For the type-checking purposes we would consider that match has always the following form:

```
match [\prod1] v = regexp  ->  [\prod2] f (v)
```

It means that in the beginning of the production `prod1` we check that stream of pseudo-tokens is recognized by the regular expression `regexp`. The matched result can be aliased in F using variable named `v`. Now, `f (v)` is an application of a function called `f` to the list of pseudo-tokens matched by `regexp`.

The main purpose of `f` is to make a transformation of the list of pseudo-tokens, generating a new input for `prod2`. The purpose of the type-checker in F is to give a static guarantees that the return type of `f` application would be recognized by `prod2`. In order to do that, we have to construct a pseudo-token based regular expression of `prod2` and to infer a type of application of `f` to `v`. We are not concerned in this paper with a question who is going to provide a regular expression for `prod2` it could be either a programmer, or it could be done in an automatic way. It is important to mention, that the transition from the grammar rule to the regular expression looses a certain information, like for example there is no way to construct a precise regular expression for a language:

```
A :=  aAb | ab
```

However, it is always possible to construct a regular expression for super-language, for example `a+b+`, and the types are considered to be correct, if we can prove that given regular expression recognizes the language produced by `f (v)`.

As we want to have static guarantees, we need to bring a regular expression at the level of types. On the other hand, the transformation in F are happening on the lists of pseudo-tokens. As a solution we are going to combine these two entities. So every cons-list except the standard signature is going to include a regular expression in its type.

Most of the functions of F are generic with respect to a regular expression, i.e. functions operate with any lists, without considering an overall structure of the list, e.g reverse, map, etc. We are going to use an abstract type called `regexp_t` in the function definition and perform a type-inference of the function applications. In our setting we always start with an application of a function with a concrete instance of the `regexp_t`.

It is easy to notice that the regular expressions bring a natural sub-typing hierarchy which is based on the fact that the language recognized by r_1 can include the language recognized by r_2. If this happens, we say that $r_1 \sqsupseteq r_2$ or $r_2 \sqsubseteq r_1$. We also have a notion of a super-type of the hierarchy, which is given by a regular expression `.*`, we are going to denote this type $\top$. It is obvious to see that $\forall t_i \in R, t_i \sqsubseteq \top$.

For further type inference procedure we have to know how to check if $r_1 \sqsubseteq r_2$. This is fairly easy to check constructively. As we know, we can always build a DFA for a regular expression and minimize it, which gives us a minimal possible automaton for the language recognized by a given regular expression. It means that $r_1 \sqsubseteq r_2 \Rightarrow min(det(r_1)) \sqsubseteq min(det(r_2))$. For two minimized automatons A_1 and A_2, $A_1 \sqsubseteq A_2$ means that there is a mapping Ψ of A_1 states to A_2 states such that:

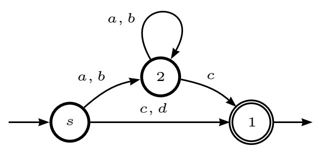
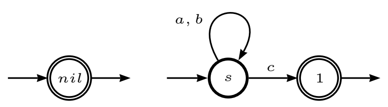
$$Start(A_1) \to Start(A_2) \in \Psi$$

$$\forall s \in States(A_1) \forall e \in Edges(s), \Psi(Transition(s, e)) = Transition(\Psi(s), e)$$

$States(x)$ denotes a set of all the states of automaton x, $Edges(s)$ is a set of pseudo-tokens which mark the outcoming edges of state s. Finally, $Transition(s, t)$ denotes a state which is reachable from s, using edge marked with t.

The second important property we are going use in the type inference is that for any instance of `regexp_t` we can always construct a regular expression for $head(x)$ and $tail(x)$. The type for $head(x)$ may be constructed as a set of all the possible transitions from the starting state of the automaton implementing the type of x. Obviously the type of the $head(x)$ can be constructed as an instance of `regexp_t` by joining all the symbols with `|` construct, but it would be easier for us to keep it as as a set.

The type for $tail(x)$ is constructed in a similar way as $head(x)$, but the resulting set will consist of the sub-automatons of x with a first transition done. In order to illustrate that consider the following example.

Automaton for `(a|b)*c|d`	Set of types for $tail(x)$
	

We can see that the type for $tail(x)$ is a set: $\{[], (a|b) * c\}$.

The next observation that we are going to use in the type-inference is the length restriction that each element of the set brings and potential error-type denoted with $\bot$. Consider an example when τ :: `a*cd*` and we want to infer a type for $head(head(\tau)) \vdash \sigma$. The problem one can anticipate is when instance of τ is $cons(c, nil)$, in which case $\sigma = \{\bot\}$, which means that in general $\tau = \{\dots, \bot\}$, and this is a type-error. In order to handle this case, we are going to assign a length constraint to each element of the set, i.e

$$head(\tau^n) \vdash \{(a : a * cd*, n > 1);\ (c : d*, n \geq 1)\}$$

and

$$head(head(\tau^n)) \vdash \{(a : a{*}cd{*}, n > 2);\ (c : d*, n \geq 2);\ (d : d*, n \geq 2);\ (\bot, n < 2)\}$$

In this case we would not produce a type-error immediately, but we would rather continue the type inference. In case $(\bot, p(n)$ is a member of a resulting set, we may reject the program being type incorrect, or leave this case for dynamic type-checker. In this paper we are going to use the first option as a user can always wrap an operation producing such a type inside the condition, providing enough information for the type-checker to eliminate this option. We are going to consider such an example further down.

As functions in F are mainly used to transform the lists, and recursive applications of *head* and *tail* is the only possible mechanism, we have to infer types for such a recursive functions. The fact that we are using here is that it is possible to construct types for *head* and *tail* closures when traversing a list forward and backwards. Further down, we are going to use a $head(\uparrow^*)$, $tail(\uparrow^*)$ to denote a froward head/tail closure and $head(\downarrow^*)$, $tail(\downarrow^*)$ to denote a backward head/tail closure.

In order to prove the fact that the type for closure exists, we are going to describe a constructive procedure for building one. The forward head/tail closures can be constructed by traversing a DFA making each vertex an initial state, and providing a sub-automaton reachable from this vertex as a type. The number of vertices is finite, so the resulting type of closure would contain N elements in worth case.

In order to construct a backward head/tail closure, one has to perform exactly the same procedure as in case of forward closure, but on the reversed regular expression. In order to build a reversed regular expression there are two well-known ways. First one is to flip all the edges in the automaton, make a new initial state, put ϵ transitions from this state to every accepting state of the original automaton and run the deterinisation procedure. Another way of doing it would be using the following rewriting rules:

$$
\begin{aligned}
&\forall x, y \in \Re, \forall s \in \Sigma && (1)\\
Rev &:: \Re \to \Re && (2)\\
Rev(\epsilon) &= \epsilon && (3)\\
Rev(s) &= s && (4)\\
Rev(xy) &= Rev(x)Rev(y) && (5)\\
Rev(x*) &= Rev(x)* && (6)\\
Rev(x|y) &= Rev(x)|Rev(y) && (7)
\end{aligned}
$$

In our case, as a resulting type allows a set of automatons we may use the following approach: flip all the edges in the automaton A, creating new automaton A'; for each accepting state of A' treat it as initial state, and build a sub-automaton reachable from this state; put all the sub-automatons in the set, and apply a forward-closure procedure for each element in the set.

`FIXME: We may want to construct an example here.`

The last important fact we would like to discuss is how the condition inside the if-expressions participate in the type-inference. In F we allow to use two built-in operators namely *type* and *len*. The *type* returns a type of an object in run-time, and *len* returns a length of an instance of `regexp_t`. The *type* construct may be used inside the if-expressions, comparing a type of object with some type using $\doteq$ operator. For example:

```
if type (x) == a|b
    ...
else
    ...
```

In that case, the type-checker will know that inside the if-branch of the if-expression the type of x will be $a|b$ and the else-branch will be $type(x) - a|b$. Now, we just have to show how to construct $\tau - \sigma$. As τ and σ are both instances of `regexp_t`, let T and S be respective DFAs for the types. Now, what we have to do is to construct $C = T \times S$ and make the final states of C be the pairs where T states are final and S states are not. The product of two automatons $A \times B$ is defined as following: Given $A_1 = (Q_1, \Sigma, \delta_1 q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2$ we can define a product $A = A_1 \times A_2$ being $A = (Q, \Sigma, \delta, q, F)$ where

$$\text{set of states} \qquad Q = Q_1 \times Q_2 \tag{8}$$
$$\text{transition function} \qquad \delta \rightarrow (r_1, r_2).a = (r_1.a, r_2.a) \tag{9}$$
$$\text{initial state} \qquad q = (q_1, q_2) \tag{10}$$
$$\text{accepting states} \qquad F = F_1 \times F_2 \tag{11}$$

`FIXME: We may want to construct an example here.`

One can also define intersection and subset relationships on regular expressions, but in order to keep the language simple we would not consider it in this paper.

Lambda definitions of F.

$$\begin{array}{rcl}
expr & ::= & var \mid \lambda var.expr \mid (expr expr) \\
 & & \mid \textbf{letrec}\ var = expr, ...\ \textbf{in}\ expr \\
 & & \mid \textbf{if}\ \text{if-expr}\ expr\ \textbf{else}\ expr \\
 & & \mid \text{const-int} \mid func \\
func & ::= & \textbf{head} \mid \textbf{tail} \mid \textbf{cons} \mid + \mid - \mid \dots \\
\text{if-expr} & ::= & \textbf{type}\ var\ \doteq regexp \mid \textbf{len}\ var\ relop\ \text{int-expr} \mid expr \\
regexp & ::= & \dots
\end{array}$$