Non-Deterministic Finite Automata and Grammars

COMP2600 — Formal Methods for Software Engineering

Ranald Clouston
Australian National University
Semester 2, 2013

Is it legal?

Yes, but it’s a non-deterministic FSA — an NFA.

How is it different to a DFA?

- Multiple edges with the same label come out of states
- For some states, there is not an edge for every token

Is it clear what it does?

For some states, some inputs are not allowed, so the NFA can get “stuck.”

For some states, there are more than one choice of next state.

If the right choices are made, strings of the form $a^i b^j c^k$ ($i, j, k \geq 1$) will take the machine to its final state.

Example: to accept the string $aaabcc$ you need to look ahead to determine the right choice, or backtrack after the wrong choice.
**DFAs vs NFAs**

For each state in a DFA and for each input symbol, there is a unique successor state.

That is, DFAs have a *transition function*.

NFAs allow zero, one or more transitions from a state for the same input symbol.

That is, NFAs have a *transition relation*.

An input sequence \(a_1, a_2, \ldots, a_n\) is accepted by an NFA if there exists some sequence of transitions that leads from the initial state to a final state.

**Example showing relative simplicity**

An NFA recognising strings of letters ending in “ion”:

\(\Sigma\) is the Latin alphabet

An Equivalent DFA

The following *minimal* DFA accepts the same language:

**Definition of NFA**

NFAs differ from DFAs only in that they have a transition relation rather than a transition function.

That is, rather than:

\[ N: S \times \Sigma \to S \]

An NFA has:

\[ R \subseteq S \times \Sigma \times S \]
**Eventual State Relation**

In NFAs we have an **eventual state relation**

\[ R^* \subseteq S \times \Sigma^* \times S \]

\( R^* (s, w, s') \) is true if \( s' \) is a state the NFA can reach, starting in state \( s \) and reading string \( w \).

We define \( R^* \) inductively:

\[ R^* (s, \varepsilon, s) \]

\[ R^* (s, x \alpha, s') = \exists s''. R(s, x, s'') \land R^* (s'', \alpha, s') \]

---

**Language of an NFA**

Let \( A = (\Sigma, S, s_0, F, R) \) be an NFA.

We say \( w \) is **accepted** by \( A \) if

\[ \exists s \in F. R^* (s_0, w, s) \]

The **language** accepted by \( A \) is the set of all strings accepted by \( A \)

\[ L(A) = \{ w \in \Sigma^* \mid \exists s \in F. R^* (s_0, w, s) \} \]

That is, \( w \in L(A) \) iff there exists a path through the diagram for \( A \), from \( s_0 \) to a final state \( s (s \in F) \), such that the symbols on the path match the symbols in \( w \).

---

**Power of Non-determinism?**

It can be easier to come up with an NFA for a given language than a DFA.

*However, NFAs are no more powerful than DFAs.*

**Theorem:**

If language \( L \) is accepted by an NFA, then there is some DFA which accepts the same language.

In fact, there is an **algorithm** which transforms any NFA to a corresponding DFA, called the **subset construction**.

However, the resulting DFA may have a relatively huge number of states and transitions.

---

**Constructing the Equivalent DFA from an NFA**

The idea is simple, the details are a bit involved.

Let \( \{ S_0, S_1, \ldots, S_n \} \) be the set of states of the NFA.

We construct a DFA in which each state is labelled by a set chosen from the symbols \( S_0, S_1, \ldots, S_n \).

If the NFA start state is \( S_0 \), the DFA’s start state will be labelled \( \{ S_0 \} \).

Given an input token \( x \): suppose that the NFA may move from the states it is in to any one of \( \{ S_1, S_3, S_4 \} \).

Then the corresponding DFA, after it has processed \( x \), will be in the state labelled \( \{ S_1, S_3, S_4 \} \). We will usually abbreviate this state’s name to \( S_{134} \).

The state of the DFA labelled \( S_{134} \) will be a final state if any of the states \( S_1, S_3, S_4 \) are final states of the NFA.
Constructing the Equivalent DFA from an NFA - Example

- Start at $S_0$.
  - If we read the token 0, then we can go to $S_1$ or $S_2$, so create a new state $S_{12}$ which is final.
  - Given 1 we can’t go anywhere, so invent a new state $S$ (for $\emptyset$).
- Consider $S_{12}$.
  - Given 0 we can’t get anywhere from $S_1$, but can go from $S_2$ to itself, so create a new state $S_2$ which is final.
  - Given 1 we can similarly only get to $S_2$.

Never forget the transitions out of the ‘empty state’ $S$ – a DFA must be defined as a total function!

Constructing the Equivalent DFA from an NFA - Result

An Aside on Regular Expressions (not assessed)

Regular expressions are used to specify languages by giving a pattern that the strings must match.

Widely used in computing — editors, commands (e.g. grep), built in to some programming languages (Perl, Ruby . . . ).

Regular expressions are built from only a few basic operators: vertical bar to indicate choice; star to indicate repetition; and string concatenation.

For example:

- $a^*$ indicates 0 or more $a$s.
- yes | no is the language with just the 2 given strings.
- $((0 | 1)^*$ indicates the set of binary numerals, including the empty string.
Regular Expressions and Finite State Automata

Regular expressions can express a variety of quite complicated properties:

- \(0|1\) is the set of (non-empty) binary numerals with no unnecessary leading zeros.
- \((1|0)^{\ast}\) is the set of binary numerals, i.e. \(\{0, 1\}\)

In fact a language can be specified by regular expression if and only if it is recognised by some Finite State Automata!

The proof is rather involved and is outside the scope of this course...

In our next lectures we will learn more about the relationship between language specifications and abstract machines.

Applying all this theory

There are standard tools (in every decent programming language) which generate "lexical analysers" or "scanners" that check whether input strings match given regular expressions. (Lex, Flex, Alex . . .)

How do they work?

1. Derive an NFA from the regular expression;
2. Convert the NFA to a DFA;
3. The resulting DFA may not be minimal, so apply the minimisation algorithm to erase equivalent states;
4. A standard driver program takes the DFA as a data structure.

Of course, real tools like Flex employ further optimisations.

A Problem of Language

The first high-level programming language was Fortran, developed in the 1950s by John Backus for IBM.

The syntax of the earliest versions of Fortran was marred by inconsistent and arbitrary design, hampering both programming and compilation.

For example, in Fortran spaces are not significant (even as separators):

\[\texttt{DO 5 I = 1.25} \quad \text{assignment to variable \texttt{D05I}}\]
\[\texttt{DO 5 I = 1,25} \quad \text{count loop}\]

Not until the scanner reaches "." or "," can these statements be distinguished.

Edsger Dijkstra referred to Fortran as an "infantile disorder" of the field of computing.

Backus-Naur to the rescue

The solution was to establish a unified and predictable format for all programming language syntax.

The foundations for this effort were provided by formal language theory, developed by a range of people including the linguist Noam Chomsky.

In particular, Backus-Naur Form (BNF), developed by John Backus and Peter Naur (for ALGOL) became, and mostly remains, the standard for programming language design.
Formal Languages – A Reminder of Terminology

- The **alphabet** or **vocabulary** of a formal language is a set of **tokens** (or **letters**). It is usually denoted $\Sigma$.
- A **string** over $\Sigma$ is a **sequence** of tokens, or the null-string $\varepsilon$.
- For example, if $\Sigma = \{a, b, c\}$ then $ababc$ is a string over $\Sigma$.
- A **language** with alphabet $\Sigma$ is some set of strings over $\Sigma$.

**Notation:**
- $\Sigma^*$ is the set of all strings over $\Sigma$.
- Therefore, every language with alphabet $\Sigma$ is some **subset** of $\Sigma^*$.

Specifying Languages

Languages may be defined by various means.

- As an explicit finite set (members are listed)
- As a set, by giving a predicate (e.g. palindromes over some alphabet)
- Algebraically (e.g. regular expression)
- **Recognisers (abstract machines)**
- **Grammars (e.g. regular or context-free)**

Introduction to Grammars

A **grammar** can be thought of as a recipe for **generating** sentences of a language.

In some sense they are dual to abstract machines (automata), which can be viewed as tools for **recognising** languages.

The relationship between these views is very close:

- For each grammar we can derive an **automaton**, and **vice-versa**
- Each kind of grammar corresponds to a particular kind of automaton, and **vice-versa**

Grammars

Formal definitions and terminology:

A **grammar** is a quadruple $\langle V_t, V_n, S, P \rangle$ where:

- $V_t$ is a finite set of **terminal symbols** (the alphabet)
- $V_n$ is a finite set of **non-terminal symbols** ($V$ denotes $V_t \cup V_n$, and $V_t \cap V_n = \emptyset$)
- $S$ is a distinguished non-terminal symbol called the **start** symbol
- $P$ is a set of **productions**, written $\alpha \rightarrow \beta$ where $\alpha \in V^*V_nV^*$ and $\beta \in V^*$.
  - $\alpha$ is a string of terminal and non-terminal symbols, **including at least one non-terminal**.
  - $\beta$ is a string of zero or more terminal and non-terminal symbols.
Example:

\[ G = \langle \{a, b\}, \{S, A\}, S, \{ S \to aAb, aA \to aaAb, A \to \epsilon \} \rangle \]

- Terminals: \{a, b\}
- Non-terminals: \{S, A\}
- Start symbol: S
- Productions:
  
  \[ S \to aAb \]
  \[ aA \to aaAb \]
  \[ A \to \epsilon \]

(Usually we only give the productions, using capital letters for non-terminals (S for the start symbol) and lower case for terminals — then the other details can be inferred. We can abbreviate \[ \alpha \to \beta | \gamma \] for the two productions \[ \alpha \to \beta \] and \[ \alpha \to \gamma \].)

Example ctd

\[ S \to aAb, \quad aA \to aaAb, \quad A \to \epsilon. \]

A sample derivation:

\[ S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb \]

The last string is a sentence. The others are merely sentential forms.

The language generated by this grammar can also be described as

\[ \{ a^n b^n | n \in \mathbb{N}, n \geq 1 \} \]

The same language is generated by the rather simpler

\[ S \to aSb, \quad S \to ab. \]

(Grammars are not 1-to-1 with languages.)

Derivations

Productions are substitution rules: If there is a production \[ \alpha \to \beta \] then we can rewrite any string \[ \gamma \alpha \rho \] as \[ \gamma \beta \rho \] we say \[ \gamma \alpha \rho \Rightarrow \gamma \beta \rho \]

Derivations are the reflexive-transitive closure of these substitutions:

\[ \alpha \Rightarrow^* \beta \quad (\beta \text{ derived from } \alpha \text{ using 0 or more steps}) \]

The language generated by a grammar is the set of strings of \[ V_1 \] — i.e. terminals only — that can be derived from the start symbol:

\[ \{ \alpha | S \Rightarrow^* \alpha \land \alpha \in V_1^* \} \]

The sentential forms are \[ \{ \alpha | S \Rightarrow^* \alpha \land \alpha \in V^* \} \]

The Chomsky Hierarchy

Chomsky classified grammars on the basis of the form of their productions:

Unrestricted: (type 0) no constraints.

Context-sensitive: (type 1) the length of the left hand side of each production must not exceed the length of the right (with one exception).

Context-free: (type 2) the left of each production must be a single non-terminal.

Regular: (type 3) As for type 2, and the right of each production is also constrained (details to come).

There are many interesting intermediate types of grammar also.

Classification of Languages

A language is type $n$ if it can be generated by a type $n$ grammar.

Going down the hierarchy of grammars, there are more restrictions placed on the form of productions that are permitted.

For example, if there is a type 2 grammar for some language then there are also type 1 and type 0 grammars for that language.

To show that a language is type 2 we must provide a type 2 grammar for it.

To show that a language is not type 2 we must show that there cannot be a type 2 grammar for it.

Example — language $\{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$

We already saw two grammars for this language:

- Unrestricted (type 0):
  $$ S \rightarrow aAb $$
  $$ aA \rightarrow aaAb $$
  $$ A \rightarrow \varepsilon $$

- Context-free (type 2):
  $$ S \rightarrow ab $$
  $$ S \rightarrow aSb $$

Last week we proved that there is no FSA for this language (and therefore, as we will see, no regular grammar), so the language must be context-free.

Regular (Type 3) Grammars

Productions are all of the form (for a right-linear grammar):

$$ A \rightarrow aB $$
$$ A \rightarrow a $$
$$ A \rightarrow \varepsilon $$

or all of the form: (for a left-linear grammar):

$$ A \rightarrow Ba $$
$$ A \rightarrow a $$
$$ A \rightarrow \varepsilon $$

There is no deep difference between right and left linear; we will make the arbitrary choice to stick to right-linear.

Their essential feature is that they generate sentences one symbol at a time, terminating on terminals $a$ or the empty string $\varepsilon$.

The languages they generate are the regular languages.

Regular Languages - Many Views

The following are equivalent:

- $L$ is the language generated by a right-linear grammar;
- $L$ is the language generated by a left-linear grammar;
- $L$ is the language accepted by some DFA;
- $L$ is the language accepted by some NFA;
- $L$ is the language specified by a regular expression.

We have already seen how DFA and NFA coincide (the subset construction).

We will now show that NFA and (right-)linear grammars specify the same languages.
From NFAs to Right-linear Grammars

Take an NFA \((\Sigma, S, S, F, R)\).

Recall - (alphabet, states, start state, final states, transition relation).

Our equivalent right-linear grammar will have

- as terminal symbols the alphabet \(\Sigma\);
- as non-terminal symbols the states \(S\);
- as start symbol the start state \(S\);
- as productions,
  - for each \(R(T, a, U)\) in the transition relation, add \(T \to aU\);
  - for each final \(T \in F\) add \(T \to \varepsilon\).

NFAs to Right-linear Grammars - Example

A right-linear grammar accepting the same language

\[
\begin{align*}
S & \to aS \\
S & \to aS_1 \\
S_1 & \to bS_1 \\
S_1 & \to bS_2 \\
S_2 & \to cS_2 \\
S_2 & \to cS_3 \\
S_3 & \to \varepsilon
\end{align*}
\]

Right-linear Grammars to NFAs - Example

\[
\begin{align*}
S & \to 0 \\
S & \to 1T \\
T & \to \varepsilon \\
T & \to 0T \\
T & \to 1T
\end{align*}
\]

generates the language of binary integers, and the automaton

From Right-linear Grammars to NFAs

Given a right-linear grammar \((V_t, V_n, S, P)\), the equivalent NFA has

- as alphabet the terminal symbols \(V_t\);
- as states the non-terminal symbols \(V_n\) along with a new state \(S_f\) (for Final);
- as start state the start symbol \(S\);
- as final states \(S_f, \text{ and all non-terminals } T\) such that there exists a production \(T \to \varepsilon\);
- as transition relation
  - for each \(T \to aU\) the transition \((T, a, U)\);
  - for each \(T \to a\) the transition \((T, a, S_f)\).
Context-Free (Type 2) Grammars (CFGs)

Productions are all of the form: $A \rightarrow \omega$ where $A \in V_n$ and $\omega \in V^*$. So the left hand side of each production must be one non-terminal on its own (as with regular grammars); the right hand side can be anything. Therefore independent of its context $A$ can be replaced by $\omega$. (Hence the name “context free”)

(This contrasts with context-sensitive grammars which may have productions like $\alpha A \beta \rightarrow \alpha \omega \beta$

There $A$ may be replaced by $\omega$ only in the context $\alpha \_ \_ \beta$).

Languages in Backus-Naur Form are context-free grammars.

Example

Can we design a CFG for the language $\{a^m b^n c^{m-n} \mid m \geq n \geq 0\}$?

Strategy: split the words in this language into sections:

- $a^m$, followed by
- $b^n$, followed by
- $c^{m-n}$.

and then use different non-terminals to generate (i) the first and third section, then (ii) the second section:

$$S \rightarrow aSc \mid T$$
$$T \rightarrow aTb \mid \varepsilon$$

Example ctd

An example derivation of the word $aaabbc$:

$$S \Rightarrow aSc$$
$$\Rightarrow aTc$$
$$\Rightarrow aaTbc$$
$$\Rightarrow aaaTbbc$$
$$\Rightarrow aaabbc$$

Parse trees

Derivations can be more naturally displayed as parse trees:
The Power of Context-Free Grammars

A fun example:

http://pdos.csail.mit.edu/scigen