EECS 490 – Lecture 21

Logic, Constraints, and Dependencies

Announcements

- Project 4 due tonight at 8pm
- HW5 due Tue 12/5 at 8pm
- Project 5 due Tue 12/12 at 8pm

Prolog Example: Quicksort

- Partition:

```
partition(_, [], [], []).
partition(Pivot, [Item|Rest], [Item|Less], NotLess) :-
  Item < Pivot,
  partition(Pivot, Rest, Less, NotLess).
partition(Pivot, [Item|Rest], Less, [Item|NotLess]) :-
  Item >= Pivot,
  partition(Pivot, Rest, Less, NotLess).
```

- Sort:

```
quicksort([], []).
quicksort([Item|Rest], Sorted) :-
  partition(Item, Rest, Less, NotLess),
  quicksort(Less, SortedLess),
  quicksort(NotLess, SortedNotLess),
  append(SortedLess, [Item|SortedNotLess], Sorted).
```

Prolog Example: Primes

- Sieve of Eratosthenes:

```
numbers(2, [2]).
numbers(Limit, Numbers) :-
  M is Limit - 1, numbers(M, NumbersToM),
  append(NumbersToM, [Limit], Numbers).

is_not_multiple(N, D) :- R is mod(N, D), R =\= 0.

filter_not_multiple(_, [], []).
filter_not_multiple(Factor, [First|Rest],
                    [First|FilteredRest]) :-
  is_not_multiple(First, Factor),
  filter_not_multiple(Factor, Rest, FilteredRest).
filter_not_multiple(Factor, [_|Rest], FilteredRest) :-
  filter_not_multiple(Factor, Rest, FilteredRest).

sieve([]).
sieve([First|Rest], [First|SievedRest]) :-
  filter_not_multiple(First, Rest, FilteredRest),
  sieve(FilteredRest, SievedRest).

primes(Limit, Primes) :-
  numbers(Limit, Numbers), sieve(Numbers, Primes).
```

Constraint Logic Programming

- Extension of logic programming to include constraints on variables
- Basic Prolog includes limited arithmetic constraints that require variables to be instantiated

```
square_sum([N, X, Y, Z]) :-
    N =:= Z * Z, N =:= X * X + Y * Y,
    X > 0, Y > 0, Z > 0, X < Y, N < 1000.
```

```
?- square_sum(S).
ERROR: =:=/2: Arguments are not sufficiently
instantiated
```

CLP(FD)

- The CLP family of libraries provide constraint logic programming as extensions to Prolog
- CLP(FD) is included in SWI-Prolog and works on finite domains (integer subsets)

Import CLP(FD) module

CLP(FD) constraint operator

Require given variables to be *grounded*

```
:- use_module(library(clpfd)).

square_sum_c([N, X, Y, Z]) :-
    N #= Z * Z, N #= X * X + Y * Y,
    X #> 0, Y #> 0, Z #> 0, X #< Y, N #< 1000,
    label([N, X, Y, Z]).
```

```
?- square_sum_c(S).
S = [25, 3, 4, 5] ;
S = [100, 6, 8, 10] ;
S = [169, 5, 12, 13] ;
...
```

Search in CLP

- Search follows the same general strategy as Prolog, except that a *constraint store* keeps track of the set of constraints
  - Start with a set of goal terms
  - For first goal term, find a clause whose head can be unified with the term
    - Unification can instantiate or bind variables
  - Insert body terms that are not constraints into the front of the set of goal terms
  - Insert body terms that are constraints into the constraint store
  - Check whether the constraint store is unsatisfiable
    - If so, backtrack
- Search succeeds when no more goal terms remain, and the constraint store is not unsatisfiable

Example: Verbal Arithmetic

- Find a solution to the following such that each digit is distinct, and leading digits are non-zero:

```
     S E N D
+    M O R E
-----------
= M O N E Y
```

- Plain Prolog:

```
money([S, E, N, D, M, O, R, Y]) :-
    is_digit(S), is_digit(E), is_digit(N), is_digit(D),
    is_digit(M), is_digit(O), is_digit(R), is_digit(Y),
    S \= 0, M \= 0,
    S \= E, S \= N, S \= D, S \= M, S \= O, S \= R, S \= Y,
    E \= N, E \= D, E \= M, E \= O, E \= R, E \= Y,
    N \= D, N \= M, N \= O, N \= R, N \= Y,
    D \= M, D \= O, D \= R, D \= Y,
    M \= O, M \= R, M \= Y,
    O \= R, O \= Y,
    R \= Y,
                      1000 * S + 100 * E + 10 * N + D
    +                 1000 * M + 100 * O + 10 * R + E
    =:= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

Takes 90 seconds to solve on Macbook

Example: Verbal Arithmetic

- Find a solution to the following such that each digit is distinct, and leading digits are non-zero:

```
      S E N D
+     M O R E
-------------
= M O N E Y
```

- Prolog + CLP(FD):

```
money_c([S, E, N, D, M, O, R, Y]) :-
    L = [S, E, N, D, M, O, R, Y],
    L ins 0 .. 9, S #\= 0, M #\= 0, all_distinct(L),
                  1000 * S + 100 * E + 10 * N + D
    +             1000 * M + 100 * O + 10 * R + E
    #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
    label(L).
```

Takes 0.2 seconds to solve on Macbook

Require variables in L to be members of set [0, 9]

Constrain variables in L to have distinct values

Example: Sudoku

- Sudoku solver:

Higher-order predicate

Partial application

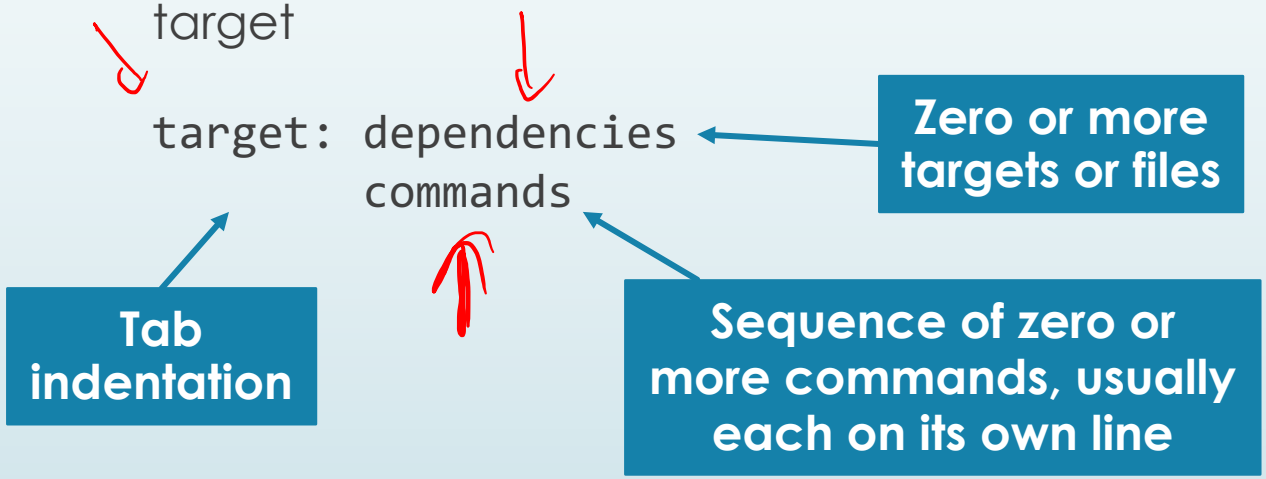
```
sudoku(Rows) :-
  length(Rows, 9), maplist(same_length(Rows), Rows),
  append(Rows, Values), Values ins 1..9,
  maplist(all_distinct, Rows),
  transpose(Rows, Columns),
  maplist(all_distinct, Columns),
  Rows = [Row1,Row2,Row3,Row4,Row5,Row6,Row7,Row8,Row9],
  blocks(Row1, Row2, Row3),
  blocks(Row4, Row5, Row6),
  blocks(Row7, Row8, Row9),
  maplist(label, Rows).

blocks([], [], []).
blocks([N1, N2, N3 | RestRow1],
       [N4, N5, N6 | RestRow2],
       [N7, N8, N9 | RestRow3]) :-
  all_distinct([N1, N2, N3, N4, N5, N6, N7, N8, N9]),
  blocks(RestRow1, RestRow2, RestRow3).
```

Break Time!

- We'll start again in five minutes.

Make

- Tool for automating the building of software packages, tracking dependencies between components
- Programming model is a combination of declarative and imperative
- A *rule* declares a relation between a target and its dependencies, specifies commands to build the target



Simple Example

- Rule contained within `Makefile`:

```
hello:
	@echo "Hello world!"
```

No dependencies

Build `hello` target

Build first target in `Makefile`

```
> make hello
echo "Hello world!"
Hello world!
> make
echo "Hello world!"
Hello world!
```

Target has no dependencies, so it will always build

Building an Executable

- More complex dependency trees can be specified

```
main: a.o b.o c.o
	g++ -o main a.o b.o c.o

a.o: a.cpp
	g++ --std=c++14 -Wall -pedantic -c a.cpp

b.o: b.cpp
	g++ --std=c++14 -Wall -pedantic -c b.cpp

c.o: c.cpp
	g++ --std=c++14 -Wall -pedantic -c c.cpp
```

```
> make
g++ --std=c++14 -Wall -pedantic -c a.cpp
g++ --std=c++14 -Wall -pedantic -c b.cpp
g++ --std=c++14 -Wall -pedantic -c c.cpp
g++ -o main a.o b.o c.o
```

Rebuilding a Target

- A target is only rebuilt when one of its dependencies has been modified

Modify timestamp on b.cpp

```
> touch b.cpp
> ls -l
-rw-r--r--  1 kamil  staff    229 Nov 17 01:01 Makefile
-rw-r--r--  1 kamil  staff     90 Nov 17 00:57 a.cpp
-rw-r--r--  1 kamil  staff   6624 Nov 17 01:01 a.o
-rw-r--r--  1 kamil  staff     31 Nov 17 01:12 b.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 b.o
-rw-r--r--  1 kamil  staff     33 Nov 17 00:58 c.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 c.o
-rwxr-xr-x  1 kamil  staff  15268 Nov 17 01:01 main
> make
g++ --std=c++14 -Wall -Werror -pedantic -c b.cpp
g++ -o main a.o b.o c.o
```

Example: Makefile for Notes

```
all: foundations functional theory data declarative

foundations: foundations.html foundations.tex

functional: functional.html functional.tex

theory: theory.html theory.tex

data: data.html data.tex

declarative: declarative.html declarative.tex

asynchronous: asynchronous.html asynchronous.tex

metaprogramming: metaprogramming.html metaprogramming.tex

%.html: %.rst
        rst2html.py --stylesheet=../style/style.css $< > $@

%.tex: %.rst
        rst2latex.py --stylesheet=../style/style.sty $< > $@
        pdflatex $@
        pdflatex $@
        pdflatex $@

clean:
        rm -vf *.html *.tex *.pdf *.aux *.log *.out
```

Not currently built

Pattern rule

Build PDF file

Dependencies

Target