Compiler Construction: Scanning

Diego Linares

Also known as **lexical analysis**, it reads the source code as a file of characters and divides it into **tokens**, which represent a unit of information. Examples:

- **Keywords:** `if`, which is fixed.
- **Identifiers:** user defined.
- **Special symbosl:** like arithmetic ones.

The methos of pattern recognition are those of **regular expressions and finite automata**. Since it is part of the compiler, it must be efficient to avoid overhead time.

I. The scanning process

The scanner loads tokens into logical units. Tokens are entitites defined as an enumerated type. They can fall into multiple categories as we saw before.
The string of characters represented by a token is the **string value/lexem**. They are kept track of in the symbol table.
A token has values associated called **attributes**. Like for example a number having the string value `"123"`, but also the actual numeric value. However this second one doesn't have to be computed immediately, because it can be computed from the string value.
All the attrributes are often saved in a class called **token record**. Another approach is to return the token value only and place the attributes in variables that can be accessed by the compiler.
Instead of processing all tokens at once, it uses a function called `getToken`, which returns the token and computes some attributes. For example in `a[index] = 4 + 2`, it would return first `a` and then the left bracket.

II. Regular expressions

It defines a certain set of strings called its **language**, or $L(r)$. It depends on the set of characters available (usually ASCII or a subset of it). The set is called the **alphabet** or Σ.
A regular indicate patterns. Some of the characters that have special meanings are called **metacharacters/metasymbols**. They usually are not part of the alphabet, but if they are, they use an **escape character** which is a metacharacter on itself.

A. Definition of Regular Expressions

Now we can see which languages are generated by each pattern, we can do it following the next steps.

1) Basic Regular Expressions: Single characters that match themselves, for example `a` matches the character a like: $L(a) = \{a\}$.
The empty string is denoted as ϵ so $L(\epsilon) = \{\epsilon\}$
The symbol that matches no string at all is ϕ, which is different from the previous one since it matches no strings at all. $L(\phi) = \{\}$.

2) Regular Expression Operations: Choice among alternatives $|$, concatenation (which is just putting the characters together), and repetition $*$.

3) Choice Among Alternatives: Being r, s regular expressions $r|s$ is the RE that matches any string matched by the two. It's the **union** of the languages:

$$L(r|s) = L(r) \cup L(s)$$

This can be extended to multiple alternatives. Long choices can be represented with dots.

4) Concatenation: Matches only strings that are concatenation of the two strings. For example:

$$L((a|b)c) = \{ac, bc\}$$

Given 2 sets of strings, it is basically all the possible concatenations of the first set, appended with the second set. It can also be extended to more than 2 RE.

5) Repetition: Also called **Kleene closure**. Accepts any finite concatenation of strings (each with matches the RE). For example: $a^* = \{\epsilon, a, aa, aaa, \cdots\}$. Given a set S of strings, the repetition can be writen as:

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

Another example:

$$L((a|bb)^*) = L(a|bb)^* = \{a, bb\}^* = \{\epsilon, a, bb, aa, bba, \ \cdots\}$$

6) Precedence of Operation and use of parentheses: Repetition has higher precedence over choice. So $a|b^* = a|(b^*)$. Concatenation is the second highest, and $|$ the lowest. So $a|bc^* = a|(b(c^*))$
To change precedence we use parentheses. Analogous to their use in arithmetic.

7) Names for regular expressions: For long regular expressions (for example, the one that matches all int numbers) we can write $digit^*$ where $digit = 0|1|2|\cdots|9$. This is **regular definition of name digit**. It becomes a metasymbol and it must be distinguished as such. A name can't be in its own definition.

B. Extensions to regular expressions

More efficient than just using the three operations we saw. New metasymbols added.

1) One or more repetitions: Similar to the r^* expression, but it disallows the ϵ string. For example, a binary number $(0|1)+$

2) Any character: Instead of writing each character of the alphabet as an alternative choice, you can just write .. For example, all strings with at least one b: $.^*b.^*$

3) A range of characters: Instead of the alternatives, we can use square brackets and a hyphen: $[A-Z]$. Must be careful of the character set being used since: $[A-z] \neq [A-Za-z]$ in the ASCII character set.

4) Any character not in a given set: We can use the symbol $\sim$ followed by the set, like: $\sim(a|b|c)$. In lex an alternative is the symbol ^.

5) Optional subexpressions: Instead of writing many similar alternatives, this allows for 0 or 1 copies of the expression to be included. For example an entire number might be: $signedNatural = (+|-)?natural$. Represented by the ?.

C. Regular Expressions for Programming Language Tokens

Tokens might fall into many subcategories. **Keywords** are fixed strings of alphabetic characters with special meaning (like `while`).
Special symbols, operators, assignments, etc.
Identifiers sequences of characters and numbers that begin with a letter.
Literals Basically constant values.

1) Numbers: Sequence of digits followed by a decimaal point and maybe an exponent.

$$nat = \ [0-9]+$$

$$signedNat = (+|-)?nat$$

$$number = signedNat(".\!"nat)?(EsignedNat)?$$

2) Reserved Words and Identifiers: A reserved word is simply an alternative choice between many fixed strings.
Identifiers start with a letter, and then contain letters and digits.

$$letter = [a-zA-Z]$$

$$digit = [0-9]$$

$$identifier = letter(letter|digit)^*$$

3) Comments: Usually ignored during scanning, so they must be recognized to be discarded. Their RE are easy to write, for example in Pascal (comments between brackets):

$$\{(\sim\})^*\}$$

Bit more difficult for delimiters that are longer than one character in length. So they are almost never used in practice. And in some language they might even be nested, so we need some counting method, or an *ad hoc* solution.

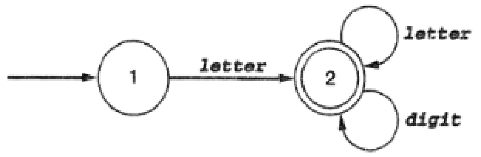
4) Ambiguity, white space, and lookahead: Some strings might match more than one RE, so we need rules to disambiguate them.
Usually if it matched a keyword and identifier, the first is preferred, since it is a **reserved word**. When it matches one token or several ones, the single token is preferred by **principle of longest substring**.
An issue with it is **token delimiters**, which imply that a longer string from there can't represent a token, like = in `x=temp`. Blanks, tabs and such are also considered as such. It is often defined as:

$$whitespace = (newline|blank|tab|comment)+$$

Languages that ignore whitespace are called **free format**, in comparison to those of fixed format or **offside rule**.
Delimeters are not prat of the token. **Lookahead** consists on, when encountering a delimeter, backing it up (putting it again on the input string, and deleting it from the token), or looking one character ahead. In most cases only done with single characters.
In the case of some languages more than one character is needed and buffers are used.
In FORTRAN, whitespaces are no delimiters, so they are ignored, and keywords can also be identifiers. Like in: `IF(IF.EQ.0)THENTHEN=1.0`, so the scanner must be able to backtrack to certain positions.

III. Finite automata

Mathematical model to describe particular machines, in this case, input strings, so they can build scanners. Can be constructed from RE. For example, the automata for an identifier can be:



Circles represent **states** and lines **transitions**. The start state is the one in which the recognition begins. The ones that declare success are **accepting states**.

A. Definition of DFA

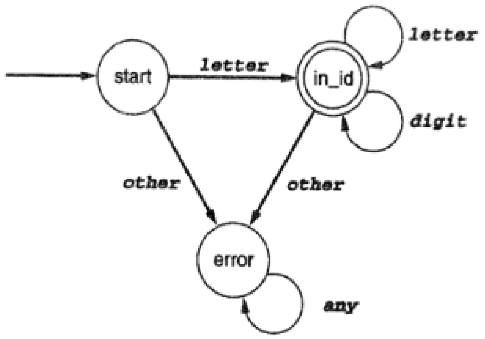
The next state is given by the current state and the current input character, the generalization is the **non-deterministic**.

A **DFA** (deterministic finite automaton) M consists of an alphabet Σ, a set of states S, a transition function $T\colon S \times \Sigma \rightarrow S$, a start state $s_0 \in S$, and a set of accepting states $A \subset S$. The language accepted by M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2 \ldots c_n$ with each $c_i \in \Sigma$ such that there exist states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$ with s_n an element of A (i.e., an accepting state).

Acceptance is the sequence of states:

$$s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), \cdots, s_n = T(s_{n-1}, c_n)$$

With s_n being an aaccepting state.
The diagram lets us customize the name of the states. And the transitions can be labeled with the name of the set of characters. Also in the diagram, missing transitions can be used to represent errors. So called **error translations** and they are assumed to always exist. Like in the previous diagram.

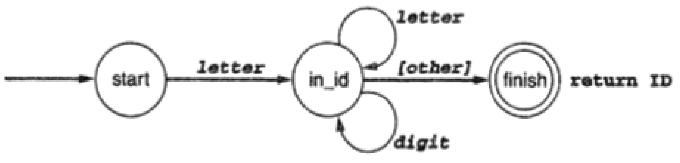


The error being of course non-accepting.

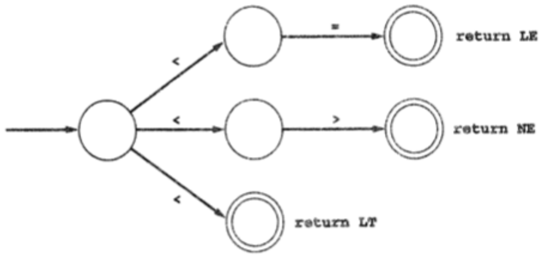
Note: it is helpful to write definitions for each set of characters before making a diagram.

B. Lookahead, backtracking, and NonDFA

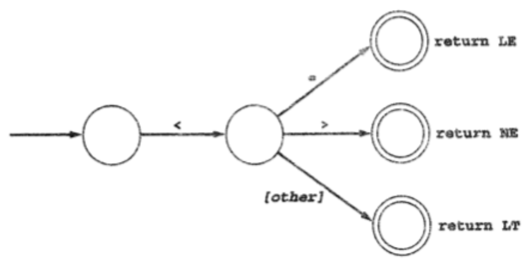
The mathematical definition doesn't explain what to do when an error occurs, or when reaching the accepting state, etc.
Usually a transition just means adding the character to the lexeme of the token. When reaching an error, we can either backtrack or generate an error token.
In reality, the previous diagram doesn't look like the proper behaviour of reading an identifier, it looks more like this.



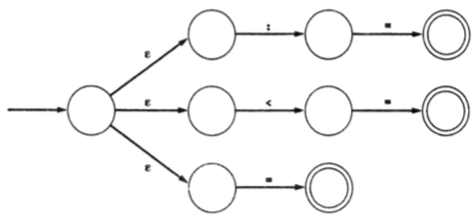
The square brackets represent that the character must be in *lookahead* and not consumed. The error begins the accepting as it means the scanner begins a new recognition after finishing that one (that's why there's no transitions).
It also follows the principle of longest substring.
In a typical programming language there are many tokens, each recognized by its own DFA, and they can be combined. However, must be done in a careful fashion.



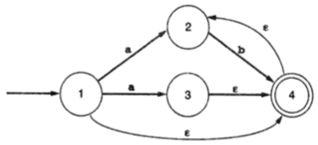
Instead, we must arrange it so that there is a unique transition to be made in each state, such as in the following diagram:



Combining all the identifiers DFA can become an enormous task. So non-DFA allows to include the case where more than one transition from a state may exist for a particular character. Through $\epsilon - transition$ which are done without consulting the input string. Or like the match of the empty string, they allow to preserve the original DFA, to combine them more easily.



In a more formal definition ϵ is added to the Σ of the DFA. And transitions are now to a set of states rather than just one. Thus the range of T is the power set of the set S of states $\wp(S)$.
Usually the accepted string is the regular one with the ϵ removed. The sequence of states are chosen from $T(s_0, c_1), \cdots, T(s_{n-1}, c_n)$ and it is not uniquely determined. It doesn't represent an algorithm but it can be simulated by one that backtracks over every nondeterministic choice.



The string **abb** can be accepted by either of the following sequences of transitions:

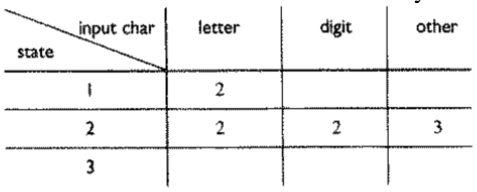
$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$$

$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\epsilon} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4$$

It can also be translated to a regular DFA.

C. Implementation of Finite Automaata in Code

The easiest way of doing it is through `if, else, while` statements. The position in code mantains the state implicitly. Works for small DFA. Downsides is that every DFA is different, and the complexity increases a lot with more states. This is called **hardwire** approach into the code. But it can also be be generalized by a data structure known as the **transition table**. Which are a 2 dimentional array of *states, char*.



state \ input char	letter	digit	other
1	2		
2	2	2	3
3			

Blank states are the same in a DFA, as they represent errors. The first state is considered the initial one, and an initial column can be added to clarify if the state is accepting or not. Now we can write the following code that accepts any DFA:

```
state := 1;
ch := next input character;
while not Accept[state] and not error(state) do
  newstate := T[state,ch];
  if Advance[state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept[state] then accept;
```

These are called **table driven** methods. Reduced code size, same code for many problems an easy to change. But the tables can become very large. Compression methods may be used but they make lookup slower.
An NFA program is similar but tarnsitions that haven't been

tried must be stored and backtracked on failure. Since backtrack is inefficient, we tend to convert them to DFA first.

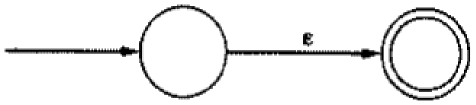
IV. From RE to DFA

RE helps as token descriptions, and then proceed through the construction of a DFA to a final scanner program. It is done through an intermediate algorithm that converts the RE to a NFA, and then another algorithm transforms it to DFA (doing it directly is complex).

A. From RE to NFA

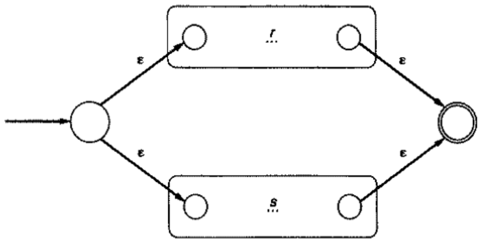
Thompson's construction glue's together the different machines that make the RE, through $\epsilon - transitions$. We exhibit a NFA for each basic RE, and then check how the operations that connect them can be achieved by connecting the NFAs.

1) Basic REs: Basic REs are a, ϵ, ϕ (though last one never really occurs). And their NFAs take this form:

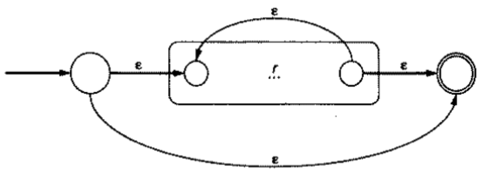


2) Concatenation: The accepting state of the first NFA is connected to the initial state of the second one through an ϵ. We can simplify this further by making them the same state if the original accepting one didn't have any traansitions.

3) Choice Among Alternatives: We create a new initial state and accepting state, that are connected to both REs start and accepting through ϵ.



4) Repetition: New states, start and accepting, connected to the old ones by epsilons. And the old final state connects to the old start one by one as well.



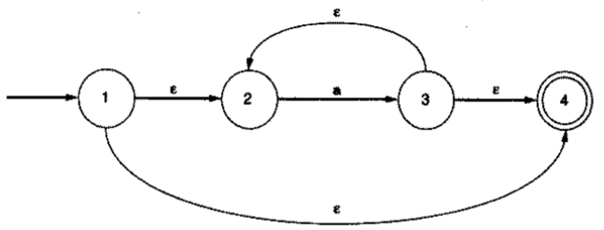
Some other rules to follow are:

1) Each state has at most 2 transition, and if they have 2, both have to be ϵ.
2) No states are deleted once constructed.
3) No transitions changed except for the addition of transitions from the accepting state.

B. From NFA to DFA

We need a method to eliminate $\epsilon - transitions$ and multiple tarnsitions from a state(s). This is done by $\epsilon - closure$ or all the states that can be reachable by those transitions from certain state. So the DFA has as its states, *sets of states* from the NFA. This algorithm is called **subset construction**.

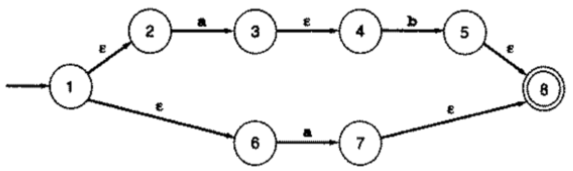
1) ϵ − Closure of a set of states: Set of states reachable by a series of zero or more $\epsilon - traansitions$, written as $\overline{s}$



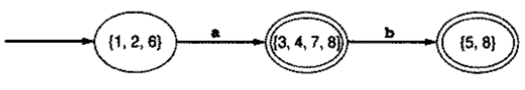
In this NFA, we have $\overline{1} = \{1, 2, 4\}$, $\overline{2} = \{2\}$, $\overline{3} = \{2, 3, 4\}$, and $\overline{4} = \{4\}$.

The closure of a set of states is just the union of the closure of each individual one.

2) Subset Construction: Supposing we have a NFA M, the DFA $\overline{M}$. The closure of the start state of M becomes the start state of $\overline{M}$. Now we follow the process: given a set S and a character a, compute the set S'_a and then its closure. Continue the process until no more new states are created. Better explained with an example:



The DFA subset construction has as its start state $\overline{\{1\}} = \{1, 2, 6\}$. There is a transition on *a* from state 2 to state 3, and also from state 6 to state 7. Thus, $\overline{\{1, 2, 6\}_a} = \overline{\{3, 7\}} = \{3, 4, 7, 8\}$, and we have $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$. Since there are no other character transitions from 1, 2, or 6, we go on to $\{3, 4, 7, 8\}$. There is a transition on *b* from 4 to 5 and $\overline{\{3, 4, 7, 8\}_b} = \overline{\{5\}} = \{5, 8\}$, and we have the transition $\{3, 4, 7, 8\} \xrightarrow{b} \{5, 8\}$. There are no other transitions. Thus, the subset construction yields the following DFA equivalent to the previous NFA:



C. Simulating an NFA using the subset construction

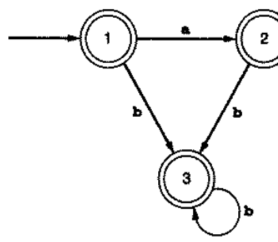
Instead of constructing all the states of the associated DFA, we construct the state at each point that is inddicated by the next input character. Therefore we construct only the states that *will* occur on a determinate input string.
Disadvantage might be that a state might be constructed many times in the case of a loop. So it isn't used in scanners, but remains a posiblity for pattern matching in editor and search programs.

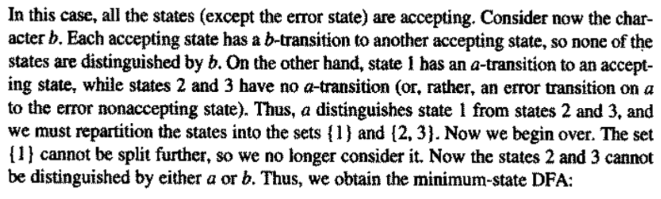
$\{1\} \xrightarrow{x} \{2, 3, 4, 5, 7, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\}$
$\xrightarrow{d} \{4, 5, 6, 7, 9, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\}$

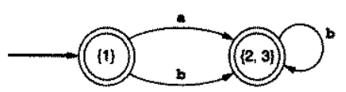
D. Minimizing the number of states in a DFA

The resulting DFA from the algorithm may be more complex than necessary. For any DFA there's an equivalent DFA with minimum number of states that can be obtained, it is unique.
First, most optimistic assumption possible: creates two sets (accepting and non-accepting states). If we make them transition with, say, a. And all the accepting states transition to a state in the accepting once, this defines an $a - transition$

from the new state (with all the accepting states) to itself. But if there are two subsets that point one to accepting and one to the non-accepting, those are **distinguished** by a. We must now repeat the process until all subsets contain 1 state, or there are no more splits to happen.




In this case, all the states (except the error state) are accepting. Consider now the character *b*. Each accepting state has a *b*-transition to another accepting state, so none of the states are distinguished by *b*. On the other hand, state 1 has an *a*-transition to an accepting state, while states 2 and 3 have no *a*-transition (or, rather, an error transition on *a* to the error nonaccepting state). Thus, *a* distinguishes state 1 from states 2 and 3, and we must repartition the states into the sets {1} and {2, 3}. Now we begin over. The set {1} cannot be split further, so we no longer consider it. Now the states 2 and 3 cannot be distinguished by either *a* or *b*. Thus, we obtain the minimum-state DFA:

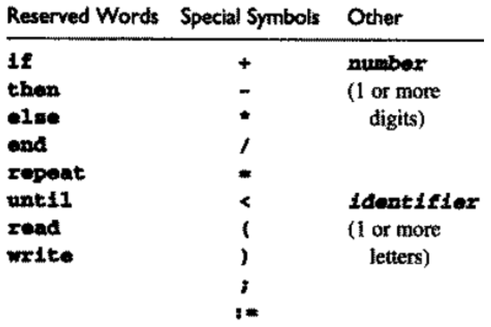


§

V. Implementation of a TINY scanner

A. Implementing a Scanner for the Sample Language TINY

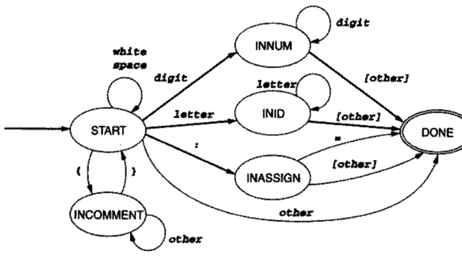
Tokens and their attributes are defined here:



Reserved Words	Special Symbols	Other
if	+	***number***
then	-	(1 or more
else	*	digits)
end	/	
repeat	=	
until	<	***identifier***
read	(	(1 or more
write	)	letters)
	;	
	:=	

Assignment is the only 2-character-long special symbol. Comments use curly brackets, and it is free-space. Since the tokens are simple, we caan develop the DFA directly, rather than the RE.

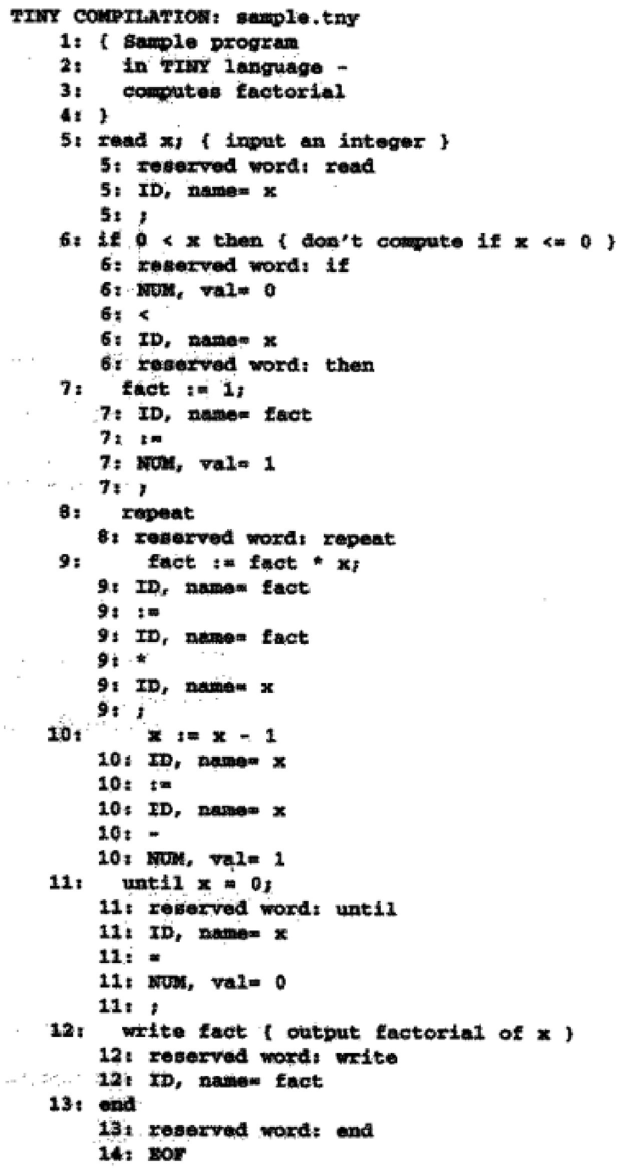
Since special symbols are mostly one character long, a DFA for thembacn be done with a single starting state that branches out to all of them,the difference of the accepting states is what value is returned to the scanner. They could even be collapsed into a single one if the value is returned in some other way. By adding the identifiers and comments we end up with this.



All other characters not included in here are labeled as errors and trerated as such. Reserved words are treated as identifiers and then checked on a table.

This is replicated by the `getToken` function in `scan.c`. Tokens along with bookkeeping are kept in `globals.h`. The lexeme of the tokens is stored in a variable called `tokenString`. Additional bookkeeping is done through `reservedWords, reservedLookup`.

Lines have to be smaller than 255 chars, due to the `getNextChar` function and its buffer. For the transition to the final state of an identifier or number we have `ungetNextChar`.


```
TINY COMPILATION: sample.tny
   1: { Sample program
   2:   in TINY language -
   3:   computes factorial
   4: }
   5: read x; { input an integer }
      5: reserved word: read
      5: ID, name= x
      5: ;
   6: if 0 < x then { don't compute if x <= 0 }
      6: reserved word: if
      6: NUM, val= 0
      6: <
      6: ID, name= x
      6: reserved word: then
   7:   fact := 1;
      7: ID, name= fact
      7: :=
      7: NUM, val= 1
      7: ;
   8:   repeat
      8: reserved word: repeat
   9:     fact := fact * x;
      9: ID, name= fact
      9: :=
      9: ID, name= fact
      9: *
      9: ID, name= x
      9: ;
  10:     x := x - 1
      10: ID, name= x
      10: :=
      10: ID, name= x
      10: -
      10: NUM, val= 1
  11:   until x = 0;
      11: reserved word: until
      11: ID, name= x
      11: =
      11: NUM, val= 0
      11: ;
  12:   write fact { output factorial of x }
      12: reserved word: write
      12: ID, name= fact
  13: end
      13: reserved word: end
      14: EOF
```

That is an example of a compilation.

B. Reserved words vs. Identifiers

In TINY, the reserved word is considered an identifier and then lineally searched for in the reserved table. But for larger languages the look for a **minimal perfect hash function** has been looked for.

Another alternative is to include them in the symbol table and mark them as reserved, but this is not done in TINY because the table is made after scanning.

C. Allocating Space for Identifiers

Identifiers can only be 40 characters long, and the compiler dynamically allocates them so no space is wasted. An alternative is doing it manually.

VI. Use of Lex to generate a scanner automatically

The **GNU compiler package** uses **flex (fast-Lex)**. It receives a RE, and prroduces C source code of a table implementation of a DFA and the `getToken` procedure.

A. Lex Conventions for RE

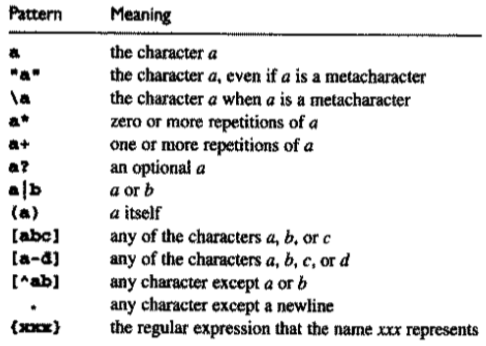
Allows matching single or strings of characters, plus metacharacters by writing them inbetween quotes (more convenient than using backslash).
Square brackets in this case means sets of characters to choose from:

$$(a|b) = [ab]$$

And range of characters can be done with the hyphen. If it is a character that *is not* in the set, we simply include the $^\wedge$ as the first character in the brackets.
In between square brackets most metacharacters don't have to be quoted anymore.

$$(" + "|" - ") = [-+]$$

In that order.
But some like backslash are still metacharacters inside. Also curly brackets represent an RE.



Pattern	Meaning
`a`	the character *a*
`"a"`	the character *a*, even if *a* is a metacharacter
`\a`	the character *a* when *a* is a metacharacter
`a*`	zero or more repetitions of *a*
`a+`	one or more repetitions of *a*
`a?`	an optional *a*
`a|b`	*a* or *b*
`(a)`	*a* itself
`[abc]`	any of the characters *a*, *b*, or *c*
`[a-d]`	any of the characters *a*, *b*, *c*, or *d*
`[^ab]`	any character except *a* or *b*
`.`	any character except a newline
`{xxx}`	the regular expression that the name *xxx* represents

B. The format of a Lex Input File

Divided in 3: definition, rules and routines. The layout is:s

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

Some parts are basically RE, and some are C source code supplied to Lex.

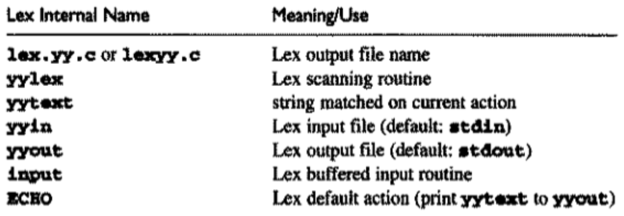
1) Definition: contains any C code that must be inserted external to any function between the delimeters %{ and %}. And the names of the regular expressions.
2) Sequence of RE along with the code that must be executed when it is matched.
3) Any auxiliary routines (like for example a main program).

An important note is to include `yylex();` in the auxiliary routine, so the C code can be compiled into an executable program.

1) Ambiguity resolution: The output will always fisrt match the longest possible substirng to aa rule. If they are the same size, the first one in the action section will be selected.

2) Insertion of code: The code written between the mentioned delimiters will be written directly to the output program, same with any text in the auxiliary procedures. AAnd any code that follows a RE in the action section will be inserted at the appropiate place.

3) Internal Names: Here is a list of internal names:



Lex Internal Name	Meaning/Use
`lex.yy.c` or `lexyy.c`	Lex output file name
`yylex`	Lex scanning routine
`yytext`	string matched on current action
`yyin`	Lex input file (default: `stdin`)
`yyout`	Lex output file (default: `stdout`)
`input`	Lex buffered input routine
`ECHO`	Lex default action (print `yytext` to `yyout`)

For example the routine `input` will automatically take input from `yyin`.

C. A TINY Scanner Using Lex

We directly include `globals.h, util.h, scan.h, tokenString`. Also we define the names of the RE that make the TINY tokens. In the action section we also list various tokens along with `return`. We could also write code as in the previous section where they mention how each reserved word is looked in a table.
We also need to include an RE for comments.
There is no code to return `EOF` since Lex has a default way of doing it. So we just include `ENDFILE` in the globals file.
We also include a definition of `getToken` so we don't have to change the code in the compiler directly.
Disadvantage: no echoing with line numbers provided as we saw before.