بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

We have, in the previous sessions, seen how we can convert a language description written as a regular expression into an efficiently executable representation (a DFA). What we want is something more: A program that does lexical analysis, i.e., a lexer:

☞ *A lexer has to distinguish between several different types of tokens, e.g., numbers, variables and keywords. Each of these are described by its own regular expression.*

☞ *A lexer does not check if its entire input is included in the languages defined by the regular expressions. Instead, it has to cut the input into pieces (tokens), each of which is included in one (or more) of the languages.*

☞ *If there are several ways to split the input into legal tokens, the lexer has to decide which of these it should use.*

A program that takes a set of token definitions (each consisting of a regular expression and a token name) and generates a lexer is called a lexer generator.

مرور مجدد مراحلی که باید طی شوند:

(1) Construct NFAs $N_1, N_2, \ldots, N_n$ for each of $r_1, r_2, \ldots, r_n$.
(2) Mark the accepting states of the NFAs by the name of the tokens they accept.
(3) Combine the NFAs to a single NFA by adding a new starting state which has ε-transitions to each of the starting states of the NFAs.
(4) Convert the combined NFA to a DFA.
(5) Each accepting state of the DFA consists of a set of NFA states, at least one of which is an accepting state which we marked by token type in step 2. These marks are used to mark the accepting states of the DFA, so each of these will indicate all the token types it accepts.

یادآوری: ترتیب لیست کردن عبارات منظم مهم است

it is common to let the lexer choose according to a prioritized list. Normally, the order in which tokens are defined in the input to the lexer generator indicates priority (earlier defined tokens take precedence over later defined tokens). Hence, keywords are usually defined before variable names, which means that, for example, the string "`if`" is recognized as a keyword and not a variable name. When an accepting state in a DFA contains accepting NFA states with different marks, the mark corresponding to the highest priority (earliest defined) token is used. Hence, we can simply erase all but one mark from each accepting state. This is a very simple and effective solution to the problem.

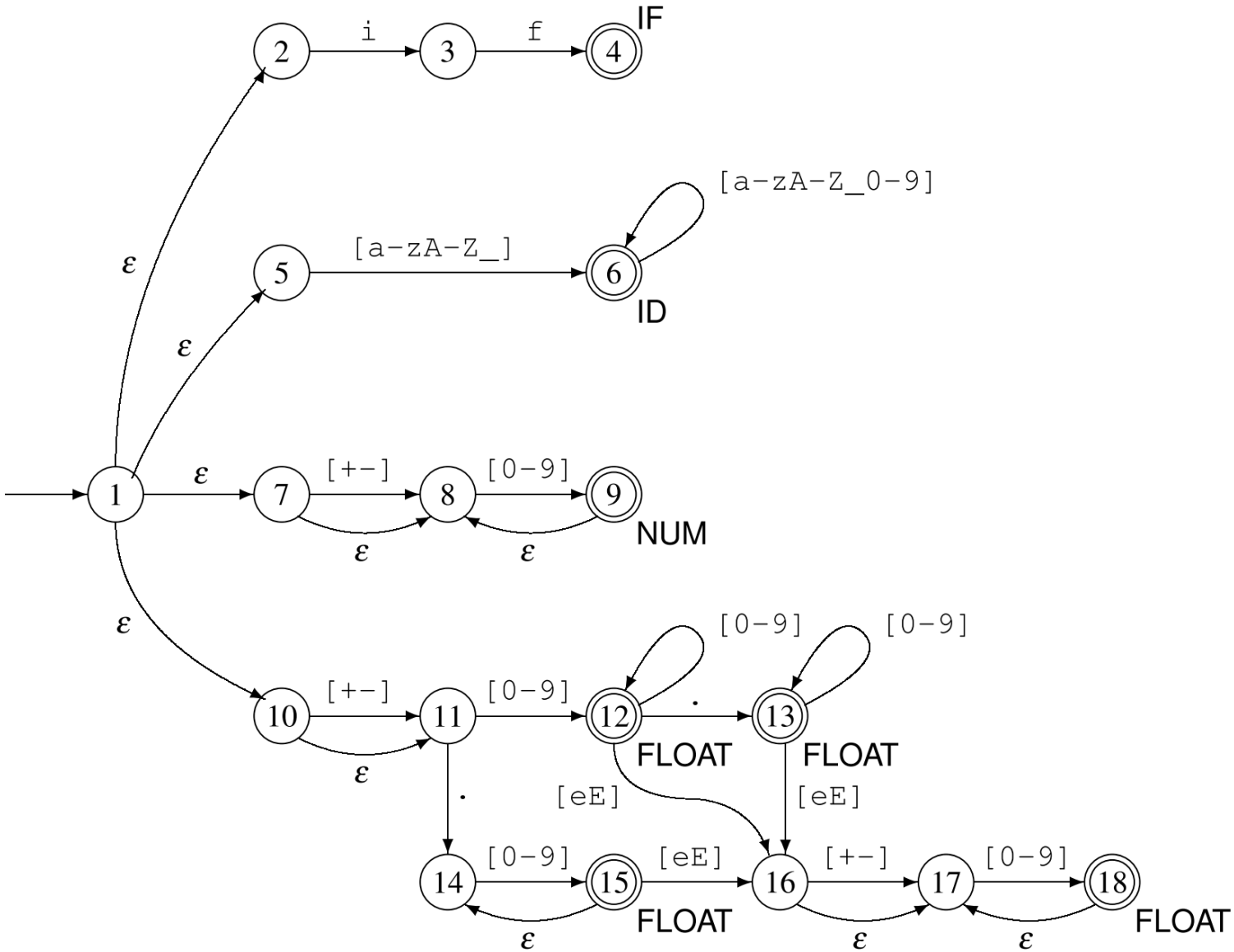
یادآوری: طولانی‌ترین پیشوند برای جدا شدن و معرفی توکن جدا می‌شود

The lexer must cut the input into tokens. This may be done in several ways. For example, the string `if17` *can be split in many different ways:*

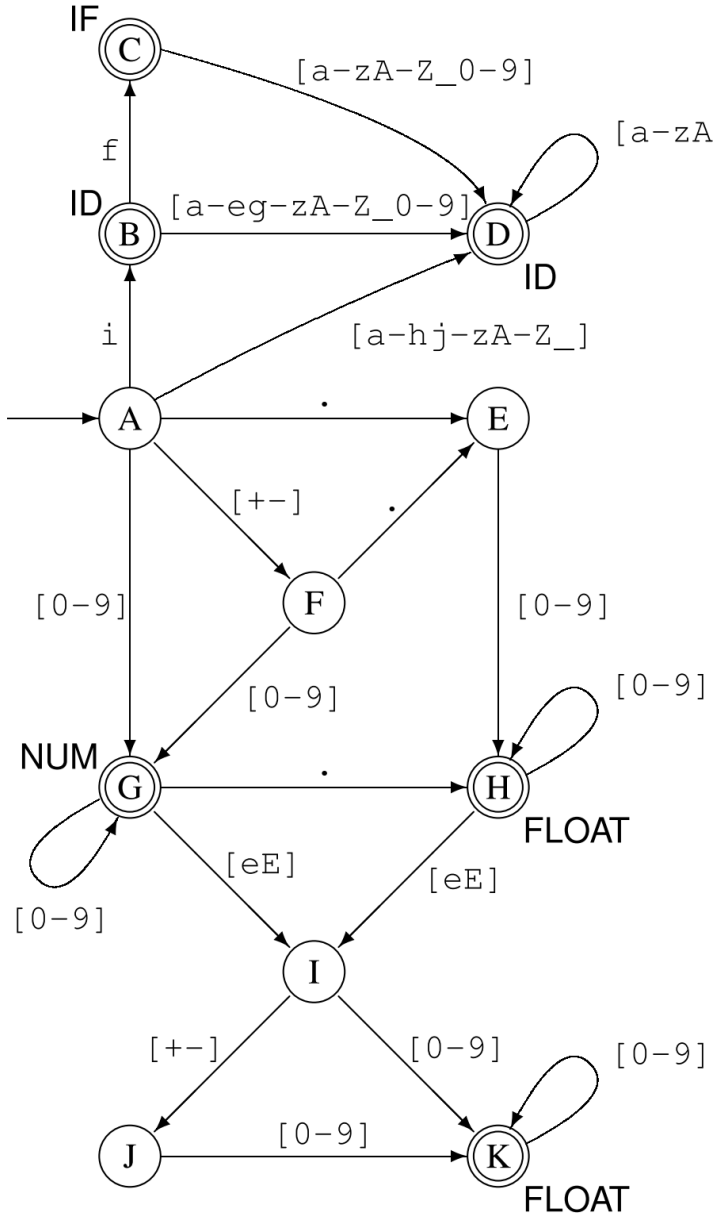
- *As one token, which is the variable name* `if17`.
- *As the variable name* `if1` *followed by the number* `7`.
- *As the keyword* `if` *followed by the number* `17`.
- *As the keyword* `if` *followed by the numbers* `1` *and* `7`.
- *As the variable name* `i` *followed by the variable name* `f17`.
- *And several more.*

A common convention is that it is the longest prefix of the input that matches any token which will be chosen. Hence, the first of the above possible splittings of if17 will be chosen. Note that the principle of the longest match takes precedence over the order of definition of tokens, so even though the string starts with the keyword if, which has higher priority than variable names, the variable name is chosen because it is longer. Modern languages like C, Java or Haskell follow this convention, and so do most lexer generators.

Combined NFA for several tokens



Combined DFA for several tokens



Transition table compression

Transition tables are not arbitrary matrices; they exhibit a lot of structure. For one thing, when a token is being recognized, only very few characters will at any point continue that token; so most transitions lead to the empty set, and most entries in the table are empty. Such low-density transition tables are called sparse. Densities (fill ratios) of 5% or less are not unusual. For another, the states resulting from a move over a character `Ch` *all contain exclusively items that indicate that a* `Ch` *has just been recognized, and there are not too many of these. So columns tend to contain only a few different values which, in addition, do not normally occur in other columns. The idea suggests itself to exploit this redundancy to compress the transition table.*

چرا فشرده‌سازی معمول و مرسوم خیلی کارامد نیست؟

The first idea that may occur to the reader is to apply compression algorithms of the Huffman or Lempel-Ziv variety to the transition table, in the same way they are used in well-known file compression programs. No doubt they would do an excellent job on the table, but they miss the point: the compressed table must still allow cheap access to `NextState[State, Ch]`*, and digging up that value from a Lempel-Ziv compressed table would be most uncomfortable! There is a rich collection of algorithms for compressing tables while leaving the accessibility intact, but none is optimal and each strikes a different compromise. As a result, it is an attractive field for the inventive mind.*

Table compression by graph coloring

There is a simple technique to compress transition tables, which works better for large tables when used in combination with a bitmap to check for empty entries. In this approach, we select a subset S from the total set of strips, such that we can combine all strips in S without displacement and without conflict: they can just be positioned all at the same location. This means that the non-empty positions in each strip in S avoid the non-empty positions in all the other strips or have identical values in those positions.

The sets are determined by first constructing and then coloring a so-called interference graph, a graph in which each strip is a node and in which there is an edge between each pair of strips that cannot coexist in a subset because of conflicts.

The relation of graph coloring to our subset selection problem is obvious: the strips correspond to nodes, the colors correspond to the subsets, and the edges prevent conflicting strips from ending up in the same subset.

The important point is that there are very good heuristic algorithms to almost always find the minimal number of colors; the problem of always finding the exact minimal number of colors is NP-complete.

A transition table (a) and its compressed form packed by graph coloring (b)

	w	x	y	z
0	1	2	–	–
1	3	–	4	–
2	1	–	–	6
3	–	2	–	–
4	–	–	–	5
5	1	–	4	–
6	–	7	–	–
7	–	–	–	–

(a)

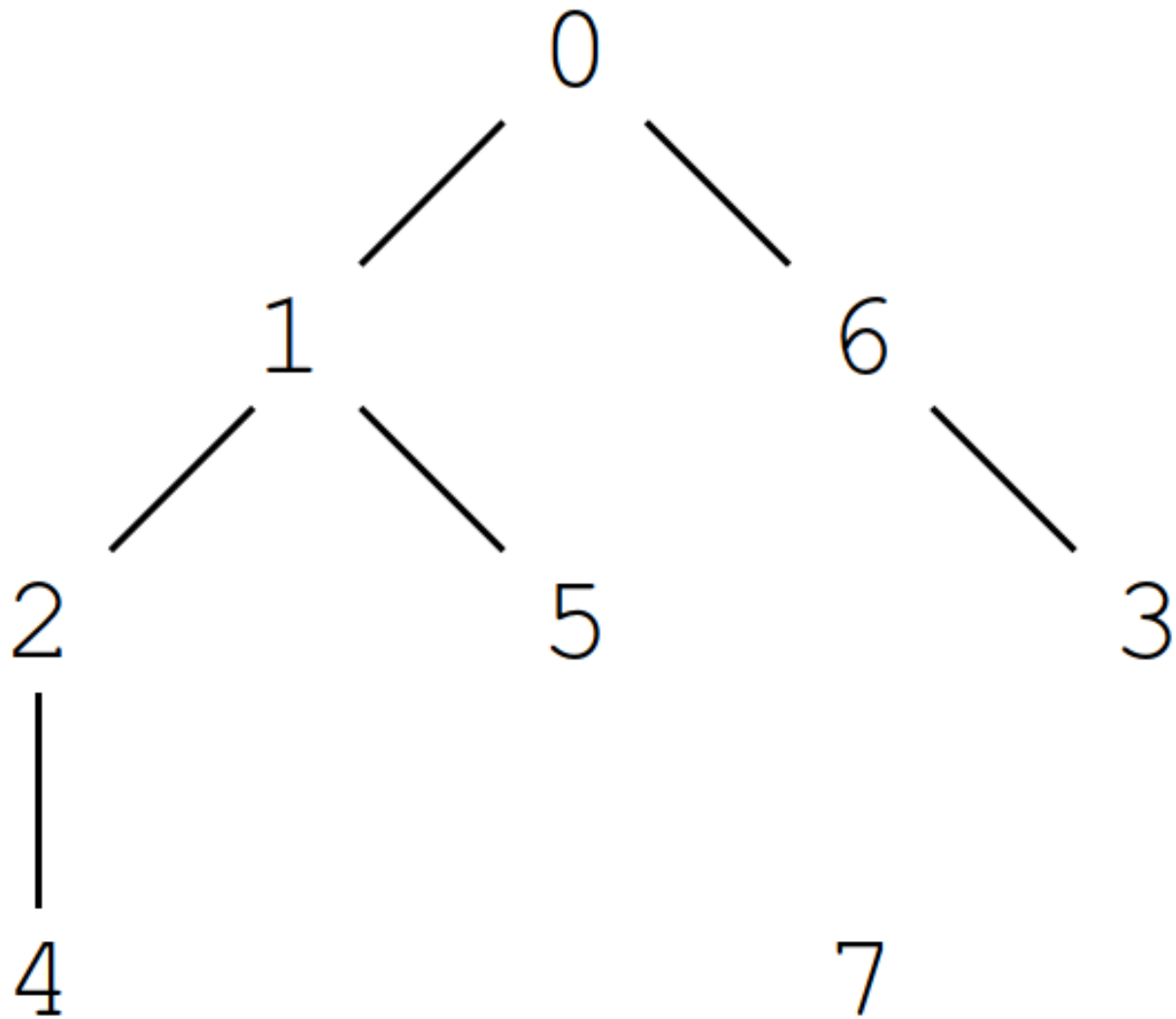
	wx	y	z
0	1	2	– –
1			
2	1 –	–	6
3	– 2	–	–
4			
5	1 –	4	–
6			
7			
	1 2 4 6		

	w	x	y	z
0				
1	3	–	4	–
2				
3				
4	–	–	–	5
5				
6	–	7	–	–
7	–	–	–	–
	3	7	4	5

(b)

Remark: *the so-called empty locations are not really empty but contain the number of the empty set.*

Interference graph for the automaton of the above transition table (a)



از ۶۴ بایت به ۳۶ بایت می‌رسیم

The cost is $8 \times 2 = 16$ ***bytes for the entries, plus 32 bits = 4 bytes for the bitmap (we use a bitmap to weed out all access to empty states), plus*** $8 \times 2 = 16$ ***bytes for the mapping from state to strip, totaling 36 bytes, against*** $32 \times 2 = 64$ ***bytes for the uncompressed matrix.***

دقت کنید که از یک بیت‌مپ با طول ۳۲، که همان ابعاد جدول اولیه است، برای مشخص کردن خانه‌های تهی و خانه‌های پر بهره می‌گیریم. این کار نیاز به ۳۲ بیت، یعنی چهار بایت، دارد.

A traditional lexical-analyzer generator—`Lex`

There is a family of software systems, including `Lex` ***and*** `Flex`***, that are lexical-analyzer generators. The user specifies the patterns for tokens using an extended regular-expression notation.*** `Lex` ***converts these expressions into a lexical analyzer that is essentially a DFA that recognizes any of the patterns.***

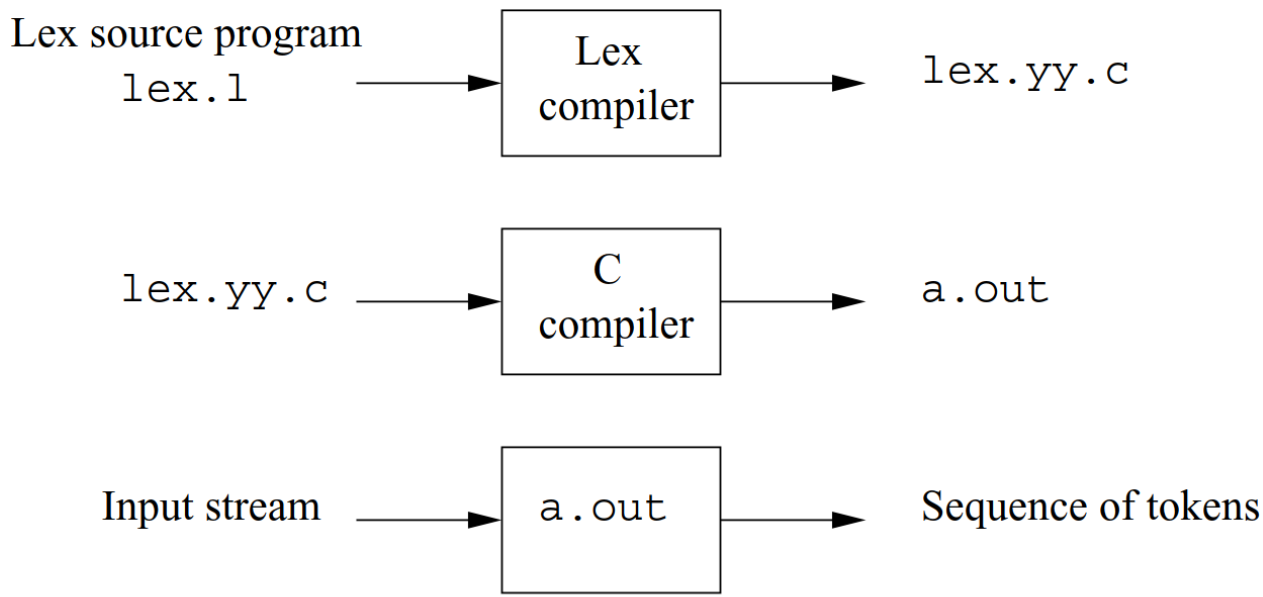


Figure 3.22: Creating a lexical analyzer with `Lex`

simpleExample.l

```
%{
#include <math.h>
%}

%%

a            {printf("token a\n");}
abb          {printf("token abb\n");}
a*b+         {printf("token a*b+\n");}
[ \t\n]+           /* eat up whitespace */
.           printf( "Unrecognized character: %s\n", yytext );

%%

int yywrap(){};
int main()
{
    yyin = stdin;
    yylex();
}
```

ابزار Flex در ویندوز هم در دسترس است: (Cygwin یا Mingw-w64)