# Introduction to Pure Prolog and Meta-logical Predicates

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

November 10, 2021

# Table of Contents

# Table of Contents

You might be wondering, "haven't we been learning Prolog this whole time? Why is this lecture titled, 'Introduction to Pure Prolog?"'

Yes, we have been learning Prolog this whole time. You can run the examples shown thus far in this course using SWI-Prolog or other Prolog implementations.

But, you have been learning a small, core subset of Prolog.

In this lecture, we will cover additional pure Prolog features, such as arithmetic, and we will also cover Prolog's execution model.

# Table of Contents

Let's recall the resolution algorithm we studied in our last lesson:

# Resolution with Unification

**Input:** A goal $G$ and a program $P$

**Output:** An instance of $G$ that is a logical consequence of $P$, or *no* otherwise

**Algorithm:** Initialize the resolvent to $G$.
*while* the resolvent is not empty *do*
choose a goal $A$ from the resolvent
choose a (renamed) clause $A' \leftarrow B_1,\ldots,B_n$ from $P$
such that $A$ and $A'$ unify with mgu $\theta$
(if no such goal and clause exist, exit the *while* loop)
replace $A$ by $B_1,\ldots,B_n$ in the resolvent
apply $\theta$ to the resolvent and to $G$
*If* the resolvent is empty, *then* output $G$, *else* output *no*.

**Figure 4.2** An abstract interpreter for logic programs

Figure: Resolution algorithm with unification [Sterling and Shapiro 1994, p. 93]

There are two decisions that Prolog implementations need to make:

There are two decisions that Prolog implementations need to make:

1. How to choose a goal $A$ from the resolvent.

There are two decisions that Prolog implementations need to make:

1. How to choose a goal $A$ from the resolvent.
2. How to replace $A$ with $A' \leftarrow B_1, ..., B_n$ from the program.

- In Prolog, the resolvent is a stack.
- When the interpreter chooses a goal $A$ from the resolvent, it pops it from the stack.
- When the interpreter chooses a $A' \leftarrow B_1, ..., B_n$ from the program:
  1. It chooses the first goal in the program that unifies with $A$. According to Sterling and Shapiro, "if no unifiable clause is found for the popped goal, the computation is unwound to the last choice made, and the next unifiable clause is chosen" [p. 120].
  2. It pushes the elements $B_1, ..., B_n$ onto the stack.

# Rule Order in Prolog

- Because of how Prolog implementations handle resolvents, rule order in Prolog matters.
- To directly quote Sterling and Shapiro, "*the rule order determines the order in which solutions are found*" [p. 130, emphasis original].
- This is unimportant for pure Prolog programs since reordering rules will not affect the results of computations, but this will be very important once the cut feature is introduced next week.

Is it possible for a Prolog query to not terminate?

Is it possible for a Prolog query to not terminate? Yes, it is possible for a Prolog query to not terminate.

# Non-termination in Prolog

- This is particularly a problem with left-recursive rules.
  - A *left-recursive* rule is one where the first goal in the body is recursive (e.g., `married(X,Y) :- married(Y,X).`).
- The best way to deal with possible non-termination in left-recursive rules is to avoid them by rewriting the rule in such a way to avoid left-recursion.
  - For example, we could define a new predicate, `are_married(Person1,Person2)` with the following rules:
    - `are_married(X,Y) :- married(X,Y).`
    - `are_married(X,Y) :- married(Y,X).`

Circular definitions are also problematic; make sure you avoid them.

The ordering that does matter considerably is the ordering of goals; i.e., each $B_i$ in $A \leftarrow B_1, ..., B_n$ where $1 \leq i \leq n$.

The ordering that does matter considerably is the ordering of goals; i.e., each $B_i$ in $A \leftarrow B_1, ..., B_n$ where $1 \leq i \leq n$.

Goal ordering is important not only for runtime efficiency reasons, but also for termination reasons.

Given the rule

```
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

What would happen if the goals were swapped, resulting in the rule

```
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

Given the rule

```
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

What would happen if the goals were swapped, resulting in the rule

```
ancestor(X,Y) :- ancestor(Z,Y),parent(X,Z).
```

This would introduce left-recursion to the rule, which means the rule would become non-terminating.

# Table of Contents

# System Predicates

- Not all operations in Prolog can be conveniently or efficiently expressed as pure logic programs.
  - For example, we want to be able to perform arithmetic using our computer's ALU and not by using custom functors.
- To provide such functionality, Prolog has system predicates.
  - This is analogous to Scheme's built-in functions such as `define` and `lambda`.

# Arithmetic in Prolog

- Prolog provides built-in predicates for performing arithmetic.
- Examples include `+`, `-`, `*`, and `/`.
- We can perform queries using the following form: `Value is Expression?`.
- Examples of queries:
  - `(X is 3+5)?` $\Rightarrow$ `X=8`.
  - `(8 is 3+5)?` $\Rightarrow$ `yes`.

What does the query `(3+5 is 3+5)?` evaluate to?

What does the query `(3+5 is 3+5)?` evaluate to?

Actually, this query will fail.

The reason this query fail is because the left side of `is` expects a value, not an expression.

# Comparisons in Prolog

- Prolog provides the following built-in comparison predicates: <, >, <=, and >=.
- We can perform queries using the following form: `A op B`, where `A` and `B` are arithmetic expressions and `op` is one of the above comparison predicates.
- Examples:
  - `(1 < 2)?` $\Rightarrow$ `yes.`
  - `(4*6 < 5*5)?` $\Rightarrow$ `yes.`
  - `(6/2 < 5-1)?` $\Rightarrow$ `no.`

What does the query `(N < 1)?` evaluate to?

What does the query `(N < 1)?` evaluate to?

This query results in an error since $N$ is not an expression, but rather, a variable.

# Arithmetic Queries

We can now write rules such as the following:

```
plus(X,Y,Z) :- Z is X+Y.
```

# Arithmetic Queries

We can now write rules such as the following:

```
plus(X,Y,Z) :- Z is X+Y.
```

Unfortunately, we can't perform queries such as the following:

```
plus(3,X,8)?
```

In order to do so, we need to use *meta-logical predicates*, which will be covered in the next lecture and is discussed in Chapter 10 of the textbook.

# Table of Contents

# Type Predicates

A type predicate is a unary predicate that is able to determine the type of a term.

# Type Predicates

A type predicate is a unary predicate that is able to determine the type of a term.

Type predicates in Prolog have an equivalent in Scheme (e.g., `number?`, `pair?`, `symbol?`, etc).

# integer

`integer` determines whether a parameter is an integer.

# integer

`integer` determines whether a parameter is an integer.

Examples:

- `integer(5)?` $\Rightarrow$ `true.`
- `integer(-3)?` $\Rightarrow$ `true.`
- `integer(x)?` $\Rightarrow$ `false.`

# `atom`

`atom` determines whether the parameter is an atom (i.e., a lowercase symbol in Scheme terminology).

# atom

`atom` determines whether the parameter is an atom (i.e., a lowercase symbol in Scheme terminology).

Examples:

- `atom(bulbasaur)?` ⇒ `true.`
- `atom(5)?` ⇒ `false.`

# compound

compound determines whether the parameter has the form of a relation with one or more arguments.

# compound

compound determines whether the parameter has the form of a relation with one or more arguments.

- `compound(evolution(bulbasaur,ivysaur))?` $\Rightarrow$ `true.`
- `compound(s(0))?` $\Rightarrow$ `true.`
- `compound(bulbasaur)?` $\Rightarrow$ `false.`

# number

`number` determines whether the parameter is a number. The number can be floating-point as well as an integer.

# number

`number` determines whether the parameter is a number. The number can be floating-point as well as an integer.

- `number(5)?` ⇒ `true.`
- `number(5.1)?` ⇒ `true.`
- `number(-5.1)?` ⇒ `true.`
- `number(x)?` ⇒ `false.`

# atomic

`atomic` determines whether the parameter is either an atom or a number.

# atomic

`atomic` determines whether the parameter is either an atom or a number.

- `atomic(5)?` $\Rightarrow$ `true.`
- `atomic(-3.14159)?` $\Rightarrow$ `true.`
- `atomic(x)?` $\Rightarrow$ `true.`
- `atomic(s(0))?` $\Rightarrow$ `false.`

Time for a demo in SWI-Prolog.

What about for variables? Are there any built-in type predicates that detect whether a term is a variable?

Yes, and we will explore them later during this lecture.

# Table of Contents

Recall that a compound term is one that has the form of a relation with one or more arguments.

Recall that a compound term is one that has the form of a relation with one or more arguments.

Examples:

- `evolution(bulbasaur,ivysaur).`
- `plus(s(0),s(s(0)),s(s(s(0)))).`
- `s(0).`

Suppose we want to obtain information about a compound term, such as its arguments or its arity (i.e., number of arguments).

Suppose we want to obtain information about a compound term, such as its arguments or its arity (i.e., number of arguments). We can obtain this information by using two built-in relations: `functor/3` and `arg/3`.

# functor

`functor(Term,Name,Arity)` is a relation that accepts a compound term `Term` and checks to see if the term's name matches with `Name` and if the arity of the term is equal to `Arity`.

# functor

`functor(Term,Name,Arity)` is a relation that accepts a compound term `Term` and checks to see if the term's name matches with `Name` and if the arity of the term is equal to `Arity`.

Example:

- `functor(evolution(eevee,jolteon),evolution,2)?` $\Rightarrow$ `true.`

# arg

`arg(N,Term,Arg)` checks the compound term `Term` to see if its `N`th argument is equal to `Arg`. Note that `N` starts at 1.

# arg

`arg(N,Term,Arg)` checks the compound term `Term` to see if its `N`th argument is equal to `Arg`. Note that `N` starts at 1.

Example:

- `arg(1,evolution(eevee,jolteon),eevee)?` ⇒ `true.`
- `arg(2,evolution(eevee,jolteon),eevee)?` ⇒ `false.`

Time for another SWI-Prolog demo.

# Table of Contents

Meta-logical predicates are predicates that "sit above" the logic programming system. They are used for exercising control over the execution of logic programs.

So, back to our earlier question? Are there any built-in type predicates that detect whether a term is a variable?

So, back to our earlier question? Are there any built-in type predicates that detect whether a term is a variable?

Yes.

# var and nonvar

- `var(Term)` checks if `Term` is a variable.
- `nonvar(Term)` checks if `Term` is *not* a variable.

# Example

Here is a new version of `plus(X,Y,Z)` that uses `nonvar`:

```
plus(X,Y,Z) :- nonvar(X),nonvar(Y),Z is X+Y.
plus(X,Y,Z) :- nonvar(X),nonvar(Z),Y is Z-X.
plus(X,Y,Z) :- nonvar(Y),nonvar(Z),X is Z-Y.
```

# Example

Here is a new version of `plus(X,Y,Z)` that uses `nonvar`:

```
plus(X,Y,Z) :- nonvar(X),nonvar(Y),Z is X+Y.
plus(X,Y,Z) :- nonvar(X),nonvar(Z),Y is Z-X.
plus(X,Y,Z) :- nonvar(Y),nonvar(Z),X is Z-Y.
```

Compared to the last `plus` example from previous lectures, this has restored some query functionality; we can now run queries such as `plus(X,6,10)?`. Note that performing queries with two variables is still not supported in this above definition, but we have the tools to extend the definition to support such queries (and this will be one of your lab exercises).

# Example

Here is a new version of `plus(X,Y,Z)` that uses `nonvar`:

```
plus(X,Y,Z) :- nonvar(X),nonvar(Y),Z is X+Y.
plus(X,Y,Z) :- nonvar(X),nonvar(Z),Y is Z-X.
plus(X,Y,Z) :- nonvar(Y),nonvar(Z),X is Z-Y.
```

The uses of `nonvar` that are placed at the initial parts of the bodies of the above clauses are examples of *meta-logical tests*. Meta-logical tests decide which clause in a procedure should be used [Sterling and Shapiro].

The == predicate checks to see if `X` and `Y` are identical. \== checks to see if `X` and `Y` are not identical.

# `freeze` and `melt`

The relation `freeze(Term,Frozen)` makes a copy of `Frozen` and makes it ground (i.e., treats it as if it had no variables). The relation `melt(Frozen,Thawed).` "unfreezes" `Frozen` and makes it un-ground. Note that the textbook describes a `melt_new` relation, but it is not defined in SWI-Prolog.

# Logical Disjunction in Prolog

- We can perform logical conjunction (i.e, AND) by using commas in Prolog rules.
- For logical disjunction, we use the semicolon:
  - Example: `(X ; Y).` means `X` OR `Y`.