Course 142A Compilers & Interpreters
Introduction

Lecture Week 1

Prof. Dr. Luc Bläser

Welcome to «Compilers & Interpreters»

Intro to theory of programming language processors

- Compiler and runtime system design
- Focus on modern OO language processing

Language Specification
Type System
Interpreter
Virtual Machine
Syntax Formalism
Scanner / Lexer
Stack & Heap Management
Parser
OO Runtime Support
Abstract Syntax Tree
Garbage Collection
Semantic Checker
Intermediate Representation
Just-In-Time Compilation
Code Generation
Register Allocation
Dataflow Analysis
Code Optimzation
Loader

Why Compiler Design?

- Understand the programming language and its concepts much better
- Contribute to existing compiler and runtime system projects
- Design and implement a new programming language
- Apply concepts in related areas, e.g. converters, analyzers, IDE tools, algorithms

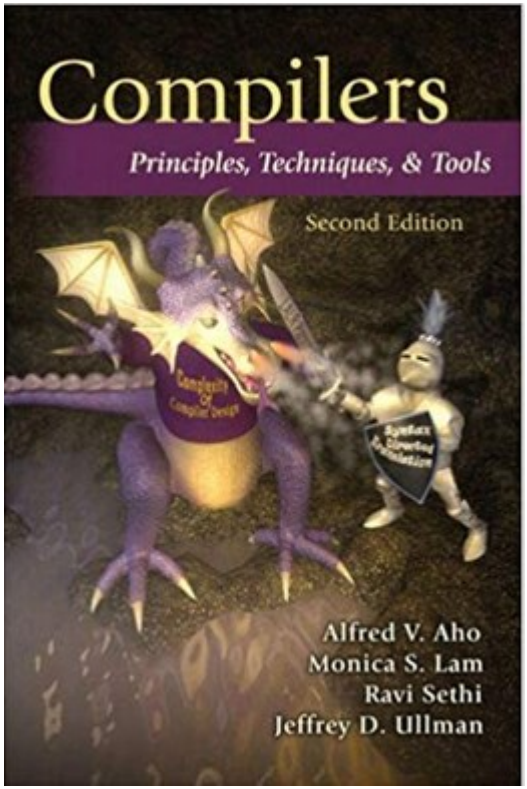
Overall Learning Goals

- Understand the fundamental architecture, concepts and techniques of compilers and runtime systems
- Implement own pieces of a compiler and runtime system for a modern OO language on a state-of-the-art platform
- Become familiar with syntax and semantic specifications of programming languages

Textbook (Selected Chapters Only)

- A. V. Aho, M. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley 2006. («Dragon Book»)

- Additional reading will be announced when needed



Compilers & Interpreters Introduction

Week 1, Monday

Today's Topics

- Compiler and runtime system overview
- Syntax formalism

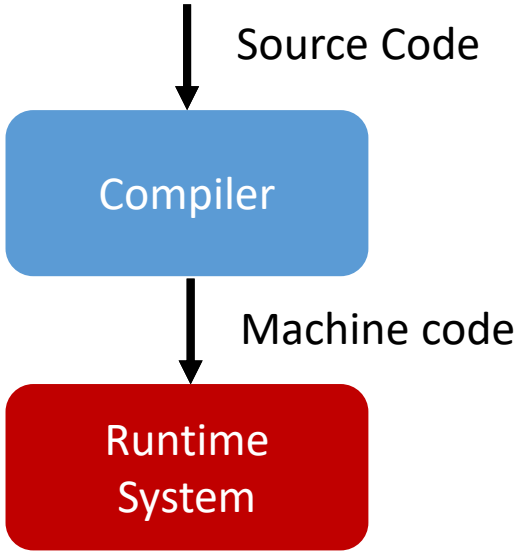
Learning Goals

- Know the fundamental structure and purpose of compilers and runtime systems
- Understand the EBNF syntax formalism

Fundamental Definitions

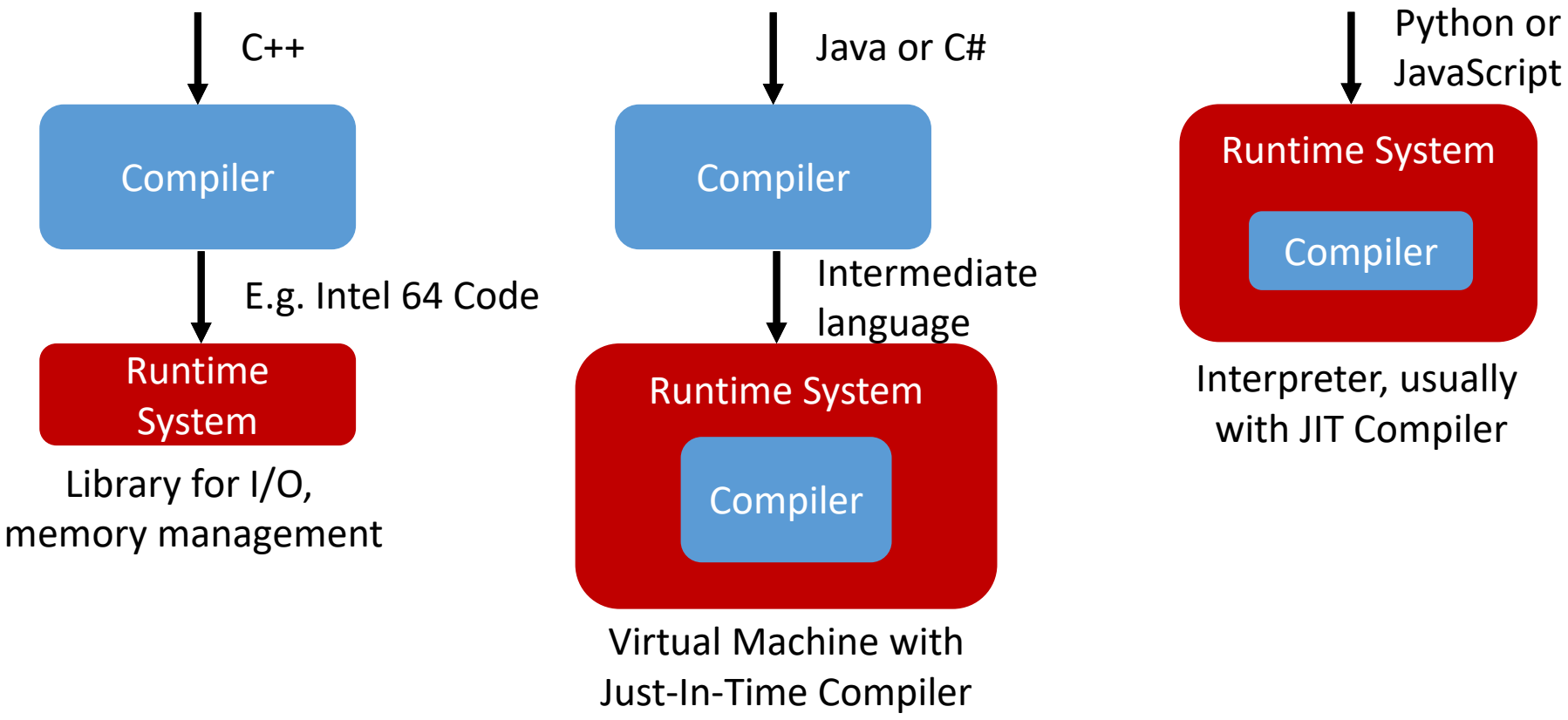
- Compiler
  - Transforms source code of a programming language to executable machine code
- Runtime System
  - Enables the program execution with software and hardware mechanisms

General Architecture

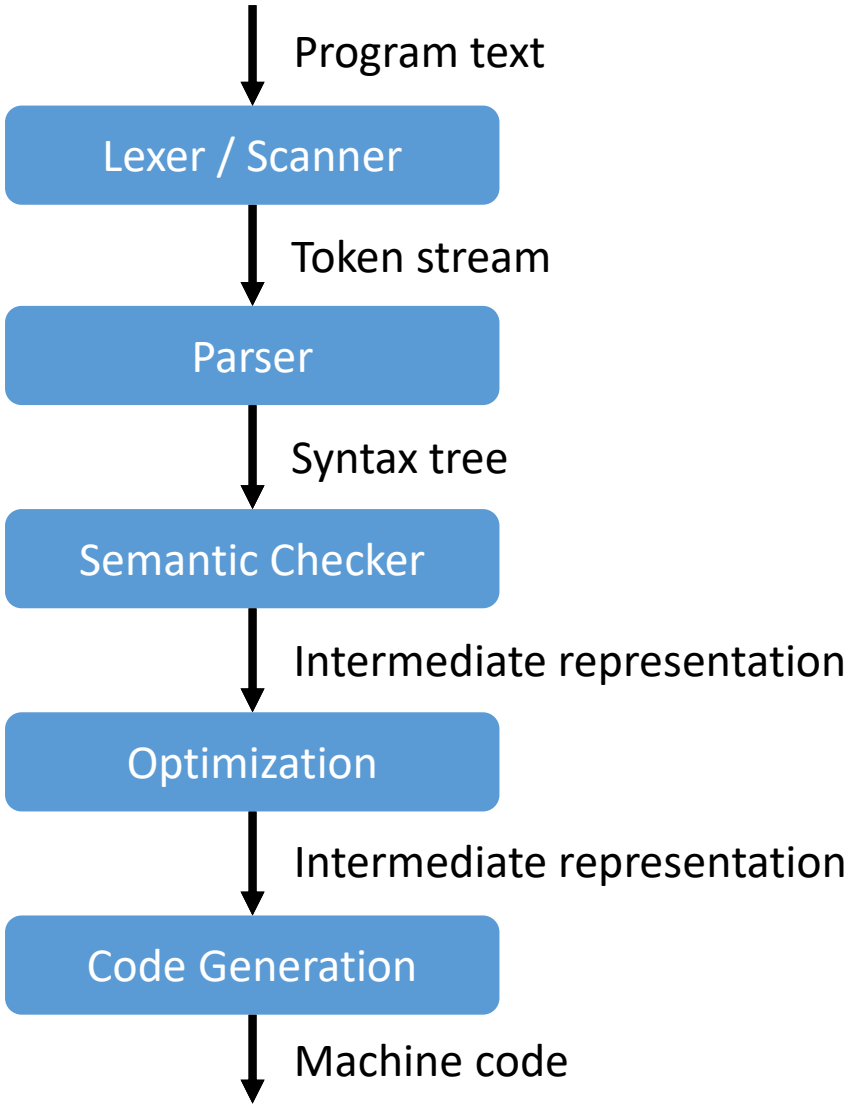


How does the specific architecture look like for C++, Java, C#, Python, and JavaScript?

Specific Architectures



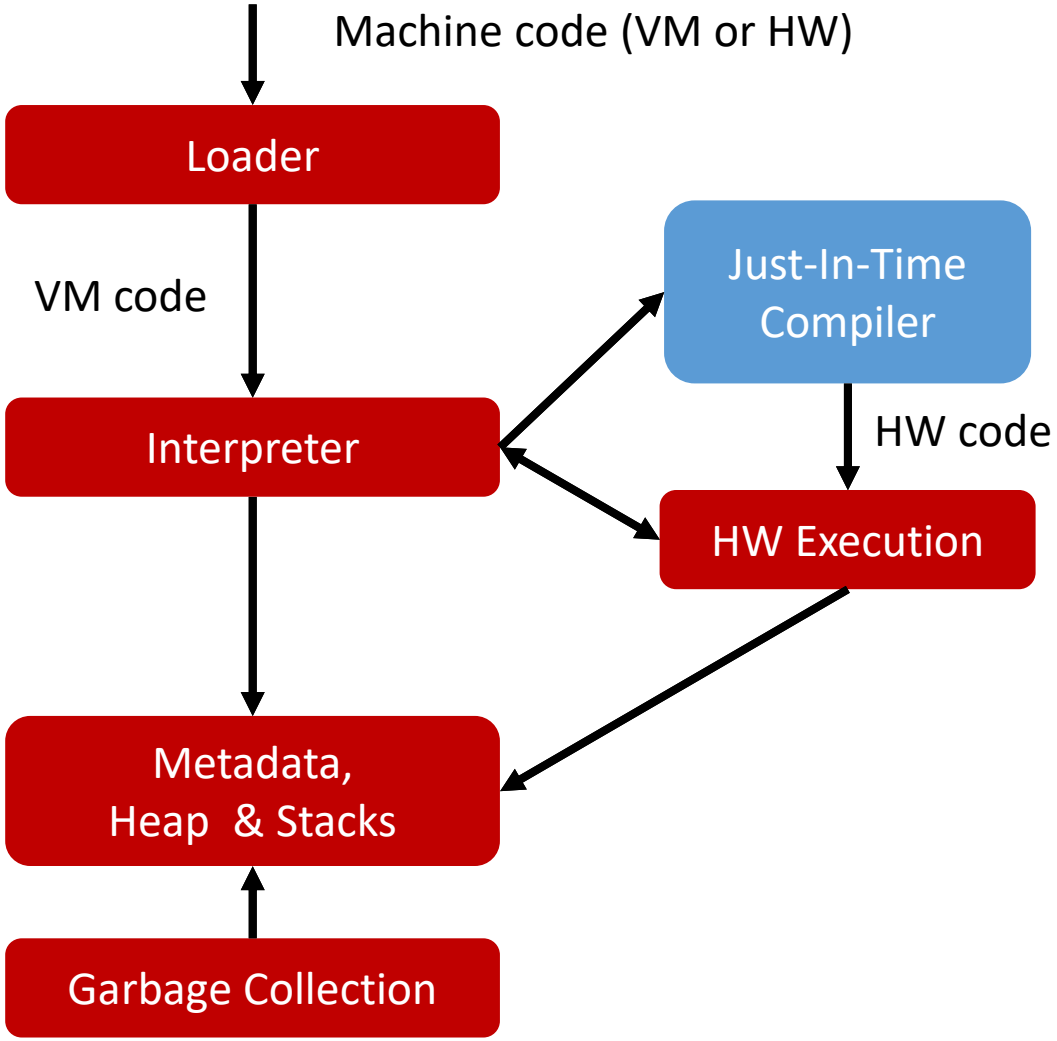
Compiler Structure



Compiler Components

- Lexer/Scanner (lexical analysis)
  - Splits program text in stream of tokens (terminal symbols)
- Parser (syntactic analysis)
  - Creates syntax tree according to program structure
- Semantic Checker (semantic analysis)
  - Resolves symbols, checks types and semantic rules
- Optimization (optional)
  - Transforms intermediate representation into more efficient one
- Code Generation
  - Produces executable machine code (VM or HW)
- Intermediate Representation
  - Describes program as data structure (various representations)

Runtime System Structure



Runtime System Components

- Loader
  - Loads machine code into memory, initiates execution
- Interpreter
  - Reads instructions and emulates them in software
- JIT (Just-In-Time) Compiler
  - Translates code fragments into hardware instruction code
- HW execution (native)
  - Runs instruction code directly on hardware processor
- Metadata, heap & stacks
  - Remember program structures, objects and method calls
- Garbage Collection
  - Dispose of unreachable objects

Definition of a Programming Language

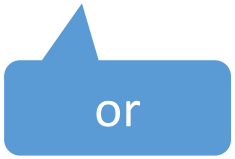
- Syntax defines the structure of programs
  - Effective formalisms for syntax used
- Semantics defines the meaning of programs
  - Usually described in prose

Syntax Formalism

- Description of syntax by rules/formulas

Language = Subject Verb.

Subject = "Anna" | "Paul".
Verb = "talks" | "listens".



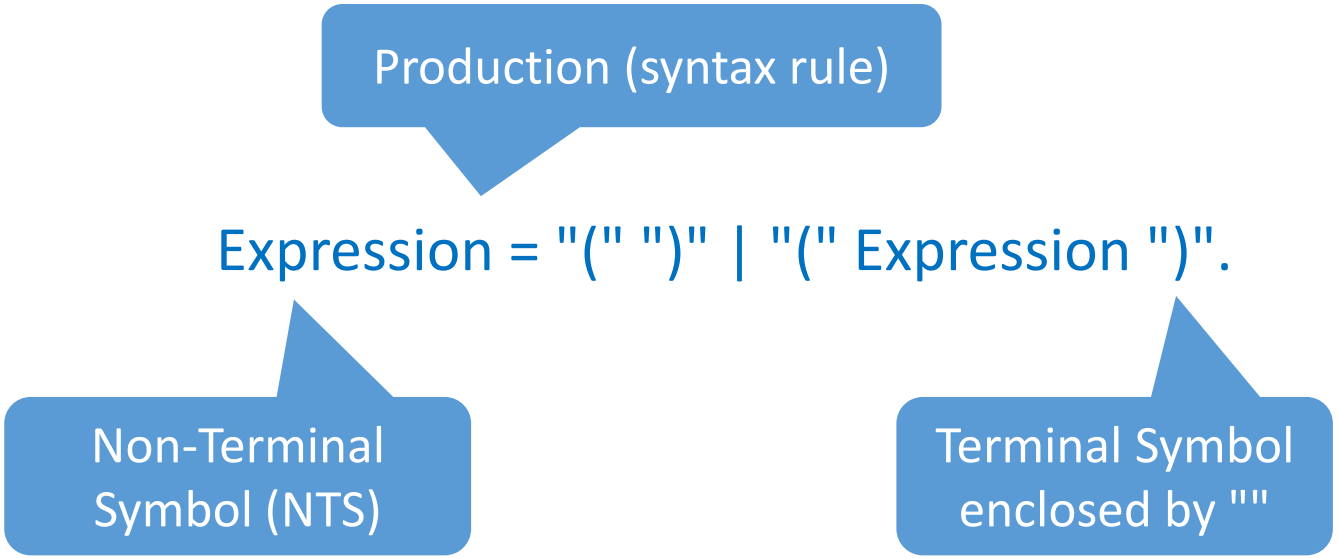
Recursion

Language = Expression.
Expression = "(" ")" | "(" Expression ")".

Which strings are contained in this language?

EBNF

- Extended Backus-Naur Form, N. Wirth, 1977



EBNF Constructs

	Example	Strings
Concatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" or "B"
Option	["A"]	empty or "A"
Repetition	{ "A" }	empty, "A", "AA", "AAA", etc.

Parentheses for grouping (highest precedence).
| has lowest precedence.

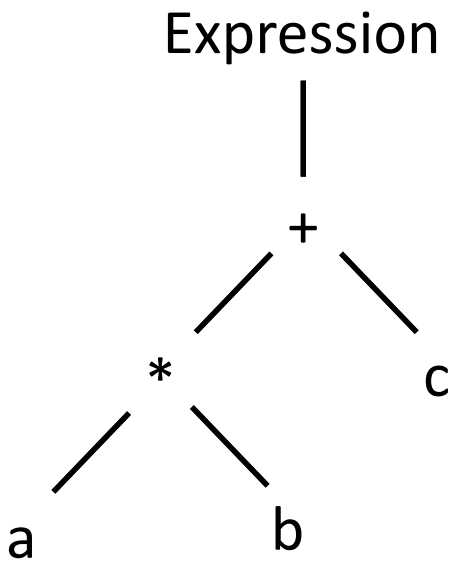
Syntax with Option

Expression = ["(" Expression ")"].

How does the language differ to the previous bracket syntax?

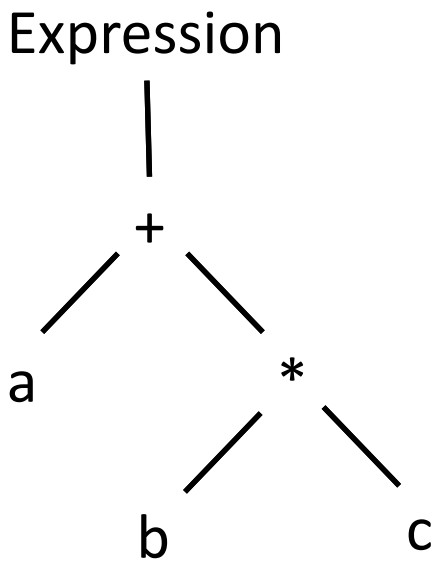
Arithmetic Expressions

a * b + c



Syntax Trees

a + b * c



How can we define this in EBNF?

Arithmetic Expressions

Precedence:
* has stronger binding than +

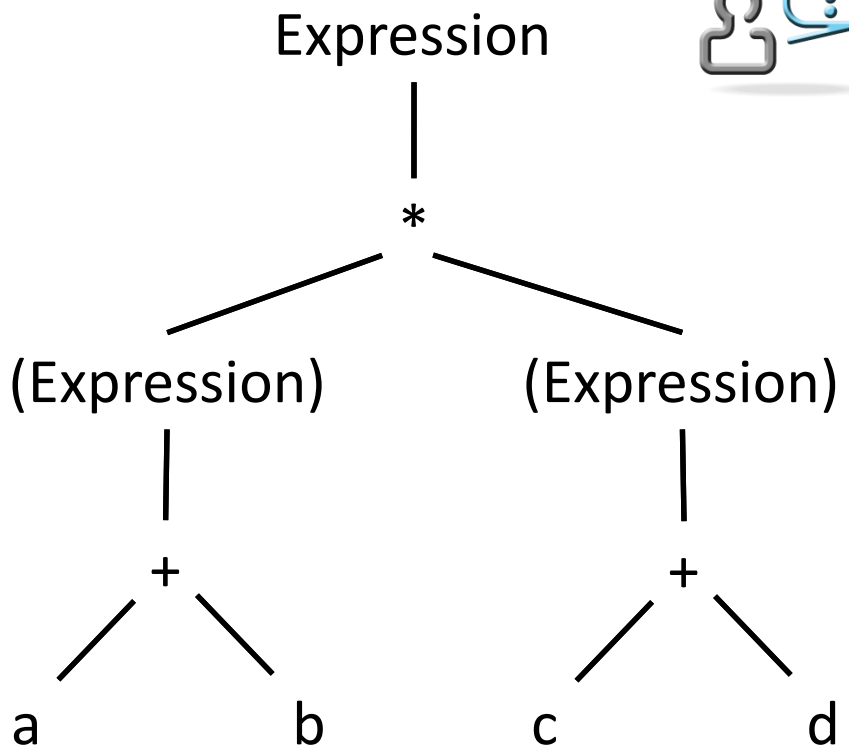
```
Expression = Term | Expression "+" Term.
Term = Variable | Term "*" Variable.
Variable = "a" | "b" | "c" | "d".
```

Left associativity for
a + b + c and a * b * c

Explicit Grouping

(a + b) * (c + d)

How do we need to extend the syntax?



Arithemtic Expressions

```
Expression = Term | Expression "+" Term.
Term = Factor | Term "*" Factor.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".
```

More Compact Syntax

Arbitrary repetition, incl. none

```
Expression = Term { "+" Term }.
Term = Factor { "*" Factor }.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".
```

Left associativity of terms and factors to be semantically defined

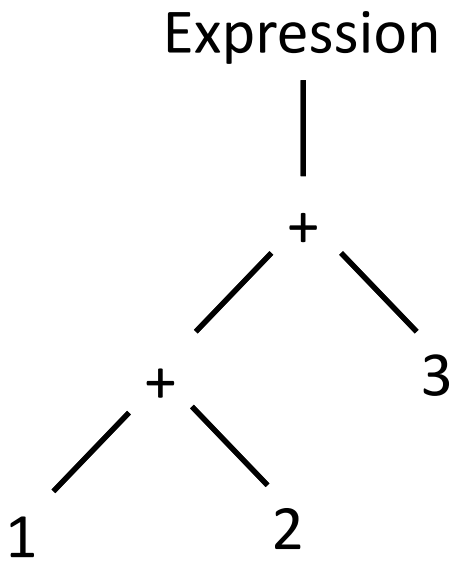
Problematic Syntax

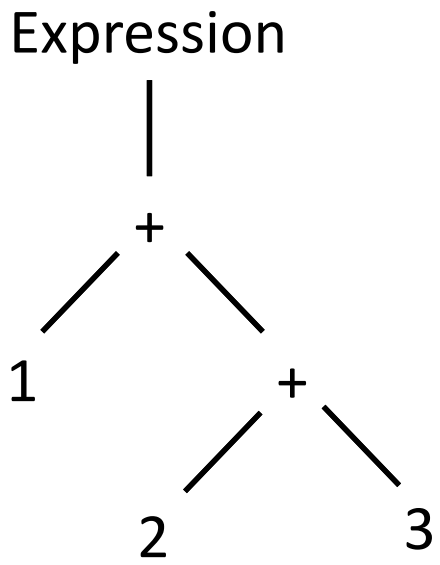
```
Expression = Number | Expression "+" Expression.
Number = "1" | "2" | "3".
```

What is the problem?

Ambiguity

1 + 2 + 3





Multiple possible syntax trees
→ Useless syntax

Special Cases

- `""""` is double quote as terminal symbol
- `"A" | .. | "Z"` is a character between A and Z
- White spaces are not covered by the syntax
  - Except if syntax depends on it, e.g. Python or F#
- Comments are not covered by the syntax either

Other Syntax Formalisms

- EBNF also has a ISO standard version, difference:
  - Concatenation with , ending with ;
  - `Expression = [ "(", Expression, ")" ];`
- Original Backus-Naur-Form (BNF)
  - Without repetition, only recursion, empty symbol ""
  - `<Expression> ::= "(" <Expression> ")" | ""`
- Other forms such as ABNF (Augmented BNF)

Summary

Syntax is formally defined by:

- Set of terminal symbols.
- Set of non-terminal-symbols
- Set of productions
- Start symbol

Language = Set of terminal symbol sequences that can be derived from the syntax

Syntax shall be unambiguous!

Review: Learning Goals

- ✓ Know the fundamental structure and purpose of compilers and runtime systems
- ✓ Understand the EBNF syntax formalism

Further Reading

- Dragon Textbook:
  - Section 2.1 (Introduction)
  - Section 2.2 (Syntax Definitions)
- Optional, if interested:
  - N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? Communications of the ACM, 20(11): 822, 823, Nov. 1977.
  - EBNF ISO Standard. ISO/IEC 14977.

Appendix

Self-Study

EBNF in EBNF

Syntax = { Production }.
Production = Identifier "=" Expression ".".
Expression = Term { "| " Term }.
Term = Factor { Factor }.
Factor = Identifier
| String
| "(" Expression ")"
| "[" Expression "]"
| "{" Expression "}".
Identifier = Letter { Letter | Digit }.
String = """ { Character } """.
Letter = "A" | ... | "Z".
Digit = "0" | ... | "9".
Character = *any character, escaped "".*

Compilers & Interpreters
Lexical Analysis

Week 1, Wednesday

Prof. Dr. Luc Bläser

Last Lecture - Quiz

```
Statement = IfStatement | Assignment.
IfStatement = "if" "(" Expression ")"
              Statement
              [ "else" Statement ].
...
```

Can you identify a problem with this language?

Ambiguity

```
if (x == 0)
  if (y == 0)
    x = 1;
else
  y = 1;
```

To which “if” does the “else” belong to?

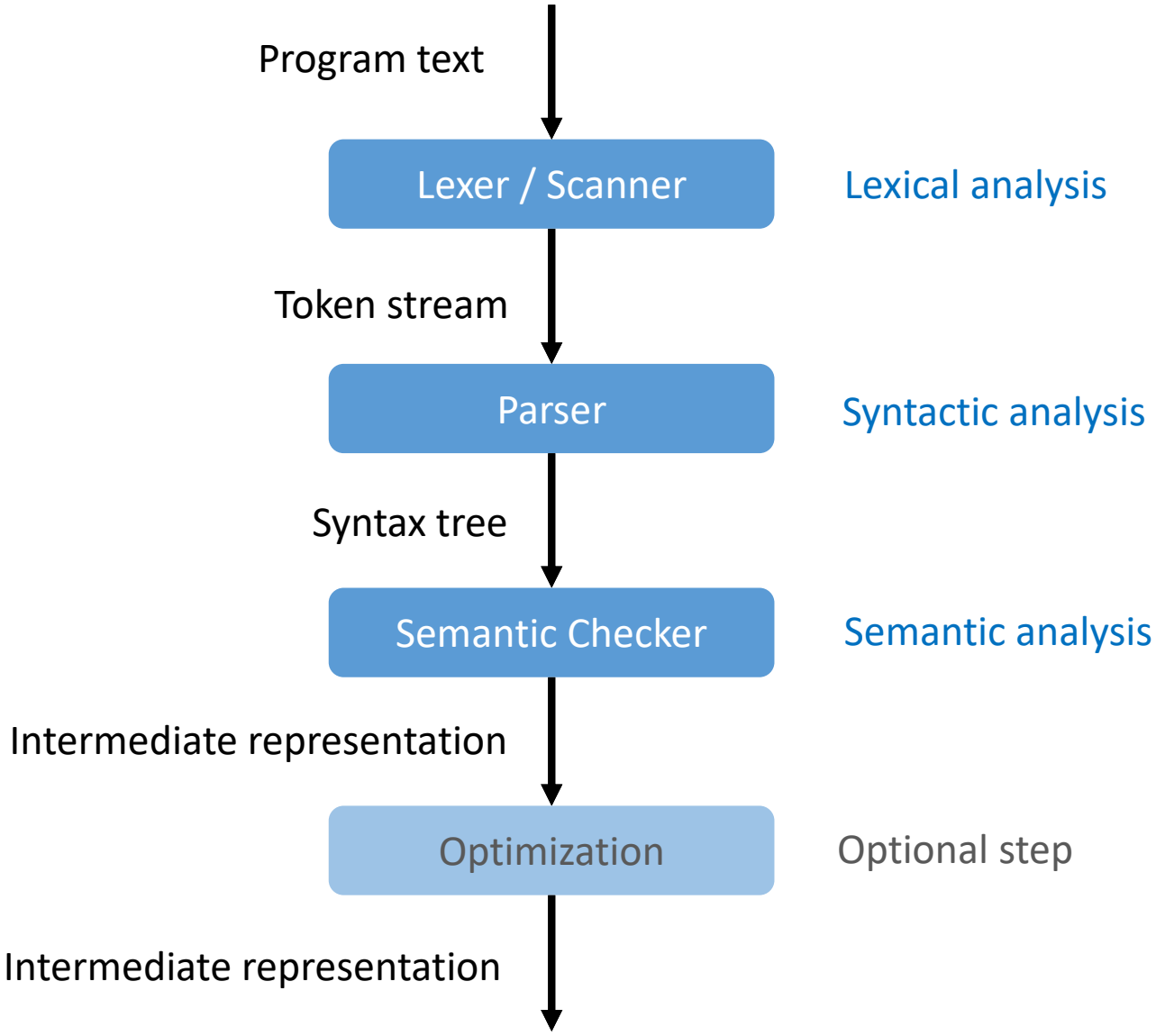
Today's Topics

- Compiler frontend
- Lexical analysis
- Tokens
- Regular vs. context-free grammar
- Tools and implementation

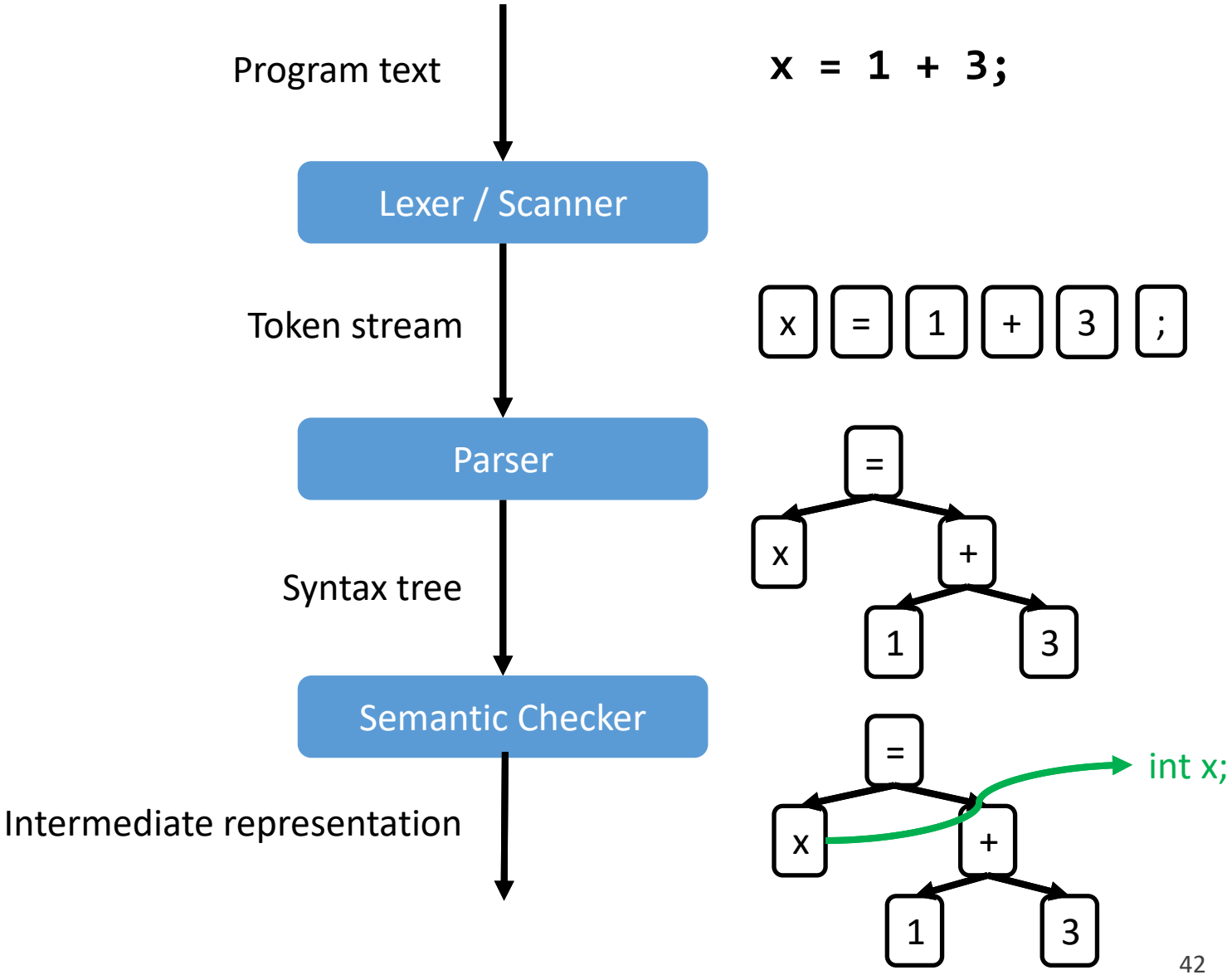
Learning Goals

- Know the purpose and functionality of a lexer
- Be able to develop a lexer on your own

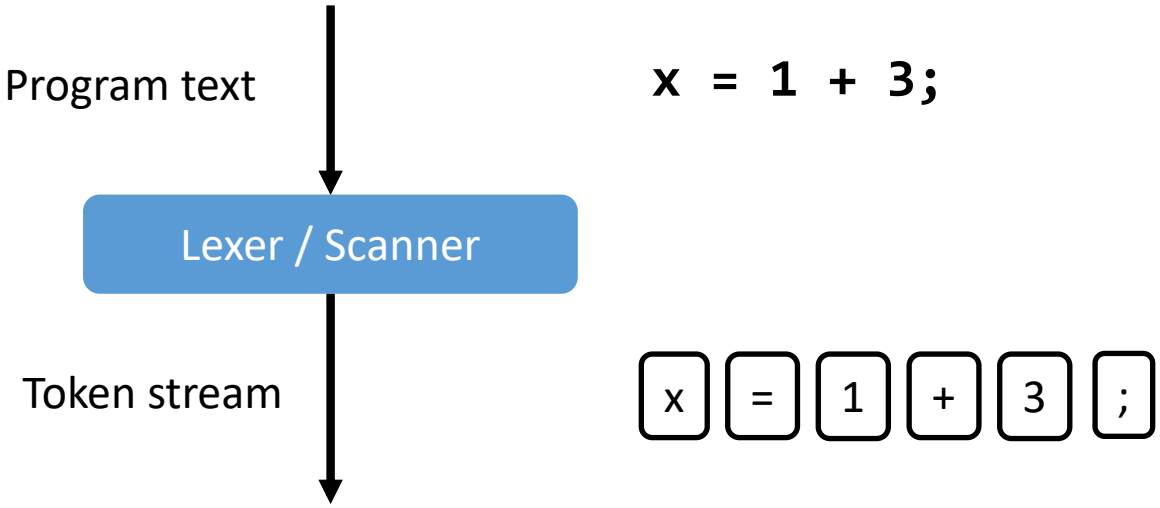
Compiler Frontend



Run-Through Example



Our Focus: Lexer



Lexer / Scanner

Cares about the lexical analysis

- Input: Character sequence (program text)
- Output: Token stream (stream of terminal symbols)

Tasks of a Lexer

- Combine characters to tokens
  - Text "1234" => Token Integer 1234
- Eliminate white spaces
  - As far as it is irrelevant in the language
- Eliminates comments
  - E.g. // line comment or /* block comment */
- Records source code position
  - For error messages, debugging

Purpose of a Lexer

Simplify the subsequent syntactic analysis (parser)

- Abstraction
  - Parser does not need to care about characters
- Simplicity
  - Parser can look ahead symbol-wise, not character-wise
- Efficiency
  - Lexer works without a stack in contrast to parser

Tokens

- Static (keywords, operators, punctuation)
  - if, else, while, *, &&, (,), ;
- Identifiers
  - MyClass, ReadFile, name2
- Numbers
  - 123, 0xfe12, 1.2e-3
- Strings
  - "Hello!", "", "01234"
- Possibly more…
  - Single characters ('a', '0')

Lexeme

- A lexeme is the specific string that forms a token
  - E.g. MyClass is a lexeme of token Identifier

Regular Languages

- Lexer only supports regular languages
- Regular = specifiable as non-recursive EBNF

Examples:

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

regular

```
Expression = [ "(" Expression ")" ].
```

not regular

Other Example

```
Integer = Digit [ Integer ].
```

Is this language regular?

Analysis: Regular Language

```
Integer = Digit [ Integer ].
```

translate into equivalent syntax

```
Integer = Digit { Digit }.
```

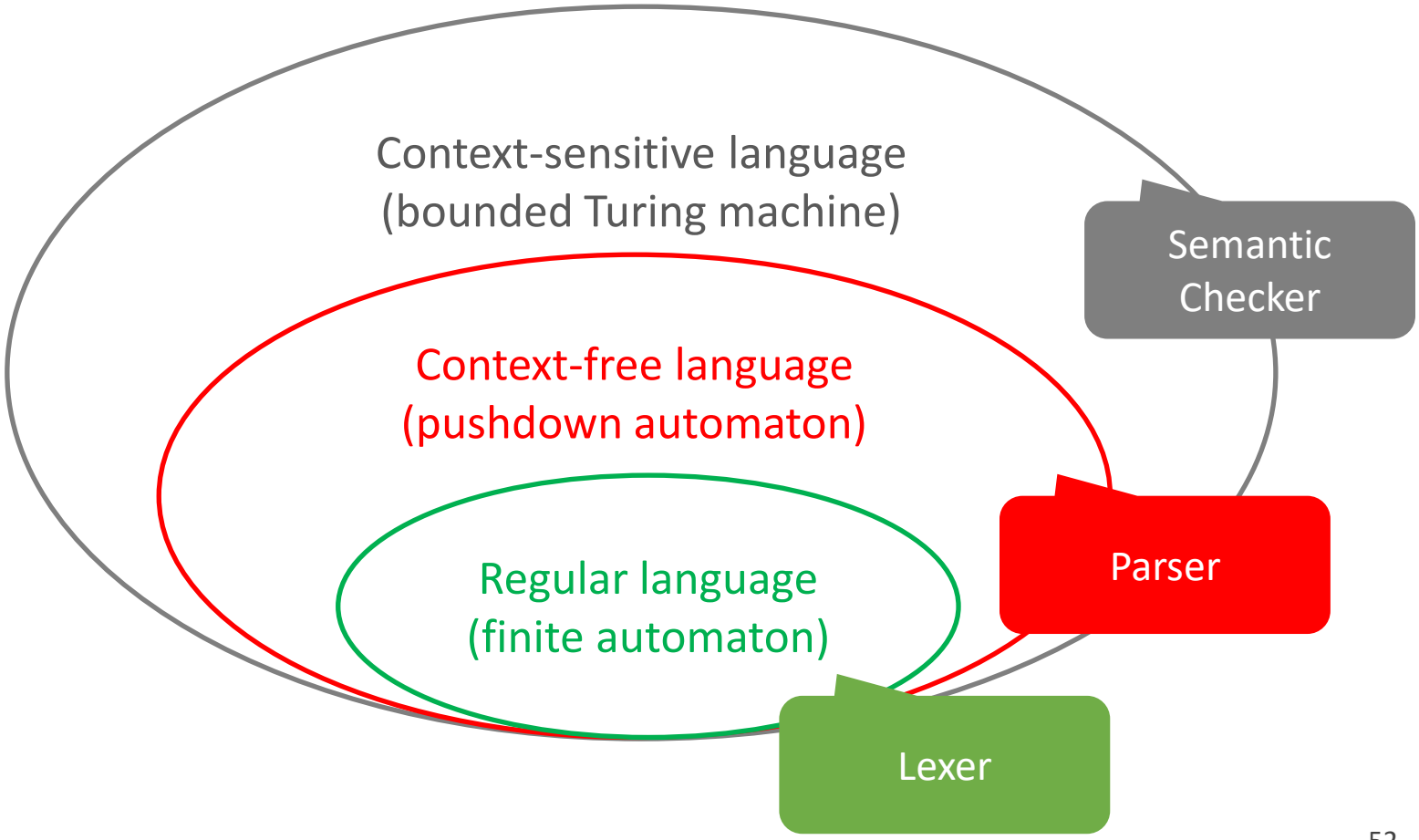
no recursion

regular

Inverse proof is more difficult:
-> use pumping lemma for regular languages

Chomsky Hierarchy

- Theoretical classification of formal languages



UCI-Java

- Exemplary language considered during this course
- Simplified version of Java
- Language report available in course material

UCI Computer Science
Spring Quarter 2019

142A Compilers & Interpreters
Prof. Dr. Luc Bläser

Language Specification

The Programming Language
UCI-Java

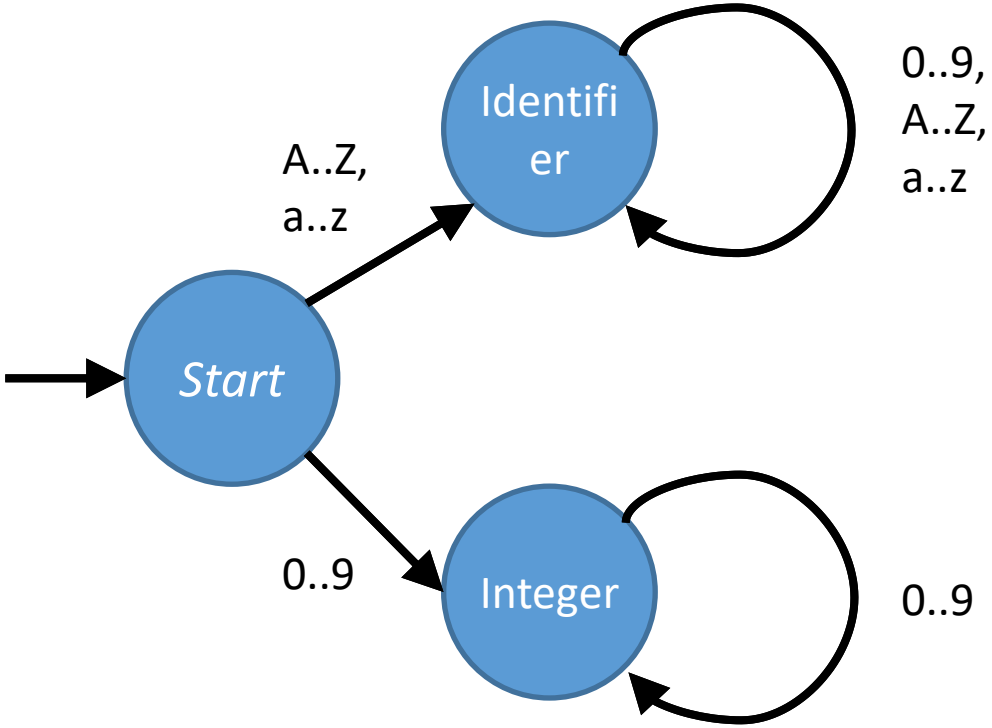
Version 1
(Revision 2019-03-07)

Identifier

- Name of classes, methods, variables etc.
- Starts with a letter, thereafter digits or letters
- (Java also supports underscores, we do not.)

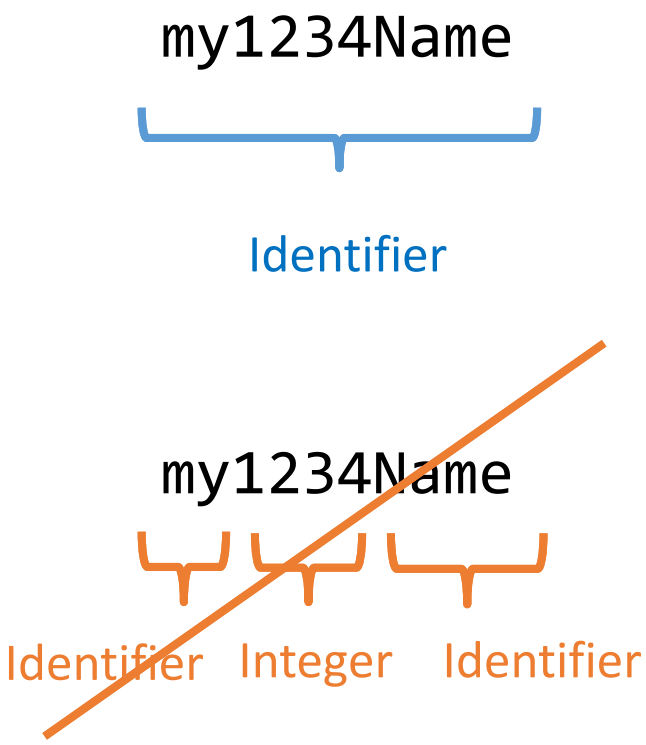
```
Identifier = Letter { Letter | Digit }.
Letter = "A" | ... | "Z" | "a" | ... | "z".
Digit = "0" | ... | "9".
```

Lexer as Finite State Machine



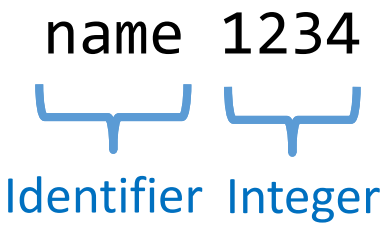
Maximum Munch

- Lexer absorbs as much as possible into a token

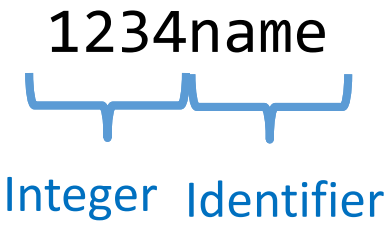


White Spaces

- Skipped by the lexer
- White space separates tokens



- Separation may also happen without white space



Comments

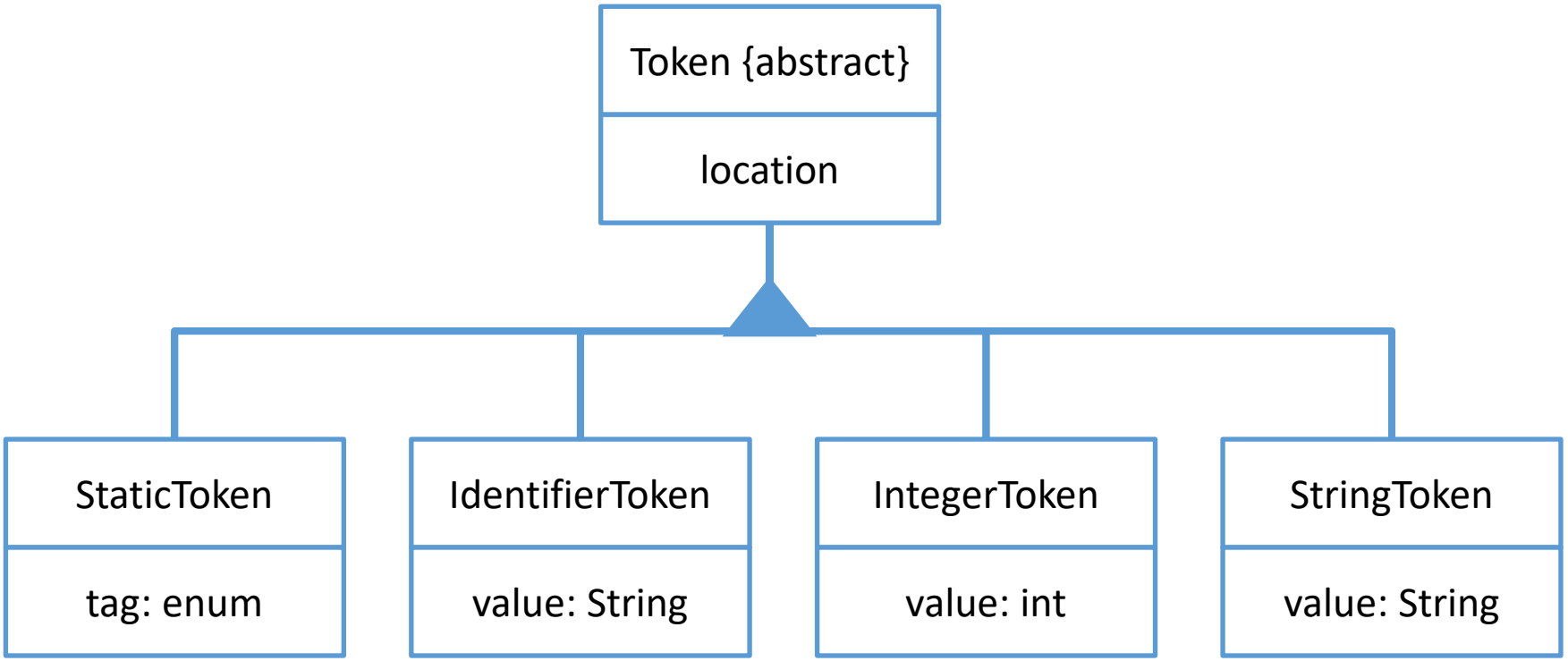
- Also skipped by the lexer
- Block comments
  - Usually not nestable => regular language

```
/* My comment block
ends here */
```

- Line comment
  - Until new line \n, actually depends on white space

```
// Comment ends at line end
```

Token Representation



Tags for Static Tokens

```
public enum Tag {
    END,
    CLASS, ELSE, IF, RETURN, WHILE, ...
    AND, OR, PLUS, MINUS, SEMICOLON, ...
}
```

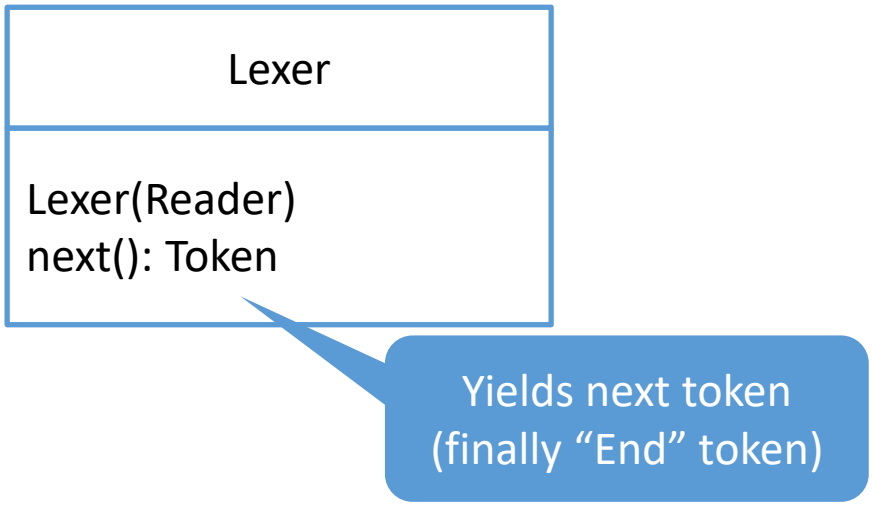
Special control symbol for lexer

Operators, punctuation

Reserved keywords

Hint: Process reserved type keywords (void, boolean, int, string) and constants (null, true, false) as identifiers in the lexer.

Lexer Class


Lexer
Lexer(Reader)
next(): Token
Yields next token
(finally “End” token)

Lexer Skeleton

```
public class Lexer {
   private final Reader reader;
   private char current;
   private boolean end;

   public Lexer(Reader reader) {
     this.reader = reader;
     readNext();
   }

   private void readNext() {
     // read next character into current
     // set end when reader has finished
   }

   ...
}
```

One Character Lookahead

Helper Methods

```
private boolean isDigit(char c) {
  return c >= '0' && c <= '9';
}

private boolean isLetter(char c) {
  return c >= 'A' && c <= 'Z' || c >= 'a' && c <= 'z';
}

private void skipBlanks() {
  while (!end && current <= ' ') { readNext(); }
}
```

Lexer Core

```
public Token next() {
  skipBlanks();
  if (end) {
    return new StaticToken(Tag.END);
  }
  if (isDigit(current)) {
    return readInteger();
  }
  if (isLetter(current)) {
    return readName();
  }
  switch (current) {
  case '"': return readString();
  case '+': readNext(); return new StaticToken(Tag.PLUS);
  case '-': readNext(); return new StaticToken(Tag.MINUS);
  ...
  }
}
```

Lexing Integers

```
private IntegerToken readInteger() {
  int value = 0;
  while (!end && isDigit(current)) {
    int digit = current - '0';
    value = value * 10 + digit;
    readNext();
  }
  return new IntegerToken(value);
}
```

Are there any special cases to be considered?

Lexing Identifiers and Keywords

```
private Token readName() {
  String name = Character.toString(current);
  readNext();
  while (!end && (isLetter(current) || isDigit(current))) {
     name += current;
     readNext();
  }
  if (KEYWORDS.containsKey(name)) {
    return new StaticToken(KEYWORDS.get(name));
  }
  return new IdentifierToken(name);
}
```

Detect Comments

- Add switch to the lexer core

```
case '/':
  readNext();
  if (current == '/') {
    skipLineComment();
    // repeat next() from beginning
  } else if (current == '*') {
    skipCommendBlock();
    // repeat next() from beginning
  } else {
    return new StaticToken(Tag.DIVIDE);
  }
```

Eliminate Comments

```
private void skipLineComment() {
  readNext(); // skip second slash
  while (!end && current != '\n') {
    readNext();
  }
}
```

How to eliminate block comments?

Lexer Extensions

- Record line/position
  - For error messages and debug info
  - Store in tokens
- Extra cases in other languages
  - Character literals
  - String/character escaping
  - Hexadecimal integers
  - Floating point numbers
- Error handling

Error Handling

- Error situations
  - Premature program end
  - String or comment not closed
  - Too large number values (e.g. int is only 32 bits)
  - …
- Error handling
  - Panic mode: Exception => compiler aborts
  - Return error token and continue
  - Correction attempts: insert, replace, swap character
  - …

Lexer Generators

- Tool generates lexer from syntax specification

	Syntax	Output	Internal
lex, flex	Regex	C/C++	C/C++
AntLR	EBNF	Java, C#, Python, C, …	Java
JavaCC	EBNF	Java	Java
Coco/R	EBNF	Java, C#, …	Java, C# etc.

and more…

Example: AntLR 4

```
grammar UCIJava;
// lexer rules
Identifier: Letter (Letter | Digit)*;
Integer: Digit+;
String: '"' .* '"';
Letter: [A-Za-z];
Digit: [0-9];
Whitespaces: [ \t\r\n]+ -> skip;
```

Lexer rules start with capital letter

ignore

Lexer Generators

- Advantages
  - Less programming work
  - Less error prone
- Disadvantages
  - Occasional conflict messages by generator
  - Predetermined token representation
  - Verbose generated code
  - Dependency on specific tool

Review: Learning Goals

- ✓ Know the purpose and functionality of a lexer
- ✓ Be able to develop a lexer on your own

Further Reading

- Dragon Textbook: Chapters 3 (Lexical Analysis):
  - Sections 3.1 – 3.5 (lexer, tokens, recognition, generator)
- Optional, if interested:
  - Dragon book, sections 3.6 – 3.9 (automaton/lexer generator theory)
  - AntLR 4.7.2, http://www.antlr.org/

Appendix

Self-Study

UCI-Java Lexer Rules in AntLR (1)

```
grammar UCIJava;

CLASS: 'class';
ELSE: 'else’;
EXTENDS: 'extends';
IF: 'if’;
INSTANCEOF: 'instanceof';
NEW: 'new';
RETURN: 'return';
WHILE: 'while';
LPAREN: '(';
RPAREN: ')';
LBRACKET: '[';
RBRACKET: ']';
LBRACE: '{';
RBRACE: '}';
SEMI: ';';
COMMA: ',’;
```

```
DOT: '.';
ASSIGN: '=';
EQUAL: '==';
UNEQUAL: '!=';
LESS: '<';
LEQ: '<=';
GREATER: '>';
GEQ: '>=';
OR: '||';
AND: '&&';
PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';
MOD: '%';
NOT: '!';
```

UCI-Java Lexer Rules in AntLR (2)

```
Identifier: Letter (Letter | Digit)*;
Integer: Digit+;
String: '"'(~'"')*'"';
Letter: [A-Za-z];
Digit: [0-9];

Whitespaces: [ \r\t\n]+ -> skip;
LineComment: '//' .+? ('\n'|EOF) -> skip;
BlockComment: '/*' .*? '*/' -> skip;
```

AntLR Lexer Integration

- Grammar file «UCIJava.g4» ending with:

```
program: EOF; // temporary syntax rule
```

- Generation:

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

- Integrate in Java program:
  - Include antlr-4.7.2-complete.jar in build path

```
var stream = CharStreams.fromString(input);
var lexer = new UCIJavaLexer(stream);
var tokens = new CommonTokenStream(lexer);
tokens.fill();
tokens.getTokens().forEach(System.out::println);
```

Automatic Lexer Generation

- RE => NFA (Thompson construction), section 3.7.4
  - Represent each rule as NFA
  - Merge with ε-transitions
- NFA => DFA (subset construction), section 3.7.1
  - May grow exponentially due to ε-transitions
- Minimize DFA (various algorithms), section 3.9.6
  - Merge equivalent states
  - Remove unreachable states
- Table-driven lexer, section 3.4.1
  - DFA transition table

RE = Regular Expressions
NFA = Non-Deterministic Finite Automaton
DFA = Deterministic Finite Automaton