Contents

Finite-State Machines

A finite-state recognising machine is described by:

- A finite set of states
- A finite set of input symbols
- A transition function `a` which assigns a new state to every combination of state and input

  Design a finite state machine to check an input sequence of 1's & 0's for odd parity

- What is parity?
  - Number of 1's in a string
- What is the parity of an empty string?
  - Empty string is 0
  - 0 is an even number
- `states: {even, odd}`
- `inputs: {0, 1}`
- Transitions
  - δ`(even, 0) => even`
  - δ`(even, 1) => odd`
  - δ`(odd, 0) => odd`
  - δ`(odd, 1) => even`
- `accepting states: {odd}`
- `starting state: {even}`

Transition Table

	0	1		
even	even	odd	0	Rejecting State
odd	odd	even	1	Accepting State

- Example:
  - `1011`

even (1)	->	odd (0)	->	odd (1)	->	even (1)	->	odd

Transition Diagram

Starting State	->	(even)	-(1)->	((odd))
		-(0)^	<-(1)-	-(0)^

Only useful when debugging sequences

> Design a FSM to check an input sequence of 0's and 1's to verify that the 1's occur in pairs

- String of 0's accepted
- `states: {waiting_pair, not_pair, pair}`
- `inputs: {0, 1}`
- Transitions
  - δ`(pair, 0) -> pair`
  - δ`(pair, 1) -> waiting_pair`
  - δ`(waiting_pair, 0) -> not_pair`
  - δ`(waiting_pair, 1) -> pair`
  - δ`(not_pair, 0) -> not_pair`
  - δ`(not_pair, 1) -> not_pair`
- `accepting states: {pair}`
- `starting state: {pair}`

Transition Table

	0	1		
pair	pair	waiting_pair	0	Rejecting State
waiting_pair	not_pair	pair	1	Accepting State
not_pair	not_pair	not_pair	0	Rejecting State

	0	1	-1
No Ones	No Ones	One One	“Yes”
One One	Error	No Ones	“No”
Error	Error	Error	“No”

Processing Machine

	0	1	-1
No Ones	No Ones	One One	“Yes”
One One	“No”	No Ones	“No”

Behaviour we want for the lexical analyser

> What is the difference between these two FSM | |0|1| | |-|-|-|-| |S|S|S|0|

- Recognises `{}` or $\varnothing$, i.e. nothing

	0	1	
S	T	T	1
T	T	T	0

- Recognises ε, i.e. null/empty string
- Any FSM whose starting state is an accepting state recognises the null string

> Design a FSM to recognise any valid sequence that can follow the keyword Integer in Fortran

```
INTEGER X(5, I, 2), Y
```

	X	(	5	,	I	,	2	)	,	Y
1	2	3	4	5	6	5	4	7	8	2

2. Name of ident to be made integer
3. Left parenthesis
4. Constant specifying a dimension
5. Comma seperating dimensions
6. Variable identifier specifying an adjustable dimension
7. Right parenthesis
8. Comma seperating items to be made integer

- *input alphabtet* = `{V, C, ',', (, )}`
  - V = Variable Identifier
  - C = Constant
  - Two finite state machines
    * One recognising the variable identifiers
    * One recognising the constants
- *states* = `{1, 2, 3, 4, 5, 6, 7, 8, E}`
- *starting state* = `1`
- *accepting states* = `{2, 7}`

Transition Table

	V	C	,	(	)	
1	2					0
2			8 3		1	
3	6	4				0
4			5		7	0
5	6	4				0
6			5		7	0
7			8			1
8	2					0
E						0

Remove extraneous states

- `{1, 2, 8, 3, 6, 4, 5, 7, E}`
- Partition states `{1, 2, 3, 4, 5, 6, 7, 8}`
  - P0: `{2, 7}`, `{1, 3, 4, 5, 6, 8, E}`
  - P1: `{2, 7}`, `{1, 8}`, `{3, 4, 5, 6, E}`

- P2: `{2, 7}`, `{1, 8}`, `{4, 6}`, `{3, 5, E}`
- P3: `{2, 7}`, `{1, 8}`, `{4, 6}`, `{3, 5}` `{E}`
- P4: `{2}`, `{7}`, `{1, 8}`, `{4, 6}`, `{3, 5}` `{E}`

- `{1, 8}` - A
- `{3, 5}` - B
- `{4, 6}` - C

	V	C	,	(	)	
A	2					0
2			A	B		1
B	C	C				0
C			B		7	0
7			A			1
E						0

Use a transliterator to reduce the size of the input alphabet

Source statements -> |Transliterator| -(Character Tokens)> |Lexical Analyser| -(Lexical Tokens)>

- Character Tokens (Class, Value)
    - (digit, ‘7’) (letter, ‘Z’) (sign ‘+’)