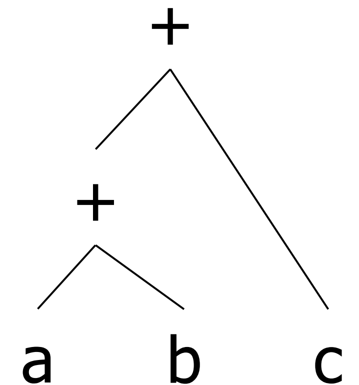
Next week!





a * b + c





a + b + c

Grammar rewriting

Ambiguous grammar:

$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

Unambiguous grammar:

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

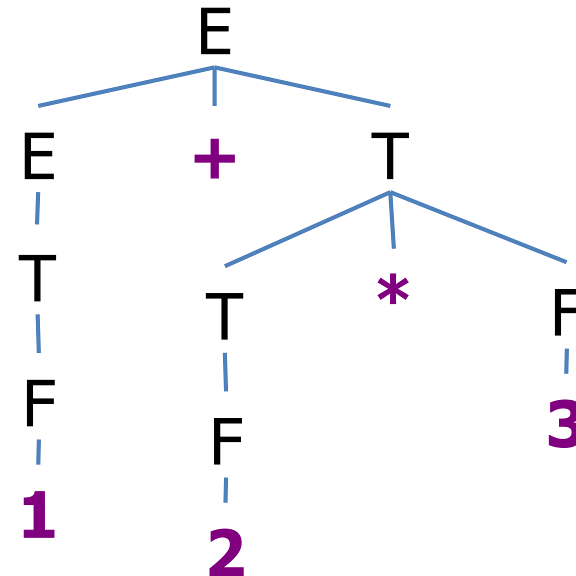
$T \rightarrow F$

$F \rightarrow id$

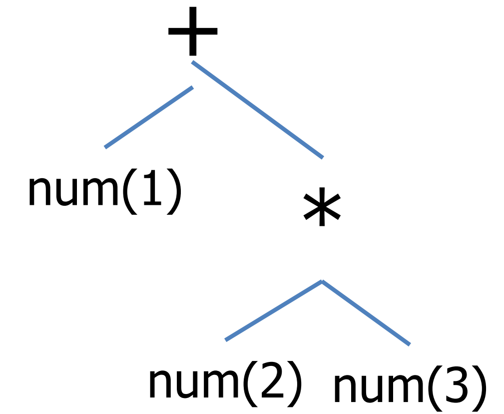
$F \rightarrow num$

$F \rightarrow (E)$

Parse tree:



AST:



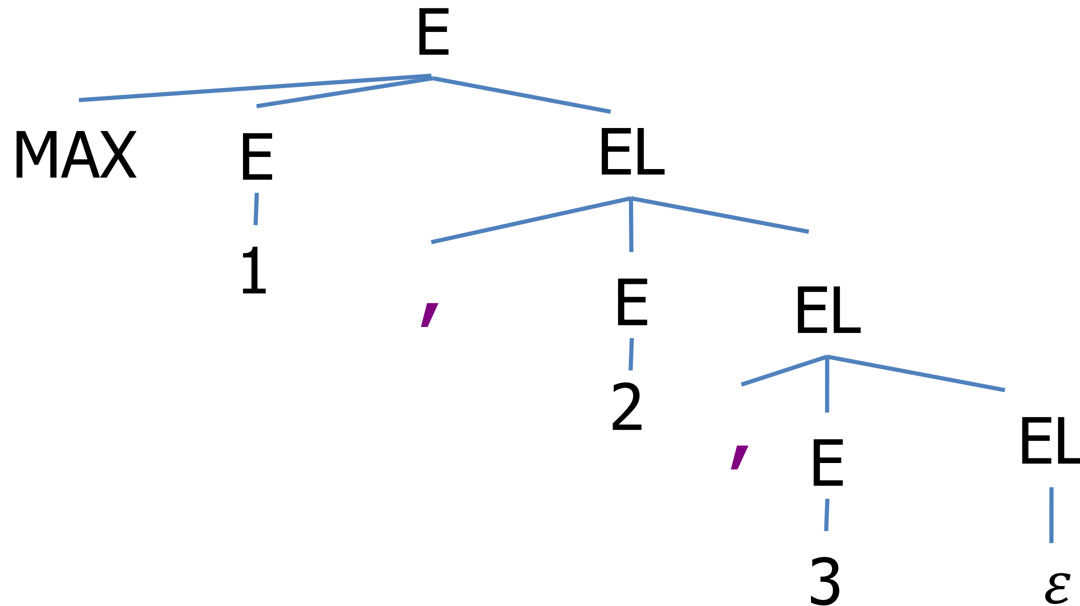
A grammar accepts a language.
A language can be accepted by many grammars.

Parsing Lists

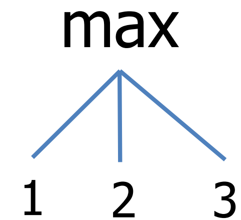
- Note where you're adding (add / addFirst)
- Two options: left or right recursion
  - Potentially generates different conflicts

Demo

Parse tree:



AST:



Parser Errors

- CUP output
- CUP code
- (How to “recover” from errors?)

Debugging CUP

- Getting internal representation
  - Command line options:
    - -dump_grammar
    - -dump_states
    - -dump_tables
    - -dump
  - Enabled in the build.xml ant task in the demos of this recitation

Summary

- Parsing
- Parser generation with CUP
- Overcoming ambiguity (more next week)
- **Time to check** that you can **build** with lexer & parser generation, in preparation to **ex. 4**
  (Template JFlex, CUP, build.xml updated)