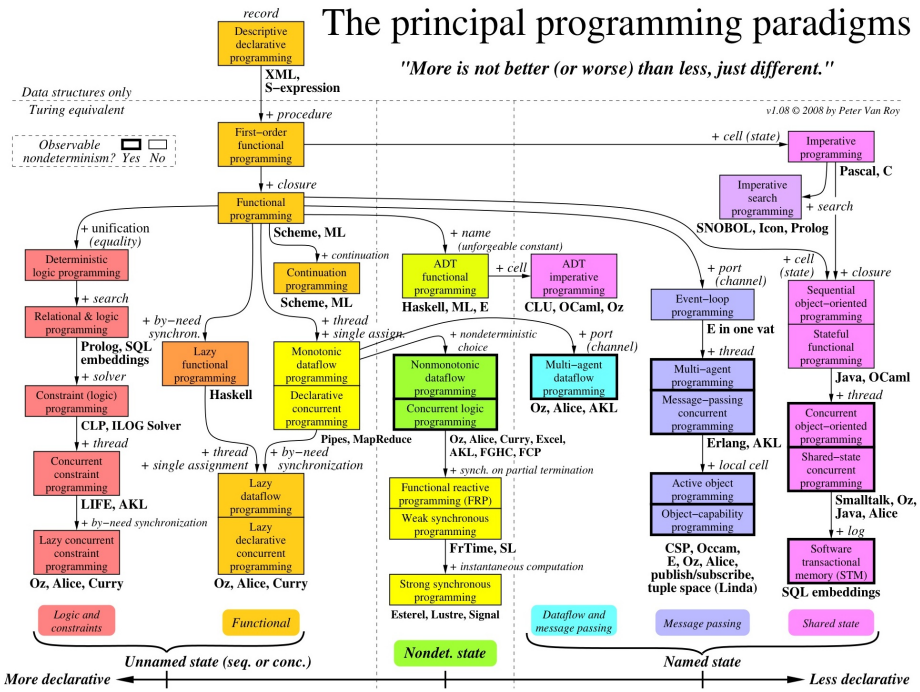
TDDA69 Data and Program Structure
Specialized Computation Models

Cyrille Berger

LINKÖPING UNIVERSITY



Lectures

Lecture content

- Constraint Programming
- Specialized Declarative Evaluators
  - Make
  - Regular Expressions

Constraint Programming

Constraint Programming

- Expresses constraints between variables
- Examples: Prolog…

Evaluator

- A(X,Y) :- X+Y>0, B(X), C(Y)
- How to evaluate?
  - Brute force?
  - Propagate and search

Propagate and search

- Partial information
- Local decuction
- Controlled search

Applications

- Circuit verification
- Real-Time Control systems
- Spreadsheets

...

Make

Specialized Declarative Evaluators

Make

- *Make* is a build automation tools which specify how to generate output files according to a set of rules and a set of input files
- Commonly used under Unix to build C/C++ program
- But can be use to control generation of anything, really (latex...)
- Alternative: Ants, nmake,

Makefile

- A Makefile is made of a set of rules:
```
TARGETS: PREREQUISITES
__RECIPE
```
- The TARGETS is the output files
- The PREREQUISITES is the list of files that you need to generate the TARGETS
- RECIPE is how to generate the output from the input
- Example:
```
.PHONY: all
all: a.out
a.out: main.cpp
___gcc main.cpp
```


How to write a Makefile interpreter (1/2)

- Parse the makefile into a set of rules
  - [TARGETS, [PREREQUISITES], RECIPE]
  - rules = [['all', ['a.out'], ''], ['a.out', ['main.cpp'], 'gcc main.cpp']
  - phony = ['all']

How to write a Makefile interpreter (2/2)

- Interpreter
```
def exectute_target(target, rules, phony):
  for rule in rules:
    if(rule[0] == target):
      should_execute = False
      for p in rule[1]:
        execute_target(p, rules, phony)
        if(not p in phony and file.date(p) > file.date(target)):
          should_execute = True
      if(should_execute and not execute(rule)):
        raise Exception('Failed to generate target:' + target)
  if(not File.exist(target)):
     raise Exception('Missing file: ' + target))

if __name__ == '__main__':
  (rules, phony) = parse('Makefile')
  execute_target(rules[0][0], rules, phony)
```


To make thing a bit more complicated

- Is it convenient to write:
```
all: myprogram
myprogram: main.o a.o b.o
__gcc main.o a.o b.o -o
myprogram
main.o: main.c
__gcc main.c -o main.o
a.o: a.c
__gcc a.c -o a.o
b.o: b.c
__gcc b.c -o b.o
```
- It is nicer to
```
all: myprogram
myprogram: main.o a.o
b.o
__gcc $^ -o $@
%.o: %.c
__gcc -c $< -o $@
```
- Rules are defined with template

Regular Expressions

Regular Expression

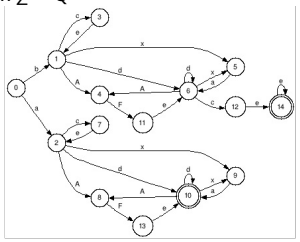
- A regular expression (regex) describes a set of possible input strings
- They can be used for matching strings and for search & replace
- They are commonly used for simple parsing
  - Makefile rules
  - Process natural languages
  - Field validation
  - ...

Basic Regular Expressions

- *Concatenation*: aabaab
  yes: 'aabaab'
  no: every other string
- *Wildcard*: .u.u.u.
  yes: 'cumulus', 'jugulum'...
  no: 'succubus', 'tumultuous'...
- *Union* aa | baab
  yes: 'aa', 'baab'
  no: every other string
- *Closure* ab*a
  yes: 'aa', 'aba', 'abba'...
  no: 'ab, 'ababa'...
- *Parentheses* a(a|b)*aab
  yes: 'aaab', 'abbaab', 'ababaab'...
  no: 'abcaab', 'acabaab'...
- *One or more* a(bc)+de
  yes: 'abcde', 'abcbcde'...
  no: 'ade', 'bcde'...
- *Range* [A-Z][a-z]*
  yes: 'Capitalized', 'Word'...
  no: 'uncapitalized', 'wOrd'...
- *Exactly k* [0-9]{2}(0[0-9]|10|12)([0-2][0-9]|30|31)-[0-9]{4}
  yes: 900431-3234...
  no: 902331-3234, 900452-3234...
- *Negations* [^aeiou]{6}
  yes: rhythm
  no: decade

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton consists of
  - Q a finite set of *states*
  - ∑ a finite set of *input symbols*
  - q_0 a *start state*
  - F a set of *final states*
  - δ a *transition function* from Q x ∑ -> Q



Interpret a Deterministic Finite Automaton

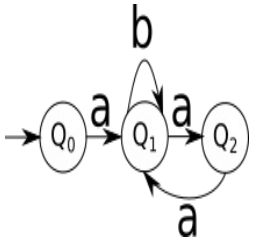
- Take a word composed of letters in ∑
- Does the word match the DFA?
  - Treat the word as a stream of input symbols
  - q is the current state
  - Start in q = q_0
  - Given c the current input symbol, then q=δ(q,c)
  - When no input symbols remain, if q ∈ F, then accept otherwise reject

DFA and Regexp

- Regexps are a concise way to describe a set of strings
- DFAs are machine to recongnize whether a given string is in a given set
- Theorem: for any DFA, there exists a regular expression to describe the same set of strings, for any regular expression, there exists a DFA that recognize the set
- Conscequence: to implement a regular expression matcher, buid a DFA and execute it

DFA and Regexp - Example

- DFA:



Q_0 and Q_2 are final states

- Regexp: (ab*a)*

Conclusion

- Constraint Programming
- Make
- Regular Expression