CS202 – System Software

Dr. Manish Khare

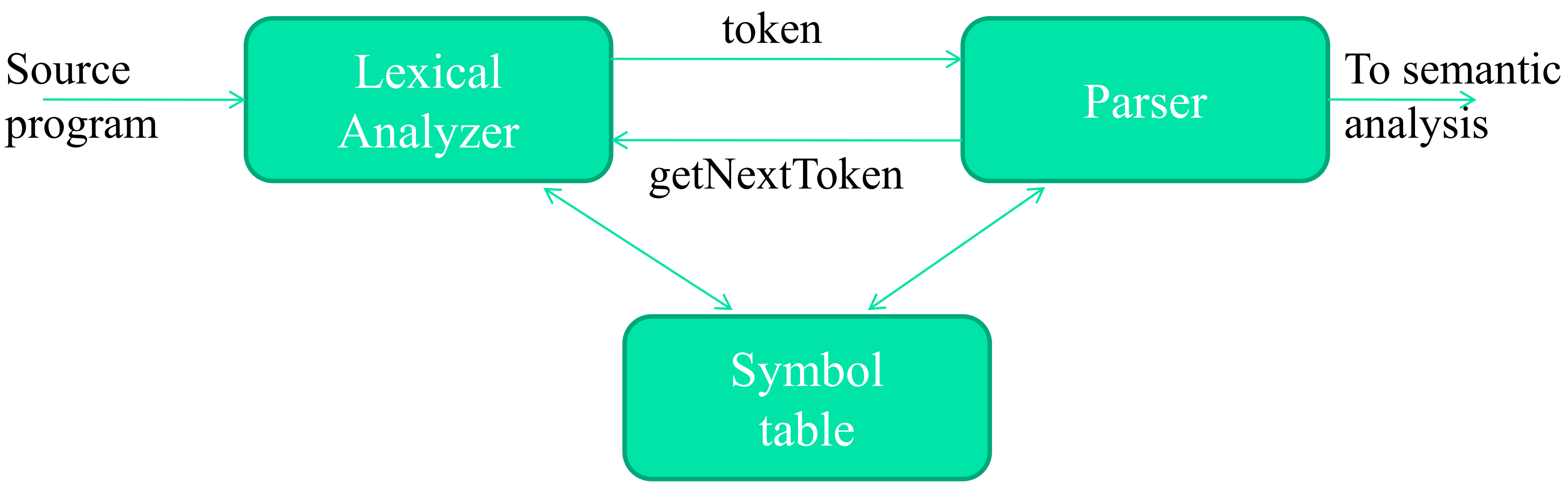
Lecture 5

Lexical analysis

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

Role of lexical analyzer

- The lexical analysis is the first phase of a compiler.
- Its main task is to read the input characters and produces as output a sequence of token that the parser uses for syntax analysis.
- Interaction of lexical analysis is summarized as below



```
while (ip < z)
    ++ip;
```

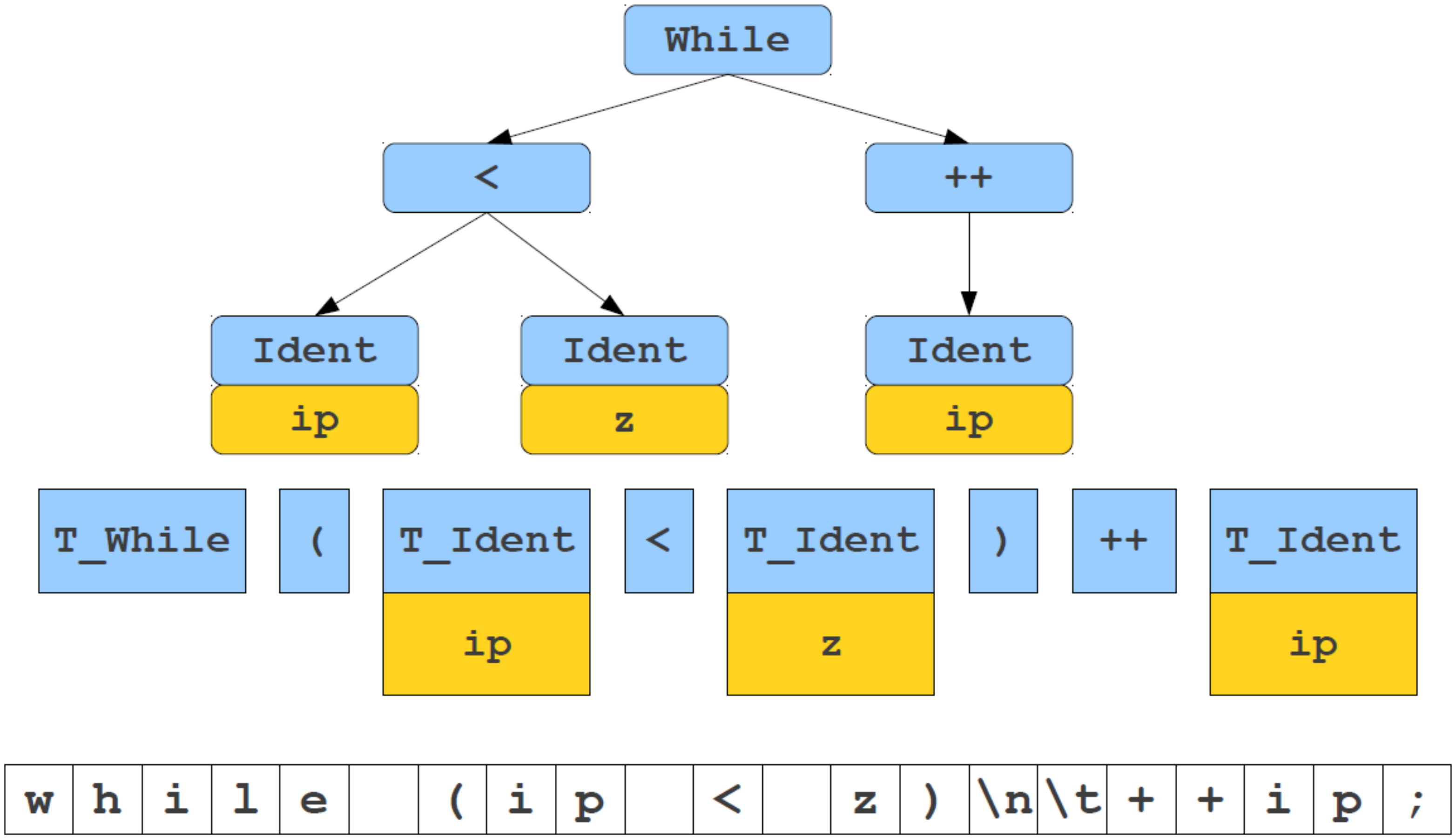
w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

T_While	(	T_Ident	<	T_Ident	)	++	T_Ident
		ip		z			ip

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```


While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
      ++ip;
```

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

The piece of the original program from which we made the token is called a **Lexeme**.

`T_While`

This is called a **token**, You can think of it as an enumerated type representing what logical entity we read out of the source code

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Sometimes we will discard a lexeme rather Than storing it for later use. Here, we Ignore white space, since it has no bearing on the meaning of the program

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

```
T_While
```

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While` `(`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

Scanning a Source File

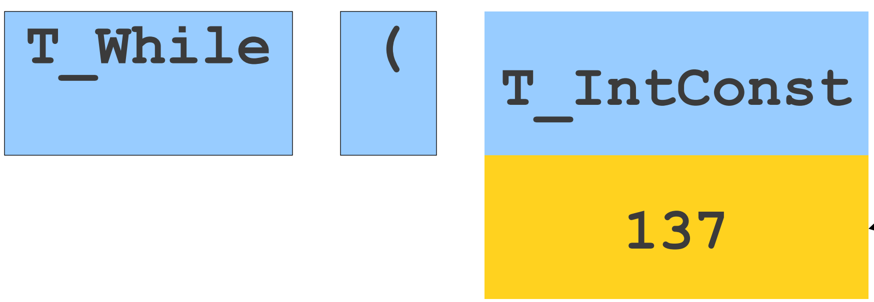
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

(

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;



Some tokens can have **attributes** that store extra information about token. Here we store which integer is represented

Role of lexical analyzer

Issues in design of lexical analysis

- Several reasons for separating the analysis phases of compiling into lexical analysis and parsing
  - Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax often allows us to simplify one or the other of these phases.
  - Compiler efficiency is improved. A separate lexical analysis allows us to construct a specialized and potentially more efficient processor for the task.

Role of lexical analyzer

Processes in lexical analyzers

- Scanning
  - Pre-processing
    - Strip out comments and white space
    - Macro functions
- Correlating error messages from compiler with source program
  - A line number can be associated with an error message
- Lexical analysis

Role of lexical analyzer

Terms of the lexical analyzer

- Token
  - Types of words in source program
  - Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas, semicolons)
- Lexeme
  - Actual words in source program
- Pattern
  - A rule describing the set of lexemes that can represent a particular token in source program
  - **Relation** {<.<=,>,>=,==,<>}

Role of lexical analyzer

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

Role of lexical analyzer

Attributes for Tokens

- A pointer to the symbol-table entry in which the information about the token is kept

E.g E=M*C**2

<**id**, pointer to symbol-table entry for E>

<**assign_op**,>

<**id**, pointer to symbol-table entry for M>

<**multi_op**,>

<**id**, pointer to symbol-table entry for C>

<**exp_op**,>

<**num**,integer value 2>

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: "137," "int," etc.
- Each token may have optional **attributes.**
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

Role of lexical analyzer

Lexical Errors

- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters(such as , fi=>if)
- Pre-scanning

Role of lexical analyzer

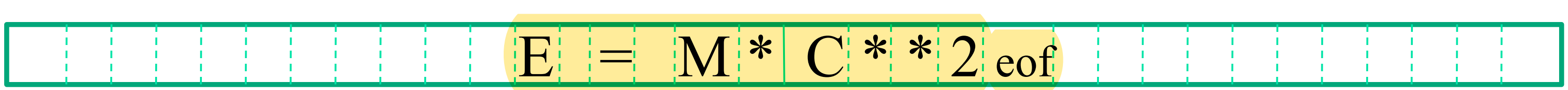
Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Role of lexical analyzer

Input Buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely



Approach for implementation of a lexical analyzer

- Three general approach to the implementation of a lexical analyzer.
  - Use a lexical analyzer generation, such as LEX compiler to produce the lexical analyzer from a regular expression based specification. In this case, the generator provides routines for reading and buffering the input.
  - Write the lexical analyzer in a conventional system programming language, using the I/O facilities of that language to read the input.
  - Write the lexical analyzer in assembly language and explicitly manage the reading of input.

What Token are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
    cout << k << endl;
}
```

```
for          {
int          }
<<           ;
= (          <
)            [
++           ]

Identifier
IntegerConstant
```

Choosing Good Tokens

- Very much dependent on the language.
- Typically
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

Syntax Definition

- A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement else statement

- That is, an if-else statement is the concatenation of the keyword if, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword else, and another statement.
- Using the variable expr to denote an expression and the variable stmt to denote a statement, this structuring rule can be expressed as

stmt → if (expr) stmt else stmt

- in which the arrow may be read as "can have the form." Such a rule is called a production. In a production, lexical elements like the keyword if and the parentheses are called terminals. Variables like expr and stmt represent sequences of terminals and are called nonterminals.

Definition of Grammars

- A context-free grammar has four components:
  - A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.
  - A set of nonterminals, sometimes called "syntactic variables." Each nonterminal represents a set of strings of terminals, in a manner we shall describe.
  - A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.
  - A designation of one of the nonterminals as the *start* symbol.

Derivations

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal.
- The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.