System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block C Compiler Construction
2. Lexical Analysis

Eugene Zouev

Outline

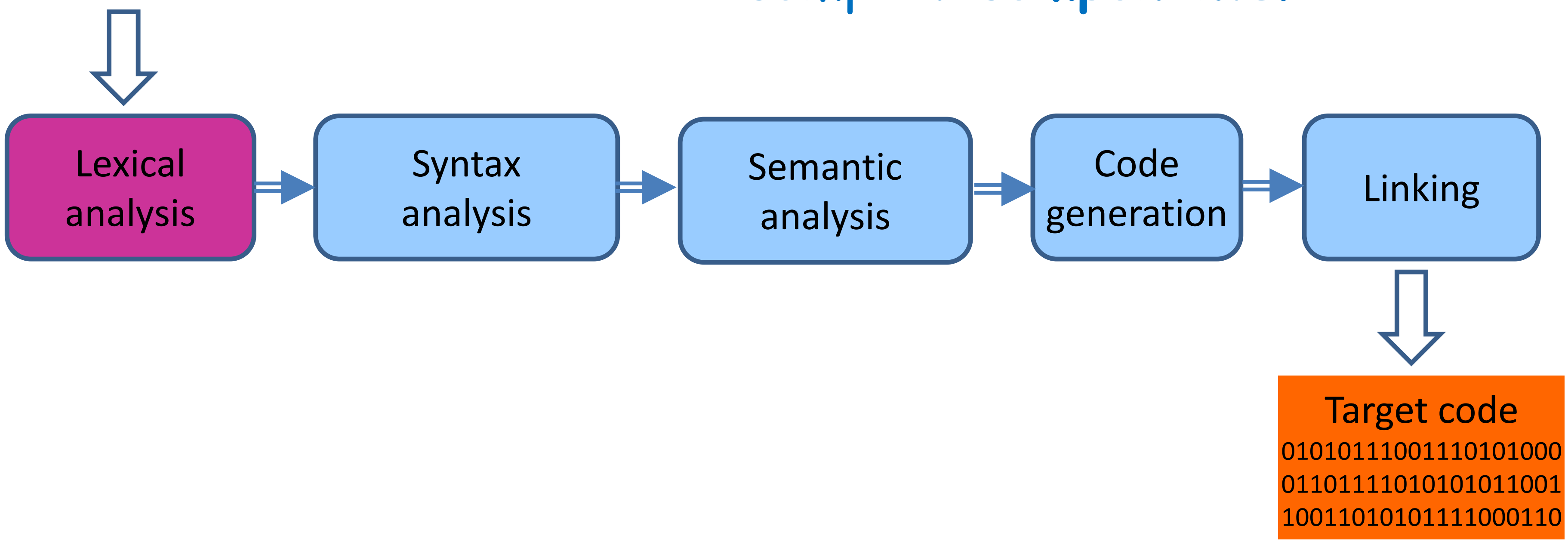
- Lexical analysis: why & what for?
- The notion of token, its meaning and implementation
- Formal basis
- Scanners: implementation techniques; scanner generation tools
- Non-standard issues
- Scanner & parser integration: the architecture

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**.

Source program text



A program binary image suitable for immediate execution by a machine

Lexical Analysis: Two aspects

- Internal structure of the lexical analyzer
- Interaction between lexical analyzer & other compiler components

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

Token categories (1)

Keywords & identifiers

- 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.

Token categories (2)

Keywords & identifiers

- 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;
```

- Keywords & identifiers are lexically identical.

Token categories (3)

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 `DO5I` (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 (4)

Spaces

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

```
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;`** ?

Token categories (4)

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 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"]
  ```

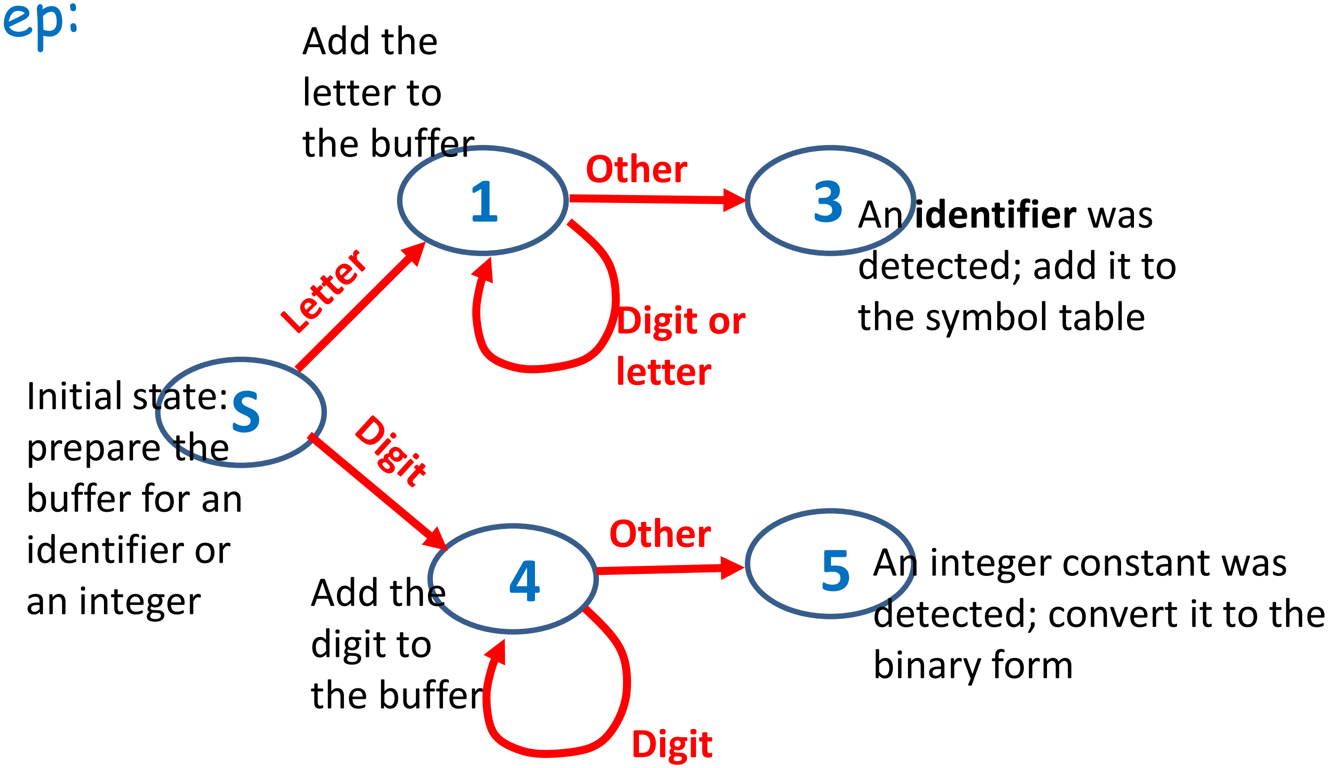
- The main statement concerning regular grammars: to scan a token successfully (and therefore, to determine that a token belongs to the given grammar) it's enough to know the **current** scanner **state** and **only one** input character.

- 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:



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**.

Scanner implementation (1)

The get() function returns next input character

```
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;
    ...
```

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;
...
```

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 1..nKW,
    // where nKW – the common amount of keywords.
}
int Table[nKW] =
  { tokIdentifier, tokSwitch, tokWhile, tokInt, ... };

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

Token implementation

- 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)

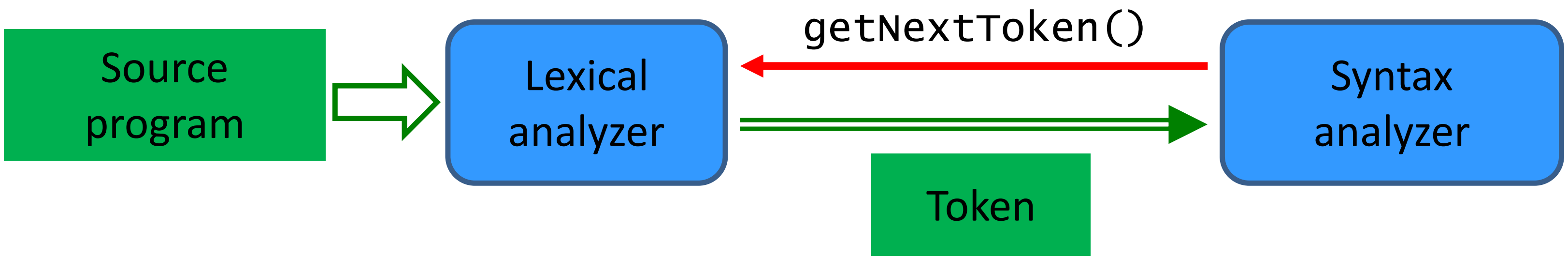
Lexical Analysis: Two Aspects

- Internal structure of the lexical analyzer

- **Interaction between lexical analyzer & other compiler components**

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)
```

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

Sub-array! ("Slice")

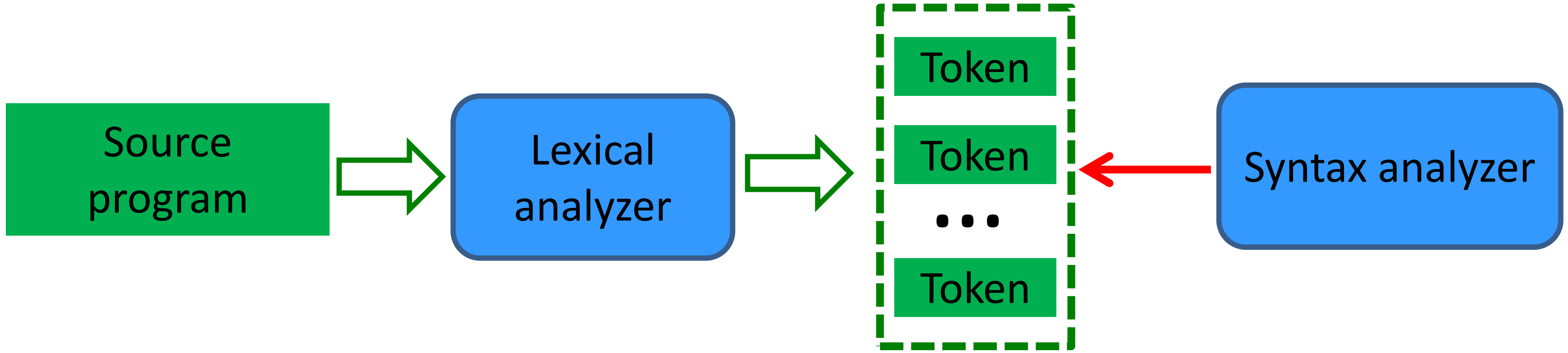
- Another example (also Ada ☺):
  Token «apostroph» (single quote) is used in two meanings: either for attributes, or for character constants:

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

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

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?
}
```

C++ Standard:

- Function member bodies are processed <u>in the full class context.</u>
- Class declaration is <u>full</u> when the final "}" token is achieved

Mixed Approach (2)

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?
}
```

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.