Recognition of tokens

Until now we showed how to specify tokens. Now we show how to recognise them, i.e., realise lexical analysis. Let us consider the following token definition:

$$
\begin{aligned}
\textbf{if} &\rightarrow \texttt{if} \\
\textbf{then} &\rightarrow \texttt{then} \\
\textbf{else} &\rightarrow \texttt{else} \\
\textbf{relop} &\rightarrow \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{digit} &\rightarrow [\texttt{0-9}] \\
\textbf{letter} &\rightarrow [\texttt{A-Za-z}] \\
\textbf{id} &\rightarrow \textbf{letter}\ (\textbf{letter} \mid \textbf{digit})^{\star} \\
\textbf{num} &\rightarrow \textbf{digit}^{+}\ (\texttt{.}\ \ \textbf{digit}^{+})?\ (\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ \textbf{digit}^{+})?
\end{aligned}
$$

Recognition of tokens/Reserved identifiers and white space

It is common to consider keywords as **reserved identifiers**, i.e.,, in this case, a valid identifier cannot be any token **if**, **then** or **else**.

This is usually not specified but instead programmed.

In addition, assume lexemes are separated by white spaces, consisting of non-null sequences of blanks, tabulations and newline characters. The lexer usually strips out those white spaces by comparing them to the regular definition **white_space**:

$$\textbf{delim} \rightarrow \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}$$
$$\textbf{white\_space} \rightarrow \textbf{delim}^{+}$$

If a match for **white_space** is found, the lexer does **not** return a token to the parser. Rather, it proceeds to find a token following the white space and return it to the parser.

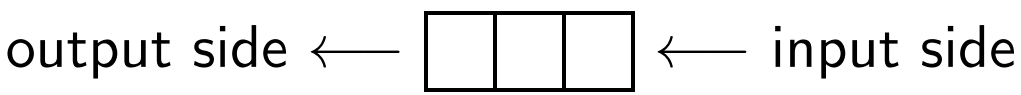
Recognition of tokens/Input buffer

The stream of characters that provides the input to the lexer comes usually from a file.

For efficiency reasons, when this file is opened, a **buffer** is associated to it, so the lexer actually reads its characters from this buffer in memory.

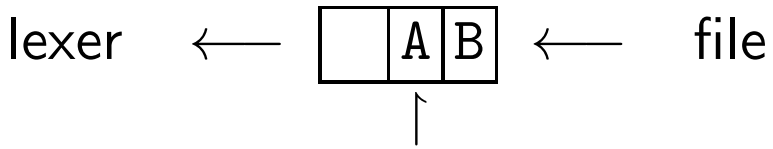
A buffer is like a **queue**, or **FIFO** (*First in, First out*), i.e., a list whose one end is used to put elements in and whose other end is used to get elements out, one at a time. The only difference is that a buffer has a **fixed size** (hence a buffer can be full).

An empty buffer of size three is depicted as

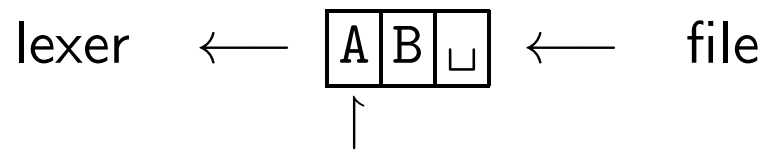


Recognition of tokens/Input buffer (cont)

If we input characters `A` then `B` in this buffer, we draw



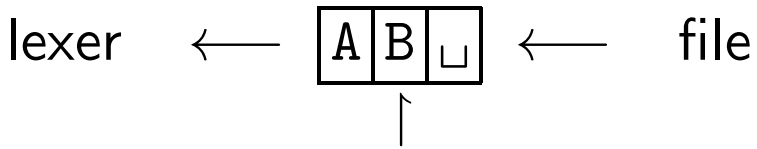
The symbol $\restriction$ is a pointer to the next character available for output. **Beware!** The blank character will now be noted $\sqcup$, in order to avoid confusion with an empty cell in a buffer. So, if we input now a blank in our buffer from the file, we get the full buffer



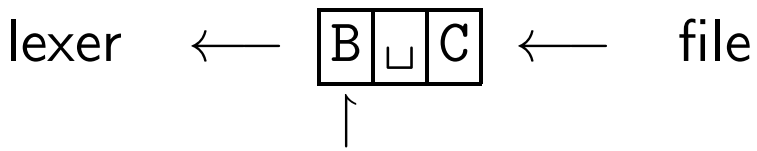
and no more inputs are possible until at least one output is done.

Recognition of tokens/Input buffer/Full buffer

Be careful: a buffer is full if and only if $\restriction$ points to the leftmost character. For example,



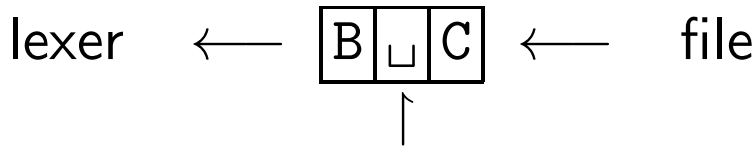
is **not** a full buffer: there is still room for one character.
If we input `C`, it becomes:



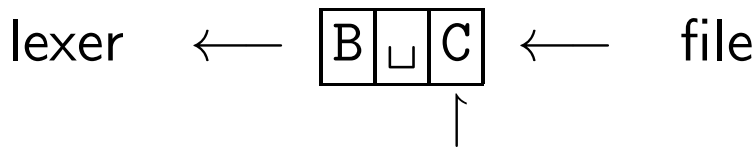
which is now a full buffer. The overflowing character `A` has been discarded.

Recognition of tokens/Input buffer (cont)

Now if we output a character (i.e.,, equivalently, the lexer inputs a character) we get



Let us output another character:

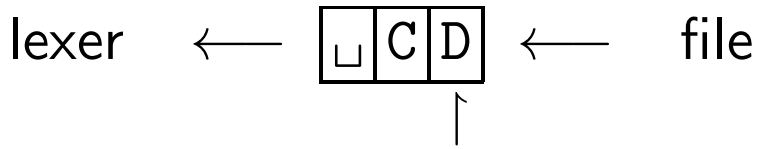


Now, if the lexer needs a character, C is output and some routine automatically reads some more characters from the disk and fill them in order into the buffer.

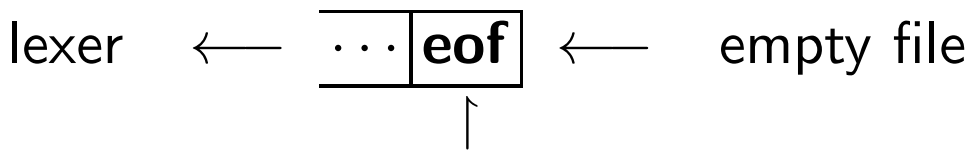
This happens when we output the rightmost character.

Recognition of tokens/Input buffer (cont)

Assuming the next character in the file is `D`, after outputting `C` we get



If the buffer only contains the **end-of-file** (noted here **eof**) character, it means that no more characters are available from the file. So if we have the situation



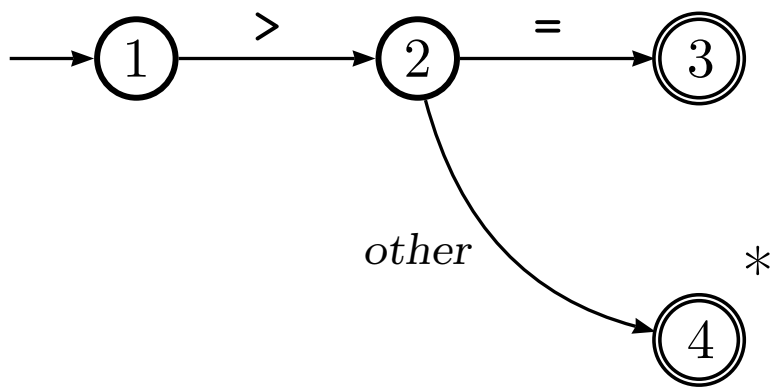
in which the lexer requests a character, it would get **eof** and subsequent requests would fail, because both the buffer and the file would be empty.

Recognition of tokens/Transition diagrams

As an intermediary step in the construction of a lexical analyser, we introduce another concept, called **transition diagram**

Transition diagrams depict the actions that take place when a lexer is called by a parser to get the next token.]

States in a transition diagram are drawn as circles. Some states have double circles, with or without a *. States are connected by arrows, called **edges**, each one carrying an input character as **label**, or the special label *other*.



Recognition of tokens/Transition diagrams (cont)

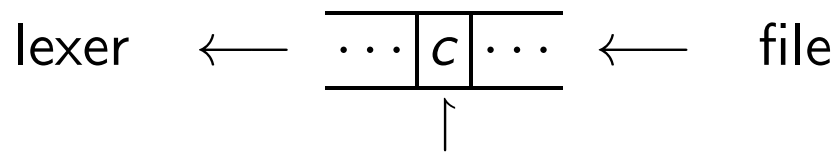
Double-circled states are called **final states**. The special arrow which do not connect two states points to the **initial state**.

A state in the transition diagram corresponds to the state of the input buffer, i.e., its contents and the output pointer at a given moment.

At the initial state, the buffer contains at least one character.

If the only one remaining character is **eof**, the lexer returns a special token **$** to the parser and stops.

Assume the character c is pointed by $\restriction$ in the input buffer and that c is not **eof**:



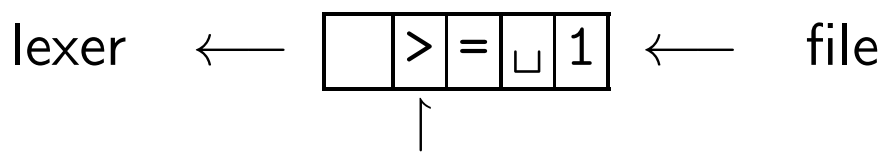
Recognition of tokens/Transition diagrams and buffering

When the parser requests a token, if an edge to state s has a label with character c, then the current state in the transition diagram becomes s and c is removed from the buffer.

This is repeated until a final state is reached or we get stuck.

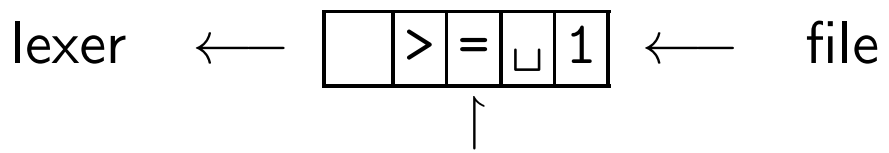
If a final state is reached, it means the lexer recognised a token — which is in turn returned to the parser. Otherwise a lexical error occurred.

Let us consider again the diagram page 94. Assume the initial input buffer is

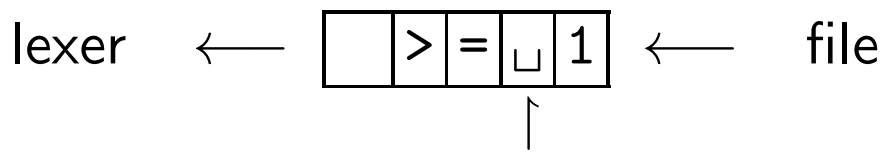


Recognition of tokens/Transition diagrams and buffering (cont)

From the initial state 1 to state number 2 there is an arrow with the label >. Because this label is present at the output position of the buffer, we can change the diagram state to 2 and remove < from the buffer, which becomes



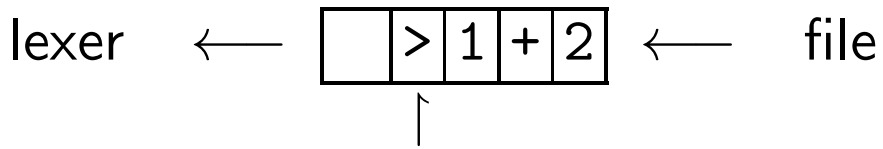
From state 2 to state 3 there is an arrow with label =, so we remove it:



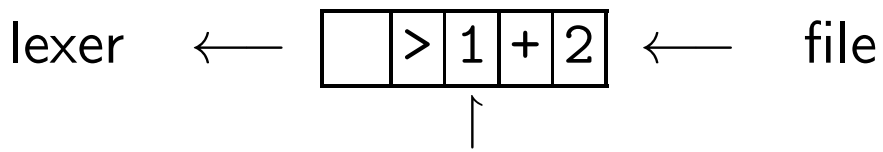
and we move to state 3. Since state 3 is a final state, we are done: we recognised the token **relop**⟨>=⟩.

Recognition of tokens/Transition diagrams and buffering (cont)

Imagine now the input buffer is



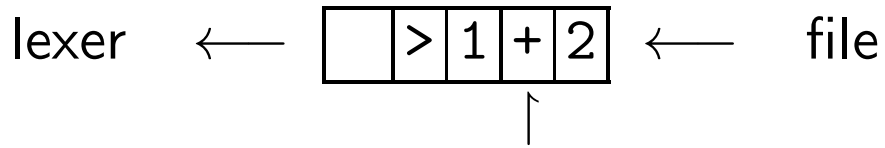
In this case, we will move from the initial state to state 2:



We cannot use the edge with label =. But we can use the one with "*other*". Indeed, *the "other" label refers to any character that is not indicated by any of the edges leaving the state.*

Recognition of tokens/Transition diagrams and buffering (cont)

So we move to state 4, the input buffer becomes

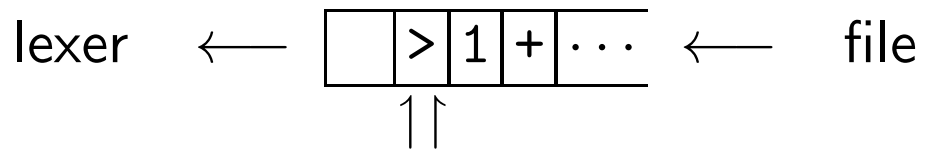


and the lexer emits the token **relop**$\langle$>$\rangle$.

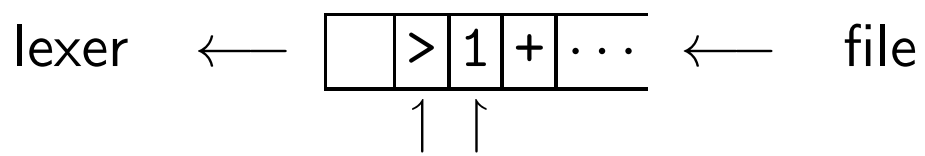
But there is a problem here: if the parser requests another token, we have to start again with this buffer but we already skipped the character $\boxed{1}$ and we forgot where the recognised lexeme starts...

Recognition of tokens/Transition diagrams and buffering (cont)

The idea is to use another arrow to mark the starting position when we try to recognise a token. Let $\uparrow$ be this new pointer. Then the initial buffer of our previous example would be depicted as



When the lexer reads the next available character, the pointer $\uparrow$ is shifted to the right of one position.

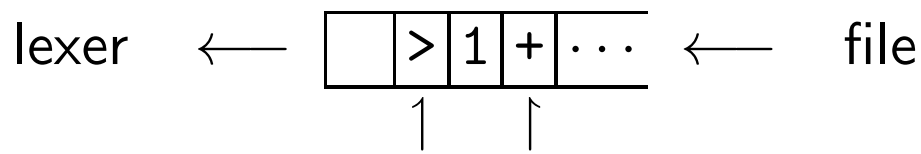


We are now at state 2 and the current character, i.e., pointed by $\uparrow$, is 1.

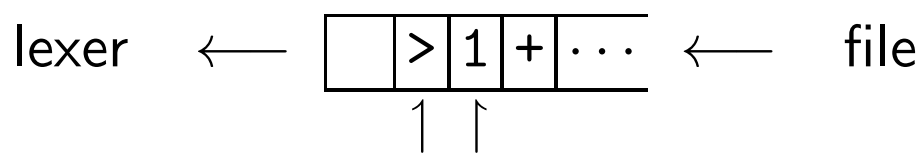
Recognition of tokens/Transition diagrams and buffering (cont)

The only way to continue is to go to state 4, using the special label *other*.

We shift the pointer of the secondary buffer to the right and, since it points to the last position, we input one character from the primary buffer:



State 4 is a final state a bit special: it is marked with *. This means that before emitting the recognised lexeme we have to shift the current pointer by one position *to the left*:



Recognition of tokens/Transition diagrams and buffering (cont)

This allows to recover the character $\boxed{1}$ *as current character.*

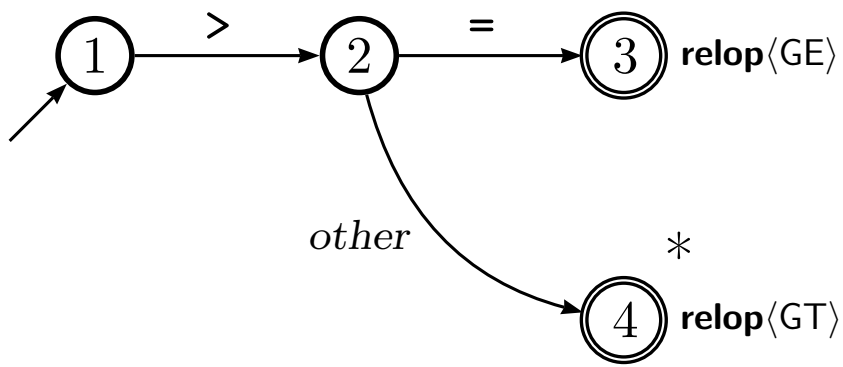
Moreover, the recognised lexeme now always starts at the $\uparrow$ pointer and ends one position before the $\restriction$. So, here, the lexer outputs the lexeme >.

Recognition of tokens/Transition diagrams (resumed)

Actually, we can complete our token specification by adding some extra information that are useful for the recognition process (as we just described).

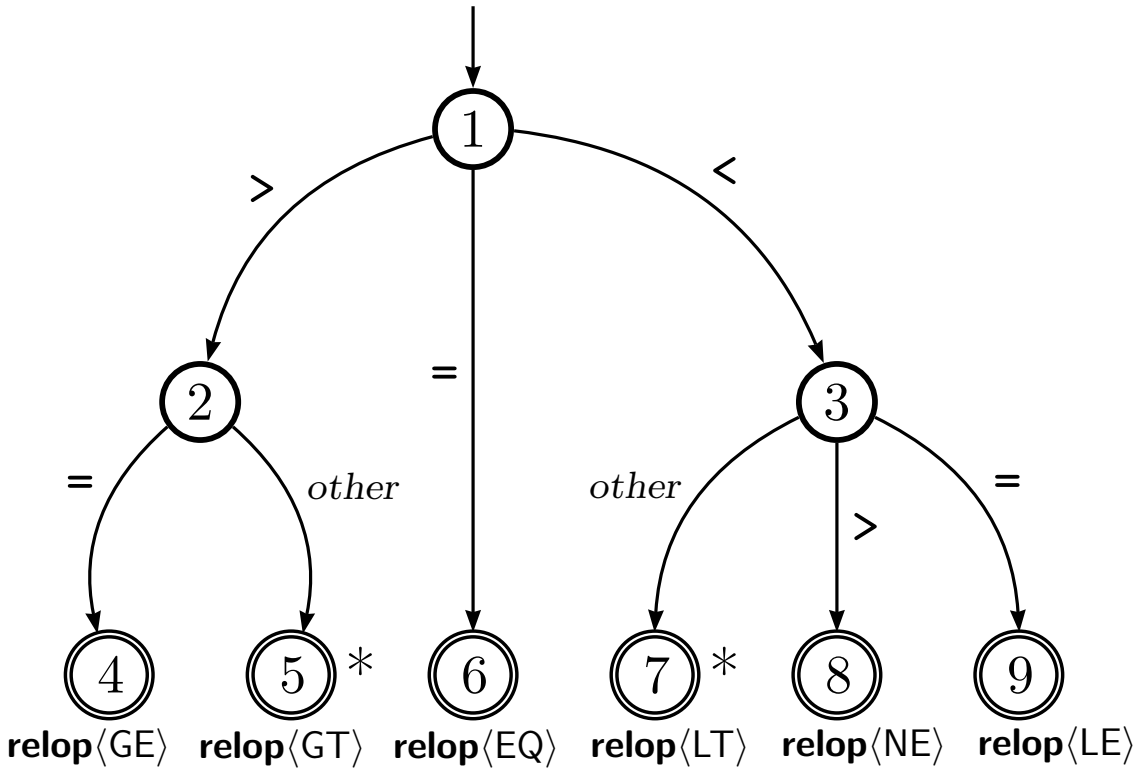
First, it is convenient for some tokens, like **relop** not to carry the lexeme verbatim, but a symbolic name instead, which is independent of the actual size of the lexeme. For instance, we shall write **relop**⟨GT⟩ instead of **relop**⟨>⟩.

Second, it is useful to write the recognised token and the lexeme close to the final state in the transition diagram itself. Consider



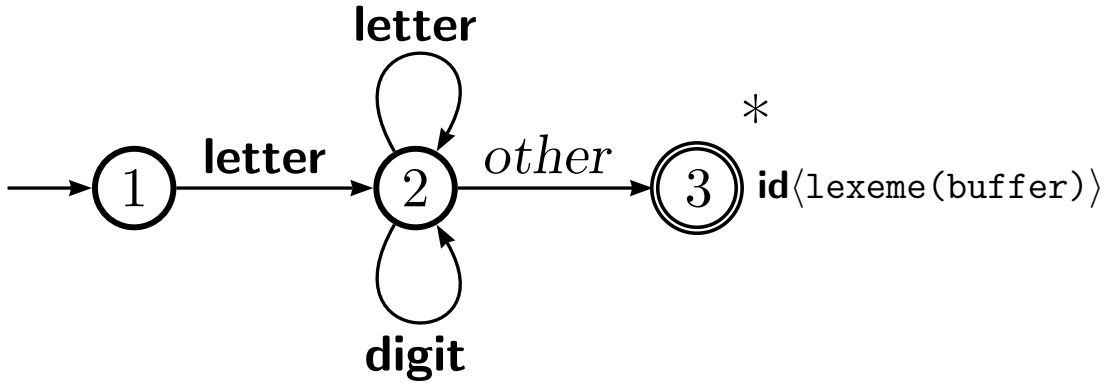
Recognition of tokens/Transition diagrams (cont)

Now let us give the transition diagram for recognising the token **relop** completely. Note that the previous diagram is a part of this one.



Recognition of tokens/Identifiers and longest prefix match

A transition diagram for specifying identifiers is



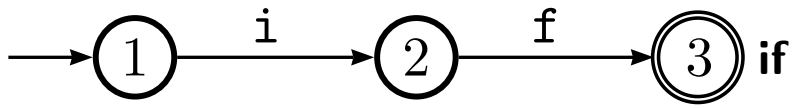
`lexeme` is a function call which returns the recognised lexeme (as found in the `buffer`)

The *other* label on the last step to final state force the identifier to be of *maximal length*. For instance, given `counter+1`, the lexer will recognise `counter` as identifier and not just `count`. This is called **the longest prefix** property.

Recognition of tokens/Keywords

Since keywords are sequences of letters, they are exceptions to the rule that a sequence of letters and digits starting with a letter is an identifier.

One solution for specifying keywords is to use dedicated transition diagrams, one for each keyword. For example, the **if** keyword is simply specified as



If one keyword diagram succeeds, i.e., the lexer reaches a final state, then the corresponding keyword is transmitted to the parser; otherwise, another keyword diagram is tried after shifting the current pointer $\upharpoonright$ in the input buffer back to the starting position, i.e pointed by $\restriction$.

Recognition of tokens/Keywords (cont)

There is a problem, though. Consider the Objective Caml language, where there are two keywords **fun** and **function**.

If the diagram of **fun** is tried successfully on the input `function` and then the diagram for identifiers, the lexer outputs the lexemes **fun** and **id**⟨`ction`⟩ instead of one keyword **function**...

As for identifiers, we want the longest prefix property to hold for keywords too and this is simply achieved by *ordering the transition diagrams*. For example, the diagram of **function** must be tried before the one for **fun** because **fun** is a prefix of **function**.

This strategy implies that the diagram for the identifiers (given page 105) must appear *after* the diagrams for the keywords.

Recognition of tokens/Keywords (cont)

There are still several drawbacks with this technique, though.

The first problem is that if we indeed have the longest prefix property among keywords, it does not hold with respect to the identifiers.

For instance, `iff` would lead to the keyword **if** and the identifier `f`, instead of the (longest and sole) identifier `iff`.

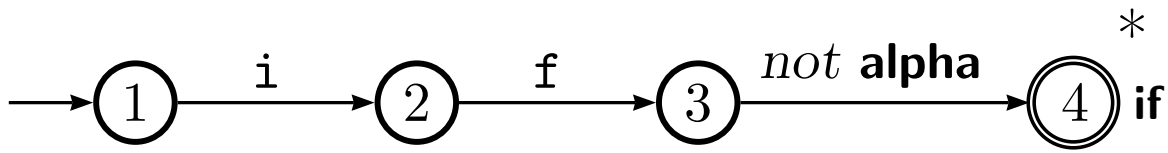
This can be remedied by forcing the keyword diagram to recognise a keyword and not an identifier. This is done by failing if the keyword is followed by a letter or a digit (remember we try the longest keywords first, otherwise we would miss some keywords — the ones which have prefix keywords).

Recognition of tokens/Keywords (cont)

The way to specify this is to use a special label *not* such as *not c* denotes the set of characters which are *not c*.

Actually, the special label *other* can always be represented using this *not* label because *other* means "not the others labels."

Therefore, the completed **if** transition diagram would be



where **alpha** (which stands for "alpha-numerical") is defined by the following regular definition:

$$\textbf{alpha} \rightarrow \textbf{letter} \mid \textbf{digit}$$

Recognition of tokens/Keywords (cont)

The second problem with this approach is that we have to create a transition diagram for each keyword and a state for each of their letters.

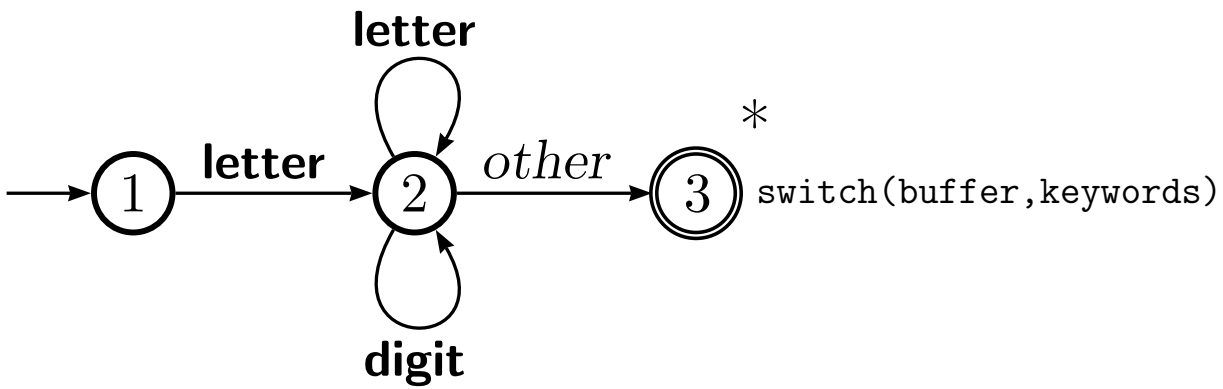
In real programming languages, this means that we get hundreds of states only for the keywords...

This problem can be avoided if we change our technique and give up the specification of keywords with transition diagrams.

Recognition of tokens/Keywords (cont)

Since keywords are a strict subset of identifiers, let us use only the identifier diagram but *we change the action at the final state*, i.e., instead of always returning a **id** token, we make some computations first to decide whether it is either a keyword or an identifier.

Let us call `switch` the function which makes this decision based on the buffer (equivalently, the current diagram state) and a **table of keywords**. We specify



Recognition of tokens/Keywords (cont)

The table of keywords is a two-column table whose first column (the entry) contains the keyword lexemes and the second column the corresponding token:

Keywords	
Lexeme	Token
`if`	**if**
`then`	**then**
`else`	**else**

Recognition of tokens/Keywords (cont)

Let us write the code for `switch` in the following pseudo-language:

$\textsc{Switch}(\mathit{buffer}, \mathit{keywords})$

$\quad \mathit{str} \leftarrow \textsc{Lexeme}(\mathit{buffer})$
$\quad \textbf{if } \mathit{str} \in \mathcal{D}(\mathit{keywords})$
$\quad\quad \textbf{then } \textsc{Switch} \leftarrow \mathit{keywords}[\mathit{str}]$
$\quad\quad \textbf{else } \textsc{Switch} \leftarrow \textbf{id}\langle \mathit{str} \rangle$

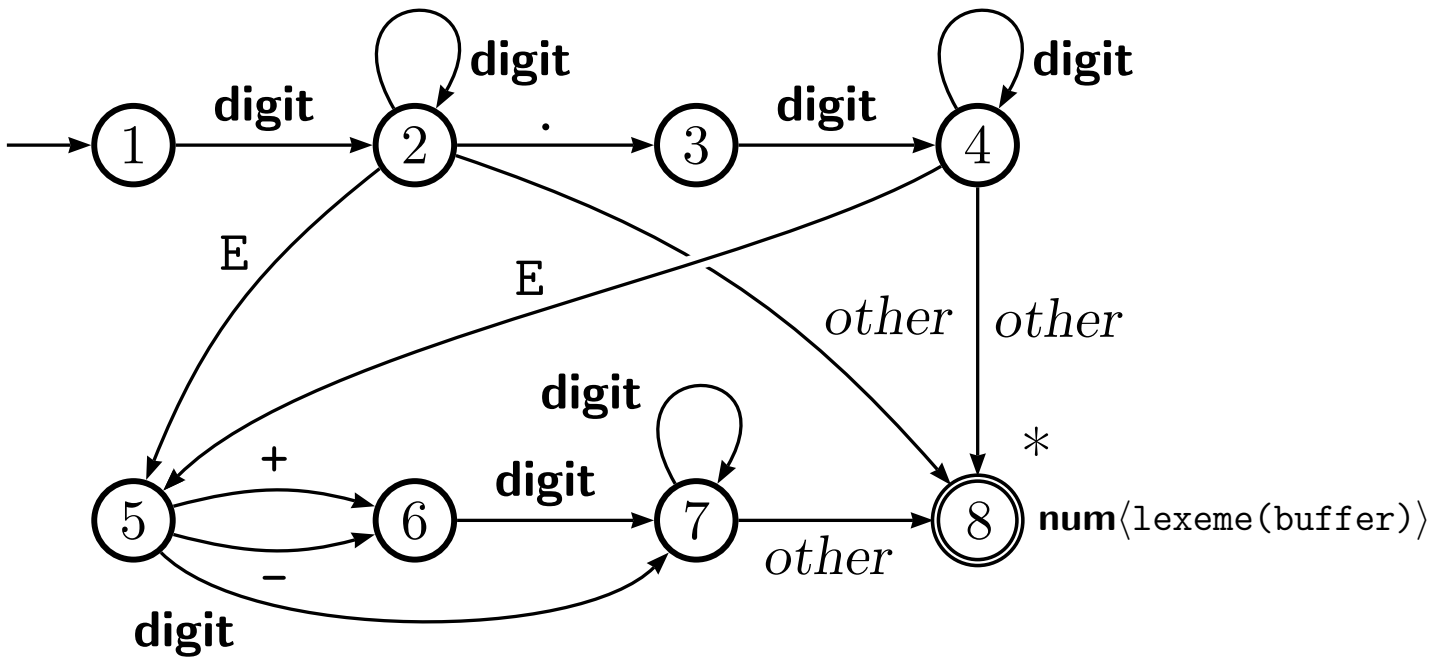
Function names are in uppercase, like $\textsc{Lexeme}$. Writing $x \leftarrow a$ means that we **assign** the value of expression a to the variable x. Then the value of x is the value of a. The value $\mathcal{D}(t)$ is the first column of table t. The value $t[e]$ is the value corresponding to e in table t. $\textsc{Switch}$ is also used as a special variable whose value becomes the result of the function $\textsc{Switch}$ when it finishes.

Recognition of tokens/Numbers

Let us consider now the numbers as specified by the regular definition

$$\textbf{num} \rightarrow \textbf{digit}^{+}\ (\texttt{.}\ \ \textbf{digit}^{+})?\ (\texttt{E}\ (\texttt{+} \mid \texttt{-})?\ \textbf{digit}^{+})?$$

and propose a transition diagram as an intermediary step to their recognition:

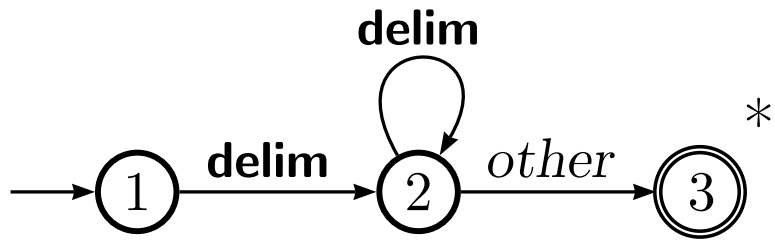


Recognition of tokens/White spaces

The only remaining issue concerns white spaces as specified by the regular definition

$$\textbf{white\_space} \rightarrow \textbf{delim}^{+}$$

which is equivalent to the transition diagram



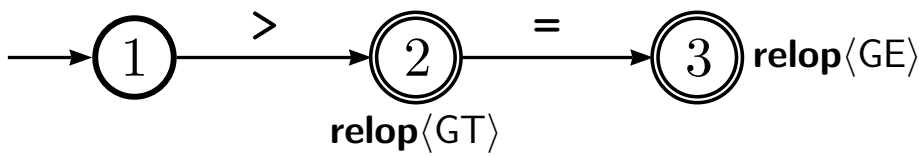
The specificity of this diagram is that there is no action associated to the final state: no token is emitted.

Recognition of tokens/Simplified

There is a simple away to reduce the size of the diagrams used to specify the tokens while retaining the longest prefix property: allow to pass through several final states.

This way, we can actually also get rid of the * marker on final states.

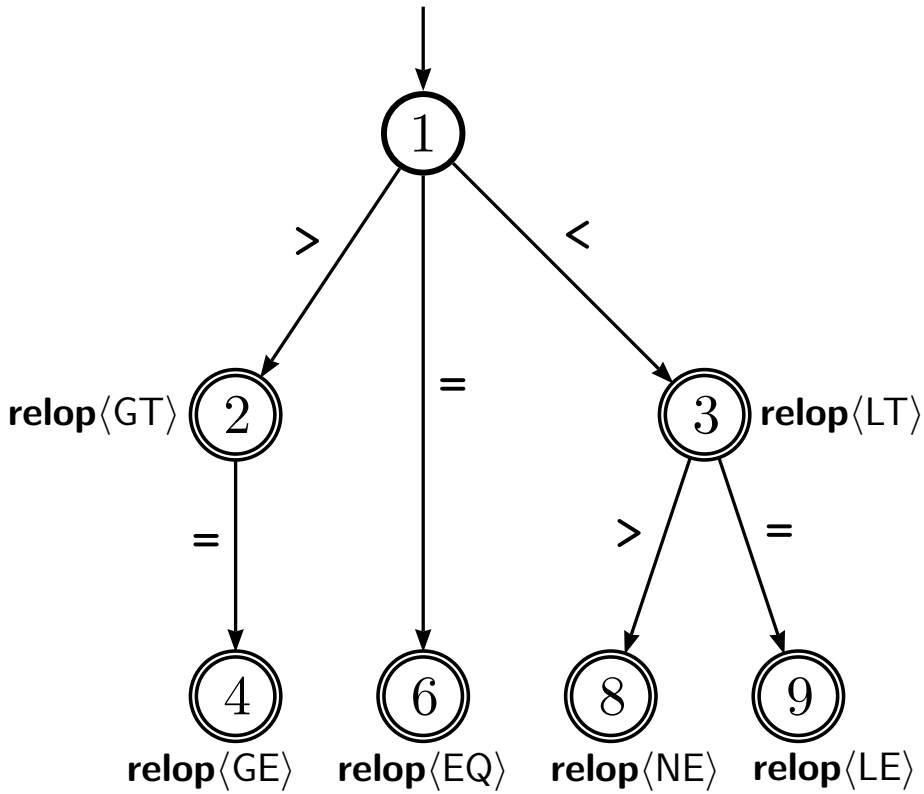
Coming back to the first example page 103, we would simply write:



But we have to change the recognition process a little bit here in order to keep the longest prefix match: we do not want to stop at state 2 if we could recognise >=.

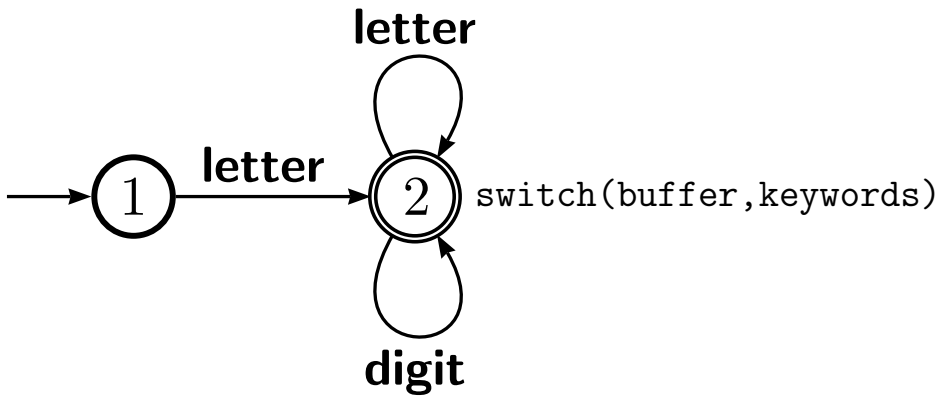
Recognition of tokens/Simplified/Comparisons

The simplified complete version with respect to the one given page 104 is



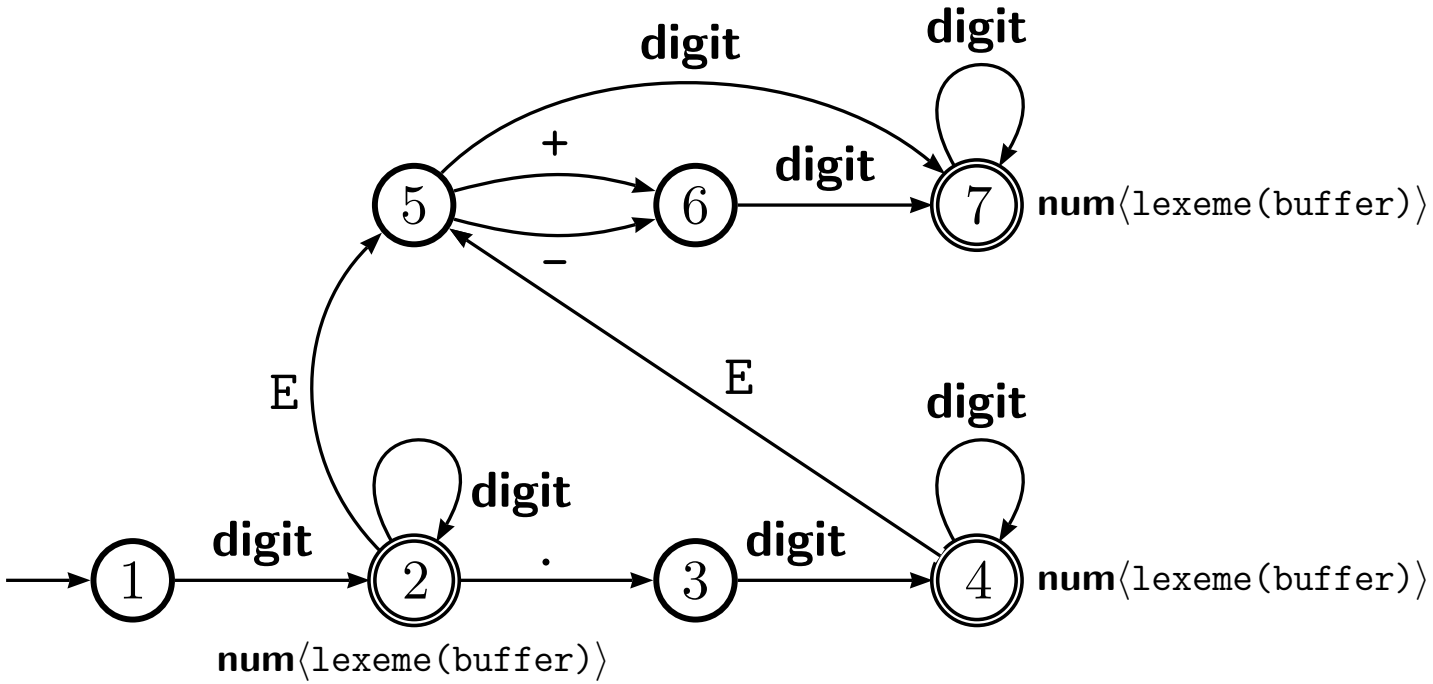
Recognition of tokens/Simplified/Identifiers

The transition diagram for specifying identifiers *and* keywords looks now like



Recognition of tokens/Simplified/Numbers

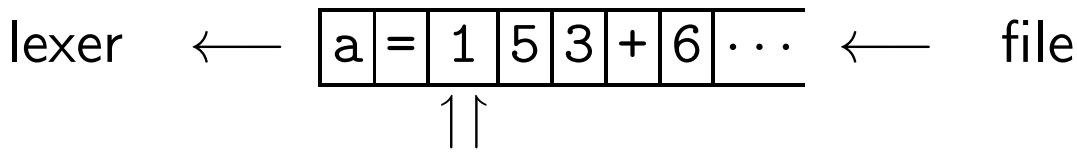
The transition diagram for specifying numbers is simpler now:



Recognition of tokens/Simplified/Interpretation

How does we interpret these new transition diagrams, where the final states may have out-going edges (and the initial state have incoming edges)?

For example, let us consider the recognition of a number:



As usual, if there is a label of an edge going out of the current state which matches the current character in the buffer, the $\restriction$ pointer is shifted to the right of one position.

Recognition of tokens/Simplified/Interpretation (cont)

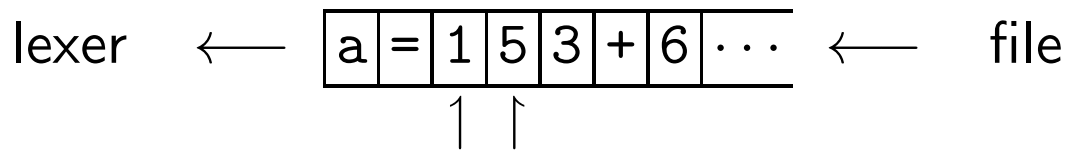
The new feature here is about final states. When the current state is final

1. the current position in the buffer is pointed to with a new pointer $\Uparrow$;
2. if there is an out-going edge which carries a matching character, we try to recognise a longer lexeme;
   - 2.1 if we fail, i.e., if we cannot go further in the diagram and the current state is not final, then we shift back the current pointer $\restriction$ to the position pointed by $\Uparrow$
   - 2.2 and return the then-recognised token and lexeme.
3. if not, we return the recognised token and lexeme associated to the current final state.

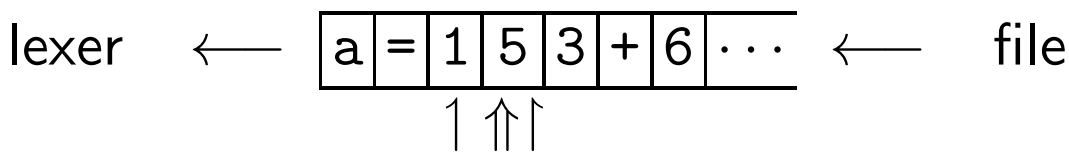
Recognition of tokens/Simplified/Example

Following our example of number recognition:

- The label **digit** matches the current character in the buffer, i.e., the one pointed by $\restriction$, so we move to state 2 and we shift right by one the pointer $\restriction$.

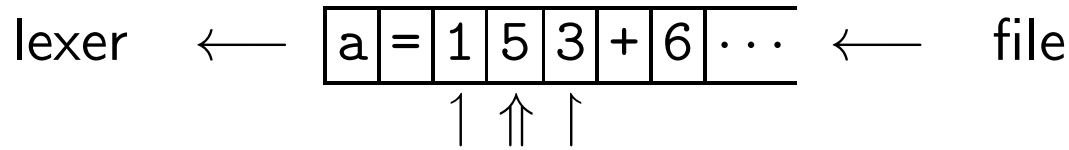


- The state 2 is final, so we set the $\Uparrow$ pointer to the current position in the buffer

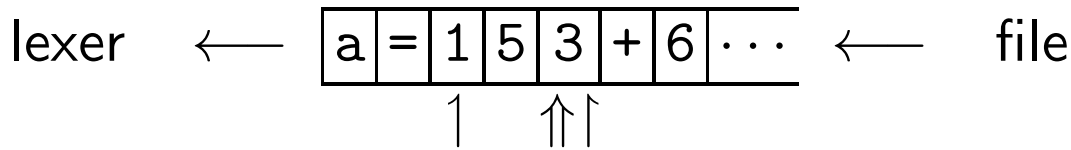


Recognition of tokens/Simplified/Example (cont)

- We shift right by one the current pointer and stay in state 2 because the matching edge is a loop (notice that we did not stop here).

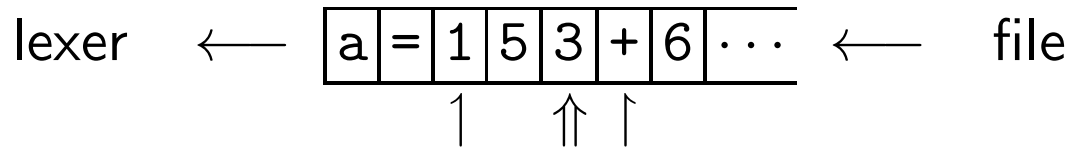


- The state 2 is final so we set the ⇑ to point to the current position:

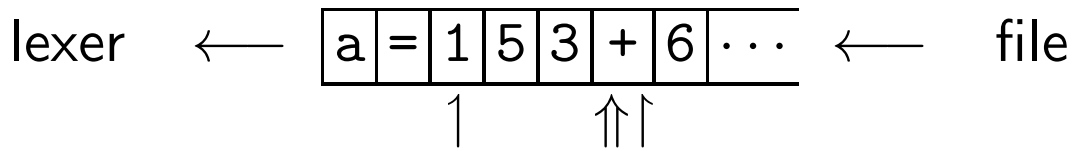


Recognition of tokens/Simplified/Example (cont)

- The **digit** label of the loop matches again the current character (here 3), so we shift right by one the current pointer.



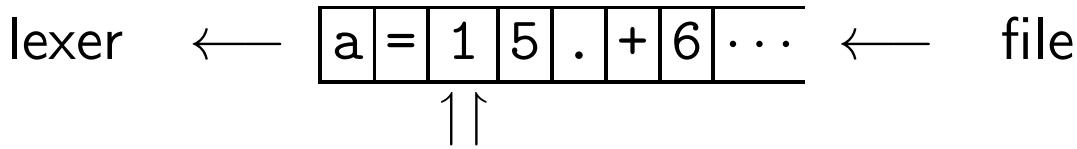
- Because state 2 is final we set the ⇑ to the current pointer ↾:



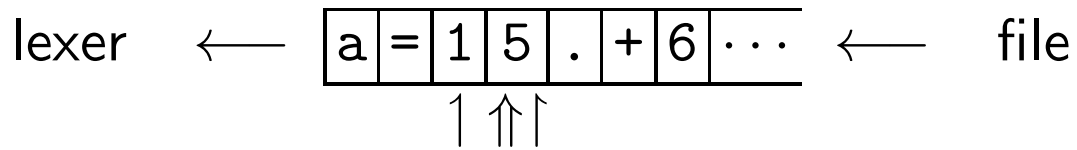
- State 2 is a final state, so it means that we succeeded in recognising the token associated with state 2: **num**⟨`lexeme(buffer)`⟩, whose lexeme is between ⇂ included and ↾ excluded, i.e., `153`.

Recognition of tokens/Simplified/Example (cont)

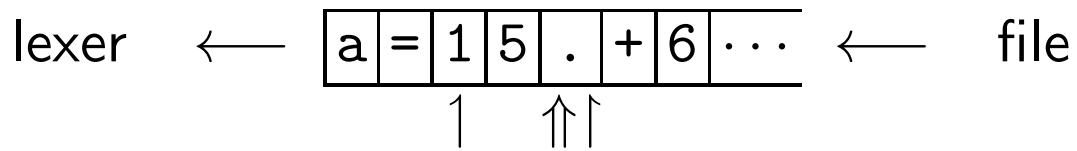
Let us consider the following initial buffer:



Character 1 is read and we arrive at state 2 with the following situation:

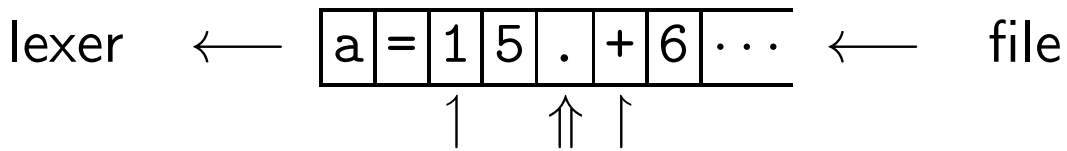


Then 5 is read and we arrive again at state 2 but with a different situation:



Recognition of tokens/Simplified/Example (cont)

The label on the edge from state 2 to 3 matches `.` so we move to state 3, shift by one the current pointer in the buffer:



Now we are stuck at state 3. Because this is not a final state, we should fail, i.e.,report a lexical error, but because the ⇑ has been set (i.e., we met a final state), we shift the current pointer back to the position of ⇑ and return the corresponding lexeme 15: