Systematic Compiler Construction

Michael Sperber Peter Thiemann

January 20, 2022

Contents

Chapter 1

Lexical Analysis

A small psychological exercise demonstrates what lexical analysis is. Read aloud the following expression:

```
return Segment (pi / 2)
```

Listening to yourself, you will notice the following pecularities:

1. You probably read the program word by word (rather than letter by letter).
2. You did not read aloud spaces and line breaks.
3. You (mentally or vocally) treated the word `return` specially because you assumed some underlying meaning.

Most programming languages since the days of FORTRAN are structured in such a way that a program splits into a linear sequence of "words." In the process of determining the boundaries between "words," certain insignificant elements of a program (such as comments) drop out. No higher-level syntactic processing is involved in this phase, so it makes sense to perform this splitting before further syntactic analysis. This splitting is called "lexical analysis." The result of lexical analysis for the above program might look like this:

⟨return⟩
⟨identifier [*Segment*]⟩
⟨lparen⟩
⟨identifier [*pi*]⟩
⟨slash⟩
⟨intlit [*2*]⟩
⟨rparen⟩

Some of the details of this splitting may seem arbitrary at this point: Why does `return` become a unit ⟨return⟩ while `Segment` is denoted as ⟨identifier [*Segment*]⟩? Well, the word `return` appears in the Python language definition as one of the *keywords* of the language. Because the language definition is finite, the number of keywords must also be finite. Hence, it makes sense to treat keywords as special units. The word `Segment`, on the other hand, does not appear in the language definition: `Segment` is an *identifier* chosen by the programmer. There are infinitely many identifiers, but they all share the same syntactic status. Hence, the ⟨identifier [*Segment*]⟩ encoding which separates this information from the actual name of the identifier.

1.1 Basic definitions

The part of a compiler responsible for lexical analysis is called a *scanner* or a *lexer*. It operates on the program as a sequence of characters and performs three main tasks:

1. it divides the input into logically cohesive sequences of characters, the *lexemes*;
2. it filters out formatting characters, like spaces, tabulators, and newline characters (*whitespace* characters);
3. it filters out comments; and
4. it maps lexemes into *tokens*, *i.e.*, symbolic names for classes of lexemes. Most tokens carry *attributes* which are computed from the lexeme. There is a one-to-one correspondence between lexemes and token/attribute pairs.

1.1 Definition (Lexical analysis)
Let Σ be the alphabet of a programming language, T a finite set of tokens, *and A an arbitrary set of* attributes. *A* scanner *is a function*

$$scan : \Sigma^* \to (T \times A)^*$$

such that there is a function

$$unscan : (T \times A)^* \to \Sigma^*$$

with the following properties

1. $scan \circ unscan = id_{(T\times A)^*}$ *and*
2. *there is a function* $untoken : (T \times A) \to \Sigma^*$ *so that*

$$unscan(t_1 t_2 \dots) = untoken(t_1) untoken(t_2) \dots$$

(i.e., *unscan is a homomorphism).*

□

The functions *scan* and *unscan* have the following derived property.

$$scan \circ unscan \circ scan = scan$$

Thus, the *scan* function splits up the input into lexemes, maps them to token/attribute pairs, and removes white space along the way. The functions *scan* and *unscan* are in general not inverses due to white space removal: *scan* removes all white space irreversibly and *unscan* has to introduce white space so that the original lexemes are properly separated.

1.2 Construction of scanners

There is a spectrum of possibilities for constructing scanners. First, a scanner can be implemented manually. A manual implementation can be tuned for efficiency and it does not place any restrictions on the language of lexemes. While this approach might be sensible for a simple language with few classes of lexemes, it is not appropriate for modern languages. Furthermore, a hand-written scanner is hard to maintain because it must be implemented from a separate specification. Last, the efficient implementation of some components of a scanner (*e.g.*, input buffering) is a non-trivial effort. A hand-written scanner must develop these components from scratch and cannot reuse previous efforts.

A second possibility is to rely on a prefabricated library of scanner components. In this approach, the implementation of the scanner can be close to the specification. Thus, it is amenable to fast development and easy maintainance. The downside is that the library cannot easily take advantage of global optimizations, so that such an implementation is significantly slower. Also, there are usually restrictions on the language of lexemes.

The third possibility is using a scanner generator like `lex`, `ocamllex`, `jflex`, or `ply`. A scanner generator generates the implementation of the scanner from a high-level specification. Thus, it combines the advantages of the library approach with efficiency. As with the library approach, the language of lexemes is usually restricted.

For instructional purposes, we concentrate on the second possibility. While sacrificing some efficiency, the library approach enables us to discuss the entire implementation of a scanner, without having to gloss over details of code generation as it would be the case with a scanner generator.

Both, the library approach and the generator approach, restrict the language of lexemes to a *regular language* R, for a number of reasons.

- All set-theoretic operations (union, intersection, difference) on regular languages yield regular languages. Hence, a specification of a regular language can rely on them.
- The word problem $w \in R$ is decidable in linear time in the length of word w.
- For each regular language there is a minimal recognizer. Hence, there exists a smallest scanner for each kind of lexeme.
- All modern languages specify their lexemes using regular languages.

1.3 Descriptions of regular languages

Theoretical computer science tells us that there are at least three equivalent means of describing regular languages. We briefly introduce each of them and conclude with a discussion of which description is best suitable for specifying and implementing a scanner.

1.3.1 Regular grammars

Each regular language may be described by a regular grammar. A regular grammar is a grammar $\mathcal{G} = (N, \Sigma, P, S)$ where each production in P has one of the following forms:

$$A \to x\,B, \qquad A \to x, \qquad A \to \varepsilon$$

where $A, B \in N$, $x \in \Sigma$, and ε is the empty word. A word w belongs to the language defined by $\mathcal{G}$ iff there is a derivation $S \stackrel{*}{\Rightarrow} w$.

1.3.2 Finite automata

Another description of a regular language R is a finite automaton that recognizes R. A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set of states, Q, a finite input alphabet, Σ, a transition operator, δ, an initial state, $q_0 \in Q$, and a set of final states $F \subseteq Q$. Finite automata come in different flavors of equal power, distinguished by the definition of δ. The basic idea, however, is the same: at any time, the automaton is in a state $q \in Q$ and δ yields a new state from the current state and an input symbol. A word belongs to the language $L(M)$ recognized by M iff the automaton is in a final state after consuming all input symbols.

Deterministic finite automata (DFA)

A finite automaton is deterministic if $\delta : Q \times \Sigma \to Q$ is a function. To define $L(M)$, we extend δ to a function $\widehat{\delta} : Q \times \Sigma^* \to Q$ as follows:

$$\begin{array}{lcl} \widehat{\delta}(q, \varepsilon) & = & q \\ \widehat{\delta}(q, a\, w) & = & \widehat{\delta}(\delta(q, a), w) \end{array}$$

With this definition, $w \in L(M)$ iff $\widehat{\delta}(q_0, w) \in F$.

Nondeterministic finite automaton (NFA)

A finite automaton is non-deterministic if $\delta \subseteq Q \times \Sigma \times Q$ is an arbitrary relation. Again, to define $L(M)$, we extend δ to a relation $\widehat{\delta} \subseteq Q \times \Sigma^* \times Q$, which is the least relation satisfying the following two equations:

$$\begin{array}{lll} (q, \varepsilon, q) \in \widehat{\delta} & & \\ (q, a\, w, q') \in \widehat{\delta} & \text{iff} & (\exists q'')\ (q, a, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \end{array}$$

With this definition, $w \in L(M)$ iff $(\exists q_f \in F)\ (q_0, w, q_f) \in \widehat{\delta}$.

Nondeterministic finite automata with autonomous transitions (NFA-ε)

An NFA may also allow autonomous (or instantaneous or ε) transitions which change the state without consuming any input. In this case, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is an arbitrary relation. Again, we extend δ to the least relation $\widehat{\delta} \subseteq Q \times \Sigma^* \times Q$ which satisfies the following equations:

$$\begin{array}{lll} (q, \varepsilon, q) \in \widehat{\delta} & & \\ (q, w, q') \in \widehat{\delta} & \text{iff} & (\exists q'')\ (q, \varepsilon, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \\ (q, a\, w, q') \in \widehat{\delta} & \text{iff} & (\exists q'')\ (q, a, q'') \in \delta \text{ and } (q'', w, q') \in \widehat{\delta} \end{array}$$

With this definition, $w \in L(M)$ iff $(\exists q_f \in F)\ (q_0, w, q_f) \in \widehat{\delta}$.

1.3.3 Regular expressions

A regular expression is a highly declarative way of specifying a regular language. The set of regular expressions over an alphabet Σ is the smallest set $RE(\Sigma)$ with:

- $\underline{\emptyset} \in RE(\Sigma)$
- $\underline{\varepsilon} \in RE(\Sigma)$
- if $a \in \Sigma$ then $\underline{\mathtt{a}} \in RE(\Sigma)$
- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 r_2 \in RE(\Sigma)$
- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 \mid r_2 \in RE(\Sigma)$
- if $r \in RE(\Sigma)$ then $r^* \in RE(\Sigma)$.

A regular expression defines a language as prescribed by the function $L : RE(\Sigma) \to \mathcal{P}(\Sigma^*)$.

$$
\begin{array}{lcl}
L(\underline{\emptyset}) & = & \emptyset \\
L(\underline{\varepsilon}) & = & \{\varepsilon\} \\
L(\underline{\mathtt{a}}) & = & \{a\} \\
L(r_1 r_2) & = & L(r_1) \cdot L(r_2) \\
 & := & \{w_1 w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\
L(r_1 \mid r_2) & = & L(r_1) \cup L(r_2) \\
L(r^*) & = & L(r)^* \\
 & := & \{w_1 w_2 \dots w_n \mid n \in \mathbf{N}, w_i \in L(r)\} \\
 & = & \{\varepsilon\} \cup L(r) \cup L(r) \cdot L(r) \cup L(r) \cdot L(r) \cdot L(r) \cup \dots
\end{array}
$$

A word w belongs to the language described by r iff $w \in L(r)$.

1.3.4 Discussion

We have looked at three different descriptions for regular languages. Now, we assess each method for its usability regarding the specification and implementation of scanners.

A grammar is a low-level means of describing a regular language. A grammar emphasizes the generative aspect of a language definition. While it is easy to generate words in the language from the grammar, it is non-trivial to check if a given word belongs to the language (the word problem). In addition, a grammar is not a concise description of a language. Even simple languages can take many rules to describe. Hence, we conclude that a grammar is neither suited for a high-level specification of a scanner nor for its implementation.

A DFA is also a low-level description of a regular language. Its definition immediately gives rise to a recognizer which is simple to implement efficiently. However, it is not a concise description because even simple languages can require a large set of states in their automaton. In conclusion, while a DFA makes a good implementation it is unsuitable for a high-level specification of a scanner.

A regular expression is a declarative description of a regular language. It can give rise to highly concise descriptions (in particular, if further operations like intersection and difference are included). However, it requires a clever implementation or a translation to a DFA to efficiently recognize words from the language.

Hence, language definitions use regular expressions to define the lexemes of a programming language.

1 Example
The JavaScript reference manual contains a section "Lexical Conventions". Figure 1.1 shows a slightly simplified description of the lexemes for identifiers and integer constants.

Lexical analysis exists mainly for pragmatic reasons: the more involved syntactic analysis which follows can be much simpler because of it. Moreover, regular grammars have well-known algorithms to recognize them. Theoretical computer science tells us that a finite, deterministic automaton (DFA) can serve as a recognizer for any regular language. A DFA is a simple machine, and thus reasonably easy to implement. The construction of the state diagram for a DFA is a tedious process. Hence, it makes things easier to avoid the explicit construction of the automaton. Fortunately, the automaton follows automagically from our simpler approach to recognizing regular languages.

$\langle$digit$\rangle$ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
$\langle$hexdigit$\rangle$ ::= $\langle$digit$\rangle$ | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
$\langle$hexprefix$\rangle$ ::= `0x` | `0X`
$\langle$sign$\rangle$ ::= $\langle$empty$\rangle$ | `-`
$\langle$empty$\rangle$ ::=
$\langle$integer-literal$\rangle$ ::= $\langle$sign$\rangle$ $\langle$digit$\rangle^+$ | $\langle$sign$\rangle$ $\langle$hexprefix$\rangle$ $\langle$hexdigit$\rangle^+$

$\langle$letter$\rangle$ ::= `A` | `B` | `C` | `...`| `Z` | `a` | `b` | `c` | `...`| `z`
$\langle$identifier-start$\rangle$ ::= $\langle$letter$\rangle$ | `$` | `_`
$\langle$identifier-part$\rangle$ ::= $\langle$identifier-start$\rangle$ | $\langle$digit$\rangle$
$\langle$identifier$\rangle$ ::= $\langle$identifier-start$\rangle$ $\langle$identifier-part$\rangle$*

Figure 1.1: Some lexical conventions of JavaScript

1.4 Mapping regular expressions to DFAs

The traditional mapping from regular expressions to DFAs goes through a number of steps. First, a regular expression is mapped to an NFA-ε. Next, the ε-transitions are removed to obtain an NFA. Finally, the "power set construction" is applied to construct an equivalent DFA from the NFA. Moreover, in a typical scanner generator, the resulting DFA is minimized to save space in the implementation.

We follow a slightly different approach, which has been used with variations in implementing regular expression search in text editors. The basic idea is to use a set of regular expressions as the set of states for an automaton. As each state q of a finite automaton (the set of words that transform q into a final state) corresponds to a regular language, we simply want to label each state with a regular expression for this language. The initial state q_0 corresponds to the regular expression that we want to recognize. But what is the regular expression for the state $\delta(q_0, a)$? To address this problem, we define the *derivative* of a regular expression, *i.e.*, a function $D : RE(\Sigma) \times \Sigma \to RE(\Sigma)$ such that

$$w \in L(D(r, a)) \qquad \text{iff} \qquad a\,w \in L(r) \tag{1.1}$$

Thus, if $a\,w \in L(r)$ then $D(r, a)$ recognizes the rest, w, of the input word after reading a. This corresponds to the language recognized by the state $\delta(q_0, a)$.

1.2 Definition
The derivative of a regular expression, $D : RE(\Sigma) \times \Sigma \to RE(\Sigma)$ is defined by induction on the definition of $RE(\Sigma)$. It relies on an auxiliary function $E : RE(\Sigma) \to RE(\Sigma)$ which is specified by

$$L(E(r)) = L(r) \cap \{\varepsilon\} \tag{1.2}$$

$$
\begin{array}{lcl}
D(\underline{\emptyset}, a) & = & \underline{\emptyset} \\
D(\underline{\varepsilon}, a) & = & \underline{\emptyset} \\
D(\underline{\mathtt{a}'}, a) & = & \begin{cases} \underline{\varepsilon} & \text{if } a = a' \\ \underline{\emptyset} & \text{otherwise} \end{cases} \\
D(r_1 r_2, a) & = & D(r_1, a) r_2 \mid E(r_1) D(r_2, a) \\
D(r_1 \mid r_2, a) & = & D(r_1, a) \mid D(r_2, a) \\
D(r^*, a) & = & D(r, a) r^* \\
\\
E(\underline{\emptyset}) & = & \underline{\emptyset} \\
E(\underline{\varepsilon}) & = & \underline{\varepsilon} \\
E(\underline{\mathtt{a}}) & = & \underline{\emptyset} \\
E(r_1 r_2) & = & E(r_1) E(r_2) \\
E(r_1 \mid r_2) & = & E(r_1) \mid E(r_2) \\
E(r^*) & = & \underline{\varepsilon}
\end{array}
$$

□

With these definitions, we obtain the following representation theorem for a regular language.

1.3 Theorem

$$L(r) = L(E(r)) \cup \bigcup_{a \in \Sigma} a \cdot L(D(r, a))$$

□

Starting from a regular expression r_0 that defines the language we are interested in, it is now easy to define an automaton that recognizes this language.

1.4 Theorem
Let $r_0 \in RE(\Sigma)$. Define the deterministic automaton $M = (Q, \Sigma, \delta, q_0, F)$ as follows:

- *Q is the smallest subset of $RE(\Sigma)$ such that*
  1. *$r_0 \in Q$;*
  2. *if $r \in Q$ and $a \in \Sigma$ then $D(r, a) \in Q$.*
- *$\delta(q, a) = D(q, a)$*
- *$q_0 = r_0$*
- *$F = \{r \in Q \mid \varepsilon \in L(r)\}$.*

Then $L(M) = L(r_0)$.

The problem with this construction is that the set Q may be infinite. To address this problem, we do not use D directly, but insert an additional pass of simplification. Simplification relies on standard equivalences of regular expressions:

$$
\begin{array}{lclcl}
r\underline{\emptyset} & = & \underline{\emptyset} r & = & \underline{\emptyset} \\
r\underline{\varepsilon} & = & \underline{\varepsilon} r & = & r \\
r \mid \underline{\emptyset} & = & \underline{\emptyset} \mid r & = & r \\
\underline{\emptyset}^* & = & \underline{\varepsilon}^* & = & \underline{\varepsilon} \\
(r^*)^* & = & r^* & &
\end{array}
$$

Using these simplification rules, it is guaranteed that the set of states Q is finite [Brz64] so that the construction actually yields a DFA.

2 Example
For an example, recall part of the regular expression for integer literals from Fig. 1.1.

$$\langle\text{integer-literal}\rangle = (\underline{\varepsilon} \mid \underline{-})\langle\text{digit}\rangle\langle\text{digit}\rangle^*$$

Now

$$\begin{array}{cl}
 & D(\langle\text{integer-literal}\rangle, -) \\
= & D((\underline{\varepsilon} \mid \underline{-})\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\
= & D(\underline{\varepsilon} \mid \underline{-}, -)\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid E(\underline{\varepsilon} \mid \underline{-})D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\
= & (D(\underline{-}, -) \mid D(\underline{\varepsilon}, -))\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid (E(\underline{-}) \mid E(\underline{\varepsilon}))D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\
= & (\underline{\varepsilon} \mid \underline{\emptyset})\langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid (\underline{\emptyset} \mid \underline{\varepsilon})D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\
 & \qquad \textit{apply simplification} \\
= & \langle\text{digit}\rangle\langle\text{digit}\rangle^* \mid D(\langle\text{digit}\rangle\langle\text{digit}\rangle^*, -) \\
 & \qquad \textit{last use of } D \textit{ simplifies to } \underline{\emptyset} \textit{ because } \texttt{-} \textit{ is not a } \langle\textit{digit}\rangle \\
= & \langle\text{digit}\rangle\langle\text{digit}\rangle^*
\end{array}$$

Hence, after reading a `-` the automaton still expects a non-empty word of $\langle\text{digit}\rangle$s. It is not a final state because $\varepsilon \notin L(\langle\text{digit}\rangle\langle\text{digit}\rangle^*)$.

In the same way, we can check that

$$\begin{array}{lcl}
D(\langle\text{integer-literal}\rangle, +) & = & \underline{\emptyset} \\
D(\langle\text{integer-literal}\rangle, \langle\text{digit}\rangle) & = & \langle\text{digit}\rangle^*
\end{array}$$

In the first case, the automaton has reached a *sink state* $\underline{\emptyset}$ which is not a final state and which cannot be left by any transition. In the second case, the automaton has consumed a digit, it has reached a final state but is also ready to read further digits.

1.5 Encoding regular expressions

This section deals with the implementation of regular expressions and the related algorithms in Python. The following code belongs to a module called `Regexp`.

We represent regular expressions using a class for each kind of expression.

```
@dataclass
class Regexp:
    'abstract class for AST of regular expressions'
    def is_null(self):
        return False

@dataclass
class Null (Regexp):
    'empty set: {}'
    def is_null(self):
        return True
@dataclass
class Epsilon (Regexp):
    'empty word: { "" }'
@dataclass
class Symbol (Regexp):
    'single symbol: { "a" }'
    sym: str
@dataclass
class Concat(Regexp):
    'concatenation: r1.r2'
    left: Regexp
    right: Regexp
@dataclass
```

```
class Alternative(Regexp):
    'alternative: r1|r2'
    left: Regexp
    right: Regexp
@dataclass
class Repeat(Regexp):
    'Kleene star: r*'
    body: Regexp
```

The data type `Regexp` can express the regular expression `Concat(x, Null)` which is equivalent to `Null`. Thus, the term language defined by `Regexp` contains ambiguities. Specifically, it is possible to make do with regular expressions which contain no internal `Null` constructors; it is always possible to transform a regular expression into one which is either `Null` or does not contain it at all. Therefore, it is a good idea to abstract over the constructors and perform some simplification on the way. Besides the elimination of internal `Null` constructors, the abstractions also get rid of some `Epsilon` constructors. Moreover, it is useful to nest concatenation and alternatives to the right to obtain a normalized form of regular expressions. Notice that the method `is_null()` provides the means to check whether a normalized `Regexp` represents the empty language.

```
null = Null()
epsilon = Epsilon()
symbol = Symbol
def concat(r1, r2):
    match (r1, r2):
        case (Null(), _) | (_, Null()):
            return null
        case (Epsilon(), _):
            return r2
        case (_, Epsilon()):
            return r1
        case (Concat(r11, r12), _):
            return Concat(r11, concat(r12, r2))
        case _:
            return Concat(r1, r2)
def alternative(r1, r2):
    match (r1, r2):
        case (Null(), _):
            return r2
        case (_, Null()):
            return r1
        case (Alternative(r11, r12), _):
            return Alternative(r11, alternative(r12, r2))
        case _:
            return Alternative(r1, r2)
 def repeat(r: Regexp) -> Regexp:
    match r:
        case Null() | Epsilon():
            return epsilon
        case Repeat(r1):      # r** == r*
            return r
        case _:
            return Repeat(r)
```

Some simple functions are useful in creating composite regular expressions:

```
def optional(r : Regexp) -> Regexp:
    'construct r?'
```

```
    return alternative(r, epsilon)

def repeat_one(r : Regexp) -> Regexp:
    'construct r+'
    return concat(r, repeat(r))

def concat_list(rs : Iterable[Regexp]) -> Regexp:
    return reduce(lambda out, r: concat(out, r), rs, epsilon)

def alternative_list(rs : Iterable[Regexp]) -> Regexp:
    return reduce(lambda out, r: alternative(out, r), rs, null)
```

The expression `optional (r)` is often written as $r?$. The regular expression `repeat_one (r)` is usually written r^+ and it recognizes the language $L(r)^+ = L(r) \cup L(rr) \cup L(rrr) \cup \dots$, *i.e.*, a finite concatenation of words from $L(r)$ where at least one word is present. The functions `concat_list` and `alternative_list` iterate the concatenation and alternation operators:

$$\begin{array}{lcl} \texttt{concat\_list}\ ([r_1; \dots; r_n]) & = & r_1 \dots r_n \\ \texttt{alternative\_list}\ ([r_1; \dots; r_n]) & = & r_1 \mid \cdots \mid r_n \end{array}$$

Now that there is functionality for *creating* regular expressions, the next job is to check, for a given sequence of alphabet symbols `symbols`, if it belongs to the language defined by a regular expression `regexp`. The function `matches` will do exactly that.

```
def matches(r : Regexp, ss: str) -> bool:
    i = 0
    while i < len(ss):
        r = after_symbol(ss[i], r)
        if r.is_null():
            return False
        i += 1
    # reached end of string
    return accepts_empty(r)
```

When reaching the end of the input, this code calls a function `accepts_empty` that checks if the empty sequence belongs to the language of the regular expression. (The implementation of `accepts_empty` is a simple exercise, cf. function E in Sec. 1.4.)

Now that the empty sequence is covered, non-empty sequences are next. This becomes easy in the presence of an auxiliary function `after_symbol`, which implements the derivative function D from Sec. 1.4. This function has the following behavior:

Let r be a regular expression describing the language $L(r)$. Let $x\xi$ be a seqence of symbols. Then:

$$x\xi \in L(r) \iff \xi \in L(\texttt{after\_symbol}(x, r))$$

Thus, `after_symbol` "subtracts" x from r.

The function `after_symbol` is defined by induction on the structure of a regular expression.

```
def after_symbol(s : str, r : Regexp) -> Regexp:
    'produces regexp after r consumes symbol s'
    match r:
        case Null() | Epsilon():
            return null
        case Symbol(s_expected):
            return epsilon if s == s_expected else null
```

```
        case Alternative(r1, r2):
            return alternative(after_symbol(s, r1), after_symbol(s, r2))
        case Concat(r1, r2):
            return alternative(concat(after_symbol(s, r1), r2),
                        after_symbol(s, r2) if accepts_empty(r1) else null)
        case Repeat(r1):
            return concat(after_symbol(s, r1), Repeat(r1))
```

(The proof for the correctness of `after_symbol` is a simple exercise.)

The `matches` function implements a deterministic automaton with `after_symbol` as its state transition function.

1.6 A real scanner

The `matches` function of the previous section is not directly usable for lexical analysis: a scanner must recognize a number of different lexeme languages, it must consider a sequence of (potentially different) lexemes, and the scanner must turn each lexeme into a token/attribute pair. In addition, ambiguities can arise if the lexeme languages have overlaps. Hence, the description of a scanner comprises not just a single regular expression, but rather a whole bunch of them, together with instructions on how to turn the lexemes into token/attribute pairs. Further, to resolve the ambiguities a scanner is not quite a DFA, but rather needs additional structure.

1.6.1 Scanner descriptions

Instructions on how to turn lexemes into token/attribute pairs can be expressed as follows:

```
class Token:pass #abstract
Position = int                    # input position
lex_result = tuple[Token, Position]
lex_action = Callable[[str, Position, Position], lex_result]
```

This declaration assumes that the input is represented as a string and the current input position as an `int`, which is renamed to `Position` for clarity. The type for tokens, `Token`, is left abstract. Attributes are introduced by subclasses of `Token` A `lex_action` is a function to be associated with a regular expression. Its parameter is a tripel (*input*, *start*, *end*) consisting of the underlying input string, the start position of the extracted lexeme, and its end position. The function returns the token and the position in the input where scanning can continue. The `lex_action` may advance the position further to implement sophisticated styles of comments, for instance.

A rule in a scanner description simply pairs up a regular expression with a `lex_action`:

```
@dataclass
class Lex_rule:
    re : Regexp
    action: lex_action
```

1.6.2 Scanner states

The job of a scanner is to successively consume symbols from the input, and, on recognizing a completed lexeme, to call the corresponding `lex_action`. To this end, the scanner must keep a state around which tracks which regular expressions may still match the part of the input consumed so far:

```
Lex_state = list[Lex_rule]
```

When the scanner consumes a symbol, it applies to all regular expressions of a `lex_state` the `after_symbol` function (just like `matches`), and filters out the sink states[1]:

```
def next_state(state: Lex_state, ss: str, i: int):
    return list(filter(lambda rule: not (rule.re.is_null()),
                [Lex_rule(after_symbol(ss[i], rule.re), rule.action)
                 for rule in state]))
```

A scanner description (a list of `lex_rules`) turns easily into an initial state for the scanner automaton:

```
def initial_state(rules: list[Lex_rule]) -> Lex_state:
    return rules
```

To determine which regular expressions have matched the consumed lexeme completely, the scanner uses the `matched_rules` function:

```
def matched_rules(state: Lex_state) -> Lex_state:
    return [rule for rule in state if accepts_empty(rule.re)]
```

It is possible for the scanner to end up in a state where no further consumption of input symbols is possible. The `is_stuck` predicate diagnoses this situation:

```
def is_stuck(state: Lex_state) -> bool:
    return not state
```

1.6.3 Resolution of ambiguities

Descriptions of lexical analysis for realistic programming languages almost always contain ambiguities because it is more convenient to specify overlapping lexeme languages. The fragments:

```
if (n < 0) then return 0 else return n * fib(n-1)
```

and

```
ifoundsalvationinapubliclavatory
```

are syntactically correct JavaScript fragments starting with `if`. Now, the lexical syntax of JavaScript would allow to partition `ifoundsalvationinapubliclavatory` into lexemes in several different ways:

- either into keyword `if` and identifier `oundsalvationinapubliclavatory` or
- just as identifier `ifoundsalvationinapubliclavatory`.

Obviously (or is it?), the latter alternative is the intended one.

The standard way of resolving this conflict is the *rule of the longest match* or the *maximum munch rule*: The first lexeme of a character sequence is its longest prefix which is a lexeme. To find the longest prefix, even if the scanner recognizes a lexeme, it must continue examining characters of the input until the current prefix is no longer a prefix of a lexeme. Then the scanner returns the last lexeme recognized. This process may involve returning characters to the input.

If there are still two different ways of tokenizing a single lexeme, then the textually preceding rule in the specification is given preference.

[1] See the appendix A.1.3 for the definition of `filter`.

1.6.4 Implementation of lexical analysis

All the building blocks for implementing lexical analysis are now in place. This section describes the central functionality for creating scanners. The main workhorse is the method `scan_one_token` in class `Scan`; it runs the state automaton starting from the specification in field `spec` to extract a single lexeme at the beginning of the input. To implement the "longest match" rule, `scan_one_token` remembers the last state in which it recognized a lexeme. Once `scan_one_token` has recognized a lexeme, it runs the corresponding action from the scanner description to yield a token and the starting position of the input still to be processed.

```
class ScanError (Exception): pass

@dataclass
class Match:
    action: lex_action
    final : Position

@dataclass
class Scan:
    spec: Lex_state

    def scan_one(self) -> Callable[[str, Position], lex_result]:
        return lambda ss, i: self.scan_one_token(ss, i)

    def scan_one_token(self, ss: str, i: Position) -> lex_result:
        state = self.spec
        j = i
        last_match = None
        while j < len(ss) and not is_stuck(state):
            state = next_state(state, ss, j); j += 1
            all_matches = matched_rules(state)
            if all_matches:
                this_match = all_matches[0]
                last_match = Match(this_match.action, j)

        match last_match:
            case None:
                raise ScanError("no lexeme found:", ss[i:])
            case Match(action, final):
                return action(ss, i, final)
        raise ScanError("internal error: last_match=", last_match)
```

A scanner for a given programming language may consist of several parts, each with its own scanner description. The components may be specified in different instances of the `Scan` class. Given a function which recognizes a single lexeme, it is easy to construct the complete scanner which turns the input—a list of symbols—into an iterable of tokens:

```
def make_scanner(
        scan_one: Callable[[str, Position], lex_result],
        ss: str
    ) -> Iterator[Token]:
    i = 0
    while i < len(ss):
        (token, i) = scan_one(ss, i)
        yield token
```

1.6.5 An example specification

As an example, we show some excerpts from the specification of a JavaScript scanner.

First, there is a number of definitions for regular expressions. The auxiliary function `char_range_regexp` takes two characters, `c1` and `c2`, and constructs a regular expression denoting the set of characters between `c1` and `c2`, inclusive.

```
def char_range_regexp(c1: str, c2: str) -> Regexp:
    return alternative_list(map(symbol, map(chr, range(ord(c1), ord(c2)+1))))
```

Another function turns a fixed string (e.g., a keyword) into a regular expression that recognizes that string.

```
def string_regexp(s: str) -> Regexp:
    return concat_list(map(symbol, s))
```

Finally, a function that turns the characters in a string in an alternative:

```
def class_regexp(s: str) -> Regexp:
    return alternative_list(map(symbol, s))
```

The regular expression for integer literals are taken directly from the specification in Fig. 1.1.

```
digit = char_range_regexp("0","9")
hexdigit = alternative_list([digit, char_range_regexp("A", "F"),
                                    char_range_regexp("a", "f")])
hexprefix = alternative(string_regexp("0x"), string_regexp("0X"))
sign = optional(symbol('-'))
integer_literal = alternative( concat(sign, repeat_one(digit)),
                               concat_list([sign, hexprefix, repeat_one(hexdigit)]))
```

Identifiers are specified as follows:

```
letter = alternative (char_range_regexp ('A', 'Z'), char_range_regexp ('a', 'z'))
identifier_start = alternative_list([letter, symbol('$'), symbol('_')])
identifier_part = alternative(identifier_start, digit)
identifier = concat(identifier_start, repeat(identifier_part))
```

A whitespace lexeme consists of a non-empty sequence of blanks, tabulators, newline, carriage return, and form feed characters.

```
blank_characters = "\t "
line_end_characters = "\n\r"
white_space = repeat_one(class_regexp(blank_characters + line_end_characters))
```

Next, we define datatypes for the tokens of the JavaScript language.

```
@dataclass
class Return(Token): pass
@dataclass
class Intlit(Token): value: int
@dataclass
class Ident(Token): name: str
@dataclass
class Lparen(Token): pass
@dataclass
class Rparen(Token): pass
@dataclass
class Slash(Token): pass
@dataclass
class Strlit(Token): value: str
```

The scanner specification itself is a list of pairs (i.e., `Lex_rules`) of a regular expression and an action function, as explained above. Typically, we define a scanner as a recursive function, so that it can call itself recursively to consume further input. The action for whitespace is an example.

```
js_spec: Lex_state = [
    Lex_rule(string_regexp("return"), lambda ss, i, j: (Return(), j)),
    Lex_rule(integer_literal, lambda ss, i, j: (Intlit(int(ss[i:j])), j)),
    Lex_rule(identifier,    lambda ss, i, j: (Ident(ss[i:j]), j)),
    Lex_rule(white_space,   lambda ss, i, j: js_token(ss, j)),
    Lex_rule(symbol("("),   lambda ss, i, j: (Lparen(), j)),
    Lex_rule(symbol(")"),   lambda ss, i, j: (Rparen(), j)),
    Lex_rule(symbol("/"),   lambda ss, i, j: (Slash(), j)),
    Lex_rule(string_literal, lambda ss, i, j: (strlit(ss[i+1:j-1]), j))
]
js_token = Scan(js_spec).scan_one()
```

Finally, we define the `scan` function :

```
def scan(ss: str):
    return make_scanner (js_token, ss)
```

The example also demonstrates, in the rule for string literals, how to integrate subsidiary scanners. String literals are tricky because they require a special treatment of escape characters like \ and quotation characters like ". Here is a simple example inspired by string literals in C, which distiguishes normal printable characters from the special characters \":

```
escaped_char = concat(symbol('\\'), alternative(symbol('\\'), symbol('"')))
content_char = alternative_list([symbol(chr(a))
                                 for a in range(ord(' '), 128)
                                 if a not in [ord('\\'), ord('"')]])
string_literal = concat_list([symbol('"'),
                              repeat(alternative(escaped_char, content_char)),
                              symbol('"')])
```

The main scanner just recognizes a string literal, but its action passes the body between the leading and trailing double quote to a subsidiary scanner that processes the special characters in the string body:

```
    Lex_rule(string_literal, lambda ss, i, j: (strlit(ss[i+1:j-1]), j))
```

The `strlit` function relies on a token scanner for `escaped_char` and `content_char` to implement this transformation:

```
string_spec: Lex_state = [
    Lex_rule(escaped_char, lambda ss, i, j: (ss[i+1], j)),
    Lex_rule(content_char, lambda ss, i, j: (ss[i], j))
]
string_token = Scan(string_spec).scan_one()

def strlit(ss: str) -> Strlit:
    "use subsidiary scanner to transform string content"
    return Strlit("".join(make_scanner(string_token, ss)))
```

1.7 Pragmatic issues

1.7.1 Recognizing keywords

One way of recognizing keywords is to include them as constant regular expressions in the scanner specification. Unfortunately, this approach can give rise to automata with a huge number of states in the traditional approach and it also leads to inefficiencies in the library-based approach that we are propagating. Hence, keyword recognition is often handled separately from scanning in the following manner.

1. Build a hash table from the keywords before starting the scanner.
2. Specify the scanner so that it recognizes all keywords as identifiers.
3. On recognizing an identifier lexeme, the scanner first checks the hash table. If the lexeme is present it is classified as a keyword. Otherwise, the scanner reports an identifier.

The hash table is constructed only once and its lookup should be performed as quickly as possible. Hence, it is appropriate to spend some effort into its construction. It is possible to search for a perfect hash function that avoids collisions because all entries of the hash table are a-priori known. This way, a lookup in the hash table can be guaranteed to run in constant time.

1.7.2 Representing identifiers

Strings are not a good representation for identifiers. In particular, later phases of compilation build so-called symbol tables that map an identifier to some information about it. Because identifier lookups occur very frequently, it is vital that these mappings are implemented efficiently. Each lookup operation involves comparison and/or computation of a hash key. Strings perform poorly with both types of operation:

- A string comparison takes time linear in the length of the string.
- Computing a (meaningful) hash key for a string is not straightforward, because several characters must be extracted from the string.

Hence, a scanner maps the strings arising as identifier lexemes to *symbols* using an open hashing algorithm and assigns a unique identifier (a number) to each entry in the table.

Bibliography

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.

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

[Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

[Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.

[Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

[Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Lee93] Rene Leermakers. *The Functional Treatment of Parsing.* Kluwer Academic Publishers, Boston, 1993.

[PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.

[PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1990.

[ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, USA, June 1995. ACM Press.

[ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung.* Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.

[WM96] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung — 2 Auflage.* Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1996.