$\boxed{16}$ Lexical Analysis

16.1 An Arithmetic Interpreter

We will write a program that interprets a small arithmetic language. An example run is:

```
Please enter your program, ending with a '.'.
Type ^D to quit.

x = 10;
y = 100;
z = x + 5 * y;
.

Value:
510
```

$\boxed{16.2}$ Interpreter (or Compiler) Program Structure

character stream (file)

↓ **lexical analyzer**

lexeme stream

↓ **parser**

parse tree

↓ **semantic algorithm**

values (or compiled code)

16.3 The Lexical Analyzer

A lexical analyzer converts a stream (sequence) of characters (i.e. a file) into a stream (sequence) of lexemes.

Each lexeme is representable by one or more character strings:

E.g.: the integer 1 by "`1`" or "`01`"
the identifier count by "`count`"
the left parenthesis punctuation mark by "`(`"

The character stream input is converted to lexeme representatives separated by ignored characters.

Example ignored characters:
whitespace characters
characters in comments

Lexemes are sometimes called tokens.

16.4 Specification of Lexemes

The following is a specification (set of rules) that describes the lexemes of an arithmetic language:

1. Whitespace characters are not part of any lexeme. They may be used to separate lexemes.
2. The characters `+`, `-`, `*`, and `/` are each single character operator lexemes.
3. The characters `=`, `.`, `(`, `)`, and `;` are each single character lexemes.
4. A number lexeme is any character string having the syntax of a C++ integer that begins with a digit.
5. A symbol is any sequence of letters and digits that begins with a letter.

$\boxed{16.5}$ Execution of the Lexical Analyzer

The process of converting a character string to a lexeme string consists of:

(1) Remove whitespace from the beginning of the character string.

(2) Remove the longest possible lexeme from the beginning of the character string.

Repeat (1) and (2) till character string is empty.

Comments:

1. Some of the material covered in this lecture is explained in the Lexical Analysis and Parsing chapter of the CS 51 project book, particularly Section 3 of that chapter, on the Lexical Analyzer.

2. Some languages have an end-of-file lexeme, but our arithmetic language uses "." for this purpose.

3. The term 'specification' refers in general to an abstract description of a computer program, but may also refer more specifically to a form of description in which rules are given which the program or its data must obey, as opposed to giving a piece of pseudo-code that describes each execution step, or a grammar that describes data.

$\boxed{16.6}$ Lexical Grammar

A lexical grammar differs from a language grammar in that a lexical grammar describes a character string, while a language grammar describes a lexeme string.

A Simplified Example:

lexeme ::= symbol | semi-colon
symbol ::= letter letter-or-digit*
semi-colon ::= ;
file ::= $\{$ whitespace lexeme $\}^\star$ whitespace
whitespace ::= white-space-character*

$\boxed{16.7}$ Lexical DFAs

DFAs must be enhanced to be useful in lexical analysis.

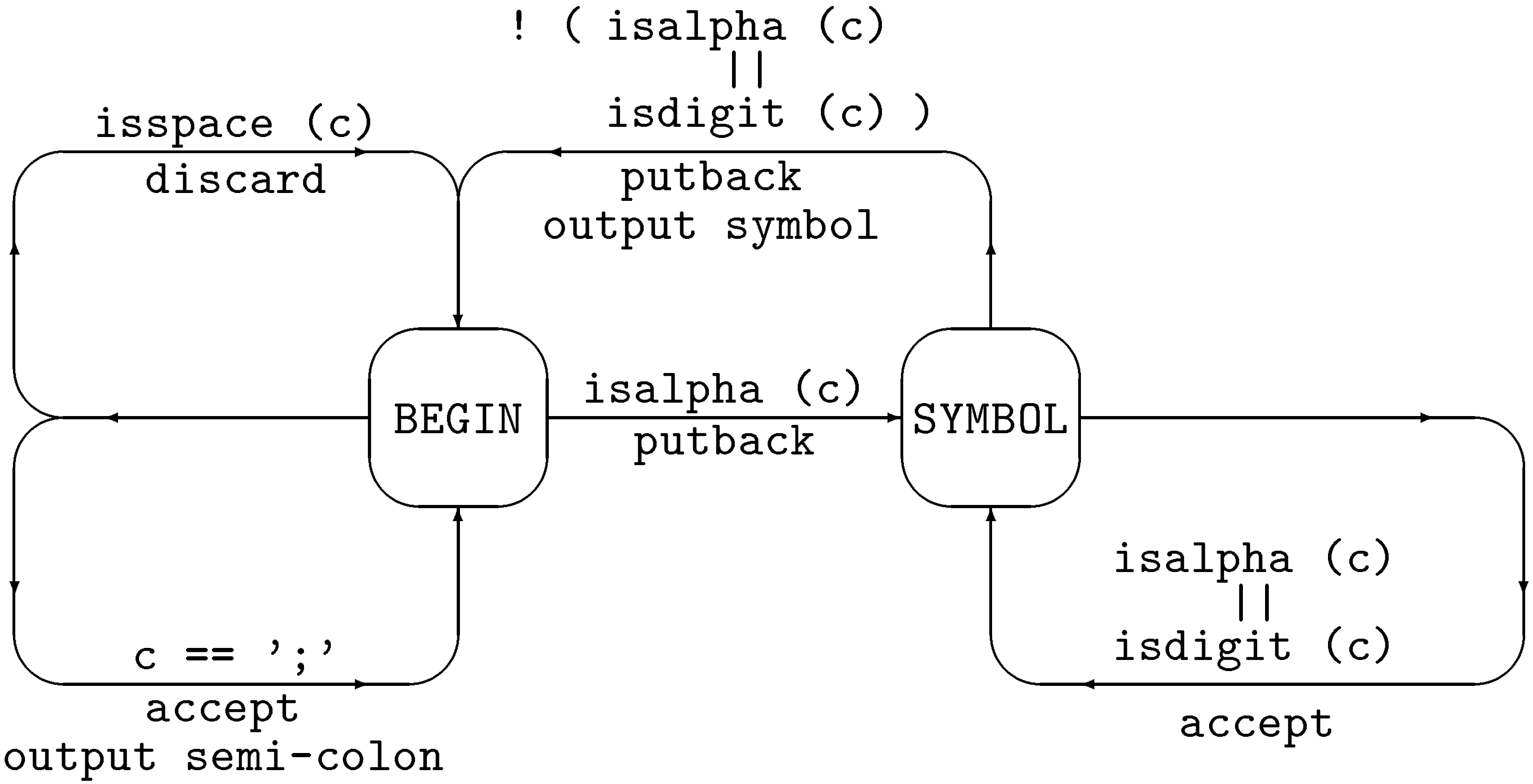
Transition labels may be tests on the current first input character, instead of just the name of the character.

Example: `isdigit(c)` [`c` = first input character]

Transitions may also be labeled with actions:

`accept`	Move first input character from input string into output buffer.
`discard`	Remove first character from input.
`putback`	Do nothing: leave first character in input.
`output ...`	Output a lexeme consisting of the characters in the output buffer, as a lexeme of type ..., and clear the output buffer.

16.8 DFA for Simplified Lexical Grammar

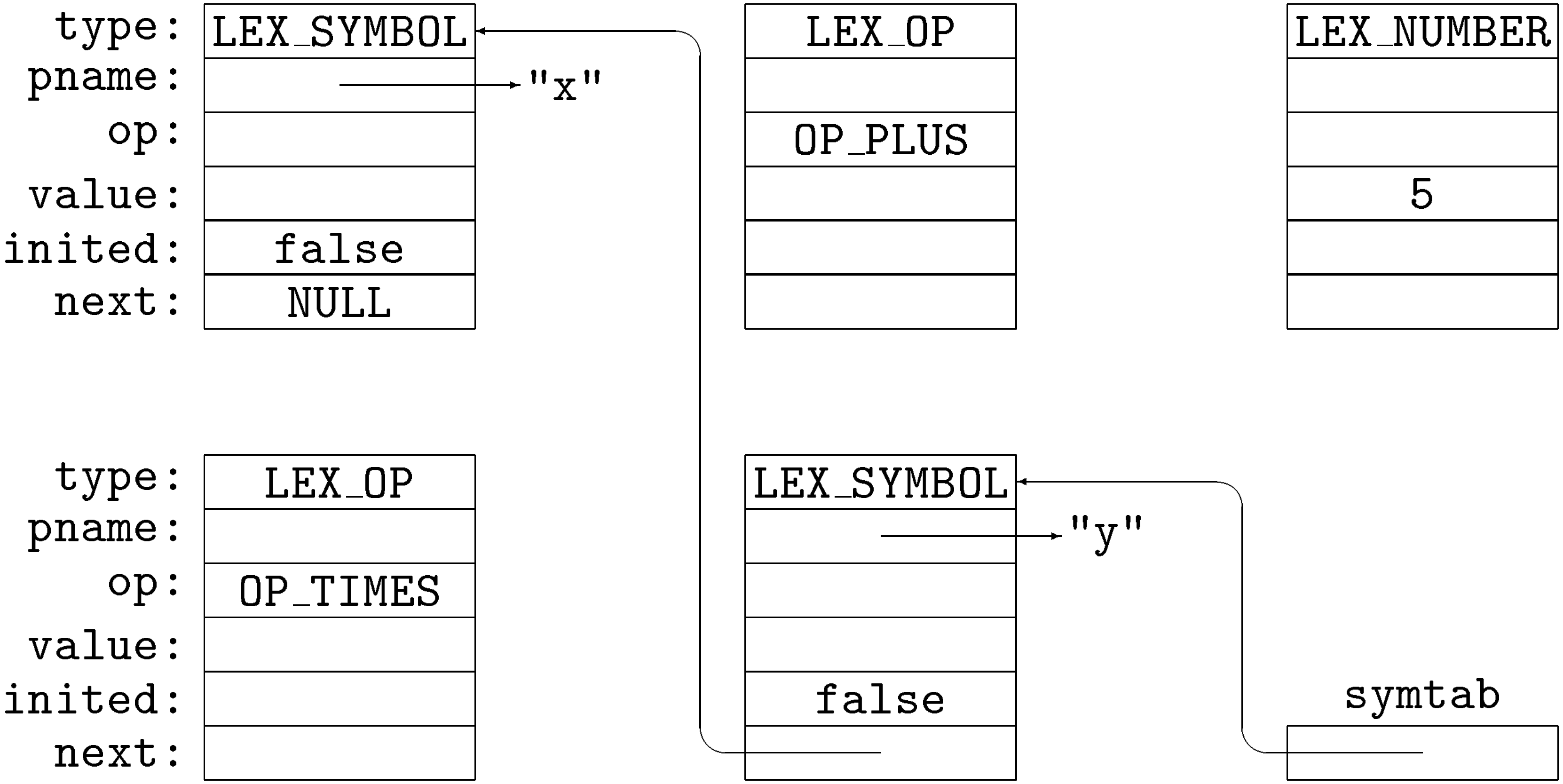


c is the current first input character

What if c == EOF?

16.9 Lexeme Memory

Output of lexical analyzer on input “`x + 5 * y`”:



Comments:

4. Symbols have their name stored in the `pname` component as a pointer to a character string. Operators have the operation stored as a numeric code in the `op` component. Numbers have their value stored in the `value` component.

5. All the symbol lexemes are placed in a common list called the symbol table.

   When a new symbol is read, this list is searched to see if the symbol is already present, and if yes, the already present symbol lexeme is returned as the next lexeme. Otherwise a new symbol lexeme is created.

   Therefore all symbol lexemes with the same name are the same.

6. Symbol lexemes are used to name variables. For convenience, the variable value is built into the symbol lexeme.

   Specifically, there is an `inited` component of the lexeme which tells whether or not the variable has ever been assigned a value, and a `value` component that holds the last value assigned, if any.