Aims

• Curriculum issues
• Regular expressions
• Syntax definition
  • BNF
  • Parse tree
• Application: language processing
  • Lexical analysis
  • Parsing
  • Evaluation
Curriculum
Curriculum

• AQA material on Theory of Computation

3.4.2 Regular languages

3.4.2.1 Finite state machines (FSMs) with and without output

<table>
<thead>
<tr>
<th>Content</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be able to draw and interpret simple state transition diagrams and state transition tables for FSMs with no output and with output (Mealy machines only).</td>
<td></td>
</tr>
</tbody>
</table>

3.4.3.1 Backus-Naur Form (BNF)/syntax diagrams

<table>
<thead>
<tr>
<th>Content</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be able to check language syntax by referring to BNF or syntax diagrams and formulate simple production rules.</td>
<td></td>
</tr>
<tr>
<td>Be able to explain why BNF can represent some languages that cannot be represented using regular expressions.</td>
<td></td>
</tr>
</tbody>
</table>
Curriculum

3.4.2.3 Regular expressions

<table>
<thead>
<tr>
<th>Content</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know that a regular expression is simply a way of describing a set and that regular expressions allow particular types of languages to be described in a convenient shorthand notation.</td>
<td>For example, the regular expression $a(a</td>
</tr>
<tr>
<td>Be able to form and use simple regular expressions for string manipulation and matching.</td>
<td></td>
</tr>
<tr>
<td>Be able to describe the relationship between regular expressions and FSMs.</td>
<td>Regular expressions and FSMs are equivalent ways of defining a regular language.</td>
</tr>
<tr>
<td>Be able to write a regular expression to recognise the same language as a given FSM and vice versa.</td>
<td>A student's ability to write very simple regular expressions and FSMs will be assessed.</td>
</tr>
</tbody>
</table>

3.4.2.4 Regular language

<table>
<thead>
<tr>
<th>Content</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know that a language is called regular if it can be represented by a regular expression.</td>
<td>Also, a regular language is any language that a FSM will accept.</td>
</tr>
</tbody>
</table>
Curriculum – OCR

- Much less theory

<table>
<thead>
<tr>
<th>1.2.2 Applications Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) The nature of applications, justifying suitable applications for a specific purpose.</td>
</tr>
<tr>
<td>b) Utilities.</td>
</tr>
<tr>
<td>c) Open source vs closed source.</td>
</tr>
<tr>
<td>d) Translators: interpreters, compilers and assemblers.</td>
</tr>
<tr>
<td>e) Stages of compilation (Lexical analysis, Syntax analysis, Code generation and Optimisation).</td>
</tr>
<tr>
<td>f) Linkers and loaders and use of libraries.</td>
</tr>
</tbody>
</table>

No clear how much detail expected
Key Ideas

• Regular expressions (RExp)
  • Expressions used to specify a pattern for search
  • … closely related to FSM
  • … also used for ‘words’ in a language
  • Python has a comprehensive RExp library

• Syntax and parsing
  • Syntax: rules of a language and ways to write the rules
  • A parse tree shows that a sentence belongs to the language
  • Syntax can be recursive, while RExp are not
Pro / Cons of AQA Theory

• Con:
  • Unfamiliar
  • Quite mathematical and abstract

• Pro:
  • Content very clear
  • Questions simple
  • Can be applied to real problems
  • Beautiful: illustrates important principles
Regular Expressions
WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

IT'S HOPELESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

PERL!

TAP TAP
Regular Expressions

• A way to specify a set of strings
  • E.g. one’s formatted like an address

• Example
  \[ a(a|b)^* \]

• Read as: “a then, repeatedly, a or b”

• Examples of strings recognised
  a aa aaa aba abb abababab
RE Concepts

• Symbols – e.g. ‘a’
  • Match themselves
• Sequence
  • No operator
• Options – uses |
  • Pattern can be either this or that
• Repetition – uses * afterwards
  • Zero or more occurrences
• Use brackets as required
Exercise

• For each of the following regular expressions:
  1. Give several example of a matching string
  2. Describe the matching strings

• \((x|y|z)(1|2|3)\)
• \((Mr|Ms|Mrs)(Smith|Jones)\)
• \((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*\)
Regular Expressions in Python
Overview

• Python has a library for RE
• The regular expression
  • Are specified as a string
  • Use a richer language
  • Can be used to search (match) another string
• Lots of complexities
  • RE language
  • Extracting matched text – groups

\texttt{re.findall(pattern, string)}

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.
# Python RE Syntax Summary

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the beginning of a line</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of the line</td>
</tr>
<tr>
<td>.</td>
<td>Matches any character</td>
</tr>
<tr>
<td>\s</td>
<td>Matches whitespace</td>
</tr>
<tr>
<td>\S</td>
<td>Matches any non-whitespace character</td>
</tr>
<tr>
<td>\w</td>
<td>Matches any word character</td>
</tr>
<tr>
<td>\W</td>
<td>Matches any non-word character</td>
</tr>
<tr>
<td>*</td>
<td>Repeats a character zero or more times</td>
</tr>
<tr>
<td>+</td>
<td>Repeats a character one or more times</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>Matches a single character in the listed set</td>
</tr>
<tr>
<td>[^XYZ]</td>
<td>Matches a single character not in the listed set</td>
</tr>
<tr>
<td>[a-zA-Z0-9]</td>
<td>The set of characters can include a range</td>
</tr>
</tbody>
</table>
>>> string = "The Joy of Coding in Python"
>>> re.findall('a', string)
[]
>>> re.findall('o', string)
['o', 'o', 'o', 'o']
>>> re.findall('[A-Z][a-z]*o[a-z]*', string)
['Joy', 'Coding', 'Python']
>>> re.findall('[A-Z][^o]*', string)
['The J', 'C', 'Pyth']
>>> re.findall('[A-Z][^o]*\s', string)
['The ']
>>>
The \ Problem

- Suppose I want a pattern to find the last word in a sentence

```python
>>> string = "The Joy of Coding in Python."
>>> re.findall('\w*.', string)
['The ', 'Joy ', 'of ', 'Coding ', 'in ', 'Python. ']
```

```python
>>> re.findall(r'\w*\.', string)
['Python. ']
```

- \ escapes special characters
- BUT \ is already special in strings – raw strings
Exercises

- A name has the following elements
  - Mr
  - First name
  - Optionally, several initials of 1 letter each, followed by ‘.’
  - Last name

- Write and test a Python RE to recognise a name
- *Simplify the problem at first*
  - *Slight problem to mention*
Application of Regular Expressions

• Search
• Extracting information from web pages
  • Other semi-structured text data
  • E.g. surveillance of the web
• Bioinformatics
• Lexical analysis
  • Specifying words in a language
Lexical Analysis

- Python numbers

2.4.4. Integer literals

Integer literals are described by the following lexical definitions:

```plaintext
integer ::= decimalinteger | octinteger | hexinteger | bininteger
decimalinteger ::= nonzerodigit digit* | "0"+
nonzerodigit ::= "1"..."9"
digit ::= "0"..."9"
octinteger ::= "0" ("o" | "O") octdigit+
hexinteger ::= "0" ("x" | "X") hexdigit+
bininteger ::= "0" ("b" | "B") bindigit+
octdigit ::= "0"..."7"
hexdigit ::= digit | "a"..."f" | "A"..."F"
bindigit ::= "0" | "1"
```
Finite State Machine
Overview

• Many applications of related ideas
  • States
  • Transition between states
• Here, FSM for language specification
  • Equivalent to regular expressions
  • Basis for implementation of RE
Regular Languages

- States
  - Start
  - Final (or accepting)
- Transition – labelled with a symbol

Figure 1
Exercise

• Give examples of the strings accepted
• Describe the strings accepted
• Write and equivalent RExp

Figure 1
Implementing FSM

```
S1 = 1; S2 = 2; S3 = 3; S4 = 4

state = 1
string = input("The string: ")
while len(string) > 0:
    c = string[0]
    string = string[1:]
    if state == S1:
        if c == '1': state = S2
        if c == '0': state = S3
    elif state == S2:
        if c == '1': state = S4
        if c == '0': state = S3
    elif state == S3:
        if c == '1': state = S2
        if c == '0': state = S4
    else:
        if c == '1': state = S4
        if c == '0': state = S4
    if state == S4:
        print("Accepted")
    else:
        print("Not accepted")
```
Exercise

• Draw a FSM to recognise:
  1. A binary string with at least 2 ‘1’ bits in succession
     • E.g. 11 is accepted
     • E.g. 111 is accepted
     • E.g. 10101 is not accepted
  2. A 6 bit binary sequence with even parity
     • E.g. 101101 is accepted
     • E.g. 101100 is not accepted
• Write a regular expression equivalent to 1)
Syntax Definition and Parsing
Overview

• Regular expressions / FSM cannot define a language as general as a programming language
  • Finite state problem

• Parsing
  • Rules for syntax
  • Parse tree
  • Abstract syntax
Syntax Definition

• Backus-Naur Format (BNF)
  • sequence
  • choice: ‘|’
  • non-terminal – <…>
  • terminal

```plaintext
<expression> ::= <factor> | <factor> * <factor> | <factor> / <factor>
<factor> ::= <term> | <term> + <term> | <term> - <term>
<term> ::= - <expression> | <number>
<number> ::= <digit> | <digit> <number>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
Parse Tree

- Shows that an expression is valid in a syntax

**2 (c)** A tree can be used to demonstrate that an `<expression>` is valid. This is known as a parse tree.

Complete the parse tree below to show that $8 * 4 + 21$ is a valid `<expression>`.

```
<expression>
  /   \
<factor>  *
  /    \
<term>    <factor>
  /    \
<number>  /
<digit>
  8
```
Exercise

• Draw parse trees for the following expressions
  1. 123
  2. 1+2
  3. 1*2+3

• Using the grammar:

<expression> ::= <factor> | <factor> * <factor> | <factor> / <factor>
<factor> ::= <term> | <term> + <term> | <term> - <term>
<term> ::= - <expression> | <number>
<number> ::= <digit> | <digit> <number>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Abstract Syntax

• Prefer to use simpler trees
• E.g. for $1*2+3$

• Exercise
  • Redraw parse trees as abstract trees
Python Language Grammar

- Part of the grammar of expressions

```python
comp_op: '<' | '>' | '==' | '>=' | '<=' | '<' | '>' | '!=' | 'in' | 'not' | 'in' | 'is' | 'is' | 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr ('(' arith_expr ')') arith_expr*
arith_expr: term ('(' arith_expr ')') term*
term: factor ('(' factor ')') factor*
factor: ('+' | '-' | ' ~') factor | power
power: atom trailer* ['**' factor]
```
Language Definition and Processing
Overview – Simple Interpreter

• Stages of transformation
  1. characters → words: “Lexical analysis”
  2. words → tree: “Parsing”
  3. tree → value

---

Diagram:

1. String
2. Lexical Analysis
3. List of words
4. Parsing
5. Parse tree
6. Evaluate
7. Answer

---

Demo
Simple Words

• Simple words for our interpreter

word ::= number | operator
number ::= digit*
operator ::= + | - | * | /

• Exercise: what are the words from the following characters?
  • 100+ 70/ 8
  • +++
  • 12 3 4*
## Tokeniser: converts strings to a list of words

```python
def tokens(cs):
    tks = []  # list of tokens
    NUM = 0  # state 1: part way through a number
    NONUM = 1  # state 2: not in a number
    state = NONUM  # current state 2: not in a number
    word = ""  # current token (or word)
    while len(cs) > 0:  # while there are more chars
        c = cs[0]  # get first character
        cs = cs[1:]  # remaining characters
        if state == NUM:
            ...
            # characters of number
        elif state == NONUM:
            ...
            # characters to start a word
        if state == NUM:
            tks.append(word)
    return tks
```

Code I
if c.isdigit():
    word = word + c
elif isop(c):
    tks.append(word)
    tks.append(c)
    word = ""
    state = NONUM
elif c.isspace():
    tks.append(word)
    word = ""
    state = NONUM
else:
    print("Illegal ...:", c)
    sys.exit()
Simple Grammar

- Expressions with numbers, operators and brackets

\[
\begin{align*}
\text{exp} & \ ::= \text{factor} \ ((\text{'+' | '-'}) \ \text{factor})^* \\
\text{factor} & \ ::= \text{term} \ ((\text{'*' | '/'}) \ \text{term})^* \\
\text{term} & \ ::= \text{number} \ | \ '(' \ \text{exp} \ ')'
\end{align*}
\]

- Exercise: which of the following are valid
  - 1 + 2 + 3
  - -1 + 3
  - (1 + 2) * 3
  - 1 + 2 * 3
Parser Results

• Tree represented by pair
  • (operator, [left, right])

Enter expression: 1+2+3
(' +', [('+', [(1, '1'), (1, '2')]), (1, '3')])

Enter expression: 2*3+4*5
(' +', [('*', [(1, '2'), (1, '3')]), ('*', [(1, '4'), (1, '5')])])

• Exercise
  • Draw the abstract syntax trees
  • … do your own examples
Evaluation

- Recursive over tree
- Simplest part!

```python
def evaluate(exp):
    op, exps = exp
    if op == INT:
        return int(exps)
    else:
        a = evaluate(exps[0])
        b = evaluate(exps[1])
        if op == '+':
            return a + b
        elif op == '-':
            return a - b
        elif op == '*':
            return a * b
        elif op == '/':
            return a // b
        else:
            print("Error")
```
Summary

• Language topics link to
  • Understanding about programming languages
  • Recursion
  • State machines

• How Python works

Challenge problem: enhance the interpreter to handle variables and assignment:

\[
\begin{align*}
  v1 &= 10 \\
  v2 &= v1 \times 2 \\
  v2 &= v2 \times 3
\end{align*}
\]