Lexical Analysis

Amitabha Sanyal

(www.cse.iitb.ac.in/~as)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

January 2016

Introduction

The input program – as you see it.

```
main ()
{
    int i,sum;
    sum = 0;
    for (i=1; i<=10; i++)
       sum = sum + i;
    printf("%d\n",sum);
}
```

Introduction

The same program – as the compiler initially sees it. A continuous sequence of characters without any structure

```
main␣()↩{↩␣␣␣␣int␣i,sum;↩␣␣␣␣sum␣=␣0;↩␣␣␣␣for␣(i
=1;␣i<=10;␣i++);␣␣␣␣sum␣=␣sum␣+␣i;↩␣␣␣␣printf("%d\n",
sum);↩}
```

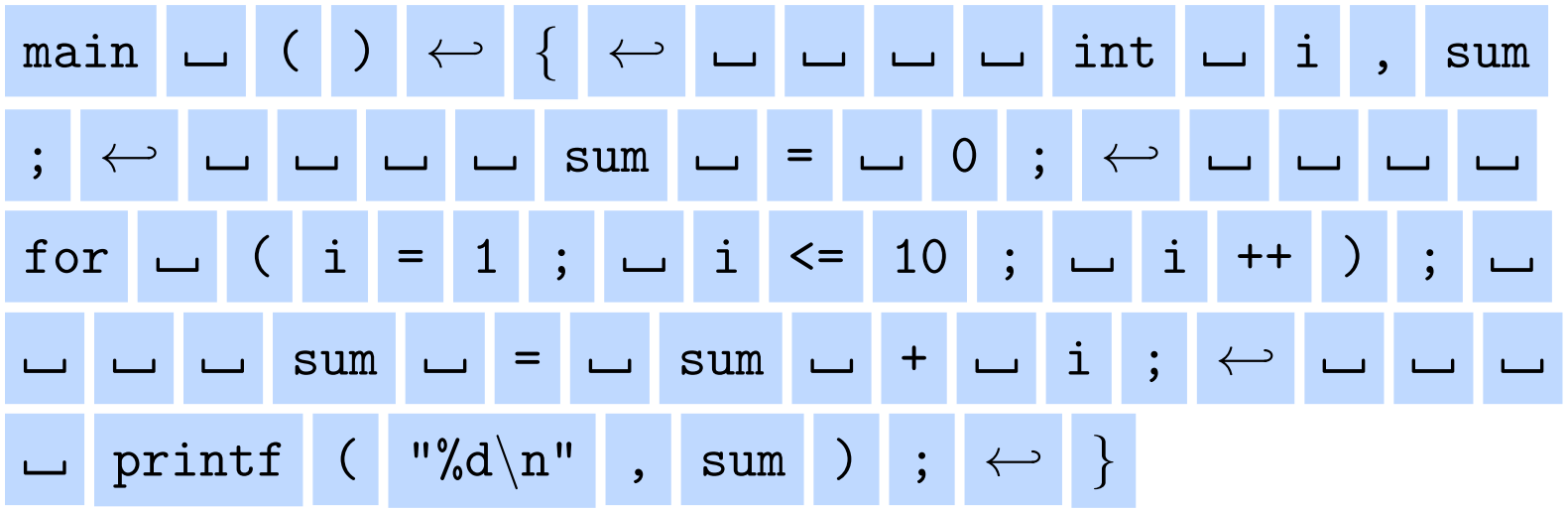
␣	–	The blank space character
↩	–	The return character

How do you make the compiler see what you see?

Discovering the structure of the program

Step 1:

a. Break up this string into the smallest meaningful units.



We get a sequence of *lexemes* or *tokens*.

Discovering the structure of the program

Step 1:

b. During this process, remove the ␣ and the ↩ characters.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```

Steps 1a. and 1b. are interleaved.

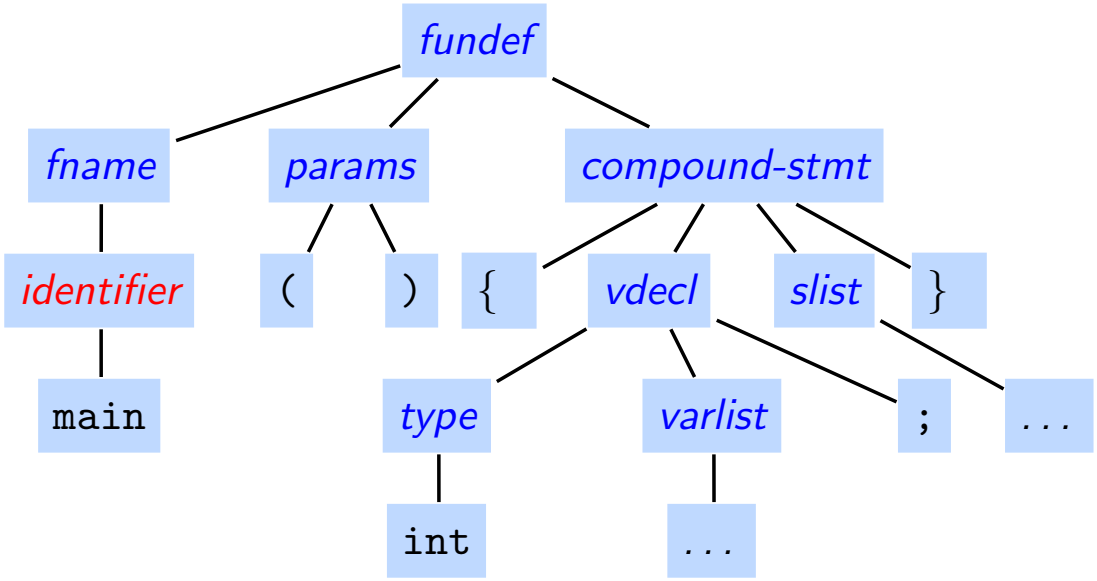
This is *lexical analysis* or *scanning*.

Discovering the structure of the program

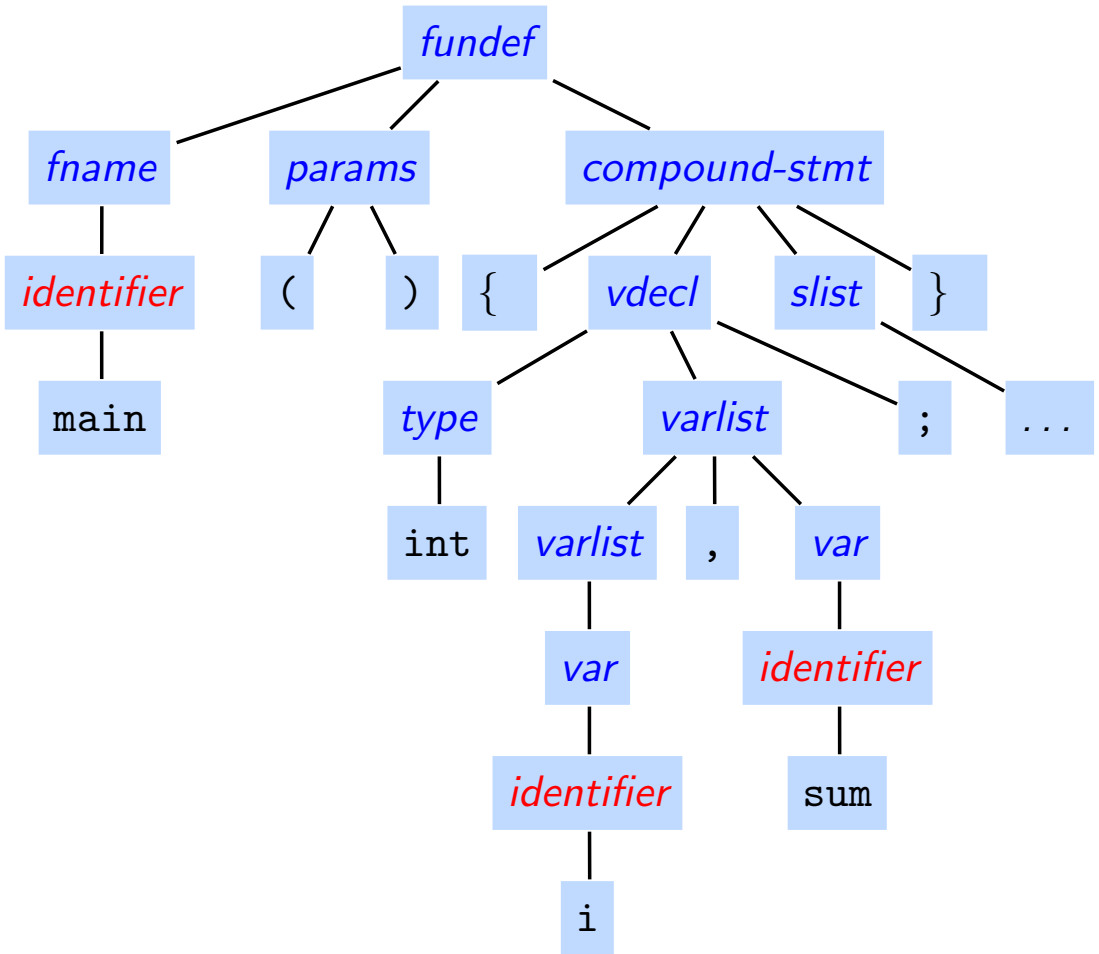
Step 2:

Now group the lexemes to form larger structures.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```



Discovering the structure of the program



This is *syntax analysis* or *parsing*.

Discovering the structure of the program

Why is structure finding done in two steps?

- The process of breaking a program into lexemes (scanning) is easier. Use a separate technique to do this.
- Reduces the work to be done by the parser.

However, there are tools (Antlr for example) that indeed combine scanning with parsing.

Lexemes, Tokens and Patterns

Definition: *Lexical analysis* is the operation of dividing the input program into a sequence of *lexemes* (*tokens*).

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i`, `sum`, `for`, `10`, `++`, `"%d\n"`, `<=`.
- *tokens* – sets of similar lexemes, i.e. lexemes which have a common syntactic description.
  Examples –
  identifier = {`i`, `sum`, `buffer`, ...}
  int_constant = {`1`, `10`, ...}
  addop = {`+`, `-`}

Lexemes, Tokens and Patterns

What is the basis for grouping lexemes into tokens?

- Why can't `addop` and `mulop` be combined? Why can't + be a token by itself?

Lexemes which play similar roles during syntax analysis are grouped into a common token.

- Operators in `addop` and `mulop` have different roles – `mulop` has an higher precedence than `addop`.
- Each keyword plays a different role – is therefore a token by itself.
- Each punctuation symbol and each delimiter is a token by itself.
- All comments are uniformly ignored. They are all grouped under the same token.
- All identifiers are grouped in a common token.

Lexemes, Tokens and Patterns

Lexemes that are not passed to the later stages of a compiler:

- comments
- white spaces – tab, blanks and newlines
    - White spaces are more like separators between lexemes.

These too have to be detected and then ignored.

Lexemes, Tokens and Patterns

Apart from the token itself, the lexical analyser also passes other information regarding the token. These items of information are called *token attributes*

EXAMPLE

lexeme	<**token, token value**>
3	< const, 3>
A	<identifier, A>
if	<if, –>
=	<assignop, –>
>	<relop, >>
;	<semicolon, –>

Lexemes, Tokens and Patterns

The lexical analyser:

- detects the next lexeme
- categorises it into the right token
- passes to the syntax analyser
  - the token name for further syntax analysis
  - the lexeme itself, in some form, for stages beyond syntax analysis

Example – tokens in Java

1. **Identifier:** A *Javaletter* followed by zero or more *Javaletterordigits*. A *Javaletter* includes the characters `a-z`, `A-Z`, `_` and `$`.
2. **Constants:**
   - 2.1 Integer Constants – 4 byte (usual int) and 8 byte (long int ending with an L) representations.
     - Binary – `0b0000011`,
     - Octal – `027` (Note the leading `0`),
     - Hex – `0x0f28`,
     - Decimal – `1`, `-1`
   - 2.2 Floating point constants
     - Float – `1.0345F`, `1.04E-12f`, `.0345f`, `1.04e-13f` – ends with `f` or `F`,
     - Double – `5.6E-120D`, `123.4d`, `0.1` – ends with d or D, or does not end with any of `f`, `F`, `d`, `D`
   - 2.3 Boolean constants – `true` and `false`
   - 2.4 Character constants – `'a'`, `'\u0034'` (Unicode hex), `'\t'`
   - 2.5 String constants – `""`, `"\""`, `"A string"`.
   - 2.6 Null constant – `null`.

Example – tokens in Java

3. **Delimiters:** `(`, `)`, `{`, `}`, `[`, `]` , `;`, `.` and `,`
4. **Operators:** `=`, `>`, `<`, . . . , `>=`
5. **Keywords:** `abstract`, `boolean` . . . `volatile`, `while`.

Lexemes, Tokens and Patterns

How does one describe the lexemes that make up the token *identifier*.

Variants in different languages.

- String of alphanumeric characters. The first character is an alphabet.
- a string of alphanumeric characters in which the first character is an alphabet. It has a length of at most 31.
- a string of alphabet or numeric or underline characters in which the first character is an alphabet or an underline. It has a length of at most 31. Any character after the 31st are ignored.

Such descriptions are called *patterns.* The description may be informal or formal. *Regular expressions* are the most commonly used formal patterns.

Lexemes, Tokens and Patterns

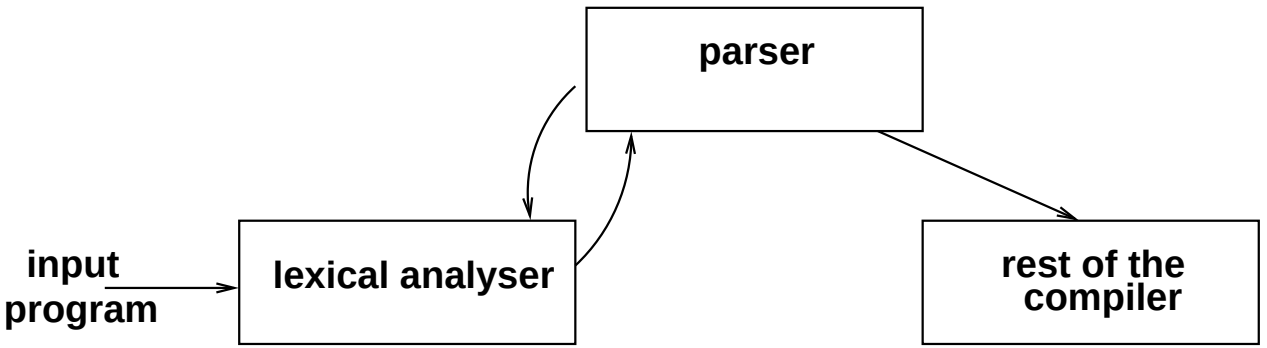
A pattern is used to

- *specify tokens* precisely
- *build a recognizer* from such specifications

Basic concepts and issues

Where does a lexical analyser fit into the rest of the compiler?

- The front end of most compilers is parser driven.
- When the parser needs the next token, it invokes the Lexical Analyser.
- Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token.
- The actions of the lexical analyser and parser are interleaved.



Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
   - Possibly more efficient.
2. *Use a generator* – To generate the lexical analyser from a formal description.
   - The generation process is faster.
   - Less prone to errors.

Automatic Generation of Lexical Analysers

- A formal description (specification) of the tokens of the source language, will consist of:
  - a regular expression describing each token, and
  - a code fragment called an action routine describing the action to be performed, on identifying each token.
- Here is a description of whole numbers and identifiers in form accepted by the lexical analyser generator Lex.

```
{DIGIT}+                          { yylval = atoi(yytext);
                                    return NUM;
                                  }

{LETTER}({LETTER}|{DIGIT})*       { yylval = yytext;
                                    return IDENTIFIER;
                                  }
```

- The global variable `yylval` holds the token attribute (henceforth to be called token value).

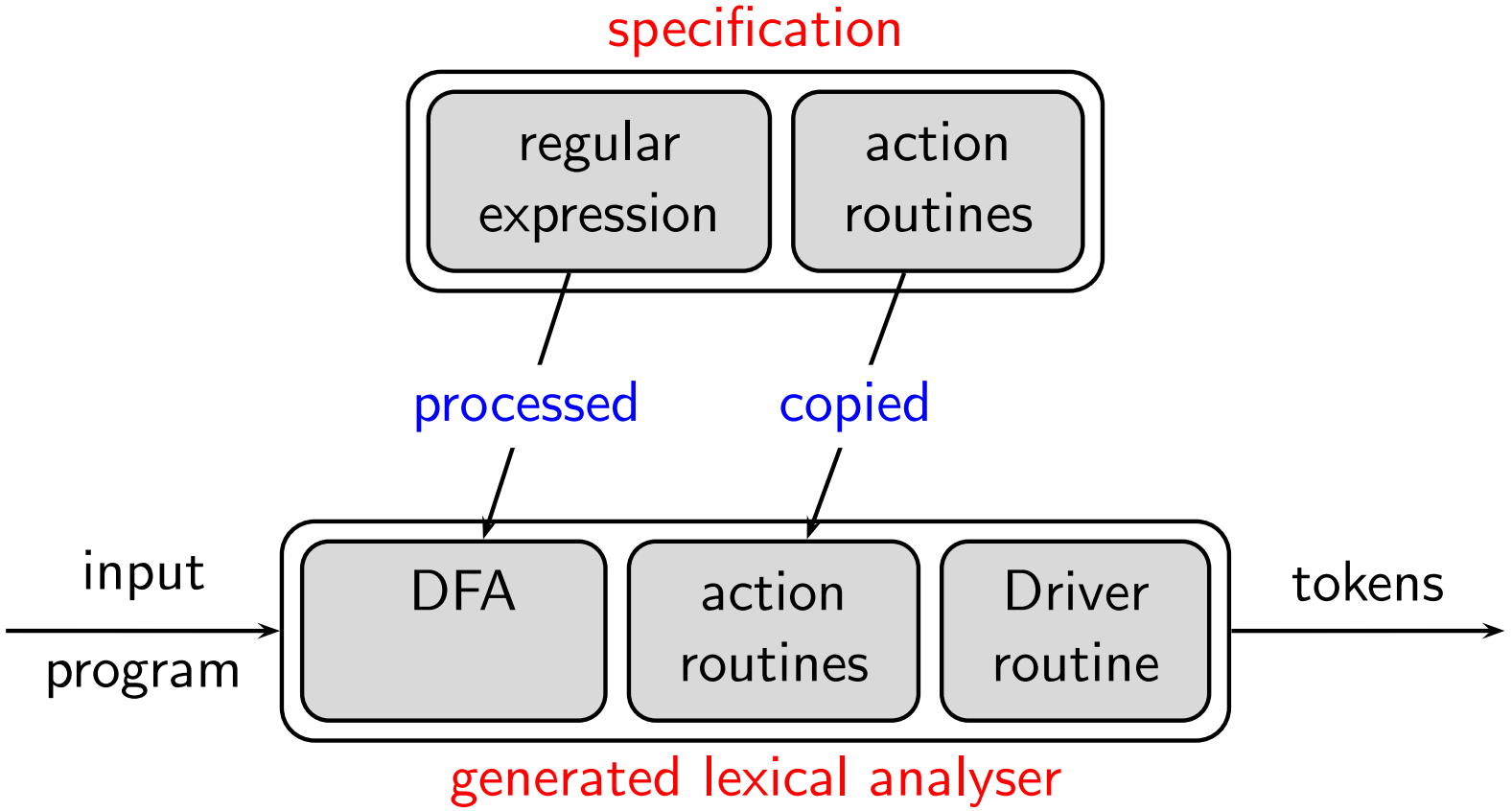
Automatic Generation of Lexical Analysers

Lex can read this description and generate a lexical analyser for whole numbers and identifiers. How?

- The generator puts together:
  - A deterministic finite automaton (DFA) constructed from the token specification.
  - A code fragment called a driver routine which can traverse any DFA.
  - Code for the action routines.
- These three things taken together constitutes the generated lexical analyser.

Automatic Generation of Lexical Analysers

- How is the lexical analyser generated from the description?



- Note that the driver routine is common for all generated lexical analysers.