CS164 Programming Languages and Compilers **Spring 2024**

Discussion Worksheet 2

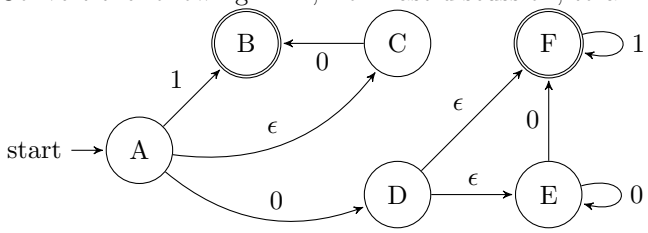
1 NFA → DFA

Recall that for any NFA we can construct an equivalent DFA (i.e. one that accepts the same language). In class, we saw the subset construction algorithm for converting NFAs to DFAs:

- Each state of the resulting DFA is a non-empty subset of states S from NFA's states Q, i.e. $S \subseteq Q$.
- The start state is the set of NFA states reachable through ε-moves from its start state.
- Add a transition $S \to^a S'$ if S' is the set of states reachable from any state in S after seeing the input a.

Exercise 1 Convert the following NFAs to DFAs using the subset construction algorithm.

1.1 Convert the following NFA, from last discussion, to a DFA.



2 Efficient Lexing

For every token type in the language, we can now generate a DFA to represent its regex. Let's see how we can use these FSMs to tokenize a program efficiently.

First, we construct the NFA for the regex $x \mid y \mid z \mid \ldots$ where $x, y, z, \ldots$ are the regular expressions for the various token-types in our language. There is a small change: instead of collapsing all the final states from the different branches into a single NFA final state, we leave each "or" branch with its own NFA final state, labeled with that branch's token type. For example, consider a silly language where the regular expressions for each token are:

- `aa|bb` (for "keywords", **kw**)
- `(a|b)(a|b)*` (for identifiers, **id**)

- (for the single whitespace, **ws**)
- `+|-` (for operators, **op**)

We construct the NFA for `(aa|bb)|((a|b)(a|b)*)| |(+|-)`, and for each **kw**, **id**, **ws** and **op**, there will be a distinct NFA final state with that label.

When converting this NFA to a DFA, a final state in the DFA is, by definition, a *subset* S of NFA states, which happens to contain an NFA final state. We then label the DFA final state S with the set of token-types associated with NFA final states in S. For example, since the same substring can be matched by both **kw** and **id**, the DFA will have a final state labelled (**kw**, **id**), along with one labeled **id**, one labeled **ws**, and one labeled **op**

Now to tokenize a stream of characters with our DFA-with-labels, we keep a record of the last final state we've reached, and its associated collection of token-types. We transition from DFA state to DFA state, following the input characters, until we reach a dead state. At this point, we rewind to the last final state we've reached, and yield the characters up to that point as a lexeme. For example, when we have seen the prefix `ab`, we will be in the final state (**id**). When we've seen (`ab+`), we reach a dead state (`ab+` cannot be matched by `(aa|bb)|((a|b)(a|b)*)| |(+|-)`), so we rewind to the last final state we reached (in this case, labeled **id**), and yield the characters (e.g. `ab`) as a lexeme, along with the token type (so, (`id`, `ab`).

Exercise 2 Token-matching subleties; we'll talk through these together.

2.1 Suppose the last final state we reached had two labels (e.g (**kw**, **id**)). What should be the reported token-type of the emitted lexeme, if the last final state reached has multiple token-types associated with it?

2.2 What should we do if, while lexing, we didn't reach any final state before reaching a dead state?

2.3 While this method is inherently based on maximum munch, it can trivially be with token priorities. Let's say each token-type has a priority number. How can we modify the algorithm so that a higher priority token will always be returned, even if it's shorter than another valid token?

3 CFG Parsing

A Context-Free Grammar supports all replacement rules of the form: $A \to \gamma_1 ... \gamma_n$, where γ_i is either a terminal symbol or any nonterminal symbol (including A) and $n \geq 0$.

Exercise 3 CFG Exercises.

3.1 We have the following CFG, where the terminal symbols are $\{+, *, id\}$ and there is only one nonterminal symbol E:

```
E -> E + E | E * E | id
```

Write a derivation and a parse tree for the following string, starting from E and deriving the given string. Only replace the leftmost nonterminal symbol every time. Evaluating the expression according to the parse tree should yield the correct mathematical result.

```
id + id * id
```

3.2 It turns out this string is ambiguously parsed by this grammar. Write another derivation and its corresponding parse tree. Evaluating this parse tree should *not* yield the expected mathematical result. This time, start with the string, and apply the derivation rules in reverse (for example, go from $E + E$ to E) to end up with just E.

3.3 We want to avoid ambiguity. Rewrite the grammar so that it resolves this ambiguity, and so that it always yields the correct mathematical result.

3.4 Check your new grammar. Does it parse $id + id + id$ unambiguously? If not, update it.

Exercise 4 Intuition-building for future material. Imagine you are trying to evaluate a very long mathematical expression of addition and multiplication. Your calculator can only perform actions on two numbers at a time. Thus, to evaluate $2 * 3 + 4$, you would first enter '2*3', which would simplify your expression to $6 + 4$, and then you would enter '6+4'. Recall order of operations: $*$ must be performed before $+$.

4.1 At the beginning of the expression, you see $5 * 6$.... Can you simplify the expression; if so, how?

4.2 At the beginning of the expression, you see $5 + 6$.... Can you simplify the expression; if so, how?

4.3 At the beginning of the expression, you see $5 + 6 + 7$.... Can you simplify the expression; if so, how?

4.4 At the beginning of the expression, you see $5 + 6 * 7$.... Can you simplify the expression; if so, how?

4.5 Generally, what do you have to check in order to simplify an addition? A multiplication?