Contents

1 Mentor Formal Languages CLI

The command-line interface (CLI) for the formal languages implemented in Mentor can be used for various operations on different types of formal language specifications. This documentation will go over the specifics of what operations can be done on which types of specifications and on the particular syntax used for each type of specification. For the purposes of this documentation we will assume that the CLI executable is named 'mentor'.

1.1 Deterministic Finite Automata (DFA)

The CLI assumes that any file ending in the suffix `.dfa` contains a description of a DFA using the appropriate syntax (described below).

1.1.1 Commands

For DFA, Mentor can take an input string and return whether it is accepted or rejected by the given DFA; output a PDF containing the state diagram of the given DFA; generate the first `<n>` words of the language described by the DFA (in shortlex order); and compare the DFA against a different regular language description (DFA, NFA, or RE) to determine whether they are equivalent or not.

1.1.1.1 Accept/reject input

Command: `mentor <name>.dfa accept <string>`

Example 1: `mentor file.dfa accept abba`
Example 2: `mentor file.dfa accept 'abba#abba'`
Example 3: `mentor file.dfa accept ""`

Note that if the input contains any special characters it may need to be wrapped in quotes. Empty quotes denote an input of the empty string.

1.1.1.2 Output state diagram

Command: `mentor <name>.dfa graph <name2>`

Example: `mentor file.dfa graph out`

The command will produce a PDF named `<name2>.pdf` containing the DFA's state diagram.

1.1.1.3 Generate words

Command: `mentor <name>.dfa gen_words <number>`

Example: `mentor file.dfa gen_words 10`

The command will output a list of words in the DFA's language in shortlex order (i.e., shorter words are sorted before longer words and within the same length words are sorted in lexicographic order).

1.1.1.4 Compare against regular language

Command: `mentor <name>.dfa compare <name2>.{dfa,nfa,re}`

Example 1: `mentor file.dfa compare file2.dfa`
Example 2: `mentor file.dfa compare file2.nfa`
Example 3: `mentor file.dfa compare file2.re`

The command will output a message that the two regular languages are equivalent *or* it will provide counterexamples: a word that is in the first language but not the second and/or a word that is in the second language but not the first.

1.1.2 DFA Syntax

A DFA description must have the following syntax:

```
alphabet: {<symbol1>, <symbol2>, ...}
start: <state>
accepting: {<state1>, <state2>, ...}
<from_state> <transition>...
```

Where a transition can have one of the following forms:

1. `(<symbol> -> <to_state>)`
2. `({<symbol1>, <symbol2>, ...} -> <to_state>)`
3. `(. -> <to_state>)`

Alphabet symbols and states can be named any alphanumeric string. Transition type (1) says that upon reading `<symbol>` transition to `<to_state>`. Transition type (2) says that upon reading any of `<symbol1>`, `<symbol2>`, etc transition to `<to_state>`. Transition type (3) says that upon reading any symbol in the alphabet transition to `<to_state>`. A DFA must have a non-empty alphabet and exactly one transition from each state for each alphabet symbol. The set of accepting states may be empty. DFA descriptions allow for C++-style comments.

1.1.2.1 Example 1

```
// Language is { w in {a,b}* | any a is immediately preceded by b }
alphabet: {a, b}
start: q0
accepting: {q0, q2}
q0 (a -> q1) (b -> q2)
q1 (a -> q1) (b -> q1)
q2 (a -> q0) (b -> q2)
```

1.1.2.2 Example 2

```
// Same language as above, but using set notation for q1's transitions.
alphabet: {a, b}
start: q0
accepting: {q0, q2}
q0 (a -> q1) (b -> q2)
q1 ({a,b} -> q1) // This is a shortcut for specifying two similar transitions.
q2 (a -> q0) (b -> q2)
```

1.1.2.3 Example 3

```
// Same language as above, but using dot notation for q1's transitions.
alphabet: {a, b}
start: q0
accepting: {q0, q2}
q0 (a -> q1) (b -> q2)
q1 (. -> q1) // This is a shortcut specifying the whole alphabet.
q2 (a -> q0) (b -> q2)
```

1.2 Nondeterministic Finite Automata (NFA)

The CLI assumes that any file ending in the suffix `.nfa` contains a description of an NFA using the appropriate syntax (described below).

1.2.1 Commands

The NFA commands are the same as for DFA (see above).

1.2.2 NFA Syntax

An NFA description must have the following syntax:

```
alphabet: {<symbol1>, <symbol2>, ...}
start: <state>
accepting: {<state1>, <state2>, ...}
<from_state> <transition>...
```

Where a transition can have one of the following forms:

1. `(<symbol> -> <to_state>)`
2. `({<symbol1>, <symbol2>, ...} -> <to_state>)`
3. `(. -> <to_state>)`
4. `(_ -> <to_state>)`

Alphabet symbols and states can be named any alphanumeric string. Transition type (1) says that upon reading `<symbol>` transition to `<to_state>`. Transition type (2) says that upon reading any of `<symbol1>`, `<symbol2>`, etc transition to `<to_state>`. Transition type (3) says that upon reading any symbol in the alphabet transition to `<to_state>`. Transition type (4) says to nondeterministically transition to `<to_state>` (the `_` symbol, which is not allowed in the alphabet, stands for the empty string). The `_` symbol can also be used in the set notation, e.g., `({a,_} -> q1)` says that either when reading an `a` or nondeterministically without reading anything transition to state `q1`. An NFA must have a non-empty alphabet. The set of accepting states may be empty. NFA descriptions allow for C++-style comments.

1.2.2.1 Example

```
// Any number of 'a' followed by any number of 'b'.
alphabet: {a, b}
start: q0
accepting: {q3}
q0 (_ -> q1) (_ -> q2) (_ -> q3)
q1 (a -> q1) ({a,_} -> q2)
q2 (_ -> q3)
q3 (b -> q3)
```

1.3 Regular Expressions (RE)

The CLI assumes that any file ending in the suffix `.re` contains a description of a regular expression using the appropriate syntax (described below).

1.3.1 Commands

The regular expression commands are the same as for DFA (see above).

1.3.2 RE Syntax

A regular expression description must have the following syntax:

```
alphabet: { <char1>, <char2>, ... }
<regular expression as described below>
```

Base regular expressions:

- `<char>` : a single character (not allowed to be any of `|&-*+?()._`)
- `_` : stands for the empty string
- `.` : stands for any character in the alphabet

Compound regular expressions, where `re`, `re1`, and `re2` are regular expressions:

- `re1re2` : concatenation of `re1` and `re2`
- `re1 | re2` : union of `re1` and `re2`
- `re1 & re2` : intersection of `re1` and `re2`
- `re*` : kleene star (zero or more repetitions of `re`)
- `re+` : one or more repetitions of `re` (same as `rere*`)
- `re?` : zero or one repetitions of `re` (same as `_ | re`)
- `-re` : the complement of `re` (all strings not in `re`)
- `(re)` : grouping with parentheses

C++-style comments are allowed.

1.3.2.1 Example

```
// Not a meaningful language, just showing all the operators.
alphabet: {a, b}
(a(_ | b) & (..)*) | ((a.a)+ & -(a?b*))
```

1.4 Context-Free Grammars (CFG)

The CLI assumes that any file ending in the suffix `.cfg` contains a description of a CFG using the appropriate syntax (described below).

1.4.1 Commands

For CFG, Mentor can take an input string and return whether it is accepted or rejected by the given CFG; generate the first `<n>` words of the language described by the CFG (in shortlex order); and generate all words up to length `<n>` (also in shortlex order).

1.4.1.1 Accept/reject input

Command: `mentor <name>.cfg accept <string>`

1.4.1.2 Generate words

Command: `mentor <name>.cfg gen_words <number>`

1.4.1.3 Generate words up to some length

Command: `mentor <name>.cfg gen_upto_length <number>`

1.4.2 CFG Syntax

A CFG description must have the following syntax:

```
<nonterminal> -> <rhs1> | <rhs2> | ...
```

Where `<rhs>` is a sequence of `<nonterminal>` and `<char>` *or* is `_` (representing the empty string). A `<nonterminal>` can be any alphanumeric string, `<char>` can be any character except `_` or `|`, and any `<nonterminal>`s in the right-hand side must be surrounded by spaces. The first `<nonterminal>` listed is the starting symbol for the CFG. C++-style comments are allowed.

1.4.2.1 Example

```
Start -> X | Y | Start Start
X -> ac X ca | cb X | abc | _
Y -> c Y | a | _
```

1.5 Pushdown Automata (PDA)

The CLI assumes that any file ending in the suffix `.pda` contains a description of a PDA using the appropriate syntax (described below). Note that the style of PDA implemented by Mentor accepts inputs using accepting states, regardless of the contents of the stack. A simple syntactic translation can convert a PDA that accepts inputs on an empty stack into one that accepts inputs using accepting states instead (and vice-versa).

1.5.1 Commands

The PDA commands are the same as for CFG (see above) with the addition of the `graph` command to output the state diagram of the PDA as a PDF file.

1.5.1.1 Output state diagram

Command: `mentor <name>.pda graph <name2>`

1.5.2 PDA Syntax

A PDA description must have the following syntax:

```
alphabet: {<symbol1>, <symbol2>, ...}
start: <state>
accepting: {<state1>, <state2>, ...}
<from_state> <transition>...
```

Where a transition can have one of the following forms:

1. `(<input>, <pop> -> <push>, <to_state>)`
2. `({<input1>, <input2>, ...}, <pop> -> <push>, <to_state>)`
3. `(., <pop> -> <push>, <to_state>)`

The alphabet specifies the **input** alphabet, *not* the stack alphabet. A `<symbol>` can be any character except one of `{}().,_`. A `<state>` can be any alphanumeric string. An `<input>` can be any input symbol *or* `_` (representing the empty string). A `<pop>` can be any stack symbol *or* `_`. A `<push>` can be any **string** of stack symbols, i.e., a PDA can push more than one symbol on the stack at once; a string is pushed s.t. the leftmost-symbol ends on the top of the stack. For `<push>` an empty string is represented as `""`. The set notation can include `_` as one of the elements; the `.` notation only includes input symbols and *not* `_`. C++-style comments are allowed.

1.5.2.1 Example 1

```
alphabet: {0, 1}
start: q0
accepting: {q0, q3}
q0 (_, _ -> $, q1) // Notice that $ is a stack symbol and not in the alphabet
q1 (0, _ -> 0, q1) (1, 0 -> _, q2)
q2 (1, 0 -> _, q2) (_, $ -> _, q3)
```

1.5.2.2 Example 2

```
// An example pushing multiple symbols onto the stack at once.
alphabet: {+, [, ], a}
start: Q0
accepting: {Q2}
Q0 (_, _ -> e$, Q1)
Q1 (_, e -> f+e, Q1)
Q1 (_, e -> f, Q1)
Q1 (_, f -> [e], Q1)
Q1 (_, f -> a, Q1)
Q1 (+, + -> _, Q1)
Q1 ([, [ -> _, Q1)
Q1 (], ] -> _, Q1)
Q1 (a, a -> _, Q1)
Q1 (_, $ -> _, Q2)
```

1.5.2.3 Example 3

```
// An example using dot notation.
alphabet: {0, 1}
start: q0
accepting: {q0, q3}
q0 (., _ -> $, q1)
q1 (0, _ -> 0, q1) (1, 0 -> _, q2)
q2 (1, 0 -> _, q2) (_, $ -> _, q3)
```

1.5.2.4 Example 4

```
// An example using set notation.
alphabet: {0, 1}
start: q0
accepting: {q0, q3}
q0 ({0,1}, _ -> $, q1)
q1 (0, _ -> 0, q1) (1, 0 -> _, q2)
q2 (1, 0 -> _, q2) (_, $ -> _, q3)
```

1.6 Turing Machines (TM)

The style of TM implemented by Mentor has a leftmost tape position and the tape extends infinitely to the right, with the tape head starting in the leftmost position (at the beginning of the input). Attempting to move the tape head left from the leftmost position leaves it in place. A TM transition reads the current tape cell, writes to the current tape cell, and moves the tape head in one of four ways: left one cell, right one cell, reset to the leftmost position, or stay in place. (An interesting assignment can be to remove one of these tape head movements and show that the TM can still decide the same set of languages.)

The CLI assumes that any file ending in the suffix `.tm` contains a description of a TM using the appropriate syntax (described below).

1.6.1 Commands

For TM, Mentor can take an input string and return whether it is accepted or rejected by the given TM; and also return a trace of the computation on an input by the given TM. In both cases we can restrict the number of computation steps the TM is allowed to take in order to prevent looping forever. A computation step consists of taking a single TM transition.

1.6.1.1 Accept/reject (or timeout on) input

Command: `mentor <name>.tm accept <string> [<number>]`

Example 1: `mentor file.tm accept abba`
Example 2: `mentor file.tm accept abba 1000`

1.6.1.2 Trace computation on input

Command: `mentor <name>.tm trace <string> [<number>]`

Example 1: `mentor file.tm trace abba`
Example 2: `mentor file.tm trace abba 1000`

The command will output a sequence of TM configurations of the form `<tape contents to the left of the tape head>|<current state>|<tape contents under and to the right of the tape head>`.

1.6.2 TM Syntax

A TM description must have the following syntax:

```
alphabet: {<symbol1>, <symbol2>, ...}
start: <state>
<from_state> <transition>...
```

Where a transition can have one of the following forms:

1. `(<symbol> -> <symbol>, {L,R,S,H} <to_state>)`
2. `({<symbol1>, <symbol2>, ...} -> <symbol>, {L,R,S,H} <to_state>)`
3. `({<symbol1>, <symbol2>, ...} -> ., {L,R,S,H} <to_state>)`
4. `(. -> <symbol>, {L,R,S,H} <to_state>)`
5. `(. -> ., {L,R,S,H} <to_state>)`

An alphabet `<symbol>` can be any character except for `{}().`, and `_` is reserved to represent a blank cell on the tape (*not* an empty string as for regular and context-free languages). A `<state>` can be any alphanumeric string; state names `accept` and `reject` always correspond to the accepting and rejecting states (and are not allowed to have any transitions coming from them). The tape directions represent:

- L: move left one cell
- R: move right one cell
- S: reset to the start of the tape (leftmost position)
- H: stay here (don't move the tape head)

A `.` on the left of an arrow represents any character of the tape alphabet, including blank. If a set of symbols or a `.` is used on the left of an arrow, then a `.` on the right of the arrow means whatever symbol was actually read. Thus, the following transitions are all equivalent if the tape alphabet is {a,b,_}:

- `q1 (. -> .,R q1)`
- `q1 ({a,b,_} -> .,R q1)`
- `q1 (a -> a,R q1) (b -> b,R q1) (_ -> _,R q1)`

A TM must be deterministic, but the syntax allows for implicit rejection: any missing transitions are assumed to go to the `reject` state. C++-style comments are allowed.

1.6.2.1 Example 1

```
// Decides L = { 0{n}1{n}2{n} | n >= 0 }
alphabet: {0, 1, 2, x}
start: q0
q0 (0 -> _,R q1) (x -> x,R q4) (_ -> _,L accept)
q1 (0 -> 0,R q1) (x -> x,R q1) (1 -> x,R q2)
q2 (x -> x,R q2) (1 -> 1,R q2) (2 -> x,L q3)
q3 (x -> x,L q3) (0 -> 0,L q3) (1 -> 1,L q3) (_ -> _,R q0)
q4 (x -> x,R q4) (_ -> _,L accept)
```

1.6.2.2 Example 2

```
// Same except uses sets and dot notation.
alphabet: {0, 1, 2, x}
start: q0
q0 (0 -> _,R q1) (x -> x,R q4) (_ -> _,L accept)
q1 ({0,x} -> .,R q1) (1 -> x,R q2)
q2 ({1,x} -> .,R q2) (2 -> x,L q3)
q3 ({0,1,x} -> .,L q3) (_ -> _,R q0)
q4 (x -> x,R q4) (_ -> _,L accept)
```

1.7 Hierarchical Turing Machines (HTM)

The Hierarchical Turing Machine (HTM) language was created specifically for Mentor. It is a high-level (compared to TM) programming language with sequencing, conditionals, and loops that compiles directly to TM and is intended to replace pseudo-code descriptions of TM algorithms with something that is executable and testable. See below for more information about how TM can be easily and simply composed into higher-level programming abstractions.

The CLI assumes that any file ending in the suffix `.htm` contains a description of an HTM using the appropriate syntax (described below).

1.7.1 Composing TM into HTM

The basic idea behind HTM is to compose smaller TMs into larger TMs. There are three possible compositions: sequencing, conditionals, and loops. These can be put together to create functions, which can be composed using function calls. We make a basic assumption that the states for all the constituent TMs are disjoint.

1.7.1.1 Sequencing

Given two Turing machines `TM1` and `TM2`, we can create a third Turing machine `TM3` that is the result of carrying out first `TM1` and then `TM2`. We create `TM3` by doing the following:

1. Union the states and transitions of `TM1` and `TM2`.
2. Make the start state of `TM3` the same as the start state of `TM1`.
3. Merge the `accept` state of `TM1` with the start state of `TM2`.

The result is that `TM3` will go through the same transitions as `TM1`, but where `TM1` would *accept* instead `TM3` goes through the same transitions as `TM2` (starting with the tape in whatever state was left from `TM1`).

1.7.1.2 Conditionals

Given three Turing machines `TM1`, `TM2`, and `TM3`, we want to create a Turing machine `TM4` that treats `TM1` as a guard to determine whether to execute `TM2` or `TM3`. We create `TM4` by doing the following:

1. Union the states and transitions of `TM1`, `TM2`, and `TM3`.
2. Make the start state of `TM4` the same as the start state of `TM1`.
3. Merge the `accept` state of `TM1` with the start state of `TM2`.
4. Merge the `reject` state of `TM1` with the start state of `TM3`.

The result is that `TM4` will go through the same transitions as `TM1`, but instead of accepting or rejecting it will then go through the same transitions as `TM2` (if `TM1` would have accepted) or `TM3` (if `TM1` would have rejected).

1.7.1.3 Loops

Given three Turing machines `TM1`, `TM2`, and `TM3`, we want to create a Turing machine `TM4` that treats `TM1` as a guard to determine whether to execute `TM2` (the body of the loop) and then re-evaluate `TM1`, or exit the loop to execute `TM3`. We create `TM4` by doing the following:

1. Union the states and transitions of `TM1`, `TM2`, and `TM3`.
2. Make the start state of `TM4` the same as the start state of `TM1`.
3. Merge the `accept` state of `TM1` with the start state of `TM2`.
4. Merge the `accept` state of `TM2` with the start state of `TM1`.
5. Merge the `reject` state of `TM1` with the start state of `TM3`.

The result is that `TM4` will go through the same transitions as `TM1`, but instead of accepting or rejecting it will go through the same transitions as `TM2` (if `TM1` would have accepted) or `TM3` (if `TM1` would have rejected). Then if `TM2` would accept, `TM4` instead re-evaluates the guard (i.e., `TM1`) and continues to iterate `TM2` as long as `TM1` would continue to accept, before finally exiting the loop when `TM1` would reject.

1.7.1.4 Functions

It would be convenient to be able to re-use TM definitions rather than redefine the same TM over and over again, and also to be able to parameterize a TM definition by the data it operates on. This is exactly what functions are for. HTM has two kinds of function definitions: leaf functions (that define standard TMs) and non-leaf functions (that use sequencing, conditionals, loops, and function calls to combine leaf TM into larger TM).

A leaf function can be parameterized by what alphabet symbols it uses, i.e., a leaf function can have parameters that are used inside the function, and the values

of those parameters will be sets of alphabet symbols. Besides those parameters, a leaf function is just an ordinary TM.

A non-leaf function is defined using a sequence of statements, where each statement is either a function call, a conditional, or a loop. A non-leaf function is translated into a TM by using the transformations described above (where a function call simply copies the TM of the function being called, which itself is either a leaf function or a non-leaf function).

See the syntax and examples below for what an HTM program ends up looking like. The key is that an HTM program can easily be translated into one large TM, which operates just like any other TM.

1.7.2 Commands

The HTM commands are the same as for TM (see above), though for convenience and readability computation steps are counted differently than for TM (they are based on the number of leaf TM functions called rather than TM transitions) and the traces show the sequence of function calls rather than TM configurations.

1.7.3 HTM Syntax

An HTM program consists of an alphabet specification and a set of function definitions, including a mandatory function named `Main` that is the program entry point. There are two kinds of function definitions: a leaf function consisting of a TM description, and a non-leaf function that consists of a sequence of instructions including function calls, conditionals, and loops. The grammar for HTM programs is:

```
Program -> alphabet: {<char1>, <char2>, ...} Function+
Function -> def <name> [[ TmBody ]]
          | def <name>(<param1>, <param2>, ...) [[ TmBody ]]
          | def <name> { Statement+ }
          | def <name>(<param1>, <param2>, ...) { Statement+ }
TmBody -> <TM syntax, except may use <param> for tape read/write symbols and
           don't specify the alphabet>
Statement -> accept | reject | Call | Condition | Loop
Call -> <name> | <name>(Arg1, Arg2, ...)
Arg -> $<param> | <char> | {<char1>, <char2>, ...} | -{<char1>, <char2>, ...}
Block -> Statement | { Statement+ }
Condition -> if [not] Call Block | if [not] Call Block else Block
Loop -> while [not] Call Block
```

The alphabet always includes the blank symbol `_`, though it isn't explicitly specified. Non-leaf function bodies always have an implicit `reject` at the end, i.e., if the function doesn't explicitly accept before the end of its execution then it rejects. Function names and parameters can be any alphanumeric string. When using a parameter inside a function, its name must be prefixed with `$`. Leaf

functions don't specify an alphabet for the TM because they inherit the alphabet of the overall HTM program.

A sequence of statements terminates as soon as any statement in the sequence rejects, and the overall function then rejects. In other words, if a statement accepts then that means to continue to the next statement and if a statement rejects then that means the entire function rejects. The only exception is the explicit `accept` statement, which means that the entire function should accept.

A function call argument can be an alphabet symbol or set of such symbols, the complement of a set of alphabet symbols, or a function parameter; note that the value of a function parameter will always end up being a (possibly singleton) set of alphabet symbols. Condition and loop guards are always function calls, which optionally can be negated. C++-style comments are allowed.

Note that recursion in an HTM program would translate into an infinite-size TM, which is invalid. Thus, recursive HTM programs are not allowed.

1.7.3.1 Example 1

```
// L = { a{n}b{n}c{n} | n >= 0 }
alphabet: {a,b,c,x}

// Reset tape head to leftmost position.
def StartOfTape [[
  start: q0
  q0 (. -> .,S accept)
]]

// Move one cell right, without writing anything.
def Right [[
  start: q0
  q0 (. -> .,R accept)
]]

// Accept if the current tape cell belongs in A, otherwise reject.
def Read(A) [[
  start: q0
  q0 ($A -> .,H accept)
]]

// Write A to the current tape cell (miscompiles if A is a non-singleton set).
def Write(A) [[
  start: q0
  q0 (. -> $A,H accept)
]]

// Move right until we read a symbol in A.
def FindRight(A) {
  while not Read($A) Right
```

```
  accept
}

// Verify whether the input is in the correct format: "_ | a+b+c+".
def CorrectFormat {
  if Read(_) accept
  if not Read(a) reject
  FindRight(-{a})
  if not Read(b) reject
  FindRight(-{b})
  if not Read(c) reject
  FindRight(-{c})
  if Read(_) accept
}

// Verify that the input is in the correct format, then iterate marking off
// matching a, b, and c symbols on the tape. If they all match then accept,
// otherwise reject.
def Main {
  CorrectFormat
  StartOfTape
  while not Read(_) {
    Read(a)
    Write(x)
    FindRight({b,_})
    if Read(_) reject
    Write(x)
    FindRight({c,_})
    if Read(_) reject
    Write(x)
    StartOfTape
    FindRight(-{x})
  }
  accept
}
```

1.7.3.2 Example 2

```
// L = { wcw | w in {a,b}* }
alphabet: {a,b,c,x}

// Reset tape head to leftmost position.
def StartOfTape [[
  start: q0
  q0 (. -> .,S accept)
]]

// Move one cell left, without writing anything.
def Left [[
  start: q0
  q0 (. -> .,L accept)
]]

// Move one cell right, without writing anything.
def Right [[
  start: q0
  q0 (. -> .,R accept)
]]

// Accept if the current tape cell belongs in A, otherwise reject.
def Read(A) [[
  start: q0
  q0 ($A -> .,H accept)
]]

// Write A to the current tape cell (miscompiles if A is a non-singleton set).
def Write(A) [[
  start: q0
  q0 (. -> $A,H accept)
]]

// Move left until we read a symbol in A.
def FindLeft(A) {
  while not Read($A) Left
  accept
}

// Move right until we read a symbol in A.
def FindRight(A) {
  while not Read($A) Right
  accept
}

// Mark off the current tape cell holding a symbol in A, then find the next
// unmarked symbol after the 'c', verify that it is the same symbol as before,
// and mark it off as well.
```

```
def Match(A) {
  if not Read($A) reject
  Write(x)
  FindRight(c)
  FindRight({a,b,_})
  if Read($A) {
    Write(x)
    accept
  }
}

// Verify whether the input is in the correct format: "{a,b}*c{a,b}*", then
// reset to start of tape.
def CorrectFormat {
  FindRight(-{a,b})
  Read(c)
  Right
  FindRight(-{a,b})
  if Read(_) {
    StartOfTape
    accept
  }
}

// Verify that the input is in the correct format, then iterate marking off
// matching {a,b} symbols on the tape. If they all match then accept,
// otherwise reject.
def Main {
  CorrectFormat
  while not Read(c) {
    if Read(a) Match(a) else Match(b)
    FindLeft(c)
    FindLeft(x)
    Right
  }
  Right
  FindRight(-{x})
  if Read(_) accept
}
```