MAU22C00 Lecture 1

John Stalker

Trinity College Dublin

Unfinished business

Languages, logic, graphs, and idealised machines are all related.

Numbers, sets, monoids, etc. are also related, although I didn't have time for them in this lecture.

The first chapter of the notes goes through the module enrollment example in much more detail, introducing a lot of ideas which I also didn't have time for here.

Most of those ideas will put in an appearance later in the module so reading that chapter will help you understand how the module fits together.

It's not necessary to understand everything on the first pass.

What is a language?

Formal languages are built from *tokens*, which are partitioned into *symbols*.

We might have a symbol `animal`, with tokens `cat`, `dog`, `rabbit`, `wombat`, etc. or a symbol `digit` with tokens `0`, `1`, `2`, …, `9`.

There could be infinitely many tokens in a symbol. We could have a symbol `integer`, for example, and every integer is an `integer`. There could be just one. We could have a symbol whose only token is ¬.

There should be only finitely many symbols though, and there should be algorithms for determining which symbol a token belongs to and for generating all the tokens for a given symbol.

A *language* is a set of lists of tokens such that if a list is a member then so is any other list where we replace one token with another belonging to the same symbol.

Example language: linear equations

A language for linear equations might contain the following symbols:

- integers, e.g. 17, 42, -5, 0
- variables, e.g. x, y, z
- the operators $+$ and $-$
- the equals sign $=$
- separators, to tell us where one equation stops and then next begins

Not all lists of these tokens are grammatical though, e.g. $+- =$.

Grammar

By a *grammar* for a language we mean something like this:

```
%token INTEGER VARIABLE OPERATOR
%token EQUALS SEPARATOR
%start equations
%%
equations : equation
          | equations SEPARATOR equation ;
equation : side EQUALS side ;
side : term | side OPERATOR term ;
term : INTEGER | VARIABLE | INTEGER VARIABLE ;
```

This is a `yacc` grammar specification. `yacc` is a *parser generator*. It takes a description of a formal language in its own formal language and produces a parser.

Finiteness

Various things need to be finite:

- Lists are finite. That's part of what I mean by "list".
- The set of symbols is finite.
- The grammar is finite, i.e. has finitely many rules of finite length.
- The algorithm which assigns symbols to tokens is finite and terminates in finite time.
- The algorithm which generates the tokens for a symbol is finite and will generate any token in finite time, but may not terminate.

The langage itself need not be finite though, and typically isn't. Even the set of tokens doesn't have to be finite.

Generative grammar

This approach to grammar is called "generative grammar". It describes a language by telling how you could generate any list of tokens in the language.

In our linear equations example, starting from the start symbol

```
equations
```

we could expand it to `equation` or to `equations SEPARATOR equation`. We'll choose the second option, so we now have

```
equations SEPARATOR equation
```

`SEPARATOR` is what's called a terminal symbol, it can't be expanded, just replace by a token. Let's say our separator is a comma. Now we have

```
equations , equation
```

Generative grammar, continued

We now have

`equations , equation`

We can expand `equations` again, either to `equation` or to `equations SEPARATOR equation`. This time we'll choose the first first option, so we have

`equation , equation`

`equation` has only one expansion, to `side = side`. In theory we should expand symbols one at a time but I'll start batching the expansions to save time. So now we have

`side EQUALS side , side EQUALS side`

Generative grammar, continued

EQUALS is a terminal symbol with only one token associated with it, namely =, so now we have

`side = side , side = side`

`side` has two possible expansions, `term` or `side OPERATOR term`. We'll choose different expansions for the different occurences:

`side OPERATOR term = term , term = term`

OPERATOR is another terminal, expanding to + or −. We'll choose − and also expand the side and the various terms

`term - INTEGER = VARIABLE , INTEGER VARIABLE = INTEGER`

Generative grammar, continued

We're currently at

```
term - INTEGER = VARIABLE , INTEGER VARIABLE = INTEGER
```

We can expand our only remaining nonterminal symbol, `term`, to `INTEGER VARIABLE` and replace the various terminals with tokens associated with them.

```
INTEGER VARIABLE - 1 = x , 2 y = 1
```

Now we have only terminals, which can replace with appropriate tokens

3 z - 1 = x , 2 y = 1

Generative grammar, continued

The preceding shows that `3 z - 1 = x , 2 y = 1` is an element of our language.

We had to make a lot of choices and making different choices would have given us different elements of the language.

Any list of tokens we can get by making *some* set of choices is an element of the language and only those lists of tokens are elements of the language.

The process didn't have to terminate. I could have kept expanding `equations` to `equations SEPARATOR equation` forever, generating longer and longer lists of symbols, never seeing a token or even a terminal symbol, for example.

Parsing

From a computer science point of view generative grammar is a weird way to describe languages.

We'd like to be able to *parse* languages. Starting from a list of tokens we'd like to assign them their terminal symbols and then combine sublists into other symbols, continuing until we reach the start symbol.

It's not obvious how you take a grammar like the one above, what's called a *phrase structure grammar* and generate a parser.

Fortunately there are tools like `yacc`.

Parsers are closely related to recognisers, a.k.a validators. A recogniser for a language is a machine (think program) which reads a list of tokens and tells you whether or not it belongs to the language.

It's like a parser, except that it doesn't parse.

Example language: balanced parentheses

The language of balanced parentheses is a language whose tokens are "(" and ")". It consists of those lists of tokens where we can match open and close parentheses in such a way that they are nested and occur in the correct order.

Every element of this language has the same number of ('s and)'s, but that's not sufficient. ")(", for example, does not belong to the language.

We could also allow pairs of "[" and "]" or "{" and "}", but won't.

A grammar for the language of balanced parentheses is

```
%start S1
%%
S1 : /* empty */ | S2 ;
S2 : ( ) | ( S2 ) | ( ) S2 | ( S2 ) S2 ;
```

Balanced parentheses, continued

```
%start S1
%%
S1 : /* empty */ | S2 ;
S2 : ( ) | ( S2 ) | ( ) S2 | ( S2 ) S2 ;
```

Every list generated by these rules has balanced parentheses.

Less obviously, every element of the language can be generated by these rules.

Still less obviously, it can be generated in only one way.

Key idea: Every non-empty element of the language starts with a (. This (must have a matching). The list of tokens in between has matching parentheses, as does the list after. Either or both of those lists could be empty.