

Section 4.8: Converting Regular Expressions and Finite Automata to Grammars

There are simple algorithms for converting regular expressions and finite automata to grammars. Since we have algorithms for converting between regular expressions and finite automata, it is tempting to only define one of these algorithms.

Copyright © 2003–5 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The \LaTeX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

(4.8) Introduction (Cont.)

But translating FAs to regular expressions is expensive, and often yields large regular expressions. Thus it would be impractical to translate FAs to grammars by first translating them to regular expressions, and then translating the regular expressions to grammars. Hence it is important to give a direct translation from FAs to grammars.

(4.8) Introduction (Cont.)

But translating FAs to regular expressions is expensive, and often yields large regular expressions. Thus it would be impractical to translate FAs to grammars by first translating them to regular expressions, and then translating the regular expressions to grammars. Hence it is important to give a direct translation from FAs to grammars.

Although it would be satisfactory to translate regular expressions to grammars by first translating them to FAs, and then translating the FAs to grammars, it's easy to define a direct translation, and the results of the direct translation will be a little nicer.

(4.8) Converting Regular Expressions to Grammars

Regular expressions are converted to grammars using a recursive algorithm that makes use of the operations on grammars that were defined in Section 4.7. The structure of the algorithm is very similar to the structure of our algorithm for converting regular expressions to finite automata.

The algorithm is implemented in Forlan by the function

```
val fromReg : reg -> gram
```

of the `Gram` module. It's available in the top-level environment with the name `regToGram`.

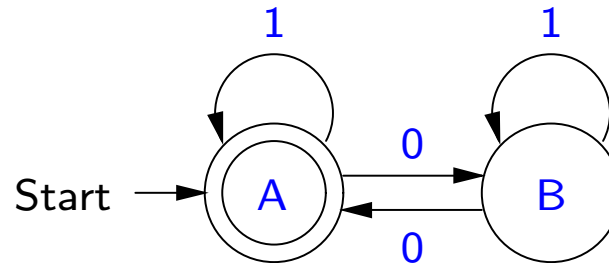
(4.8) Converting Regular Expressions to Grammars (Cont.)

Here is how we can convert the regular expression $01 + 10(11)^*$ to a grammar using Forlan:

```
- val gram = regToGram(Reg.input "");
@ 01 + 10(11)*
@ .
val gram = - : gram
- Gram.output("", Gram.renameVariablesCanonically gram);
{variables}
A, B, C, D, E, F
{start variable}
A
{productions}
A -> B | C; B -> 01; C -> DE; D -> 10; E -> % | FE;
F -> 11
val it = () : unit
```

(4.8) Converting Finite Automata to Grammars

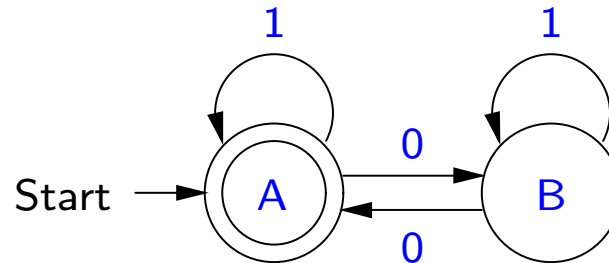
We'll explain the process of converting finite automata to grammars using an example. Suppose M is the DFA



The variables of our grammar G consist of the states of M , and its start variable is the start state A of M . (If the symbols of the labels of M 's transitions conflict with M 's states, we'll have to rename the states of M first.) We can translate each transition (q, x, r) to a production

(4.8) Converting Finite Automata to Grammars

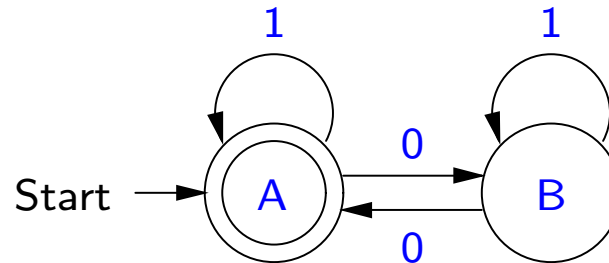
We'll explain the process of converting finite automata to grammars using an example. Suppose M is the DFA



The variables of our grammar G consist of the states of M , and its start variable is the start state A of M . (If the symbols of the labels of M 's transitions conflict with M 's states, we'll have to rename the states of M first.) We can translate each transition (q, x, r) to a production $q \rightarrow xr$. And, since A is an accepting state of M , we add the production

(4.8) Converting Finite Automata to Grammars

We'll explain the process of converting finite automata to grammars using an example. Suppose M is the DFA



The variables of our grammar G consist of the states of M , and its start variable is the start state A of M . (If the symbols of the labels of M 's transitions conflict with M 's states, we'll have to rename the states of M first.) We can translate each transition (q, x, r) to a production $q \rightarrow xr$. And, since A is an accepting state of M , we add the production $A \rightarrow \%$. This gives us the grammar

$$A \rightarrow \% \mid 0B \mid 1A,$$

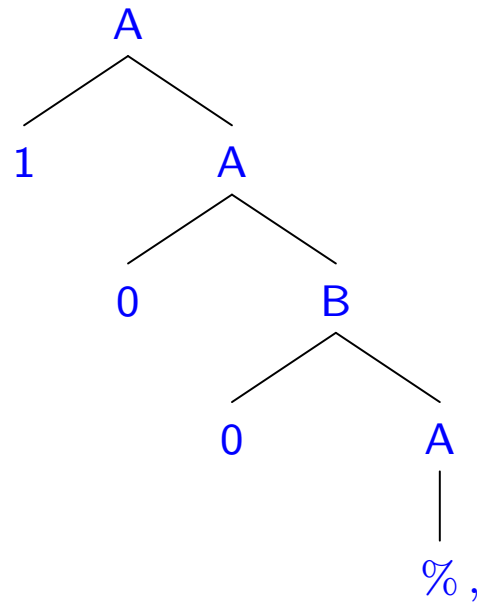
$$B \rightarrow 0A \mid 1B.$$

(4.8) Converting FA's to Grammars (Cont.)

Consider, e.g., the valid labeled path for M

$$A \xRightarrow{1} A \xRightarrow{0} B \xRightarrow{0} A,$$

which explains why $100 \in L(M)$. It corresponds to the valid parse tree for G



which explains why $100 \in L(G)$.

(4.8) Converting FA's to Grammars (Cont.)

The Forlan module `Gram` contains the function

```
val fromFA : fa -> gram
```

which implements our algorithm for converting finite automata to grammars. It's available in the top-level environment with the name `faToGram`.

(4.8) Converting FA's to Grammars (Cont.)

Suppose `fa` of type `fa` is bound to M . Here is how we can convert M to a grammar using Forlan:

```
- val gram = faToGram fa;
val gram = - : gram
- Gram.output("", gram);
{variables}
A, B
{start variable}
A
{productions}
A -> % | 0B | 1A; B -> 0A | 1B
val it = () : unit
```

(4.8) Consequences of Conversion Functions

Because of the existence of our conversion functions, we have that every regular language is a context-free language.

On the other hand, the language $\{0^n 1^n \mid n \in \mathbb{N}\}$ is context-free, because of the grammar

$$A \rightarrow \% \mid 0A1,$$

but is not regular, as we proved in Section 3.13.

Summarizing, we have:

Theorem 4.8.1

The regular languages are a proper subset of the context-free languages: $\text{RegLan} \subsetneq \text{CFLan}$.