CS24210 Syntax Analysis and Topics in Programming Languages

Edel Sherratt

About this Module

Your Aim: to become familiar with the concepts required for specifying and implementing programming lanuguages

Why?

Learning Outcomes – for the whole course

On successful completion of this course, you should be able to

- make effective use of compilers and other language processing software;
- apply the techniques and algorithms used in compilation to other areas of software engineering;
- understand the need for implementation independent semantics of programming languages;

How the course will work

- lectures, will require input from you
- scheduled practicals – absolutely essential
- reading – web based materials and library
- own practical work

About the Course

- Syllabus – www.aber.ac.uk/modules/current/CS24210.html
- The order will change
- The content may vary from year to year
- The examination will be based on the course *as taught*
- For the highest marks, evidence of additional reading, and more specially, *coherent thought* about the taught material will be demanded
- This means that your work should be paced
- Please point out any problems with the course content or presentation quickly – during the lecture, if necessary

Useful sources of information

- www.aber.ac.uk/~ dcswww/Dept/Teaching/Courses/CS24210
- It is not absolutely essential to purchase 'Bennett'; but you will need ready acces to some reasonably substantial book about compilation systems.
- Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools,* ISBN 0-201-10194-7
- Fischer, C.N. and Leblanc, R.J. *Crafting a compiler with C,* ISBN 0-8053-2166-7 (or you may prefer the Java edition).
- Wirth, *Compiler Construction*, ISBN 0-201-40353-6

Language Processing Software

- Editors – e.g. nedit, gvim, vi, sed, ...
- Compilers and Interpreters – e.g. javac, gcc, perl, bash, csh ...
- Text formatters – e.g. LaTeX, nroff ...

...and the Languages they process

- gvim: ascii text, programming languages, markup languages
- javac: java programs
- gcc: ANSI C programs
- LaTeX: text marked up with latex formatting instructions
- html: text marked up with html formatting instructions
- perl: perl scripts
- bash: bash shell commands
- csh: C shell commands

How do we learn to use Language Processing Software Effectively?

- Learn all of these languages and software systems individually?
- In a 10-credit module – 80 hours of your time???!
- How do we cope with new languages and tools?
- How do we make sure our knowledge is good for life?

Lasting, transferable knowledge and skills in Language Processing

- We need some practice
- and also some abstract knowledge – transferable skills

Abstraction as a tool to support effective use of language processing software

- What kind of information do we need to keep in mind?
  depends on what we're thinking about
  - all languages: the fact that languages have structure;
  - Java: class structure;
  - perl: pattern structure, loop structure, ...
- What kind of information do we omit?
  - all languages: possible structures
  - Java: semicolons etc.
  - perl: * / < > . etc.
- Different abstractions for different purposes?
  - different levels of abstraction level (as above)
  - OR different perspectives (next slide)

Note - there are many other ways to use abstraction when thinking about formal language.

Three aspects of language

- syntax – form, shape, structure
- semantics – meaning
- pragmatics – use

Lexical errors in Java

What's wrong with

```
fruit = 7peach;
```

An identifier may not begin with a digit

Lexical errors in Java

How about

```
fred = _fred_value:
joan = _joan_value;
```

statement delimiter is ; not :

Lexical errors in Java

Or maybe

```
Abstract Class StackD {
  Abstract Object accept(StackVisitorI ask);
}
```

keywords 'abstract', 'class' and 'object' should contain only lowercase letters

Lexical errors in Java

Or even

```
my_counter = 7 * (loop_counter + 5o5);
```

the letter 'o' should not appear in a number

Lexical errors in Java

What elements of the Java programming language were affected by these lexical errors?

- identifier
- delimiter
- keyword
- number (unsigned integer)

What constitutes the lexis of a programming language?

identifiers, delimiters, keywords, numbers (of various kinds), comments, ...

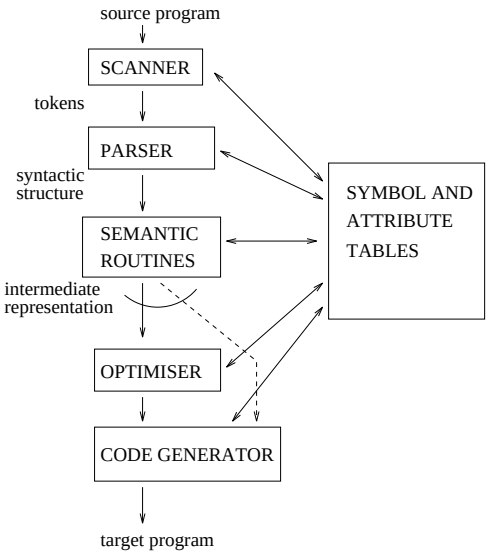
What part of the compiler catches lexical errors?

the scanner

How do we specify the lexis of a programming language?

We use regular expressions – we'll come back to this later.

Structure of a Compiler



Main Jobs of a Compiler

ANALYSIS of the Source Program

- lexical analysis
- syntactic analysis
- context checking

SYNTHESIS of the Target Program

- generation of intermediate representation
- optimisation
- code generation

The Scanner

The first job a compiler must do is to read the input character by character, and group the characters into useful bits called lexemes. This task is called lexical analysis.

Here are some lexemes:

␣for␣i␣in␣NumberofItems␣loop

␣ins␣:=␣42␣;␣

There are some nontrivial problems; illustrated by these FORTRAN lexemes:

␣2.E3␣

␣2␣.EQ.␣I␣

The Scanner

The lexemes are represented as tokens – often integers – to facilitate further processing by the compiler.

Tokens indicate the lexical class of the lexeme – identifier, reserved word, delimeter, ... &c. – and usually include a reference to the lexeme itself.

A typical encoding uses one or two digits to represent the lexical class of a token, with other digits acting as indices into tables where the lexemes themselves are stored.

For example,

ins := 42;

might be encoded as

0233 0076 0062 0011

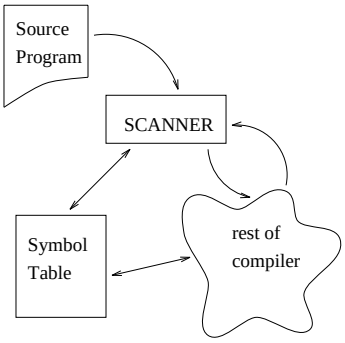
where the lowest order digit indicates the lexical class of the token, and the other digits index an entry for the token itself.

Other jobs done by the scanner

As well as doing its main job – recognising lexemes and encoding them as tokens – the scanner

- eliminates comments and "white space",
- formats and lists the source program, and
- processes compiler directives.

The Scanner in Context



Regular Expressions

Regular expressions provide a notation for describing the tokens used in modern programming languages.

A regular expression specifies a pattern. The set of character strings that match a regular expression is called a regular set.

For example,

$(underscore|letter)(underscore|letter|digit)*$

is a regular expression that specifies the set of Java identifiers – provided we have defined 'underscore', 'letter' and 'digit'.

Regular expressions and what they match

Conventional notation	Unix notation	matches
ϕ		no string
ϵ		the empty string
s	s	the string s
$A\|B$	$[AB]$	any string that matches A or B
AB	AB	any string that can be split into a part that matches A followed by a part that matches B
$A*$	$A*$	any string that consists of zero or more A's

s is any character string

A and B are regular expressions

Shorthands

- $A+$ one or more A's
  equivalent to $AA*$
- $A?$ zero or one A
  equivalent to $[\epsilon A]$
- $[0-9] = [0123456789]$

Constructing regular expressions

Write a regular expression that matches the Java keyword **class**

Constructing regular expressions

Using conventional notation, extend your regular expression so that it matches any of the Java keywords: **abstract class extends**

Write an equivalent regular expression using Unix notation.

Constructing regular expressions

Using conventional notation, write a regular expression that matches unsigned integers

Do the same, this time using Unix notation.

Constructing regular expressions

Extend each of your previous regular expressions so that they match signed integers.

More information

To find out all about regular expressions in Unix, see

```
man -s 5 regexp
```


Using Regular Expressions

- Unix applications
  - substitution using gvim
    $< range > s/ < regexp > / < substitution >$
  - searching files for patterns
    $grep \ < regexp > < path >$
- Specifying valid input

```
month   (JAN|FEB|MAR|APR|MAY|JUN|JUL|
            AUG|SEP|OCT|NOV|DEC)
day     [1-9]|([12][0-9])|(3[01])
year    ([19][0-9][0-9])|2000
```

- perl pattern specifications
  $t.e$ matches the or tee but not $tale$

Unix Pattern Matching

- There are inconsistencies between the ways different characters are used in different Unix applications and situations.
- In particular, characters are treated differently by the shell in filename expansion and by applications for pattern matching.
- For example, * means match 0 or more of the preceding expression in vi, gvim, sed, awk, grep and egrep, but * means match any string of characters in filename expansion

Unix Pattern Matching

- You can use single quotes (or, in some cases, double quotes) to bypass the shell and pass special characters to an application.
- Look up 'Unix in a Nutshell', section 6, Pattern Matching, for more detail
- Alternatively, look up the man pages for the applications you are using (man vi will tell you all you need for gvim).

Perl - Practical Extraction and Report Language

- scanning text files
- extracting information
- printing reports
- features of C, sed, awk and sh
- When sed, awk and shell scripts are not quite enough ...
- ...and C is a bit too much
- ...Perl is likely to be just right.

Perl example – derived from examples by Lynda Thomas

```
manuel% more read_a_file.pl
#!/usr/local/bin/perl

print "Filename? ";
$file = <STDIN>;        # read filename
open(FILE,$file);       # open the file
@file_content = <FILE>; # read into an array
close(FILE);
print @file_content;    # print the array
```


Perl example

```
manuel% cat names
lynda dave tom rory lynda rory lynda

manuel% read_a_file.pl
Filename? names
lynda dave tom rory lynda rory lynda

manuel% ls -l read_a_file.pl
-rwxr-----   1 eds      staff
    264 Oct 13 14:33 read_a_file.pl
```

Perl identifiers and types

```
manuel% cat trythis.pl
#!/usr/local/bin/perl

$scalar_string_id = "this is a string";
print $scalar_string_id . "\n";

$scalar_number_id = 7;
print "scalar_number_id = $scalar_number_id\n";

@array_id = ("peach","fig","apricot");
print @array_id[0].@array_id[2].@array_id[1]."\n\n";

%hash_id = ("peach", 2, "fig", 8, "apricot", 3);
print "We have ".@hash_id{"peach"}." peaches\n";
print "We have ".@hash_id{"fig"}." figs\n";
print "We have ".@hash_id{"apricot"}." apricots\n";
```


Perl identifiers and types

```
manuel% trythis.pl
this is a string
scalar_number_id = 7
peachapricotfig

We have 2 peaches
We have 8 figs
We have 3 apricots
```


Perl loops – from an example by Mike Slattery and Lynda Thomas

```
manuel% cat loopy.pl
#!/usr/local/bin/perl

while (<>) {
  @words = split;
  foreach $w (@words) {
    $count{$w}++;
  }
}

@sortedkeys = sort keys(%count);

foreach $w (@sortedkeys) {
  print "$w\t$count{$w}\n";
}
```


Perl loops

```
manuel% cat names
lynda dave tom rory lynda rory lynda

manuel% loopy.pl names
dave    1
lynda   3
rory    2
tom     1
```

Pattern matching in Perl

```
manuel% cat pattern.pl
#!/usr/local/bin/perl

while (<>) {
  @words = split;
  foreach $w (@words) {
    $count{$w}++;
  }
}

@sortedkeys = sort keys(%count);

foreach $w (@sortedkeys) {
  print "$w\t$count{$w}\n" if ($w =~ m/^l/);
}
```


Pattern matching in Perl

```
manuel% cat names
lynda dave tom rory lynda rory lynda marilyn

manuel% pattern.pl < names
lynda   3
```


A Perl subroutine that uses pattern matching

```
manuel% cat subroutine.pl
#!/usr/local/bin/perl

# from a program by Mike Slattery, June 1996

# swap takes a string, and replaces any words in %table
# by corresponding replacements

%table = ('i','you', 'you','i', 'my','your',
   'your','my');

sub swap {
  local ($in) = @_;
  local ($w, $head, $tail, $new);

  if ($in =~/[a-z]+/) {
    $w = $&;    # the match
    $head = $`; # before the match
    $tail = $'; # after the match

    # look up $new in %table; if not found, $new = $w
    $new = $table{$w} || $w;

    # put the sentence back together
    $head.$new.&swap($tail);
  }

  else { $in }
}
```


...and the main program

```
while ($input = <>) {
   chop $input;
   $input =~ tr/A-Z/a-z/;
   print &swap($input)."\n";
}
```

Running the program

```
manuel% subroutine.pl
i hate computers
you hate computers
i like running
you like running
you are daft
i are daft
i'm daft too
you'm daft too
i love computers really
you love computers really
^D
```


Perl modules

Perl modules encapsulate useful functions so that they can be made available to other Perl programs.

If you want to use the functions in a Perl module, you place a line calling the `use` function near the top of your program:

For example, provided the CGI module has been installed on your system,

```
use CGI;
```

will make the functions in the CGI module available to your program.

Example – Hello web-world

```
#!/usr/local/bin/perl

use CGI qw/:standard/;

print header;
print start_html, "Hello web-world!", end_html;
```


Example – Hello web-world!

If this program is in

```
/aber/eds/cgi-bin/hello_world.pl
```

then going to the url

```
http://users.aber.ac.uk/cgi-bin/user/eds/hello_world.pl
```

will cause it to print 'Hello web-world!' to the browser window.

A more interesting form – processForm

```
#!/usr/local/bin/perl

use CGI qw/:standard :html3/;

print header, title("form.pl");

sub processForm {
# this is done if the 'submit' button has been pressed

  print start_html,

    "You, ".param('name')."., whose favorite colors are ";

    foreach (param('colour')) {print "$_, " }

    print " are on a quest which is '", param('quest'),
    "', and are looking for the weight of an ",
    param('swallow'),
    ".  And this is what you have to say for yourself:",
    p, param('text'),

    hr,
    "And here is a list of the parameters you entered...",
    dump();

  end_html;
}
```


A more interesting form – printForm

```
sub printForm {
# this is run if 'submit' has not been pressed

  local @colour = qw/chartreuse azure puce cornflower
    olive opal mustard/;

  local @swallow = ("African Swallow",
    "Continental Swallow");

  print start_html,
    h2 ("Pop Quiz"),
    start_form,
      "What is thy name:",
         textfield(-name=>'name',-size=>20),

      p, "What is thy quest:",
         textfield(-name=>'quest',-size=>20),
```



```
      p, "What is thy favorite colour:",
      checkbox_group (-name=>'colour',
        -values=>\@colour),

      p, "What is the weight of a swallow:",
      radio_group (-name=>"swallow",
        -values=>\@swallow),

      p, "What do you have to say for yourself",
      textarea(-name=>'text',-rows=>5,-cols=>60),

      submit(-name=>'submit',
        -value=>"Press here to submit your query."),
    end_form,
    hr,

    "CGI.pm version of Steven E. Brenner's combined form",
  end_html;
}
```

A more interesting form – main program

```
if (param('submit')) {
  processForm;
} else {
  printForm;
}
```

Perl - some url's

- http://language.perl.com/
- http://www.perl.com/pace/pub
- http://www.perl.org/

Finite State Automata

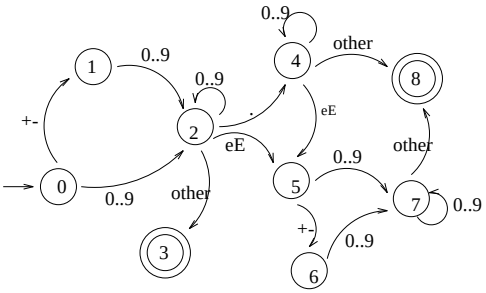
A finite state automaton (FSA) recognises strings that belong to a regular set.

It consists of

- a finite set of states,
- a set of transitions from state to state,
- a start state and
- a set of final states, called "accept states".

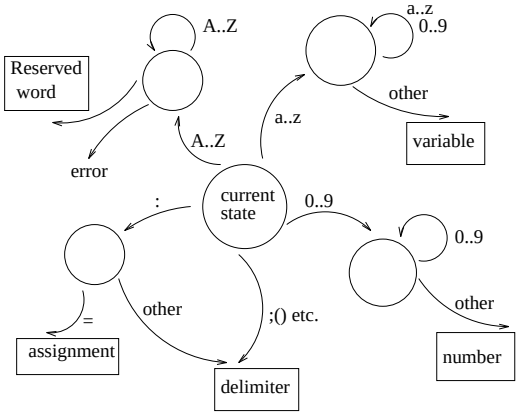
A FSA can be represented by a transition diagram – a directed graph whose nodes are labelled with state symbols, and whose edges are labelled with characters.

FSA that recognises numbers



This FSA is deterministic – for a given state and input character, the next state is uniquely determined.

Fragment FSA for a typical programming language



Pseudo code for a scanner

Assume one character, 'ch', has been read by 'getch', which reads a character, skips white space and produces a listing.

```
case ch of:
    ';' : token := semi
          getch
    ':' : getch; if ch = '='
                  then token := assign
                       getch
                  else token := colon
    '0' .. '9' : token := number
                 while ch in ['0' .. '9']
                       getch
                 endwhile
    ... &c.
```


Transition Tables

In general, a deterministic FSA can be programmed as a language independent driver that interprets a transition table. The transition table for the FSA that recognises numbers looks like this.

State	Character						
	+	-	.	e	E	0 ... 9	other
0	1	1				2	
1						2	
2			4	5	5	2	3
3							
4				5	5	4	8
5	6	6				7	
6						7	
7						7	8
8							

A scanner driver

```
State:= InitialState;
Read(CurrentChar);
loop
    NextState := TT(State,CurrentChar);
    exit when NextState = Error or
              CurrentChar = EOF;
    State := NextState;
    Read(CurrentChar);
end loop
if State in FinalStates
    then return valid token
    else lexical error
end if;
```


Using a Scanner Generator

There are algorithms for translating regular expressions to nondeterministic finite state automata (NFA), and for translating NFA to deterministic finite state automata (DFA).

These are used in scanner generators like lex, developed by M.E. Lesk and E. Schmidt of Bell labs.

Programming effort is limited to describing lexemes – an example of declarative programming.

Given input consisting of character class definitions, regular expressions with corresponding actions, and subroutines, lex generates a scanner called yylex.

Symbol Tables

The compiler uses symbol tables to collect and use information about names that appear in a source program.

Each time the scanner encounters a new name, it makes a symbol table entry for the name. Further information is added during syntactic analysis, and this information is used (and further extended) during semantic analysis and code generation.

Symbol Tables

The kind of information stored in a symbol table entry includes items like

- the string of characters denoting the name,
- the block or procedure in which the name occurs,
- attributes of the name (e.g. type and scope),
- the use to which the name is put (e.g. formal parameter, label ...)
- parameters, such as the number of dimensions in an array and their upper and lower limits, and
- the position in storage to be allocated to the name (perhaps).

The Symbol Table as an Abstract Data Type

Conceptually, a symbol table is a collection of pairs:

(name,information)

with operations to

- determine whether or not a name is in the table,
- add a new name,
- access the information associated with a given name,
- add new information for a given name, and
- delete a name, or a group of names.

Symbol Table Implementation

Symbol tables are typically implemented using

- linear lists,
- hash tables, or
- various tree structures.

In selecting an implementation, speed of access is traded against structural complexity.