Compiler Construction: Practical Introduction

(Almost) no theory
but a lot of code

A lot of information is
outside of the course

Samsung Compiler Bootcamp

Samsung Research Russia
Moscow 2019

Lecture 1
Introduction to the Area

- Why the Course?
- Who we are?
- References & Textbooks
- Some history
- Compilation: an ideal picture and the reality

- Phase 1: **Lexical analysis**

Why the Course?

- The area of compiler construction traditionally is one of three cornerstones of Computer Science (two others are databases & operating systems).
- The area includes many fundamental results: formal grammar theory, type theory, graph theory, algorithm theory and many others.
- The similar courses are given in universities all over the world.
- It's the extremely interesting area ☺.

Who we Are?

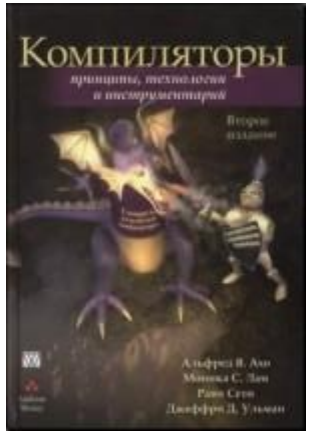
- **Eugene Zouev**
  Moscow Univ., Swiss Fed Inst of Technology (ETH Zürich), EPFL (Lausanne); PhD (1999).
  Interstron: 1st Russian **C++** compiler, 1999-2000.
  Samsung: **Swift** language prototype compiler for Tizen & Android.
  Recent books: **«Редкая профессия»**, ДМК Пресс, Москва 2014.
  Software Design for Resilient Computer Systems, Springer, 2019

- **Dmitry Botcharnikov**
  Interstron: ***C/C++* compiler back-end** for several microprocessors.
  Samsung: V8 (JavaScript) optimization, Swift compiler prototype, .NET for Tizen, NPU compiler, DSP compiler.

- **Sergey Ignatov**
  Sun: *C/C++* optimizing back-end for Intel platform.
  Intel: loop optimizations for *C++* and Fortran
  Samsung: .NET for Tizen, NPU compiler, DSP compiler.
  Books: Translation in Russian of **Muchnik's book**!

References (1)

- Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D. Ullman, **Compilers. Principles, Techniques, & Tools**, Second Edition, Addison-Wesley, 2007, ISBN 0-321-48681-1. ("**Dragon book**")



Russian translation:
Ахо, Альфред В., **Лам**, Моника С., **Сети**, Рави, **Ульман**, Джеффри Д. **Компиляторы: принципы, технологии и инструментарий**, 2-е изд.: Пер. с англ.- М.: ООО «И.Д.Вильямс», 2008.- 1184 с.: илл. ISBN 978-5-8459-1350-4.

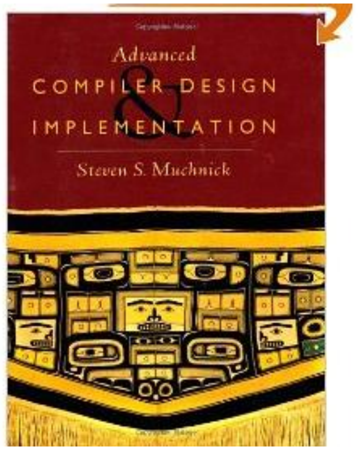
References (2)

- N.Wirth, **Compiler Construction**, Addison-Wesley, 1996, ISBN 0-201-40353-http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf, 2005.

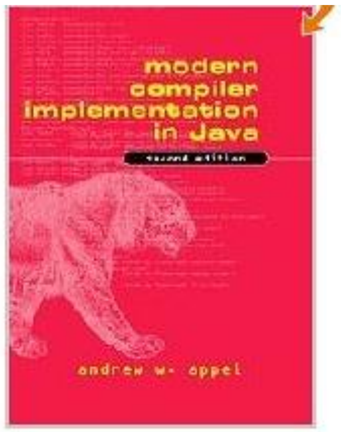
  Russian translation:
  Никлаус Вирт, **Построение компиляторов**,
  ДМК-Пресс, 2010,
  ISBN 978-5-94074-585-3.

References (3)

- S.Muchnik.
  Advanced Compiler Design and Implementation
  1997, ISBN 1-55860-320-4.



- Andrew Appel
  Modern Compiler Implementation in {Java, C, ML}
  1998

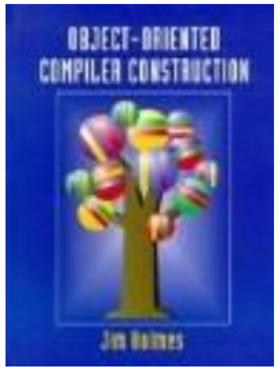


References (4)

Jim Holmes

Object-Oriented Compiler Construction

1994



Textbooks: Problems (1)

- A lot of theory but a few actual examples (even if there are many “exercises”).
- Many books in compilers (except some ☺) are not *practically-targeted*.
- The books consider **artificial**, “model” languages: small, simple, regular, “academic”. Real languages are much more complicated and typically have (very) bad design – hard to implement.

  Typically, books do not discuss how **real language features** are implemented in compilers.

Textbooks: Problems (2)

- Language semantics is the most important part of any language (syntax is just “decoration”), and the major part of any compiler deals with semantics, not with syntax. However, textbooks consider syntactic aspects of languages and compilers very carefully, in contrast with semantic implementation issues…
- Some aspects are not presented in those books **at all**: e.g., development tools, error recovering, garbage collection, language-specific optimizations etc.

Extra Information Sources

- Magazine papers, PhD theses, conference presentations & talks.
- Source codes of real compilers.
- International standards for PLs.
- Some extra books/papers.

Prerequisites

- Some knowledge of one or two programming languages is highly recommended (the more the better ☺)
  - C, **C++**, ***C#***, **Java**, Pascal, Python, …
- The preference is Object-Oriented paradigm.
- Some experience in working with modern IDEs is also needed.
  - Visual Studio, Eclipse, IDEA, NetBeans, …

Semantic Gaps

Application domain

Train traffic management system:
Trains, velocity, distance, railways, switches, train stations, conflict resolution, time schedule etc.

Simulation of the Universe evolution:
Planet movement rules, planet systems, stars classification, star activity etc.

…

Computer program/ programming language

- Variable, array, …
- Function, operator, procedure
- Control structures
- Data types
- Module, class, interface
- …

Computer hardware

- Memory cell
- Memory address
- Register
- Instruction, instruction set
- …

Language Design

Compilation

Compilation: The Goal & The Task

- **The Goal:**
  To overcome the «semantic gap» between human's way of thinking and computer solving task defined by a human.
- **The Task:**
  To transform a human-written and human-friendly ☺ program to a form which can be executed by a computer.

Machine Code & Assembly Language

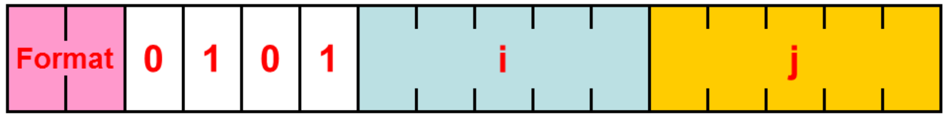
Off-topic

Mnemonic notation:
ADD 5 7

ADD i j

- **The ADD instruction denotes the two's complement arithmetic addition. The contents of registers Ri and Rj are arithmetically added, and the result is put into the register Rj.**
- The instruction format is as follows:

Binary code of the instruction:
1101010010100111



Hexadecimal code of the instruction:
D4A7

- Memory state is not considered in the instruction, and the memory state does not change.
- If the addition gives a result which cannot be put into the format specified in the instruction, then **overflow** happens

Assembler code:
```
.format 32
R5 += R7
```

Suggested assembly statement for the ADD instruction:

```
Rj += Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

Compilers: Some History (1)

Off-topic

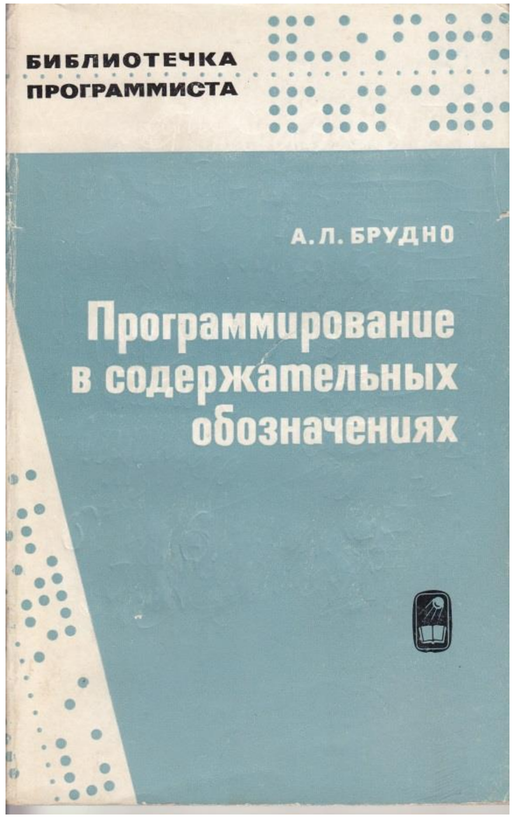
~~Автокод~~	->	Ассемблер (Язык ассемблера)
~~Автокод 1:1~~		Assembler (Assembly language)

~~Программирующая программа~~

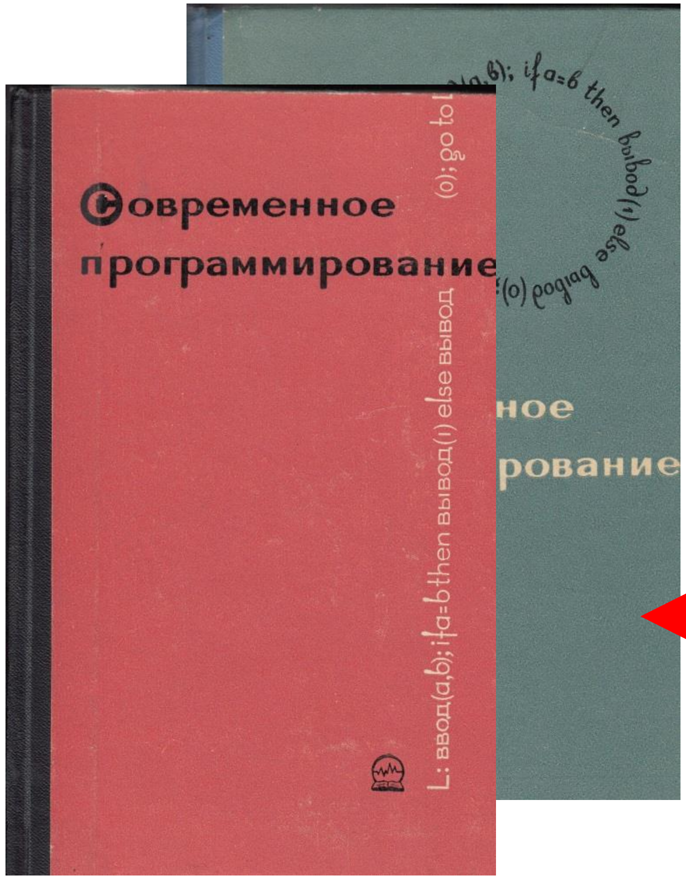
->	Транслятор	->	Компилятор
	Translator		Compiler

Compilers: Some History (2)

Off-topic



1968


Современное программирование
L: ввод(a,b); if a=b then вывод(1) else вывод(0); go to L
ное
рование

1966-1967

Appendix:
More than **300** translators for Fortran, Cobol, Jovial etc. for various computers

Compilers: Now and Then

Off-topic

The Past:

- Compiler construction **was** the high level of the programming art, the sign of the top command and/or individual mastery.
- Compilers were among the most complicated software components: one of three cornerstones of computer science (two others are Operating Systems and Databases).

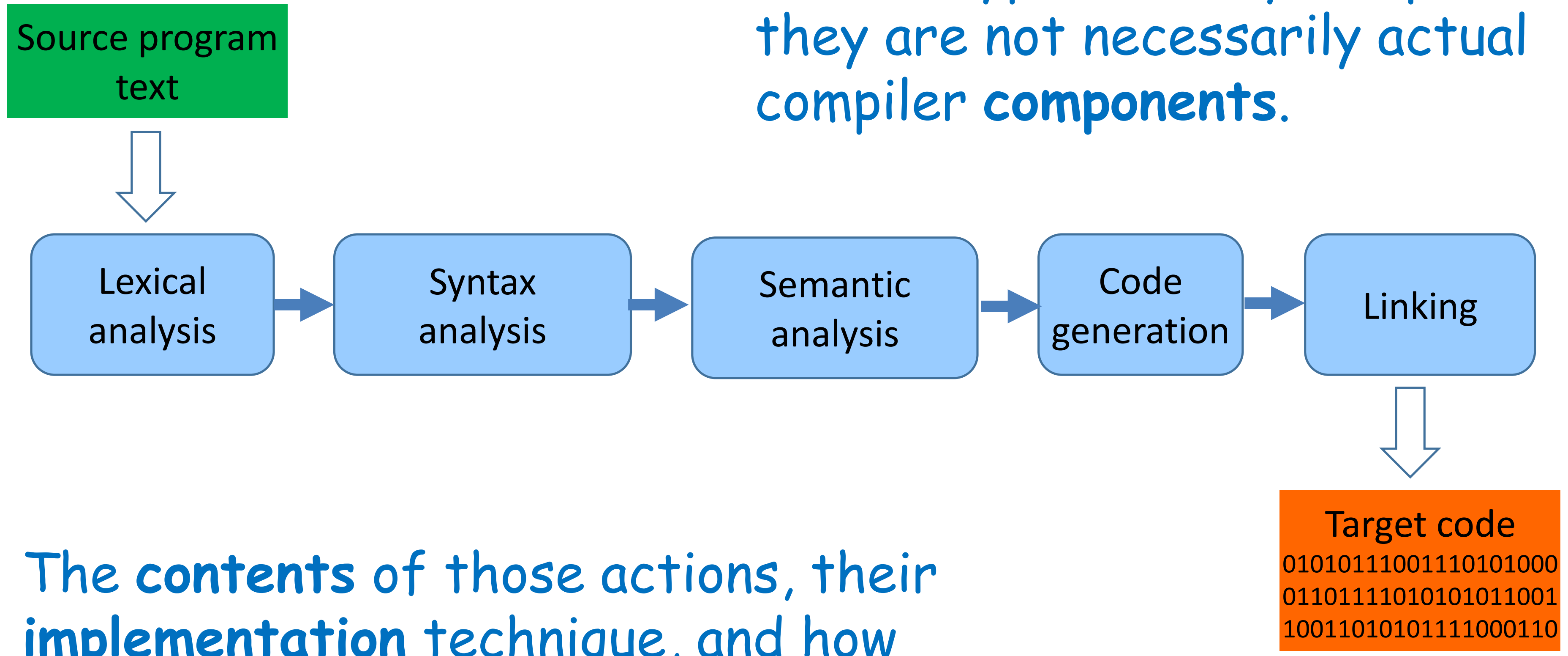
The Present:

- Compiler construction **is** a sort of usual programming job. The average programming qualification became much higher; to make a compiler is under the force of many.
- Software in general becomes much more complicated; for example, implementation of an Internet browser or an XSLT processor is not an easier job than to write a compiler.
- Compiler construction field is now very monopolized. It's hard for a new compiler to compete with existing ones – even if it's better.

Compilation: An Ideal Picture

A program written by a human (or by another program)

Blue squares just denote some **actions** typical to any compiler; they are not necessarily actual compiler **components**.



A program binary image suitable for immediate execution by a machine

The **contents** of those actions, their **implementation** technique, and how they **interact** with other actions – is just the subject of the course.

Compilation Phases: C++

ISO/IEC 14882:2011(E) Programming Languages -- C++, Third Edition (Excerpt from the section 2.2).

Initial source text analysis

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set…
2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines…
3. The source file is decomposed into **preprocessing tokens**…

Preprocessing

4. **Preprocessing directives are executed**, macro invocations **are expanded**, and _Pragma unary operator expressions are executed...

Literal processing

5. Each source character set member in a character literal or a string literal, as well as each escape sequence and universal-character-name in a character literal or a non-raw string literal**, is converted** to the corresponding member of the execution character set…
6. Adjacent string literal tokens **are concatenated**.

Translation

7. Each preprocessing token is converted into a **token**.
8. The resulting tokens are **syntactically and semantically analyzed** and translated as a **translation unit**. Each translated translation unit is examined to produce a list of required instantiations… The definitions of the required templates are located. All the required instantiations are performed to produce **instantiation units**.

Linking

9. All external entity references are **resolved**. Library components are **linked** to satisfy external references to entities not defined in the current translation. All such translator output is collected into **a program image** which contains information needed for execution in its execution environment.

Compilation stages: C# compiler

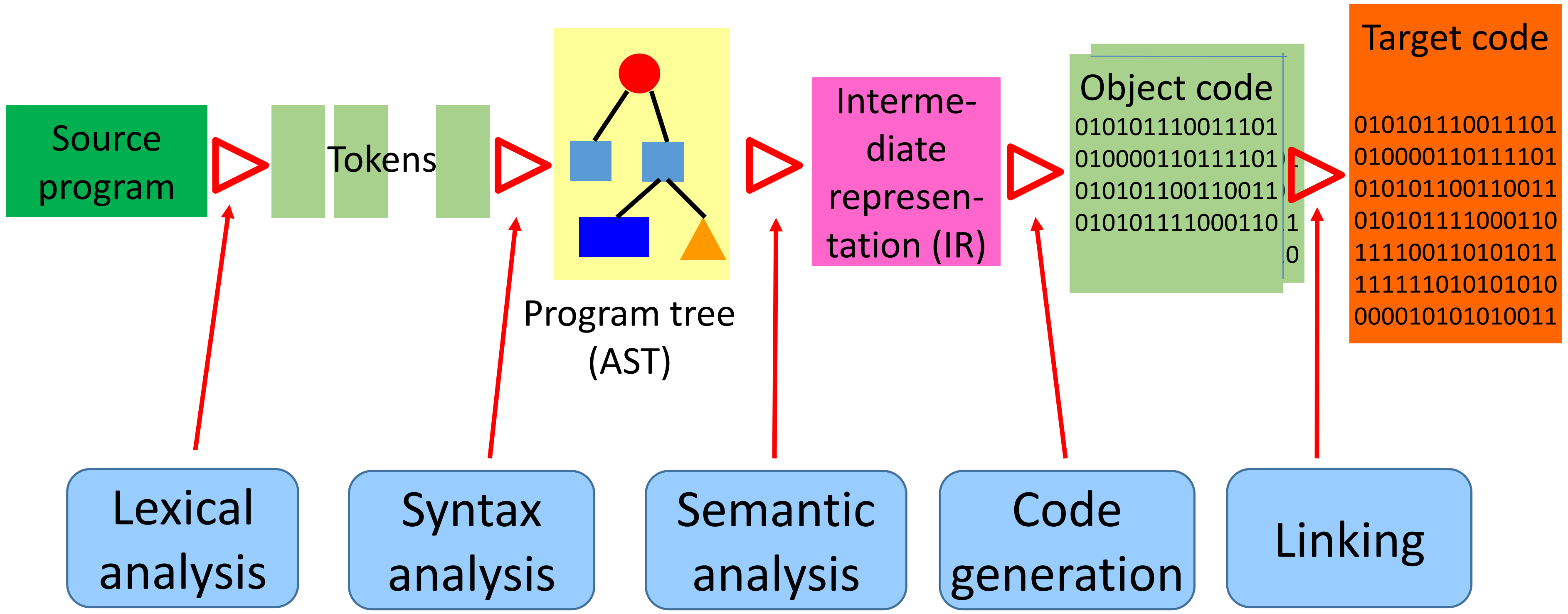
1. Primary lexical analysis (the result is a stream of tokens).
2. High-level syntax analysis (method bodies are not processed but get stored).
3. Declaration processing.
4. Two passes checking cyclic type & generic dependencies.
5. Two passes for some semantic checks and constant fields processing.
6. Lexical analysis (!) of method bodies, constructors, properties and indexers.
7. Type analysis in method bodies.

Subsequent stages are actually AST tree “walkers” modifying the AST.

- Loop conversion: replacing loops for goto’s & labels.
- **Eight** passes looking for various types of errors.
- **Fourteen** (!) passes for optimization and simplifying AST.
- **Four** passes generating MSIL-код.

Compilation Data Structures

This is a different, **data-centric** view at the compilation process. Compilation is **a sequence of transformations** of a source program.



The Ideal Picture: Modifications (1)

- There may be **several** source units. They can be represented not only as disk files, but may have arbitrary nature (generally, a stream of bytes).
- Lexical analysis may be implemented as a **separate "pass"**, or as a component invoking **"by demand"**, or…
- Lexical analysis can comprise **several "passes"**.
  Example: *C/C++*: first preprocessing (macro expansion), and then "true" lexical analysis.
- Lexical & syntax analyses can work either **separately** or **simultaneously** (interacting with each other). Will be examples for C++ later.

The Ideal Picture: Modifications (2)

- Syntax analysis can be implemented as a single “pass”, or as a **sequence of “passes”** (Java, Scala are examples), and/or can run together with semantic analysis.
- Syntax & semantic analyses can be implemented **as the single stage** (only for simple languages like Pascal or Oberon).
- Typically, semantic analysis can include **several “passes”**, or tree traverses (examples for C# and Scala follow); the AST gets modified while each “pass”.

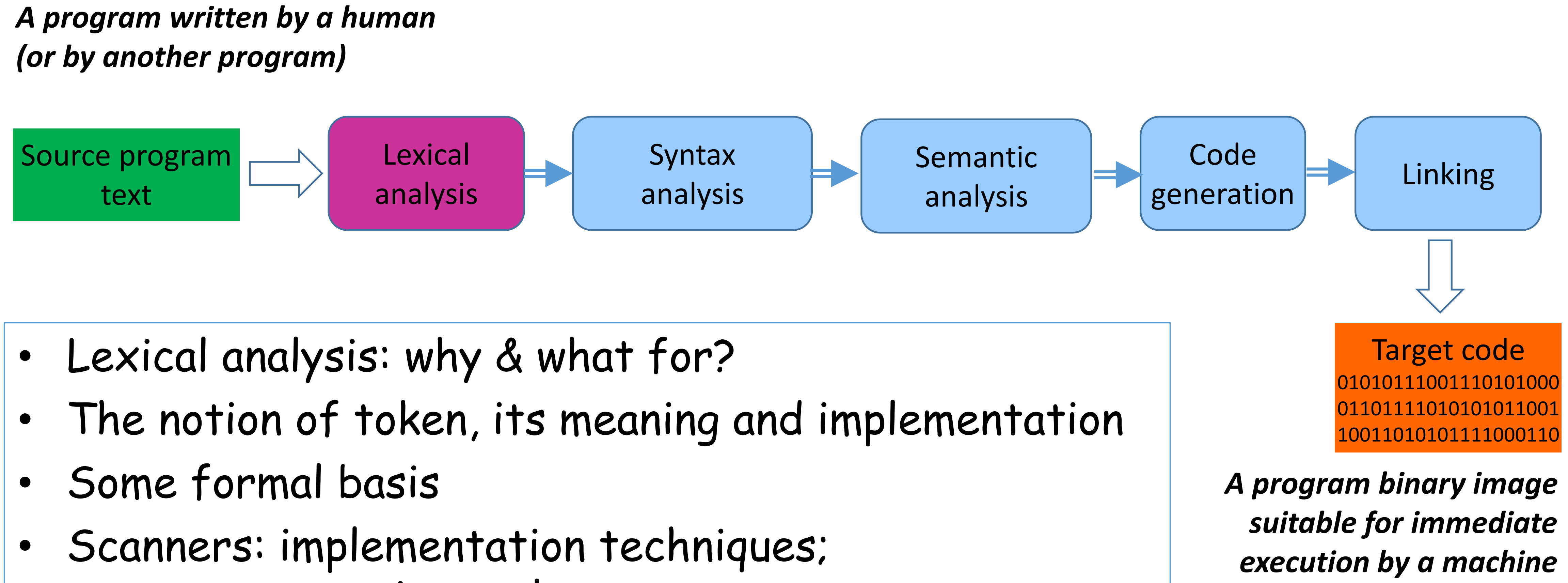
The Ideal Picture: Modifications (3)

- Program tree is build for the complete program or sequentially, for its parts (functions, classes).
- Intermediate representation: either specially designed *compile-time structures*, (e.g. Control Flow Graph – CFG) or *the program tree* itself, or some (other) language.
- Examples: lower-level language *C--*; standard *C* language as intermediate program representation.
- Can be several intermediate languages/structures:
- see Muchnik; gcc: GENERIC, GIMPLE, RTL.

The Ideal Picture: Modifications (4)

- On each compilation phase (from lexical analysis to code generation) external sources can be added to the program: either in the textual form (include files in C/C++), or as precompiled components (libraries).
- Some languages assume/require **linking** stage which is often considered as a standalone phase.
- Code generation phase often (typically) includes some **optimizing sub-phases**.

Phase 1: Lexical Analysis



- Lexical analysis: why & what for?
- The notion of token, its meaning and implementation
- Some formal basis
- Scanners: implementation techniques;
  scanner generation tools
- Non-standard issues
- Scanner & parser integration: the architecture

Lexical Analysis: The Idea

- To convert the source stream of bytes
  to a sequence of tokens: the symbols from the language alphabet.

```
class Derived : public Base
{ int m1 …
```

Source program

Stream of bytes

c	l	a	s	s		D	e	r	i	v	e	d		:	p	u	b	l	i	c		B	a	s	e	\n	{		i	n	t		m	1	...

Sequence of tokens

Token: class
Category: *The keyword*

Token: ␣
Category: *The whitespace*

Token: Derived
Category: *The whitespace*

Token: ␣
Category: *The whitespace*

Token: :
Category: *The delimiter*

Token: Base
Category: *The whitespace*

Token: \n
Category: *The whitespace*

Token: {
Category: *The delimiter*

Tokens & Lexemes: Terminology

From "Dragon Book"

Token Токен: Lexeme's category	**Pattern** Generalized category description	**Lexeme** A concrete text snippet, that falls under a certain category
Keyword	Joint sequence of characters, with neither letter nor digit after it. The set of sequences **is fixed** in the language definition	`if`
Comparison operator sign	One of signs < or >, or one of sequences `<=`, `>=`, `==` or `!=`	`>=`
Identifier	A sequence of letters, digits and underscore characters starting with letter or underscore.	`abracadabra` `a_long_identifier`

Integer unsigned constant	A sequence of decimal digits.	`0`
		`17`
		`123456789`

The full set of token categories (the <u>language alphabet</u>) is defined by the language.

Typical categories:

- Identifiers
- Keywords: `const`, `int`, `class`, `public`
- Literals: numerics, strings
- Delimiters: `[ ] { } ( ) , . ;`
- Operators: `+ - / * & < <=`
- Whitespaces: blanks, EOLs, comments

Token categories: Some Comments (1)

Keywords & identifiers

Typically, identifiers and keywords are **lexically identical**

The category can depend on the context:

PL/I:

```
IF IF == THEN THEN THEN = ELSE ELSE ELSE = END END
```

```
IF IF == THEN
THEN
     THEN = ELSE
ELSE
     ELSE = END
END
```

```
IF IF == THEN
THEN
     THEN = ELSE
ELSE
     ELSE = END
END
```

C#:

```
class C {
     int x { get; set; }
}
```

get, set are **keywords**

```
void f() {
     int get;
     double set;
}
```

get, set are **identifiers**

Token categories: Some Comments (2)

Keywords & identifiers

- Keywords are explicitly marked: by leading underscore or by quotes.

Эль-76:

```
_если итерация = последняя _то
    ЗакончилиЦикл := _истина; Выход!(777)
_все;
```

Algol-60:

```
'if' Iteration = Last 'then'
    'goto' Finish
'end';
```

- Keywords are a fixed set of identifiers with special meaning: Pascal, C/C++, Java/C# etc.

Token categories: Some Comments (3)

Spaces

- Spaces always separate tokens and are never a part of any token (except strings).

```
A  B
A  5
5  10
```

Error: this is **not** identifier AB

Error: this is **not** identifier A5.

Error: this is **not** integer value 510.

Tricky question ☺: how to interpret C++ constructs like `C c;` ?

Comments

- Typically, comments are treated as *whitespaces*; they do not alter the program semantics => they are dropped by the lexical analyzer.
- ***Documenting comments***: they do not alter the program semantics, but compiler should process them somehow (they even may go to the object code!)

```
// This is just a comment

/// <summary>This is «documenting» comment (C#).
/// </summary>

/** This is also documenting comment (Java) */
```

Typically, documenting comments serve as a prototype for creating program documentation - either by the compiler itself, or by a standalone tool.

Formal Basics (1)

- Lexeme's structure is typically described by **regular grammars**. All the grammar rules have the following configuration:

  `A -> Ba` or `A -> a`

  Here, A, B – **nonterminal** symbols, a – a **terminal** symbol: an element of the grammar's alphabet.

  Example:

```
identifier -> letter
identifier -> identifier letter
identifier -> identifier digit
letter -> "a"
letter -> "b"
...
letter -> "z"
digit -> "0"
...
digit -> "9"
```

Formal Basics (2)

- Regular grammars are often represented in a more compact notation called **regular expressions**.

  Example:

  ```
      identifier -> letter [ letter | digit ]*
      letter -> ["a".."z"]
      digit -> ["0".."9"]
  ```

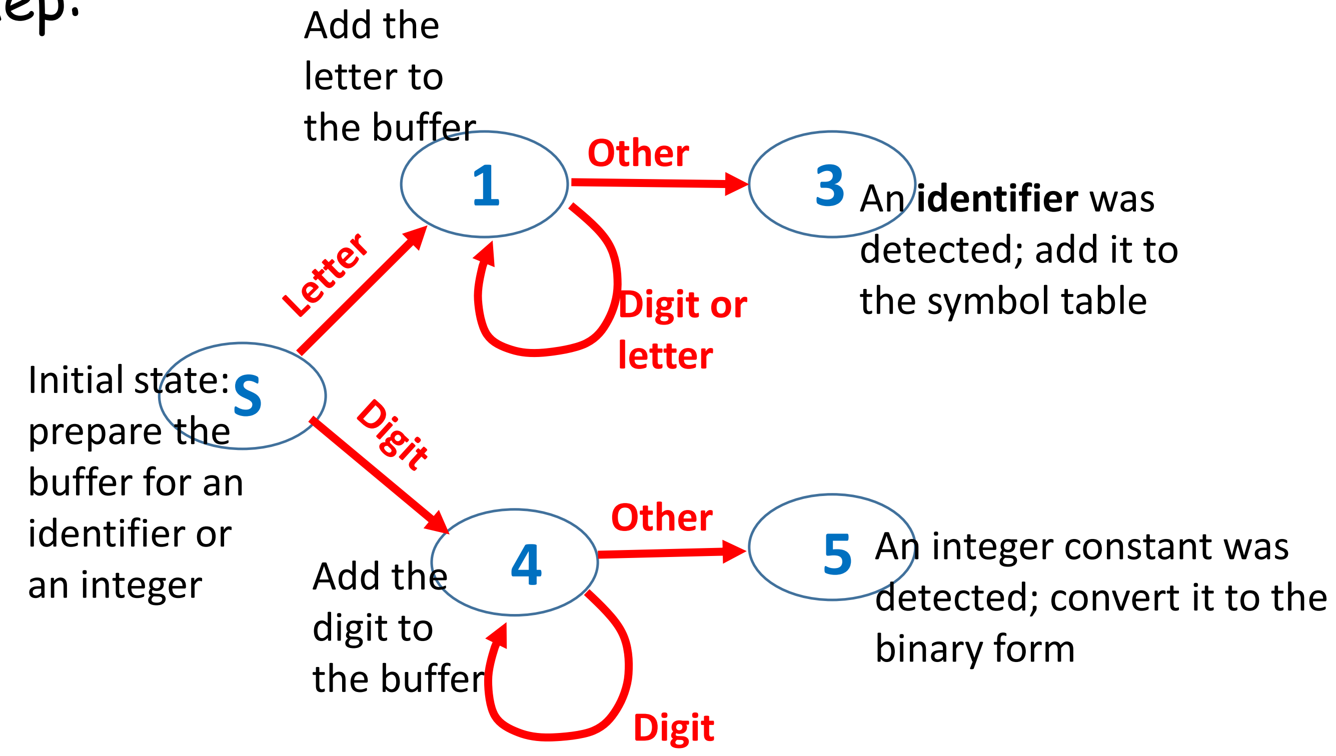
- The main statement concerning regular grammars: to scan a token successfully (and therefore, to determine that a token belongs to the given grammar) it's enough to know the **current** scanner **state** and **only one** input character.

- A scanner for regular expressions can be defined by the notion of the **finite state machine**.

Formal Basics (3)

- **Finite State Machine**:
  A (virtual) system that can have a **state** at each moment. When a character comes to the machine it changes its state and performs an action.

  Пример:



Add the letter to the buffer
1
Other
3
An **identifier** was detected; add it to the symbol table
Letter
Digit or letter
Initial state: prepare the buffer for an identifier or an integer
S
Digit
Other
4
5
An integer constant was detected; convert it to the binary form
Add the digit to the buffer
Digit

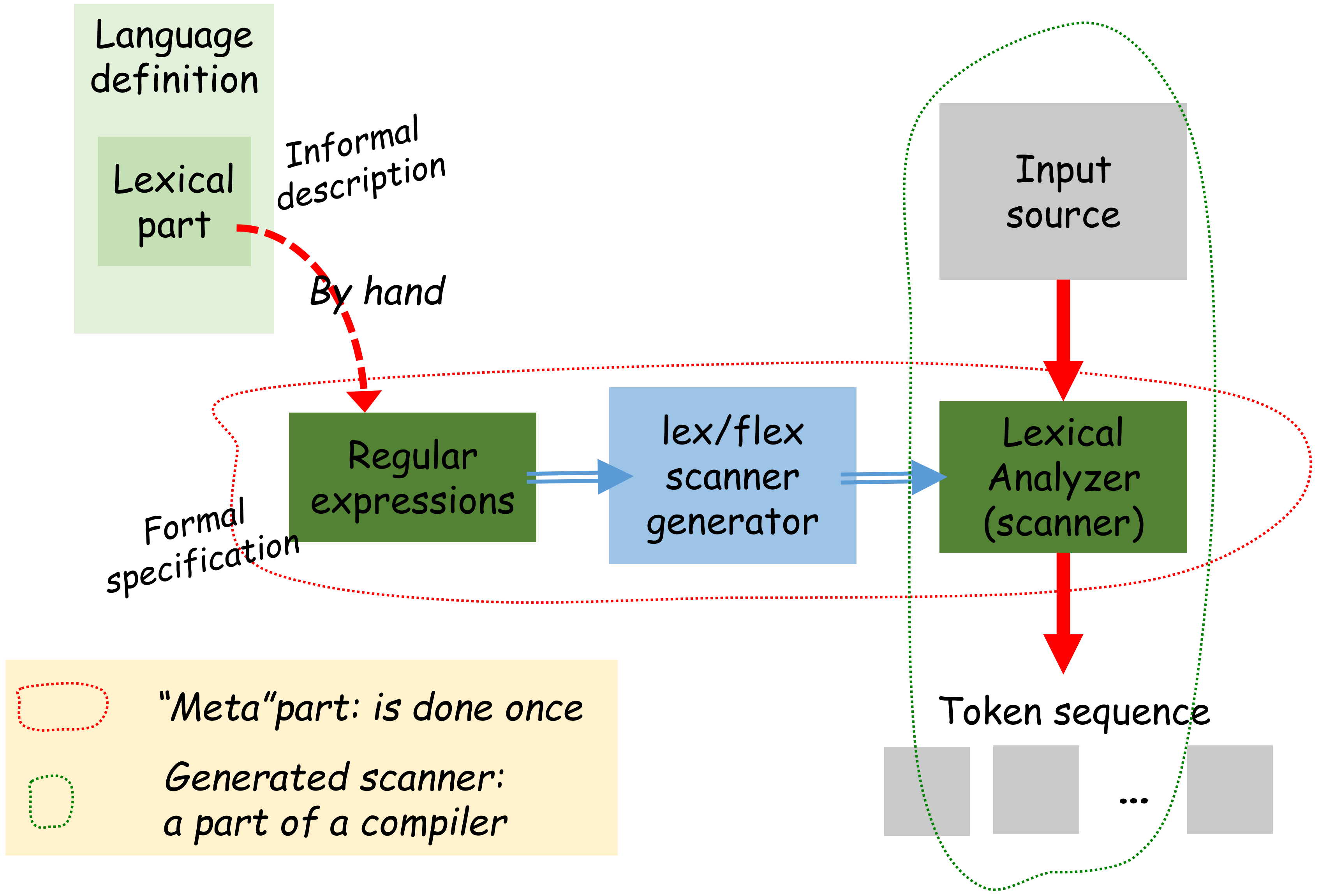
lex/flex Scanner Generator

- **lex/flex**
  For a given formal specification (consisting of regular expressions) generates a program that detects tokens in accordance with the specification.

- **Lex - A Lexical Analyzer Generator**
  M. E. Lesk and E. Schmidt
  http://dinosaur.compilertools.net/lex/index.html

- Typically, lex/flex is used together with the parser generator **yacc/bison**.

lex/flex Scanner Generator



Scanner implementation (1)

```
enum TokenCodes
{
	tkEQUAL_EQUAL,
	tkNOT_EQUAL,
	tkEXCLAMATION,
	tkPERCENT_EQUAL,
	tkPERCENT,
	...
	tkIdentifier,
	tkKeyword
}
```

The get() function returns next input character

```
switch (ch)
{
    ...
    case '!':                              // ! or != or !==
        if (get() == '=')
        {
            if (get() == '=')
                tokCode = tkEQUAL_EQUAL;
            else
                tokCode = tkNOT_EQUAL;
        }
        else
            tokCode = tkEXCLAMATION;
        break;
    case '%':                               // % or %=
        if (get() == '=')
            tokCode = tkPERCENT_EQUAL;
        else
            tokCode = tkPERCENT;
        break;
    ...
```

Scanner implementation (2)

Taken from the sources of the **GNAT Ada compiler**

```
...
elsif Slen = 3 then
    C1 := Source (Token_Ptr + 1);
    C2 := Source (Token_Ptr + 2);
    C3 := Source (Token_Ptr + 3);
    if (C1 = 'A' or else C1 = 'a') and then  -- AND
       (C2 = 'N' or else C2 = 'n') and then
       (C3 = 'D' or else C3 = 'd')
    then
       Token_Name := Name_Op_And;
    elsif (C1 = 'A' or else C1 = 'a') and then -- ABS
          (C2 = 'B' or else C2 = 'b') and then
          (C3 = 'S' or else C3 = 's')
    then
       Token_Name := Name_Op_Abs;
...
```

Scanner implementation (3)

How to distinguish identifier from a keyword?

- **Directly:**

```
char buffer[maxLen];
...
if      (strcmp(buffer,"switch")==0) return tokSwitch;
else if (strcmp(buffer,"while")==0) return tokWhile;
else if (strcmp(buffer,"int")==0) return tokInt;
...
else return tokIdentifier;
```

Scanner implementation (4)

How to distinguish identifier from a keyword?

- A bit smarter: using hash functions

```
int hash(char* keyword)
{
    // Maps the set of keywords of the given language
    // to the set of integers in the range 0..N-1,
    // where N – the common amount of keywords.
}
int Table[N] =
  { tokIdentifier, tokSwitch, tokWhile, tokInt, ... };

char buffer[maxLen]; // the buffer with the id/keyword
...
return Table[hash(buffer)];
```

Scanner implementation (5)

How to distinguish identifier from a keyword?

- **High-level solution - in case implementation language supports the following:**

```
// A sequence of letters/digits from the input
string WordFromInput;
...
switch ( WordFromInput ) {
    case "switch": return tokSwitch;
    case "while": return tokWhile;
    case "int": return tokInt;
    ...
    else return tokIdentifier;
}
...
```

Hashing is under the hood!

Token implementation

- Simple case:
  each token is encoded by an integer value.

- What about identifiers and literals?

- A structure with attributes:

  - Token code (pattern)
  - Token source coordinates ("span")
  - Token category (optional)
  - Binary representation (for literals)
  - Token source image (for identifiers)

Token implementation

"Usual" implementation

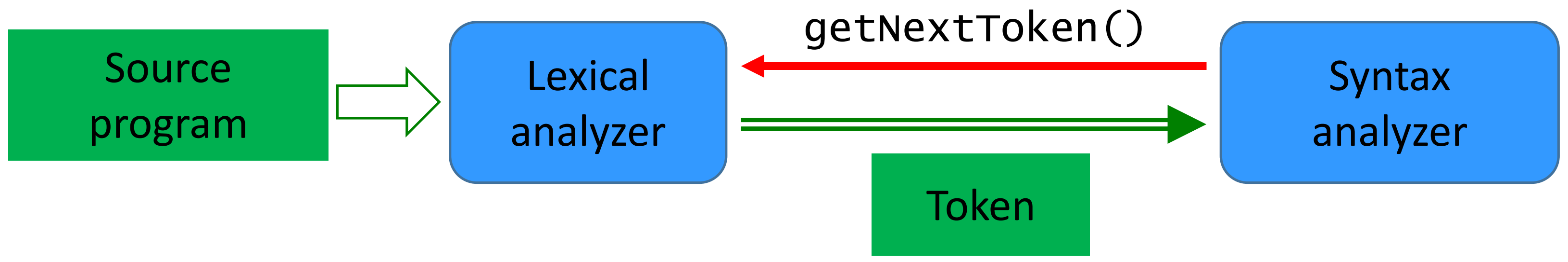
```
struct Token {
    TokenCode code;     // Token category
    Span span;          // Token position
    union {
        double dValue;  // Bin.repr. (for reals)
        long iValue;    // Bin.repr. (for integers)
    };
    char* image;        // Token source image
    ...
```

```
abstract class Token {
    Span span;   // Token position
    char* image; // Token source image
    ...
}
class Keyword : Token {
    int kwdCode; // Keyword code
}
class Integer : Token {
    long value;  // Binary form
}
```

A smarter ("OOP") implementation

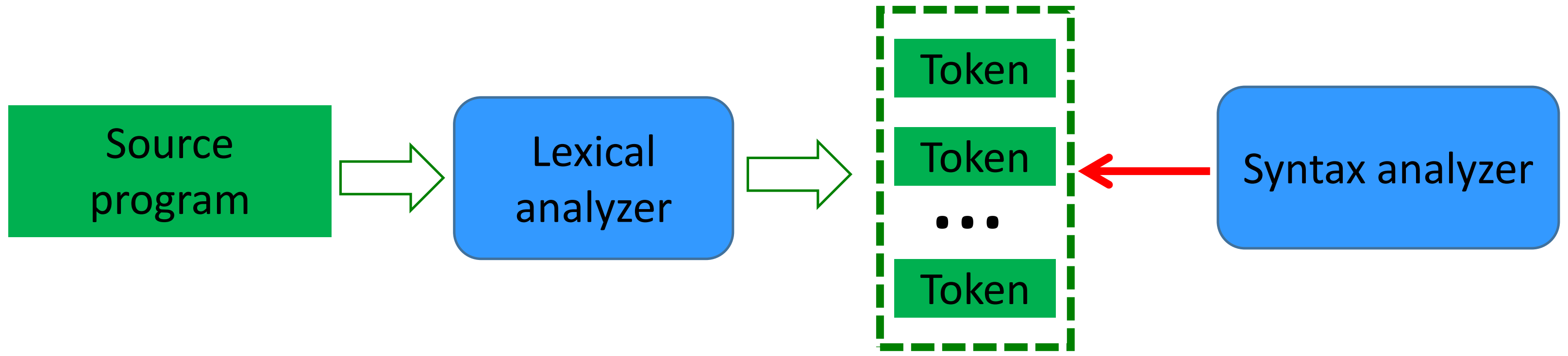
Lexical Analysis: Interaction with Parser

1. Scanner & parser: getting token on demand



For a language with the simple syntax rules, where lookahead is not necessary (that is, we do not need to look at the next token to detect the current one).

2. Scanner & parser: two independent components



- **Pros**: More flexible alternative; more convenient for the parser; allows to process non-trivial language grammars.
- **Cons**: Consumes more memory for storing all program tokens.

Why Use a Token Buffer?

Scanner & parser: getting token on demand

- **Ambiguity example: the Ada language.**

```
A(0..10)
```

- A function call?
- An access to an array element?
- A sub-array?

Sub-array! ("Slice")

- **Another example (also Ada ☺):**
  Token «apostroph» (single quote) is used in two meanings: either for attributes, or for character constants:

```
x := A'Range;
y := 'L';
```

- Lookahead?
- Keep previous tokens?
- Anything else?