Language Design and Compilers

Muchang Bahng

Summer 2025

Contents

Compiling is the process of converting code that you write (essentially a giant string) into assembly, for a specific ISA. You can use a *cross-compilation toolchain*, where you have a machine for one ISA (e.g. x86) but compile it into ARM with—say the `gccxarm` package.

Grammars (regular, context-free).

1. *Lexer.* Convert a sequence of characters into a sequence of tokens. Whitespace and comments are not tokens, since they get dropped out. Need to talk about DFA, NFA, and regular expressions.
2. *Parsing.* Building the abstract syntax tree. e.g. LL parsing (what we are doing) vs LR parsing. MLyac. Trees make everything explicit and so is easier to work with.
3. *Type Checking.* Just a bit of work.
4. *IR.* Top of the mountain.
5. *Instruction Selection.*
6. *Liveness Analysis.* Data flow analysis, which is at the core of a lot of optimization.
7. *Register Allocation.* This gives the MIPS assembly, which is text.

Expression evaluates to a value, while a statement is ... SMl-NJ, sort of functional.

In the code below, `if` is an expression (i.e. evaluates to a value), not a statement (which perform an action).

```
fun fact(n) = if (n <= 0) then 1 else n * fact(n-1)
```

1 Lexers

Definition 1.1 (Alphabet)

An **alphabet** Σ is a set.

Definition 1.2 (Kleene Star)

Given a set Σ, we define

$$\Sigma^* \coloneqq \bigcup_{n=0}^{\infty} \Sigma^n \tag{1}$$

Each element of Σ^* is called a **word** or a **string**, and a subset of Σ^* is called an **expression**. The **empty word**, denoted ϵ, is the unique string of length 0.

Now let's define some operations.

Definition 1.3 (Concatenation)

Given two words $u, v \in \Sigma^*$, the **concatenation** $uv = u \cdot v$ is the word formed by appending the sequence of symbols in v to the sequence of symbols in u.

Note that $(\Sigma^*, \cdot, \epsilon)$ is a monoid, where ϵ represents the empty word.

Definition 1.4 (Formal Language)

A **formal language** over alphabet Σ is a subset $L \subset \Sigma^*$.
1. A word $w \in \Sigma^*$ is **well-formed** if $w \in L$.
2. An expression $E \subset \Sigma^*$ is **well-formed** if $E \subset L$.

Example 1.1 (C Identifiers)

The set $\Sigma = \{\_, a, \dots, z, A, \dots, Z, 0, \dots, 9\}$ can be the alphabet of the formal language L representing all variable identifiers in the C programming language.

Definition 1.5 (Product of Formal Languages)

Given two formal languages $L_1, L_2 \subset \Sigma^*$, their product is defined

$$L_1 L_2 \coloneqq \{uv \mid u \in L_1, v \in L_2\} \tag{2}$$

Generally, a subset of Σ^* doesn't give us much structure, so we would like to define some way to pick the subset L out. We can do this with *grammars*.

Definition 1.6 (Formal Grammar)

A **formal grammar** is a 4-tuple $G = (V, \Sigma, R, S)$ consisting of the following.
1. *Non-terminal Symbols.* V is a finite set consisting of **non-terminal symbols**, also known as **variables**.
2. *Terminal Symbols.* Σ is a finite set—disjoint from V—consisting of **terminals**.

3. *Production Rule.* R is a relation, i.e. is a finite subset of

$$(V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^* \tag{3}$$

where the LHS represents the product of the languages. We usually write the relation as $\alpha \to \beta$.[a]
4. *Start Symbol.* $S \in V$ is the initial variable from which derivation begins.

[a] In math, this is usually written $\alpha R \beta$, or $\alpha \sim \beta$ if it is an equivalence relation.

We can think of the production rule as a set of transformations—formally called *derivations*—that you can apply to a word.

Definition 1.7 (Derivation)

Let $G = (V, \Sigma, R, S)$ be a formal grammar. We define a binary relation $\implies$ on the set of all strings $(V \cup \Sigma)^*$ as follows: A string u **directly derives** v, written $u \implies v$, if there exists strings $\phi, \psi \in (V \cup \Sigma)^*$ and a production rule $(\alpha \to \beta) \in R$ such that

$$u = \phi\alpha\psi, \qquad v = \phi\beta\psi \tag{4}$$

We say a word w is **derived** from v, written $v \implies w$, if w can be obtained by replacing a part of v according to a rule in R. Given grammar G, the language derived from G, denoted $L(G)$, is

$$L(G) \coloneqq \{w \in \Sigma^* \mid S \implies {}^* w\} \tag{5}$$

where $\implies {}^*$ is the transitive closure (meaning derived in 0 or more steps). In other words, this is the set of all terminal strings that can be derived from S using the rules in R.

Example 1.2

Let $V = \{E\}$ be the non-terminals, and $\{+, -, \texttt{num}\}$ be our terminals, which come from our lexer. Then, our production rules can look something like.
1. $E \to E + E$.
2. $E \to E - E$.
3. $E \to \texttt{num}$.

So basically, if we have some word, then we can replace any E in the word by any of the three choices on the right hand side. For example, we can do the following sequence of derivations.

$$\begin{aligned} E &\implies E + E && (6)\\ &\implies E - E + E && (7)\\ &\implies E - \texttt{num} + E && (8)\\ &\implies E + E - \texttt{num} + E && (9) \end{aligned}$$

You can keep doing this until there is no more production rules you can apply. For context free grammars, you basically do this until there are only terminals left, e.g. $\texttt{num} + \texttt{num} - \texttt{num} + \texttt{num}$.

Definition 1.8 (Transitive Closure)

The relation $\implies {}^*$, called the **reflexive transitive closure** of $\implies$, is defined as follows. $u \implies {}^* v$ if
1. $u = v$, or
2. There exists a finite sequence of strings $u = w_0, w_1, \ldots, w_n = v$ such that $w_i \implies w_{i+1}$ for all

$0 \leq i < n$.

Definition 1.9 (Language Generated by Grammar)

Given grammar $G = (V, \Sigma, R, S)$, the language $L(G)$ is the set of all terminal strings that can be derived from the start symbol S.

$$L(G) \coloneqq \{w \in \Sigma^* \mid S \implies {}^*w\} \tag{10}$$

Therefore, you can basically think of the "complexity" or size of a language L as being determined by the "complexity" of the generating grammar G. Usually, the alphabet is kept fixed, and the main contributor to the complexity are the production rules R. Depending on how much we restrict R, we get different levels of languages in the *Chomsky Hierarchy*.

Definition 1.10 (Unrestricted)

An **unrestricted language** L is a language that can be generated by some grammar G.

Definition 1.11 (Context Sensitive)

A grammar is **context sensitive** if

$$R \subset \{(\alpha, \beta) \mid \alpha, \beta \in (V \cup \Sigma)^+, |\alpha| \leq |\beta|\} \cup \{(S, \epsilon) \text{ if S does not appear on any RHS}\} \tag{11}$$

That is, the length of the string on the right must be greater than or equal to the length of the string on the left. You cannot "delete" symbols are you derive. A language L is **context-sensitive** if $L = L(G)$ for some context-sensitive grammar G.

For theorists, context sensitive grammars are important, but for engineers, context free grammars matter much more.

Definition 1.12 (Context Free)

A grammar is **context free** if

$$R \subset V \times (V \cup \Sigma)^* \tag{12}$$

That is, it asserts that the left-hand side can only contain a single non-terminal. A language L is **context-free** if $L = L(G)$ for some context-free grammar G.

This allows for a parse-tree structure because each variable "branches" into a new string independently of its neighbors.

Example 1.3 (Context Free Grammar)

Let $V = \{E\}$ be the non-terminals, and $\{+, -, \texttt{num}\}$ be our terminals, which come from our lexer. Then, our production rules can look something like.

1. $E \to E + E$.
2. $E \to E - E$.
3. $E \to \texttt{num}$.

So basically, if we have some word, then we can replace any E in the word by any of the three choices

on the right hand side. For example, we can do the following sequence of derivations.

$$\begin{aligned} E &\implies E + E && (13)\\ &\implies E - E + E && (14)\\ &\implies E - \texttt{num} + E && (15)\\ &\implies E + E - \texttt{num} + E && (16) \end{aligned}$$

You can keep doing this until there is no more production rules you can apply. For context free grammars, you basically do this until there are only terminals left, e.g. $\texttt{num} + \texttt{num} - \texttt{num} + \texttt{num}$.

But note that this is ambiguous since there are multiple ways to derive. For example, if we wanted to get something like `num - num - num`, we can do it in either of the following ways.

$$\begin{aligned} E &\implies E - E && (17)\\ &\implies E - (E - E) && (18)\\ &\implies \texttt{num} - (\texttt{num} - \texttt{num}) && (19)\\ E &\implies E - E && (20)\\ &\implies (E - E) - E && (21)\\ &\implies (\texttt{num} - \texttt{num}) - \texttt{num} && (22) \end{aligned}$$

This is not good, so we want to modify our grammar so that it is safe from these ambiguities.

Example 1.4

If we have a grammar with the following production rules.
1. $E \to E + \texttt{num}$
2. $E \to E - \texttt{num}$
3. $E \to \texttt{num}$

Then, we have no ambiguities since there is only one possible way

$$\begin{aligned} E &\implies E - \texttt{num} && (23)\\ &\implies (E - \texttt{num}) - \texttt{num} && (24)\\ &\implies (\texttt{num} - \texttt{num}) - \texttt{num} && (25) \end{aligned}$$

Note that figuring out whether a grammar is ambiguous or unambiguous is a bit tricky.

Definition 1.13 (Regular)

A grammar G is **regular** if it is either of the following:
1. *Right Linear.*
$$R \subset V \times (\Sigma \cup \Sigma V \cup \{e\}) \tag{26}$$
Meaning that every rule must look like $A \to a$ or $A \to aB$.
2. *Left Linear.*
$$R \subset V \times (\Sigma \cup V\Sigma \cup \{\epsilon\}) \tag{27}$$
Meaning that every rule must look like $A \to a$ or $A \to Ba$.

A language L is **regular** if $L = L(G)$ for some regular grammar G.

Theorem 1.1 (Kleene's Theorem)

A language L is regular if and only if there exists a DFA M such that $L(M) = L$. T

Why is this useful?

Definition 1.14 (Categories of Words)

In a language,

1. a **literal** is a word in L that represents a fixed value directly in the source code.
2. an **identifier** is a word in L used to name entities such as variables, functions, and types.
3. a **reserved word**, or **keyword**, is an identifier that has a fixed meaning in the language's grammar.

Definition 1.15 (Expression)

Let $G = (V, \Sigma, R, S)$ be a grammar and let $E \in V$ be the non-terminal symbol representing expressions.

Definition 1.16 (Statement)

We want to convert a series of characters (our program) into tokens. What are some types of tokens?

Note that a whitespace is not a token.

The whole process of lexing is to convert a giant string (your code) into a sequence of tokens. But this requires us to classify what the next substring is, i.e. whether it's an identifier, a keyword, a literal, etc. So basically, we need to match these incoming words to their respective token type, and we can do this by pattern matching. By doing this pattern matching, we can determine whether a given string is of a certain token type. The most intuitive way to do this is to use regular expressions, but this turns out to be hard to compute. Therefore, we must convert it to NFAs, then DFAs, and at this point, it becomes easy for the computer to compute.

1.1 Regular Expressions

Definition 1.17 (Regular Expression)

Given an alphabet Σ, a **regular expression (regex)** is defined with the following axioms.

1. *Symbol.* Any character $a \in \Sigma$ is a regex.
2. *Epsilon.* ϵ, which stands for the empty string, is a regex.
3. *Or.* Given regexes R_1, R_2, $R_1 \mid R_2$, defined to be either R_1 or R_2, is a regex.
4. *Concatenation.* Given regexes R_1, R_2, $R_1 R_2$, defined to be the concatenation of them, is a regex.
5. *Kleene Star.* Given regex R, R^* represents a concatenation of 0 or more R.[a]

[a] Note that this is essential since the number of concatenations is unbounded. It is *not* syntactic sugar.

To make things more convenient to write, we use the following common *syntactic sugar*. This doesn't add functionality.

1. *1 or More.* $R^+ = RR^*$

2. *Optional.* $R? = R \mid \epsilon$[1]
3. *Any Character.* . stands for any character.
4. *Brackets.* This just means a big "or" of all the characters. We can write $[abc] = (a|b|c)$, and can do $[0-9]$, $[a-z]$.
5. *Negation.* `[R̂]` means any character that is not R.

We can have more syntactic sugar, but these are the most common.

Definition 1.18 (Recognized Language of a RegEx)

Let $w = a_1 a_2 \dots a_n$ be a string over an alphabet Σ. The regex R **accepts** the string w if it matches the string. The set of all words that are accepted by a regex R is called the **language generated by** R.

Definition 1.19 (Computability of RegEx)

The process of matching a given regex to an arbitrary string is called **computing the regex**.

Example 1.5

Let us have a string `abcdef` and regex

```
a (b | g) e? cd (ef)^\ast
```

Does this string match the regex? Yes.

One of the nice things about regexs is that they aren't as powerful as arbitrary programming, which is nice since it limits the complexity. We can write a simple declarative specification, but we can't do things like the following. In fact, it is proven with math.

1. write balanced parentheses, i.e. write regexes that outputs all parentheses that are properly nested
2. type checking

So, with this in mind, we can write regex's for things that we might care about in programming. For keywords, it's trivial.

Example 1.6 (Counterexamples to Writing Balanced Parentheses)

Example 1.7 (C Identifiers)

The syntax for all numbers might look something like

```
0 | (-? [1-9][0-9])
```

[1] This is pretty much the only use case for ϵ, so you really never write down ϵ directly in a regex.

Example 1.8 (Numeric Identifiers)

1. We may write `[1-9][0-9]*`, but this does not account for negative numbers.
2. We may write `-?[1-9][0-9]*`, but this does not account for 0.
3. We may write `0 | (-?[1-9][0-9]*)`. But should we include `-0`? This then becomes more of a design choice.

1.2 Finite Automata

In general, computing regular expressions (i.e. matching a string with a regular expression) is hard. To see if we can make this easier, we can look at *DFAs*, which are generally seen as easy to compute.

Definition 1.20 (Deterministic Finite Automaton)

A **deterministic finite automaton (DFA)** is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ consisting of

1. a finite set of states Q
2. a finite set of input symbols called the alphabet Σ
3. a transition function $\delta : Q \times \Sigma \to Q$
4. an initial state $q_0 \in Q$
5. a set of accepting states $F \subset Q$

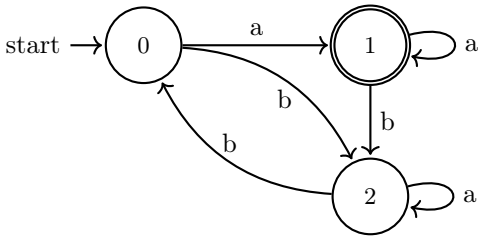


Figure 1: Note that by definition of the transition function, every single state must have exactly one outgoing transition for every symbol in Σ.

Definition 1.21 (Recognized Language of a DFA)

Let $w = a_1 a_2 \dots a_n$ be a string over alphabet Σ. The DFA M **accepts** (or computes, or matches) the string w if a sequence of states $r_0, r_1, \dots, r_n$ exists in Q with the following conditions.

1. $r_0 = q_0$.
2. $r_{i+1} = \delta(r_i, a_{i+1})$ for $i = 0, \dots, n-1$
3. $r_n \in F$.

The set of all words that are accepted by a DFA $L(M)$ is called the **language generated by** M.

Definition 1.22 (Nondeterministic Finite Automaton)

A **nondeterministic finite automaton (NFA)** is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ consisting of

1. a finite set of states Q
2. a finite set of input symbols called the alphabet Σ
3. a transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^{Q\,a}$
4. an initial state $q_0 \in Q$
5. a set of accepting states $F \subset Q$

where ϵ denotes an empty string. There are edges labeled with ϵ, which allows you to traverse to another node at no cost.

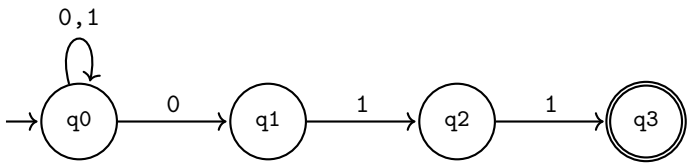


Figure 2: An NFA accepting strings that end in `011`. Nondeterminism is evident at state `q0` upon reading a `0`.

The **epsilon closure** of a set of nodes is the set plus any other nodes that you can traverse through ϵ-edges.

[a] 2^Q denotes the power set of Q.

Definition 1.23 (Recognized Language of an NFA)

Let $w = a_1 a_2 \dots a_n$ be a string over alphabet Σ. The DFA M **accepts** (or computes, or matches) the string w if a sequence of states $r_0, r_1, \dots, r_n$ exists in Q with the following conditions.

1. $r_0 = q_0$.
2. $r_{i+1} \in \delta(r_i, a_{i+1})$ for $i = 0, \dots, n-1$
3. $r_n \in F$.

The set of all words that are accepted by a DFA $L(M)$ is called the **language generated by** M.

Algorithm 1.1 (Converting DFA into NFA)

This may seem obvious, but we need to be slightly careful.

Definition 1.24 (Operations on Finite Automata)

Let M_1, M_2 be two finite automata over the same alphabet Σ. Then,

1. *Union.* The union $M_1 \cup M_2$ is defined to be the FA M satisfying $L(M) = L(M_1) \cup L(M_2)$.
2. *Complement.* The complement M_1^c is defined to be the FA M satisfying $L(M) = L(M_1)^c \subset \Sigma^*$.
3. *Intersection.* The intersection $M_1 \cap M_2$ is defined to be the DFA M satisfying $L(M) = L(M_1) \cap L(M_2)$.

Lemma 1.2 (Union of NFA)

The union of two NFAs M_1, M_2 is simple since you can make a new start node and connect it to the start nodes of the two NFAs with an ϵ-edge.

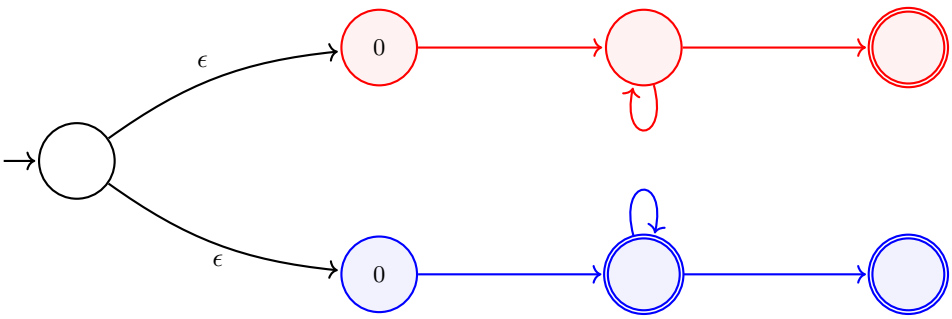


Figure 3: Combined NFA with epsilon transitions and a shared start node.

Lemma 1.3 (Union of DFA)

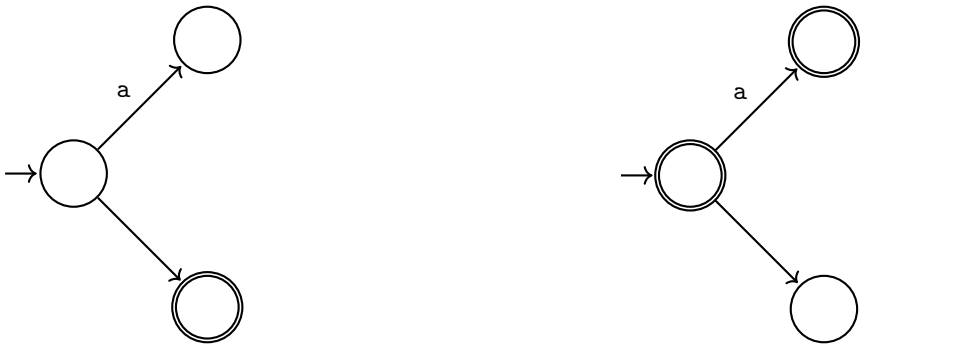
You should convert them to NFAs (which is trivial), then take the union, and then convert them back.

Lemma 1.4 (Complement of DFA)

We just flip all the accept and not accept states.

However, this doesn't work for an NFA.

Example 1.9 (Cannot Flip Accept States in NFA)



(a) Accepts `a`.

(b) Also accepts `a`, as well as the empty string.

Figure 4

Lemma 1.5 (Complement of NFA)

Therefore, you must turn an NFA into a DFA and do the complement, and then turn it back.

Theorem 1.6 (Accepting Nothing)

We can check if a DFA or NFA accepts nothing by doing BFS and seeing if we can get to an accepting state.

Theorem 1.7 (Equivalence of NFA/DFAs)

We can check if two DFAs or two NFAs are equivalent using the rule

$$A = B \iff \overline{A} \cap B\emptyset, A \cap \overline{B} = \emptyset \tag{28}$$

Therefore, we can check if two regex's are equivalent by converting them into DFAs first.

Note that checking equivalence is a very nontrivial thing for algorithms, and in fact is provably impossible to check for arbitrary algorithms (called undecidability).

In general, DFAs are known to be the easiest to compute. You can implement this basically as a giant hash table, with the accepting states stored in a list or something.

Example 1.10 (Computing a DFA)

Consider the DFA above, and the string `ababba`.

In general, DFAs are preferred by computers, while humans prefer regular expressions. We want to bridge them somehow so that we can convert regexes to DFAs, and we do this with nondeterministic finite automata. This allows us to write a nice declarative.

The problem with NFAs is that it may take an exponential time to compute due to possible branching factors at every node. It turns out that we can turn an NFA into a DFA, which may theoretically have exponentially more nodes, but in general does not.

Algorithm 1.2 (Converting NFA to DFA)

This is basically a fancy BFS algorithm. A DFA state is going to be a set of NFA states. Finally, the accepting state of the NFA is any state that contains the accepting state of the DFA.

Example 1.11 (Converting NFA to DFA)

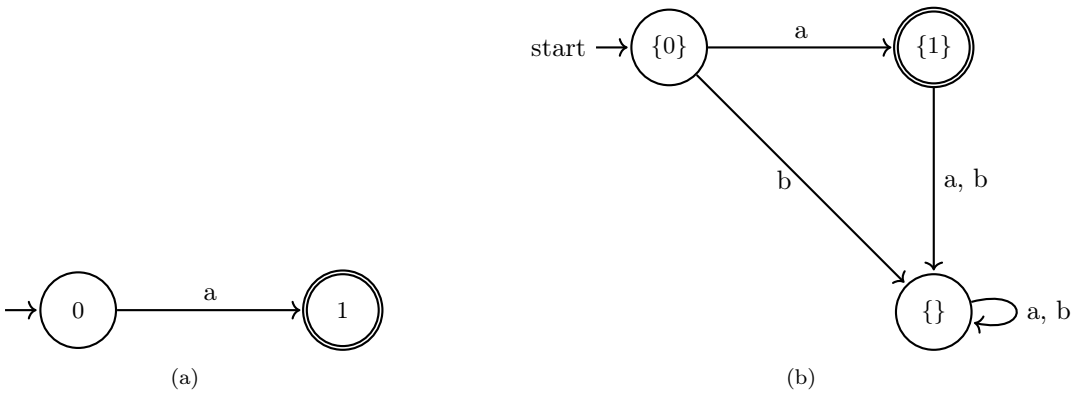


Figure 5

Example 1.12 (Converting NFA to DFA)

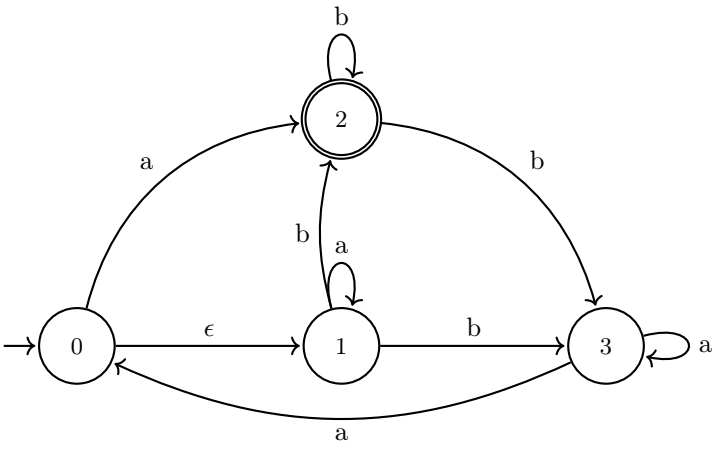


Figure 6: NFA.

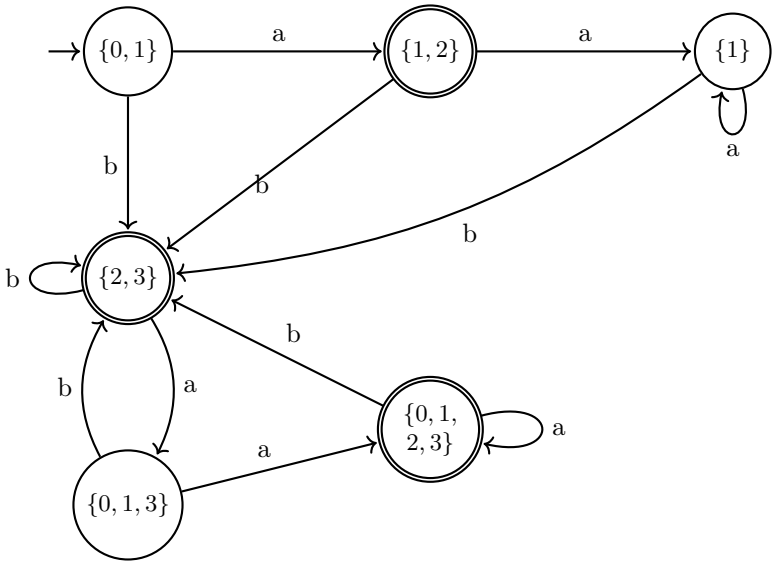


Figure 7: DFA.

Algorithm 1.3 (Converting RegEx to NFA)

This is a recursive algorithm.

1. *Symbol.* A symbol $a \in A$ is converted to a simple one edge graph. This is one of the base cases.

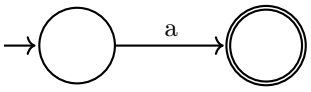


Figure 8

2. *Epsilon.* The ϵ is converted to just an accepting state. Note that it can't take in an input. This

is another base case.

Figure 9

3. *Or.* Given two NFAs representing R_1 and R_2, we take the start node and connect it to the start nodes of the two NFAs with an ϵ-edge.[a]

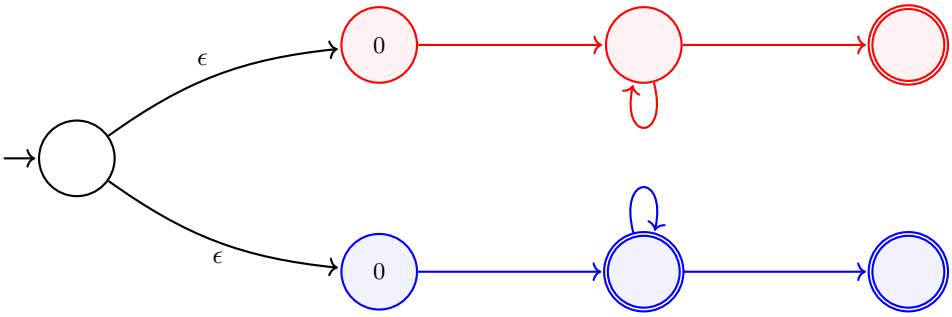


Figure 10: Combined NFA with epsilon transitions and a shared start node.

4. *Concatenation.* For R_1R_2, we combine the two NFAs by taking all accepting states in R_1 and connect them to the start of R_2 with an ϵ-edge. Then, we change the accepting states of R_1 to regular states.

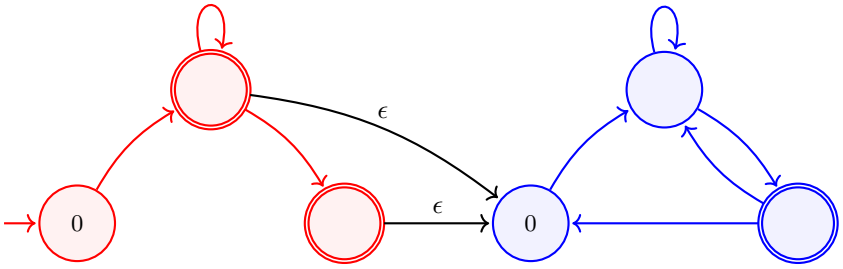


Figure 11: Sequential composition of two NFAs via epsilon transition.

5. *Kleene Start.* Given the NFA of a regex R with start node 0, to find the NFA of R^*, we do the following. First, make a new start accepting node s and draw an edge $s \xrightarrow{\epsilon} 0$. This makes 0 not a start node anymore. Finally, for each accepting node a in the NFA of R, draw edges $a \xrightarrow{\epsilon} s$.

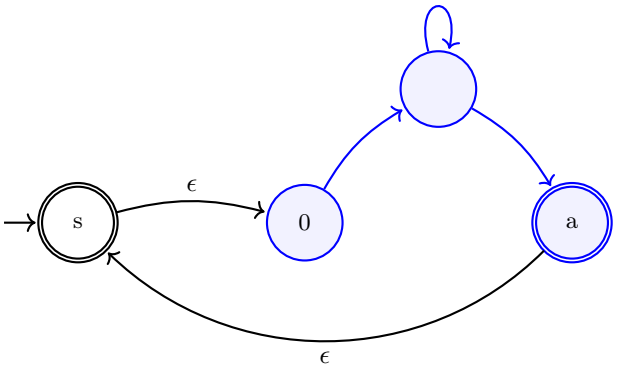


Figure 12: The new start node is necessary to avoid the possibility of going around in the NFA and ending at the old start node.

[a]Note that the presence of ϵ-edges makes things a lot easier for us.

So given a regular expression, we convert this to an NFA and then a DFA, which may be (but generally not) exponential complexity with respect to the regex alphabet. But it is independent of the length of the string that we are matching. The matching itself is linear, so we can run millions of bytes-long strings through this DFA.

So far, establishing these algorithms to compute DFAs doesn't have an obvious connection to NFAs. It turns out that lexers end up having a bunch of DFAs in them, with different accepting states for different tokens.

Example 1.13 (NFAs are More Verifiable to Humans)

Consider the language L of all comments that have delimiters of the form `/# ... #/`. How would we write this as a regular expression?

1. Our first intuition would be something like `/#(.*)#/`, but this includes strings of form `/##//##/`, which is two comments.
2. We may try to write `/#([^#]|#[^/])*#/`, but this includes strings of form `/###/...#/`, which again isn't viable.

Generally, when you have to keep thinking of edge cases and you're hacking things, you're on the losing side. However, if we think of this in terms of an NFA, this becomes much better.

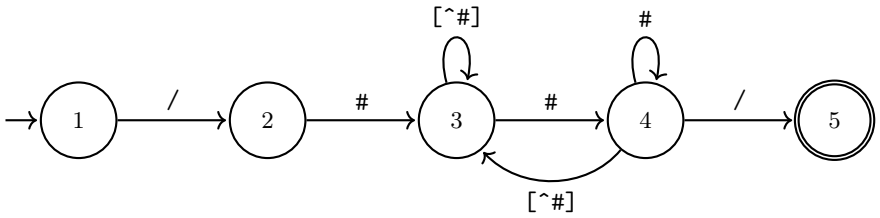


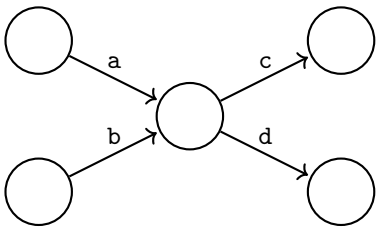
Figure 13: An NFA is much more intuitive to verify.

Algorithm 1.4 (NFA to RegEx)

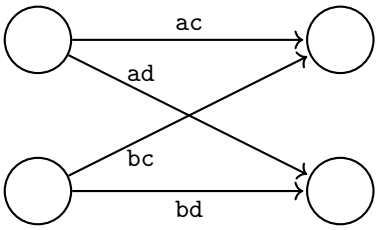
There are three rules to converting an NFA M with start state s and accepting states F to RegEx's. To do this, we actually put this into an intermediate form called a *generalized NFA*, which has regex's in edge labels, not just symbols. First, add a new start S and connect $S \xrightarrow{\epsilon} s$. Also add a new "end state" E and for every $f \in F$, connect $f \xrightarrow{\epsilon} E$. We do this is so that we avoid messy problems where

there are multiple edges or loops.

1. *Eliminate State without Loop.* To delete a node that doesn't have an outgoing edge that loops back to itself, look at all incoming edges and outgoing edges, and directly connect all edges.



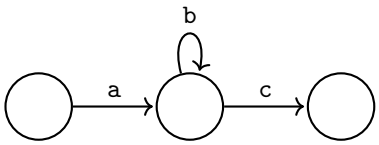
(a) NFA with intermediate state.



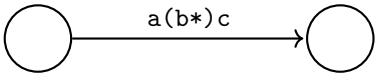
(b) Direct connections after state removal.

Figure 14: Comparison of an NFA before and after removing a central state via state elimination.

2. *Eliminate State with Loop.* To delete a node that does have an outgoing edge that loops back to itself,



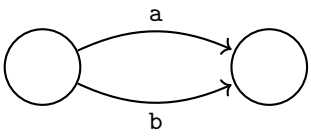
(a) NFA with a self-looping intermediate state.



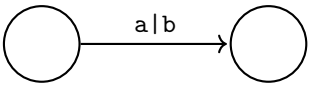
(b) Resulting NFA after eliminating the middle state.

Figure 15: Demonstration of the state elimination rule for self-loops.

3. *Collapse Multiple Edges Between 2 States.* If there are two nodes n, m with multiple edges pointing from n to m, then we can just replace it with a single edge that represents an or.



(a) Two nodes connected by parallel edges.



(b) Parallel edges combined using the union operator.

Figure 16: State elimination rule for combining parallel edges into a single regular expression.

At the end, we will have one edge from s

Example 1.14 (Deriving RegEx for Comments from NFA)

Let's do the conversion on the NFA above.

1. *Start.* We add the new start and end states.

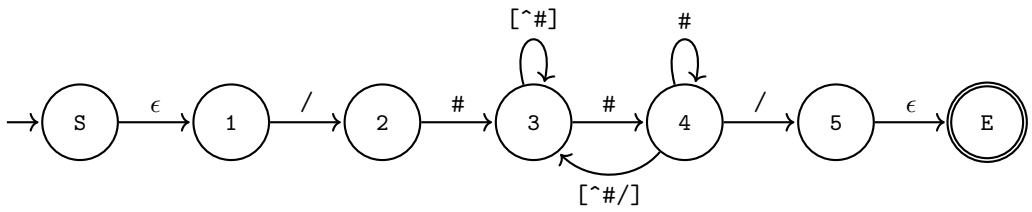


Figure 17: Condensed NFA with `node distance=2cm`.

2. *State 2.*

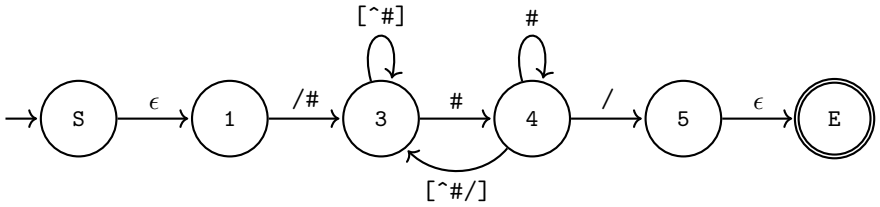


Figure 18: Reduced NFA with `node distance=2cm` and corrected `^` notation.

3. *State 1.*

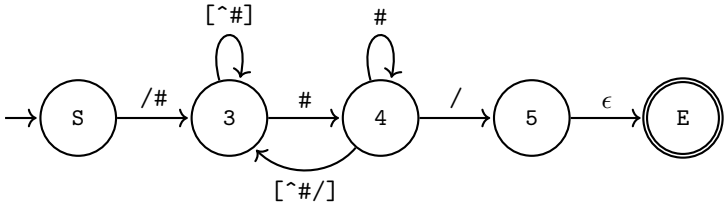


Figure 19: The NFA after eliminating state `1`. The transition label `/#` is moved to the edge starting from `S`.

4. *State 5.*

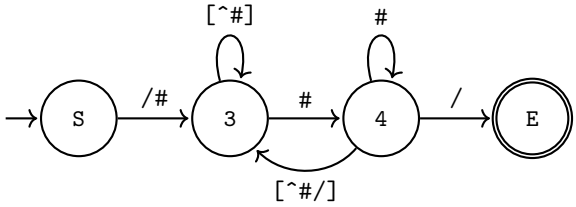


Figure 20: The NFA after eliminating state `5`. The transition to the end state `E` is now the single character `/`.

5. *State 4.* Since state 4 has a loop, we use the second rule. It has an incoming edge from 3 with a `#`, a loop edge with a `#`, and an outgoing edge to state E with a `/` and another outgoing edge to state 3 with a `[^#]`. Therefore,

$$(3 \xrightarrow{\texttt{\#}} 4 \xrightarrow{\texttt{/}} E) \implies (3 \xrightarrow{\texttt{\#(\#*)/}} E) = (3 \xrightarrow{\texttt{(\#+)/}} E) \tag{29}$$

$$(3 \xrightarrow{\texttt{\#}} 4 \xrightarrow{\texttt{[\^{}\#]}} 3) \implies (3 \xrightarrow{\texttt{\#(\#*)[\^{}\#]}} 3) = (3 \xrightarrow{\texttt{(\#+)[\^{}\#]}} 3) \tag{30}$$

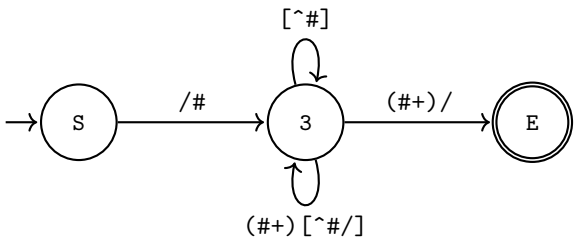


Figure 21: The NFA after eliminating state 4. The paths through state 4 are converted into a direct edge to E and an additional loop component on state 3.

6. *State 3.* Since there are two edges that have the same source and destination nodes (both state 3), we should collapse them using rule 3.

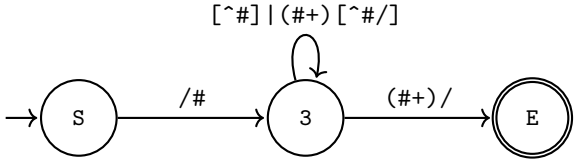


Figure 22: The NFA after eliminating state 4. The paths through state 4 are converted into a direct edge to E and an additional loop component on state 3.

7. *State 3.* Now we can get rid of state 3.

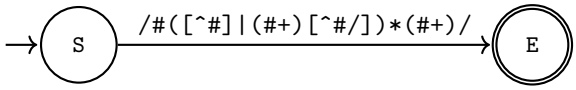


Figure 23: The final NFA after eliminating all intermediate states. The label on the edge represents the final regular expression.

Now we are done. Our regex is

```
/\#([\^{}\#]|(\#+)[\^{}\#/])*(\#+)/
```

1.3 Tokenizing

Now that we have the tools for lexing, we can finally do it. Really, lexing is just the same thing as tokenizing. In principle, a lexer has a bunch of different regex's that you are trying to match. But what happens if you have multiple matches? Consider the example below.

Example 1.15 (Ambiguities in Tokenizing)

Say that we have a language with the following tokens.
1. `<x>: x`.
2. `<nxy1>: (x*)y`
3. `<z>: z`

This leads to the following DFA.

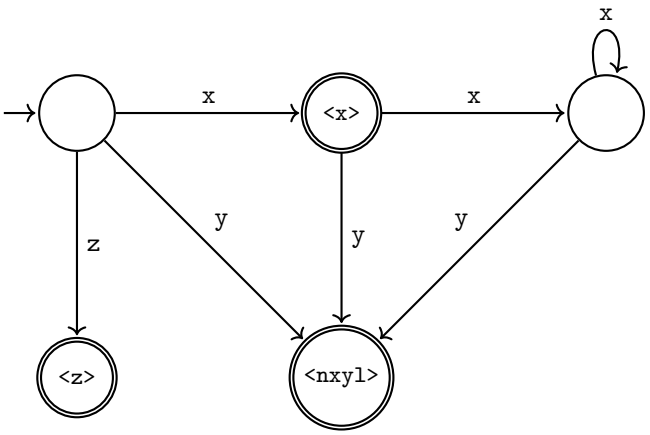


Figure 24

Then, to parse "`xxxzxxxy`," we have 3 choices, which leads to ambiguity.

1. `<x> <x> <x> <z> <nxy1>`
2. `<x> <x> <x> <z> <x> <nxy1>`
3. `<x> <x> <x> <z> <x> <x> <nxy1>`

Therefore, we run it through the principle:

1. Take the longest match.
2. Rules have an ordering.

Therefore, to detect tokens, we run it through a DFA and see the accepting state it lands on. If it is in a dead state, then output the most recent accepting state.

Lexers can have multiple DFAs, and depending on context (through start state), you traverse different DFAs (e.g. code vs comments). To see keyword vs identifier, you usually use a hashmap to check for keywords (e.g. `if` $\to$ `<IF>`), and if there are no matches, then you catch it with a regex.

2 Parsing

The whole purpose of parsing is to derive an order of operations of our tokens. There are two types or parsing. We will start out with *LL parsing*, which is easier to construct by hand.

Definition 2.1 (LL Parsing)

LL parsing refers to parsing while reading the input from left-to-right, with leftmost derivation, aka "top-down" or "recursive descent." As the name suggests, we write a bunch of mutually recursive functions—one function per non-terminal symbol. Any time we need to parse the nonterminal symbol, we call that function.

Let us take the grammar $G = (V, \Sigma, R, S)$ with nonterminals $V = \{S, C, A, B, Q\}$, terminals $\Sigma = \{x, a, d, b, q\}$, start symbol S, and production rules. Note that for any symbol X, $X \to$ is shorthand for $X \to \epsilon$, i.e. the rule that can "delete" X without any cost. Furthermore, \$ indicates an *end of input*.

1. $\texttt{S} \to AC\$$
2. $C \to c$
3. $C \to$
4. $A \to aBCd$
5. $A \to BQ$
6. $B \to bB$
7. $B \to$
8. $Q \to q$
9. $q \to$

Ideally, we would like an algorithm to parse these things similar to the functions below (`eat(c)` is a function that consumes the next token and requires it to match `c`).

```
parseS():
  parseA()
  parseC()
  eat("$")
parseC():
  if (peek() == "x"):
    eat("x")
```

```
parseA():
  if (peek() == "a"):
    parseB()
    parseC()
    eat("d")
  else:
    parseB()
    parseQ()
```

The potential problem is that there may be multiple choices for rules. To fix this, we talk about 4 things: *SDE*, *RDE*, *First Set*, *Follow Set*

Definition 2.2 (Symbol Derives Empty)

A **SDE** is a function that takes in a non-terminal and returns a boolean, indicating true if the non-terminal symbol could derive the empty string.

Example 2.1

In the example above, B, Q can both derive empty since $B \implies \epsilon, Q \implies \epsilon$. Consequently, since $A \implies BQ$, A can also derive empty. S cannot since it has a "$."

Definition 2.3 (Rule Derives Empty)

An **RDE** is a function that takes in a rule $(\alpha \to \beta)$ and returns a boolean, indicating true if the rule can be used to derive the empty string.

Theorem 2.1 (Recursive Definition of SDE, RDE)

Let $\wedge, \vee$ is the logical AND and OR operations, respectively. Then,

$$\text{SDE}(S) = \bigvee_{S \to *} \text{RDE}(S \to *) \tag{31}$$

For nonterminal n and terminal t,

$$\text{RDE}(N \to *t*) = 0 \tag{32}$$

since t must be in all future derivations. Given nonterminals $R_0, \dots, R_n$, we have

$$\text{RDE}(n \to R_0 \dots R_n) = \bigwedge_{i=0}^{n} \text{SDE}(R_i) \tag{33}$$

This is a good definition in itself mathematically, but this has no guarantee of termination. We can implement this fix by using a graph algorithm, keeping track of the visited states, and saying that if a symbol ever derives back to itself, then we return False. A different way is to iterate to a *fixed point*, which is a great algorithmic approach in general.

Algorithm 2.1 (Iterative Computation of SDE, RDE)

The general idea is that you have a system of equations and want a solution to it. You start with a solution and keep refreshing it until it doesn't change. Start by assuming no symbols derive empty and no rules deriving empty. You iteratively correct this assumption by looking at the rules, starting with the simplest (e.g. $C \to \epsilon$ and $Q \to \epsilon$).

Basically, this is a 0th order optimization problem in $\{0, 1\}^N$. This will terminate since once you switch over to True, you can't go back to false.

Definition 2.4 (First Set)

Given nonterminal n and terminal t, the `FirstSet` function takes in any string of nonterminals/terminals and returns a set of terminals. More specifically, a terminal t' is in

$$\texttt{FirstSet}((n|t)^*) \tag{34}$$

if and only if there exists some derivation that has its first character as t'.

Example 2.2

In the example above, we can compute

$$\text{FirstSet}(aBqD) = \{a\} \tag{35}$$
$$\text{FirstSet}(B) = \{b, q\} \tag{36}$$
$$\text{FirstSet}(BqD) = \{b, q\} \tag{37}$$

Theorem 2.2 (Recursive Definition of FirstSet)

We can define

$$\text{FirstSet}() = \{\} \tag{38}$$
$$\text{FirstSet}(t(n|t)^*) = \{t\} \tag{39}$$
$$\text{FirstSet}(n(n|t)^*) = \begin{cases} \bigcup_{n\to *} \text{FirstSet}(*) & \text{if SDE}((n|t)^*) = 0 \\ \text{FirstSet}((n|t)^*) \cup \bigcup_{n\to *} \text{FirstSet}(*) & \text{if SDE}((n|t)^*) = 1 \end{cases} \tag{40}$$

Note that the $\bigcup_{n\to *} \text{FirstSet}(*)$ may infinitely recurse. So to actually compute this, we again do an iterative approach, designed so that it is guaranteed to converge.

Algorithm 2.2 (Iterative Computation of FirstSet)

Example 2.3

So after computing this, we can see that a string in this language must start with an a, b, q, c, or \$, but not with a d!

Definition 2.5 (Follow Set)

Let $G = (V, \Sigma, R, S)$ be a grammar. Then for any nonterminal $n \in V$, $\text{FollowSet}(n) \subset \Sigma$ is the set of terminals satisfying the following property: $t \in \text{FollowSet}(n)$ if there exists a some word w derived from S that contains a n immediately followed by a t. That is,

$$S \implies w, \quad w = \dots nt \dots \tag{41}$$

2.1 Context-Free Grammars

2.2 Parsing Strategies

2.3 Abstract Syntax Trees

3 Semantic Analysis

3.1 Symbol Tables

3.2 Type Checking

3.3 Scope Analysis

4 Intermediate Representation

4.1 IR Design

4.2 Lowering

5 Optimization

5.1 Local Optimization

5.2 Data Flow Analysis

5.3 Global Optimization

6 Backend