Lecture Notes on Lexical Analysis

15-411: Compiler Design
André Platzer

Lecture 6

1 Introduction

Lexical analysis is the first phase of a compiler. Its job is to turn a raw byte or character input stream coming from the source file into a token stream by chopping the input into pieces and skipping over irrelevant details. The primary benefits of doing so include significantly simplified jobs for the subsequent syntactical analysis, which would otherwise have to expect whitespace and comments all over the place. The job of the lexical analysis is also to classify input tokens into types like INTEGER or IDENTIFIER or WHILE-keyword or OPENINGBRACKET. Another benefit of the lexing phase is that it greatly compresses the input by about 80%. A lexer is essentially taking care of the first layer of a regular language view on the input language. We follow a presentation similar to a recent book [WSH12, Ch. 2]. Further presentations can be found in [WM95, Ch. 7] and [App98, Ch. 2].

2 Lexer Specification

We fix an alphabet Σ, i.e., a finite set of input symbols, e.g., the set of all letters a-z and digits 0-9 and brackets and operators +,- and so on.[1] The the set Σ^* of words or strings is defined as the set of all finite sequences of elements of Σ. For instance, `ifah5+xy-+` is a string, but not necessarily a very interesting one, from a grammatical perspective (which is what lexers

[1]Real lexers also have to deal with capital letters, but we simply pretend to be ignorant about capitalization in these lecture notes to make things easier.

will not have to know about, because that's the parser's job). The empty string with no characters is denoted by ϵ, but you will sometimes also find the name λ for it, which we don't use here in order to not get confused with Church's λ-calculus.

A lexer specification has to say what kind of input it accepts and which token type it will associate with a particular input. For example, the fragment `15411` of the input string should be tokenized as an INTEGER. For reasons of representational efficiency, it is a very good idea to specify the input that a lexer accepts by regular expressions. On a side note, regular expressions and their extensions [Sal66, Koz97, HKT00, Pla12] actually turn out to be very useful for many purposes.

Regular expressions r, s are expressions that are recursively built of the following form:

regex	**matches**
a	matches the specific character a from the input alphabet
$[a-z]$	matches a character in the specified range of letters a to z
ϵ	matches the empty string
$r\|s$	matches a string that matches r or one that matches s
rs	matches a string that can somehow be split into two parts, the first matching r, the second matching s
r^*	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 0$

For instance, the set of strings over the alphabet $\{a, b\}$ with no two or more consecutive a's is described by the regular expression $b^*(abb^*)^*(a|\epsilon)$. Other common regular expressions are

regex	**defined**	**matches**
r^+	rr^*	matches a string that consists of n parts where each part matches r, for any natural number $n \geq 1$
$r?$	$r\|\epsilon$	optionally matches r, i.e., matches the empty string or a string matching r

To specify a lexical analyzer we can use a sequence of regular expressions along with the token type that they recognize (the last one, `LPAREN`, for instance, recognizes a single opening parenthesis, whose occurrence on the right hand side we need to quote to distinguish it from brackets used to describe the regular expression, likewise for space):

`IF`	$\equiv i\,f$
`GOTO`	$\equiv g\,o\,t\,o$
`FOR`	$\equiv f\,o\,r$
`IDENTIFIER`	$\equiv [a-z]([a-z] \vert [0-9])^*$
`INT`	$\equiv [0-9][0-9]^*$
`REAL`	$\equiv ([0-9][0-9]^*.[0-9]^*) \vert (.[0-9][0-9]^*)$
`LPAREN`	$\equiv$ `"("`
`ASSIGN`	$\equiv$ `"="`
`SKIP`	$\equiv$ `" "*`

In addition, we would say that tokens matching the `SKIP` whitespace recognizer are to be skipped and filtered away from the input, because the parser does not want to see whitespace. Likewise with comments. Note, however, that whitespaces and comments are still significant for the lexer because they separate tokens. For example, `if xyz` gives `IF IDENTIFIER`, while `ifxyz` gives `IDENTIFIER`, even if the `SKIP` token in between is never shown to the parser.

Regular expressions themselves are not unambiguous for splitting an input stream into a token sequence. The input `goto5` could be tokenized as `IDENTIFIER` or as the sequence `GOTO INT`. The input sequence `if 5` could be tokenized as `IF INT` or as `IDENTIFIER INT`.

As disambiguation rule we will use the principle of the *longest possible match*. The longest possible match from the beginning of the input stream will be matched as a token. And if there are still multiple regular expression rules that match the same length, then the first rule with longest match takes precedence over others.

Why do we choose the longest possible match as a disambiguation rule instead of the shortest? The shortest would be easier to implement. But with the shortest match, `ifo = ford_trimotor` would be tokenized as `IF IDENTIFIER ASSIGN FOR IDENTIFIER` and not as `IDENTIFIER ASSIGN IDENTIFIER`. And, of course, the latter is what one would have meant by assigning the identifier for the 1925 Ford Trimotor aircraft "Tin Goose" to the identified flying object (ifo).

3 Lexer Implementation

Lexers are specified by regular expressions. Classically, however, they are implemented by finite automata.

Definition 1. *A* finite automaton *for a finite alphabet* Σ *consists of*

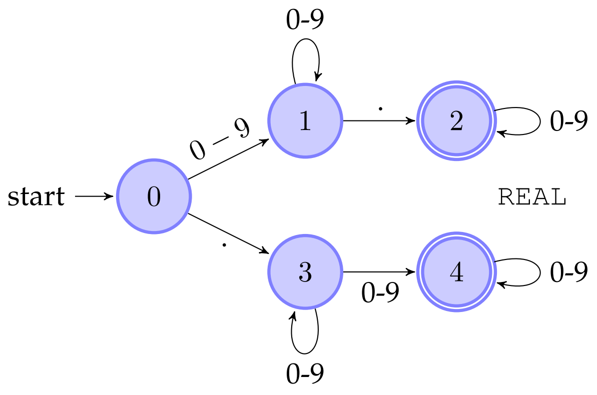
- *a finite set Q of states,*
- *a set $\Delta \subseteq Q \times \Sigma \times Q$ of edges from states to states that are labelled by letters from the input alphabet Σ. We also allow ϵ as a label on an edge, which then means that (q, ϵ, q') is a spontaneous transition from q to q' that consumes no input.*
- *an initial state $q_0 \in Q$*
- *a set of accepting states $F \subseteq Q$.*

The finite automaton accepts an input string $w = a_1 a_2 \dots a_k \in \Sigma^*$ iff there is an $n \in \mathbb{N}$ and a sequence of states $q_0, q_1, q_2, \dots, q_n \in Q$ where q_0 is the initial state and $q_n \in F$ is an accepting state such that $(q_{i-1}, a_i, q_i) \in \Delta$ for all $i = 1, \dots, n$.

In pictures, this condition corresponds to the existence of a set of edges in the automaton labelled by the appropriate input:

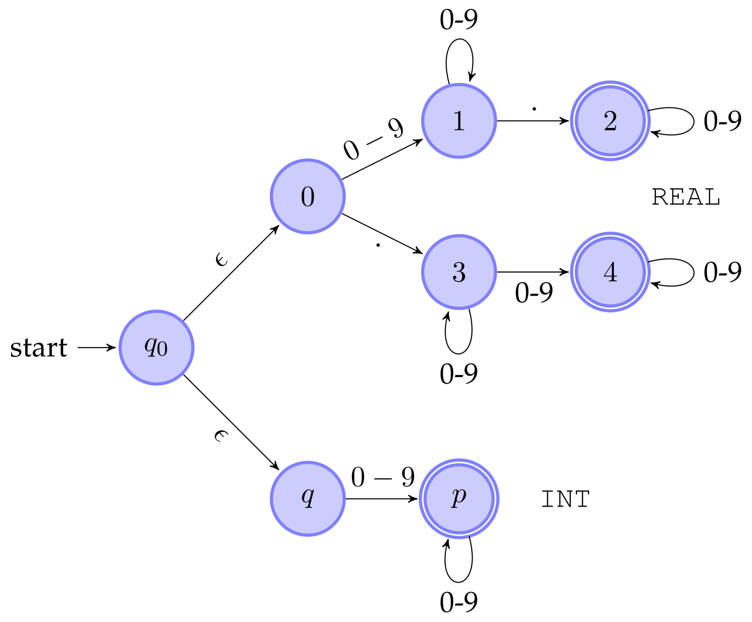
$$q_0 \overset{a_1}{\to} q_1 \overset{a_2}{\to} q_2 \overset{a_3}{\to} q_3 \overset{a_4}{\to} \cdots \overset{a_{n-1}}{\to} q_{n-1} \overset{a_n}{\to} q_n \in F$$

As an abbreviation for this situation, we also write $(q_0, w) \to^* (q_n, \epsilon)$. We also write $(q_{i-1}, a_i w_i) \to (q_i, w_i)$ when $(q_{i-1}, a_i, q_i) \in \Delta$. By that we mean that the automaton, when starting in state q_0 can consume all input of word w with a series of transitions and end up in state q_n with no remaining input to read (ϵ). For instance, an automaton for accepting `REAL` numbers is



Of course, when we use this finite automaton to recognize the number `3.1415926` in the input stream `3.1415926-3+x;if`, then we do not only want to know that a token of type `REAL` has been recognized and that the remaining input is `-3+x;if`. We also want to know what the value of the token of type `REAL` has been, so we store it's value along with the token type.

The above automaton is a *deterministic finite automaton* (DFA). At every state and every input there is at most one edge enabling a transition. But in general, finite automata can be *nondeterministic finite automata* (NFA). That is, for the same input, one path may lead to an accepting state while another attempt fails. That can happen when for the same input letter there are multiple transitions from the same state. In particular, in order to be able to work with the longest possible match principle, we have to keep track of the last accepting state and reset back there if the string cannot be accepted anymore. Consider, for instance, the nondeterministic automaton that accepts both `REAL` and `INT` and starts of by a nondeterministic choice between the two lexical rules.

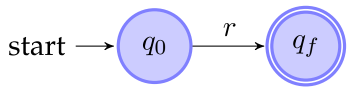


In the beginning, this poor NFA needs to guess which way the future input that he hasn't seen yet will end up. That's hard. But NFAs are quite convenient for specification purposes (just like regular expressions), because the user does not need to worry about these choices.

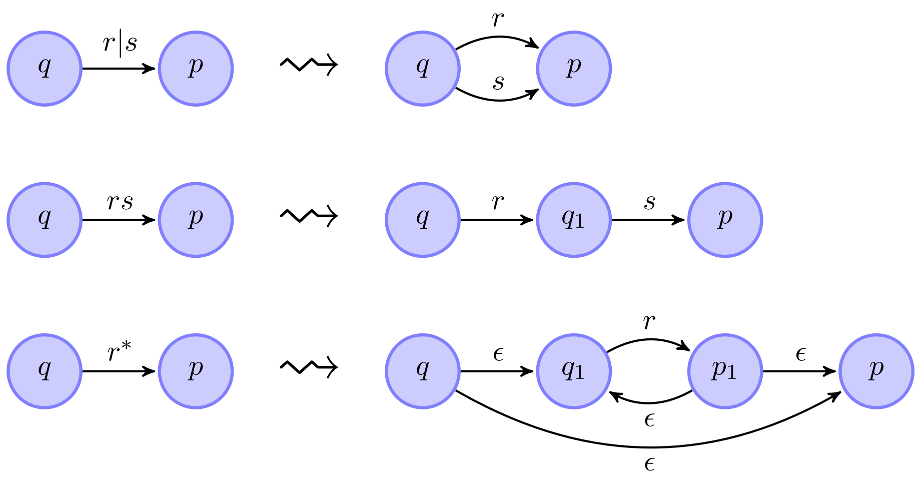
4 Regular Expressions $\rightsquigarrow$ Nondeterministic Finite Automata

Regular expressions are very nice for representing what a lexer is supposed to read. Fortunately, the regular expressions can be converted into a finite automaton (and also backwards, which we will not need here).

For converting a regular expression r into a nondeterministic finite automaton (NFA), we define a recursive procedure. We start with an extended NFA that still has regular expressions as input labels on the edges.



Then we successively transform edges that still have regular expressions as improper input labels by their defining automata patterns. That is, whenever we find a regular expression on an edge that is not just a single letter from the input alphabet then we use the transformation rule to get rid of it



When applying the rule we match on the pattern on the left in the current candidate for an NFA and replace it by the right, introducing new states q_1, q_2 as necessary.

5 Nondeterministic Finite Automata $\rightsquigarrow$ Deterministic Finite Automata

The conversion from regular expressions to NFAs is quite simple. NFAs are convenient for specification purposes, but bad for implementation. It is easy to implement a DFA, however. We just store the current state in a program variable, initialized to q_0, and depending on the next input character, we transition to the next state according to the transition table Δ. Whenever there is an accepting state, we notice that this would be a token that we recognized. But in order to find the longest possible match, we still keep going. If we ultimately find an input character that is not recognized or accepted, then we just backtrack to the last possible match that we have remembered (and unconsume the input characters we have read from the input stream so far). But how would we implement an NFA? There are so many choices that we do not know which one to choose. There is no canonical last accepting choice in an NFA even.

What we could do to implement an NFA is to follow the input like in a DFA implementation, but whenever there is a choice, we follow all options at once. That will branch quickly and will require us to do a lot of work at once, which is inefficient. Nevertheless, it gives us the right intuition about what has to be done. We just need to turn it around and follow the same principle in a precomputation step instead of at runtime. We follow all options and keep the set of choices of where we could be around.

This is the principle behind the *powerset construction* that turns an NFA into a DFA by following all options at once. That is, instead of a single state, we now consider the set of states in which we could be. We, of course, want to start in the initial powerset state $\{q_0\}$ that only consists of the single initial state q_0. But, first we have to follow all possible ϵ-transitions that lead us from q_0 to other states. When $S \subseteq Q$ is a set of states, we define $Cl^\epsilon(S)$ to be the ϵ-closure of S, i.e., the set of states we can go to by following arbitrarily many ϵ-transitions from states of S, which do not consume any input.

$$Cl^\epsilon(S) := \bigcup_{q \in S} \{q' \ : \ (q, \epsilon) \to^* (q', \epsilon)\}$$

Now from a set of states $S \subseteq Q$ we make a transition, say with input letter a and figure out the set of all states to which we could get to by following a-transitions from any of the S states, again following ϵ-transitions:

$$N(S, a) := Cl^\epsilon(\{q' \in Q \ : \ (q, a) \to (q', \varepsilon) \text{ and } q \in S\})$$

The condition $(q,a) \to (q',\varepsilon)$ is equivalent to $(q,a,q') \in \Delta$. We can summarize all these transitions by just a single a-transition from S to successor $N(S,a)$. Repeating this process results in a DFA that accepts exactly the same language as the original NFA. The complexity of the algorithm could be exponential, though, because there are exponentially many states in the powerset that we could end up using during the DFA construction.

Definition 2 (NFA⇝DFA)**.** *Given an NFA* finite automaton (Q,Δ,q_0,F)*, the corresponding DFA* (Q',Δ',q_0',F') *accepting the same language is defined by*

- Q' *is a subset of the sets of all subsets of* Q*, i.e., a part of the powerset* $Q' \subseteq 2^Q$
- $\Delta' := \{(S,a,N(S,a)) \;:\; a \in \Sigma\}$.
- $q_0' := Cl^{\epsilon}(q_0)$
- $F' := \{S \subseteq Q \;:\; S \cap F \neq \emptyset\}$

After turning the NFA into a DFA, we can directly implement it to recognize tokens from the input stream.

It should be noted that there are direct ways of obtaining DFAs from regular expressions, without going through the construction of NFAs. Those techniques are very algebraic and elegant using Brzozowski derivatives [Brz64].

6 Minimizing Deterministic Finite Automata

Another operation that is often done by lexer generator tools is to minimize the resulting DFA by merging states and reducing the number of states and transitions in the automaton. This is an optimization and we will not pursue it any further.

7 Regular Expression ⇝ Deterministic Finite Automata

It turns out that there is a very elegant and purely algebraic way of directly translating regular expressions into DFAs without having to go through explicit automata construction, determinization, and possibly minimization. This algebraic approach uses Brzozowski derivatives [Brz64] and *Antimirov's partial derivatives* [Ant96]. For this, we identify regular expressions by the set of words that they match. So instead of saying that regular expression r matches the word w, we simply write $w \in r$. The derivative, $D_a(r)$ of a regular expression r by alphabet letter a is defined as

$$D_a(r) = \{w \ : \ aw \in r\}$$

The derivative represents the set of continuations after letter a that the regular expression r can match. The derivative of a regular expression can be computed syntactically in a very similar way as the usual derivatives of functions. The result is a regular expression.

$$\begin{aligned}
D_a(\emptyset) &= \emptyset \\
D_a(\epsilon) &= \emptyset \\
D_a(a) &= \epsilon \\
D_a(b) &= \emptyset \qquad (b \neq a) \\
D_a(r|s) &= D_a(r) \mid D_a(s) \\
D_a(rs) &= D_a(r)s \mid \delta(r)D_a(s) \\
D_a(r^*) &= D_a(r)r^*
\end{aligned}$$

If we tilt our head a little bit and pretend $|$ was addition $(+)$ and pretend that rs would be multiplication, this looks very much like a standard derivative of functions with ϵ playing the role of 1 and $\emptyset$ playing the role of 0. The primary difference is the occurrence of operator $\delta(r)$ in $D_a(rs)$, which we still need to define. The operator $\delta(r)$ is supposed to detect whether r matches the empty word ϵ. Thus, $\delta(r)$ is defined as follows

$$\delta(r) = \begin{cases} \varepsilon & \text{if } \epsilon \in r \\ \emptyset & \text{otherwise} \end{cases}$$

This operator can be computed entirely syntactically as well

$$\begin{aligned}
\delta(\emptyset) &= \emptyset \\
\delta(\epsilon) &= \epsilon \\
\delta(a) &= \emptyset \\
\delta(r|s) &= \delta(r) \mid \delta(s) \\
\delta(rs) &= \delta(r)\delta(s) \\
\delta(r^*) &= \epsilon
\end{aligned}$$

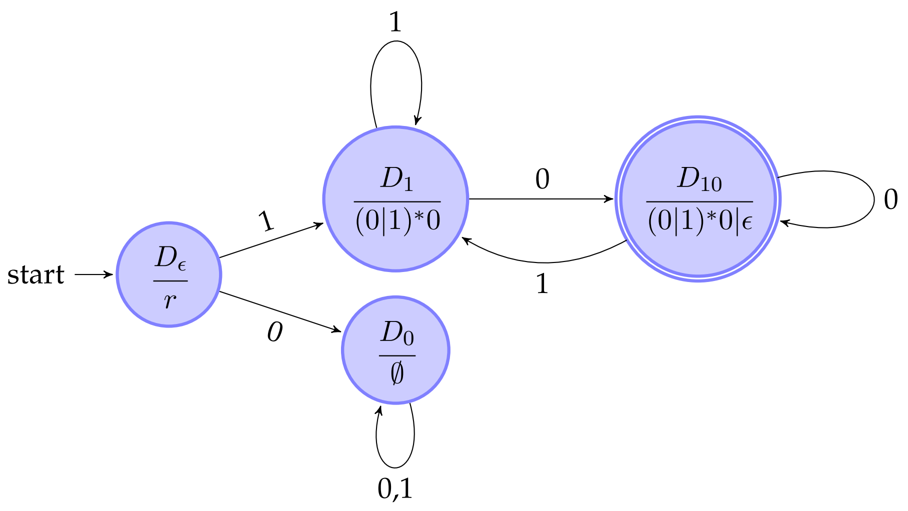
For ordinary functions, higher derivatives can be defined by deriving multiple times. The same thing makes sense for derivatives of regular expressions where we define $D_w(r)$ for a word w by a simple inductive definition on w in which we derive successively by the next letter:

$$\begin{aligned} D_\epsilon(r) &= r \\ D_{wa}(r) &= D_a(D_w(r)) \end{aligned}$$

A number of very interesting theoretical and practical results can be proved about Brzozowski derivatives and their extensions. Here we only show how an automaton can be constructed systematically using successive derivatives. It can be shown that this process terminates.

The idea is that $D_a(r)$ represents the "remainder" regular expression of r after input a has been read. Thus, there is a transition with input a from the state r to the state $D_a(r)$. We simply use regular expressions as the states of an automaton (not as their actions like in Section 4).

As an example, consider the regular expression $r = 1(0|1)^*0$. Thus, we construct a DFA for it by starting from a state $D_\epsilon(r) = r$ and successively following all letters a_1 to states $D_{a_1}(r)$ and then on following all letters a_2 to states $D_{a_1a_2}(r)$ and so on. The states where the regular expression matches the empty word ϵ are the ones that are final states. State s is a final state iff $\delta(s) = \epsilon$. In fact, it can be shown that $w \in r$ iff $\delta(D_w(r)) = \epsilon$.



In this automaton graph, we use the notation $\dfrac{D_w}{s}$ to say that $D_w(r) = s$. It can also be shown that every regular expression can be written in the following linear form

$$r = \delta(r) + \sum_{a \in \Sigma} a D_a(r)$$

8 Summary

Lexical analysis reduces the complexity of subsequent syntactical analysis by first dividing the raw input stream up into a shorter sequence of tokens, each classified by its type (`INT`, `IDENTIFIER`, `REAL`, `IF`, ...). The lexer also filters out irrelevant whitespace and comments from the input stream so that the parser does not have to deal with that anymore. The steps for generating a lexer are

1. Specify the token types to be recognized from the input stream by a sequence of regular expressions
2. Bear in mind that the longest possible match rule applies and the first production that matches longest takes precedence.
3. Lexical analysis is implemented by DFA.
4. Convert the regular expressions into NFAs (or directly into DFAs using derivatives).
5. Join them into a master NFA that chooses between the NFAs for each regular expression by a spontaneous ϵ-transition
6. Determinize the NFA into a DFA
7. Optional: minimize the DFA for space
8. Implement the DFA for a recognizer. Respect the longest possible match rule by storing the last accepted token and backtracking the input to this one if the DFA run cannot otherwise complete.

Quiz

1. Why do compilers have a lexing phase? Why not just do without it?
2. Should a lexer return whitespaces and comments?
3. Why do we categorize tokens into token classes, instead of just working with the particular piece of the input string they represent?

4. Why are there programming languages that do not accept inputs like `x----y`?
5. What aspects of the programming language does a lexer not know about?
6. Do lexer tools work with regular expressions or automata internally? Should they?
7. Why can lexers not work with nondeterministic finite automata? They are so useful for description purposes.
8. Should a reserved keyword of a programming language be a token class of its own? What are the benefits and downsides?

References

[Ant96] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*. MIT Press, Cambridge, 2000.

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[Pla12] André Platzer. Logics of dynamical systems. In *LICS*, pages 13–24. IEEE, 2012.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[WSH12] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design: Syntactic and Semantic Analysis*. Addison-Wesley, 2012.