

Compiler Construction: Practical Introduction

Lecture 2 Lexical Analysis

Eugene Zouev
Spring Semester 2022
Innopolis University

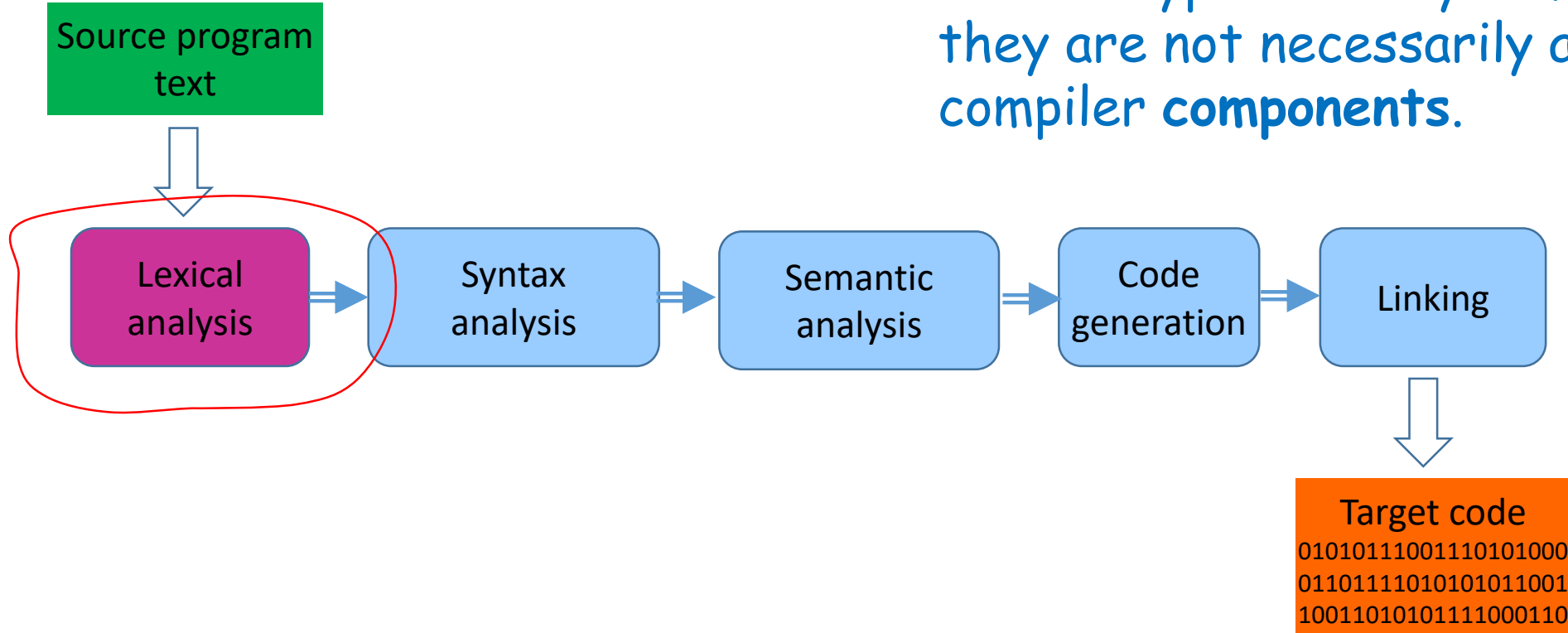
Lexical Analysis: the Outline

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

Compilation: An Ideal Picture

From the previous lecture

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



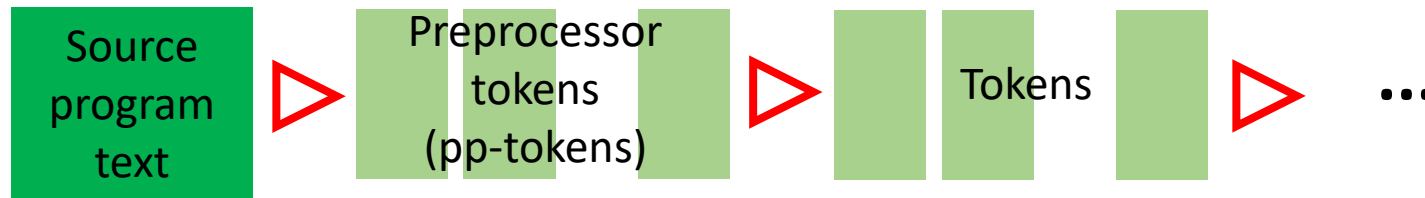
Blue squares just denote some actions typical to any compiler; they are not necessarily actual compiler components.

*A program binary image
suitable for immediate
execution by a machine*

Compilation Stages: Lexical Analysis

From the previous
lecture

Components: preprocessor, scanner



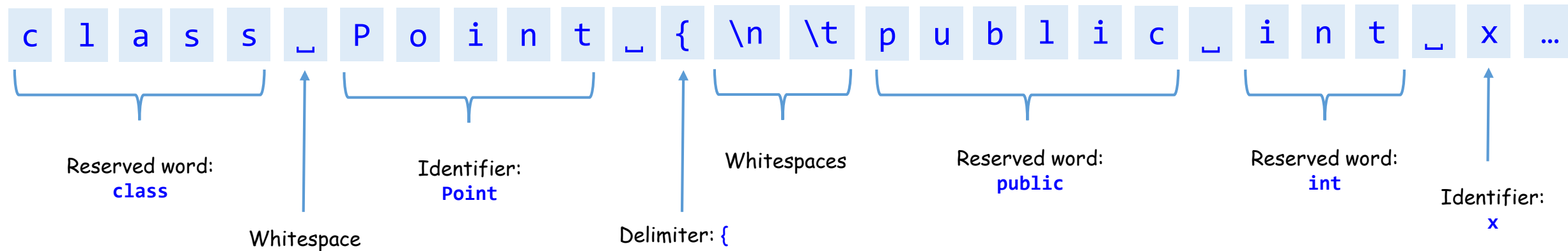
- **Token** is the minimal element of the language alphabet. Examples are operator signs, delimiters, identifiers, keywords, literals.
- **Token representations** in the compiler can be either very simple (e.g., coded by integer values) or be a structure with a set of attributes.
- How lexical analysis **interacts** with other compiler components: either passing the current token "on demand" or buffering tokens and traversing those buffers (with returns) - see next lecture(s).

Lexical Analysis: the Overall View

Source program:
written by a developer and saved in a **disk file**

```
class Point{  
    public int x, y;  
}
```

Source program text from a compiler's point of view:
a **sequence of characters**



Lexical analyzer scans **character by character** trying to **identify** a subsequence of characters as a **minimal unit ("token")** that has a special meaning in the language

Lexical Analysis: the Overall View

Source program:
written by a developer
and saved in a **disk file**

```
class Point{  
    public int x, y;  
}
```

**Lexical
analyzer**



Sequence of **tokens**

Reserved word **class**

Whitespace **_**

Identifier **Point**

Delimiter **{**

Newline **\n**

Hor. tab **\t**

Reserved word **public**

Reserved word **int**

...

End-of-file EOF

Lexical Analysis: Preprocessing

file.c

```
int C=0;
```

```
#include "file.c"
#define A ++a
int x=A;
```

Sequence of **pp-tokens**

Include directive

String `file.c`

Newline `\n`

Define directive

Identifier `A`

Operator sign `++`

Identifier `a`

Newline `\n`

Reserved word `int`

Whitespace `_`

Identifier `x`

Delimiter `=`

Identifier `A`

Delimiter `;`

...

The contents of `file.c` gets scanned and **included** into the current token sequence

The **macro definition** (the name and its token sequence) for `A` is stored by the preprocessor...

This is the **use** of the macro: its token sequence gets passed to the current sequence

Sequence of tokens

Reserved word `int`

Whitespace `_`

Identifier `C`

Delimiter `=`

Integer literal `0`

Delimiter `;`

Reserved word `int`

Whitespace `_`

Identifier `x`

Delimiter `=`

Operator sign `++`

Identifier `a`

Delimiter `;`

...

Tokens & Lexemes: Terminology

From the "Dragon Book"

Token Lexeme's category	Pattern Generalized category description	Lexeme A concrete text snippet, that falls under a certain category
Keyword if	Joint sequence of characters i and f , with neither letter nor digit after it.	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

Token categories: Identifiers & Keywords (1)

- The category depends 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
```

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

Java: 51+ keywords

abstract, assert, boolean, break, byte, ...

C++: 81 keywords

alignas, alignof, asm, auto, bool, break, ...

Token categories: Identifiers & Keywords (2)

- Keywords are explicitly marked (by leading underscore or by quotes): Algol-60, Algol-68, Эль-76

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

```
'if' a > b  
'then' 'begin'  
    a := b;  
    b := 0  
'end'
```

- Keywords & identifiers are lexically identical.

```
if ( alpha != beta ) alpha = 0;  
else gamma = 777;
```

Token categories: Identifiers & Keywords (3)

- The meaning of the token depends on the context

```
class C {  
    public int p { get; }  
    public f() {  
        int get;  
    }  
}
```

C#

The first **get** is the keyword,
whereas the second one is a usual
local variable...

Token categories: Spaces (1)

Spaces (blanks, whitespaces)

- Spaces are treated non-meaningful everywhere in the program (*and* inside identifiers)

```
This is the valid identifier in some langs := 777;
```

Fortran: dramatic error:

```
DO 5 I = 1.25
```

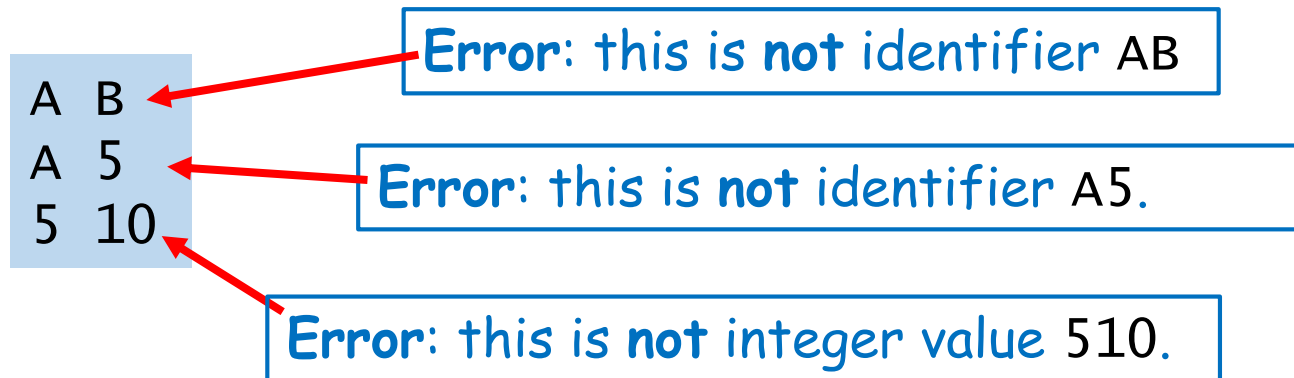
```
DO 5 I = 1,25
```

This is the **assignment**: a real value of 1.25 is assigned to the variable D05I (spaces are not considered).

This is the **loop header**! Loop variable I sequentially gets values from the range 1..25. (The end of the loop is marked by the label 5.)

Token categories: Spaces (2)

- Spaces always separate tokens and are never a part of any token (except strings).
- Two adjacent identifiers, or a constant following the identifier, or two adjacent constants - all are treated as lexical errors.



- Tricky question 😊: how to interpret C++/Java constructs like `C c;` ?

Token categories: Comments (1)

- Typically, comments are treated as *whitespaces*; they do not alter the program semantics => they are dropped by the lexical analyzer.
- Short & long comments:

```
// This is the short (one-line) comment
```

```
/* This is the “long”,  
   or multi-line  
   comment */
```

- Comment can OR cannot nest:

```
/* Some languages  
   do /* not */ allow  
   nesting comments  
*/
```

Token categories: Comments (2)

- *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 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:

$A \rightarrow Ba$ or $A \rightarrow a$

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.

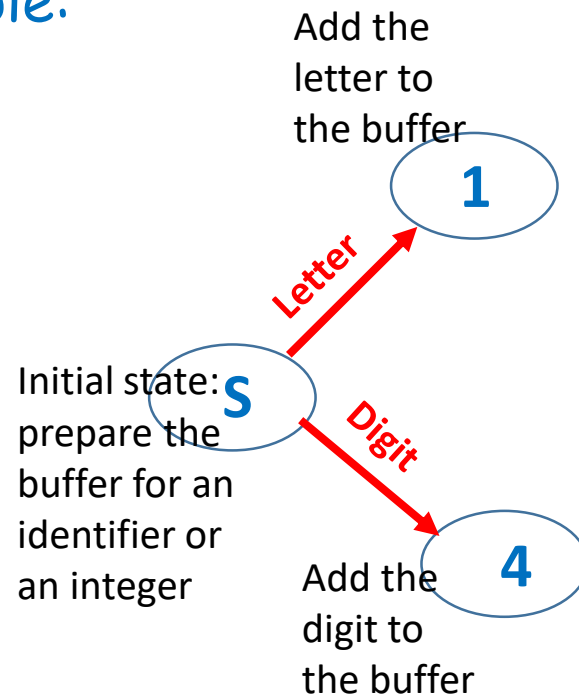
Example:

Initial state:  **S**
prepare the
buffer for an
identifier or
an integer

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**.

Example:

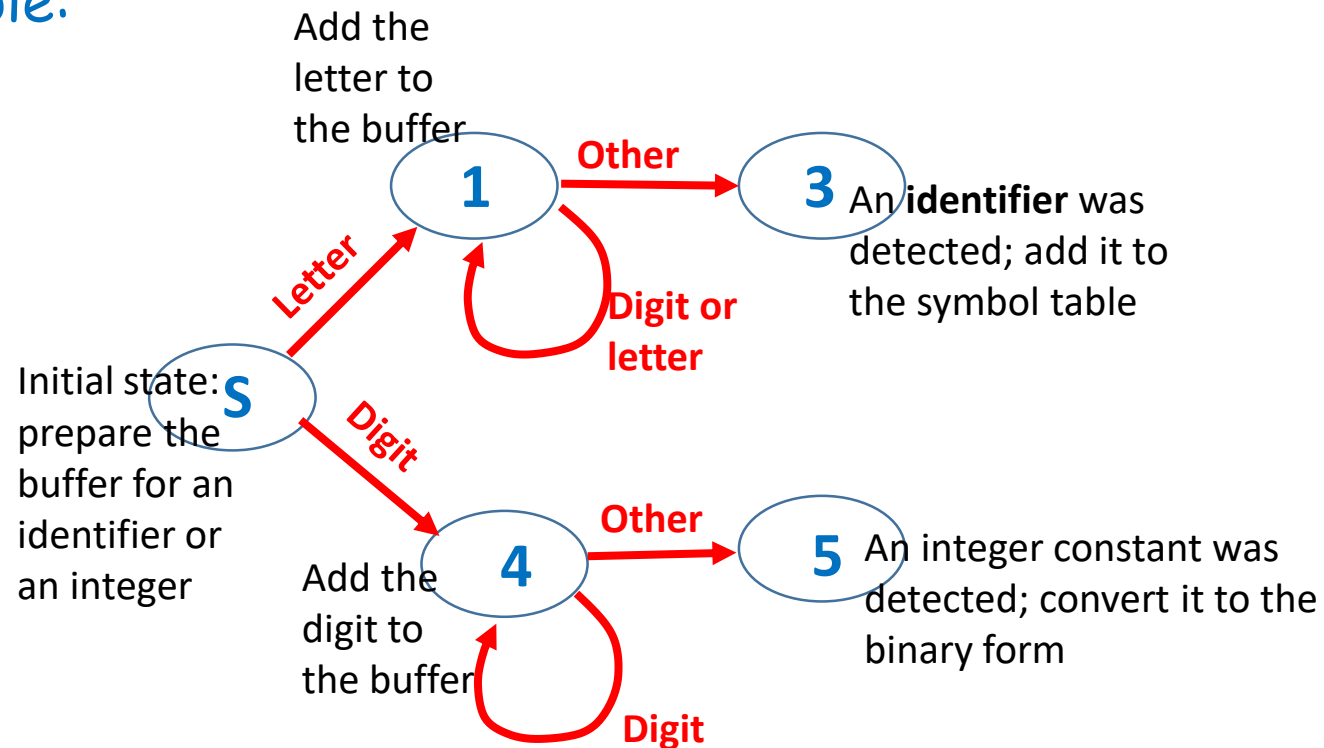


Formal Basics (3)

- **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**.

Example:



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**

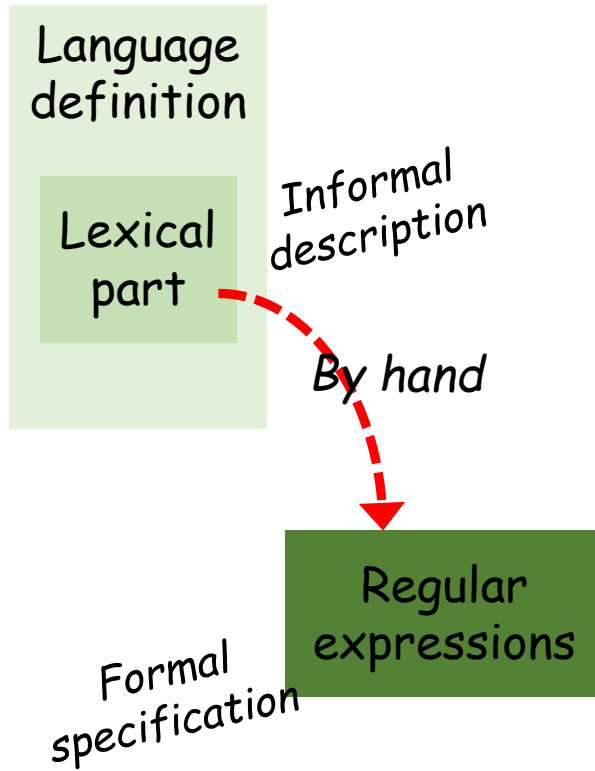
M. E. Lesk and E. Schmidt

<http://dinosaur.compilertools.net/lex/index.html>

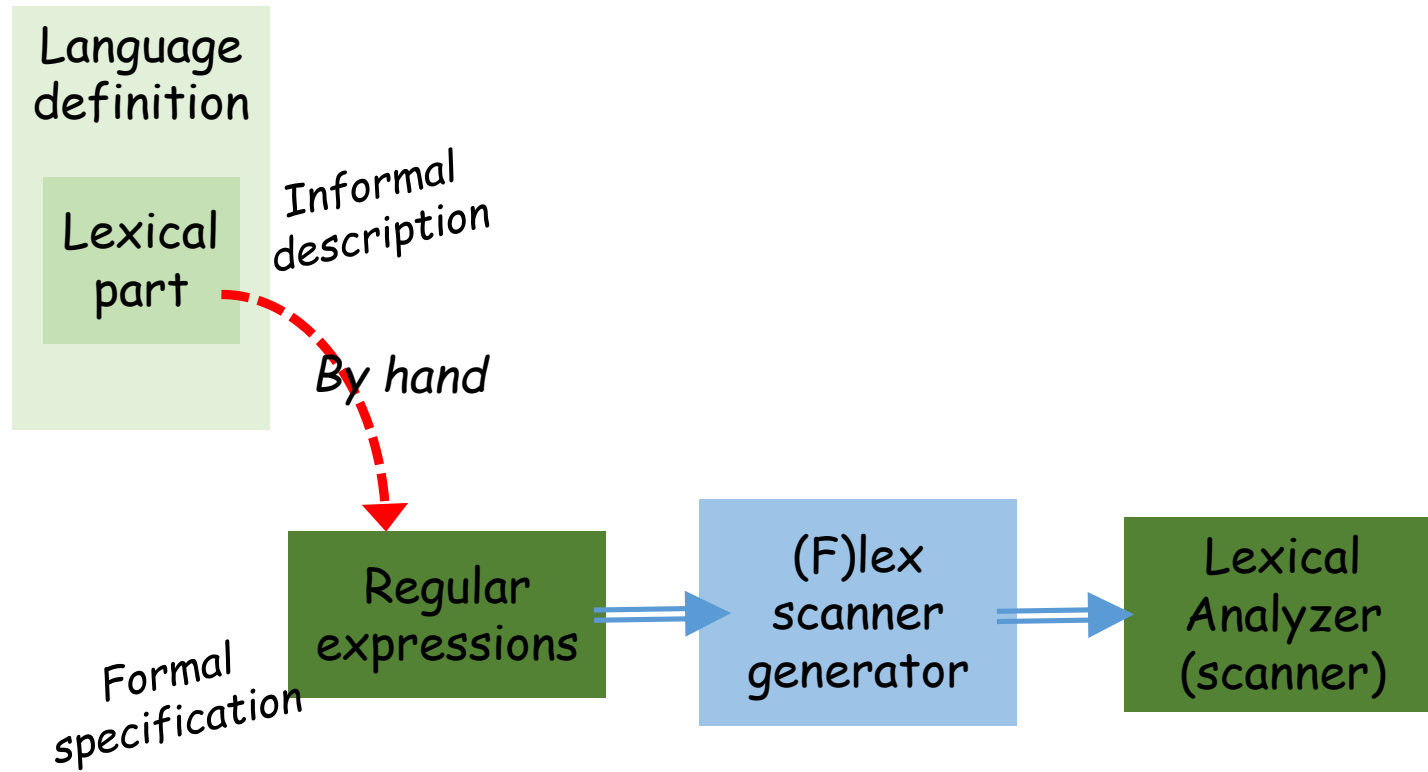
- Typically, lex/flex is used together with the parser generator **yacc/bison**.

Versions of lex &
yacc are available
for many languages

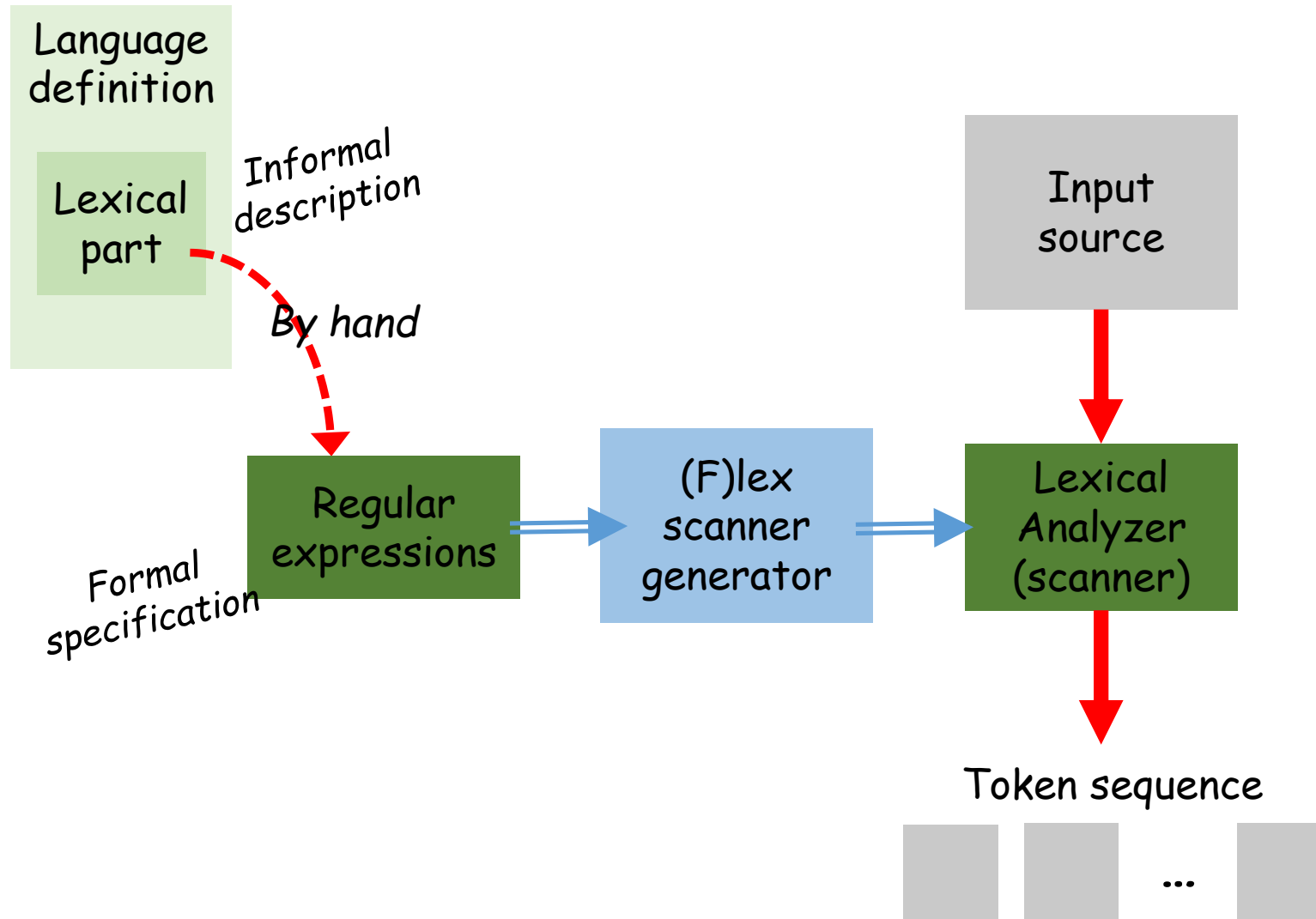
(F)lex Scanner Generator



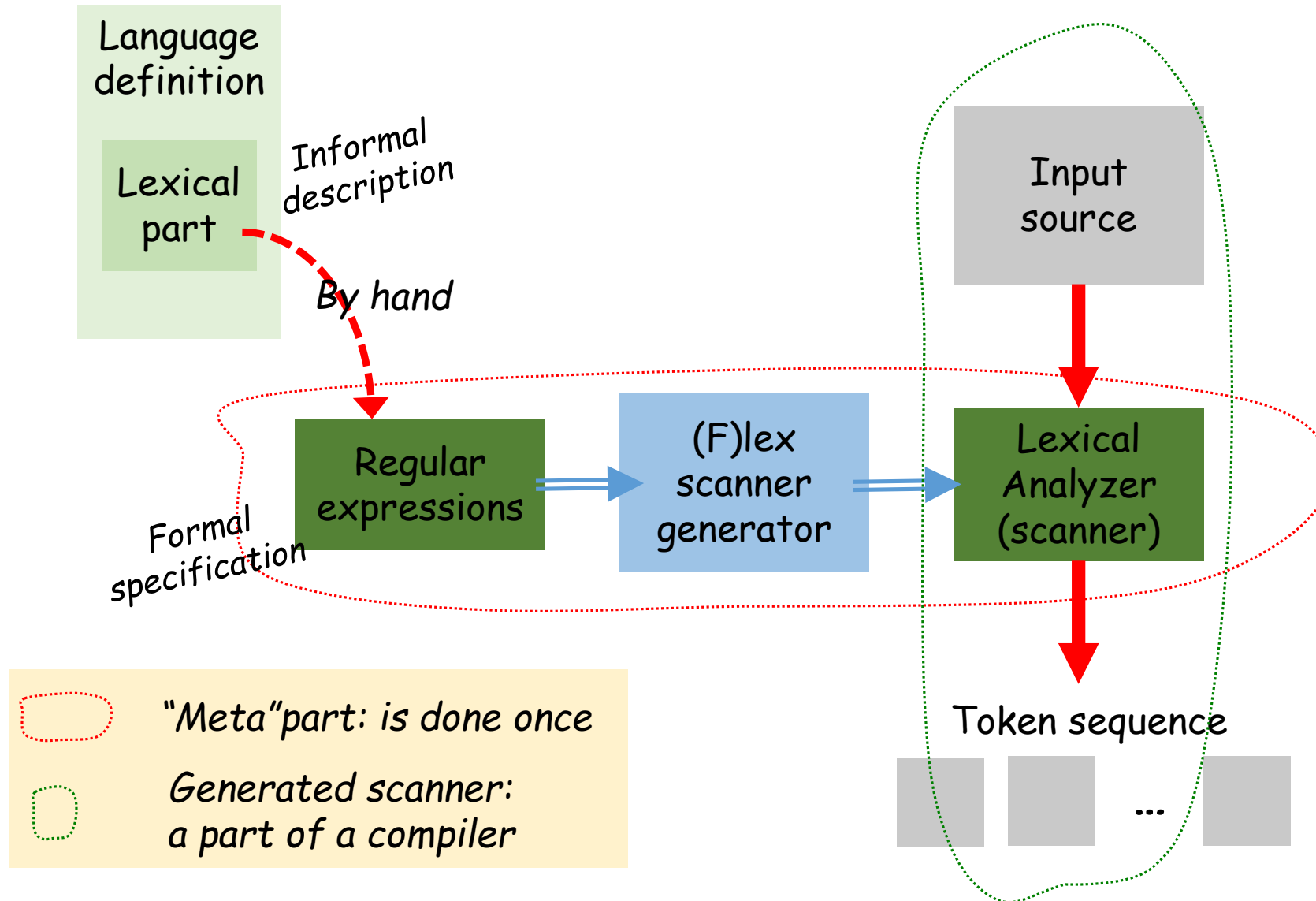
(F)lex Scanner Generator



(F)lex Scanner Generator



(F)lex Scanner Generator



Scanner implementation (1)

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

The `get()` function returns the next character from the input

Scanner implementation (1)

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

The `get()` function returns the next character from the input

This is the way I usually recommend for implementing scanners in student projects

Scanner implementation (2)

```
...
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;
...

```

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

Scanner implementation (3)

How to distinguish identifier from a keyword?

- Directly:

```
// Token codes
enum Code {
    ...
    tokInt,
    tokFloat,
    tokSwitch,
    tokWhile,
    ...
}
```

```
// Suppose letters composing identifier are here
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

Hash function maps an unlimited set of strings (e.g., identifiers) to a set of integer values within a range $[1..N]$, where N is the number of keywords so that for the given set of strings (representing keywords) the function returns different values.

```
int hash(char* keyword)
{
    // Maps the set of keywords of the given
    // language to the set of integers in the
    // range 1..nKW, where nKW - the common
    // amount of keywords.
}

int Table[nKW] =
{ tokIdentifier, tokSwitch,
  tokWhile, tokInt, ... };

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

Hash Function: An Example

```
uint HashFunction ( string identifier )
{
    const uint hash_module = 511; // or 211: a simple number
    uint g; // for calculating hash
    const uint hash_mask = 0xF0000000;

    uint hash_value = 0;

    for ( int i=0; i<identifier.Length; i++ )
    {
        // calculating hash: see Dragon Book, Fig. 7.35
        hash_value = (hash_value << 4) + (byte)identifier[i];
        if ( (g = hash_value & hash_mask) != 0 )
        {
            // hash_value ^= g >> 24 ^ g;
            hash_value = hash_value ^ (hash_value >> 24);
            hash_value ^= g;
        }
    }
    // final hash value for identifier
    return hash_value % hash_module;
}
```

Scanner implementation (5)

How to distinguish identifier from a keyword?

- High-level solution - in case the 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;
}
...
```

ML does support this
(pattern matching)

Hashing is under
the hood!

Token Representation (1)

- Simple cases:
each token is encoded by an integer value.
- What about identifiers and literals?

Token Representation (1)

- Simple cases:
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 Representation (1)

- Simple cases:
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)

```
struct Token {  
    Span span;        // for all tokens  
    unsigned int code; // token code  
    int intValue;     // for integer constants  
    long realValue;   // for real constants  
    string id;        // for identifiers  
    ...  
};
```

```
struct Span {  
    long lineNum;  
    int posBegin, posEnd;  
};
```

Token Representation (2)

- Advanced approach: OO architecture

```
// Common notion of token
class Token {
    // Attributes common
    // to all token categories
    Span span;
    unsigned int code;
};
```

```
class Identifier : Token {
    // Attributes specific for identifiers
    string identifier;
}
```

```
class Integer : Token {
    // Attributes specific for integer constants
    long value;
}
```

```
class Real : Token {
    // Attributes specific for real constants
    long double value;
}
```

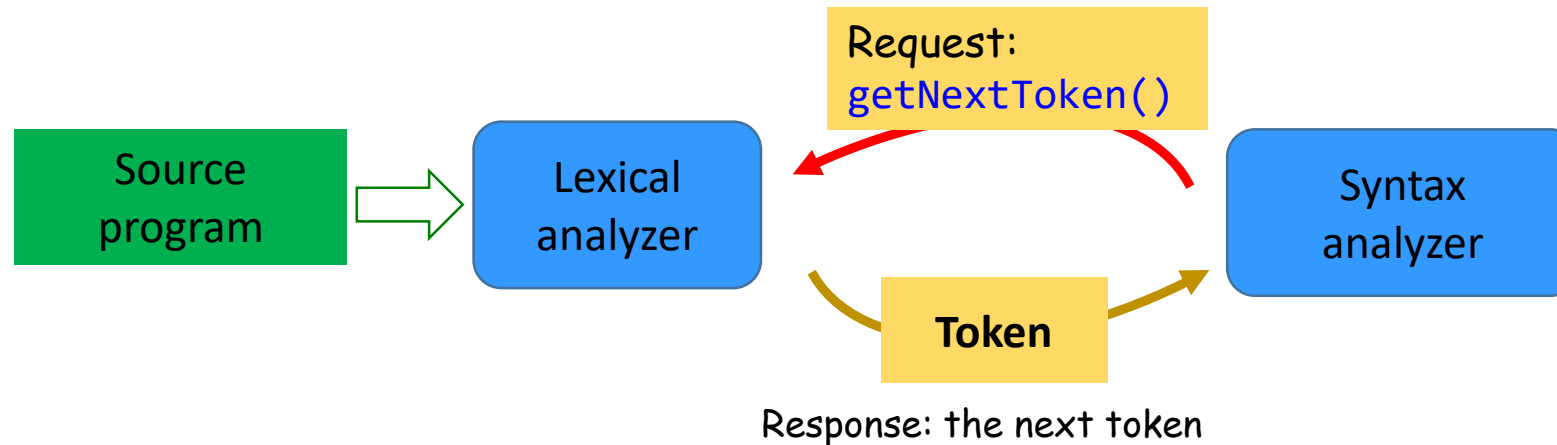
Lexical Analysis: Two Aspects

How lexical analyzer interacts with other compiler components?

- Two "extreme" cases, and...
- Many mixed cases.

Extreme Case (1)

Scanner & parser: getting token on demand



- For 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).

Extreme Case (2)

Scanner & parser: getting token on demand

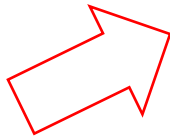
- Ambiguity example: the Ada language.
A(0..10)

Extreme Case (2)

Scanner & parser: getting token on demand

- Ambiguity example: the Ada language.

A(0..10)



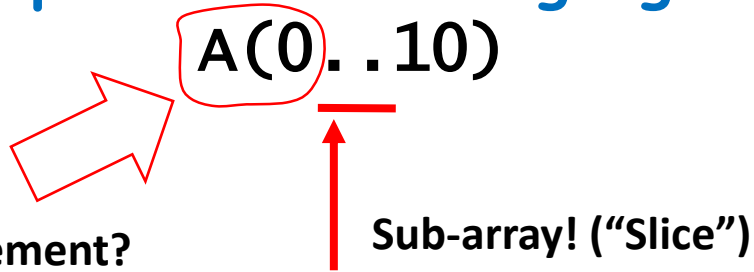
- A function call?
- Access to an array element?
- A sub-array?

Extreme Case (2)

Scanner & parser: getting token on demand

- Ambiguity example: the Ada language.

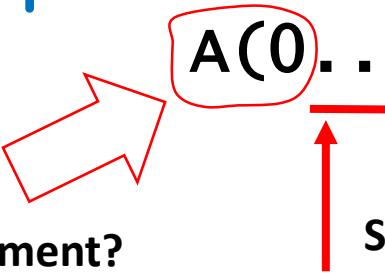
- A function call?
- Access to an array element?
- A sub-array?



Extreme Case (2)

Scanner & parser: getting token on demand

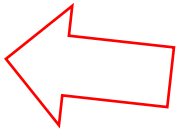
- Ambiguity example: the Ada language.


A(0..10)

- A function call?
- Access to an array element?
- A sub-array?

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

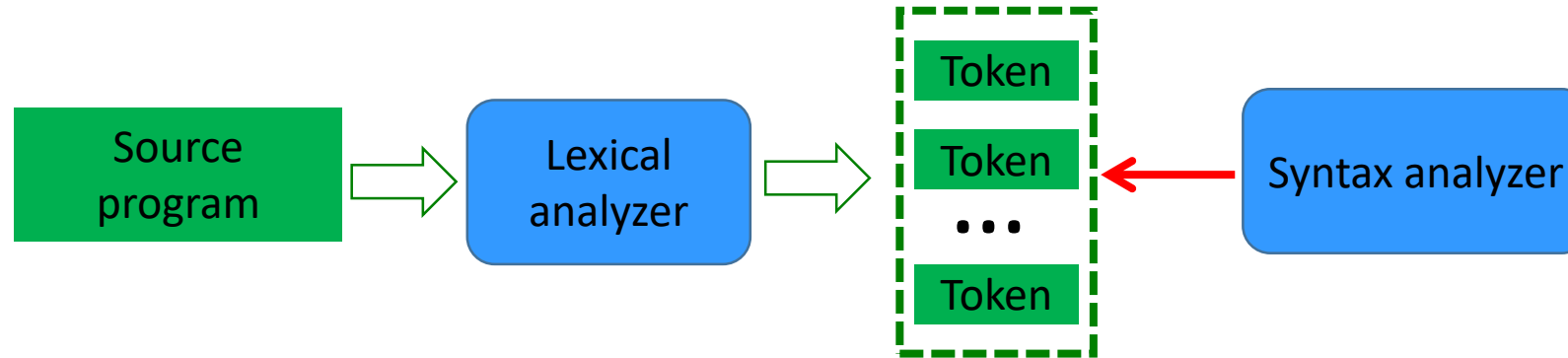
x := A'Range;
y := 'L';



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

Another Extreme Case

- 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.

Mixed Approach (1)

Example:

```
int a = 0;
class C {
public:
    void f() { a = 7; }
    int a;
};
int main() {
    C c;
    c.f();
    cout << a; // what is output: 0 or 7?
}
```

Mixed Approach (1)

Example:

```
int a = 0;
class C {
public:
    void f() { a = 7; }
    int a;
};
int main() {
    C c;
    c.f();
    cout << a; // what is output: 0 or 7?
}
```

C++ Standard:

- Function member bodies are processed in the full class context.
- Class declaration is full when the final “}” token is achieved

Mixed Approach (1)

Example:

```
int a = 0;
class C {
public:
    void f() { a = 7; }
    int a;
};
int main() {
    C c;
    c.f();
    cout << a; // what is output: 0 or 7?
}
```

C++ Standard:

- Function member bodies are processed in the full class context.
- Class declaration is full when the final “}” token is achieved

Mixed Approach (2)

Example: C++

```
int a = 0;
class C {
public:
    void f() { a = 7; }
    int a;
} * ;
int main() {
    C c;
    c.f();
    cout << a; // what is output: 0 or 7?
}
```

Conclusions for compiler developers:

- Member function body should be processed only after completing processing all class members.
- Tokens comprising function bodies should be kept until the final “}” is achieved and should be compiled after it.

Mixed Approach (3)

Example: Java

```
class oneClass
{
    secondClass a;
    ...
    int x = a.m;
}

...
class secondClass
{
    ...
}
```


Mixed Approach (3)

Example: Java

```
class oneClass
{
    secondClass a;
    ...
    int x = a.m;
}

...
class secondClass
{
    ...
}
```

The problem:

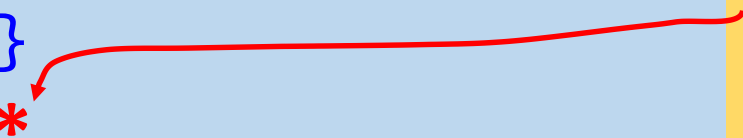
- Syntax/semantic analyzers cannot determine the type of `a`.
- Syntax/semantic analyzers cannot conclude whether `a.m` is valid or not.

Mixed Approach (3)

Example: Java

```
class oneClass
{
    secondClass a;
    ...
    int x = a.m;
}

...
class secondClass
{
    ...
}
*
```



The problem:

- Syntax/semantic analyzers cannot determine the type of `a`.
- Syntax/semantic analyzers cannot conclude whether `a.m` is valid or not.

Conclusion for compiler developers:

- All cases with using `a` should be processed after `secondClass` declaration is processed.
- This means that tokens comprising class bodies should be kept until all class names are recognized. Only then token sequences of class bodies should be syntactically analyzed.