the language machine

a model of language

languagemachine.sourceforge.net

what is the language machine?

- a model of language
- a way of thinking about language
- free software licensed under Gnu GPL
- software that has a long history
- software that exists now
- software that is directly applicable now
- a project for language and software

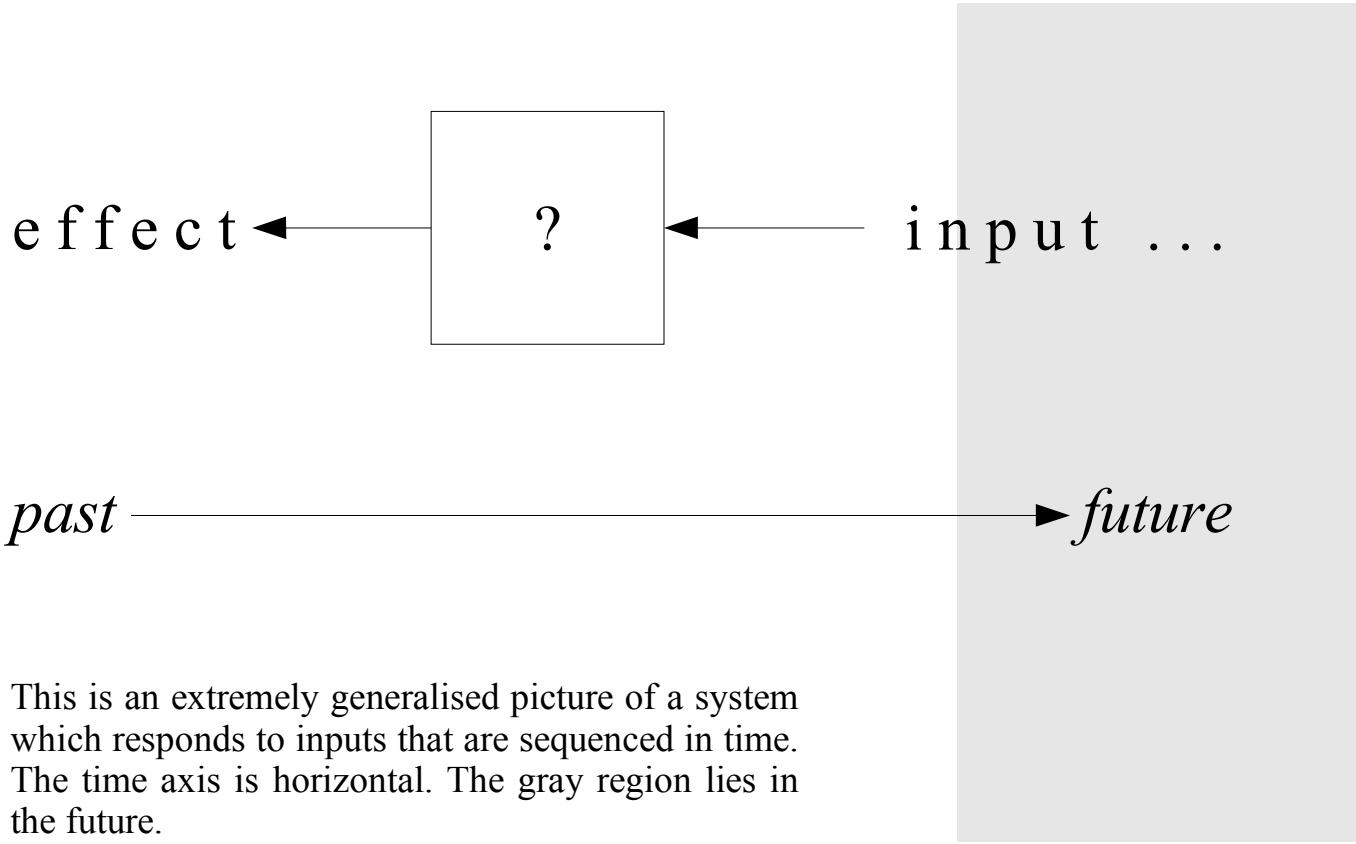
what is this all about?

- how does language work?
- what is the simplest possible system that
  - can deal with language in general?
  - is practical and usable over a wide spectrum?
  - can be made reasonably efficient?
- where do we start?

consider a system ...

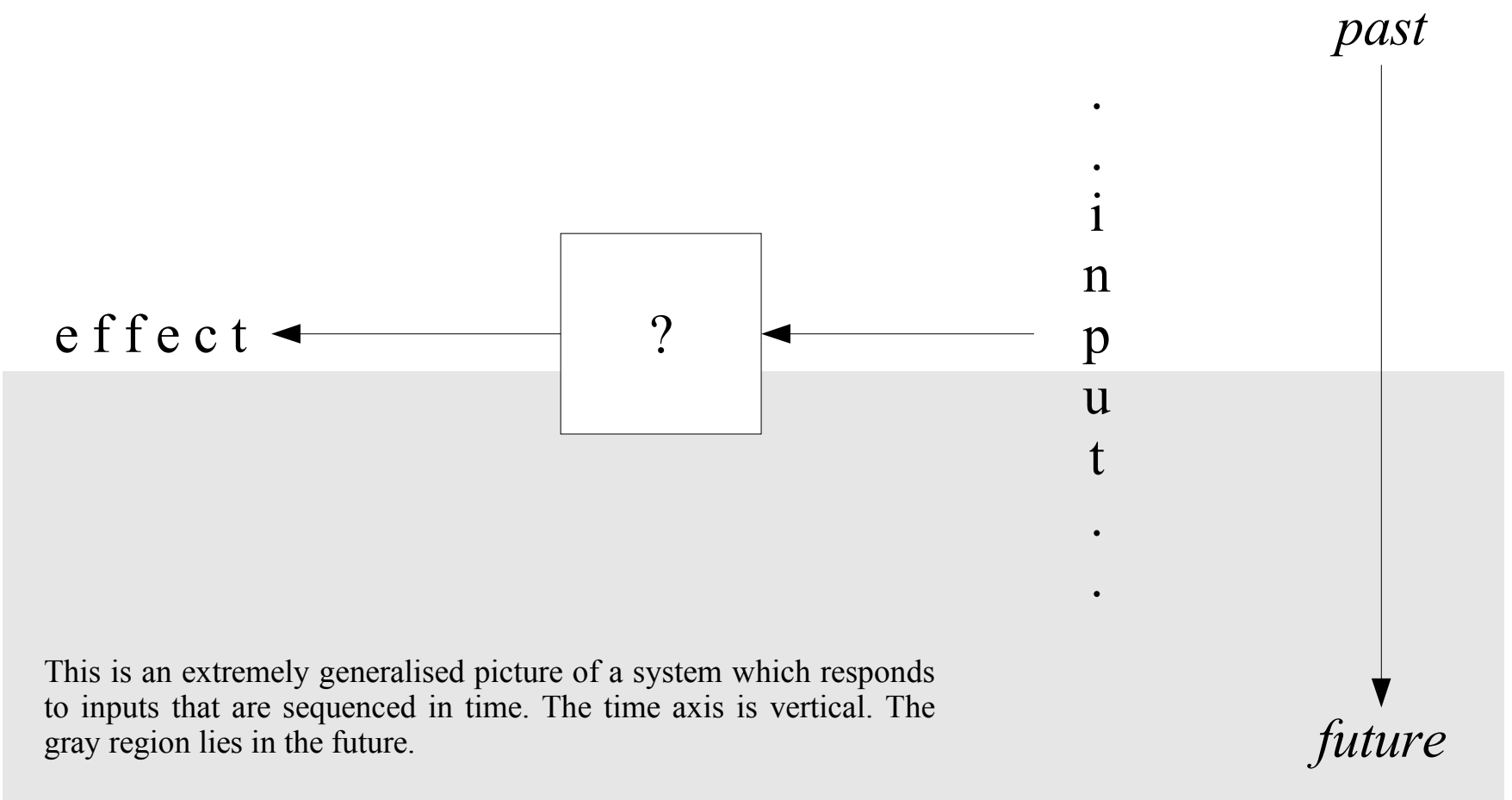
- it responds to inputs
- the inputs are sequenced in time
- the inputs have structure
- what does it look like?

a system with input: horizontal layout



This is an extremely generalised picture of a system which responds to inputs that are sequenced in time. The time axis is horizontal. The gray region lies in the future.

a system with input: vertical layout



This is an extremely generalised picture of a system which responds to inputs that are sequenced in time. The time axis is vertical. The gray region lies in the future.

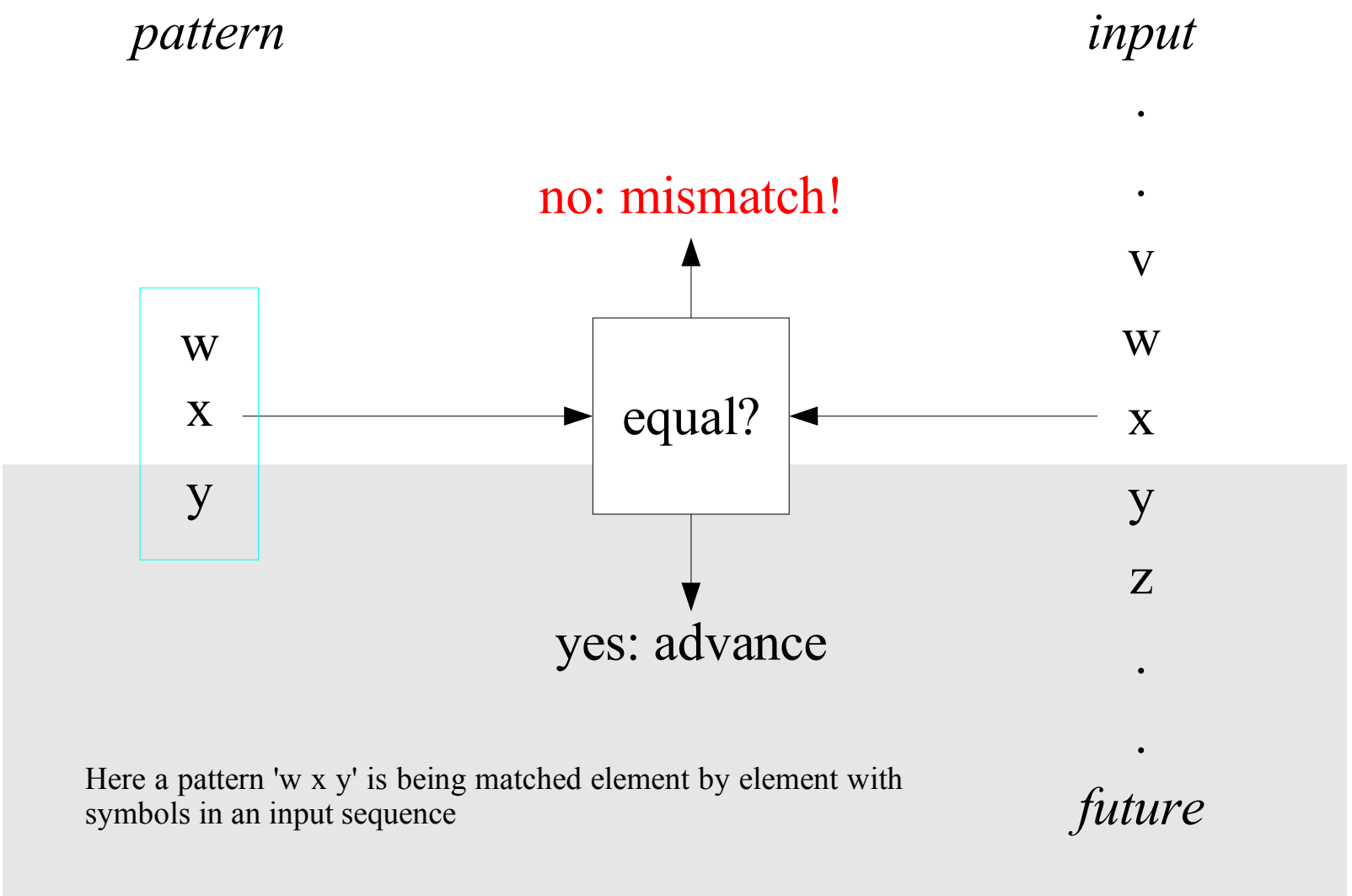
inputs, effects and symbols

- symbols for inputs and external effects
- external effect: no impact on system
- a symbol is merely a token or sign
- no intrinsic meaning
- in computer: values
- in physical model: coins, marbles, beads

recognising patterns

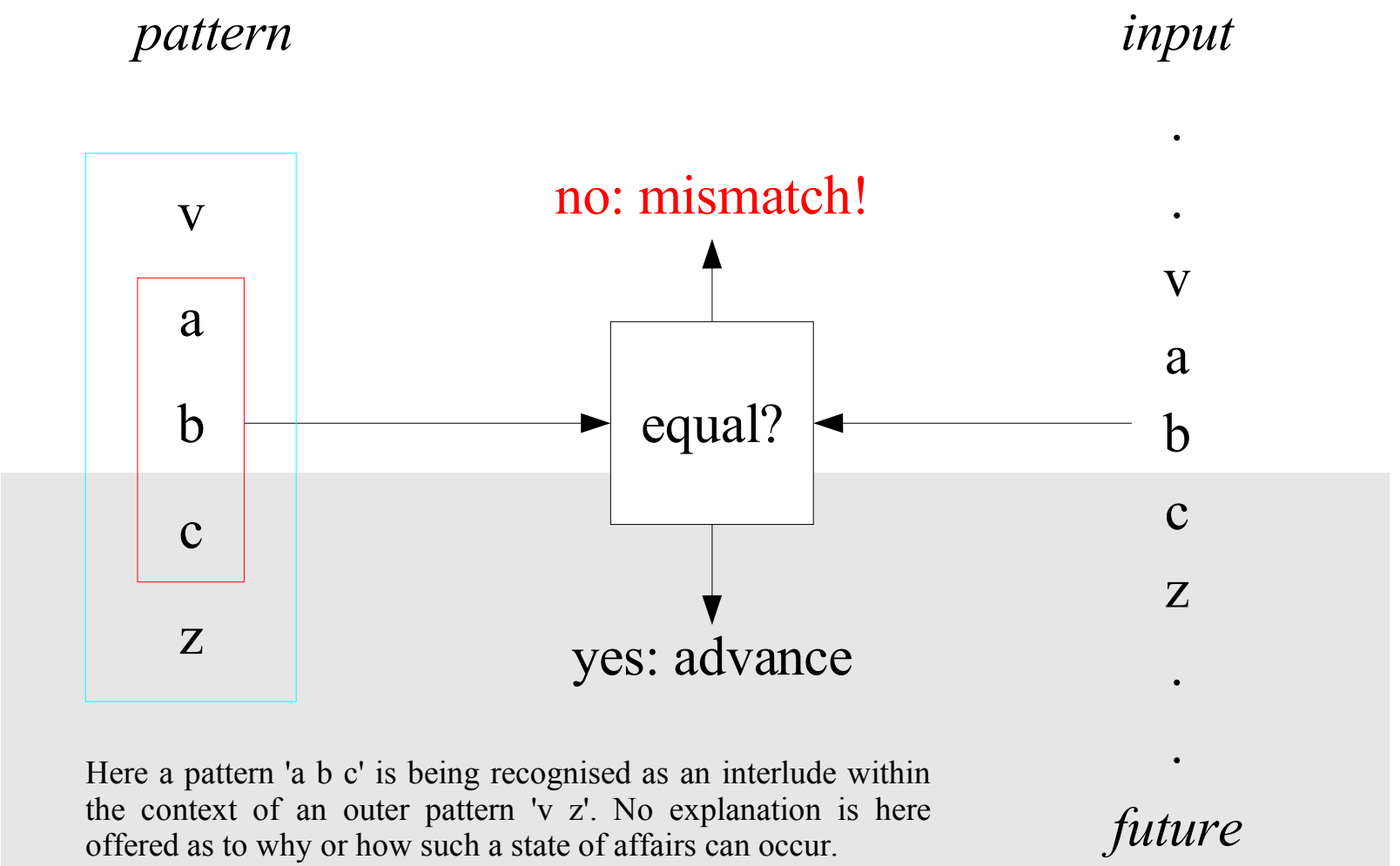
- map different sequences to different responses
- the fundamental activity
- input can have structure
- structure involves nesting
- analysis involves substitution: treat X as Y
- what actually happens?

recognition: a simple sequence



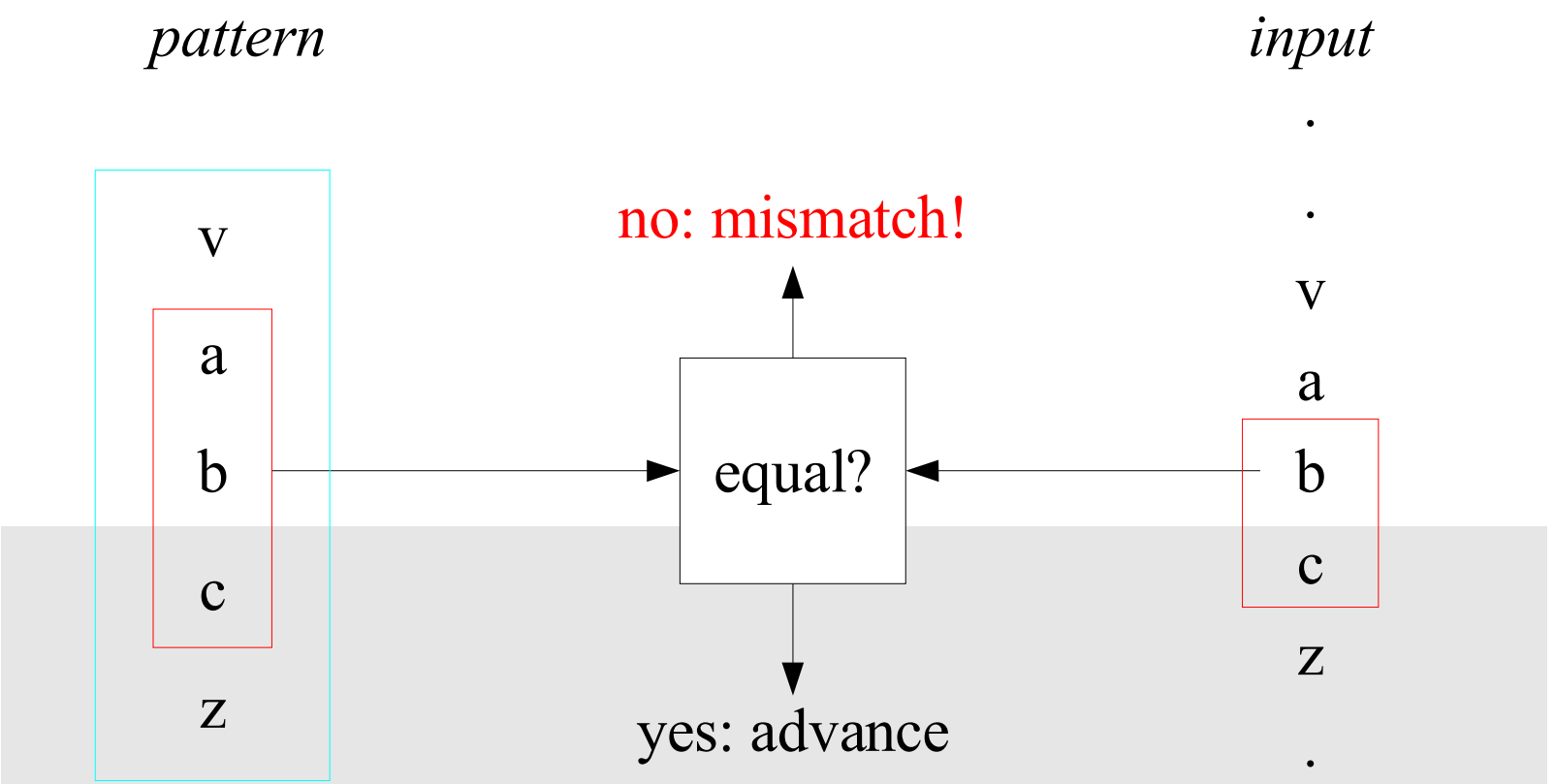
Here a pattern 'w x y' is being matched element by element with symbols in an input sequence

recognition: nested patterns



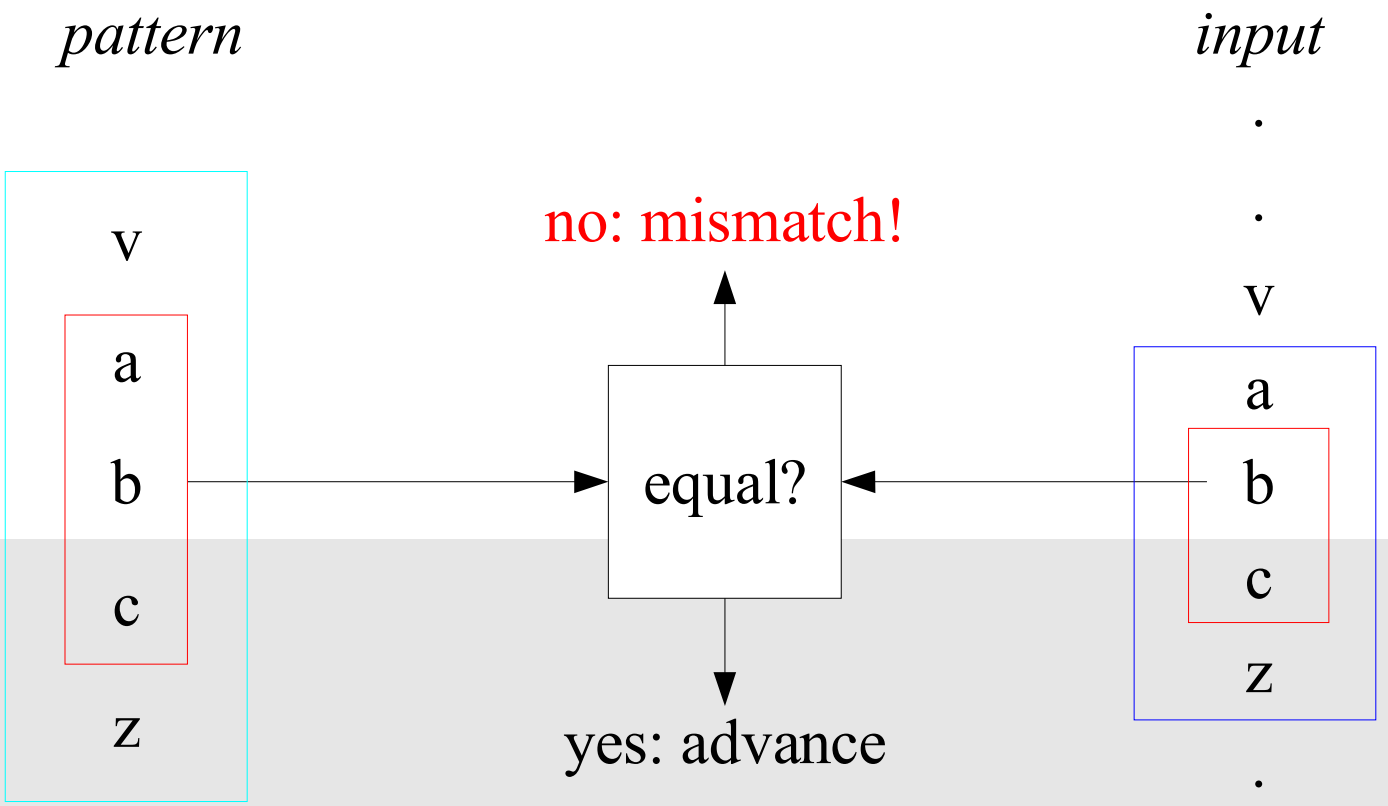
Here a pattern 'a b c' is being recognised as an interlude within the context of an outer pattern 'v z'. No explanation is here offered as to why or how such a state of affairs can occur.

injected input



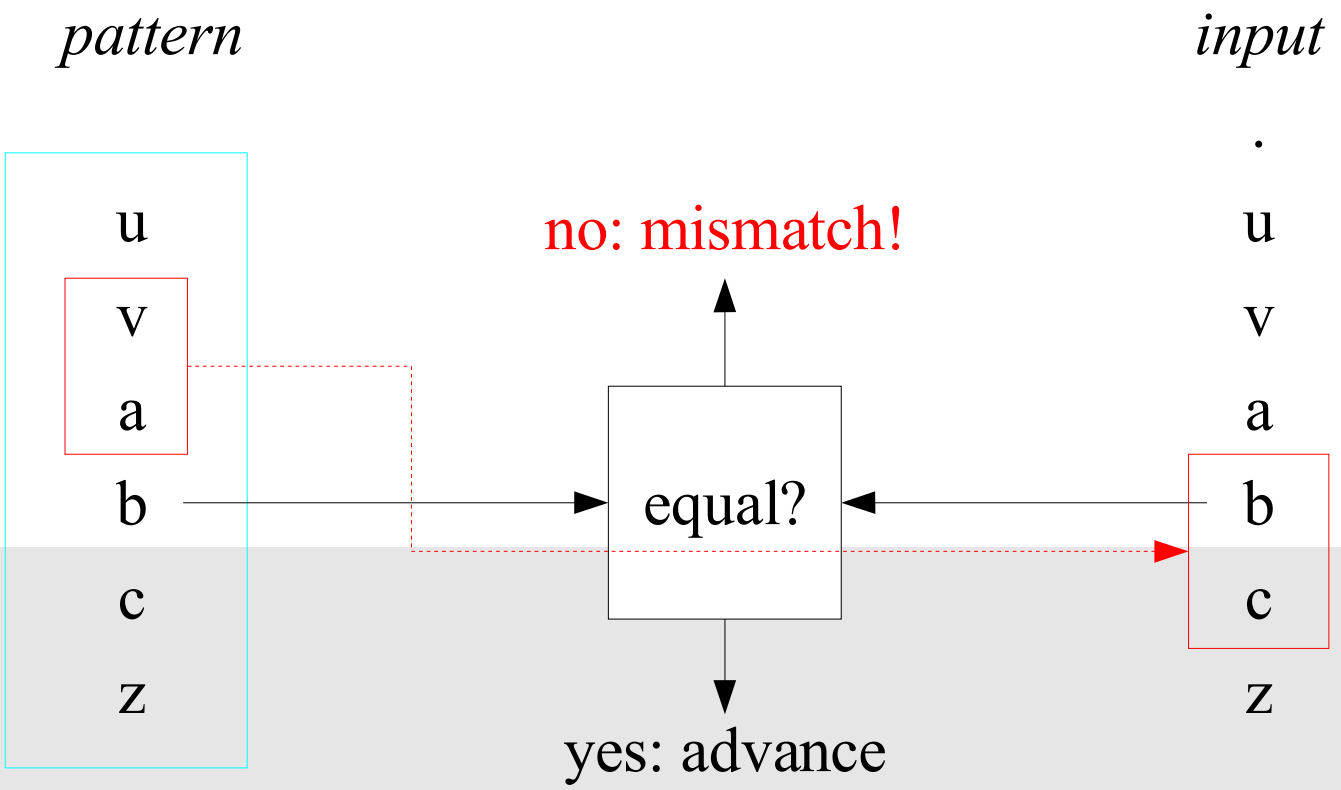
The sequence 'b c' has been inserted or injected and is treated as if it had occurred in the original or external input. Again, no explanation is here offered as to why or how such a state of affairs can occur.

nested input



The sequence 'b c' has been inserted or injected and is treated as if it had occurred before 'z' in the sequence 'a z' which itself has been inserted or injected. Again, no explanation is here offered as to why or how such a state of affairs can occur.

recognition and substitution

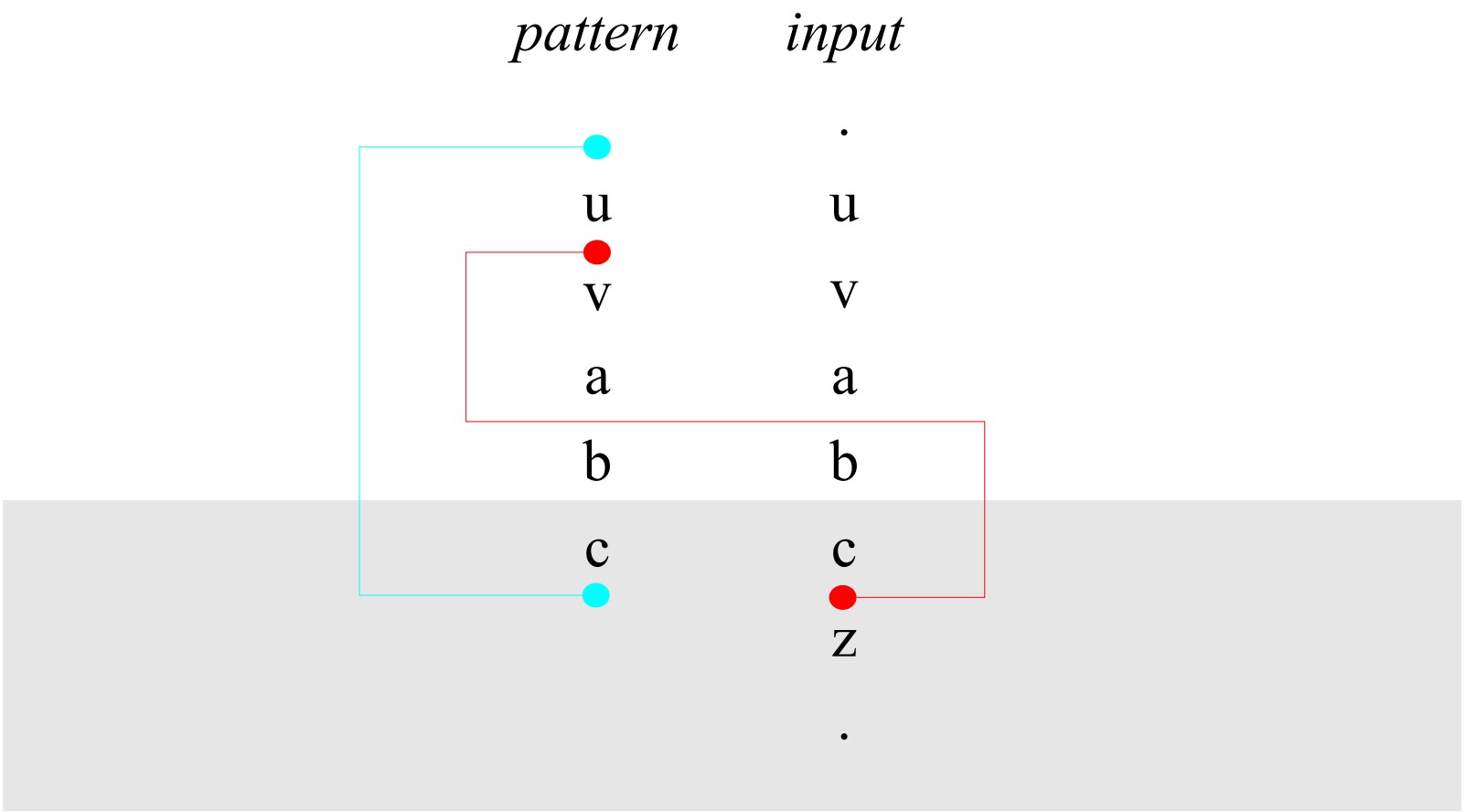


A substitution is being performed: the pattern 'v a' is being replaced by or treated as the sequence 'b c'. The gray region lies in the future. The replacement is an insertion. The substitution occurs when each symbol in the pattern has been recognised.

the lm-diagram

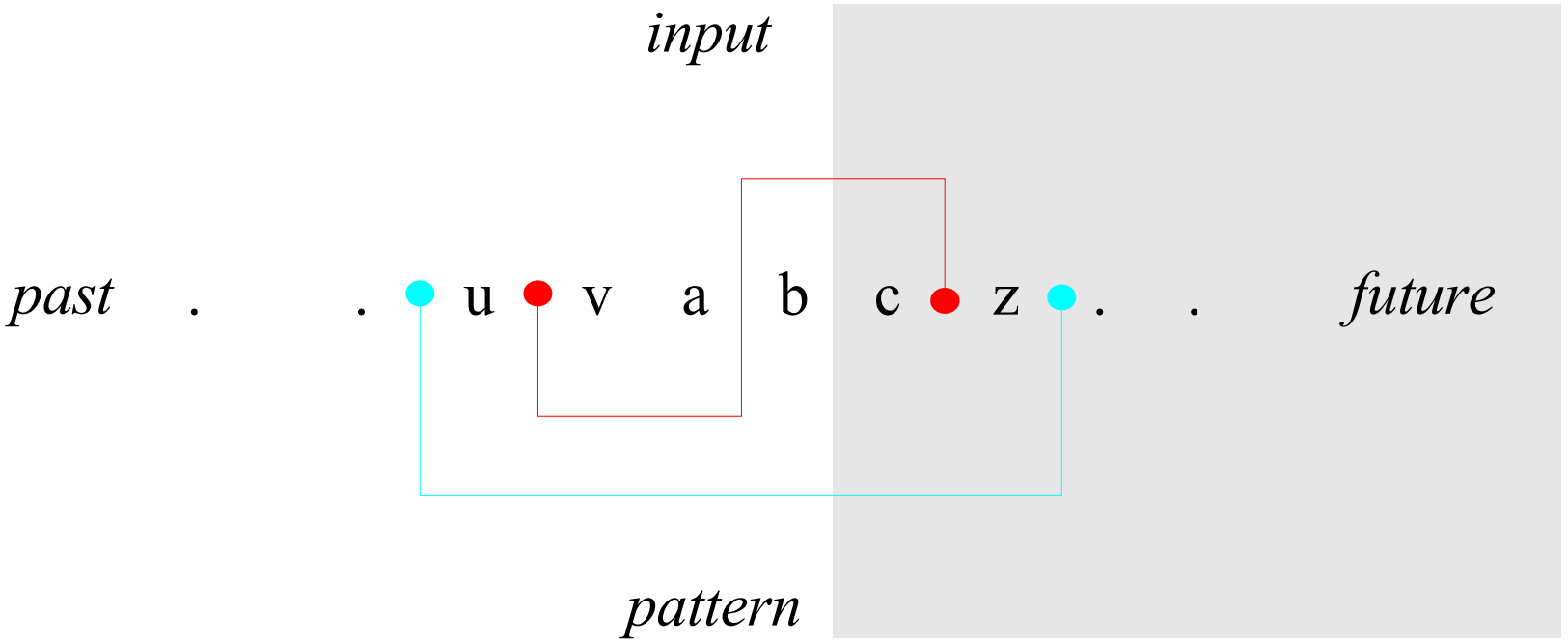
- devised by Peri Hankey during late 1970s
- how to show sequence, symbols and structure?
- vertical form: pattern and input in columns
- unified horizontal form: line of matched symbols
- overlapping motif to show structure
- vertical form better for diagnostics
- clarifies every aspect of grammatical analysis

the lm-diagram: vertical layout



A substitution is being performed: the pattern 'v a' is being replaced by or treated as the sequence 'b c'. The replacement is an insertion. The substitution occurs when each symbol in the pattern has been recognised. The gray region lies in the future. The outer pattern 'u b c' has not yet been completely matched, so its replacement is not shown.

the lm-diagram: horizontal layout

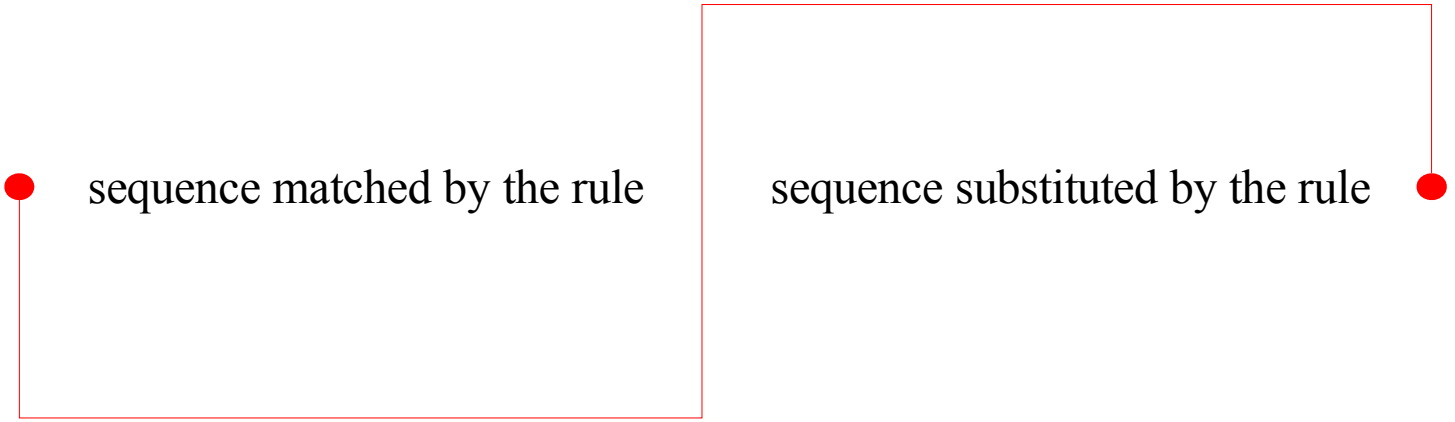


A substitution is being performed: the pattern 'v a' is being replaced by or treated as the sequence 'b c'. The replacement is an insertion. The substitution occurs when each symbol in the pattern has been recognised. The gray region lies in the future. The outer pattern 'u b c z' has not yet been completely matched, so its replacement is not shown. Symbols that are matched are written along the line. Pattern nesting is shown below the line, input substitution nesting is shown above the line. Any symbol that is not contained within some level of substitution nesting can be identified as being part of the original or external input.

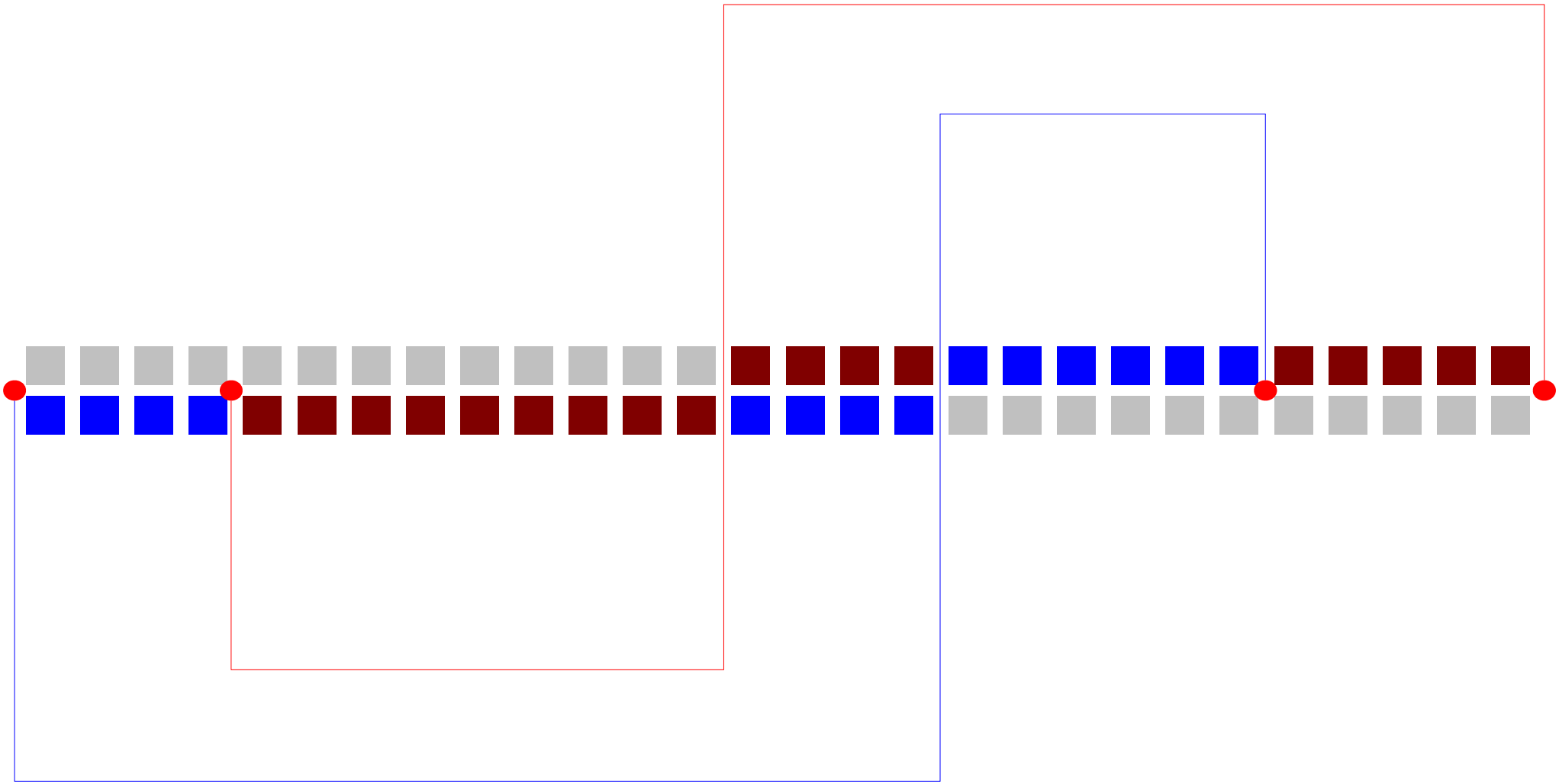
rewriting rules with nesting

- each rule defines a pattern and a substitution
- rules are applied in two phases
- recognition phases may be nested in others
- substitution phases may be nested in others

lm-diagram: rule applications



nested rule applications



two kinds of symbol

- terminal symbols – occur in the input
- non-terminal symbols – occur only in rules
- non-terminal symbols – generalisation
- non-terminal symbols – logic
- nasty jargon, but useful
- just symbols

the mismatch event

- characterised by two symbols
- goal symbol and input symbol
- goal: current symbol, current pattern
- input: current symbol, current input
- must match to advance
- key event: find rule to resolve mismatch
- may consume input symbols
- must eventually provide goal symbol

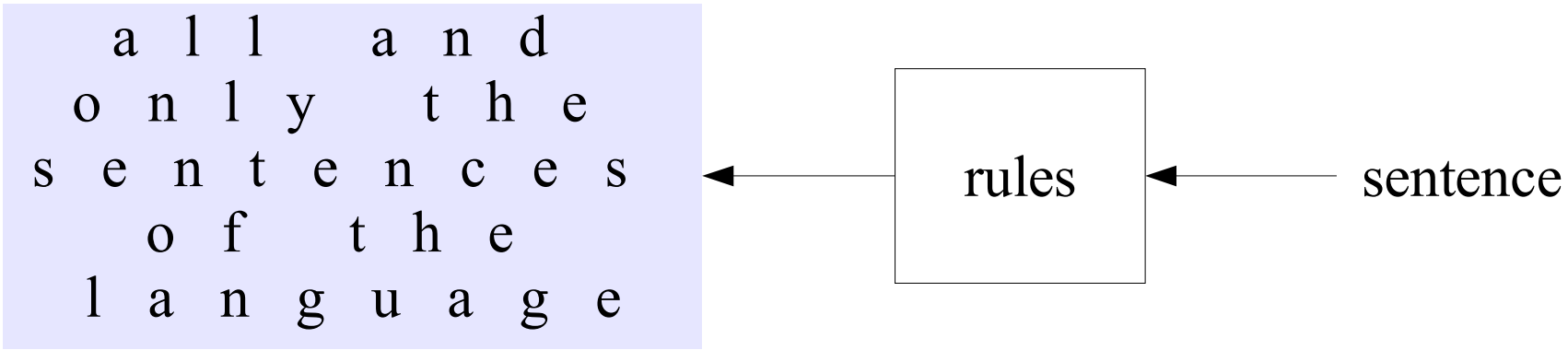
the logic of rewriting rules

- If the sequence recognised by a rewriting rule contains a non-terminal symbol, that rule can only be applied if some rule exists which can itself be applied and which substitutes that symbol so that it can be matched
- for any symbol that occurs in a sequence that is either recognised or substituted by a rewriting rule, the rest of the sequence in which that symbol occurs cannot be reached until that symbol has been successfully matched

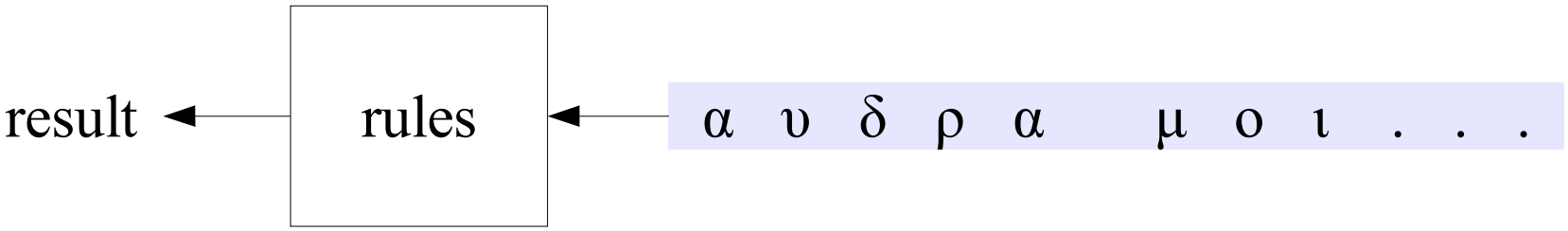
grammar

- we start from Chomsky's description
- alphabet of terminal symbols
- alphabet of non-terminal symbols
- preferred symbol
- rewriting rules

generative grammar



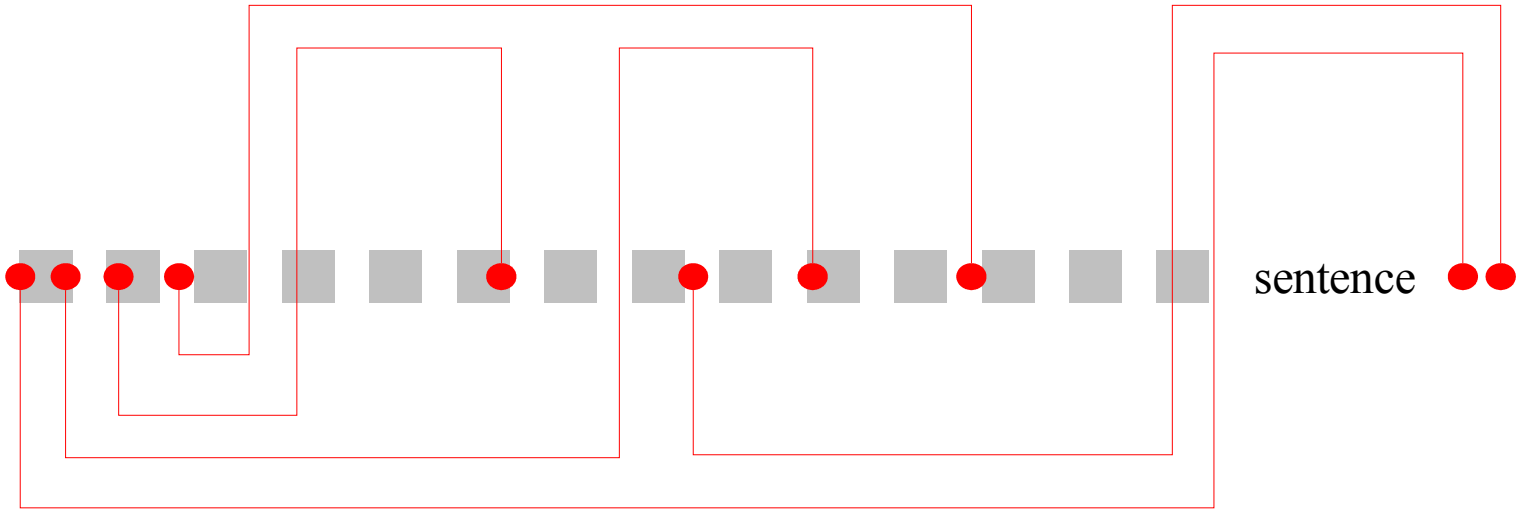
analytic or recognition grammar



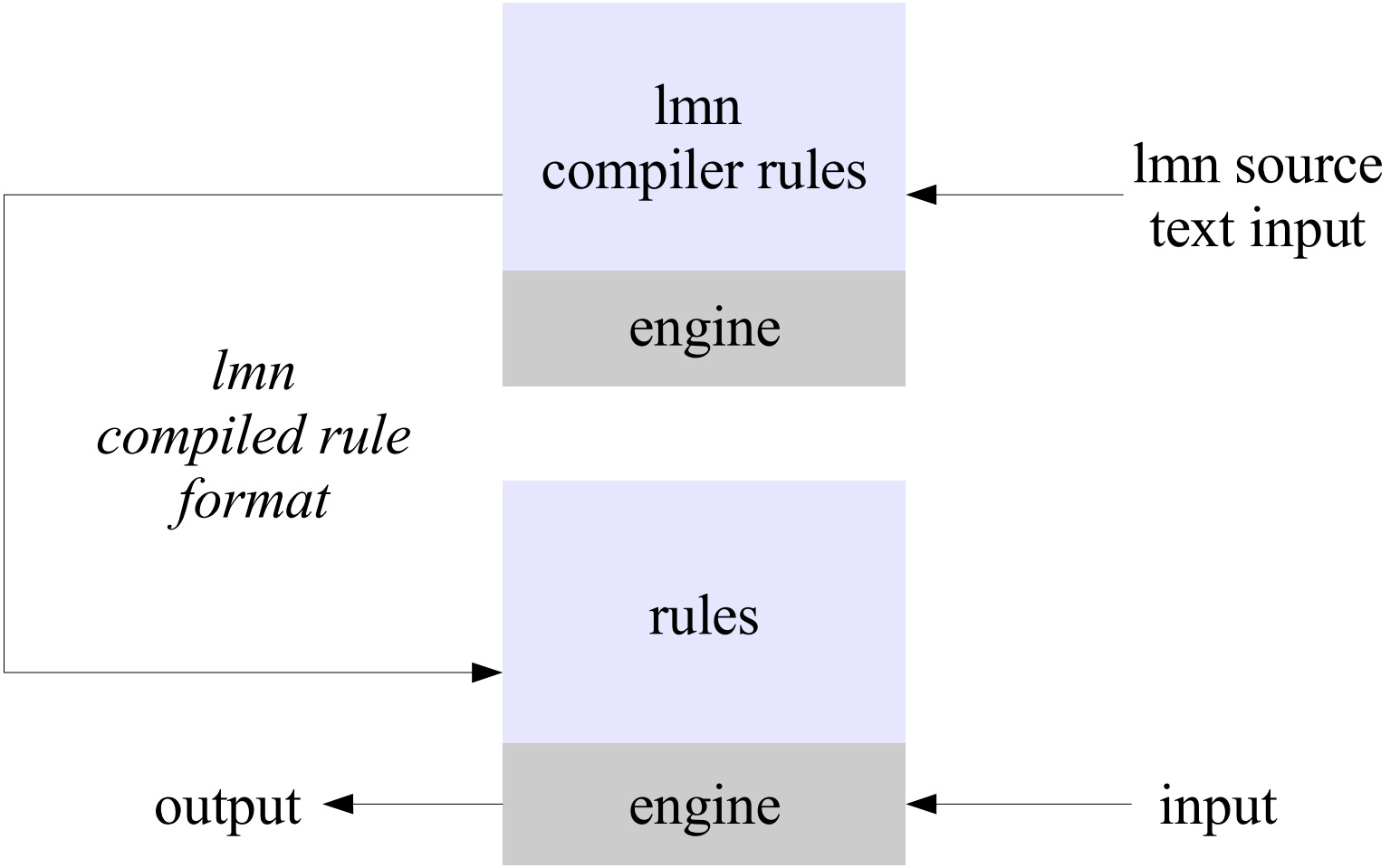
unrestricted analytic grammar

- if unrestricted rules are applied in the analytic sense they can either replace many symbols by few, or few symbols by many
- if rules replace many symbols by few they tend to operate as analysers
- if rules replace few symbols by many they tend to operate as generators
- unrestricted recognition grammar contains generative grammar

the lm-diagram: a complete analysis



the language machine: overview



the language machine: fundamentals

- applies rewriting rules to an input sequence
- recognises and substitutes sequences of symbols
- rules can use terminal and non-terminal symbols
- rule applications triggered by mismatch events
- recognition and substitution phases can each nest
- no restrictions on the form of a rule
- either phase may be empty

the language machine: features

- for efficiency, priority and other constraints limit the number of rules that can be tried
- sequences may contain variable bindings and variable references
- the value of a variable may be a sequence
- the value of variable may be used for matching and substitution
- sequences may contain side-effect actions and external calls

the language machine: the system

- consists of an engine for applying rules, and a metalanguage compiler for translating rules into the form in which the engine can use them
- the engine is implemented as a shared library written in the D programming language, with a small main progam to drive it
- the metalanguage compiler is written in *lmn* metalanguage notation
- the system is developed on linux using the *gdc* D language front-end for Gnu GCC

the language machine

some details

no restrictions on rules

- left-side of rule: sequence to be recognised
- right-side of rule - sequence to substitute
- there is no restriction on the number of symbols that a rule may recognise or substitute
- either left- or right-side may be empty
- right-side empty: delete or ignore
- left-side empty: something from nothing
- rules are tied to mismatch events

deciding which rule to apply

- a rule is tried as a way of resolving a mismatch
- try as few rules as possible
- mismatch event: goal and input in context
- context is enclosing recognition phase
- rules are tried only if relevant to mismatch event
- only one collection of rules at a time
- priority constraints as further restriction

choosing a rule

- each rule belongs to a named collection of rules
- each has priority direction and value, and two symbols L and R
- L relates rule to the mismatch input symbol I
- R relates rule to the mismatch goal symbol G
- rule can be tried if its collection is active, if on basis of L, R, I, and G, it is relevant to the mismatch, and if priority constraints permit this rule in this context

four kinds of rule

- the most efficient kind of rule relates directly to both symbols in a mismatch event
- input-driven or bottom-up rules relate only to the input symbol, and can be tried without regard to the goal symbol in a mismatch event
- top-down or goal-seeking rules relate only to the goal symbol, and can be tried without regard to the input symbol in a mismatch event
- speculative rules relate to neither symbol and can always be tried

alternative rules

- all the rules that are available at a particular mismatch event represent alternative ways of resolving that mismatch.
- if no rule is available to resolve the current mismatch event, an alternative is tried at the enclosing level of rule recognition
- the analysis fails if no alternative at outermost level

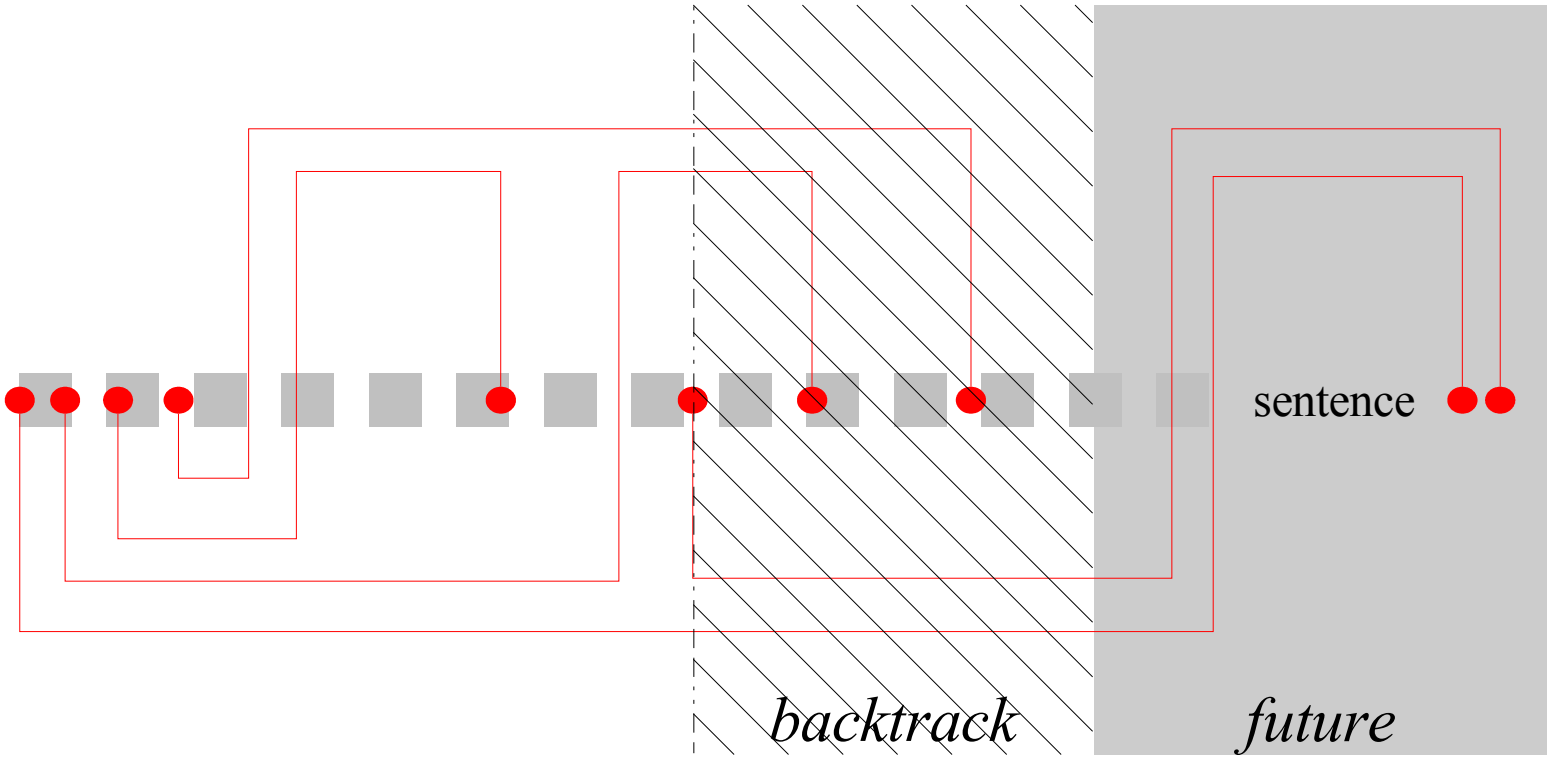
ordering of alternatives

- within each of the four different kind of rule, recently defined rules are tried before older rules, and rules that recognise many symbols are tried before rules that match fewer symbols.
- the different kinds of rule are tried in order: specific, bottom-up, speculative, and top down

backtracking

- reset to the state the system was in at the start of the recognition phase of a rule application, try an alternative analysis from that point
- once the recognition phase is complete, it cannot be reopened for alternative analyses to be tried
- the lm-diagram can help to illustrate what has to be done to reset the system to the state it was in at the start of the recognition phase of a rule

the lm-diagram: a backtrack event



lmn – language meta notation

- we need a language for rules
- high-level assembler for language machine
- direct representation of the actual rules
- intended for use by people
- as far as possible, concise and easy to use
- must be readable, must use ordinary symbols
- must provide its own documentation
- must be capable of compiling itself

a bit of notation

- unicode characters: single quotes
- symbols: alphanumeric, start lowercase
- variable identifiers: alphanumeric, start uppercase.
- rule: left-side `'<-'` right-side `';'`
- structure: `'{'` and `'}'`
- variable bindings: `':'`

writing a rule

- what mismatch events does this rule address?
- is the rule specific to a particular mismatch? The initial symbols L and R must match the mismatch symbols I and G
- is the rule applicable to specific input in any context? The initial symbol L must match the mismatch symbol I – don't care about R
- is the rule applicable to this goal regardless of input? the initial symbol R must match the mismatch symbol G – don't care about L

writing a rule: a minimal rule set

- a minimal rule set can consist of just one rule:

```
- out <- eof - ;
```

- the symbol “out” is predefined - it matches any input symbol. The text representation of that symbol is written to the standard output stream
- the rule is applicable to any input when the goal symbol is “eof”. But it delivers nothing at all
- the “eof” symbol represents the end of all input and is used as the outermost goal

what does that hyphen mean?

- as the initial symbol on the left-side of a rule, the hyphen means: never mind the input, consider only the goal
- as the initial symbol on the right-side of a rule, the hyphen means: never mind the goal, consider only the input
- as the second symbol on the right-side of a rule, the hyphen means that the rule promises to substitute the first symbol on the right-side but in fact delivers whatever follows the hyphen

writing a rule: examples

- specific

```
'cat' <- noun;
```

- input-driven, bottom-up: never mind what the current goal symbol is, try this rule for this input

```
' '   <- - ;
```

- goal-driven, top-down: never mind what the current input symbol is, try this rule for this goal

```
- term  <- expression;
```

writing a rule: hints

- goal symbol represents goal, obstacle, context
- top-down rules do recursive descent
- something from nothing suprisingly useful
  - default conditions
  - terminating conditions
- implied loop at every goal symbol
  - try rules until goal is delivered
- promise to deliver ties rule to specific goal

```
binary :F :A :B <- code — A '+' B;
```

writing a rule: grammars, priorities

- named collections of rules

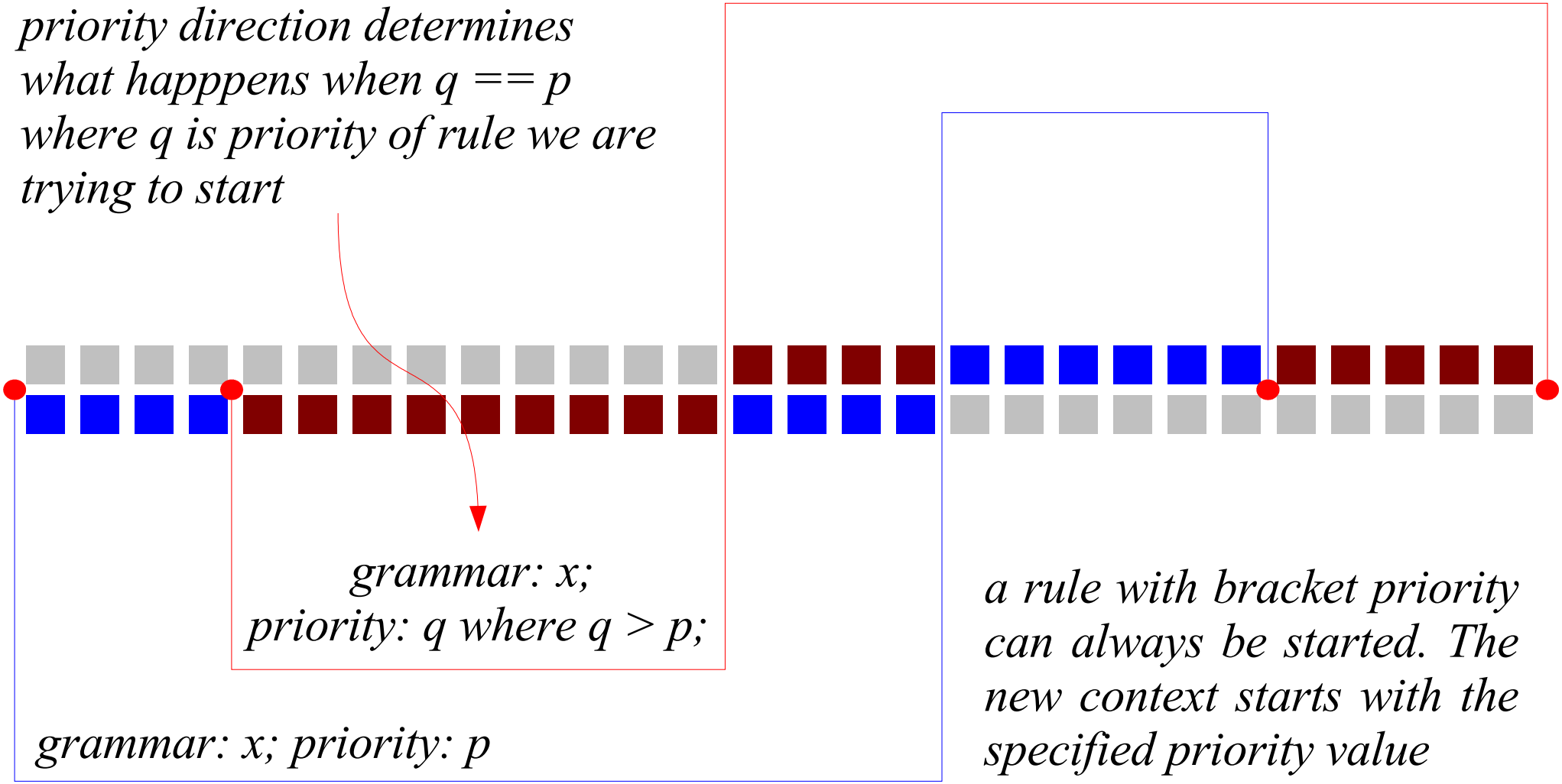
```
.g() ...
.g() { ... }
.g() :
```

- priority direction and value

```
.g(10L) ... .g(B) ... .g(10R)
```

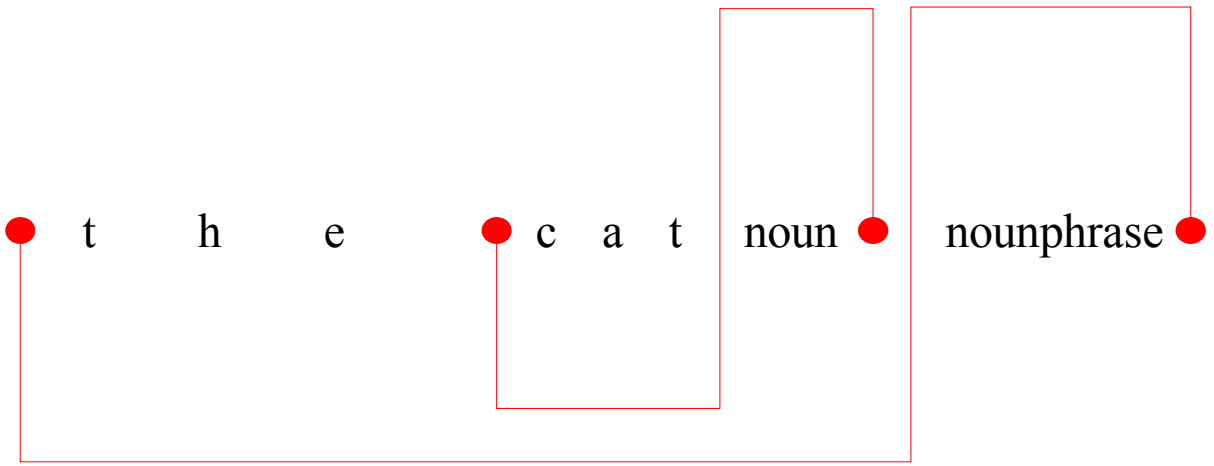
- context takes priority level from rule
- context priority can limit available rules

the lm-diagram: priority



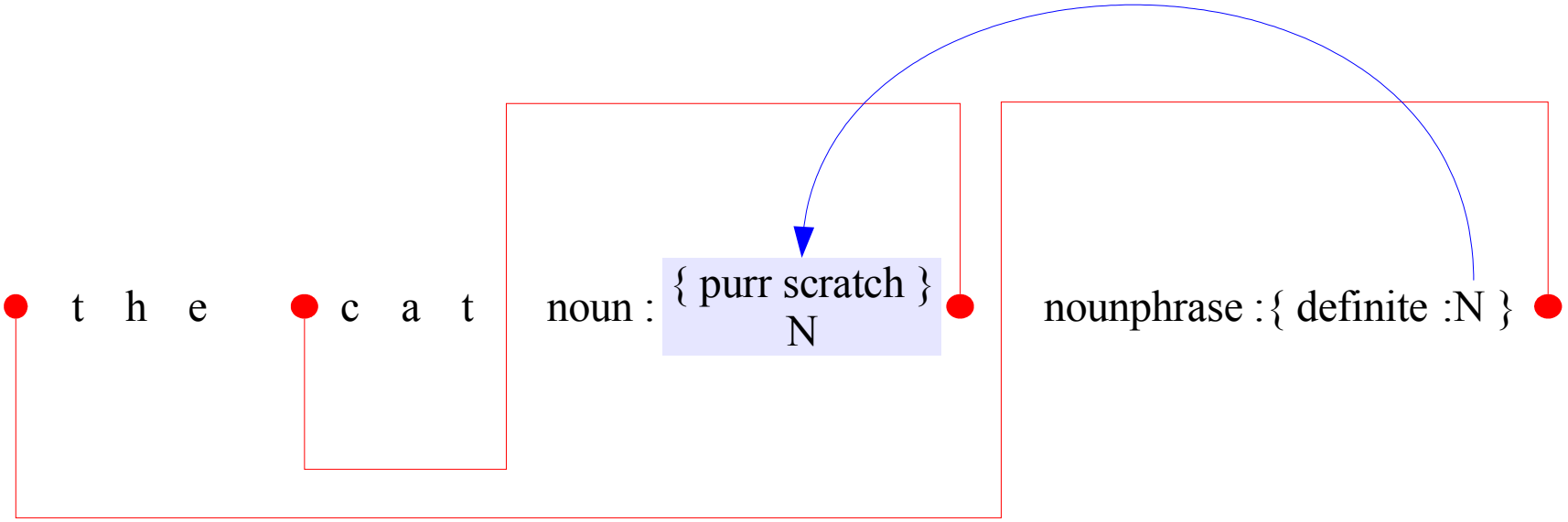
in the abstract all cats are nouns

```
'the ' noun <- nounphrase;
'cat'       <- noun;
'dog'       <- noun;
```

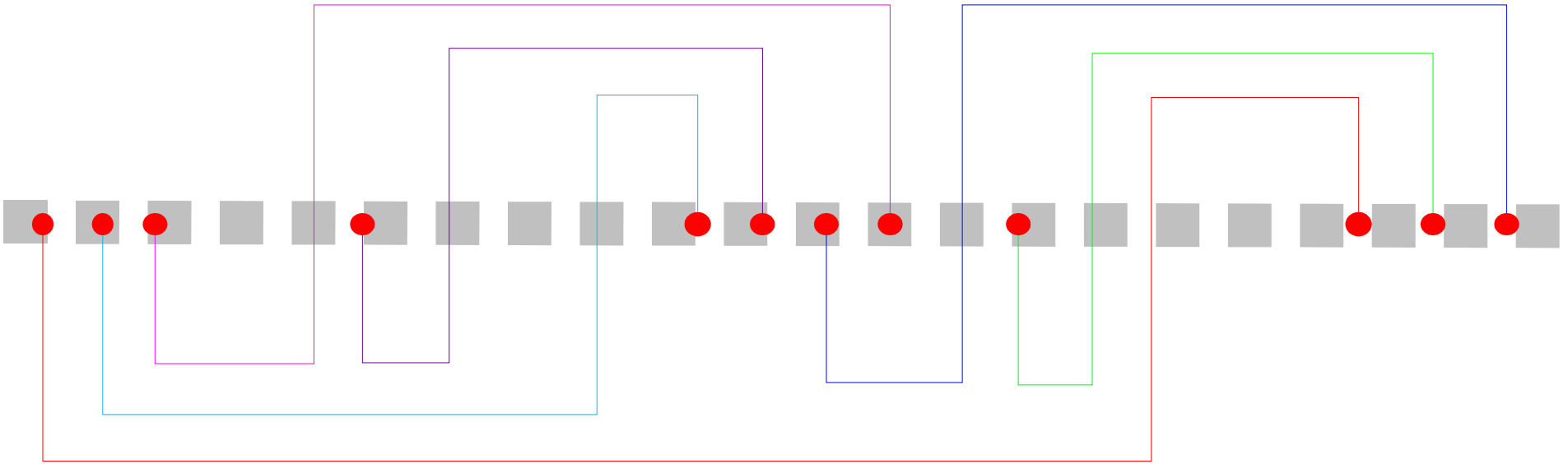


variable bindings and references

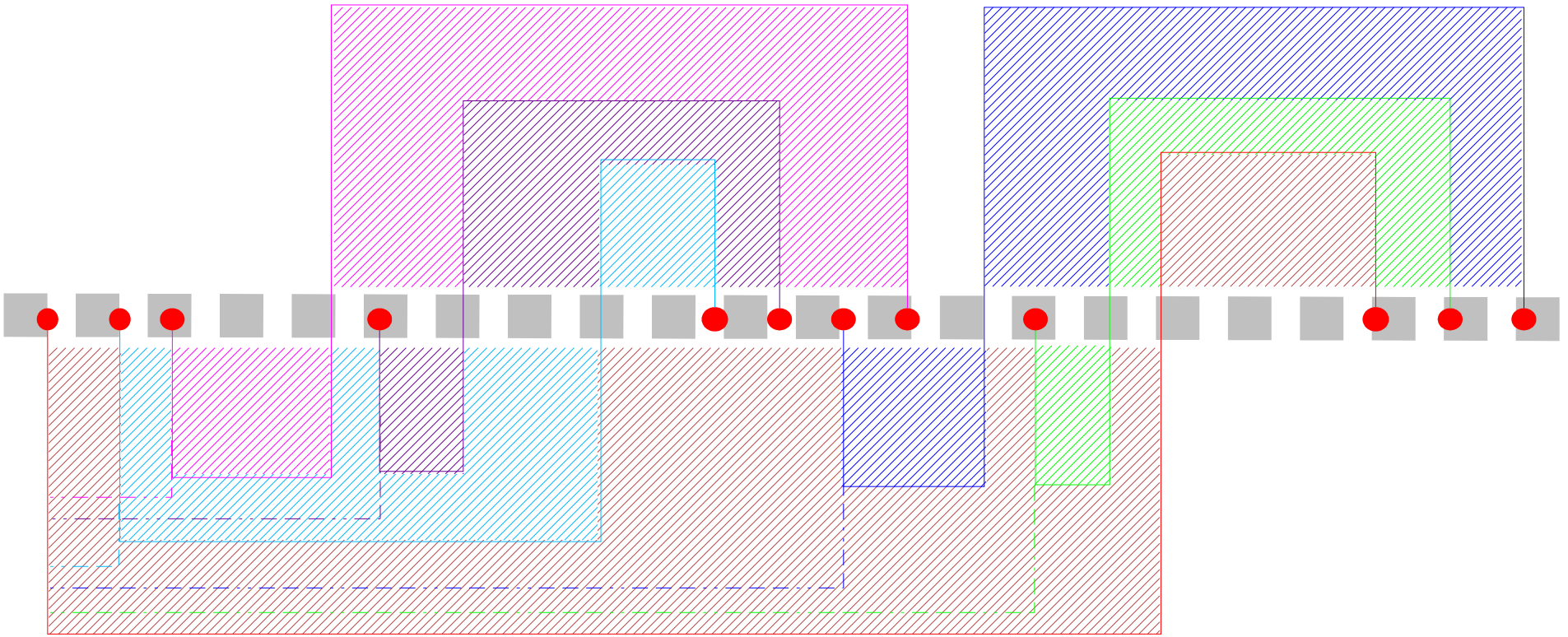
```
'the ' noun :N <- nounphrase :{ definite :N };
'cat'          <- noun :{ purr scratch };
'dog'          <- noun :{ bark lick };
```



variable reference scope



a map of reference scope



summary of variable scope

- a reference to X refers to the most recent instance of X that belongs to the current rule application, or to the most recent instance of X in enclosing recognition phases at the start of the recognition phase of the current rule application.
- when a sequence is being evaluated as the value of a variable, the current rule application for purposes of variable scope is the rule application which provided that sequence as the value of that variable

the language machine

conclusions

performance

- use all available information to select rules
- named grammar, mismatch and initial symbols
- priority constraints allow fewer rules to be used
- variable references are mostly local to context
- care taken to do minimal work going forward
- care taken to require minimal work at backtrack
- backtracking is hard without garbage collection
- two strands – fast co-routine switching required

the lambda calculus

- the lambda calculus does computation by substitution. The lambda calculus and equivalent systems are Turing-complete, that is capable in theory of computing any computable function
- generative grammars with unrestricted rules are known to generate the languages that can be analysed by systems that are Turing-complete
- just the rule and variable system of the language machine can be shown to contain the lambda-calculus, without recourse to side-effect actions

language in general

- the language machine can reasonably claim to be the analytic or recognition equivalent of systems of generative grammar with unrestricted rules – the most general kind of formal grammar
- in other words the language machine can reasonably claim to deal with 'language in general'
- a generative grammar cannot do analysis, but an analytic grammar with unrestricted rules can operate as a generative grammar