Unit 2: An Introduction to Lexical Analysis

SCC 312 Compilation

Barry Porter
b.f.porter@lancaster.ac.uk

Unit 2: An Introduction to Lexical Analysis

SCC 312 Compilation

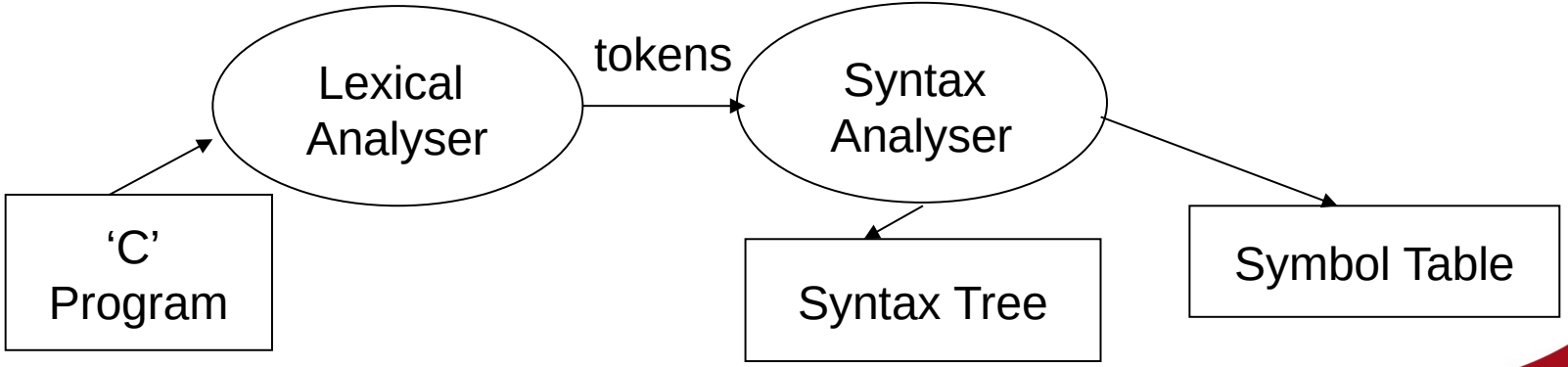
Barry Porter
b.f.porter@lancaster.ac.uk

Aims of this Unit

- What does the LA do?
- What are tokens?
- Why have a separate LA phase?
- The API to an LA
- Dealing with various features of the (input) language
  - String literals
  - Numeric literals
  - Identifiers
  - Reserved words

What is the job of the LA?

- One way to think of the LA's job is that it "purifies" or "simplifies" the input by throwing away unnecessary information (like comments) and by identifying and classifying language components.
- It produces a stream of "tokens"



Tokens

- This phase breaks the source program up into a sequence of tokens, to pass to the syntax analysis phase. These tokens will be:
  - The symbols of the language (including keywords)
    - *, :=, for, if, while, …
  - Identifiers
    - names of variables, constants, values of enumeration types, methods, classes, …
  - Literal constants
    - numbers, characters, character strings, …
- There will also be whitespace and comments, all of which are to be ignored by the syntax analysis phase

Why is this not just a tokeniser?

- Most programming languages have way to “tokenise” a string using a set of token-separators (like spaces, or commas)
- This kind of tokenisation is not powerful enough for the source code of programming languages
  - Or even for things like CSV files, JSON, or XML

Example CSV file:

```
Name, Age, Occupation, Country
Sam, 28, “Window Cleaner”, UK
Alex, 23, Student, Australia
```

Lexical Analysis

- The word **lexical** means “of or relating to words or the vocabulary of a language”, derived from the Greek word *lexis* meaning speech

- A *lexical analyser* has added intelligence beyond a simple tokeniser, allowing it to understand things like quoted strings, varying number formats, or parenthetical nesting

- Lexical analysers are useful to parse many kinds of syntax-bearing text data, including source code, CSV, Latex, JSON, XML, SVG, etc.

Why a Separate LA Phase?

- It may be easier and clearer to specify the format of the input if we separate it into the two phases (because of comments, etc.)
- Since all the above types of tokens can be specified by a regular grammar, the lexical analyser can execute more efficiently

Why a Separate LA Phase?

- Since all handling of input is now dealt with by this phase, we may be able to process it more efficiently
- Some programming languages are designed with alternative input formats, and these may most easily be dealt with by alternative lexical analysers with the same syntax analyser

Programming Interface to LA

```
Yylex yy = new Yylex(s2);
Yytoken t;
while ((t = yy.yylex()) != null) /* get next token */
{
        System.out.println(t);
};
```

```
type Yytoken {
   int class;
   char text[];
   int startPos;
   int endPos;
   }
```

- We create a new LA for input file “s2”.
- We have a “current token” object (or data structure) “t”
- The LA returns tokens until there are no more and finally returns null.
- In the loop we handle the token.
- This program reads a file, splits it into tokens, and prints out each token.

How to Produce a LA

- Write one yourself
- Use a lexical analyser generator
  - Beyond the scope of this course
- Generate it formally from a regular grammar

A variation of Ada : Ada-

- As our example language we are using a simple variation of the Ada imperative procedural programming language.

```
procedure program2 is -- this is not
                                  -- standard Ada
x1, x2 :  integer ;
text : string
begin
call get(x1) ;                    -- input number
x2 := 1 ;
while x1 /= 0 loop
        x2 := x2 * x1 ; -- multiply by next term
        x1 := x1 - 1              -- decrement count
end loop ;
text := "the result is " ;
call put(text) ;
call put(x1)                      -- output factorial
end
```

- Take the first line of our example program
- `procedure program2 is`
- “`procedure`” and “`is`” are examples of reserved or key words of the language.
- “`program2`” is an example of a user-named identifier, in this case the name of the procedure.
- “`procedure`”, “`is`” and “`program2`” are all examples of tokens.
- When the LA returns a token data structure it may contain more than just the character value of the token.

- The LA will normally have an enumerated list of all the kinds of tokens it will encounter.
- In Java we could say
- `final int PROC_TOK = 0, IS_TOK = 1, IDENTIFIER = 2;`
- When returning a token data structure (or, in Java, a token object) it may have additional attributes.
- `(PROC_TOK,”procedure”), (IS_TOK,”is”), (IDENTIFIER,”program2”)`

Example

- Tokenising the following assignment statement.
- `int count = 5;`
- This should produce the following token stream.

INT_TOKEN	“int”
IDENTIFIER_TOKEN	“count”
ASSIGNMENT_SYMBOL_TOKEN	“=”
NUMERIC_VALUE_TOKEN	“5”
SEMI_COLON_TOKEN	“;”

- What happens when?
- `currentTok = la.getNextToken();`
- The LA has to work out what the next token is, in order to return it.
- It starts by considering the current character which it holds. (this should be the next character after the preceding token that has been dealt with).
- It could start by checking to see if it is dealing with “whitespace” and try to move on past it.

Strings : Literal Value Bracketing Symbols

- `text:="the result is ";`
- When we have a statement such as the above, we have a “string literal” which is the value of a string.
- A string literal is detected by the appearance of a single double-quote character (“), and goes on until we reach the matching double-quote at the other end of the string.

Strings : Literal Value Bracketing Symbols

- `text:="the result is ";`
- Within the double-quote “brackets” anything, including white space, reserved words, etc. can appear.
- If your string contains a double-quote, it has to be “escaped”. i.e. “He said \”Hello\”, and smiled.”
- If your string contains an escape character, it has to be “escaped”.

Numeric Literals

- Once we have detected a numeric value at the start of a token, we need to absorb and return that value as a numeric token.
- Some languages allow some form of prefix/suffix to specify a base other than 10
- Some languages allow the characters a to f in hexadecimal constants, usually with a leading digit to indicate that it is not an identifier
- Real/float numbers like 123.456 or 43.3e-12 could easily be accommodated
- In all cases we would still be within a regular grammar

Identifiers

- We assume that identifiers are of the traditional form
  - An alphabetic character followed by any number of alphabetic characters and digits
- Some languages allow other characters to appear
  - For example “_” in Java (but not usually as the first character)

Reserved words. Keywords

- There will be a set of reserved words, (if, for, while, etc.)
- These have a similar form to identifiers (though usually they consist only of alphabetic characters)

Reserved words : Table

- We recognise these by looking up the character string we have found in a table of these reserved words
- We return either the “appropriateKeyword”Token (for example, ifToken)
- Or we return IdentifierToken and “tokenText”, if it is not in the table
- Generally the language will not allow the reserved words to be used as identifiers

Worked Example

```
if (z < 5) System.out.printf("Problem!")
int count = 5;
for (int i = 0; i < 10; i++) {
```

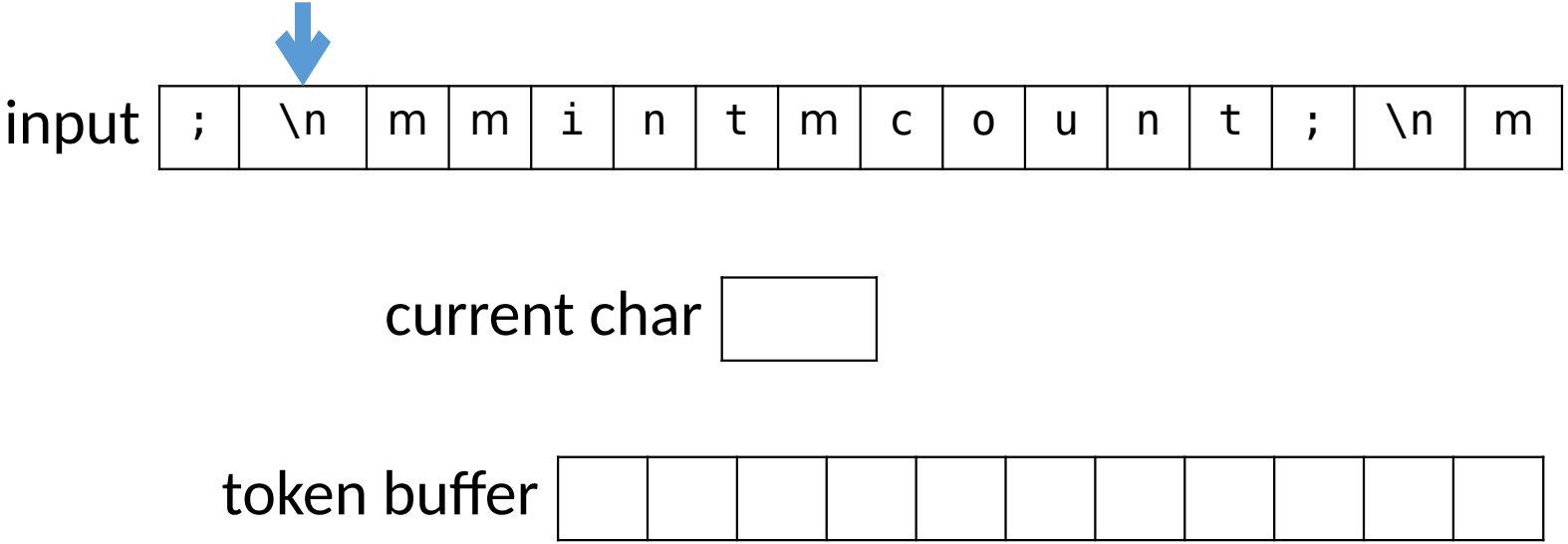
)	;	\n			i	n	t		c
o	u	n	t	;	\n			f	o

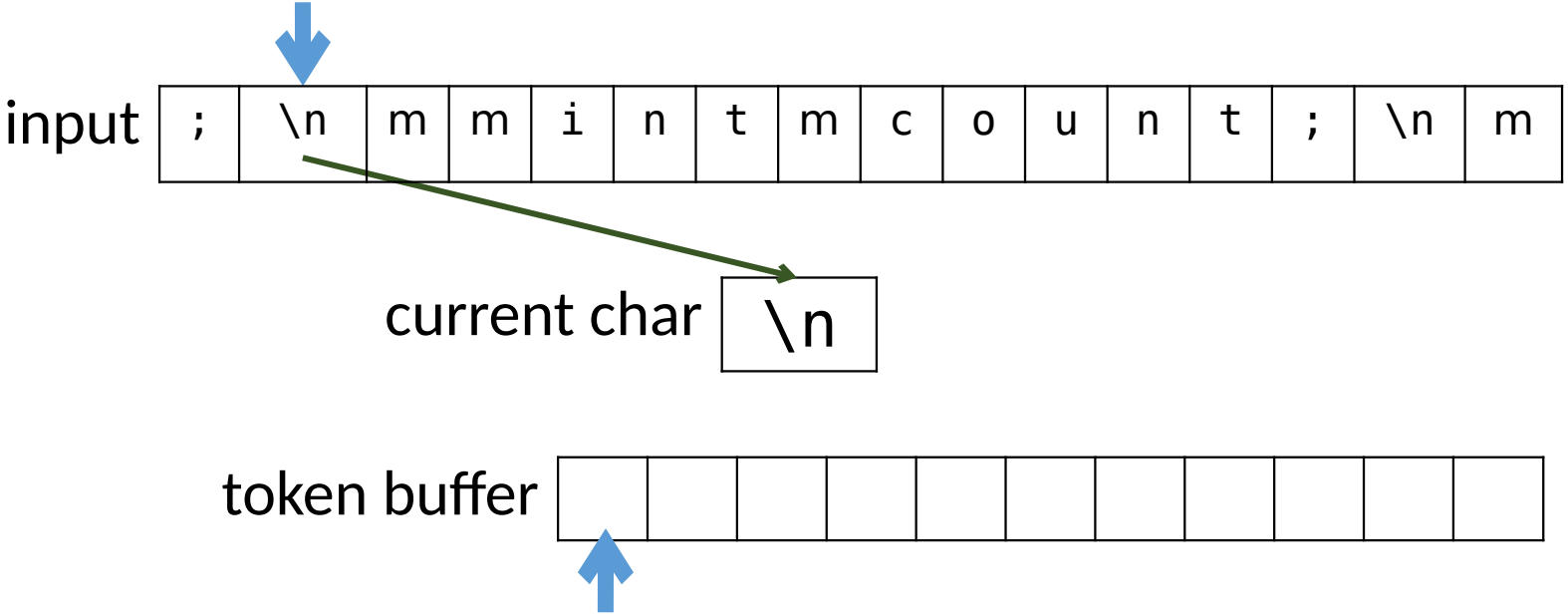
a portion of the input buffer

input

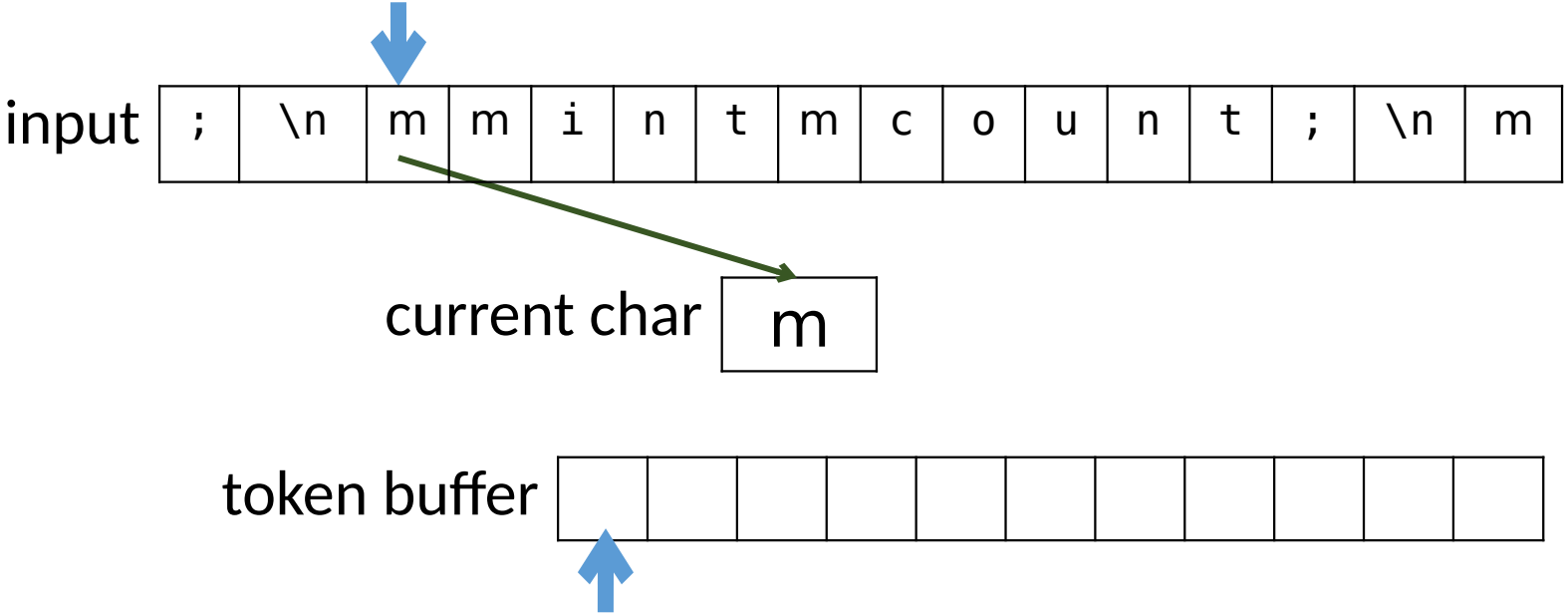
;	\n	m	m	i	n	t	m	c	o	u	n	t	;	\n	m

the portion of the input buffer we will use.

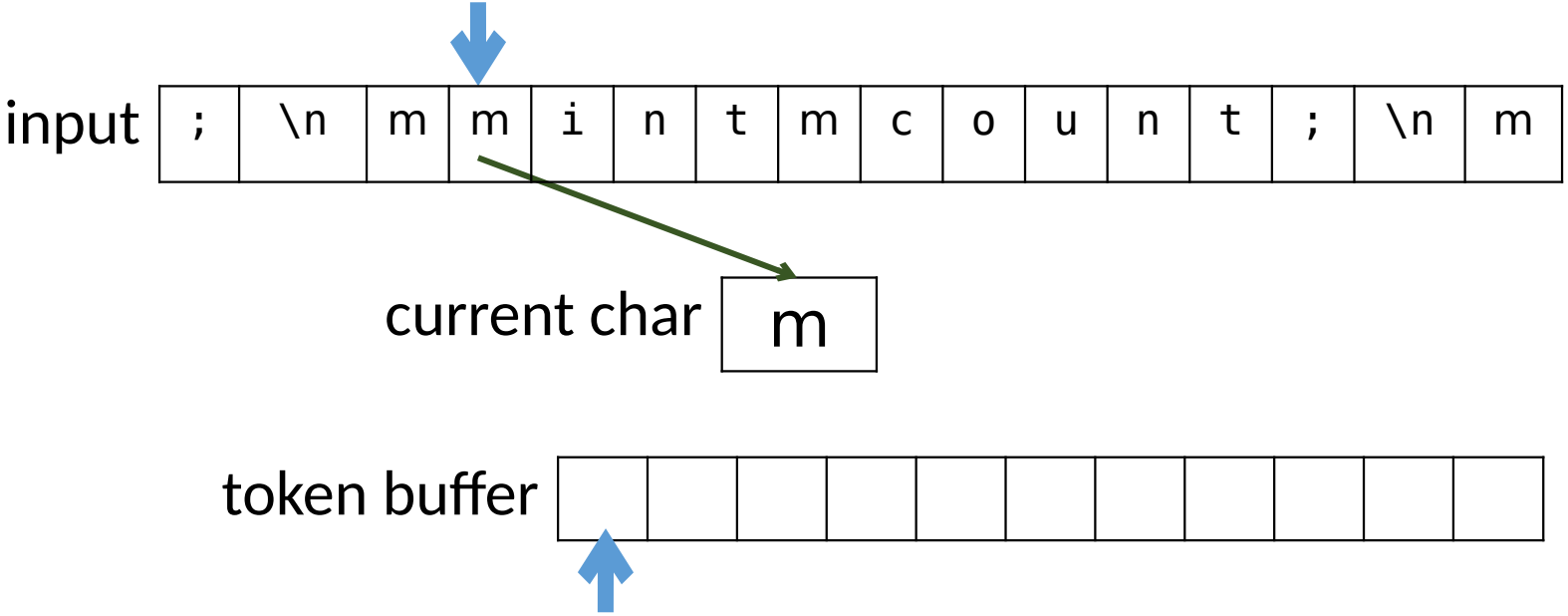

input
; \n m m i n t m c o u n t ; \n m
current char
token buffer



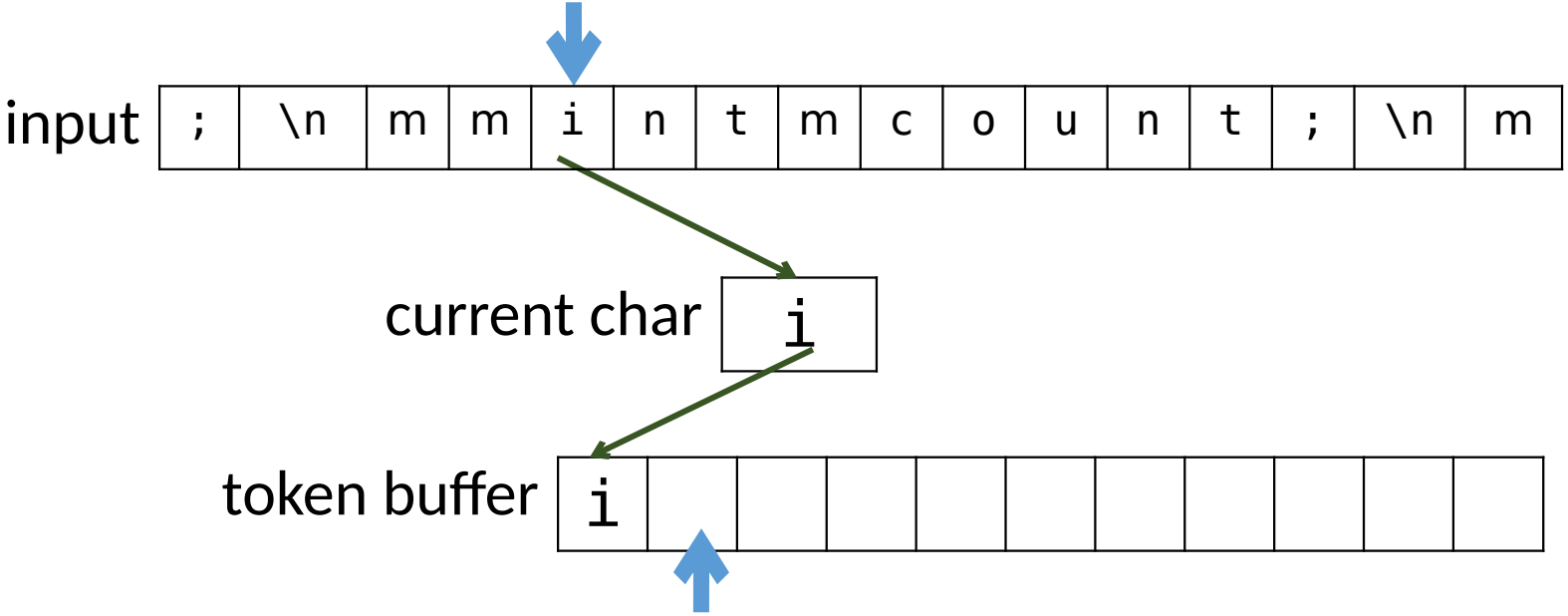
A newline character is treated as whitespace; discard

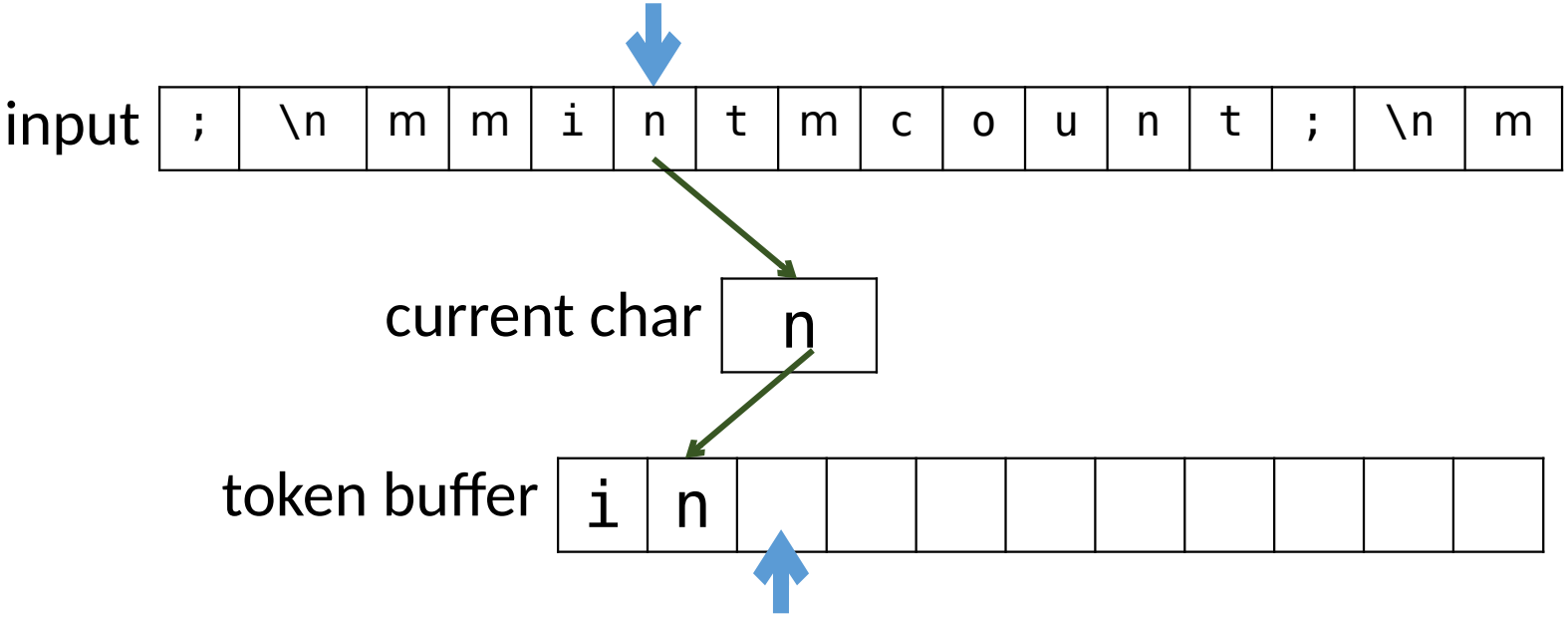


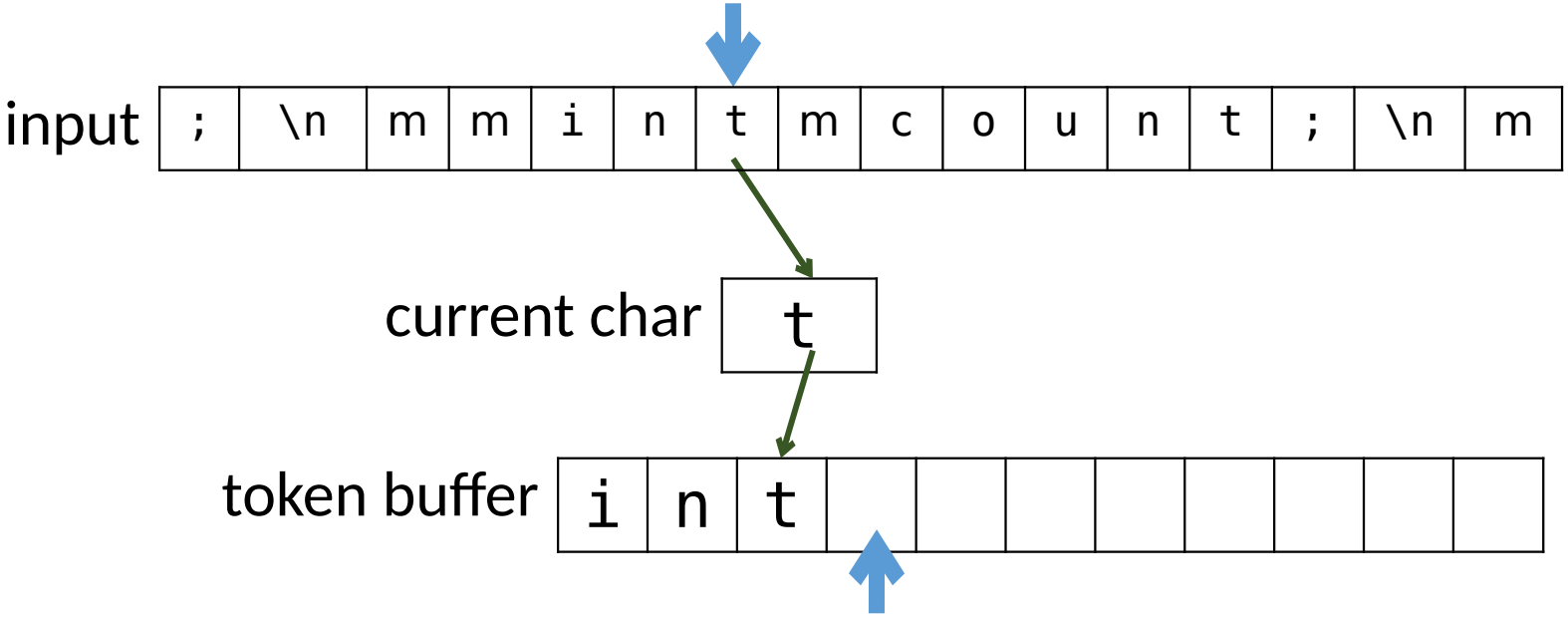
A space character is treated as whitespace; discard

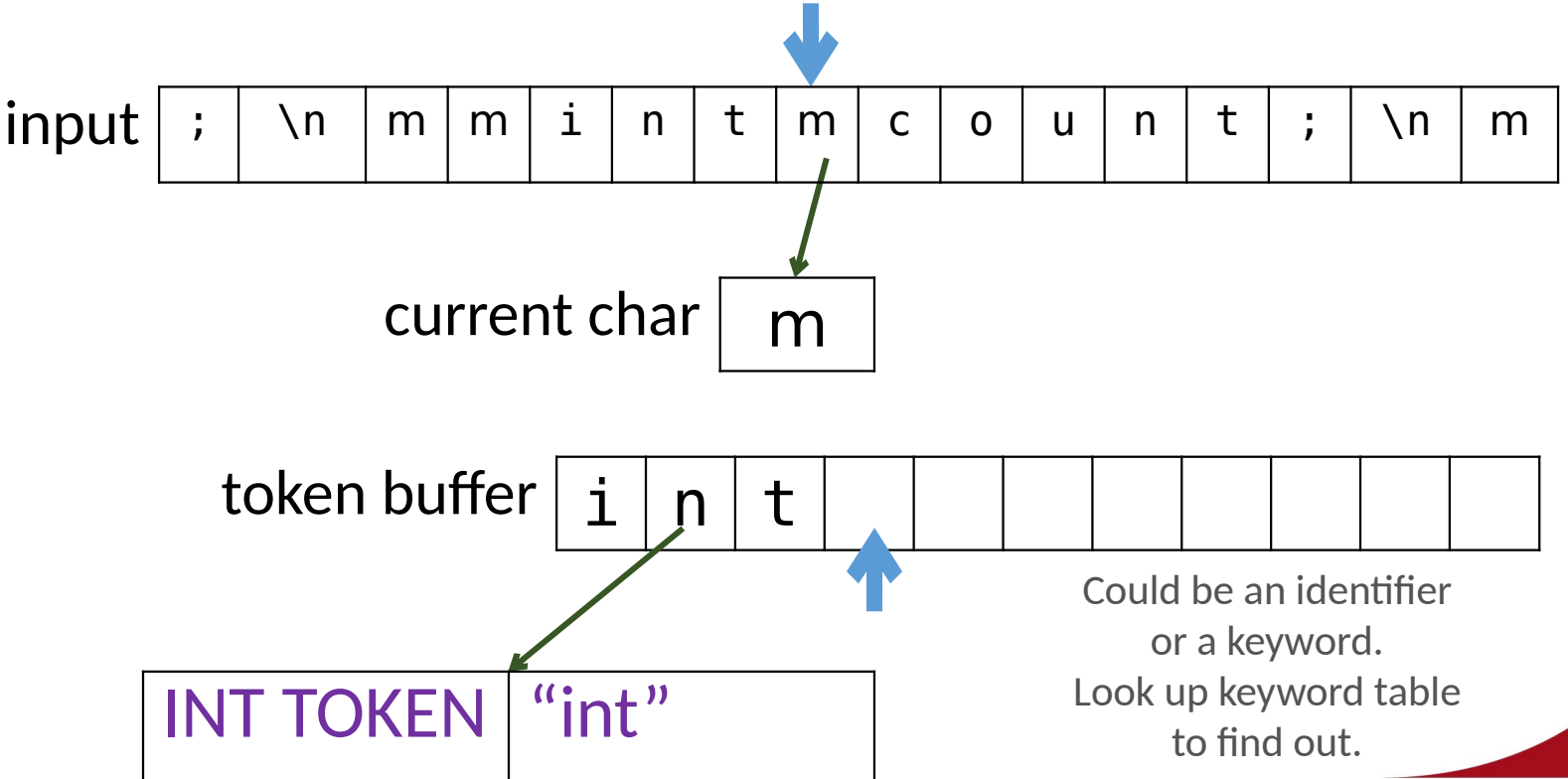


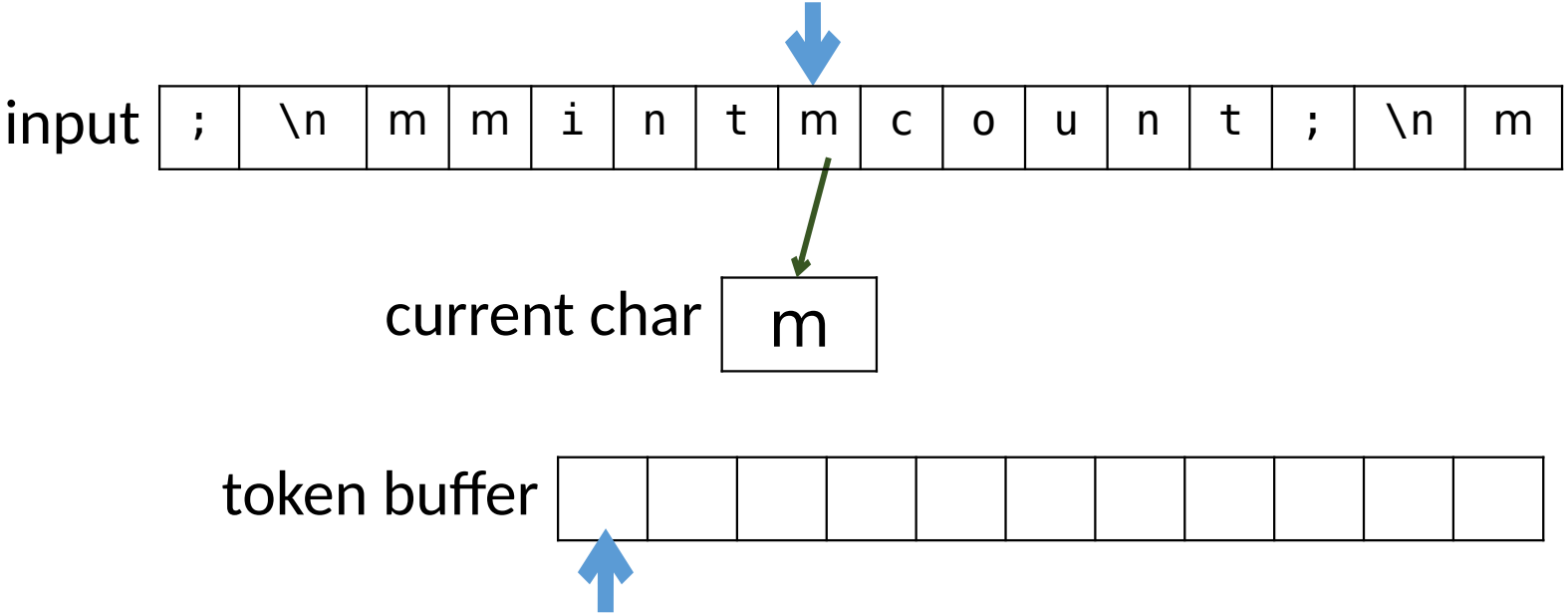
A space character is treated as whitespace; discard

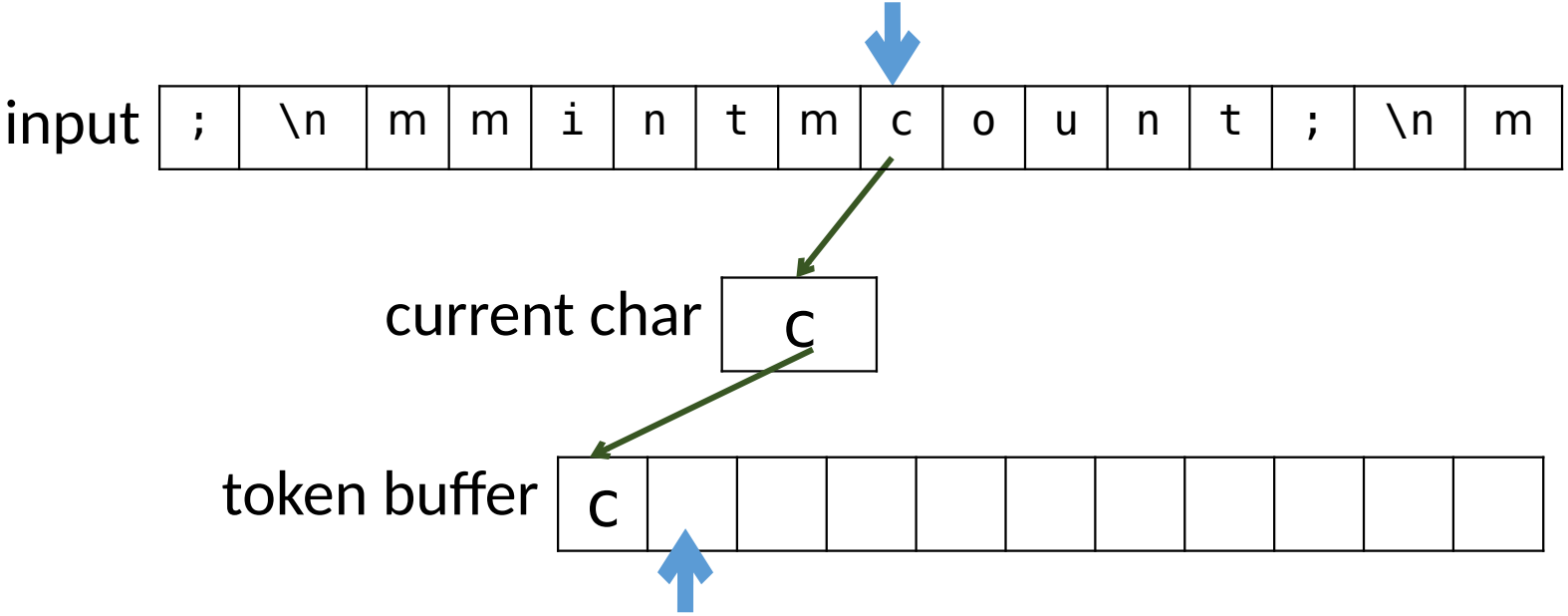

input
; \n m m i n t m c o u n t ; \n m
current char
i
token buffer
i

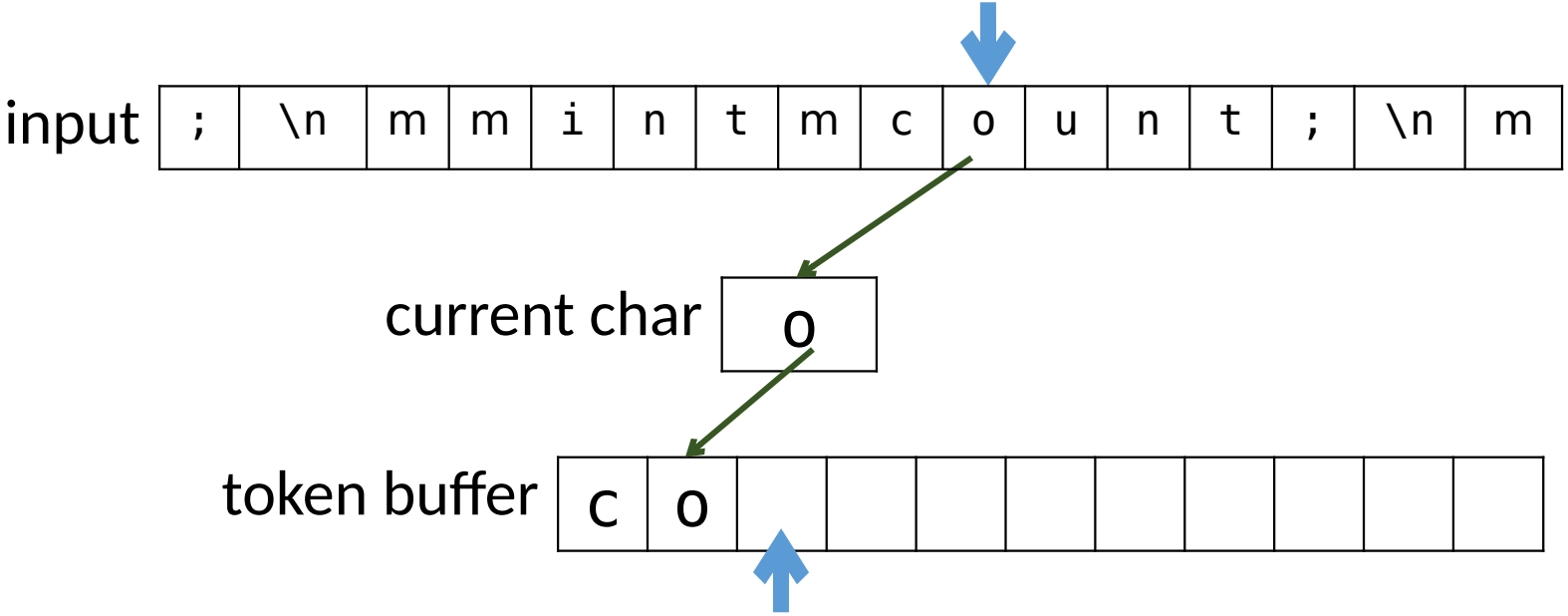

input
; \n m m i n t m c o u n t ; \n m
current char
n
token buffer
i n

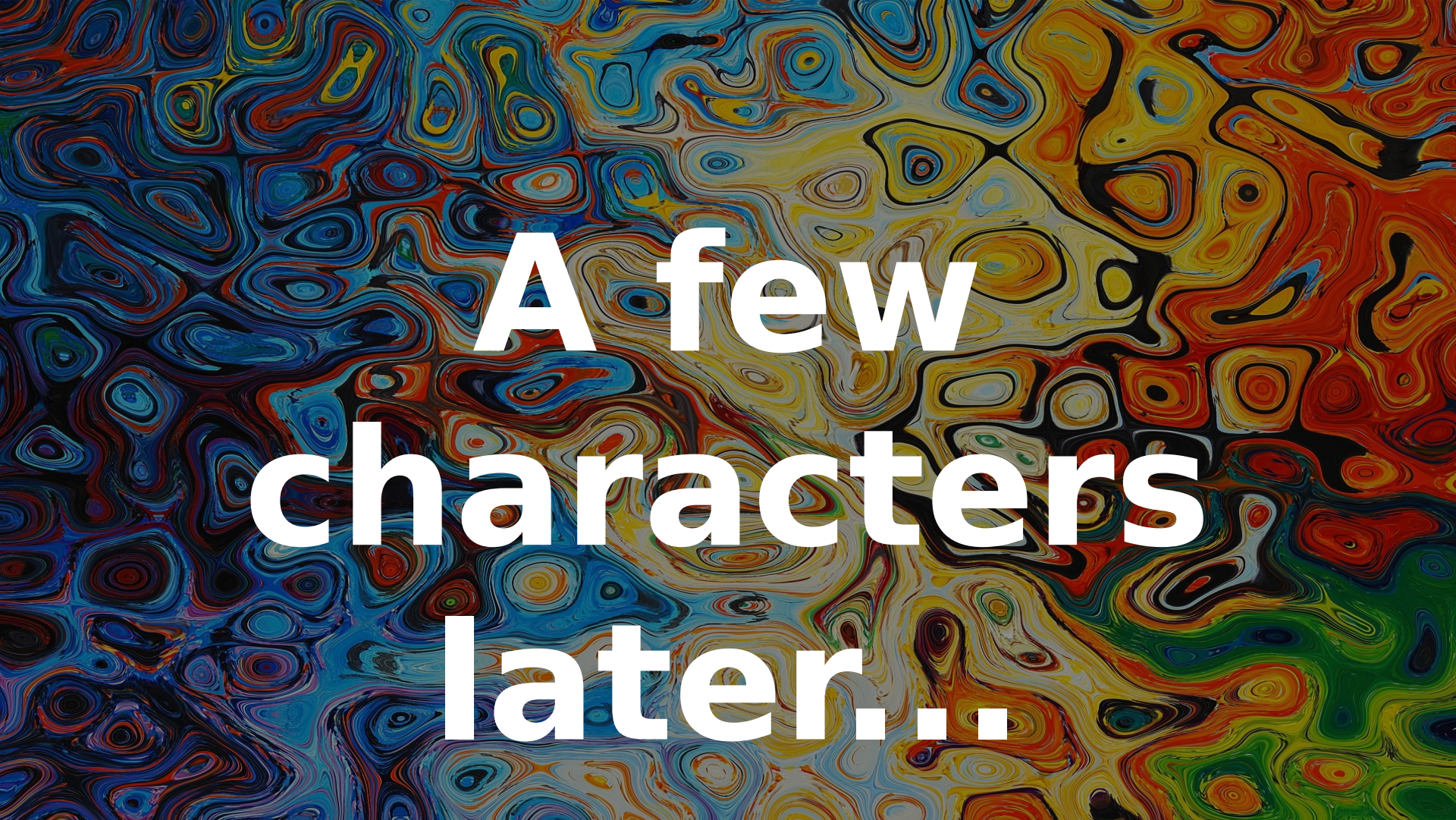

input
; \n m m i n t m c o u n t ; \n m
current char
t
token buffer
i n t


input
; \n m m i n t m c o u n t ; \n m
current char
m
token buffer
i n t
INT TOKEN
“int”
Could be an identifier
or a keyword.
Look up keyword table
to find out.

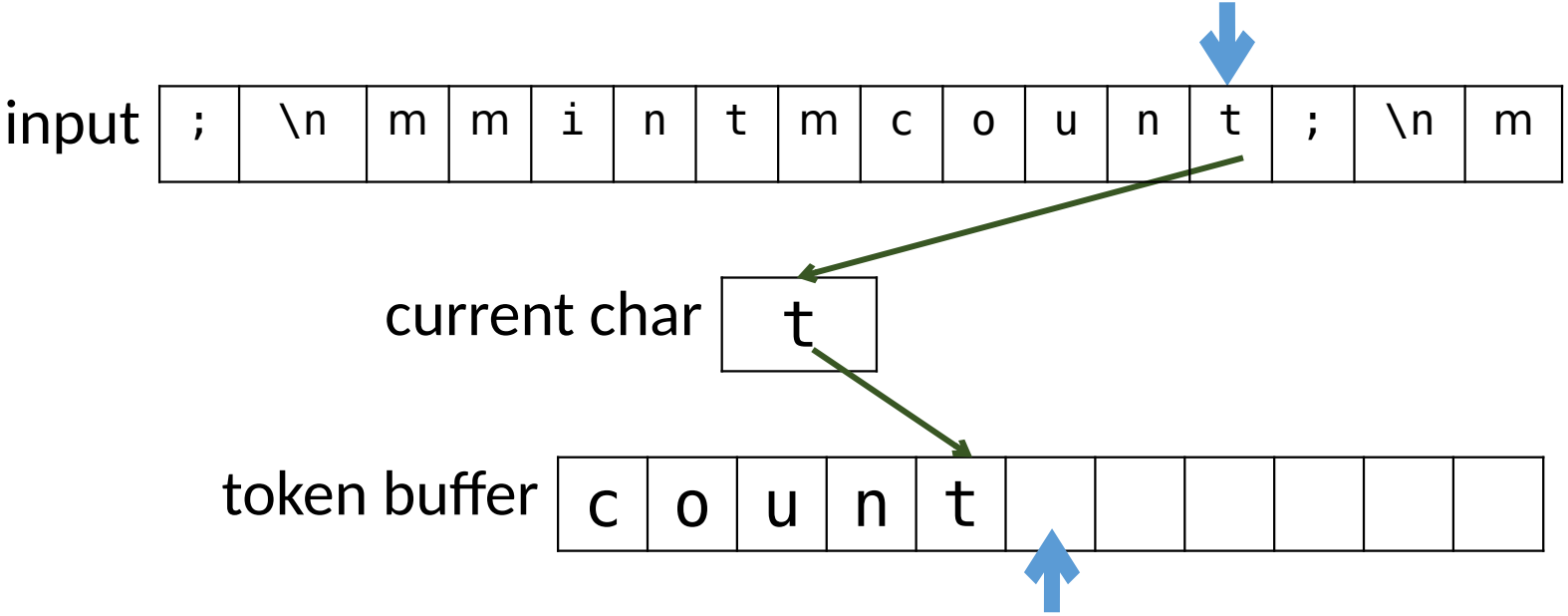

input
; \n m m i n t m c o u n t ; \n m
current char
m
token buffer

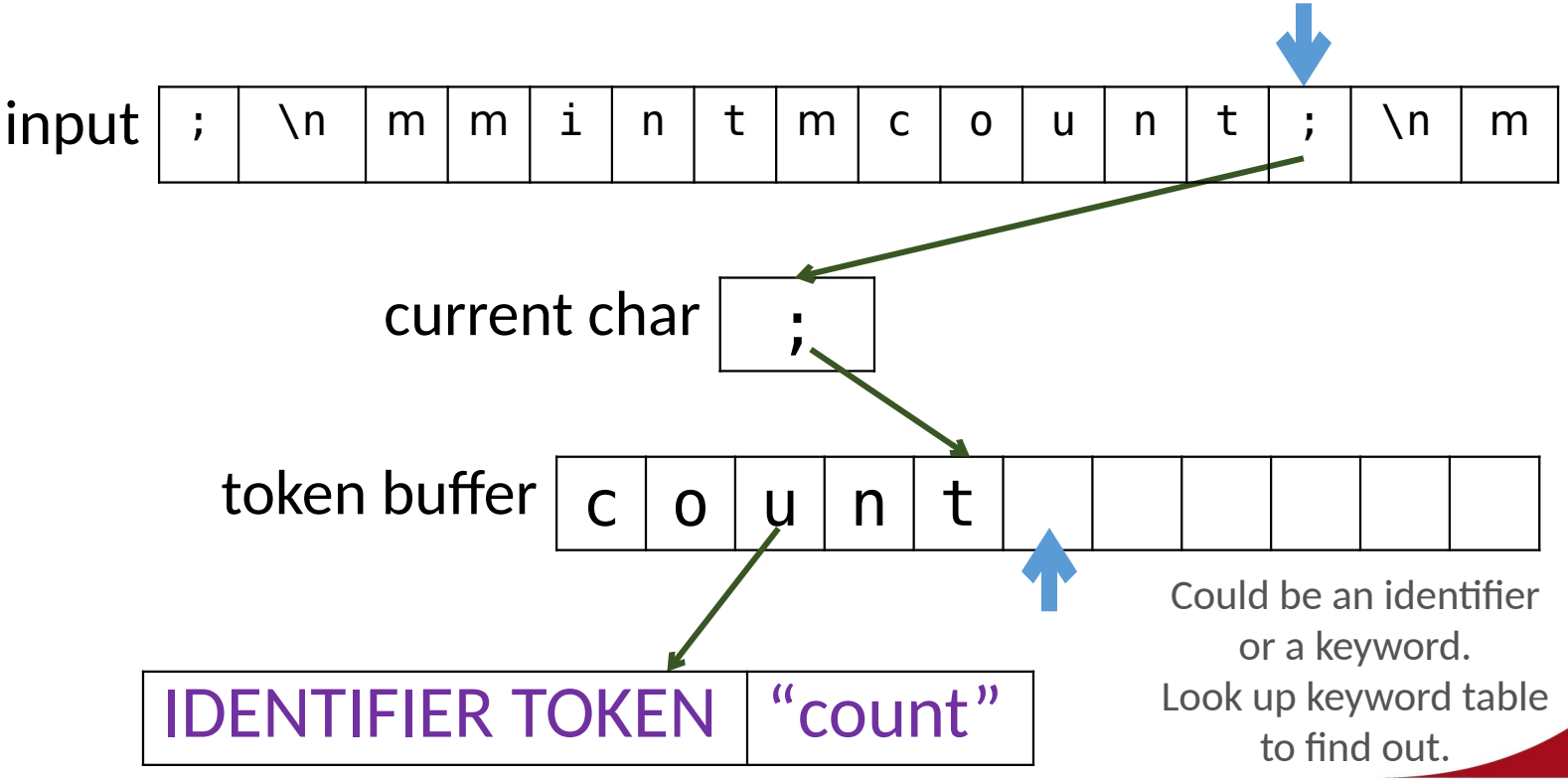

input
; \n m m i n t m c o u n t ; \n m
current char
c
token buffer
c

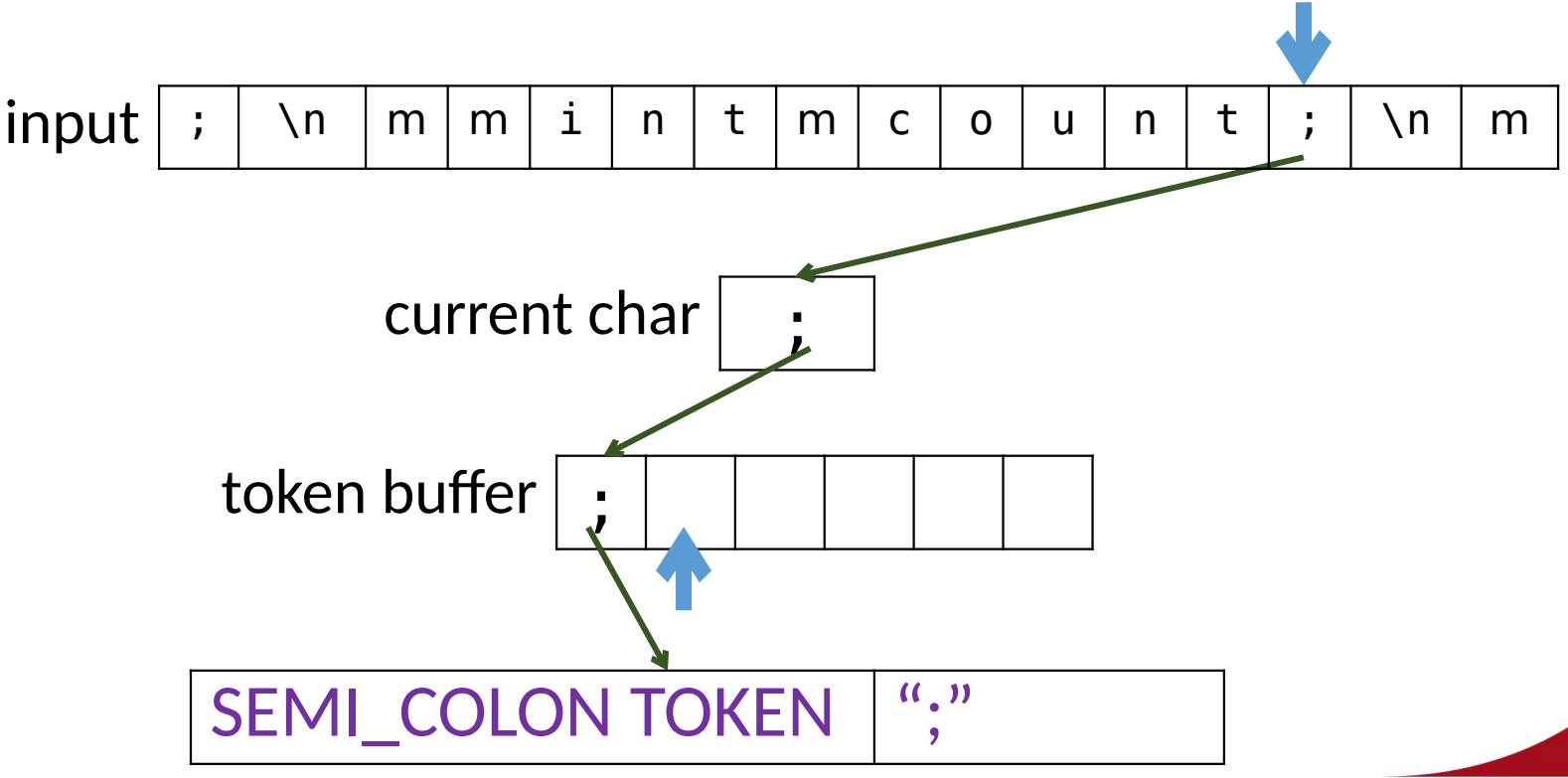

input
; \n m m i n t m c o u n t ; \n m
current char
o
token buffer
c o

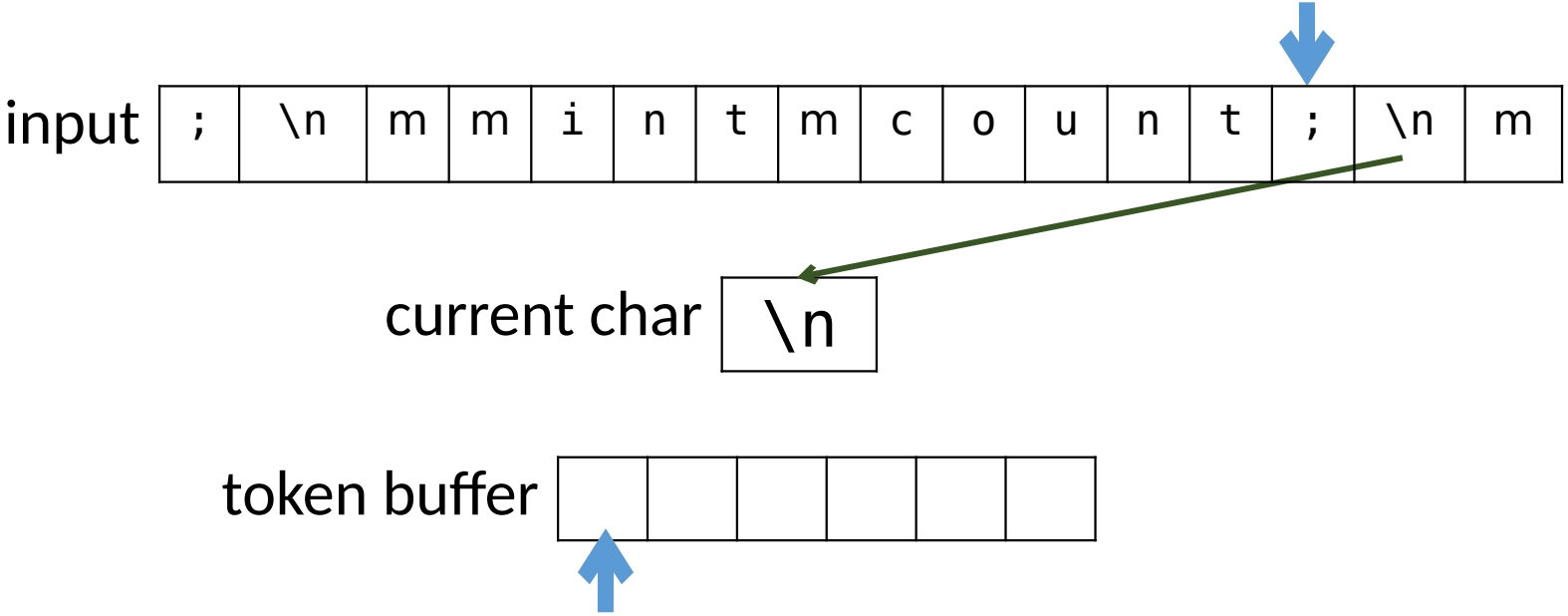

A few
characters
later...

A few characters later ..




input
; \n m m i n t m c o u n t ; \n m
current char
;
token buffer
c o u n t
IDENTIFIER TOKEN
“count”
Could be an identifier
or a keyword.
Look up keyword table
to find out.


input
; \n m m i n t m c o u n t ; \n m
current char
;
token buffer
;
SEMI_COLON TOKEN
“;”



A newline character is treated as whitespace; discard

Outcomes

- You should have gathered an understanding of…
  - The function of an LA
  - Identifying tokens
  - Handling various features of a language to ensure they are tokenised correctly