MAU22C00 Lecture 3

John Stalker

Trinity College Dublin

Announcements

- Lecture notes are now on the module webpage and Blackboard.
- Tutorials start this week.
- The first assignment is on the module webpage and Blackboard.
- It is due Friday, via Blackboard. Contact me if you don't yet have Blackboard access.
- We don't have markers yet so I'm not sure how quickly they will be marked.

Hierarchy of languages

Recall that by language in this module we always mean a formal language. The most important classes of languages are

- Finite, i.e. just a finite set of lists of tokens
- Regular: We'll define this later. We'll also see that these are the languages recognised by a finite state automaton.
- Context free: These are the languages generated by phrase structure grammars, i.e. grammars of the sort in all our examples so far, where you replace a single symbol by lists of symbols. They are also the languages recognised by pushdown automata.
- Context sensitive: These are languages which have a more complicated grammar, where you can replace longer lists of symbols by other lists of symbols.

Hierarchy of languages, continued

- Recursively enumerable: This are the languages which can by recognised by a Turing machine.
- General: Any language, including ones for which there is no grammar and no machine which can recognise them.

Every language in one category is in all later ones as well, e.g. a context free language is also context sensitive, recursively enumerable and general!

There is a (rough) correspondence between language levels and machine types and also between language levels and grammar types.

Languages may admit multiple grammars and multiple recognisers. Level is determined by the *lowest* level grammar or machine which works, not the one you happen to be staring at!

Language example: multiples of 3

Consider the language of integer multiples of 3, e.g. 27, -9, 0, etc. The tokens are just the - sign and the digits 0, 1, ..., 9.

We'll disallow variant forms like 036, -0, --12, etc.

There are a few different grammars you can use for this language. One is

```
multiple_of_3 : '0' | pos_integer_0_mod_3
              | '-' pos_integer_0_mod_3 ;
pos_integer_0_mod_3 : pos_digit_0_mod_3
                    | pos_integer_0_mod_3 digit_0_mod_3
                    | pos_integer_1_mod_3 digit_2_mod_3
                    | pos_integer_2_mod_3 digit_1_mod_3 ;
```

Multiples of 3, continued

```
pos_integer_1_mod_3 : digit_1_mod_3
                    | pos_integer_0_mod_3 digit_1_mod_3
                    | pos_integer_1_mod_3 digit_0_mod_3
                    | pos_integer_2_mod_3 digit_2_mod_3 ;
pos_integer_2_mod_3 : digit_2_mod_3
                    | pos_integer_0_mod_3 digit_2_mod_3
                    | pos_integer_1_mod_3 digit_1_mod_3
                    | pos_integer_2_mod_3 digit_0_mod_3 ;
digit_0_mod_3 : '0' | pod_digit_0_mod_3 ;
pos_digit_0_mod_3 : '3' | '6' | '9' ;
digit_1_mod_3 : '1' | '4' | '7' ;
digit_2_mod_3 : '2' | '5' | '8' ;
```

We can construct a finite state automaton recogniser because we don't really need to know which digits we've already seen, just what the remainder is on dividing by `3`. Initially we need to keep track of whether we've seen a `0` or a `-` as well.

A diagram

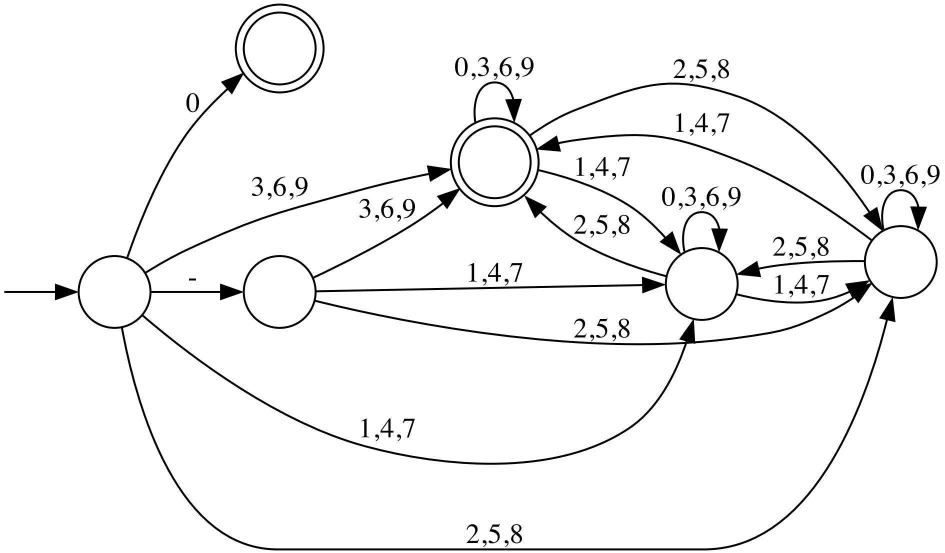

Figure 1: A finite state automaton

A pushdown automaton recogniser

A pushdown automaton is an idealised machine with a finite number of states and access to a stack.

We need some input mechanism. The usual way to describe this as reading tokens from a tape.

An equivalent, and more convenient, approach is to think of the list of input tokens as a restricted stack: we're allowed to pop items off and query whether it's empty, but not push items onto it.

So we start off with two stacks: the input stack, which initially contains the input tokens, with the first at top and the last at the bottom, and the working stack, which is initially empty.

A pushdown automaton recogniser, continued

- Unless the input stack is empty pop of the top element. If it's a (push it onto the working stack.
- If it's a) then check whether the working stack is empty. If it is then push the) onto it. If it's not empty then pop it as well.
- If that element is a) then push two)'s onto the working stack. If it was a (then do nothing.
- In any case, return to the first step.

At the end the input stack will be empty. If the working stack is also empty then the input belonged to the language. If not then it didn't.

This works because after each step the working stack contains the input read so far, with any pairs of matching parentheses removed.

A diagram

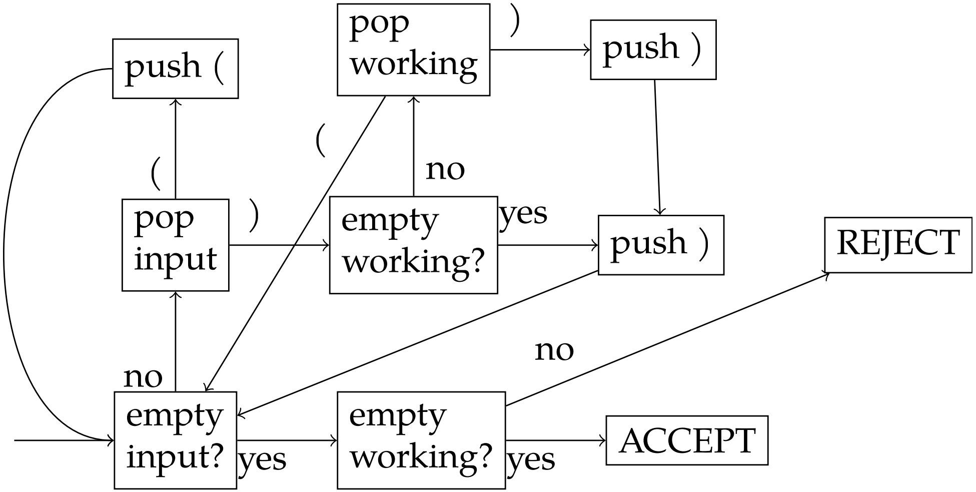

Figure 2: Recogniser for balanced parentheses

Comments

There are slightly easier ways to do this but this one has the advantage of generalising to multiple types of braces.

Because there is a pushdown automaton which recognises it the language of balanced parentheses is context free, so also context sensitive, and recursively enumerable.

We already knew that from the grammar for this language.

Is the language of balanced parentheses finite? No.

Is it regular? We don't know (yet).

Grammar and arithmetic

How much of arithmetic can we turn into grammar?

We saw that we can describe divisibility by 3 purely grammatically, in fact with a regular language. In fact you can do the same for divisibility by any number.

What about the language of prime numbers?

It turns out this isn't a regular language, but we can't (yet) prove this.