Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2013

Ana Bove

May 23rd 2013

Overview of the course

Overview of the Course

We have covered/you should know chapters 1–5 + 7 + (8):

Formal proofs: mainly proofs by induction

Regular languages: DFA, NFA, ϵ-NFA, RE
Algorithms to transform one formalism to the other
Properties of RL

Context-free languages: CFG
Properties of CFL

Turing machines: Just a bit

Formal Proofs

We have extensively used formal proofs in the course; mainly proofs by (course of value) induction:

- On the structure of the argument;
- On the length of the string;
- On the length of the derivation;
- On the height of a parse tree.

Finite Automata and Regular Expressions

FA and RE can be used to model and understand a certain situation/problem.

Example: Consider the problem with the man, the wolf, the goat and the cabbage.

Also the Gilbreath's principle. There we went from NFA $\rightarrow$ DFA $\rightarrow$ RE.

They can also be used to describe (parts of) a certain language.

Example: RE are used to specify and document the lexical analyser (*lexer*) in languages (the part of the compiler reading the input and producing the different *tokens*).

The implementation performs the steps RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ min DFA.

Example: Using Regular Expression to Identify the Tokens

```
Tokens = Space (Token Space)*
Token  = TInt | TId | TKey | TSpec
TInt   = Digit Digit*
Digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' |
         '7' | '8' | '9'
TId    = Letter IdChar*
Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
IdChar = Letter | Digit
TKey   = 'i''f' | 'e''l''s''e' | ...
TSpec  = '+''+' | '+' | ...
Space  = (' ' | '\n' | '\t')*
```

Regular Languages

Intuitively, a language is regular when a machine needs only limited amount of memory to recognise it.

We can use the Pumping lemma for RL to show that a certain language is not regular.

Closure properties can help us to deduce if a certain language is regular or not.

There are several decision properties for RL.
Some of them are:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}? \qquad \mathcal{L} \subseteq \mathcal{L}'? \qquad \mathcal{L} = \mathcal{L}'?$$

Context-Free Grammars

CFG play an important role in the description and design of programming languages and compilers.

CFG are used to define the syntax of most programming languages.

Parse trees reflect the structure of the word and contain concrete features.

In a compiler, the parser builds the abstract syntax tree of the input.

A grammar is ambiguous if a word in the language has more than one parse tree.

LL grammars are unambiguous.

There are algorithms to decide if a grammar is LL(1).

Context-Free Languages

These languages are generated by CFG.
It is enough to provide a stack to a NFA in order to recognise CFL.

We can use the Pumping lemma for CFL to show that a certain language is not context-free.
Closure properties can help us to deduce if a certain language is context-free or not.

There are fewer decision properties for CFL. We have seen

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}?$$

However there are no algorithms to determine whether $\mathcal{L} \subseteq \mathcal{L}'$ or $\mathcal{L} = \mathcal{L}'$.

There is no algorithm either to decide if a grammar is ambiguous or a language is inherently ambiguous.

Turing Machines

Simple but powerful devices.

They can be though of as a DFA plus a tape which we can read and write, and move to the right and to the left.

Define the recursively enumerated languages.

It allows the study of *decidability*: what can or cannot be done by a computer (halting problem).

Computability vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

Church-Turing Thesis

In the 1930's there has been quite a lot of work about the nature of *effectively computable (calculable) functions*:

- Recursive functions by Stephen Kleene;
- λ-calculus by Alonzo Church;
- Turing machines by Alan Turing.

The three computational processes were shown to be equivalent by Church, Kleene, (John Barkley) Rosser (1934—6) and Alan Turing (1936—7).

The *Church-Turing thesis* states that if an algorithm (a procedure that terminates) exists then, there is an equivalent Turing machine, recursively-definable function, or a definable λ-function for that algorithm.