Genetic programming with regular expressions

Børge Svingen
Chief Technology Officer, Open AdExchange
bsvingen@openadex.com

2009-03-23

Pattern discovery

Pattern discovery: Recognizing patterns that characterize features in data

Type of data	*Example feature*
Meteorological data	Bad weather
DNA	Predisposition for disease
Seismic data	Presence of oil
Financial data	Changes in stock prices

Purpose of this lecture

Three things:

- Practical how-to on pattern discovery
- Provide an example of using formal methods for solving a practical problem
- Demonstrate a promising topic for future work

Pattern discovery in sequences

We focus on finding patterns in sequences:

- Biological sequences (DNA, RNA, amino acids etc.)
- Time series (temperature, stock prices, etc.)
- Mathematical sequences (arithmetic, geometric etc.)

What do sequences have in common?

What do sequences that share a feature have in common?

- What do genetic sequences that give a predisposition for a disease have in common?
- What do stock price time series that lead to a crack have in common?
- What do geometric sequences have in common?

Training sets

- Training sets: Input to the pattern discovery algorithm
- Positive training set: Contains sequences that have the feature
- Negative training set: Contains sequences that do not have the feature
- Negative training set not always present
  - One solution: Use random sequences as negative training set

Representing sequences - languages

Formal definitions:

- *Alphabet*: A set of characters.
- *String*: A finite sequence over an alphabet.
- *Language*: A set of strings.

We want to represent languages, i.e., the set of strings of the training sets

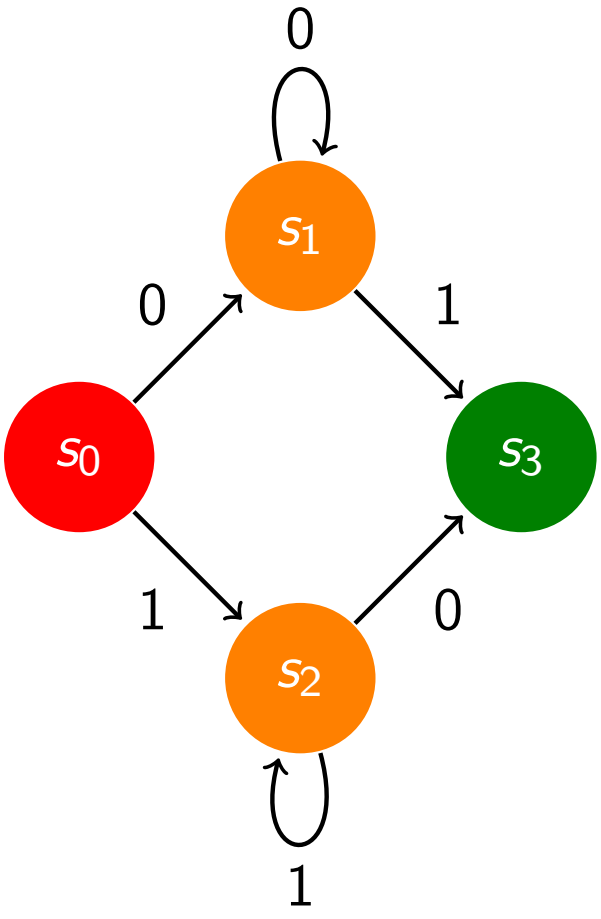
Representing sequences - types of languages

Types of languages:

- *Regular languages*. Can be decided by a finite automaton.
- *Context-free languages*. Can be decided by a push-down automaton.
- *Context-sensitive languages*. Can be decided by a Turing machine with finite memory.
- *Recursive languages*. Can be decided by a Turing machine.
- *Recursively enumerable languages*. Can be enumerated by Turing machines.

We will focus on regular languages.

Deterministic finite automata (DFA)



Represents the language described by the strings

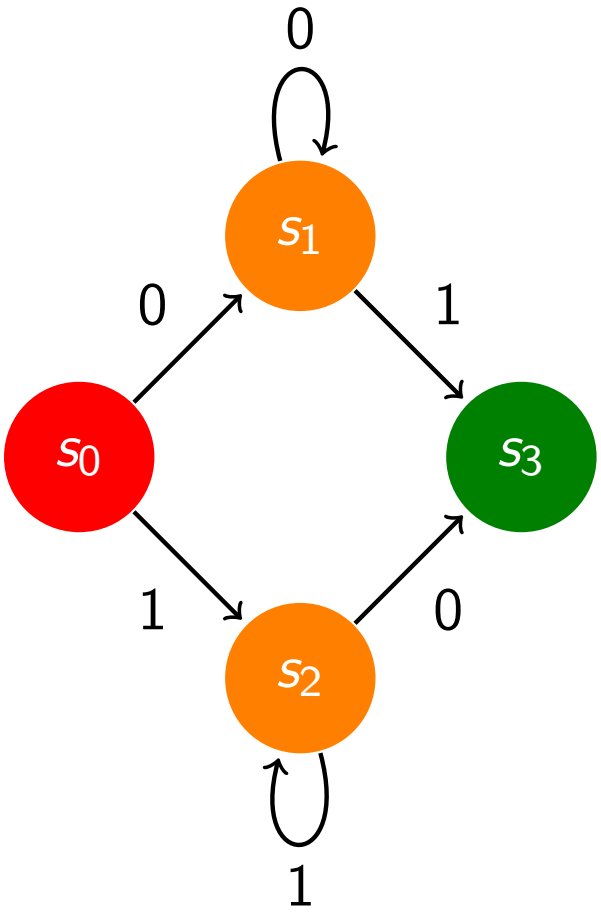
01
001
0001
00001
...
10
110
1110
11110
...

DFA definition

A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- A finite set of states Q.
- An alphabet Σ.
- A transition function $\delta : Q \times \Sigma \to Q$.
- A start state $q_0 \in Q$.
- A set of accept states $F \subseteq Q$.

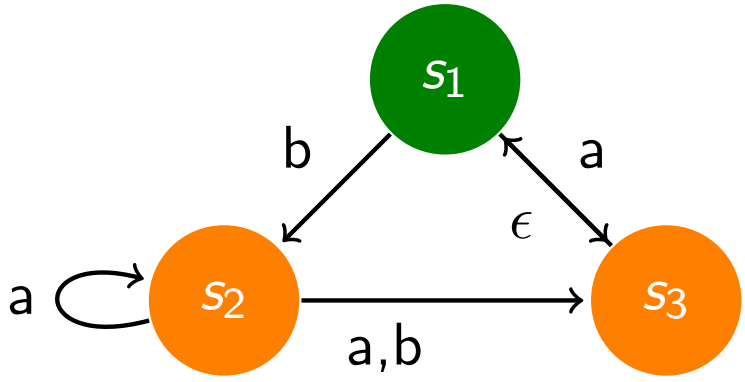
DFA example



$$
\begin{aligned}
Q &= \{s_0, s_1, s_2, s_3\} \\
\Sigma &= \{0, 1\} \\
q_0 &= s_0 \\
F &= \{s_3\}
\end{aligned}
$$

	s0	s1	s2
0	s1	s1	s3
1	s2	s3	s2

Nondeterministic finite automata (NFA)



- NFAs have multiple choices for moving between states.
- Must evaluate all options.
- In multiple states "at once".

NFA definition

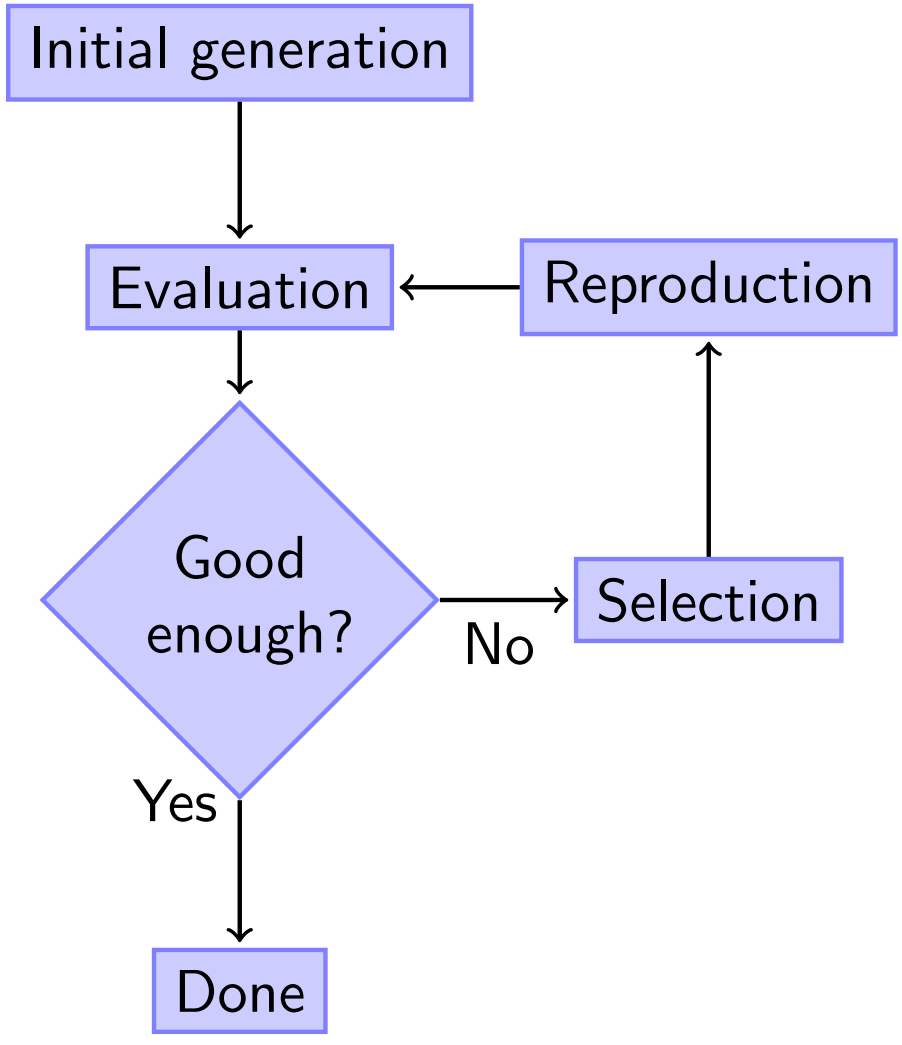
A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- A finite set of states Q.
- An alphabet Σ.
- A transition function $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$.
- A start state $q_0 \in Q$.
- A set of accept states $F \subseteq Q$.

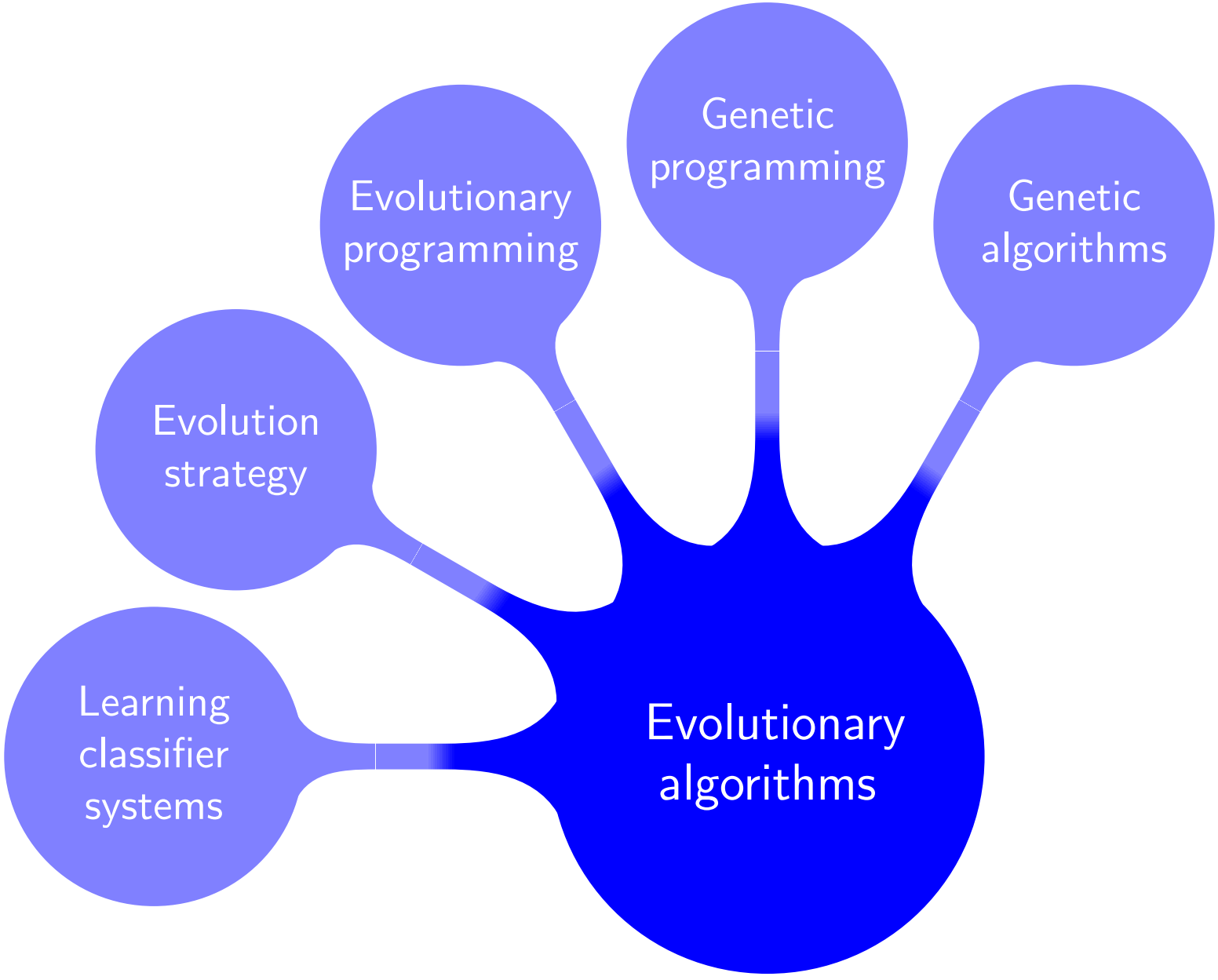
Evolutionary algorithms

Using evolution for solving problems:

- A population of solutions
- Selection based on fitness (how well the solution solves the problem)
- Reproduction with mutation
- Repeat for a number of generations

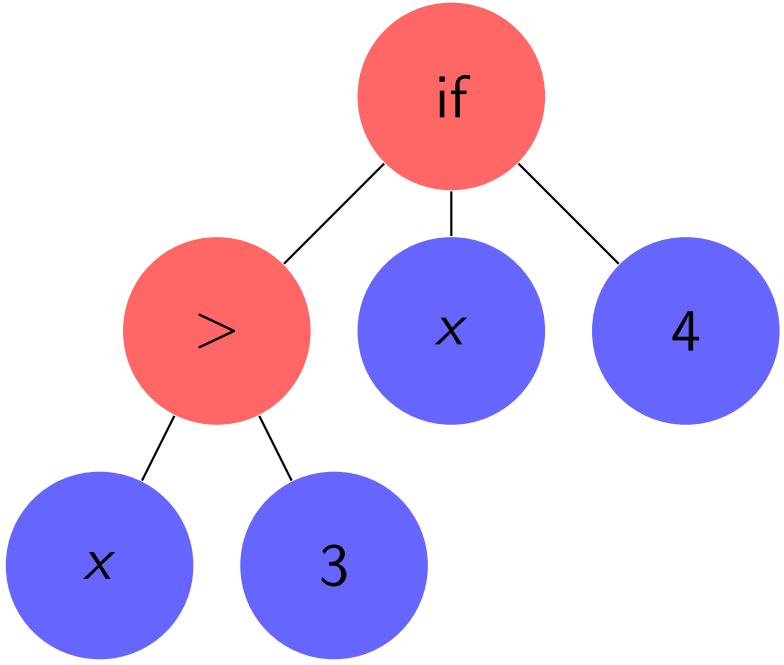


Types of evolutionary algorithms



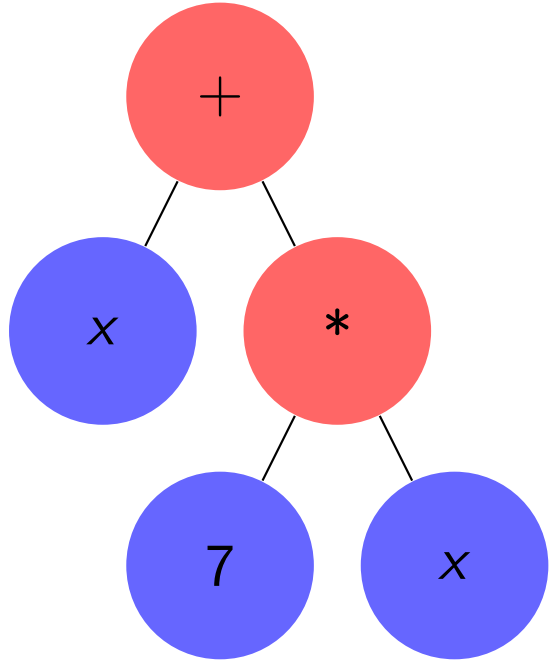
Genetic programming - evolving programs

- In GP the individuals of the population are programs
- The programs are in the form of trees (can be seen as parse trees)
- Fitness is evaluated by running the program



Examples of GP applications

- Designing electric circuits
- Optimization problems
- Robot control
- Pattern discovery
- Symbolic regression



Fitness

- *Fitness* tells us how good a program is at solving the problem.
- Fitness is calculated by a *fitness function*.
- The fitness of a program decides the probability of being selected for the next generation.
- The goal of genetic programming is to optimize the fitness function.

Important: The fitness function needs to allow for gradual improvements.

The fitness function

Different types of fitness:

- *Raw fitness*. Application specific context. $f_r(i,t)$ gives raw fitness for individual i in generation t.
- *Standardized fitness*. Standardized fitness $f_s(i,t)$ is raw fitness adjusted so that lower values are better and 0 is best.
- *Adjusted fitness*. Adjusted fitness is standardized fitness adjusted so that all fitness values fall between 0 and 1, with 1 being the best. $f_a(i,t) = \frac{1}{1+f_s(i,t)}$.
- *Normalized fitness*. Normalized fitness is adjusted fitness normalized so that the sum of program fitness over the whole population is 1. $f_n(i,t) = \frac{f_a(i,t)}{\sum_{k=1}^{M} f_a(k,t)}$ where M is the population size.

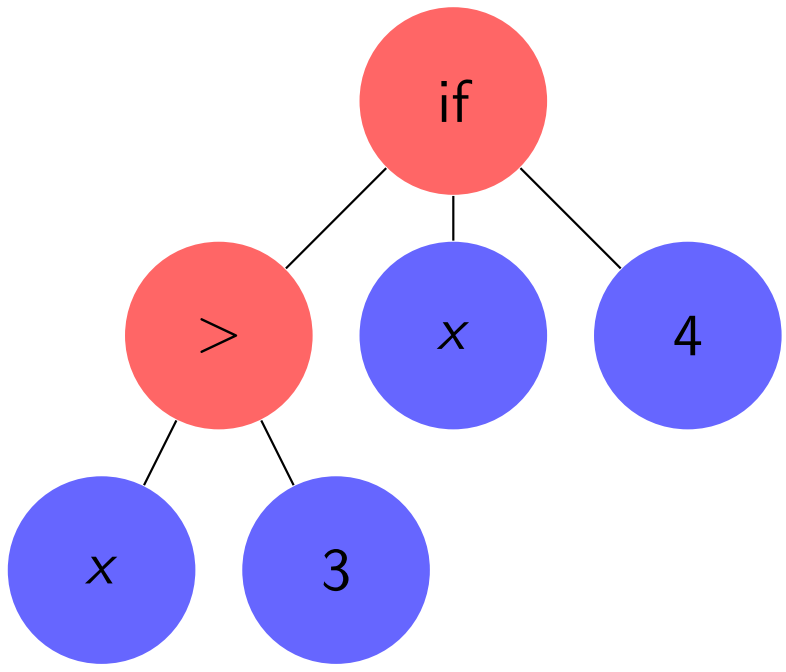
Program primitives

- Programs are built from a function set and a terminal set.
- An important property is *closure*: All functions should accept all values returned by other functions or terminals.
- In this example,

$$F \subseteq \{if, >\}$$

and

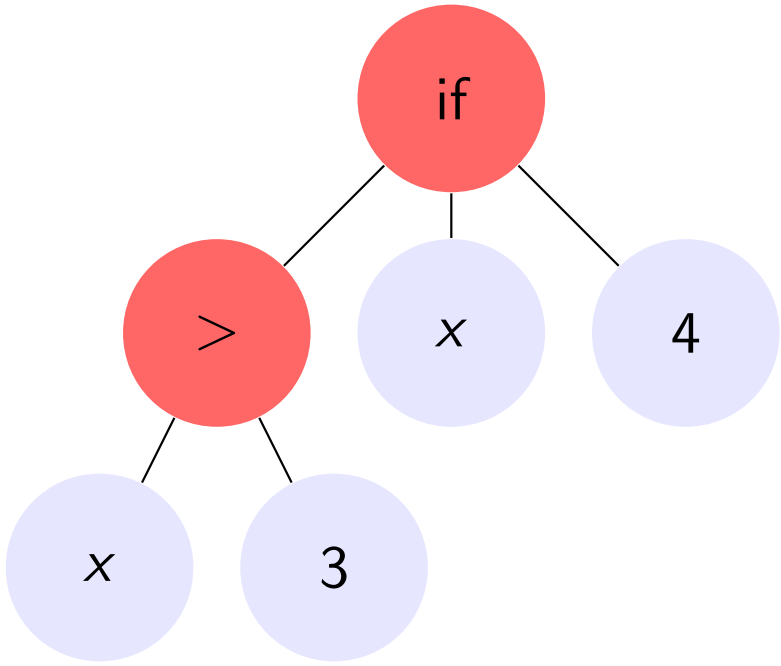
$$T \subseteq \{x, N\}$$



The function set

The function set

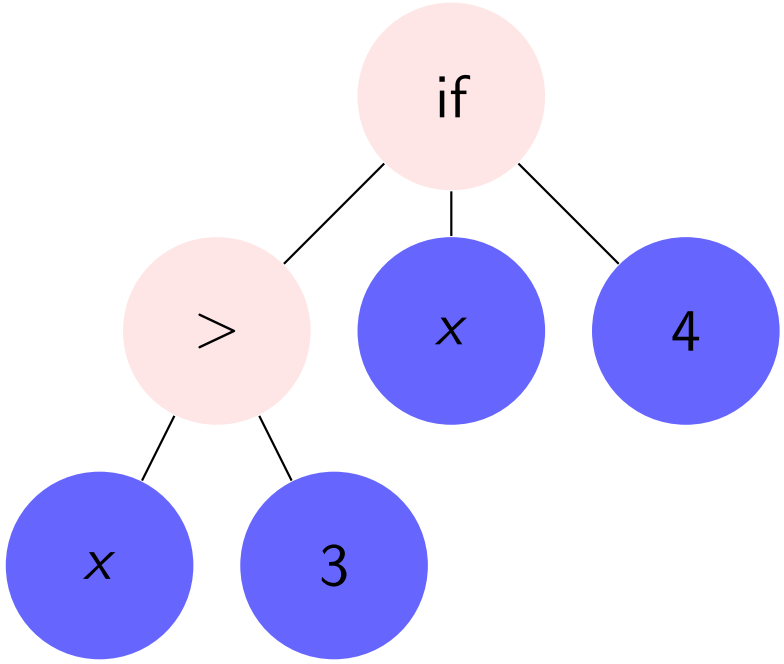
- Are the internal nodes of the program tree
- Has one or more children providing input
- Can be functional or have side effects



Terminal set

The terminal set

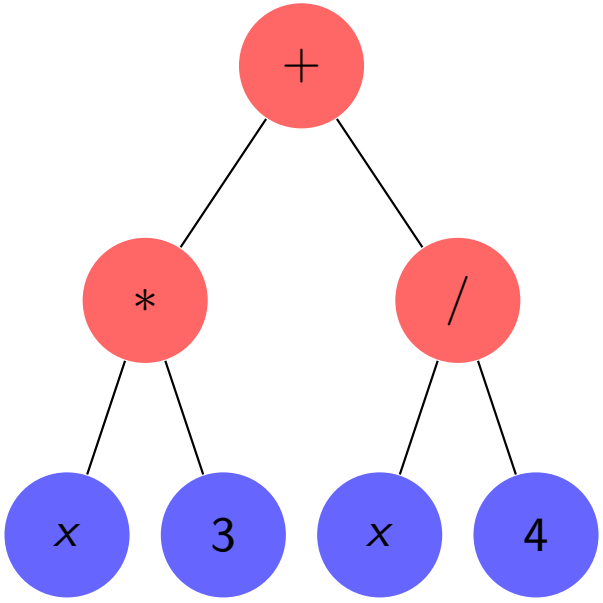
- Are the leaf nodes of the program tree
- Can have side effects
- *Ephemeral* terminals is a special case, typically used for constants

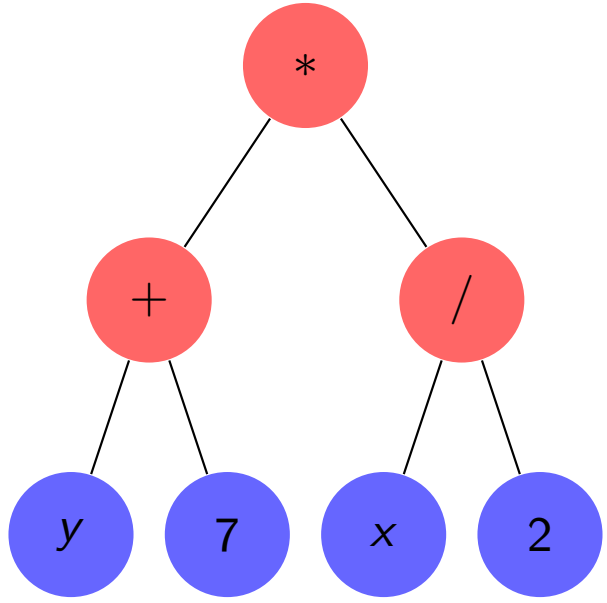


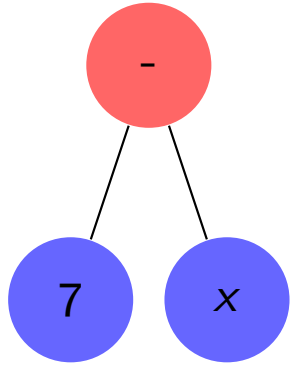
Growing trees

- The initial population consists of random trees
- Functions and terminals randomly selected
- Two main ways of building random trees of a given depth:
  - The full method: All leaves have the same depth.
  - The grow methods: Randomly choose between functions and terminals, create leaves of different depth.
- The "ramped half-and half" method:
  - Equally distributed between different depths
  - For each tree of a given depth, randomly choose between the full or grow method
  - Creates tree shape diversity

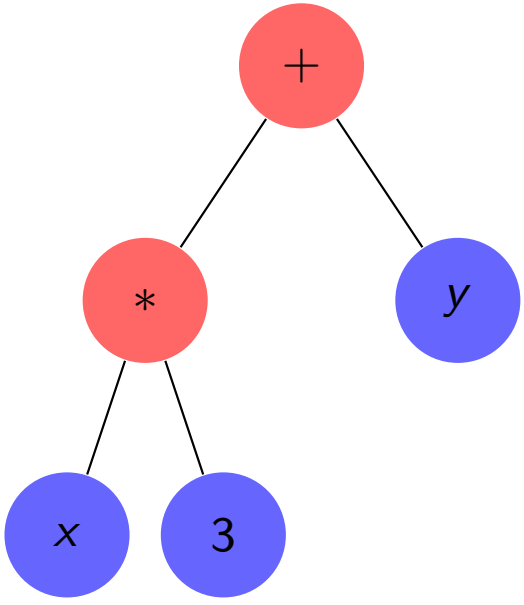
Growing trees - full

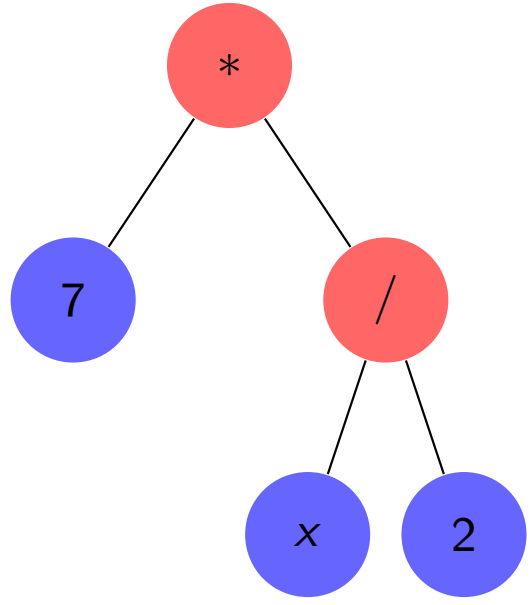


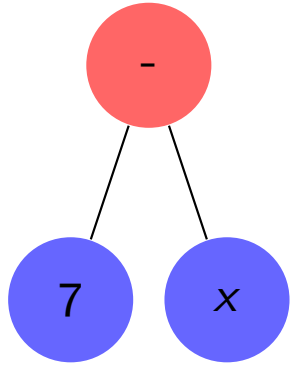




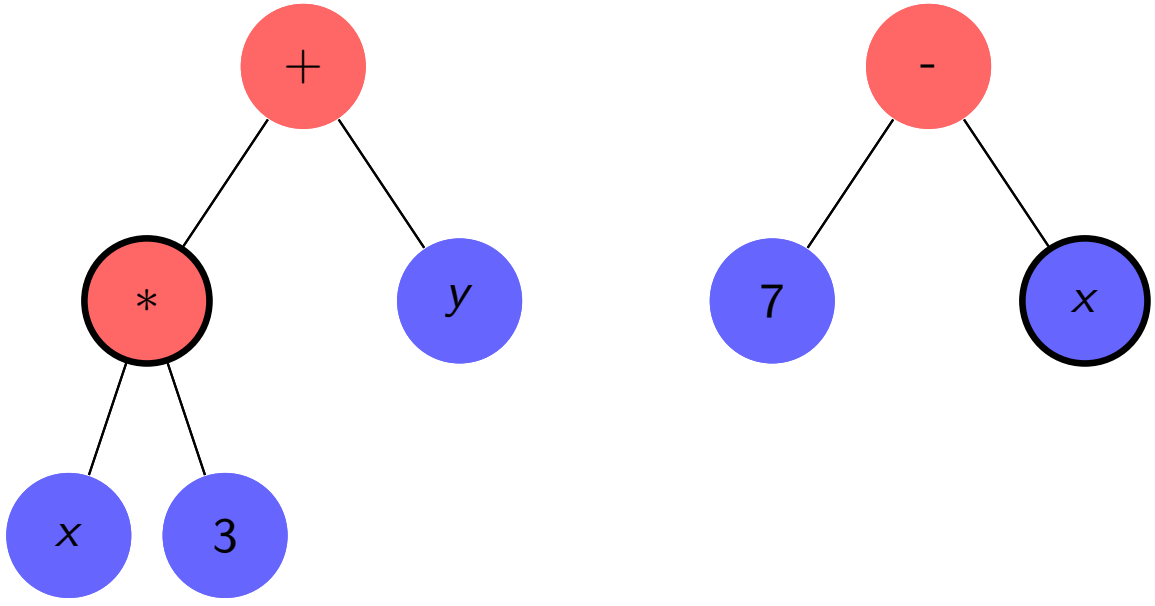
Growing trees - grow



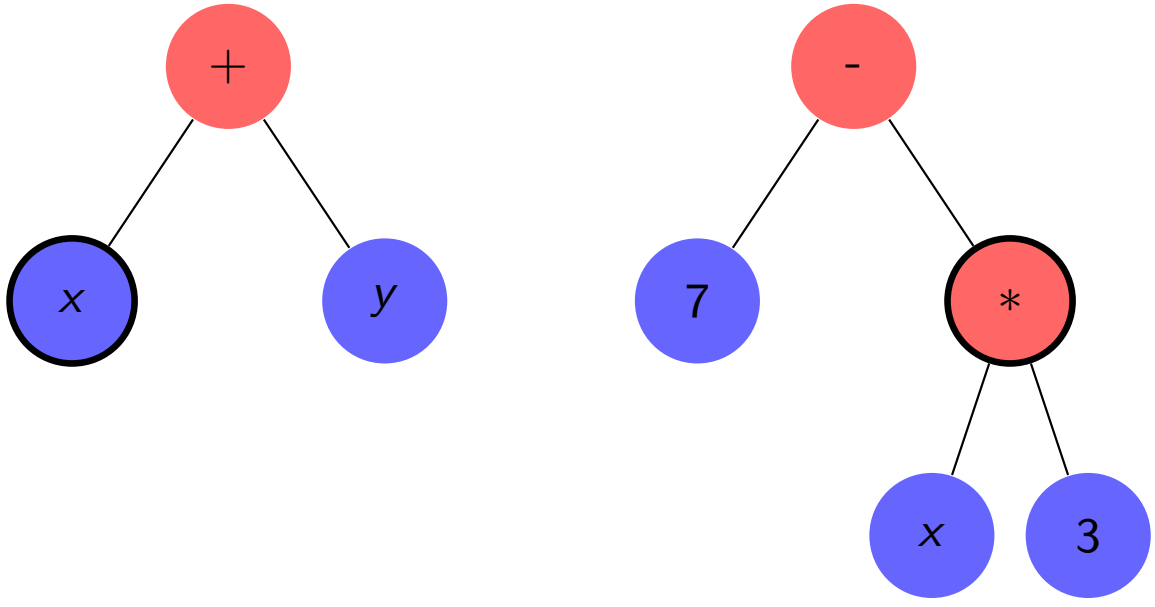




Reproduction - crossover

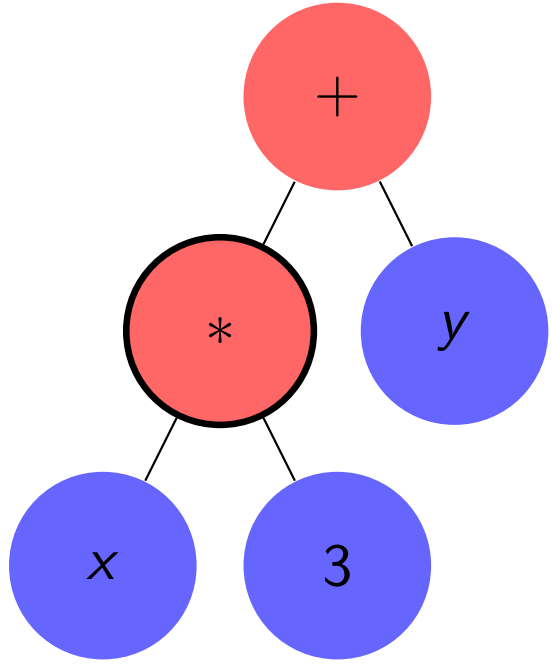


Reproduction - crossover results



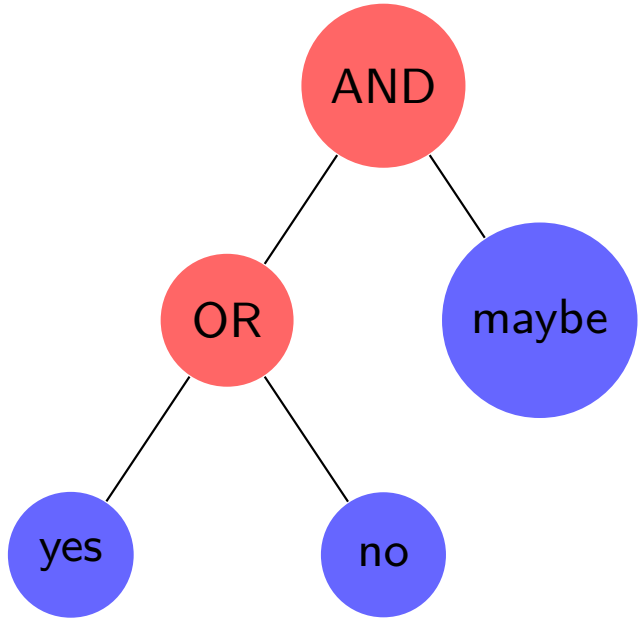
Crossover maintains building blocks

- Crossover point is selected randomly.
- Whole subtrees are exchanged between programs.
- The subtrees represent a separate piece of functionality.
- This causes building blocks of good solutions to survive to future generations, and then recombine.



Genetic programming with search

- We want to find patterns.
- Solution: Genetic programming where the programs are queries.
  - The patterns are represented by queries.
  - The programs are queries.



Evolving queries

- Every member of the population is a query.
- We evaluate each query by searching the training sets.
- The fitness function is given by how close the queries match the training sets.
- Trivial fitness: Count number of incorrect classifications.

Genetic programming with search - an example

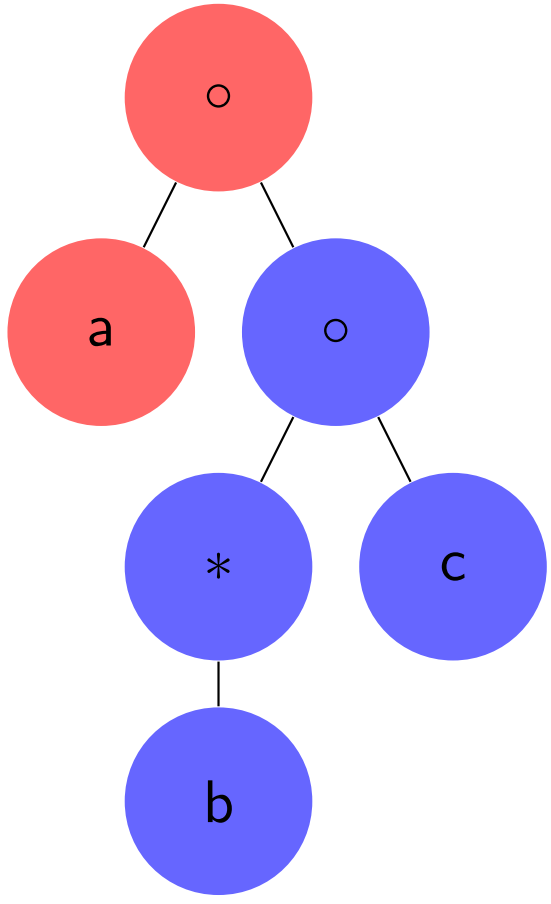
An example of genetic programming with search ([3, 2, 5]):

- Genetic programming done on the genetic programming mailing list.
- Simple single word based search.
- Trying to classify articles about GP selection methods.
- GP done on positive and negative training sets.
- Results tested on separate test set.

```
ADF1  (IF (OR P0 (PRESENT candidate))
        (IF (+
              (PRESENT tournament)
              (PRESENT demes)
            )
          1
          P0
        )
        (IF (PRESENT tournaments)
          8607
          (IF (PRESENT tournament)
            1
            (PRESENT (- (PRESENT scant) 1))
          )
        )
      )
ADF2  (+ 3980 (NOT P0))
ADF3  (IF (PRESENT tournament)
        1
        (- (ADF1 P0))
      )
RPB0  (IF (ADF2 1 1)
        (-
          (- (PRESENT deme))
          (ADF3 (PRESENT pet))
        )
        (ADF3 0))
RPB1  (IF (PRESENT galapagos)
        5976
        (PRESENT deme)
      )
RPB2  1
```

Picking a query language

- There are a number of query languages available (SQL, XQuery, SPARQL ...)
- For sequences: Regular expressions
- Advantage with regard to GP: Regular expressions can be seen as trees



ab∗c

Regular expressions

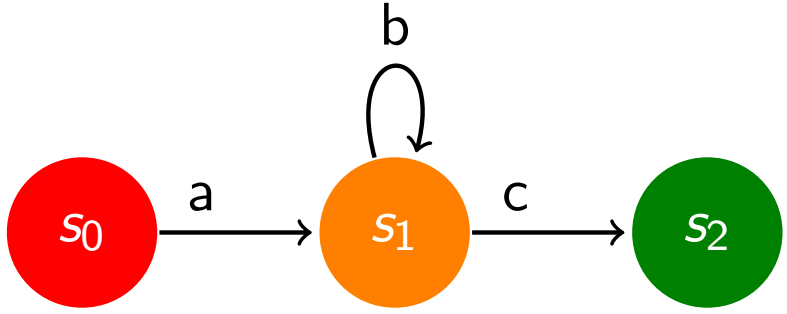
Regular expressions can be defined by the following grammar:

$$
\begin{aligned}
R &\rightarrow a, \text{ for some } a \in \Sigma && (1)\\
R &\rightarrow \epsilon && (2)\\
R &\rightarrow \emptyset && (3)\\
R &\rightarrow (R \cup R) && (4)\\
R &\rightarrow (R \circ R) && (5)\\
R &\rightarrow (R*) && (6)
\end{aligned}
$$

Σ is here the alphabet used, $(R_1 \cup R_2)$ matches either R_1 or R_2, and $(R_1 \circ R_2)$ matches R_1 followed by R_2. (R_1*) matches any number of occurrences of R_1.

Why regular languages are called regular

- Regular expressions represent regular languages.
- Important consequence of this: *Regular expressions and DFAs are equivalent*.
- DFA equivalent to **ab**$*$**c** shown in the figure.



Equivalence proof for DFA and regular expressions

Proof outline:

- DFAs and NFAs are equivalent
  - DFA $\longrightarrow$ NFA is trivial, DFA $\subseteq$ NFA.
  - NFA $\longrightarrow$ DFA: Create DFA with "collective" states.

  Regular expression $\longrightarrow$ DFA

  1. Build NFA recursively for regular expression
  2. Convert NFA to DFA

  DFA $\longrightarrow$ regular expression

  - More complex ... The main idea is to use GNFAs, NFAs where the edges may contain regular expressions, and convert the GNFA to a regular expression

Pattern evolution

An algorithm for evolving DFAs:

1. Use GP to find regular expressions.
2. Convert the regular expressions to DFA.

A practical example

- Used the Tomita benchmark languages, a set of seven regular languages.
- For each language, used positive and negative training sets of 500 strings, the latter randomly created.
- Each GP individual was a regular expression tree.
- Each regular expression tree was evaluated on the training sets by creating a DFA.
- Population size of 10000 over 100 generations.

[4]

The Tomita benchmark languages

Language	*Description*
TL1	1*
TL2	(10)*
TL3	no odd 0 strings after odd 1 strings
TL4	no 000 substrings
TL5	an even number of 01's and 10's
TL6	number of 1's - number of 0's is multiple of 3
TL7	0*1*0*1*

Function set

Function	*Arity*	*Explanation*
$+$	2	Builds an automaton that accepts any string accepted by one of the two argument automata.
.	2	Builds an automaton that accepts any string that is the concatenation of two strings that are accepted by the two argument automata, respectively.
*	1	Builds an automaton that accepts any string that is the concatenation of any number of strings where each string is accepted by the argument automaton.

Terminal set

Terminal	*Explanation*
0	Returns an automaton accepting the single character '0'.
1	Returns an automaton accepting the single character '1'.

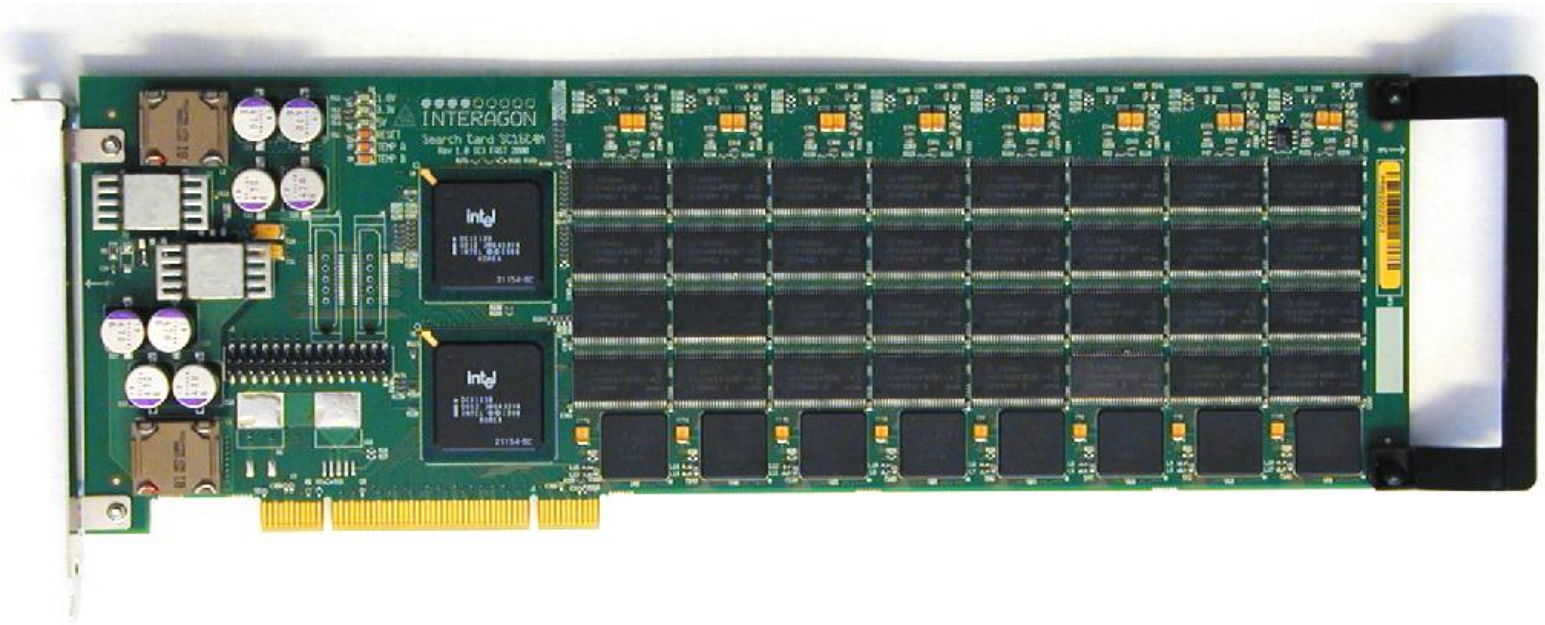
Results 1-4

Language	*Solution*	*Simplified Solution*
TL1	(* (* 1))	1*
TL2	(* (* (. 1 0)))	(10)*
TL3	(. (* (+ (. 1 (+ (+ 1 1) (. 1 0))) (* 0))) (. (* (+ (. 1 (+ (+ 1 (. 0 0)) (. 1 0))) (* (. 0 0)))) (* 1)))	(11 | 110 | 0)*(11 | 100 | 110 | 00)*1*
TL4	(+ 0 (. (+ (* (+ 1 (. 0 (+ 1 (. 0 1))))) 1) (+ (+ (. (. 0 0) (* 1)) 0) (* (+ 1 (. (+ (. 1 0) 1) 0))))))	((1 | 01 | 001)*|001*|0)(1 | 100 | 10)*

Results 5-7

Language	*Solution*	*Simplified Solution*
TL5	(+ (* (+ 0 (. (. 0 (* (* (. (* 0) 1)))) 0))) (* (. (. 1 (* (. (* 0) 1))) (* 1))))	(0 | 0(0*1)*0)* | (1(0*1)*1*)*
TL6	(* (+ (* (+ (. 1 (. (* (. 1 0)) 0)) (. (. 0 (* (* (. 0 1)))) 1))) (+ (* (+ (. 1 (. 1 1)) (. (. (. 1 1) (* (. 0 1))) 1))) (. (. (. 0 (* (* (. 0 1)))) 0) 0))))	(1(10)*0 | 0(01)*1 | 11(01)*1 | 0(01)*00)*
TL7	(. (. (. (* (* 0)) (* 1)) (* 0)) (* (+ 1 1)))	0*1*0*1*

Pattern Matching Chip (PMC)



The end.

Bibliography I

Arne Halaas, Børge Svingen, Magnar Nedland, Pål Sætrom, Ola Snøve, and Olaf Birkeland.
A recursive MISD architecture for pattern matching.
IEEE Transactions on Very large Scale Integration (VLSI) Systems, 12(7):727–734, July 2004.

Børge Svingen.
GP++ an introduction.
In John R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 231–239, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.

Børge Svingen.
Using genetic programming for document classification.
In John R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 240–245, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.

Bibliography II

Børge Svingen.
Learning regular languages using genetic programming.
In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 374–376, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

Børge Svingen.
Using genetic programming for document classification.
In Diane J. Cook, editor, *Proceedings of the Eleventh Interational Florida Artificial Intelligence Research Symposium Conference*. AAAI Press, 1998.

Bibliography III

Michael Sipser.
Introduction to the Theory of Computation.
PWS Publishing Company, 1997.

M. Tomita.
Dynamic construction of finite-state automata from examples using hill climbing.
In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108, Ann Arbor, MI, 1982.