Logic Programming and Prolog

R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

Slides at **https://github.com/mvolkmann/talks/**

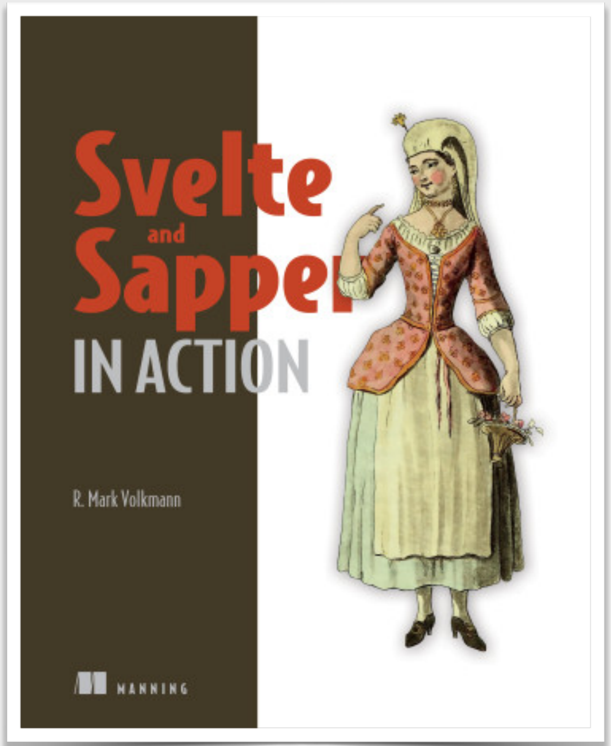
OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Thanks!

- Many thanks to **Dr. Markus Triska**
  for taking the time to provide detailed answers to my many questions
- Professor at Vienna University of Technology, Austria
  - Vienna Center for Logic and Algorithms (VCLA)
- https://www.metalevel.at/
- **The Power of Prolog**
  - https://www.metalevel.at/prolog
- **The Power of Prolog: Videos**
  - https://www.metalevel.at/prolog/videos/

About Me

- Partner and Distinguished Software Engineer
  at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and teacher
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte ... in Action”



Logic Programming

- Very different from imperative, functional, and object-oriented programming
- Focuses on describing **what** is true, not **how** to do something
- Examples include
  - **Prolog** (most popular)
  - **DataLog**
  - **Answer Set Programming** (ASP)
- Use cases
  - explainable AI (as opposed to approaches like neural networks), Natural Language Processing (NLP), parsing, problem solving, rule-based systems, and more

NLP includes automatic translation from one human language to another. It also includes translating human language to direct a computer and possibly generating a human language response (ex. Alexa and Siri).

Prolog Overview

- Name is contraction of “Programming in Logic”
- Provides a search engine over facts and rules
- Initially developed by Alain Colmerauer and Philippe Roussel in 1972
- Two dialects
  - Edinburgh Prolog - 1972
  - ISO Prolog - 1995
- Many current implementations strive to conform to the ISO standard, but also add features

Popular Implementations

- **Scryer Prolog** - https://www.scryer.pl/
  - implemented in Rust (64%) and Prolog (36%)
- **SICStus Prolog** - https://sicstus.sics.se/
  - high-performance, commercial implementation
- **GNU Prolog** - http://www.gprolog.org/
  - implemented in C (84%) and Prolog (15%)
- **SWI-Prolog** - https://www.swi-prolog.org/
  - implemented in C (48%) and Prolog (39%)
- **Ciao** - https://ciao-lang.org
  - implemented in Prolog (72%) and C (23%)
- **Tau Prolog** - http://tau-prolog.org/
  - implemented in JavaScript (95%) and Prolog (5%)

There are MANY MORE implementations!

Challenge

- One thing that makes learning Prolog challenging is that it defines a large number of built-in operators and predicates
  - getting proficient at learning these takes a considerable amount of practice
- Another challenge is choosing and learning a specific implementation

Running

- Can feed a Prolog source file to a command-line tool that typically provides a "**top level**" (REPL) where compiler directives and queries can be entered
  - in Scryer Prolog, enter **`scryerp [file-name.pl]`** — **`scryerp`** is an alias you can define
  - to load source file into top level, enter **`[f].`** or **`consult(f).`** — ***f*** is file name or file path in quotes
  - to reload a source file after modifying it, load again
    - in SWI-Prolog, can enter **`make.`** to reload all loaded source files
  - to exit, press ctrl-d or enter **`halt.`**
- Can be compiled to abstract machine code and executed in a virtual machine
  - many implementations use the Warren Abstract Machine (WAM) instruction set

Comments

- Single-line comments
  - begin with `%` and extend to end of line
- Multi-line comments
  - surrounded by `/*` and `*/`

Atoms and Variables

- **Atoms** - symbols treated like string constants
  - start with lowercase letter **OR** any text wrapped in single quotes
  - ex. `a`, `apple`, or `'I like apples!'` (not typically used)
- **Variables** - represent a value to be determined
  - start with uppercase letter or underscore (ex. `Fruit` or `_fruit`)
  - often a single letter or followed by `s` for plural (ex. `C` or `Cs`) when meaning is clear from context

Predicates: Facts and Rules

- **Facts** state things that are always true (hold)

```
female(amanda).
female(tami).

male(clarence).
male(jeremy).
male(mark).
male(richard).

father(clarence, tami).
father(richard, mark).
father(mark, amanda).
father(mark, jeremy).

mother(tami, amanda).
mother(tami, jeremy).
```

Functor names are `female`, `male`, `father`, `mother`, and `grandfather`.

Functor names are typically followed by a list of **arguments** in parentheses.

Facts, rules, and queries all **end with a period.**

A functor name is also referred to as a "**principal functor**".

- **Rules** state things that are conditionally true
- **Rules** have a head and body separated by `:-` read as "if"
- **Rules** "relate" their LHS to RHS

```
grandfather(X, Y) :-
  male(X),
  father(X, Z),
  (father(Z, Y); mother(Z, Y)).
```

Comma operator forms a **conjunction** and is read as "and".
Semicolon operator forms a **disjunction** and is read as "or".

Queries (aka Questions)

- Use to **test**, **complete**, or **generate** solutions
  - referred to as “working in multiple directions” or having “multiple usage modes”

```
% Test
?- grandfather(clarence, amanda).
true.
?- grandfather(mark, jeremy).
false.

% Complete
?- grandfather(X, jeremy).
X = clarence ;
X = richard ;
false.
?- grandfather(richard, X).
X = amanda ;
X = jeremy ;
false.
```

means no more solutions were found

```
% Generate
?- grandfather(X, Y).
X = clarence,
Y = amanda ;
X = clarence,
Y = jeremy ;
X = richard,
Y = amanda ;
X = richard,
Y = jeremy ;
false.
```

Solutions are generated lazily, one at a time.

Press semicolon key to see next solution or period key to stop searching.

Primitive Types ...

- **Boolean**
  - represented by built-in predicates `true` and `false`
- **Number**
  - supports integers and floating point numbers, optionally using exponential notation

... Primitive Types

- **String**
  - three representations: atom, list of character atoms, or list of ASCII codes
  - delimiters

    use list predicates to operate on these lists

    - **single quotes** -> atom
    - **double quotes** -> depends on **`double_quotes`** compiler flag
      - **`atom`**: "**`apple`**" becomes atom **`apple`**
      - **`chars`**: "**`apple`**" becomes list of character atoms **`[a,p,p,l,e]`**

        recommended setting; default in Scryer Prolog
      - **`codes`**: "**`apple`**" becomes list of ASCII codes **`[97,112,122,108,101]`**
      - **`string`**: "**`apple`**" remains string **`"apple"`** (SWI-Prolog-only) non-standard
    - **backticks** -> list of ASCII codes (SWI-Prolog-only) non-standard

Structure

- Named collection of values
  - ex.
    ```
    fruit(apple, red)
    fruit(banana, yellow)
    ```
  - can think of **`fruit`** as the structure type
- Can nest
  - ex.
    ```
    person(
      name("Mark", "Volkmann"),
      address(
        "123 Some Street",
        "Somewhere", "MO", 12345
      )
    )
    ```

Collections - List

- List of values of any type inside square brackets
  - example: `Fruits = [apple, banana, cherry]`
- Can destructure into head and tail
  - using = operator
    `[H|T] = Fruits`

    `H` is `apple`
    `T` is `[banana, cherry]`
  - in rule argument

```
write_all([]).
write_all([H|T]) :- writeln(H), write_all(T).
```

`writeln` is a built-in predicate in SWI-Prolog.
In other implementations can be defined as
`writeln(X) :- write(X), nl.`

List append

- A built-in predicate, but we could write it
- Note all the usage modes

```
% Appending an empty list to
% any list gives the second list.
append([], L, L).

% Appending two lists is the same as
% appending head of first list (H)
% to result of
% appending tail of first list (L1)
% to second list (L2).
append([H|L1], L2, [H|L3]) :-
  append(L1, L2, L3).
```

```
% Is the result of appending
% two lists a given result list?
?- append([1, 2], [3, 4], [1, 2, 3, 4]).
true.

% What is the result of
% appending two lists?
?- append([1, 2], [3, 4], X).
X = [1, 2, 3, 4].

% What list must be appended to a
% given list to obtain a given result?
?- append([1, 2], X, [1, 2, 3, 4]).
X = [3, 4].

% What list must be prepended to a
% given list to obtain a given result?
?- append(X, [3, 4], [1, 2, 3, 4]).
X = [1, 2]
```

```
% What lists can be appended
% to obtain a given result?
?- append(X, Y, [1, 2, 3, 4]).
X = [],
Y = [1, 2, 3, 4] ;
X = [1],
Y = [2, 3, 4] ;
X = [1, 2],
Y = [3, 4] ;
X = [1, 2, 3],
Y = [4] ;
X = [1, 2, 3, 4],
Y = [] ;
false.
```

More Builtin List Predicates

- **`findall`** - collects all solutions in a list
- **`foldl`** - like reduce in JavaScript
- **`list_max`** and **`list_min`**
- **`maplist`** - like map in JavaScript
- **`member`** - like includes in JavaScript
- **`reverse`**
- **`select`** - relates a list to another list with a given element removed
- **`sum_list`**

All these examples assume `Numbers = [1, 2, 3].`

```
findall(X, grandfather(richard, X), L).
% L = [amanda, jeremy]

add(A, B, C) :- C #= A + B.
foldl(add, Numbers, 0, Sum). % Sum = 6

list_max(Numbers, Max). % Max = 3

list_min(Numbers, Min). % Min = 1

Fruits = ["apple", "banana", "cherry"],
maplist(length, Fruits, Ls). % Ls = [5, 6, 6]

reverse(Numbers, R). % R = [3, 2, 1]

select(2, Numbers, L). % L = [1, 3]

sum_list(Numbers, Sum). % Sum = 6
```

Collections - Pair

- Key and value separated by a dash
- **`pairs`** library provides predicates that operate on pairs

list of pairs

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

pairs_keys([a-apple, b-banana, c-cherry], Ks).
% Ks = [a, b, c]).

pairs_values([a-apple, b-banana, c-cherry], Vs).
% Vs = [apple, banana, cherry]).

pairs_keys_values([a-apple, b-banana, c-cherry], Ks, Vs).
% Ks = [a, b, c]), Vs = [apple, banana, cherry]).

% The pairs must be sorted on their keys.
group_pairs_by_key([a-apple, a-ape, b-banana, b-bear], Gs).
% Gs = [a-[apple,ape], b-[banana,bear]]).

first_letter(Atom, Letter) :- atom_chars(Atom, [Letter|_]).
map_list_to_pairs(first_letter, [apple, banana, cherry], Ps).
% Ps = [a-apple, b-banana, c-cherry]).
```

Assoc

- Key/value pairs implemented as an AVL tree (balanced, binary)
- Supported by **assoc** library, not ISO standard
  - https://www.scryer.pl/assoc.html
- Highlights

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

demo :-
  empty_assoc(A0),
  put_assoc(name, A0, 'Mark', A1),
  get_assoc(name, A1, Name),
  write(Name), nl, % Mark
  del_assoc(name, A1, _, A2).
  % A2 does not contain name key.
```

The third argument can be a variable that gets set to the value of the key/value pair that is deleted.

Specific to SWI-Prolog and not in ISO standard

Collections: Dict

- A dictionary or hash map
- To create
  `Car = car{ make: "MINI", model: "Cooper", color: orange }`
  - **`car`** above is called the "tag"; can think of as the type
- To get a key value
  `Color = Car.get(color) % orange`
- To get a key value with default value to use when missing
  `Color = Car.get(wrong, black) % black`
- To create a new dict with a modified key value
  `Car2 = Car.put(color, yellow), Color = Car2.get(color) % yellow`
- Many built-in predicates operate on dicts

More Terminology

- **term** - a **number**, **atom**, **variable**, or **compound term**
- **compound term** - combination of basic terms (aka **structure**), arguments can be compound terms to nest them
- **goal** - compound term that appears in a rule body or query
- **clause** - a single fact or rule
- **predicate** - defined by a collection of clauses (logical alternatives) that all have the same name (aka **principal functor**)
- **knowledge base** - collection of facts and rules (aka database)

Help

- In SWI-Prolog
  - for basic help on all predicates that contain given text in their name or description, enter **`apropos(`*`text`*`).`**
  - for more detailed help on a specific predicate, enter **`help(`*`name`*`).`**
  - for listing of all defined predicates, enter **`listing.`**
  - for listing of clauses for a given predicate name, enter **`listing(`*`name`*`).`**

Functors

- Way to refer to a predicate using its **functor name** and **arity** (number of arguments) separated by a slash
- For example, earlier we saw these functors
    - **`female/1`**, **`male/1`**, **`father/2`**, **`mother/2`**, and **`grandfather/2`**
- Functors often appear in error messages

Unification

- Process of finding variable values that cause goals to hold
  - can use = and `#=` operator
    - `X = 7` is unification not assignment; `7 = X` is the same
    - `X #= Y * 2, Y in 1..5, label([X, Y]).`

      `Y * 2 #= X` is the same
  - can use single predicate with variables
    - ex. `grandfather(X, Y)`
  - can use conjunction of predicates with variables
    - ex. `grandfather(X, jeremy), father(X, Y)`

      finds person `X` that is grandfather of `jeremy` and father of `Y`
    - finds `X = clarence, Y = tami` and `X = richard, Y = mark`
- Default search strategy is “depth-first with chronological backtracking”

Choicepoints

- Occur when multiple clauses of a predicate match
  - ex. `mother(tami, Child)` matches multiple values for `Child`
- Prolog selects first match, marks it as a choicepoint, and continues evaluating subsequent goals
- After finding a solution or failing to find a solution, if there are remaining clauses to select then Prolog **backtracks**
  - changes choicepoint to next matching clause after current choicepoint
  - evaluates subsequent goals using that selection

Supported Notations

- **Function notation**: `functor-name(arg1, arg2)`
  - ex. `father(F, jeremy)`
- **List notation**: `[functor-name, arg1, arg1]`
  - useful for dynamically creating goals
  - ex. `P = jeremy, Goal =.. [father, F, P], call(Goal). % F = mark`
- **Operator notation**
  - operators can be prefix, infix, or postfix
  - common arithmetic operators are infix
  - ex. `Y = 3, X is Y * 2`. is the same as `Y = 3, X is *(Y, 2)`.

custom operators can be defined

operator notation

function notation

Compiler Directives

- Several uses, all starting with `:-` prefix operator
- **Set flag** that tells compiler to interpret code differently
  - ex. `:- set_prolog_flag(double_quotes, chars).`
- **Include another source file**
  - ex. `:- include(util).`

    includes contents of file `util.pl`
- **Import a library**, making its predicates and operators available
  - ex. `:- use_module(library(`*`name`*`)).`

    To load a library from a top level, enter `[library[`*`name`*`]).`
- **Evaluate goals when file is loaded**
  - ex. `:- initialization(`*`some-goal`*`)`

Debugging Options

- Can experiment with predicates in top level
- Can use **`format`** predicate for print-style debugging
  - ex. **`format('~w has ~w points.~n', [Name, Score]).`**
  - remember **`[ ]`** around values!
  - many more supported placeholders, but **`~w`** and **`~n`** are most common
- Can step through each goal evaluation
  - enter **`trace.`**
  - enter a query
  - press spacebar after each output
- Can write unit tests

Unit Tests

- SWI-Prolog supports unit tests
- Convention is to use `.plt` file extension
- Run with **`swipl`** ***`file-name`*****`.plt`**

family.pl

facts omitted

```
grandfather(X, Y) :-
  male(X),
  father(X, Z),
  (father(Z, Y); mother(Z, Y)).

is_father(X) :- father(X, _).

is_mother(X) :- mother(X, _).

is_son(X) :- male(X), (father(_, X); mother(_, X)).

sibling(X, Y) :-
  dif(X, Y), % can't be sibling of self
  father(F, X), father(F, Y),
  mother(M, X), mother(M, Y).

sister(X, Y) :-
  dif(X, Y), % can't be sister of self
  female(X), sibling(X, Y).
```

family.plt

```
:- consult(family).
:- begin_tests(family).

test(grandfather) :-
  grandfather(clarence, jeremy),
  grandfather(clarence, amanda),
  grandfather(richard, jeremy),
  grandfather(richard, amanda), !.

test(is_father) :-
  is_father(clarence),
  is_father(richard),
  is_father(mark), !.

test(is_mother) :-
  is_mother(gerri),
  is_mother(judi),
  is_mother(tami), !.

test(is_son) :-
  is_son(jeremy),
  is_son(mark), !.

test(sibling) :-
  sibling(amanda, jeremy).

test(sister) :-
  sister(amanda, jeremy),
  \+ sister(amanda, amanda).

:- end_tests(family).
:- run_tests.
:- halt.
```

Cuts prevent warning "Test succeeded with choicepoint".

Knowledge Base Changes ...

- Can modify knowledge base clauses
- To **enable changes** to a given functor, use **`dynamic(`*`functor`*`)`**

  ex. `dynamic(father/2).`

- To add a clause **before** all for same functor, use **`asserta(`*`clause`*`)`**

  ex. `asserta(father(richard, laura)).`

- To add a clause **after** all for same functor, use **`assertz(`*`clause`*`)`**

  ex. `assertz(father(richard, pam)).`

- To **remove** a clause, use **`retract(`*`clause`*`)`**

  ex. `retract(father(richard, mark)).`

- To **remove all** clauses for a functor, use **`retractall(`*`head`*`)`**

  ex. `retractall(father(richard, _)).`

... Knowledge Base Changes

```
fruit(apple).
fruit(banana).

person(mark, 74).
person(tami, 65).

tall(N) :-
  person(N, H),
  H >= 72.

add_predicates :-
  % Dynamically add a fruit fact.
  dynamic(fruit/1),
  assertz(fruit(cherry)),

  % Dynamically add a tall fact.
  dynamic(tall/1),
  assertz(tall(giraffe)),

  % Dynamically add a rule.
  dynamic(sum/3),
  assertz(sum(X, Y, Z) :- Z is X + Y).
```

Rules succeed or fail. They do not return a value.

There is no reason to add this particular rule dynamically, but this demonstrates that it is possible.

```
report(Thing) :-
  ( tall(Thing) ->
    report_(Thing, 'tall')
  ; report_(Thing, 'not tall')
  ).

report_(Thing, X) :-
  format('~w is ~w.~n', [Thing, X]).

:- initialization
  add_predicates,

  findall(F, fruit
  writeln(Fruits),


  Things = [mark, tami, gi
  maplist(report, Things),

  sum(2, 3, S
  writeln(S),
  halt.
```

Auxiliary rule names end in an underscore by convention.

`findall` gathers all solutions into a list

```
[apple,banana,cherry]
```

```
mark is tall.
tami is not tall.
giraffe is tall.
```

```
5
```

Input

- By default, input is read from stdin
- Can specify input stream associated with file or network connection
- Stream aliases
  - **`user_input`** and **`current_input`** can change default bindings
- Many predicates read input
  - **`read(Term)`** - reads any Prolog term; user must end with period AND press return
  - **`read_term(Term, Options)`** - like read, but configurable
  - **`read_line_to_string(Stream, S)`** - reads any text; user can just press return SWI-Prolog-only
  - **`get_line_to_chars(Stream, S)`** - similar to above; defined in Scryer **`charsio`** library Scryer Prolog-only
  - a few others not defined by ISO standard

Output

- By default, output is written to stdout
- Can specify output stream associated with file or network connection
- Stream aliases
  - **`user_output`**, **`user_error`**, and **`current_output`**

can change default bindings

- Many predicates write output
  - **`write(Term)`** - writes a Prolog term
  - **`nl`** - writes a newline character
  - **`writeln(Term)`** - combines **`write`** and **`nl`**; SWI-Prolog-only
  - **`format(Format, Arguments)`** - **`format('~w, ~w!~n', ['Hello', 'World'])`**
  - and a few more

Many placeholders other than **`~w`** and **`~n`** are supported, but these are the most useful.

Constraint Logic Programming (CLP)

- Uses constraint propagation to solve problems in specific domains
- Different library for each supported domain
  - integers (**`clpfd`**) in Scryer Prolog, use **`clpz`** library
  - booleans (**`clpb`**)
  - rational numbers (**`clpq`**)
  - floating point numbers (**`clpr`**)
- Each library provides additional Prolog operators and predicates
- Include with directive like

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

CLP Operators

- CLP operators include
  - equal **#=**, not equal **#\=**
  - greater **#>**, greater or equal **#>=**
  - less **#<**, less or equal **#=<**
  - **in** (single value), **ins** (list of values) constrain variables to a given range

```
N = 3, N in 1..5. % N = 3 (success)
L = [2, 4, 7], L ins 1..5. % false
```

- CLP operators support multiple usage modes
  - for example, with **is** operator, RHS variables must be instantiated

```
add1(A, B, C) :- C is A + B.
```

  - **#=** operator does not have this restriction

```
add2(A, B, C) :- C #= A + B.
```

CLP Predicate Highlights

There are many more CLP predicates not described here.

- Enumeration
  - **labeling** and **label** (examples on next slide)
  - **label** is the same as **labeling** with no options
- Constraint predicates
  - **all_distinct** - all elements in a list have distinct values

```
L = [2, 4, 2],
all_distinct(L). % false
```

  - **global_cardinality** - relates a list to a list of pairs
    - in each pair, the key is a value from the list and its value is a count
    - both arguments must be instantiated

```
Vs = [2, 4, 2, 3, 2, 4],
global_cardinality(Vs, [2-3, 3-1, 4-2]). % true
```

CLP Labeling

- Finds concrete solutions for abstract solutions

```
X in 5..10, Y in 7..14, X #> Y.
% Without labeling, this outputs the following
% without giving specific values for X and Y.
% X in 8..10,
% Y in 7..9

X in 5..10, Y in 7..14, X #> Y., label([X, Y]).
% With labeling, this gives
% specific combinations of X and Y values.
% X = 8, Y = 7 ;
% X = 9, Y = 7 ;
% X = 9, Y = 8 ;
% X = 10, Y = 7 ;
% X = 10, Y = 8 ;
% X = 10, Y = 9.
```

Definite Clause Grammar (DCG)

- Set of grammar rules of form `GRHead --> GRBody`
- Not yet part of ISO standard, but may be added
- Enabled with `:- use_module(library(dcgs)).`
    - enabled by default in SWI-Prolog
- Name in head typically describes allowed sequences
- Body uses
    - `,` operator for concatenation - read as “and then” or “followed by”
    - `|` operator for alternatives - read as “or”
    - `{ prolog-predicates }` switches from DCG syntax to standard Prolog syntax and can set variables used in containing DCG rule

DCGs in Scryer Prolog

- Parse lines like “Player Gretzky wears number 99.”
- Extract player name and number

```
player(Name, Number) -->
  "Player ",
  seq(Name),
  " wears number ",
  % seq(Number),
  integer(Number),
  ".".
```

```
?- once(phrase(
     player(Name, Number),
     "Player Gretzky wears number 99."
   )).
Name = "Gretzky", Number = 99
```

DCG rule bodies contain lists, DCG operators, and DCG predicates.

```
% This matches any single digit.
digit(D) --> [D], { char_type(D, decimal_digit) }.

% This matches any non-empty list of digits.
digits([D|Ds]) --> digit(D), digits_(Ds).

% This matches any list of digits including an empty list.
digits_([D|Ds]) --> digit(D), digits_(Ds).
digits_([]) --> [].

% This matches any non-empty list of digits
% AND converts it to an integer.
integer(I) --> digits(Ds), { number_chars(I, Ds) }.
```

`seq` , **`seqq`**, and **`...`** are defined by the Scryer library **`dcg`**.
`char_type` is defined by the Scryer library **`charsio`**.

Unfair Enumeration

- Some queries have an infinite number of solutions
- In an unfair enumeration there are solutions that will never be output
- The **`length`** predicate can be used to advance multiple variables together rather than one at a time
  - called “**iterative deepening**”
- Example

```
as --> "a", as_.
as_ --> [] | as.

bs --> "b", bs_.
bs_ --> [] | bs.

as_and_bs --> as, bs.
```

```
?- phrase(as_and_bs, Cs).
   Cs = "ab"
;  Cs = "abb"
;  Cs = "abbb"
% and more!
```

will never output solutions beginning with more than one **a**

```
?- length(Cs, _), phrase(as_and_bs, Cs).
   Cs = "ab"
;  Cs = "abb"
;  Cs = "aab"
;  Cs = "abbb"
;  Cs = "aabb"
;  Cs = "aaab"
;  Cs = "abbbb"
% and more!
```

outputs all solutions of a given length before outputting solutions with the next larger length

DCGs for Compiling

- DCGs can be used to implement a parser for a programming language
- Let's create one that can ...
  - use single-line comments
  - evaluate basic math expressions
  - assign to variables
  - define and call functions that return a value
  - print values
- Variable values are stored in a stack of hash maps
  - top scope and one per current function scope

```
# This function adds two integers.
fn add(n1, n2)
  sum = n1 + n2
  return sum
end

# This function multiplies two integers.
fn multiply(n1, n2)
  return n1 * n2
end

v = add(2, 3) + multiply(3, 4)
print v # 17

print add(2, 3) + multiply(3, 4) # 17
```

This is code in my made-up programming language.

Abstract Syntax Tree (AST)

- Nested Prolog structures can be used to represent an AST as a single Prolog term
- DCGs can be used to parse custom syntax and generate such an AST
- A Prolog "compiler" program can parse custom syntax and write AST to a text file on a single line

Key:
a = assignment
c = function call
f = function definition
k = constant
m = math expression
p = print
pr = program
r = return

`pr` argument is a list of statements

Spaces and newlines were added manually to make this more readable.

```
pr([
  f(add, [n1,n2], [
    a(sum, m(+,n1,n2)),
    r(sum)
  ]),
  f(multiply, [n1,n2], [
    r(m(*,n1,n2))
  ]),
  a(v, m(+,
    c(add,[k(2),k(3)]),
    c(multiply,[k(3),k(4)])
  )),
  p(v),
  p(m(+,
    c(add,[k(2),k(3)]),
    c(multiply,[k(3),k(4)])
  ))
]).
```

DCG AST Generator

highlights of code from
`lim_compile_scryer.pl`

```
assign(a(I, V)) --> id(I), ws, "=", ws, value2(V).
comment([]) --> "#", to_eol(_).
constant(k(V)) --> integer(V).
fn_call(c(Name, Args)) --> id(Name), "(", call_args(Args), ")".
fn_def(f(Name, Args, Stmts)) -->
  "fn ", id(Name), "(", def_args(Args), ")", ws, eol,
  statements(Statements), ws, "end".
math(m(Op, V1, V2)) --> value1(V1), ws, operator(Op), ws, value1(V2).
print(p(V)) --> "print", ws, value2(V).
program(pr(Stmts)) --> statements(Stmts).
return(r(V)) --> "return ", value2(V).

statement(S) --> assign(S) | comment(S) | fn_call(S) | fn_def(S) | print(S) | return(S).
statement_line([]) --> ws, eol.
statement_line(S) --> ws, statement(S), ws, eol.
statements(Stmts) -->
  statement_line(S),
  % This avoids including empty lists from comments and blank lines.
  { S == [] -> Stmts = []; Stmts = [S] }.
statements(Stmts) -->
  statement_line(S), statements(Ss),
  % This avoids including empty lists from comments and blank lines.
  { S == [] -> Stmts = Ss; Stmts = [S|Ss] }.
```

start here!

matches when there is only one statement

matches when there are multiple statements

DCG AST Runner

highlights of code from
lim_run_scryer.pl

```
% Skip empty statements from blank lines and comments.
eval([]).

% This assigns a value to a variable.
eval(a(Name, Value)) :- lookup(Value, V), vtables_put(Name, V).

% This calls a function, but does not use its return value.
eval(c(Name, Args)) :- process_call(Name, Args).

% This stores a function definition in the current vtable.
eval(f(Name, Params, Stmts)) :- vtables_put(Name, [Params, Stmts]).

% This evaluates all the statements in a program.
eval(pr(Stmts)) :- maplist(eval, Stmts).
```

start here!

```
% This prints a value to stdout.
eval(p(Value)) :- lookup(Value, V), writeln(V).

% This stores a value being returned from a function
% so the caller can find it.  See "lookup(c...) below."
eval(r(Value)) :-
  lookup(Value, V),
  % Store the return value so caller can retrieve it.
  bb_put(return_, V).
```

```
% This calls a function and uses its return value.
lookup(c(Name, Args), V) :-
  process_call(Name, Args),
  bb_get(return_, V).

% This gets the value of a constant.
lookup(k(Value), Value).

% This evaluates a math expression.
lookup(m(Operator, LHS, RHS), Result) :-
  lookup(LHS, L),
  lookup(RHS, R),
  ( Operator == (+) ->
    Result is L + R
  ; Operator == (-) ->
    Result is L - R
  ; Operator == (*) ->
    Result is L * R
  ; Operator == (/) ->
    Result is L / R
  ; writeln('lookup math: Operator not matched'),
    fail
  ).

% This gets a value from the vtables.
lookup(Name, Value) :- vtables_get(Name, Value).
```

Jugs Problem ...

- Have three jugs a, b, and c with capacities 4, 3, and 7
- Start with a empty, b empty and c full
- Can pour water from any non-empty jug to any non-full jug
- When we do, we just pour as much as possible
- Goal: find sequence of pours that result in a jug containing 2 units

The solution requires 4 moves.
Starting state is 0, 0, 7.
Pour from c to b. 0, 3, 4
Pour from b to a. 3, 0, 4
Pour from c to b. 3, 3, 1
Pour from b to a. 4, **2**, 1

- **`jug`** structure arguments are label, capacity, and current level
- **`from_to`** structure arguments are “from jug number” and “to jug number”
- Solution is list of **`from_to`** structures

... Jugs Problem

```
:- use_module(library(clpz)).
:- use_module(library(dcgs)).
:- use_module(library(format)).
:- use_module(library(lists)).

% A solution has been found when any jug contains 2 units.
% Jugs is a list of three jug structures.
moves(Jugs) --> { member(jug(_, _, 2), Jugs) }.

moves(Jugs0) -->
  [from_to(From, To)],
  {
    select(jug(From, FromCapacity, FromFill0), Jugs0, Jugs1),
    select(jug(To, ToCapacity, ToFill0), Jugs1, Jugs),

    % Calculate the number of units that can be moved
    % from the From jug to the To jug.
    Amount #= min(FromFill0, ToCapacity - ToFill0),

    % Calculate the new amount in the From jug.
    FromFill #= FromFill0 - Amount,

    % Calculate the new amount in the To jug.
    ToFill #= ToFill0 + Amount
  },
  moves([
    jug(From, FromCapacity, FromFill),
    jug(To, ToCapacity, ToFill) | Jugs
  ]).
```

Scryer Prolog places all non-ISO predicates in libraries that must be explicitly included.

It is a convention to end variable names that represent an **initial state** with zero.

The **`select`** predicate takes an element and two lists. It succeeds when the last list matches the first list with the element removed.

```
print_move(from_to(F, T)) :-
  format("Pour from ~w to ~w.~n", [F, T]).

:- initialization((
   length(Moves, L), % for iterative deepening
   phrase(moves([jug(a,4,0), jug(b,3,0), jug(c,7,7)]), Moves),
   format("The solution requires ~d moves.~n", [L]),
   maplist(print_move, Moves),
   halt
 )).
```

Sudoku ...

```
:- use_module(library(clpz)).
:- use_module(library(format)).
:- use_module(library(lists)).

sudoku(Rows) :-
  % Verify that Rows is a list with 9 elements.
  length(Rows, 9),

  % Verify that all elements are lists
  % with the same length as Rows which is 9.
  maplist(same_length(Rows), Rows),

  % Create a flattened list of all the values (Vs),
  % and verify that all elements in Vs
  % are a number in the range 1 to 9.
  append(Rows, Vs), Vs ins 1..9,

  % Verify that all element values in all rows
  % are unique within their row.
  maplist(all_distinct, Rows),

  % Create a list of lists that represent the columns.
  transpose(Rows, Columns),

  % Verify that all element values in all columns
  % are unique within their column.
  maplist(all_distinct, Columns),

  % Assign a variable name to each of the 9 rows.
  [R1, R2, R3, R4, R5, R6, R7, R8, R9] = Rows,

  % Verify that the element values in every 3x3 block
  % are unique within their block.
  blocks(R1, R2, R3),
  blocks(R4, R5, R6),
  blocks(R7, R8, R9).
```

... Sudoku

```
% When a block is empty, its element values
% (which are none) can be considered unique.
blocks([], [], []).

% When a block is not empty, get its 9 values
% and verify that they are unique.
blocks(
  [R1C1,R1C2,R1C3|T1],
  [R2C1,R2C2,R2C3|T2],
  [R3C1,R3C2,R3C3|T3]) :-
  all_distinct([R1C1, R1C2, R1C3, R2C1, R2C2,
R2C3, R3C1, R3C2, R3C3]),
  blocks(T1, T2, T3).

% When there a no more rows, stop printing.
print_rows([]).

% When there are more rows, print the first row.
print_rows([H|T]) :-
  print_row(H),
  print_rows(T).

% When the last element of a row
% has been printed, print a newline.
print_row([]) :- nl.

% When there are more row elements,
% print the first one followed by a space.
print_row([H|T]) :-
  format("~w ", [H]),
  print_row(T).
```

```
% Each puzzle must contain at least 17 clues.
problem(1,
  [[_,_,_, _,_,_, _,_,_],
   [_,_,_, _,_,3, _,8,5],
   [_,_,1, _,2,_, _,_,_],

   [_,_,_, 5,_,7, _,_,_],
   [_,_,4, _,_,_, 1,_,_],
   [_,9,_, _,_,_, _,_,_],

   [5,_,_, _,_,_, _,7,3],
   [_,_,2, _,1,_, _,_,_],
   [_,_,_, _,4,_, _,_,9]]).

  [_,_,_, _,_,_, _,_,_]]).

:- initialization((
  problem(1, Rows),
  sudoku(Rows),
  print_rows(Rows),
  halt
)).
```

Scryer Prolog From Other Languages

- **Rust**
  - ongoing work to enable this is described at https://github.com/mthom/scryer-prolog/pull/1880
- **HTTP Server**
  - see https://www.scryer.pl/http/http_server.html

SWI-Prolog From Other Languages

- **C**
  - SWI-Prolog provides a “Foreign Language Interface”
    that allows C to call Prolog and Prolog to call C
  - https://www.swi-prolog.org/pldoc/man?section=calling-prolog-from-c
- **JavaScript**
  - use npm package **`swipl`**, to call Prolog from JavaScript

    described in my Prolog blog page
    - see **`swipl.call`** and **`swipl.query`**
- **HTTP Server**
  - SWI-Prolog has built-in predicates that
    - start an HTTP server
    - load predicates from Prolog source files
    - register routes
    - respond to HTTP GET requests with HTML generated from query results

Wrap Up

- Prolog **requires thinking about problems differently**, describing what is true instead of how to do things
- Prolog is **not a fit for most applications**, but it is ideal for certain kinds of problems
- Prolog can be **used for specific portions of an application**, while using other programming languages for the remainder

Resources

- Read “**The Power of Prolog**”
  by Markus Triska from Vienna, Austria
  at https://www.metalevel.at/prolog
- Watch corresponding YouTube videos
  at https://www.metalevel.at/prolog/videos/
- Read my Prolog blog page at https://mvolkmann.github.io/blog/
  - click “Prolog” in hamburger menu