Course 142A Compilers & Interpreters

Syntactic Analysis

Lecture Week 2

Prof. Dr. Luc Bläser

Last Lecture - Quiz

```
Expression    = Term { ( "+" | "-" ) Term }.
Term          = Number | "(" Expression ")".

Number        = Digit { Digit }.
Digit         = "0" | ... | "9".
```

Which are the tokens recognized by the lexer?

Regular Language Subset

```
Expression    = Term { ( "+" | "-" ) Term }.
Term          = Number | "(" Expression ")".

Number        = Digit { Digit }.
Digit         = "0" | ... | "9".
```

Regular subset =>
Detected by the lexer

Why is not the entire language regular?

Regular & Context-Free

- Regular language
  - Specifiable as EBNF without recursion
  - Finite Automaton
  - Case for the lexer

- Context-free language
  - Specifiable as arbitrary EBNF
  - Push-Down Automaton (stack)
  - Case for the parser

Our focus today

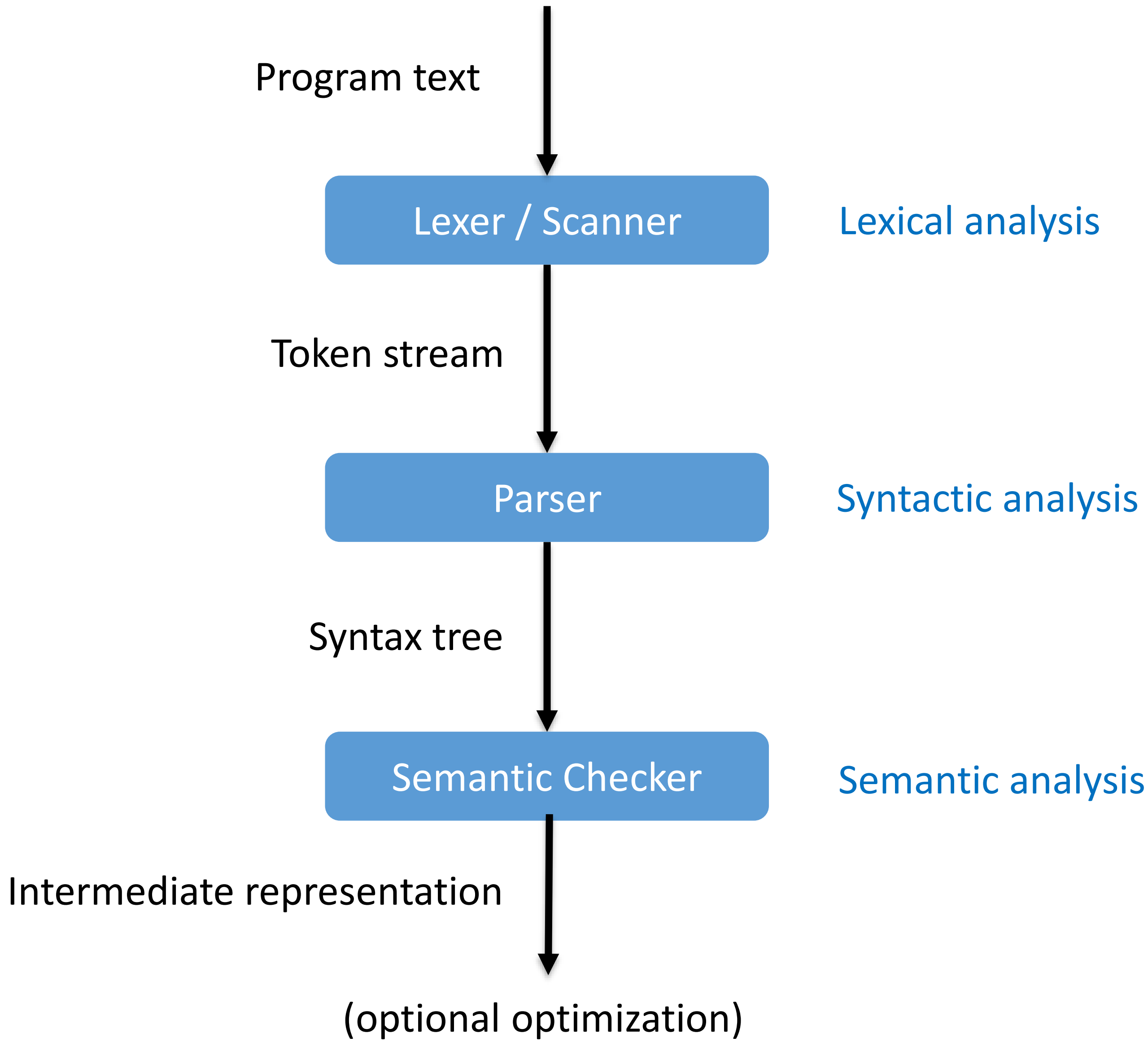
Today's Topics

- Top-down parser
- Syntax tree
- Recursive descent
- Tools and implementations

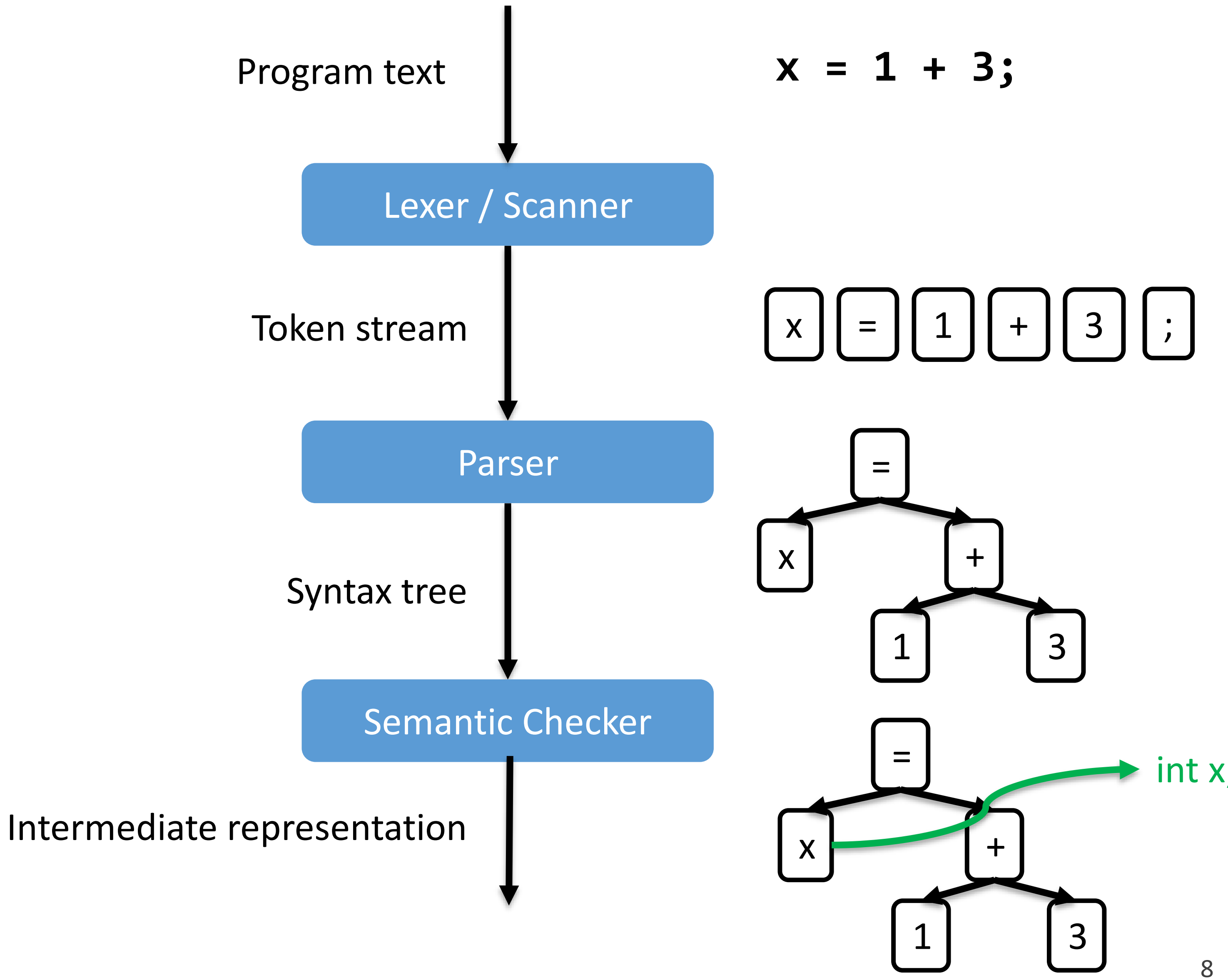
Learning Goals

- Know the functionality of a recursive descent top-down parser
- Be able to implement such a parser

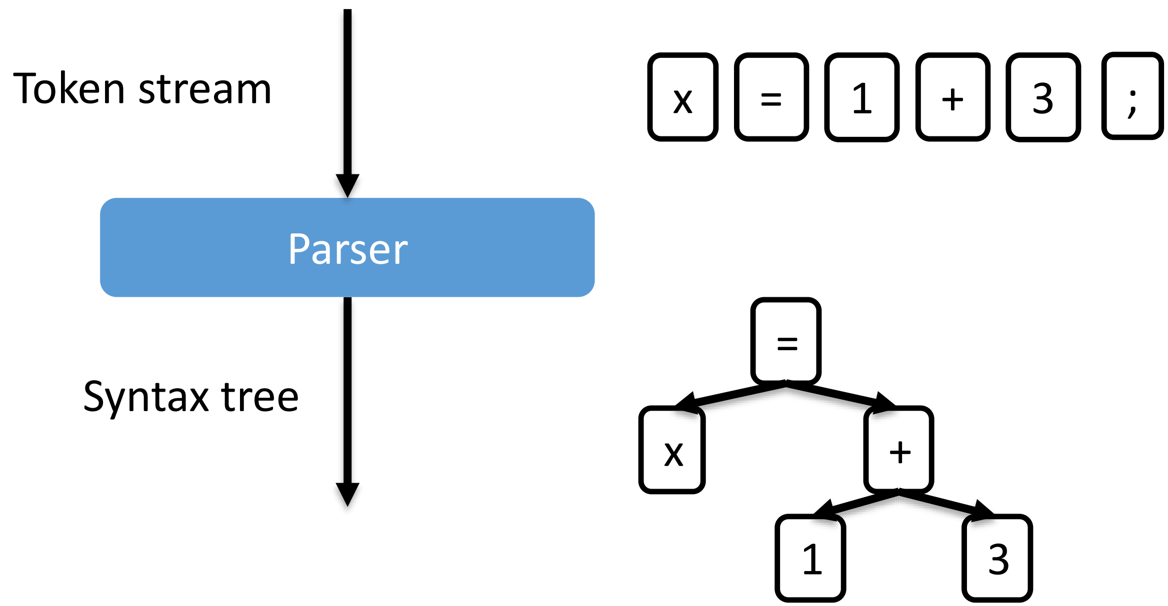
Compiler Frontend



Run-Through Example



Our Focus: Parser



Parser

Cares about the syntactic analysis

- Input: Token stream (stream of terminal symbols)
- Output: Syntax tree / parse Tree

Context Freedom

- Parser restricted to context-free languages
  - Context-free = expressible in EBNF
- However, many aspects are context-sensitive
  - E.g. Variables to be declared before use
  - E.g. Boolean values cannot be added
  - E.g. Arguments must match to parameters
  - ...
- Context sensitivity will be checked later
  - Semantic Checker

Task of a Parser

- Find the unambiguous derivation of syntax rules, to obtain a given input

Input: 1 + (2 - 3)

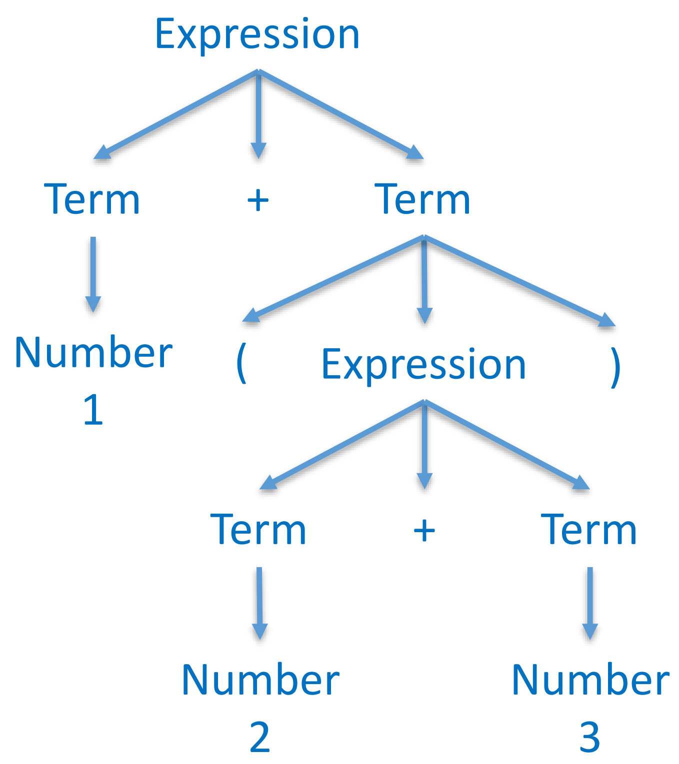
Derivation:
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

Purpose of a Parser

- Analyze the entire syntax definition
  - With and without recursive rules
- Determine whether input fulfils the syntax or not
  - Syntactically valid: `1 + (2 - 3)`
  - Syntactically invalid: `1 ++ (3(`
- Unambiguous derivation wanted
  - Otherwise, we have a problem with syntax definition
- Creates a syntax tree
  - For further compilation steps

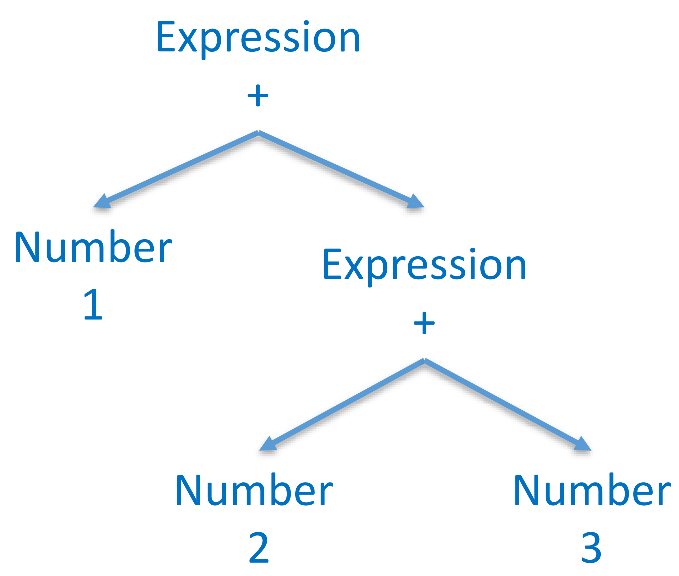
Parse Tree

- Also called Concrete Syntax Tree (CST)
- Derivation of syntax rules reflected as tree



Abstract Syntax Tree

- Skip irrelevant details, simplify structure, customize for subsequent processing
- Custom design of the compiler designer



Concrete vs. Abstract Syntax Tree

- Both are possible Intermediate Representations (IR)
- Generated parser can yield Concrete Syntax Tree
  - Manual conversion to Abstract Syntax Tree
- Hand-written parser can yield Abstract Syntax Tree
  - No intermediate step via parse tree

Parser Strategies

- Top-down
  - Begin with start symbol
  - Apply productions
  - Expand start symbol to input text
    - Expr -> Term + Term -> ... -> 1 + (2 - 3)
- Bottom-up
  - Begin with input text
  - Apply productions
  - Reduce input text to start symbol
    - Expr <- Term + Term <- ... <- 1 + (2 - 3)

Top-Down Parsing

Input: 1 + (2 - 3)

Derivation:

Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

Top-Down

Left-most expansion

Today's focus

Bottom-Up Parsing

Input: 1 + (2 - 3)

Derivation:
Expression
Term "+" Term
Term "+" "(" Expression ")"
Term "+" "(" Term "-" Term ")"
Term "+" "(" Term "-" Number ")"
Term "+" "(" Number "-" Number ")"
Number "+" "(" Number "-" Number ")"

Bottom-Up

Right-most reduction

Next week

First Parser Attempt

```
Term = Number | "(" Expression ")".
```

```
if (isNumber()) {
  next();
} else if (is(Tag.OPEN_PARENTHESIS)) {
  next();
  ???? Read next expression ????
  if (is(Tag.CLOSE_PARENTHESIS)) {
    next();
  } else {
    error();
  }
} else {
  error();
}
...
```

How can we read the expression in the term?

Recursion

- Expression has terms, terms may contain expression
  - Recursive definition
- A flat routine is insufficient for parsing
  - In contrast to the lexer
- We can now use recursive programming
  - Recursive descent parsing

Recursive Descent

- Write a method per non-terminal symbol
  - Implement the detection according to EBNF rule
- If a non-terminal symbol occurs in syntax
  - Call the corresponding method

Works for recursive and non-recursive productions

Method Per Non-Terminal Symbol

```
void parseExpression() {
  parseTerm();
  ...
}
```

Implements detection of
`Expression = ....`

```
void parseTerm() {
  ...
  parseExpression();
  ...
}
```

Implements detection of
`Term = ....`

Call each other recursively

Parser Skeleton

```
public class Parser {
  private final Lexer lexer;
  private Token current;

  public Parser(Lexer lexer) {
    this.lexer = lexer;
    next();
  }

  private void next() {
    current = lexer.next();
  }

  ...
}
```

One token lookahead

Helper Methods

```
private boolean is(Tag tag) {
  return current instanceof StaticToken &&
     ((StaticToken)current).getTag() == tag;
}

private boolean isInteger() {
  return current instanceof IntegerToken;
}

private boolean isEnd() {
  return is(Tag.END);
}
```

Parser Entry

```
Program = Expression.
```

Simplification for the moment:
Check only syntactic correctness

```
public void parseProgram() {
  parseExpression();
  if (!isEnd()) {
    Error();
  }
}
```

Later, parser should also generate syntax tree

Expression

```
Expression = Term { ( "+" | "-" ) Term }.
```

```
void parseExpression() {
  parseTerm();
  while (is(Tag.PLUS) || is(Tag.MINUS)) {
    next();
    parseTerm();
  }
}
```

Term

```
Term = Number | "(" Expression ")".
```

```
void parseTerm() {
  if (isNumber()) {
    next();
  } else if (is(Tag.OPEN_PARENTHESIS)) {
    next();
    parseExpression();
    if (is(Tag.CLOSE_PARENTHESIS)) {
      next();
    } else {
      error();
    }
  } else {
    error();
  }
}
```

Discussion

- Recursive descent is a top-down parser
  - Implicit stack through method calls
  - Corresponds to push-down automaton
- Predictive direct parsing
  - Always clear which production to take
  - Simple and preferred approach
- Other approach: Backtracking
  - If unclear which production to take
  - Select potential production, undo on syntax error and start over with next production

Other Example

```
Statement = Assignment | IfStatement.
Assignment = Identifer "=" Expression.
IfStatement = "if" "(" Expression ")" Statement.
```

```
void parseStatement() {
  if (???) {
    parseAssignment();
  } else if (???) {
    parseIfStatement();
  } else {
    Error();
  }
}
```

Which conditions to use for the branches?

One Symbol Lookahead

- Determine all possible first terminal symbols that can be derived by a production (FIRST-set)

```
FIRST(Assignment) = { Identifier }

FIRST(IfStatement) = { "if" }
```

- Use FIRST to decide branches on predictive parsing

Branch Decisions

```
void parseStatement() {
  if (isIdentifier()) {
    parseAssignment();
  } else if (is(Tag.IF)) {
    parseIfStatement();
  } else {
    error();
  }
}
```

FIRST(Assignment)

FIRST(IfStatement)

Multiple Elements in FIRST

```
LoopStatement  = WhileStatement | DoStatement.
WhileStatement = "while" "(" Expression ")" Statement.
DoStatement    = "do" Statement "while" "(" Expression ")".
```

```
FIRST(WhileStatement) = { "while" }
FIRST(DoStatement) = { "do" }
FIRST(LoopStatement) = { "while" , "do" }
```

```
if (is(Tag.WHILE) || is(Tag.DO)) {
    parseLoopStatement();
}
```

Other Example

```
Statement = Assignment | Invocation.
Assignment = Identifier "=" Expression.
Invocation = Identifier "(" ")".
```

```
FIRST(Assignment) = { Identifier }
FIRST(Invocation) = { Identifier }
```

Ambiguous decision:
Lookahead of one symbol is not sufficient

Lookahead with k Symbols

- Need more than one symbol lookahead, k > 1
- Here, k = 2 is sufficient

```
FIRST(Assignment) = { Identifier "=" }
FIRST(Invocation) = { Identifier "(" }
```

How can we implement this?

Technical Syntax Rewriting

```
Statement = Assignment | Invocation.
Assignment = Identifier "=" Expression.
Invocation = Identifier "(" ")".
```

```
Statement = Identifier (AssignmentRest | InvocationRest).
AssignmentRest = "=" Expression.
InvocationRest = "(" ")".
```

Lookahead 1 is again sufficient

Parse with Longer Lookahead

```
void parseStatement() {
  var identifier = readIdentifier();
  next();
  if (is(Tag.ASSIGN)) {
    parseAssignmentRest(identifier);
  } else if (is(Tag.OPEN_PARENTHESIS)) {
    parseInvocationRest(identifier);
  } else {
    error();
  }
}
```

Left Recursion

```
Sequence = Sequence [ Statement ].
```

```
void parseSequence() {
  parseSequence();
  if (!is(Tag.CLOSE_BRACE)) {
    parseStatement();
  }
}
```

What is the problem here?

Problem Case

Input: x = 1

Derivations (recursive descent)

Sequence
Sequence [Statement]
Sequence [Statement] [Statement]
Sequence [Statement] [Statement] [Statement]
Sequence [Statement] [Statement] [Statement] [Statement]
...

Infinite recursion

Avoid Left Recursion

- Use EBNF-Repetition instead

```
Sequence = { Statement }.
```

```
void parseSequence() {
  while (!is(Tag.CLOSE_BRACE)) {
    parseStatement();
  }
}
```

Parse Class According to D. E. Knuth

L = top-down parser
R = bottom-up parser

LL(k)

Read input from left to right

k symbols lookahead, e.g. LL(1)-parser

Parser Generator

- Tool generating parser from syntax definition

	Class	Lexer	Syntax	Output	Internal
Bison	LR(1)	separate	BNF	C/C++	C/C++
Yacc	LR(1)	separate	BNF	C/C++	C/C++
AntLR	LL(k)	integrated	EBNF	Java, C#, C etc.	Java
JavaCC	LL(k)	integrated	EBNF	Java	Java
CUP	LR(1)	separate	EBNF	Java	Java
Coco/R	LL(1)	integrated	EBNF	Java, C#, C++ etc.	Java, C# etc.

Example: AntLR4

```
grammar UCIJava;

// lexer rules
Number: Digit+;
Digit: [0-9];
Whitespaces: [ \t\r\n]+ -> skip;

// parser rules
expression: term ( ('+' | '-') term )*;
term: Number | '(' expression ')';
```

Parser rules begin with lower case

Generated parse tree

Generator: Discussion

- Less busy work
  - Generate parser and syntax tree
- Less error-prone
  - No boilerplate code, mere grammar
- Less flexibility
  - Conflict problems, sometimes predefined syntax tree, no custom error handling

Review: Learning Goals

- ✓ Know the functionality of a recursive descent top-down parser
- ✓ Be able to implement such a parser

Further Reading

- Dragon Book, Chapter 4 (Syntax Analysis)
  - Sections 4.1 – 4.4 (Top Down Parsing)
- Optional, if interested:
  - AntLR 4.7, http://www.antlr.org/

Appendix

Self-Study

UCI-Java Parser Rules in AntLR (1)

```
grammar UCIJava;

// lexer rules => see last week

// parser rules

program: classNode* EOF;
classNode: CLASS Identifier (EXTENDS Identifier)?
       LBRACE (variable | method)* RBRACE;

variable: type Identifier SEMI;
method: methodHead methodBody;
methodHead: type Identifier LPAREN parameterList RPAREN;
parameterList: (parameter (COMMA parameter)*)?;
parameter: type Identifier;
methodBody: statementBlock;
type: Identifier (LBRACKET RBRACKET)*;
```

UCI-Java Parser Rules in AntLR (2)

```
statementBlock: LBRACE statement* RBRACE;
statement: SEMI | variable | assignment | ifStatement |
           whileStatement | callStatement | returnStatement;
assignment: designator ASSIGN expression SEMI;
ifStatement: IF LPAREN expression RPAREN statementBlock
             (ELSE statementBlock)?;
whileStatement: WHILE LPAREN expression RPAREN
statementBlock;
callStatement: methodCall SEMI;
returnStatement: RETURN expression? SEMI;
```

UCI-Java Parser Rules in AntLR (3)

```
expression: logicTerm (OR logicTerm)*;
logicTerm: logicFactor (AND logicFactor)*;
logicFactor: simpleExpression (compareOperator simpleExpression)*;
compareOperator: EQUAL | UNEQUAL | LESS | LEQ | GREATER | GEQ |
                 INSTANCEOF;
simpleExpression: term ((PLUS | MINUS) term)*;
term: factor ((MULT | DIV | MOD) factor)*;
factor: operand | unaryExpression |
        LPAREN expression RPAREN | typeCast;
unaryExpression: (NOT | PLUS | MINUS) factor;
operand: literal | designator | objectCreation |
         arrayCreation | methodCall;
```

UCI-Java Parser Rules in AntLR (4)

```
typeCast: LPAREN Identifier RPAREN designator;
literal: Integer | String;
designator: Identifier | designator DOT Identifier |
            designator LBRACKET expression RBRACKET;
objectCreation: NEW Identifier LPAREN RPAREN;
arrayCreation: NEW Identifier LBRACKET expression RBRACKET;
methodCall: designator LPAREN argumentList RPAREN;
argumentList: (expression (COMMA expression)*)?;
```

AntLR Parser Integration

- Generate (grammar file «UCIJava.g4»)

```
java -jar antlr-4.7.2-complete.jar -Dlanguage=Java UCIJava.g4
```

- Java-integration

```java
var stream = CharStreams.fromString(input);
var lexer = new UCIJavaLexer(stream);
var tokens = new CommonTokenStream(lexer);
var parser = new UCIJavaParser(tokens);
parser.setBuildParseTree(true);
var tree = parser.program();
```

Course 142A Compilers & Interpreters

Syntactic Analysis

Lecture Week 2, Wednesday

Prof. Dr. Luc Bläser

Last Lecture - Quiz

```
Expression = Term { ( "+" | "-" ) Term }.
```

```
void parseExpression() {
  parseTerm();
  while (is(Tag.PLUS) || is(Tag.MINUS)) {
    next();
    parseTerm();
  }
}
```

What is missing in this parser?

Today's Topics

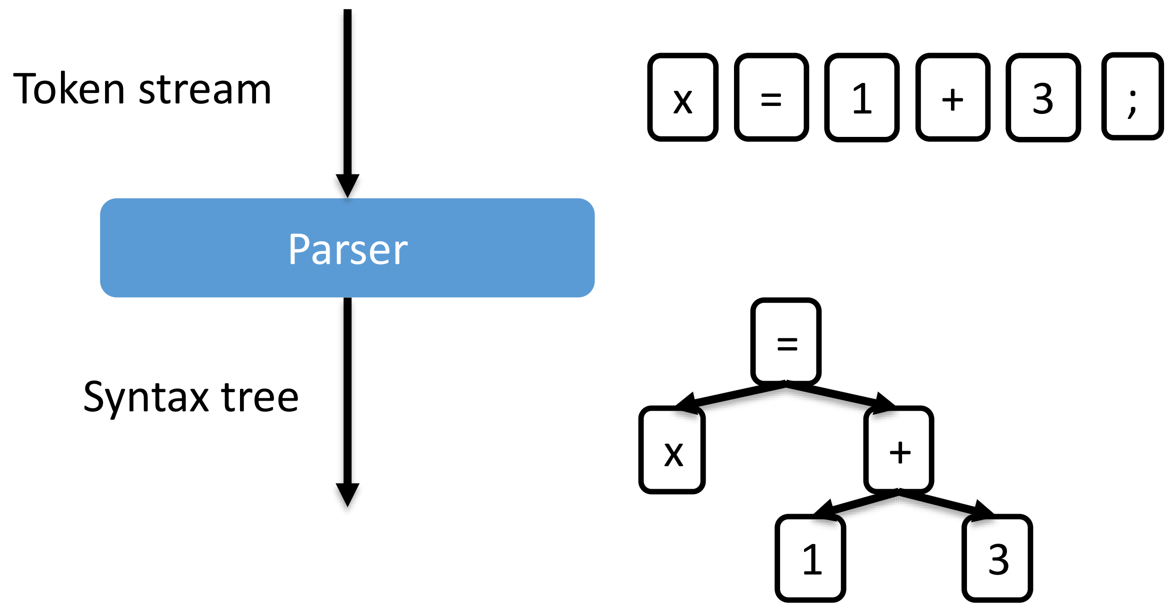
- Syntax Tree Creation
- Parser Error Handling
- Syntax-Directed Translation

Learning Goals

- Know how to create a syntax tree in your parser
- Be able to handle parser errors
- Understand the principles of syntax-directed translation

Parser Produces Syntax Tree

- Concrete or abstract – depending on design



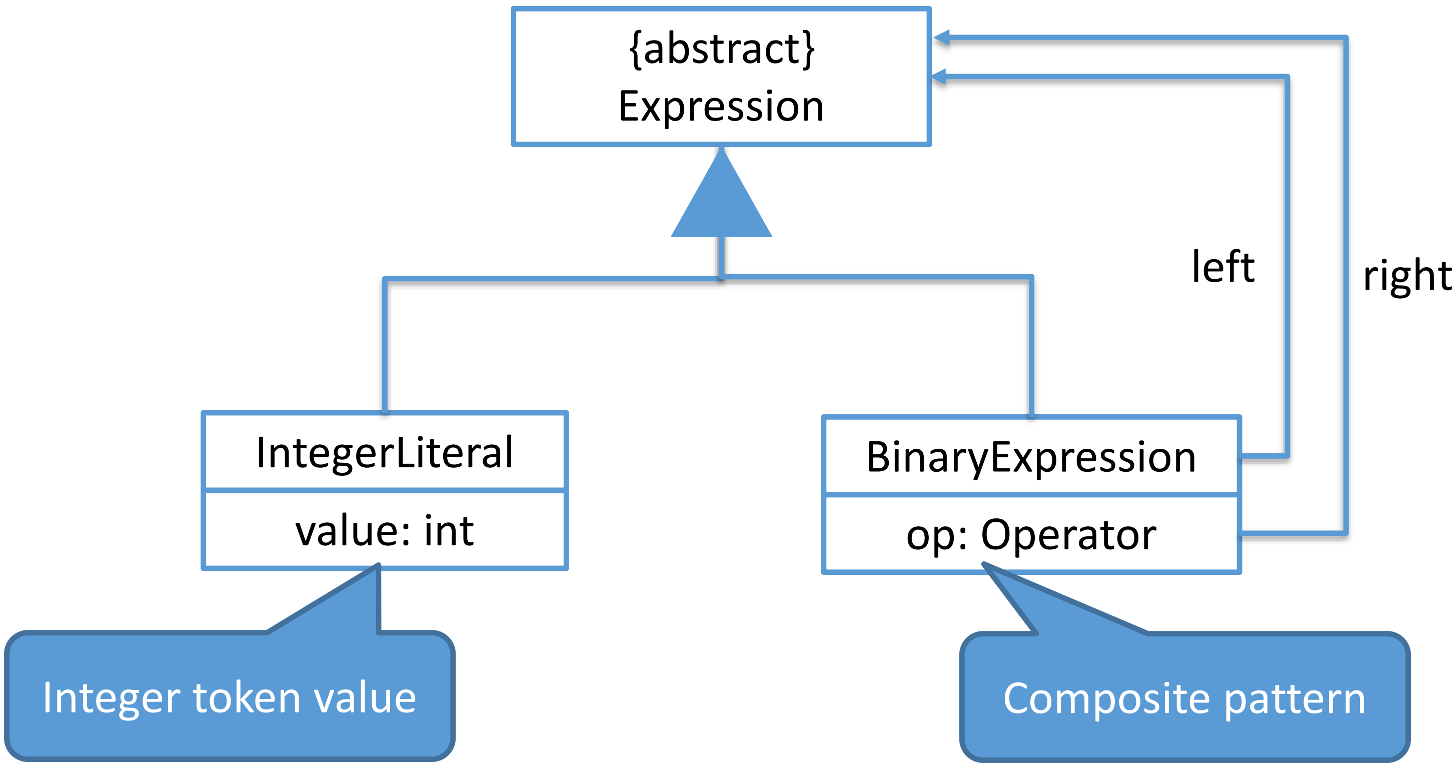
How do we have to extend our parser?

Extension

- Design syntax tree
  - OO class design
- Build during parsing
  - In parse-methods

Syntax Tree Design

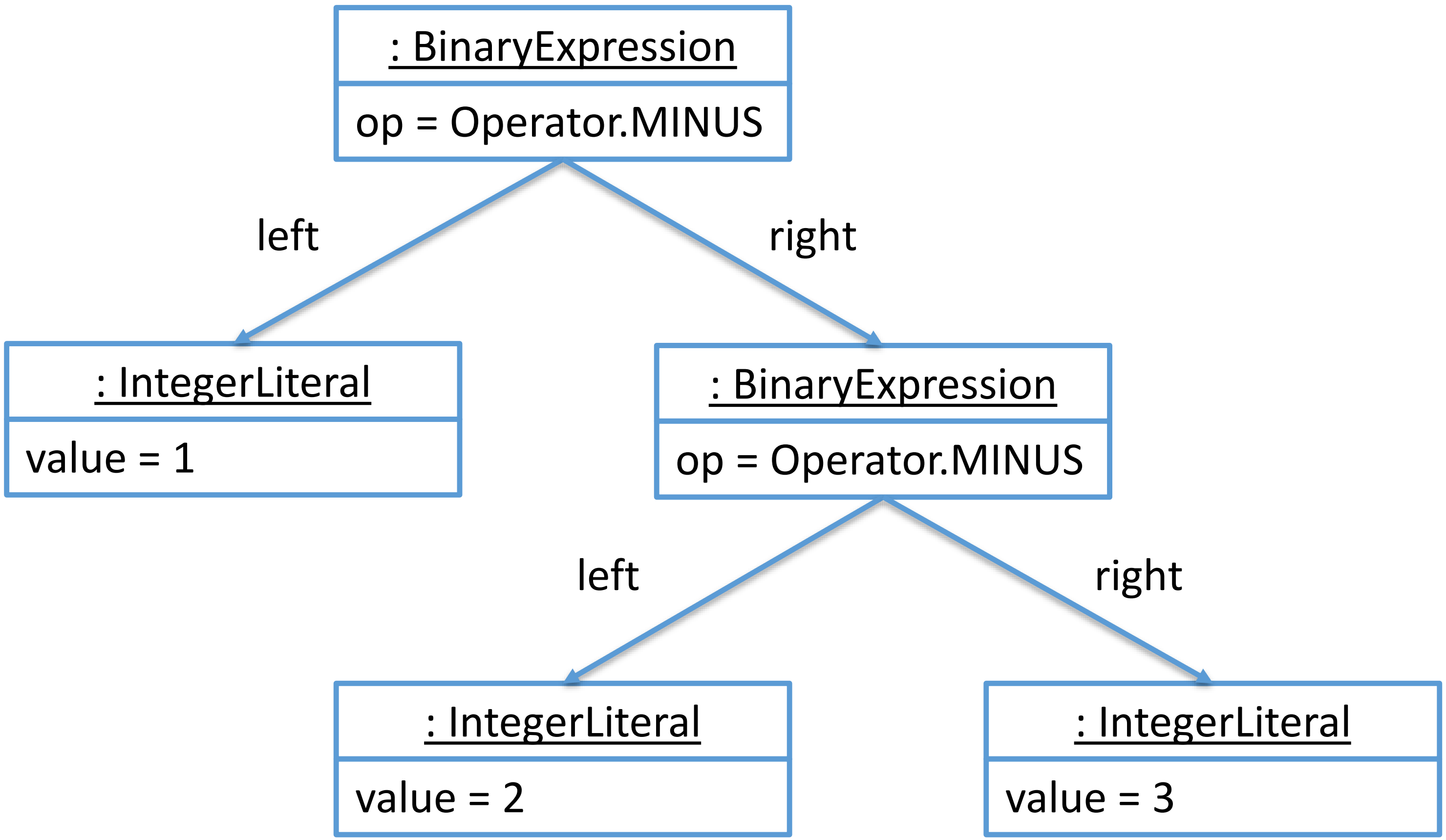
- Class diagram for abstract syntax tree



To what extent is this abstract?

Syntax Tree Design

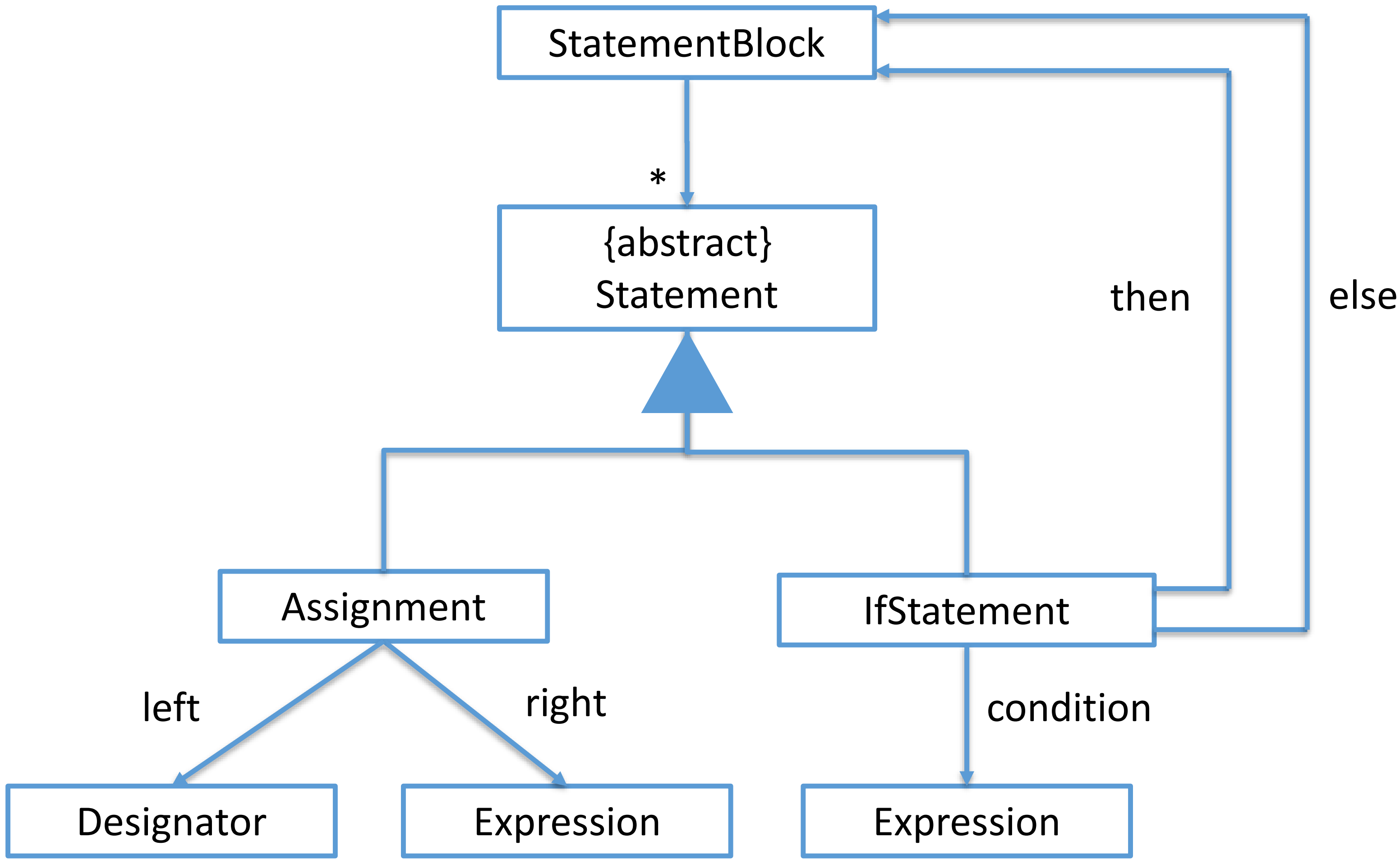
- Object diagram for 1 - (2 - 3)



Design Questions

- Abstract vs. concrete
  - AST (abstract syntax tree) by custom design
  - CST (parse tree) by generated parser
- Other expression sub-classes
  - UnaryExpression (e.g. -3 or +4)
  - Other types of literal (e.g. boolean, string)
  - Designator (e.g. x or y[0].z)
- Record source code positions
  - For error messages and debugging
  - Determine from lexer token stream

Statement Syntax Tree



Construct Syntax Tree

```
Expression parseTerm() {
  if (isInteger()) {
    var value = readInteger();
    next();
    return new IntegerLiteral(value);
  } else if (is(Tag.LEFT_PARENTHESIS)) {
    next();
    var expression = parseExpression();
    if (is(Tag.RIGHT_PARENTHESIS)) {
      next();
    } else {
      error();
    }
    return expression;
  } else { error(); }
}
```

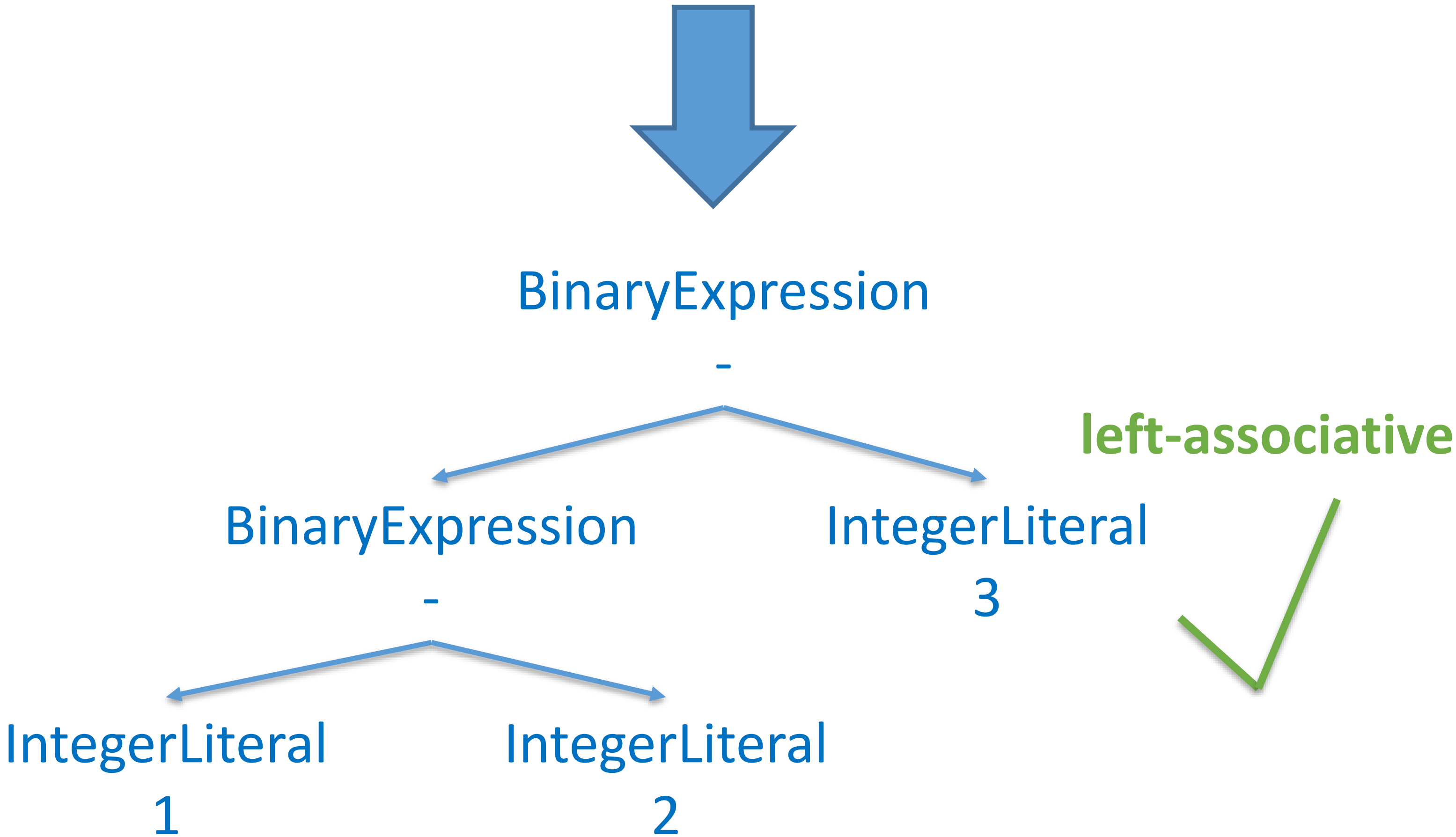
Other Construction Example

```
Expression parseExpression() {
  var left = parseTerm();
  while (is(Tag.PLUS) || Is(Tag.MINUS)) {
    var op =
      is(Tag.PLUS) ? Operator.PLUS: Operator.MINUS;
    next();
    var right = parseTerm();
    left = new BinaryExpression(op, left, right);
  }
  return left;
}
```

How about associativity?

Generated Tree

1 - 2 - 3



Syntax Error Handling

- Continue on error
  - More syntax errors are likely
- Requires hypothesis
  - Punctuation errors are frequent (e.g. missing semicolon)
  - Forgotten operator is seldom (e.g. missing plus)
- Frequent error cases
  - Missing symbol (e.g. semicolon, closing brace)
    - Ignore
  - Wrong symbol (e.g. wrong bracket, comma instead of ;)
    - Replace

Missing Symbol

- E.g. closing parenthesis in expression

```
if (is(Tag.RIGHT_PARENTHESIS)) {
   next();
} else {
   error();
}
```

```
if (is(Tag.RIGHT_PARENTHESIS)) {
  next();
} else {
  error(") missing");
}
```

Mark error with position & continue

Wrong Symbol

- Square bracket instead of parenthesis

```
if (is(Tag.RIGHT_PARENTHESIS)) {
  next();
} else {
  error(") missing");
  if (is(Tag.RIGHT_BRACKET)) { next(); }
}
```

Error Synchronization

- Search for new entry point on error
  - Skip symbols until it matches again
- Non-skippable symbols: class, if etc.
  - Synchronization points on parser errors

```
Expression parseTerm() {
  if (!isNumber() || !is(Tag.LEFT_PARENTHESIS)) {
     error("invalid term");
     while (!isNumber() && !is(Tag.LeftParenthesis)
            && canSkip()) {
       next();
     }
  }
  ...
```

canSkip = skippable symbol

Undetected Errors

- Incompatible types
- #arguments ≠ #parameters
- Undeclared variables
- Undeclared methods
- Inapplicable operators
- ...

No syntax error, but semantic error

Task of the Semantic Checker (next week)

Syntax-Directed Translation

- Annotations to the syntax rules
  - Attributes to symbols
  - Semantic rules to the productions
  - Semantic actions inside production RHS
- Purpose: Additional effects during parsing
  - Type checks
  - Syntax tree construction
  - Code generation
  - Direct evaluation
- Often used in parser generators (e.g. yacc, bison)

First Example

Syntax rule (production)

Semantic action

```
Term       = Number                        (. print(Number) .)
           | "(" Expression ")".

Expression = Term "+" Term                 (. print("+")    .)
           | Term "-" Term                 (. print("-")    .)
           .
```

On the detection of syntax construct, the action is automatically applied in parser.

Semantic Actions

```
Term       = Number                     (. print(Number) .)
           | "(" Expression ")".

Expression = Term "+" Term              (. print("+")     .)
           | Term "-" Term              (. print("-")     .)
```

(1 - 2) + (3 - 4)

What is the parser output?

Integration in Parse-Methods

```
void parseExpression() {
  parseTerm();
  if (is(Tag.PLUS)) {
    next();
    parseTerm();
    print("+");
  } else if (is(Tag.MINUS)) {
    next();
    parseTerm();
    print("-");
  } else {
    Error();
  }
}
```

action inserted

action inserted

Postfix output:
1 2 - 3 4 - +

More General Approach

Attribute

Semantic rule

```
Term<x>       = Number<y>                    (. x = y .)
              | "(" Expression<y> ")"        (. x = y .)
              .

Expression<x> = Term<y> "+" Term<z>          (. x = y + z .)
              | Term<y> "-" Term<z>          (. x = y - z .)
              .
```

Associate attributes with symbols
Use these attributes in semantic rules

Synthesized Attributes

- Parent attribute defined by children attributes

```
Term<x>        = Number<y>                (. x = y .)
               | "(" Expression<y> ")"    (. x = y .)
               .

Expression<x> = Term<y> "+" Term<z>       (. x = y + z .)
               | Term<y> "-" Term<z>      (. x = y - z .)
               .
```

Bottom up processing of rules

Integration in Parse-Methods

```
int parseExpression() {
  int y = parseTerm();
  if (is(Tag.PLUS)) {
    next();
    int z = parseTerm();
    return y + z;
  } else if (is(Tag.MINUS)) {
    next();
    int z = parseTerm();
    return y - z;
  } else {
    error();
  }
}
```

Parser directly evaluates the expression

Other Example

```
Term<x>       = Number<y>
                (. x = new IntegerLiteral(y) .)
                | "(" Expression<y> ")
                (. x = y .)
                .

Expression<x> = Term<y> "+" Term<z>
                (. x = new BinaryExpression(Op.PLUS, y, z); .)
                | Term<y> "-" Term<z>
                (. x = new BinaryExpression(Op.MINUS, y, z); .)
                .
```

Syntax tree construction

Inherited Attributes

- Child attribute defined by parent, itself or sibling attributes
- Usually only left siblings

```
VariableList<offset> =
   Variable<location> (. location = offset .)
   [ "," VariableList<next> (. next = location + 4 .)
```

Top-down, left-to-right processing

S versus L

- S-attributed grammar
  - Allows only synthesized attributes

```
Expression<x> = Term<y> "+" Term<z>
              (. x = new BinaryExpression(Op.PLUS, y, z); .)
```

- L-attributed grammar
  - Allows both synthesized and inherited attributes

```
StatementBlock<x> =
       "{"
            { Statement<y> (. x.addStatement(y); .) }
        "}".
```

Examples

- Bison/Yacc: S-Attributed

```
expression : term[x] "+" term[y]
             { $$ = new BinaryExpression(PLUS, $x, $y); }
```

- CoCo/R: L-Attributed

```
StatementBlock<in List x, ref int depth> =
       (. depth++; .) "{"
            { Statement<out Node y> (. x.Add(y); .) }
        "}" (. depth--; .).
```

Discussion

- Vision: Put everything into grammar
  - Generate entire compiler
  - "Compiler-Compiler"
- Disadvantages
  - Side effects through parsing
  - Spread code snippets in rules
  - Syntax and semantics mixed
  - Usually only applied for tree generation

Review: Learning Goals

- ✓ Know how to create a syntax tree in your parser
- ✓ Be able to handle parser errors
- ✓ Understand the principles of syntax-directed translation

Next week: Bottom Up Parser and more…

Further Reading

- Dragon Book, selected sections:
  - 4.1.3-4.1.4 (Error Handling)
  - 2.5.1, 2.8.2, 4.2.4 (Syntax Trees)
  - 5.1-5.3.1 (Syntax-Directed Translation)
- Optional, if interested
  - Chapter 5 (Syntax-Directed Translation)