From String to AST: parsing

PDF-XChange Product
Click to buy NOW!
www.tracker-software.com

Whether you have to do with data in form of CSV, JSON or a full-blooded programming language like C, JavaScript, Scala, or maybe a query language like SQL, you always transform some sequence of characters (or binary values) into a structured representation. Whatever you'll do with that representation depends on your domain and business goals, and is quite often the core value of whatever you are doing. With a plethora of tools doing the parsing for us (including the error-handling), we might easily overlook how complex and interesting process it is.

Formal grammars

First of all, most input formats that we handle follow some formal definition, telling e.g. how key-values are organized (JSON), how do you separate column names/values (CSV), how do you express projections and conditions (SQL). These rules are defined in an unambiguous way so that the input could be interpreted in a very deterministic way. It directly opposed language we use to communicate with other people, which often is ambiguous and put into the context. *Wanna grab some burger?* might be a nice suggestion if you are talking to a colleague that have to skip lunch and likes burgers, but might be offensive if told in a sarcastic tone to someone who doesn't like meat. Then, words can have different meaning depending on the culture you are currently in, in which times you live or what are you and your conversationalist social position (*vide*, e.g. Japanese and how your position and suffixes you add at the end of the name change the tone of the whole conversation). Languages we use when communicating with a computer must be free of such uncertainties. The meaning should depend only on the input we explicitly entered and interpreted deterministically. (Just in case: by deterministically, I mean, deterministically interpreted, which doesn't mean that it would always produce the same result. If I write `currentTimeMillis()`, the function will always return a different result, but the meaning will be always the same - compiler/interpreter will understand that I want to call `currentTimeMillis()` function, and it won't suddenly decide that I want to e.g. change the compiler flag. Of course, the meaning of the function can change in time - for instance, if I edit the source code in between runs - and surely the value returned by it, which is bound to time).

Initially, it wasn't known, how to parse languages. The reason, that we had to start with punching cards, sometime later moved on to assembly, and later on invent Fortran and Lisp, go through whole spaghetti code with Basic, get *The case against goto statement* by Dijkstra, until we could - slowly - started developing more sophisticated compilers we have today, was that there were no formal foundations to it.

linguists know, that we can distinguish some *parts of speech* like: *noun* (specific thing, e.g. *Alice*, *Bob*), *pronoun* (generic replacement for a specific thing, e.g. *I*, *you*, *he*, *she*), *verb* (action), *adjective* (description or trait of something, e.g. *red*, *smart*), etc. However, the also know that the function of part of speech changes depending on how we construct a sentence - that's why we also have *the parts of the sentence*: *subject* (who performs the action: e.g. *Alice* in *Alice eats dinner*), *object* (who is the target of the action, e.g. *dinner* in *Alice eats dinner*), *modifiers* and *compliments*, etc. We can only tell which part of the speech and sentence the word is in the context of a whole sentence:

- An alarm *is set* to 12 o'clock - here, *set* is a verb,
- This function returns an infinite *set* - here, *set* is a noun and an object,
- The *set* has the cardinality of 2 - here, *set* is a noun and a subject,
- All is *set* and done - here, *set* is an adverb and a modifier.

As we can see the same work might be a completely different thing depending on the context. This might be a problem when we try to process the sentence bottom-up, just like we (supposedly) do when we analyze them in English lessons. *This is a noun. That is a verb. This noun is subject, this verb is an object. This is how subsentences relate to one another. Now we can analyze the nice tree of relations between words and understand the meaning.* As humans, we can understand the relationship between the words on the fly, the whole exercise is only about formalizing our intuition.

But machines have no intuition. They can only follow the rules, we establish for them. And when dealing with computers we quite often establish them using the divide-and-conquer strategy: split the big problem into smaller ones, and then combine the solutions. With **natural languages** the context makes it quite challenging, which is why no simple solution appeared even though we were regularly trying. Current progress was made mostly using machine learning, which tackles the whole problem at once, trying to fit whole parts of the sentence as patterns, without analyzing what is what. However, when it comes to communication with a computer, ambiguities can be avoided, simply by designing a language in a way that doesn't allow them. But how to design a language?

One of the first researchers, that made the progress possible was Noam Chomsky. Interestingly, he is not considered a computer scientist - he is (among others) linguists, who is credited with cognitive revolution. Chomsky believes, that how we structure languages is rooted in how our brains process speech, reading, etc. Therefore similarities between languages' structures (parts of speech, parts of sentences, structuring ideas into sentences in the first place, grammar cases) are a result of how processes inside our brain. While he wasn't the first one who tried to formalize a language into a **formal grammar** (we know of e.g. Pāṇini), Chomsky was the first to

…ize **generative grammars**, that is grammars where you define a set of rules… create a language by combining the rules.

…n we define these rules? Well, we want to be able to express each *text* in… grammar as a tree - at leaves, we'll have *words* or *punctuation marks* of sorts. Then, there will be nodes aggregating words/punctuation marks by their function (part of a sentence). At the top of the tree we'll have a root, which might be (depending on grammar) a sentence/a statement/an expression, or maybe a sequence of sentences (a *program*). The definitions will work this way: take a node (starting with root) and add some children to it: the rules will say how the specific node (or nodes) can have children appended (and what kind of children). The grammar definitions will rarely be expressed with specific values (e.g. you won't write down all possible names), but rather using symbols:

$$\text{Sentence} \rightarrow \text{Subject verb Object.}$$

$$\text{Subject} \rightarrow \text{name surname} \mid \text{nickname}$$

$$\text{Object} \rightarrow \text{item} \mid \text{animal}$$

Here, Sentence could be a **start symbol**. We would build a sentence by unrolling notes according to rules. Here there is only one rule going from Sentence - one that allows adding Subject, verb, Object and the dot sign (.) children (order matters!). verb is written with a small capital because it is (or eventually will be) a leaf - since unrolling ends (terminates) at leaves, we would call symbols allowed to be leaves as **terminal symbols**. As you might guess, nodes become **nonterminal symbols**. Terminal symbols will eventually be replaced with an actual word, unless they are **keywords** (have you noticed, how *if*, *else*, *function*, *class*, … get special treatment in many languages?) or special symbols (;, (,), ,, …).

Having, $\text{Subject verb Object.}$, we can continue unrolling. Our second rule lets us turn Subject into name surname or nickname (the vertical line $\mid$ is a shortcut - $\text{A} \rightarrow \text{B} \mid \text{C}$ should be understood as $\text{A} \rightarrow \text{B}$ *or* $\text{A} \rightarrow \text{C}$). And our third rule allows us to turn Object to item or animal. We can go both times with the first option and obtain $\text{name surname verb item.}$ (e.g. *John Smith eats cereals.*). We might go both times with the second option - $\text{nickname verb animal.}$ (*Johnny likes cats.*). And so on.

Notice that in the end, we'll always end up with a sequence of terminals. If we couldn't, there would be something wrong with a language. This definition that takes a sequence of symbols and returns another sequence of symbols is called **production rules**. We can describe each **formal language** as a quadruple $\text{G} = (\text{N}, \Sigma, \text{P}, \text{S})$, where T is a finite set of nonterminal symbols, Σ a finite set of terminal symbols, P is a set of production rules and $\text{S} \in \Sigma$ is a start symbol.

Besides formalization of generative grammars, Chomsky did something else. He was responsible for the organization of formal languages in a hierarchy called after him the **Chomsky hierarchy**.

The Chomsky hierarchy

On the top of the hierarchy are **type-0 languages** or **unrestricted languages**. There is no restriction placed upon how we define such language. A production rule might be any sequence of terminals and nonterminals into any sequence of terminals and nonterminals (in the earlier example there was always nonterminal symbol on the left side - that is not a rule in general!). These languages are hard to deal with, so we try to define data format and programming languages in term of a bit more restrained grammars, that are easier to analyze.

First restriction appears with **type-1 languages** or **context-sensitive grammars** (**CSG**). They require, that all production rules would be in form of:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in N$, $\alpha, \beta \in (N \cup \Sigma)^*$, $\gamma \in (N \cup \Sigma)^+$ (where $*$ and $+$ are Kleene star and Kleene plus). In other words, we can perform $A \rightarrow \gamma$ only, if immediately before A is α and immediately after is β (rule is applied in *context*). But, even these grammars appears to be difficult to deal with. That is why we apply even more restrictions to grammars we use in our everyday life.

More specifically, we might want our grammars to be independent of context. **Type-2 languages** or **context-free grammars** (**CFG**), are CSGs where context is always empty, or in other words, where each production rule is in form of:

$$A \rightarrow \gamma$$

where $A \in N$ and $\gamma \in (N \cup \Sigma)^+$. These grammars, are quite well researched, so we have plenty of tools helping us analyze them, check if something belongs to language, how to generate parser etc. That's why the majority (all?) of programming languages and data inputs are defined using CFGs.

To be precise, when it comes to programming languages, we quite often deal with context-sensitive grammars, but it is easier to deal with them as if they were context-free - call that **syntactical analysis** (what meaning we can attribute to a words basing on their position in a sentence) - and then take the generated tree, called **abstract syntax tree**, and check if makes **semantic** sense (is the name a function, a variable or a type? Does it makes sense to use it in the *context* it was placed?). If we expressed it as a context-sensitive grammar we could do much (all?) of semantic analysis in the same

check syntax, but the grammar could get too complex for us for understa
(or at least to handle it efficiently).

illustrate the difference between syntax and semantics we can get back to yo
earlier example.

$$\text{nickname verb item.}$$

It is a correct semantics in the language. Let's substitute terminals with some specific values.

Johnny eat integral.

What we got is correct according to the rules based on words' positions in the sentence (syntax), but as a whole - when you analyze the function of each word (semantics) - it makes no sense. Theoretically, we could define our language in an elaborate way, that would make sure that there would always be e.g. *eat**s*** after the third person in a sentence and something edible after some form of *to eat* verb, but you can easily imagine, that the number of production rules would explode.

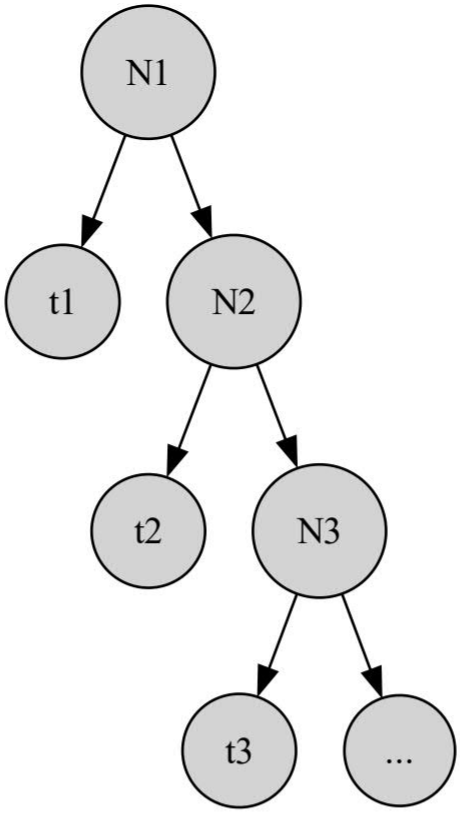
Finally, there is the most restricted kind of grammar in the Chomsky hierarchy. **Type-3 grammar** or **regular grammar** is a language, where you basically either prepend or append terminals. That is each production rule must be in the form of one of:

- $\mathrm{A} \to \mathrm{a}$ - where $\mathrm{A} \in \mathrm{N}$, $\mathrm{a} \in \Sigma$,
- $\mathrm{A} \to \epsilon$, where ϵ in an empty string,
- $\mathrm{A} \to \mathrm{aB}$ - where $\mathrm{B} \in \mathrm{N}$.

(We call it **right regular grammar** - if we instead required that the third rule would be in the form $\mathrm{A} \to \mathrm{Ba}$ it would be **left regular grammar**). While regular grammars are too restricted to define many programming languages on its own, is we'll find out later on, that - when combined with CFG - they allow us to build modern parsers.

Regular languages

Let's start with the most limited grammars, that is regular grammars. No matter how we define production rules, we will end up with a tree of form:



Of course, it doesn't mean, that each such tree would be the same. For instance we could define our grammar like this:

- $$A_0 \rightarrow aA_1$$
- $$A_1 \rightarrow aA_2$$
- $$A_2 \rightarrow aA_3$$
- $$A_3 \rightarrow \epsilon$$

If we started from A_0, the only possible sentence in such language would be $aaa\epsilon$. And - since ϵ represents an empty string, it would be actually just aaa. Another example could be grammar like this:

- $$S \rightarrow aB$$
- $$B \rightarrow bB \mid bC$$
- $$C \rightarrow c$$

If our starting symbol would be S, the sentences we could accept as belonging to grammar would be abc, $abbc$, $abbbc$, ... If you've been programming for a while and you ever had to find some pattern in a text, you should have a feeling that looks familiar. Indeed, the regular language is formalism used to describe the **regular expressions**.

The first example you be described just as aaa, while the second as $a(b+)c$ (or $ab(b*)c$). Here, $*$ and $+$ corresponds directly with Kleene star and Kleene plus. Now, that we know we are talking about regexpes, we can provide another definition of what could be a regular language, that would be equivalent to production-rule-based, but easier to work with.

A regular expression is anything build using the following rules:

- is a regular expression accepting the empty word as belonging to the language,
- a regular expression accepting `'a'` belonging to some alphabet A (nonterminals) as a word belonging to the language,
- when you **concatenate** two regular expressions, e.g. AB, you accept words made by concatenating all valid words in A with all valid words in B (e.g. if a accepts only `"a"` and b accepts only `"b"`, then ab accepts `"ab"`),
- you can **sum up** regular languages $A \mid B$, to accept all words valid either in A or in B (e.g. $a \mid b$ would accept `"a"` or `"b"`, but not `"ab"`),
- you can use Kleene star A^* and Kleene plus A^+, to define (respectively) *any number of occurrences* or *at least one occurrence* of a pattern A as words accepted by regular language (e.g. $a(b^+)c$).

That is enough to define all regular expressions, though usually, we would have some utilities provided by regexp engines, e.g. $[a-z]$, which is a shortcut for $a \mid b \mid ... \mid y \mid z$ or $A?$ which is a shortcut for $A \mid \epsilon$. Just in case, I'll also mention, that some implementations of regexp allows so-called *backreferences* - expression build using there are no longer regular languages, which has some practical implications. What are these implications?

Well, we haven't discussed it so far, but there are some very close relationships between types of formal grammars and computation models. It just happens, that if we wanted to define a function checking whether a word/sentence/etc belongs to a regular grammar/a regular expression - which is equivalent to defining the language - is done by defining a **finite-state automaton** (**FSA**), that accepts this language. And vice-versa, each FSA defines a regular language. That correspondence dictates, how we implement regexp patterns - basically, each time we compile a regexp pattern, we are building a FSA, that would accept all words of grammar and only them.

In case you've never met FSA, let us remind what they are. Finite-state automaton or **finite-state machine** (**FSM**) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of **states**,
- Σ is a finite set of **input symbols** (an **alphabet** - equal to set of terminals without ϵ),
- a **transition function** $\delta : Q \times \Sigma \to Q$, which would take the current state and next input symbol to return the next state,
- an **initial state** $q_0 \in Q$,
- a set of accepting states $F \subseteq Q$.

---

*On a side note: an automaton - singular, meaning a machine, automata - plural, meaning machine**s**. Other nerdy words which works like that: a criterion vs criteria.*

For instance: our alphabet contains 3 possible characters $\Sigma = \{a, b, c\}$. We want to build finite state machine accepting regular expression $a(b^*)c$. Such machine:

- would have to start with a state indicating that nothing was yet matched, but also that nothing is wrong yet. Let's mark it as q_0,
- if first incoming input symbol is a, everything is OK, and we can move on to matching $(b)^*$. However, if b or c arrives, we can already tell, that the result is wrong. Let's mark error as e. On e no matter, what comes next, we'll just stay at e state,
- in this particular case we can safely assume, that if things started to go wrong, there is no way to recover, but it is not a general rule (if there was e.g. an alternative, then failing to match one expression, wouldn't mean that we will fail to match the other expression),
- to indicate that we matched a, let's create a new state q_1. We have to do it, because FSM (which shares its acronym with its noodle excellency Flying Spaghetti Monster) can only remember the current state it is in, so we want to change the behavior (and when we move to match b^* we will), we can only achieve that, by creating a different state for each behavior (and step of the algorithm) and using state transition to indicate progress of computation,
- OK, we arrived at state q_1, so a was matched, and we want to match b^*. b^* means that there could be 0 or more occurrences of b. In case there is 0, we have to match whatever is immediately after b^*, that is c. Accidentally, that would be the case when we are accepting input. Let's mark that case as q_2 and let's put it into the set accepting states,
- In case we are in q_1 and b arrives, we can… simply keep the current state. Until anything from $\Sigma - \{b\}$ arrives, we can reuse current state,
- if we are in q_1 and a arrives, input is wrong and we'll go to e,
- at this point, we are at q_2 (or e which means that things mismatched and we cannot recover). q_2 is accepting state, so, if there is nothing else, we matched the input. However, if there is anything else incoming, this means we have e.g. $abbca$ which shouldn't be matched. So no matter what will come, we are moving to e.

What we defined right, now could be described like this:

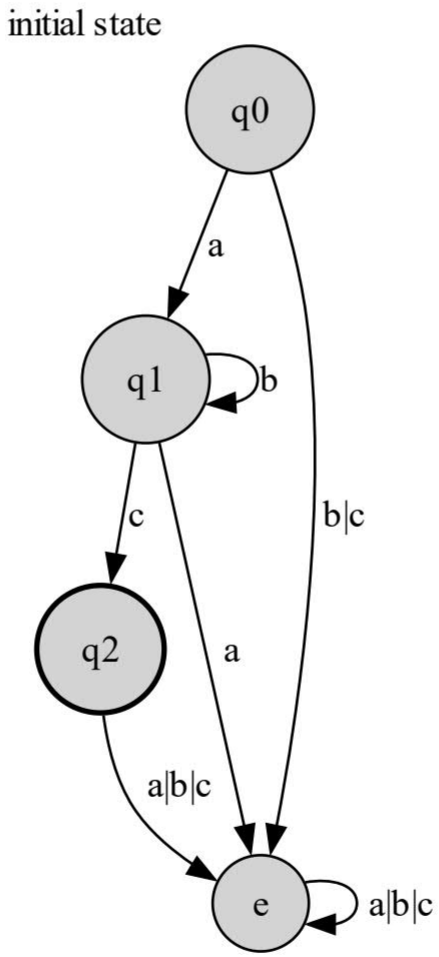
$$\Sigma = \{a, b, c\}$$

$$Q = \{q_0, q_1, q_2, e\}$$

$$
\begin{aligned}
\delta = \{&(q_0, a) \to q_1,\\
&(q_0, b) \to e,\\
&(q_0, c) \to e,\\
&(q_1, a) \to e,\\
&(q_1, b) \to q_1,\\
&(q_1, c) \to q_2,\\
&(q_2, a) \to e,\\
&(q_2, b) \to e,\\
&(q_2, c) \to e,\\
&(e, a) \to e,\\
&(e, b) \to e,\\
&(e, c) \to e\}
\end{aligned}
$$

$$F = \{q_2\}$$

We could also make it more visual (bold border for accepting state):

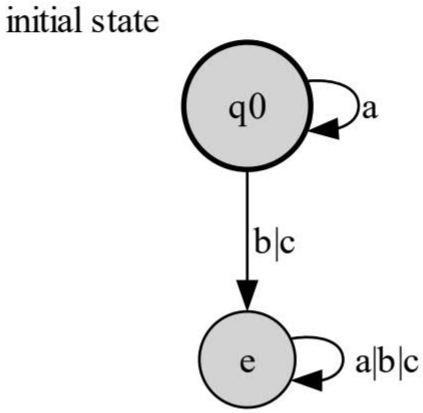


As we can see, each state has to have defined transition for every possible letter of the alphabet (even if that transition is returning the current state as the next state). So, the size of machine definition (all possible transitions) is $|Q| \times |\Sigma|$.

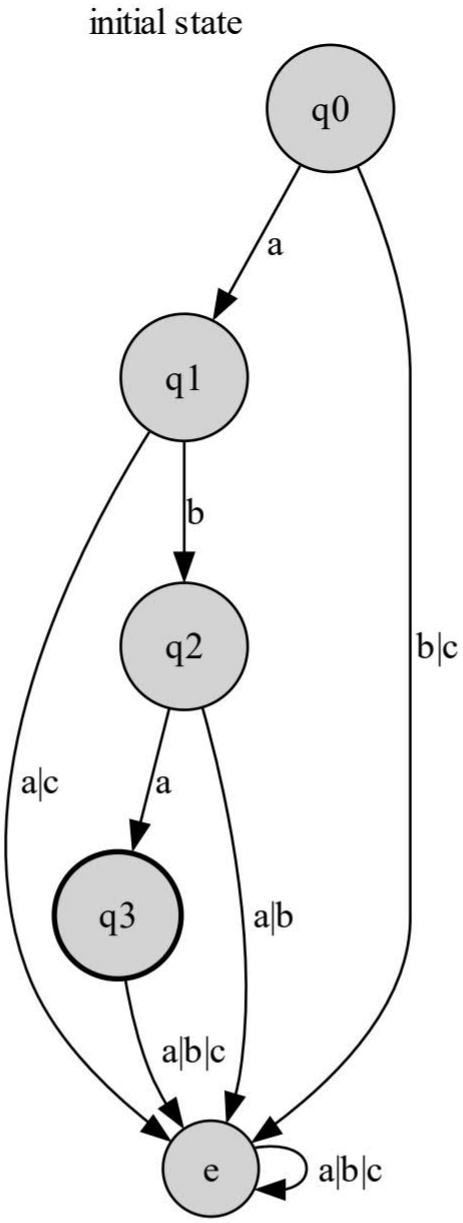
Additionally, constructing the machine required some effort. We would like to automate the generation of FSM from regular expressions, and creating it in the final version might be troublesome. What we created is actually called **deterministic finite state machine** / **deterministic finite automaton** (**DFA**). It guarantees, that every single time we will deterministically get accepted a state for accepted input and non-accepted state for non-accepted input.

In practice, it is usually easier to define a **non-deterministic finite automaton (NFA)**. The difference is that NFA **can have several possible state moves for each state-input pair and picks one at random**. So, it cannot match the right input always. However, we can say that **it accepts input if there exists path within a graph, that accepts the whole input**, or it **accepts input if there is a non-zero probability of ending up in accepting state**.

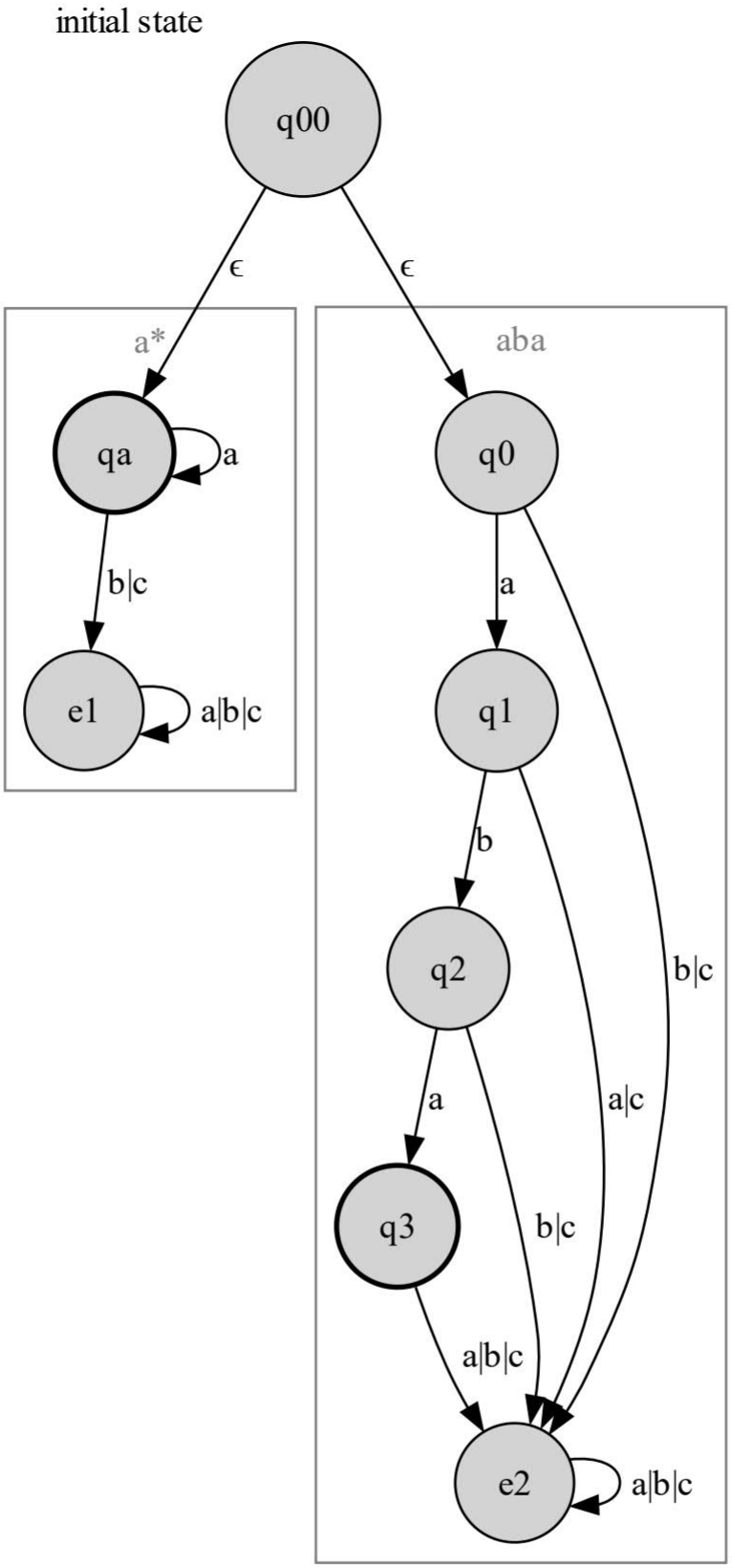
Let's say we want to parse `a* | aba`. It's an alternative of two regular expressions. The first could be expressed as:



and the second as:



Now, if we wanted to simply merge these two DFAs, we would have a problem: they both start with accepting a, so we would have to know beforehand which one to choose in order to accept a valid input. With NFA we can make some (even all!) of transitions non-deterministic, because we are checking **if a path exists**, and we don't require that we will always walk it on valid input. So let's say we have 2 valid choices from an initial state - with an empty string ϵ go into the first machine or the second machine (yes, we can use the empty string as well!):

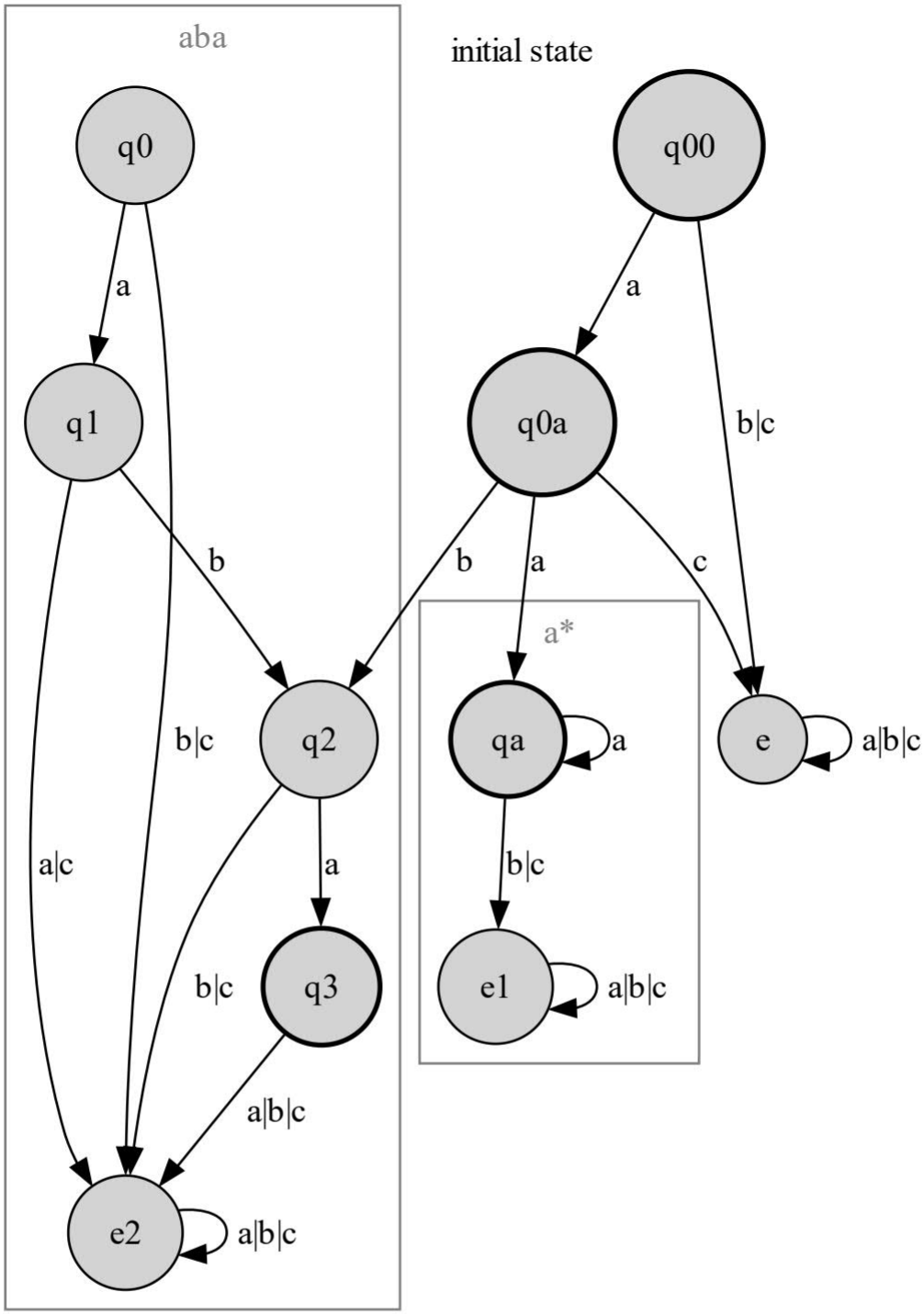


(q_0 and es were relabelled to distinct which one came from which automaton).

What would we have to do to make it deterministic? In this particular case, we can notice, that:

- correct input is either empty or starts with a,
- if it starts with a what comes next is either a sequence of more as or ba.

Let's modify our NFA for that observation:



Let us think for a moment what happened here. We now have a **deterministic** version of `a* | aba` expression! In order to remove non-determinism, we had to look ahead to determine which branch we should go with. That was achieved by introducing and using extra states - which could be mapped into a corresponding state in either of branches - until we received the first piece of information, that would make it clear, which branch we need to go from now on because the other is no longer an option:

- if we just started we could assume that if nothing arrives we are OK,
- however if we got `a`, there is uncertainty - should we expect following `a*` or `ba`, so we carry on the information that `a` arrived and we will delay our decision,
- if nothing else arrives we are at valid input so we accept the state,
- if `a` or `b` arrives we finally resolved ambiguity - from now on, we can simply go into a branch directly copy-pasted from the original DFA that created.

Of course, this could be optimized a bit - states e, e_1 and e_2 could be merged into one, while states q_0 and q_1 are inaccessible so we can remove them. This way we would arrive at the final DFA.

The process, that we showed here is called **determination of NFA**. In practice, this tracing of things until we have enough data to finally decide, requires us to create a node for each combination of "it can go here" and "it can go there", so we effectively end up building a powerset. This means that in the worst case we would have to turn our n-state NFA into 2^n DFA before we even run optimizer! Once we've done that, testing if a string is accepted by the language is relatively cheap and requires going through the whole input once. This is an important observation, because using regular expressions is so simple nowadays, that we can forget that building a matcher is itself a costly operation (**very costly**).

That explains, why in older generations of compilers the proffered flow was to generate source code with already build DFA which could be compiled into a native code, that didn't require any building in the runtime - you paid the cost of building DFA once before you even started the compilation of a program.

However, it is not the most comfortable flow, especially, since now we have a bit faster computers and a bit higher requirements about the speed of delivery and software maintenance. For that reason, we have 2 alternatives: one based on a lazy evaluation - you build the required pieces of DFA lazily as you go through the parsed input, or with the usage of backtracking. The former is done by simulating NFA internally and building DFA states on demand. The later is probably the easiest way to implement regular expression, though the resulting implementation is no longer $\Theta(n)$ but $O(2^n)$. (Confusingly, the algorithm is called NFA, though the implementation is not a finite-state automaton at all).

Regular expressions in practice

The format(s) used to describe regular expressions are directly taken from, how regular languages are defined: each symbol normally represents itself (so regexp `a` would match `a`), $*/+$ after an expression represent Kleenie star/plus (0 or more, 1 or more repetitions of an input - `a*` would match empty string, `a`, `aa`, ...), concatenation of expressions represents concatenated language (`aa` would match `aa`, `a+b` would match `ab`, `aab`, ...). Or `|` or a sum of regular languages is represented by `|` (`a|b` matches `a`, `b`). Parenthesis can be used to clarify in which order regular expression are concatenated (`ab+` means `(ab)+`, so if we wanted `a(b+)` the parenthesis helps us achieve what we want). There are also some utilities like `[abc]` which translates to `(a|b|c)` and allows us to use ranges instead of listing all characters manually (e.g. `[a-z]` represents

`..|z)`), `?` which means zero or one occurrence (`a?` is the same as `(|a)`) or predefined sets of symbols like `\s` (whitespace character), `\S` (non-whitespace character) and so on. For details, you can always consult the manual for the particular implementation that you are using.

If you are interested about the process of implementing regular expressions and building finite state machines out of the regexp format I recommend getting a book like *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman. There are too many details about implementations which aren't interesting to the majority of the readers to justify rewriting and shortening them just so they would fit into this short article.

Since we got familiar with RE, we can try out a bit more powerful category of languages.

Context-Free Grammars and Push-Down Automata

Any finite state machine can store a constant amount of information - namely the current state, which is a single element of a set of values defined upfront. It doesn't let us dynamically store some additional data for the future and then retrieve data stored somewhere in the past.

An example of a problem, that could be solved, if we had this ability is checking if a word is a *palindrome*, that is you read it the same way left-to-right and right-to-left. *Anna*, *exe*, *yay* would be palindromes (assuming case doesn't matter). *Anne*, *axe*, *ay-ay* would not be. If we wanted to check for some specific palindrome, we could use a finite state machine. But if we wanted to check for any? *A*. *Aba*. *Ab(5-million b's)c(5-million b's)ba*. No matter what kind of FSA we came up with is easy to find a word that it would not match, but which is a valid palindrome.

But let's say, we are a bit more flexible than finite state automaton. What kind of information would be helpful in deciding if we are on the right track? We could, for instance, write each letter on a piece of paper, e.g. sticky notes. We met *a*, we write down *a* and stick it to someplace. Then, we see *b*, we write it down and stick it on top of a previous sticky note. Now, let's go non-deterministic. It some point if we see the same letter arriving as we see on the top of the sticky notes stack, we don't add a new one, but take the top one instead - we are *guessing*, that we are in the middle of a palindrome. Then each time top note patches with an incoming letter you take it off. If you had an even-length palindrome you should end up with an empty stack. Well, we

have to think a bit more to handle the odd-length case as well, but hey! W… on the right track as the length of the word is no longer an issue!

…helped us get there? We had a state-machine of sorts with 2 states: *insert-card-mode* (push) and *take-matching-card-mode* (pop) (for odd-length palindrome we could use a third state for skipping over one letter - the middle one - without pushing and popping anything). Then we had a **stack** that we can push things on top, take a look at the top element, and take an element from the top. Actually, this data structure (which could be also thought of as a *last-in-first-out queue*) is **really named stack**. In combination with finite state automaton, it creates **push-down automaton** (**PDA**).

As a matter of the fact, what we defined for our palindrome problem is an example of **non-deterministic push-down automaton**. We could define deterministic PDAs (DPDA) as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where:

- Q is a finite set of **states**,
- Σ is a finite set of **input symbols** or an **input alphabet**,
- Γ is a finite set of **stack symbols** or a **stack alphabet** (because we can use different sets for input and stack, e.g. the later could be a superset of the former),
- a **transition function** $\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$, which would take the current state, the top (popped) stack symbol and next input symbol (possibly empty) to return the next state and what to push to stack (which can be represented as a word made of a stack alphabet) ,
- an **initial state** $q_0 \in Q$,
- an initial stack symbol $Z \in \Gamma$,
- a set of accepting states $F \subseteq Q$.

A non-deterministic version (NDPDA) would allow ϵ as a valid symbol in Σ, and return several possible values in transition function δ instead of one.

The palindrome example showed us that there are problems that PDA can solve that FSA cannot. However, PDA can solve all problems that FSA - all you need to do is basically ignore the stack in your transition function, and you get the FSA. Therefore, **push-down automata are a strict superset of finite-state automata**.

But we were supposed to talk about formal languages. Just like finite-state machines are related to regular languages, pushdown automata are related to **context-free grammars**. Reminder: it's a formal language where all production rules are in the form of:

$$A \to \gamma$$

where $A \in N$ (non-terminals) and $\gamma \in (N \cup \Sigma)^+$ (non-terminals and terminals). Another reminder: with CFG we are parsing structures, that are basically trees (in case of

...mming languages, where this tree represent the language's syntax it's ca... **abstract syntax tree**), and terminals are these parts of the syntax which are lea...es o... tre..., while non-terminals are nodes. The names come from the fact, that wh... you expand the tree according to production rules, you have to end up with terminals in all leaves. They are *the ends* of the tree.

Thing is, when we are parsing, we are actually given a sequence of terminals, and we must combine them into non-terminals until we get to the root of the project. Kind of opposite to what we are given in language description. How could that look like? Let's do some motivating example.

Normally when we describe the order of arithmetic operations like $+$, $-$, $\times$, $\div$ we are inserting them in-between numbers. Because operations have priorities ($\times/\div$ before $+/-$) if we want to change the default order we have to use parenthesis ($2+2\times 2$ vs $(2+2)\times 2$). This is called **infix notation** as the operator is between operands. But, you could use alternative notations: one where operator is before operands (**prefix notation** aka **Polish notation**) or after operands (**postfix notation** aka **Reverse Polish notation/RPN**). Both of them doesn't require usage of parenthesis, as the order of operation is unambiguous due to their position. The later is quite useful when you are working with compilers.

$$(1+2)\times(3+4)$$

becomes

$$1\ 2\ +\ 3\ 4\ +\ \times$$

When it comes to calculating the value of such expression, we can use stack:

- we start with an empty stack,
- when we see the number, we push it to the stack,
- when we see $+$ we take the top 2 elements on the stack, we add them and we push the result to the stack,
- same with $\times$, take 2 top elements from the stack, multiply them and push the result to the stack,
- at the end the result of our calculation would be on top of a stack.

Let's check for $1\ 2\ +\ 3\ 4\ +\ \times$:

- we start with an empty stack,
- 1 arrives, we push it to the stack,
- stack is: 1,
- 2 arrives, we push it to the stack,
- stack is $1\ 2$,

- + arrives, we take 2 top elements from the stack (1 2), add them (3) and push th
  result to the stack,
- stack is: 3,
- 3 arrives, we push it to the stack,
- stack is: 3 3,
- 4 arrives, we push it to the stack,
- stack is: 3 3 4,
- + arrives, we take 2 top elements from the stack (3 4), add them (7) and push the result to the stack,
- stack is: 3 7,
- × arrives, we take 2 top elements from the stack (3 7), multiply them (21) and push the result to the stack,
- stack is: 21,
- input ends, so our result is the only number on stack (21).

If you ever wrote (or will write) a compiler, that outputs assembler or bytecode, or something similar low-level - that's basically how you write down expressions. If there is an expression in an infix form, you translate it into postfix, as it pretty much aligns with how mnemonics works in many architectures.

To be precise, quite a lot of them would require you to have the added/multiplied/etc values in registers instead of stack, however to implement a whole expression you probably use stack and copy data from stack to registers and vice-versa, but is an implementation detail irrelevant to what we want to show here.

Of course, the example above is not a valid grammar. We cannot have a potentially infinite number of non-terminals (numbers) and production rules (basically all results of addition/multiplication/etc). But we can describe the general idea of postfix arithmetics:

$$\mathrm{BinaryOperator} \to + \mid - \mid \times \mid \div$$

$$\mathrm{Expression} \to \mathrm{Number} \mid \mathrm{Expression}\ \mathrm{Expression}\ \mathrm{BinaryOperator}$$

We have terminals $\Sigma = \{\mathrm{Number}, +, -, \times, \div\}$ and non-terminals $\mathrm{N} = \{\mathrm{BinaryOperator}, \mathrm{Expression}\}$. We could create an ADT:

```
sealed trait Terminal

final case class Number(value: java.lang.Number)
    extends Terminal

sealed trait BinaryOperator
case object Plus extends BinaryOperator with Terminal
case object Minus extends BinaryOperator with Terminal
case object Times extends BinaryOperator with Terminal
```

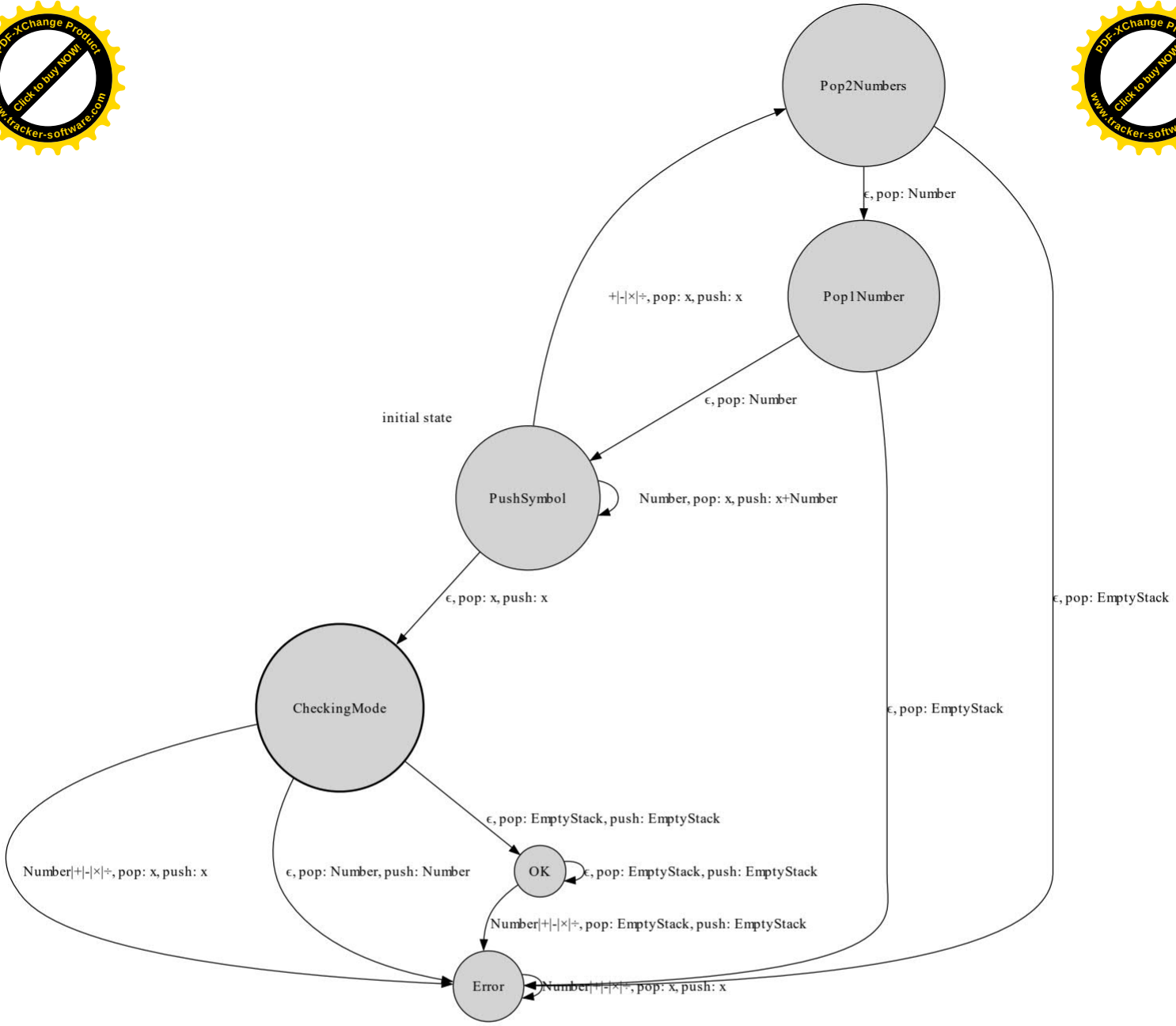
```
case object Div extends BinaryOperator with Terminal

sealed trait Expression
final case class FromNumber(number: Number) extends Expression
final case class FromBinary(operand1: Expression,
                            operand2: Expression,
                            bin: BinaryOperator)
    extends Expression
```

and now it should be possible to somehow translate `List[Terminal]` into `Expression`. (Assuming the input is a correct example of this grammar - if it isn't we should fail). In this very simple example, it could actually be done in a similar way we evaluated the expression:

- if `Terminal` is `Number`, wrap it with `FromNumber` push it to the stack,
- if `Terminal` is `BinaryOperation`, we take 2 `Expression`s from the stack, put it as `operand1` and `operand2`, and together with `BinaryOperator` put it into `FromBinary` and push to stack,
- if the input is correct, we should end up with a stack with a single element,
- if the input is incorrect, we should end up with a stack with more than one element, or during one of the operations we will miss some `Expression`s while popping on a stack.

It is *almost* enough to represent our language as PDA. To create a binary operation we look at the two elements on top of the stack, while it is legal to only know one. But we could, represent that as a state. Initial stack symbol could be a single $\mathrm{EmptyStack}$. Actually, we could also make sure that we end up with an empty stack at the end - if there are elements on stack, it's an error (becasue no operator consumed some elements). If at some point we are missing some elements it's also an error. We could end up with something like:



This PDA doesn't calculate the value of RPN. It only checks if it is valid. We are pushing `Number`s on a stack, and on a binary operation, we consume 2 `Number`s from the stack. At any point we can start "checking" - if we are at the end of input, a stack is empty (meaning that `EmptyStack` is the top element) we can assume that the input was correct so we move to `OK` through `CheckingMode`. However, if we start checking and there is some input left or there are elements on the stack - we are erring.

To make sure we understand what happened here we should remember that this is non-deterministic PDA - so for each valid input there *should exist* a valid path (and each path ending in an accepted state should describe a valid input), but we don't have to necessarily walk it each time. The other thing is that on each step of PDA we have to pop from stack - if we don't want to change stack we have to pop the same element back, if we want to add something we can pop 2 elements or more and if we want to get rid of top elements, then we simply don't pop it back.

Parsers in practice

Actually, there are 2 approaches to parsing context-free grammars:

- **top-down** approach: We start from the root of the AST tree and take a look at the possible transitions. We try to make a prediction - if we get the next alphabet element, do we know, which transition to go? If that is not enough you could try to look at transitions going from these transitions and check if any prediction is possible to do know, etc. We don't necessarily look 1 symbol ahead to determine our path - we could set some k and assume that we can look up to k symbols ahead before making a decision (which would be potentially reflected in the number of states). If our language contains recursion it might affect how we can and what is the minimal number of lookahead to decide. We are parsing input **left-to-right**, and the top-down strategy with lookahead will make us choose branch basing on **leftmost** non-terminal. That is why this approach is called LL (left-to-right, leftmost derivation). The LL parser with k tokens lookahead is called $\mathrm{LL}(k)$,
- **bottom-up** approach: We start with terminals and look at the production rules in reverse - we try to combine incoming terminals into terminals and then terminals and non-terminals until we get to the root. (This is what we have done in the PDA example above). Just like with LL we might need to make some predictions so we can look ahead of k elements. Just like with LL we read **left-to-right**. However, *contrary* to LL we can make a decision when we get *the last element* of a production rule, **rightmost** non-terminal. This is why this approach is called LR and if our parser requires k tokens lookahead it is an example of $\mathrm{LR}(k)$ parser. For $k = 1$ we can use some specific, simple implementation like *Simple LR* (*SLR*). There is also general implementation of $\mathrm{LR}(k)$ called *look-ahead* $\mathrm{LR}(k)$ ($\mathrm{LALR}(k)$) which, for $k = 1$ are called simply *LALR*.

Both approaches are usually used to build a parsing table, though they differ in how you arrive at the final table.

With $\mathrm{LL}(k)$ you can pretend that you can look ahead k chars while simply applying production rules - that k-symbol lookahead is simulated by adding additional states. When we simulate seeing kth symbol ahead, we are actually already at this symbol, but with state transitions arranged, so that we end up in a state that we should end up if we really were k symbols ago and made the decision based on a prediction. Notice, that for $k = 1$ this basically means that we are always following first matching production rule and never going back, which results in a quite simple parser.

$\mathrm{LR}(k)$, on the other hand, uses things called **shift** and **reduce**. Shift advances parsing by one symbol (*shifts* it by one symbol) (which doesn't apply any production rule), while reduce combines (*reduces*) several non-terminals and/or terminals into a single terminal (goes into the reverse direction of production rule). When an algorithm generates such a table for an input we passed it, we might see a complaint about *shift-*

…ion conflict - since well-defined LR grammar should for each PDA assign e… a *shift* …eration or a *reduce* operation, it shows that there is an ambiguity in th… …ar, that the parser generator managed to resolve (and produce a workin… code), but which will bite us by *parsing some inputs not the way we wanted*.

For defining context-free grammars parser generators quite often use syntax heavily influenced by **(extended) Backus-Naur form** ((E)BNF). In EBNF, the previous example:

$$\mathrm{BinaryOperator} \rightarrow + \mid - \mid \times \mid \div$$

$$\mathrm{Expression} \rightarrow \mathrm{Number} \mid \mathrm{Expression\ Expression\ BinaryOperator}$$

could look like this:

```
binary operator = "+" | "-" | "*" | "/" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
number = number, digit | digit ;
expression = number, | expression, expression binary operator ;
```

Notice, that here terminal symbols are defined as digits. It might be quite inconvenient, which is why a lot of parser generators would rather:

- assume that terminals are results of regular expression matching - the input would be matched against a set of regular expressions, each of which would be related to a terminal symbol. We would require them to accept whole input as a sequence of words matched by any of regular expressions. This way we would turn a sequence of input symbols into a sequence of terminal symbols. The part of a program responsible for this **tokenization** is called **lexer**. Such approach is seen e.g. with parser generators based on `lex` (lexer) and `yacc` (Yet Another Compiler-Compiler) and their GNU reimplementations `flex` (free lex) and `bison` (an allusion to gnu as a lot of GNU tooling is based on `bison`). (It should also explain why certain languages have weird rules regarding class/method/function/variable names - since tokenization takes place in the very beginning, it has to reliable classify each piece of code unambiguously as a terminal symbol),
- alternatively allow you to use regular expressions directly in a parser-defining syntax. As this approach is much more readable it was also used in *parser combinators*.

Right, we haven't mentioned parser combinators. What are they, and why they became more popular recently?

Parser combinators

When computer resources were really scarce, we didn't have the comfort of building parsers in the most convenient way - the idea behind parsing generators was

ting fast, ready to use PDA which would parse input with linear time and me (that is, directly proportional to the input). Overhead had to be limited to the inim n, so the best way was to do all the calculations (both lexing and parsing during code generation, so when we would run the program, it would be able to parse as soon as the code was loaded from the disk to the memory. All in all generating imperative code was the way to go.

But nowadays the situation is different. We have much faster computers with a lot more memory. And the requirements we have regarding programs are much higher, so the process of validating the parsed input became much more complex - so small overhead for parsing is not as painful. Additionally, we made much more progress when it comes to functional programming.

This opened the gate to an alternative approach called parser combinators (which is *not that new* considering, that it was described in *Recursive programming Techniques* by Burge from 1975 as *parsing functions*). What we do is basically, a function composition.

Let's try by example. This time we'll try to implement infix syntax. At first we'll do something about lexing terminal symbols (and using spaces for separation):

```
def number(input: String) = """\s*([0-9]+)(\s*)"""
    .r
    .findPrefixMatchOf(input)
    .map { n ⇒
       val terminal = Number(n.group(1).toInt)
       val unmatched = input.substring(n.group(0).length)
      terminal → unmatched
    }

def plus(input: String) = """\s*(\+)(\s*)"""
    .r
    .findPrefixMatchOf(input)
    .map { n ⇒
       val terminal = Plus
       val unmatched = input.substring(n.group(0).length)
      terminal → unmatched
    }

def minus(input: String) = """\s*(-)(\s*)"""
    .r
    .findPrefixMatchOf(input)
    .map { n ⇒
       val terminal = Minus
       val unmatched = input.substring(n.group(0).length)
      terminal → unmatched
    }

def times(input: String) = """\s*(\*)(\s*)"""
    .r
    .findPrefixMatchOf(input)
    .map { n ⇒
       val terminal = Times
       val unmatched = input.substring(n.group(0).length)
      terminal → unmatched
    }
```

```
d…iv(input: String) = """\s*(\/)(\s*)"""
    …
    …i…PrefixMatchOf(input)
    .ma…{ n ⇒
      …al terminal = Div
      val unmatched = input.substring(n.group(0).length)
      terminal → unmatched
    }
```


It's quite repetitive so we can introduce a helper utility:

```
type Parser[+A] = String ⇒ Option[(A, String)]

object Parser{

  def apply[A](re: String)(f: String ⇒ A): Parser[A] =
    input ⇒ s"""\\s*($re)(\\s*)""".r
      .findPrefixMatchOf(input)
      .map { n ⇒
        val terminal = f(n.group(1))
        val unmatched = input.substring(n.group(0).length)
        terminal → unmatched
      }
}
```

and simplify definitions a bit:

```
val number = Parser[Number]("""[0-9]+""")(n ⇒ Number(n.toInt))
val plus = Parser[Plus.type]("""\+""")(_ ⇒ Plus)
val minus = Parser[Minus.type]("""-""")(_ ⇒ Minus)
val times = Parser[Times.type]("""\*""")(_ ⇒ Times)
val div = Parser[Div.type]("""\/""")(_ ⇒ Div)
```

then we could start combining them:

```
val binaryOperator: Parser[BinaryOperator] = in ⇒ {
  if (in.isEmpty) None
  else plus(in) orElse minus(in) orElse times(in) orElse div(in)
}
```

The curious reader might notice that this is a good candidate for a ReaderT/Kleisli composition, but we'll try to keep this example as simple as possible. That is why we'll create some specific utility for this case:

```
implicit class ParserOps[A](parser: Parser[A]) {

  // making another by-name param helps to prevent
  // stack overflow in some recursive definitions

  def |[B >: A](another: ⇒ Parser[B]): Parser[B] =
    input ⇒ if (input.isEmpty) None
            else parser(input) orElse another(input)
}
```

and rewrite `binaryOperator` as:

```
val binaryOperator: Parser[BinaryOperator] =
  plus | minus | times | div
```

Now we are missing the concatenation - or moving input forward as we matched something already:

```
def expression: Parser[Expression] = {
  val fromNumber: Parser[FromNumber] = in => {
    number(in).map { case (n, in2) => FromNumber(n) -> in2 }
  }

  def fromBinary: Parser[FromBinary] = in => for {
    (ex1, in2) <- (fromNumber | inParenthesis)(in)
    (bin, in3) <- binaryOperator(in2)
    (ex2, in4) <- (fromNumber | inParenthesis)(in3)
  } yield FromBinary(ex1, ex2, bin) -> in4

  fromBinary | fromNumber
}

def inParenthesis: Parser[Expression] = in => for {
  (_, in2) <- Parser[Unit]("""\(""")(_ => ())(in)
  (ex, in3) <- expression(in2)
  (_, in4) <- Parser[Unit]("""\)""")(_ => ())(in3)
} yield ex -> in4
```

If we tested that code (which now looks like a candidate for a state monad) we would find that it parses one step of the way (so it doesn't run recursion infinitely):

```
expression(""" 12  + 23 """)
res1: Option[(Expression, String)] =
  Some((FromBinary(FromNumber(Number(12)), FromNumber(Number(23)), Plus), ""))
```

We can prettify the code a bit:

```
implicit class ParserOps[A](parser: Parser[A]) {

  def |[B >: A](another: => Parser[B]): Parser[B] =
    input => if (input.isEmpty) None
             else parser(input) orElse another(input)

  def &[B](another: => Parser[B]): Parser[(A, B)] =
    input => if (input.isEmpty) None
             else for {
               (a, in2) <- parser(input)
               (b, in3) <- another(in2)
             } yield (a, b) -> in3

  def map[B](f: A => B): Parser[B] =
    input => parser(input).map { case (a, in2) => f(a) -> in2 }
}
```

```
def expression: Parser[Expression] = {
  def fromNumber =
    number.map(FromNumber(_))

  def fromBinary =
```

```
    (fromNumber | inParenthesis) &
    naryOperator &
    (fromNumber | inParenthesis)).map {
      case ((ex1, bin), ex2) => FromBinary(ex1, ex2, bin)
    }

  fromBinary | fromNumber
}

def inParenthesis: Parser[Expression] =
  (Parser[Unit]("""\("""")(_ => ()) &
   expression &
   Parser[Unit]("""\)""")(_ => ())).map {
    case ((_, ex), _) => ex
  }
```

```
expression(""" 12 + 23 """).map(_._1).foreach(println)
// FromBinary(FromNumber(Number(12)),FromNumber(Number(23)),Plus)
```

(Complete example you can see on gist).

Not bad! It already shows us the potential of **creating small parsers and composing them as higher order functions**. It should also explain to us why such a concept was named **parser combinators**.

But, can we have parser combinators out-of-the-box? We would need an implementation which:

- is statically typed,
- gives us concatenation `&`, alternative `|`, and `map`ping of parsers,
- let us gives suggestions if certain matching should be *greedy* (match whatever it can, potentially indefinitely) or *lazy* (finish ASAP),
- is probably more complex than a simple function from input into output with the unmatched part. It could e.g. make use of lookahead,
- give us a lot of utilities like e.g. regular expression support.

Luckily for us, such implementation already exists, so we can just use it. FastParse is a parser combinator library written by Li Haoyi (the same guy who created Ammonite and Mill). While it provides us a nice, functional interface, it uses Scala macros to generate fast code with little overhead (which gives us hardly any reason for considering parser generators, at least for Scala).

Our parser can be rewritten into fastparse this way:

```
// import $ivy.`com.lihaoyi::fastparse:2.1.0`
import fastparse._
import ScalaWhitespace._ // gives us Scala commens
                         // and whitespaces out-of-the-box

object Parsers {

  // terminals
  def number[_ : P] =
```

```
    ( CharIn("0-9").rep(1).!).map(n ⇒ Number(n.toInt) )
                    // ! makes parser catch input as String
  def plus[_ : P] = P("+").map(_ ⇒ Plus)
  def minus[_ : P] = P("-").map(_ ⇒ Minus)
  def times[_ : P] = P("*").map(_ ⇒ Times)
  def div[_ : P] = P("/").map(_ ⇒ Div)

  // non-terminals
  def binaryOperator[_ : P] = P(plus | minus | times | div)
  def fromNumber[_ : P]: P[FromNumber] =
    P(number.map(FromNumber(_)))
  def fromBinary[_ : P]: P[FromBinary] =
    P(((fromNumber | inParenthesis) ~
        binaryOperator ~
       (fromNumber | inParenthesis)).map {
      case (ex1, op, ex2) ⇒ FromBinary(ex1, ex2, op)
    })
  def expression[_ : P] =
    P(fromBinary | fromNumber)
  def inParenthesis[_ : P] =
    P("(" ~ expression ~ ")")

  def program[_ : P] = P( (expression | inParenthesis ) ~ End)
}

parse("12 + 23", Parsers.program(_))
```

Before we jump the hype train - parser combinators are not equal to LL parsers and/or LR parsers. As we saw, we could define a parser accepting reverse Polish notation. However, if we tried to write a parser combinator that would accept it, then we would find, that recursive definition of `expression` would translate into a recursive function call without a terminating condition (parser combinators are just higher-order functions after all). LL or LR parser would push a symbol on the stack and take an input symbol from the input sequence, so at some point, they would have to stop (at least when the input finished). A parser combinator would need some *hint* e.g. closing block (which means that usually, it is not a problem), but we can see that parser combinators are not covering all context-free grammars.

Actually, LL parsers are not equal to LR parser either. Seeing how they work, one might argue that LL parsers correspond to Polish notation (because they make a decision at leftmost symbol - a prefix) while LR corresponds to reverse Polish notation (because they take a decision at rightmost symbol - a postfix). (See a nice post about it: LL and LR parsing demysitfied). Both can be treated as special cases of PDA, while it a set of all PDAs that corresponds with a whole CFG set.

Turing machines, linear-bounded automata, unrestrained and context-sensitive grammars

For the sake of completion, we can mention remaining computational models and grammar types, though this post is supposed to talk about parsing, so I'll try to keep it short

Turning machines and unrestrained grammars

A finite state machine at any given time remembers only in which one of a finite number of states it is. We read each symbol in the input once.

A push-down automaton remembers a current state and the last thing it put on a stack - it can "remember" things from a stack in reverse order in which it stored them there for later. You cannot remember something from the middle of the stack without forgetting everything that was stacked before it. In a way, you can think that you can read each input element twice - once in incoming order, once in reverse order, and the only nuance is how you entangle these two modes.

A **Turing machine** (defined by Alan Turing, the same guy who designed cryptologic bombe against German Navy's improved Enigma, the cryptologic bombe against original Enigma was designed by Polish Cipher Bureau) improved upon, that by using infinite tape, where the automaton could read and store symbol in one cell of that tape, and then move forward or backward. This allows us to "remember" something as many times as we need it.

Because of that ability to read the thing as many times as we want, it is possible that your machine will get into an infinite loop and never end. The question whether we can guess if a specific machine will ever return for a given input is called the **halting problem** (**HP**) and is proven to be in impossible to solve for a general case. The proof assumes, that you have a program that could use the halting problem solver on itself and loop if solvers says it should return and returns if solvers says it should loop - so it shows by contradiction that such thing cannot be constructed. A halting problem is used in a lot of proofs, that certain problem is impossible to solve - a reduction from the halting problem makes you use that problem to solve HP - since it is impossible to solve HP the problem is also unsolvable.

Turing machines are equal to **unrestrained grammars**, that is formal grammars that have no restriction about how you define a production rule. They are also equivalent to lambda calculus, register machine, and several other models. Usually, if we want to have a universal programming language, we make it Turing-complete (equal in power to TM, allowing you to simulate TM on it).

Linear-Bounded Automata and Context-Sensitive Grammars

Between push-down automata and Turing machines lies **linear-bounded automata** (**LBA**). I decided to describe them after TMs because they are basically restricted form of TM - it puts some limits on both sides of the infinite tape, that your automaton cannot cross.

It was proven that LBAs are equal to context-sensitive grammars, that is grammars in the form of:

$$\alpha A \beta \rightarrow \alpha B \beta$$

meaning that you can turn A into B only if it appears in the context of α and β.

Back to parsing

Majority of programming languages are Turing-complete. However, the first part of interpretation or compilation doesn't require that we have this much power.

Some very simple interpreters can be build when you lexing (tokenization) and parsing and on reduction you immediately evaluate the computation inside parser. However, it is quite messy to maintain in the long run.

After all, parsers and context-free grammars can only take care of syntax analysis. So, you could preserve the results of syntax analysis into a data structure - abstract syntax tree - and then perform semantic analysis. Was variable with this name already defined? Is this identifier describing a class, object, constant? Actually, when you take into consideration how complex some of these things are, you might not be surprised, that certain compilers could decide to introduce several steps of a whole compilation process - just for verifying, that the AST is correct. `scalac` has over 20 phases in total:

```
$ scalac -Xshow-phases
    phase name  id  description
    ----------  --  -----------
        parser   1  parse source into ASTs, perform simple desugaring
         namer   2  resolve names, attach symbols to named trees
packageobjects   3  load package objects
         typer   4  the meat and potatoes: type the trees
        patmat   5  translate match expressions
superaccessors   6  add super accessors in traits and nested classes
    extmethods   7  add extension methods for inline classes
       pickler   8  serialize symbol tables
     refchecks   9  reference/override checking, translate nested objects
       uncurry  10  uncurry, translate function values to anonymous classes
        fields  11  synthesize accessors and fields, add bitmaps for lazy vals
     tailcalls  12  replace tail calls by jumps
    specialize  13  @specialized-driven class and method specialization
  explicitouter  14  this refs to outer pointers
       erasure  15  erase types, add interfaces for traits
   posterasure  16  clean up erased inline classes
    lambdalift  17  move nested functions to top level
  constructors  18  move field definitions into constructors
       flatten  19  eliminate inner classes
         mixin  20  mixin composition
       cleanup  21  platform-specific cleanups, generate reflective calls
```

```
lambdafy  22  remove lambdas
     jvm  23  generate JVM bytecode
minal     24  the last phase during a compilation run
```


By the way, this is a good moment to mention what compilation actually is. From the point of view of formal languages theory, a compilation is just translation work from one formal grammar into another. Scala into JVM byte code, C++ into binary code, Elm into JavaScript, TypeScript into JavaScript, ECMAScript 6 into ECMAScript 5... There is no need to introduce something like transpiler to describe compilation from one language to another. If we would use this word, then only to specify a compiler that translates into another high-level language, not because a compiler doesn't cover that case.

Interpreter would be something, that instead of translating into another formal grammar, translates directly into a computation. However, if we assume that we want to be pure, we would return something, that could be turned into a computation - e.g. free algebra. That explains, why Typed Tagless Final Interpreter has interpreter in its name, even though it doesn't necessarily run computations immediately.

Separation of phases serves two purposes. One is maintainability. The other is that we can separate front-end of a compiler (parsing and validating AST) and back-end (using AST to generate output). For instance, in case of Scala, we can have one front-end and several back-ends: JVM Scala, Scala.js and Native Scala (though, truth to be told Scala.js and Native Scala need to expand a language a bit).

If we go fully functional with all the phases (so each phase is a function working on AST element), then we have option to *compose* functions (*phase fusion*) - if our language of choice allows us to optimize combined functions, then we can obtain a compiler which is both maintainable and performant.

Of course, the parser doesn't have to be a part of a compiler. The resulting tree might be our goal after all. XML, JSON or YML parsers exist in order to take some text representation and turn it into a tree of objects that is easier to work on. Notice, that grammars of languages like XML or HTML are too complex to be handled by something like regular expression, so if you want to use it, you'd better grab a parser.

Error handling

If you want to discover and return to a user all errors, possibly with some meaningful description of what could go wrong and how they could fix it, it is problematic.

As you noticed our naive parser combinator simply returned unmatched part of the input - hardly helpful. fastparse is slightly better - it can also tell you around which

ter things broke and what terminal/non-terminal it was (especially if you u
cut, to tell the parser where it *should not perform a backtracking*).

With parsers created by generators, the situation is similar - out of the box you u
only get the information that parsing failed. So, how come all these compilers and document parsers can give you meaningful messages? More meaningful than *things broke at n-th character*?

When it comes to parser generators like Bison, you have a special terminal symbol - `error`. When you match it, you can extract the position in text and create an element of AST which could mock the element, that should be there but put the error element instead. Whether you do it by having each element of AST being a coproduct of valid and invalid version, or if you will make each step of the way something like $(\mathrm{A}, \mathrm{B}, \mathrm{C}) \to \mathrm{Either}[\mathrm{Error}, \mathrm{Element}]$ is up to you. If that sound, as if you had to predict all possible ways a user can break the code and putting `error` there - that is exactly what you have to do there.

With parser combinators like fastparse, things are similar - you can tell the parser to *consume* some input after your current match, so you could e.g. try to match all right cases and - if they fail - consume the part of the input, that would fail and turn it into invalid AST element version.

Now, you should understand why this is not something, that you get for every language and why only some of them have user-friendly error handling. It increases the effort related to parser maintenance tremendously.

Summary

In this post, we had a rather high-level overview of parsing (text) into AST. We briefly talked about the Chomsky hierarchy and relations between regular languages and different models of computation. We talked a little more about regular languages and context-free grammars, though without an in-depth description of algorithms used to create those.

How regular languages, computational models and compilers work in greater detail you can learn from books like *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman or *Structure and Interpretation of Computer Programs* by Abelson, Sussman, and Sussman.

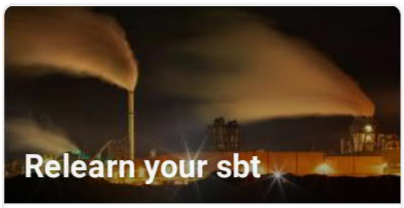
I hope, that it will help you appreciate how many thoughts and effort went into letting us build REPLs, a plethora of languages - and all of that with much better syntax, than those of the first programming languages, which very unwelcome. This unblocked us from thinking about how to *design* the language to be friendly and readable. And,

…e are still trying to figure out better ways of designing our tools, we should remember that all of that was possible thanks to pioneers in the formal languages theory

ALSO ON **KUBUSZOK.COM**

7 years ago · 1 comment

When we want to better describe our domain, at some point we might …

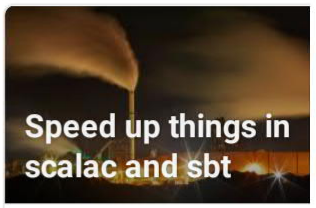


6 years ago · 15 comments

When I started to learn sbt, I noticed, that there is a huge gap between how I'm told …

7 years ago · 3 comments

A long time ago in the land of Scala emerged new type-safe way of dependency …



6 years ago · 2 comments

Scala is not the fastest language to compile. sb adds its own overhead.