# Prolog and the Warren Abstract Machine

or: why logic programming is really, really cool.

---

Gabriel Soule

# Part I: An Introduction to Logic Programming

# Prologue...

Most programming languages are **imperative**. In an imperative language, we design a set of procedures which tell the computer how to solve a given problem. Object oriented programming is generally imperative, and is ubiquitous because people find objects intuitive and useful.

**Logical** programming languages are very different.

# ...Prolog!

Prolog is a programming language based on **first order logic**.

A Prolog **program** is a series of **clauses**, which are either **facts** or **rules**.

We then submit **queries**. We might ask whether a value satisfies a rule (is this statement true?), or what values satisfy a rule (what makes this statement true?)

Let's look at some examples.

# Facts About Facts

Facts are declarative statements about something. They are always true. Here are some facts:

```
it_is_rainy.
animal(dog).
animal(cat).
eats(cat, rabbit).
eats(coyote, cat).
```

We can then query Prolog:

```
?- it_is_rainy.
true.
?- animal(dog).
true.
?- animal(pig).
false.
?- eats(rabbit, cat).
false.
```

# Facts And Rules

Rules can be interepreted as conditional facts; they are true if their body facts and rules are also true.

```
edible(X)  :- animal(X).
carnivore(X) :- eats(X, Y).
```

We query Prolog in the same way:

```
?- edible(cat).
true.
?- carnivore(cat).
true.
?- carnivore(rabbit).
false.
?- cold_and_angry(wolf).
true.
```

Note that **variables**, unlike atoms, start with an uppercase letter. This distinguishes the two.

# A Familiar Example

Let us consider a slightly more complex example: a family tree. We will define two fundamental relations that define a family tree: gender and parentage. We will write rules that define more complex relationships.

# A Familiar Example

First, let's add some **facts** about the people in our (my) family.

```
man(gabe).
man(trevor).
man(andre).
man(john).
man(neil).

woman(renee).
woman(lucinda).
woman(suzanne).
```

# A Familiar Example

Now, we need to know who is the parent of who. So we will add some **facts** for that as well.

```
parent(renee, gabe).
```

# A Familiar Example

Now, we need to know who is the parent of who. So we will add some **facts** for that as well.

```
parent(renee, gabe).
parent(renee, trevor).
parent(andre, gabe).
parent(andre, trevor).
parent(lucinda, renee).
parent(lucinda, suzanne).
parent(john, andre).
parent(john, neil).
```

These fundamental facts define a family, though it would take some work to calculate more complex relationships by hand.

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).

aunt(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).

aunt(X, Y) :- sister(X, Z), mother(Z, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).

aunt(X, Y) :- sister(X, Z), mother(Z, Y).
uncle(X, Y) :-
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).

aunt(X, Y) :- sister(X, Z), mother(Z, Y).
uncle(X, Y) :- brother(X, Z), father(Z, Y).
```

# A Familiar Example

We will now implement **rules** that define more complex relationships based on our fundamental facts.

```
father(X, Y) :- man(X), parent(X, Y).
mother(X, Y) :- woman(X), parent(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), dif(X, Y).
brother(X, Y) :- man(X), sibling(X, Y).
sister(X, Y) :- woman(X), sibling(X, Y).

aunt(X, Y) :- sister(X, Z), mother(Z, Y).
uncle(X, Y) :- brother(X, Z), father(Z, Y).
```

# The Power of Prolog

We've seen how we can use atoms in Prolog queries to determine whether the queried rule is true or unprovable (false) with the given atoms. For example:

```
?- aunt(suzanne, gabe).
true.

?-aunt(suzanne, andre).
false.
```

# The Power of Prolog: Variables

What happens if we use **variables** in our query? Suppose we ask Prolog:

```
?- aunt(suzanne, Who).
```

Prolog will respond:

```
Who = gabe;
Who = trevor.
```

What's going on? Recall, when we use atoms in a query, we are asking: *with these things, is this rule satisfied?* When we pass in a **variable**, we are asking: *for what things, when associated with this variable(s), is this rule satisfied?* From above, we see that `?- aunt(suzanne, gabe).` and `?- aunt(suzanne, trevor).` are both `true`.

# The Power of Prolog: Variables

Notice that, when we use variables in our queries, they act much like imperative functions.

```
aunt(suzanne, X) <--> findNephews(suzanne);
aunt(X, Y) <--> findAllAunts();
```

It would be cumbersome to implement these procedures procedurally. But in Prolog, we can do this easily, without explicitly telling Prolog how.

# Logic is Power

This is an an incredibly powerful principle that illustrates the difference between logical and imperative programming.

In an imperative language, we tell the computer *how* to solve a problem, and hope that our instructions produce the solution correctly.

In a logical language, we tell the computer *what* a solution looks like, and the computer finds such a solution.

This makes Prolog almost stupidly powerful when it comes to solving problems whose solutions are easily defined, but hard to calculate procedurally.

# Unification

Prolog is fundamentally based on the logical operation of **unification**.

We call the facts and rules in a query **goals**. Prolog will attempt to satisfy all goals by finding appropiate unifications, or bindings, for all variables within, such that the goals (and their sub-goals, and those goals' sub-goals, and so forth) can be matched with one or more given facts.

To reiterate: **Prolog seeks to assign unbound variables to things such that all rules eventually match to one or more facts.**

# A Unifying Example

```
q(a, b).
r(b, c).
p(X, Y) :- q(X, Z), r(Z, Y).

p(V, W)?
```

First, Prolog notices that `P(V, W)` and `P(X, Y)` unify. Now it can work on the goals of `p(X, Y)`. It notices that if it unifies, or associates, `X` with `a`, `Y` with `c`, and `Z` with `b`, then it has:

```
q(a, b), r(b, c).
```

These are facts! So the query is successful, and Prolog responds with the unifications it made, namely

```
V = a,
W = c.
```

# Will It Unify?

```
f(X).

?- f(a)
```

# Will It Unify?

```
f(X).

?- f(a)


true.
```

# Will It Unify?

```
f(a).

?- f(X)
```

# Will It Unify?

```
f(a).

?- f(X)


X = a.
```

# Will It Unify?

```
f(a, b).

?- f(X, Y)
```

# Will It Unify?

```
f(a, b).

?- f(X, Y)


X = a.
Y = b.
```

# Will It Unify?

```
g(a, b).

?- g(X, X).
```

# Will It Unify?

```
g(a, b).

?- g(X, X).


false.
```

# Will It Unify?

```
f(X, g(h(X))).

?- f(a, X).
```

# Will It Unify?

```
f(X, g(h(X))).

?- f(a, X).


X = g(h(a)).
```

# Will It Unify?

```
q(a, b, c).
r(c).
p(X, Y) :- q(X, Z, Y), r(Y).

?- p(U, V).
```

# Will It Unify?

```
q(a, b, c).
r(c).
p(X, Y) :- q(X, Z, Y), r(Y).

?- p(U, V).
```

```
U = a.
V = c.
```

# Will It Unify?

```
p(f(X), h(Y, f(a)), Y).

?- p(Z, h(Z, W), f(W)).
```

# Will It Unify?

```
p(f(X), h(Y, f(a)), Y).

?- p(Z, h(Z, W), f(W)).

Z = f(f(a)).
W = f(a).
```

# Backtracking

Another feature essential to Prolog execution is **backtracking**.

Whenever Prolog makes a choice—to attempt unification with one definition of a given rule as opposed to another, for exanple—it creates a **choice point**.

Under the right conditions, Prolog will **backtrack** to the nearest choice point, undoing any unifications made after that choice point. It will then try again, ignoring that path it backtracked from.

There are two condititions under which backtracking occurs.

# Condition 1: Prolog Screws Up

```
q(X, X).
q(X, Y).

?- q(a, b).
```

Prolog has two facts that might unify with the query `q(a, b)`, so it has to make a **choice**. Suppose it chooses `q(X, X)`. This does not unify, so Prolog will **backtrack**, and try again, ignoring `q(X, X)`. It is now left with `q(X, Y)`, which unifies successfully with the query.

# Condition 2: Prolog Wants More

Recall that when we use a variable in a query, Prolog will tell us all unifications for that variable that satisfy the query.

```
?- aunt(suzanne, X).

X = gabe;
X = trevor;
false.
```

# Condition 2: Prolog Wants More

Recall that when we use a variable in a query, Prolog will tell us all unifications for that variable that satisfy the query.

```
?- aunt(suzanne, X).

X = gabe;
X = trevor;
false.
```

First, Prolog notices that `X = gabe` satisfies the query. We then ask Prolog again, and Prolog backtracks until it notices that `X = trevor` also satisfies the query. We ask Prolog a third time, and Prolog backtracks and backtracks until it has gone through the entire problem space without finding a solution. Then, since there is no solution (excepting `gabe` and `trevor`), it prints `false`.

# Prolog IRL

What about… actual programs?

Prolog is a Turing-complete language. It is particularly suited for compilers, natural language processing, and yes, even web development.

But how do you write actual code in the language of logical clauses?

# Prolog is Backwards

When I was first learning Prolog I tried to implement a *remove-all* procedure.

```
?- removeall([a, b, c], c, Out)
Out = [a, b].

?- removeall([c, b, c], c, Out)
Out = [b].
```

# Prolog is Backwards

I tried to solve this problem imperatively, designing a series of clauses that were intended to build a new list which was the original list minus all instances of the item to remove.

This did not go well.


Why eant you just be normal?
Prolog
*Screams*

# Prolog is Backwards

Recall: if we can do this...

```
?- removeall([a, b, c], c, Out)
Out = [a, b].
```

# Prolog is Backwards

Recall: if we can do this...

```
?- removeall([a, b, c], c, Out)
Out = [a, b].
```

...then we can do *this*.

```
?- removeall([a, b, c], c, [a, b])
true.
```

# Prolog is Backwards

```
?- removeall([a, b, c], c, Out)
Out = [a, b].

?- removeall([a, b, c], c, [a, b])
true.
```

This tells us that our `removeall` procedure should not build a list or modify a list, but rather, verify that, given two lists, the second list is equal to the first but with the item removed.

So instead I wrote some clauses that define the desired *solution* of the `removeall` procedure.

And in doing so, I solved the problem.

# Part II: Implementing Prolog with the Warren Abstract Machine

# The Warren Abstract Machine

The Warren Abstract Machine, or **WAM**, is an abstract machine designed for the execution of Prolog code.

It was developed by David H. D. Warren in 1987.

The WAM remains the dominant model for Prolog execution today. Many Prolog implementations, such as GNU Prolog, are still based on the WAM.

# What *is* an abstract machine, anyway?

An abstract machine is a virtual model of computer hardware and software.

The Warren Abstract machine was developed with VAX machines in mind. It is rather low level.

The WAM is composed of memory areas, registers, and an instruction set.

# The WAM Compiler

Since the WAM does not directly read Prolog source code, a **compiler** is needed to turn Prolog source into WAM instructions.

The compiler first tokenizes Prolog source code. As a rule of thumb, each token—each functor, each variable—is translated into a single WAM instruction. Each clause is translated into a labeled block of WAM instructions.

The set of instructions are then fed into the WAM and evaluation can begin.

# The Anatomy of a Clause

Before we begin, we must formally define some vocabulary.

```
p(X, Y) :- q(X, Z), r(Z, Y).
```

The above **clause** describes the **predicate** *p/2*.

We call it *p/2* because its name is *p* and it has arity 2.

The red part is the **head** of the clause.

The blue part is the **body** of the clause.

All these Prolog elements—`X`, `q(X, Z)`, `Y`—are all Prolog **terms**.

Non variable terms, such as `q/2`, are **structures**.

# The WAM: Overview

We will learn about the WAM and its compiler via example.

Suppose we have a **program fact**

```
f(g(X), h(X)).
```

and a **query**

```
f(Z, h(a)).
```

# The WAM: Overview

We will learn about the WAM and its compiler via example. Suppose we have a **program fact**

```
f(g(X), h(X)).
```

and a **query**

```
f(Z, h(a)).
```

Running this query on this program will unify `Z` with `g(X)` and `X` with `a`, thus producing the result

```
Z = g(a).
```

We will illustrate the WAM by compiling and running this program and query by hand. Let us begin!

# Parse Trees

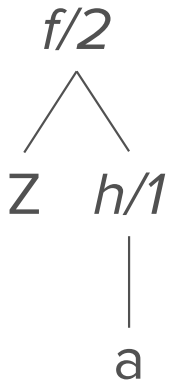
The WAM first builds a **parse tree** of the query.

Then, it builds the solution by matching the parse tree to the **program fact**, going term by term, much like we do when unifying predicates by hand.

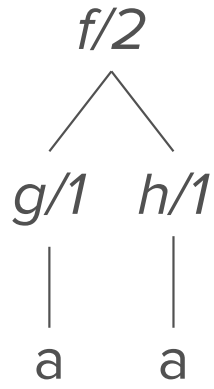
Thus, instructions generally fall into two categories: *put* instuctions, responsible for building the query upon the heap, and *get* instructions, responsible for matching the query against the program.

If any term fails to unify, then unification fails as a whole.

# Parse Trees




1: The query *q* = `f(Z, h(a))`




2: *q* after uniting with program fact *p* = `f(g(X), h(X))`

# The Heap

However, the WAM is a simple machine. For both simplicity and performance reasons, it is undesirable to represent a parse tree in a modern object-oriented tree structure.

# The Heap

However, the WAM is a simple machine. For both simplicity and performance reasons, it is undesirable to represent a parse tree in a modern object-oriented tree structure.

Instead, the WAM uses a stack-based model called the **heap**—essentially, a tree in stack form. The heap is an indexable list of **data cells**. Each data cell lives at an address, namely, its position in the heap array.

A data cell comes in the form `<tag, a>`. The *tag* of the data cell defines the cell's type. *a* is an integer address, pointing somewhere in the heap. For the purposes of this lesson, there are three types of data cells.

# REF Cells

Reference cells `<REF, a>` represent variables in Prolog. An unbound reference cell, i.e. a variable that has not yet been unified with anything else, points to itself, i.e. its address field *a* is equal to its own heap address.

# STR Cells

Structure cells `<STR, a>` are special reference cells that point to non-variable terms, namely, structures on the heap.

For example, if we have a structure $p(a, b)$ somewhere in a Prolog expression we must have a structure cell on the heap pointing to this structure's heap representation. There must always be one or more structure cells for each unique structure on the heap.

Structure cells are just special cases of reference cells, except the point to structures. Structures themselves are actually stored using...

# Functor Cells

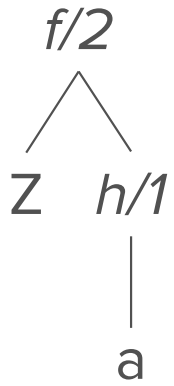
Functor cells `<f/n>` denote the heap representation of a structure. A functor cell is special, and contains only the data `f/n`, which, recall, describes the name and arity of the described functor.

A functor cell `<f/n>` is *immediately* followed by *exactly n* cells that represent its arguments.

For example, a functor cell `<p/2>` will be followed by exactly two cells which represent the arguments of *p*.

# A Heap of Examples

Recall, that the WAM first builds the query on the heap, before unifying it with the program.




| | |
|---|---|
| 0 | STR 1 |
| 1 | *a/0* |
| 2 | STR 3 |
| 3 | *h/1* |
| 4 | STR 1 |
| 5 | STR 6 |
| 6 | *f/2* |
| 7 | REF 7 |
| 8 | STR 3 |

1: The query *q* = `f(Z, h(a))`

2: Also the query *q* = `f(Z, h(a))`

# Why We Need Registers

There still remain some problems with our heap representation of Prolog expressions.

First of all, a parse tree has a **root**. The root of a parse tree, when represented as an actual tree, is obvious. However, from the heap representation, it is not clear where the root of the tree is.

# Why We Need Registers

There still remain some problems with our heap representation of Prolog expressions.

First of all, a parse tree has a **root**. The root of a parse tree, when represented as an actual tree, is obvious. However, from the heap representation, it is not clear where the root of the tree is.

It is our intent to go through the query tree and match it to the program term by term. Thus, we must know where the root of the tree is so we can start matching.

# The X Registers

Luckily, the WAM is equipped with a series of **registers** $X_1 \ldots X_n$. Registers store references to data cells on the heap.

It would be intuitive, then, to store a reference to the root of the tree in `X1`. In the case of our example, `f(Z, h(A))`, we would set `X1 = STR 6`.

However, there is another problem...

# Argument Registers

The WAM, in fact, does not include the outermost functor of a Prolog expression on the heap. Recall that each clause in a Prolog program is compiled into a labeled block of WAM instructions. Thus, the WAM uses the outermost functor in the head of a clause as a label, and does **not** include it on the heap.

Now we have a parse tree without a root.

# Argument Registers

We will solve this problem with **argument registers**. Now, each argument gets its own parse tree. For a predicate of arity *n*, the first *n* registers `X1...Xn` are each assigned to the heap cell representing the root of each argument tree.

When a register `Xi` is used to represent the root of an argument parse tree, we call it an argument register and denote it `Ai`. If a root is a variable, it is assigned both an argument register and a term (`X`) register.

# Argument Registers

| | |
|---|---|
| 0 | STR 1 |
| 1 | *a/0* |
| 2 | STR 3 |
| 3 | *h/1* |
| 4 | STR 1 |
| 5 | STR 6 |
| 6 | *f/2* |
| 7 | REF 7 |
| 8 | STR 3 |

*q* = `f(Z, h(a))`. The register `X1 = STR 6`

| | |
|---|---|
| 0 | REF 0 |
| 1 | STR 1 |
| 2 | *a/0* |
| 3 | STR 4 |
| 4 | *h/1* |
| 5 | STR 2 |

*q* = `f(Z, h(a))`. $f/2$ has been eliminated; `A1 = REF 0`, `A2 = STR 4`.

# Variable Registers

We have established how we use the first $n$ registers to signify the root of each of the $n$ argument trees on the heap.

We also use the remaining $X_{n+1} \ldots X_i$ to store variables and other terms before they are placed on the heap. When matching, we use these registers to quickly find and unify the appropiate terms (instead of having to search through the tree). This is a vague and unsatisfactory explanation, but it should make more sense once we run through an example.

# Compiling Queries

To compile a query, we must first convert it into its **flattened** form, and, in doing so, assign registers to its terms. A flattened form is a series of Prolog expressions with nested depth 1, i.e. of the form `Vn = f(Vi, Vj, Vk...)` where `V` is either an `X` or `A` register.

Below are the register assignments for the query `f(Z, h(A))`.

```
A1, X3 = Z
A2 = f(X4)
X4 = a
```

We map this flattened form directly into Prolog instructions. However, there is one problem we must resolve first.

# Ordering a query's flattened form

Recall that the heap is a stack. As such, we can not insert or remove data cells at arbitrary indices; rather, we can only push and pop data cells to/from the top. Our query instructions thus must build our stack via push operations.

However, notice that for a heap representation to be well-formed, it must be ordered from the bottom up! That is, the innermost terms with no subterms must be built first, then the terms of which those innermost terms are the subterms, and so on and so forth until the outermost terms.

# Ordering a query's flattened form

Since our flattened form is translated almost directly into WAM instructions, we must order its terms from the bottom up to ensure a well-formed heap representation. This ensures that a register cannot be referenced before its assignment. To order the query `f(Z, h(A))`, we must move the term `X4 = a` *above* `A2 = f(X4)`; otherwise the register `X4` would be referenced before being assigned to *a*.

```
A1, X3 = Z
X4 = a
A2 = f(X4)
```

# A Query, Compiled

```
A1, X3 = Z
X4 = a
A2 = h(X4)
```

The ordered flattened form of the query `f(Z, h(a))`.

```
put_variable A1 X3
put_structure a/0 X4
put_structure h/1 A2
set_value X4
call f/2
```

WAM instructions for the query `f(Z, h(a))`, derived from the flattened query.

# TL;DR: Compiling a Query

We now understand how to compile a query into a labeled block of WAM instructions. To recapitulate:

**First**, we convert a query into its flattened form, which is a series of register assignments of the form `Xi = f(X1...Xn)`.

**Then**, we order the query's flattened form to ensure that no register is referenced before its assignment.

**Finally**, we read and translate the ordered flattened form into WAM instructions. The effect of these instructions is to build a model of the query on the heap.

# Compiling Programs

Now we will compile our program fact `f(g(X), h(X))` into WAM instructions.

Whereas query instructions build the query on the heap, program instructions match the program with the query, building onto the heap and performing unifications as necessary.

Though program facts correspond to a different set of instructions, compiling one is very similar to compiling a query fact.

# Compiling Programs

Like with our query, we must first convert the program fact `f(g(X), h(X))` into its flattened form:

```
A1 = g(X3)
A2 = h(X3)
```

Note that we do *not* need to order the flattened form of program terms from the bottom up, since we already have an exemplar of of the query built on the heap (which we expect to be able to match with the program). Registers have already been assigned; there is no risk of using a register before its assignment.

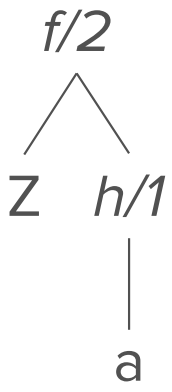
# A Program, Compiled

```
A1 = g(X3)
A2 = h(X3)
```

The flattened form of the program `f(g(X), h(X))`.

```
f/2:
  get_structure g/1 A1
  unify_variable X3
  get_structure h/1 A2
  unify_value X3
```

WAM instructions for the program `f(g(X), h(X))`, derived from the flattened program.



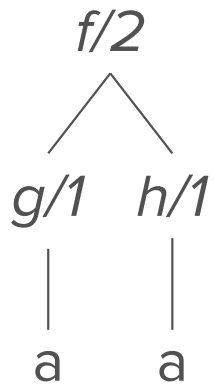

1: The query *q* = `f(Z, h(a))`

| | |
|---|---|
| 0 | REF 0 |
| 1 | STR 1 |
| 2 | *a/0* |
| 3 | STR 4 |
| 4 | *h/1* |
| 5 | STR 2 |

The heap representation of *q*.

`Z = A1 = REF 0`,

`h/1 = A2 = STR 4`




2: *q* after uniting with program fact *p* = `f(g(X), h(X))`

| | |
|---|---|
| 0 | REF 6 |
| 1 | STR 1 |
| 2 | *a/0* |
| 3 | STR 4 |
| 4 | *h/1* |
| 5 | STR 2 |
| 6 | STR 7 |
| 7 | *g/1* |
| 8 | REF 2 |

The heap representation of *q* after uniting with *p*. Observe that the cell representing `Z` now points to `g(a)`, which is what we expect!

# Unifying Rules

We now understand how the WAM unifies a query and a fact.

However, this is not very useful. We would like to be able to process more complex Prolog programs that contain rules. Rules, unlike facts, have bodies.

```
father(X, Y) :- man(X), parent(X, Y).
```

```
a(X) :- b(X).
```

It turns out that it is remarkably simple to extend our treatment of facts and queries to more complex clauses such as the one above.

# Unifying Rules

*A rule is a fact with a body. It is true if its body is true.*

```
p(X, Y)
```

# Unifying Rules

*A rule is a fact with a body. It is true if its body is true.*

```
p(X, Y) :- q(X, Z), r(Z, Y).
```

Thus, we can treat the *head* of a rule as a fact, and its *body goals* as queries.

To process a rule, a query must unify with its head, and if this succeeds, its body goals can be dispatched as queries.

Similarly, a user query entered into a console can be interepreted as a rule without a head.

# Compilation of Rules

Below is the compilation pattern for a generic rule $f(\ldots) :- a_1(\ldots), a_2(\ldots) \ldots a_n(\ldots).$

```
f/n:
  get/unify arguments of f
```

# Compilation of Rules

Below is the compilation pattern for a generic rule $f(\ldots) :- a_1(\ldots), a_2(\ldots) \ldots a_n(\ldots).$

```
f/n:
  get/unify arguments of f

  put arguments of a1
  call a1
  put arguments of a2
  call a2
  ...
  put arguments of an
  call an
```

# Stack Frames

Our solution for rules *almost* works out of the box. However, there are still a couple problems we need to fix.

Recall that each clause in a Prolog source file is translated to a series of WAM instructions. When we `call` a clause, we jump to the beginning of that clause's block of WAM instructions. When we reach the end of that clause's instructions, we need to return to the instruction after the original call.

# Stack Frames

Our solution for rules *almost* works out of the box. However, there are still a couple problems we need to fix.

Recall that each clause in a Prolog source file is translated to a series of WAM instructions. When we `call` a clause, we jump to the beginning of that clause's block of WAM instructions. When we reach the end of that clause's instructions, we need to return to the instruction after the original call.

Thus, we need **stack frames**. When we `call` a procedure, we allocate a stack frame. For now, a stack frame contains a pointer `cp` that points to the instruction after the `call` instruction, i.e. where to resume execution after the invoked procedure concludes.

# Permanent Variables

There is still one more problem. The same registers `X1...Xn` are used by each procedure, which can be trouble. Consider the example

```
P(X, Y):- q(X, Z), r(Z, Y).
```

After executing `q`, we must `put` the arguments of `r` on the heap and call $r/2$.

However, there is no guarantee that the registers originally assigned to `Y` and `Z` still point to their respective variable cells; they may have been overwritten during execution of `q`.

# Permament Variables

Thus, we must distingish ”permanent” variables from ”temporary” variables.

A permanent variable is a variable that appears more than once in the body, such that is affected by more than one body goal’s `put` instructions.

To protect permanent variables, each stack frame is equipped with variable registers `Y1...Yn`. When a permanent variable would be assigned an `X` register, it is assigned to a `Y` register instead.

# Compilation of Rules with Stack Frames

We now `allocate` a stack frame with *m* permament variables before processing a clause, and `deallocate` it when we are done.

```
f/n:
  allocate m
  get/unify arguments of f
  put arguments of a1
  call a1
  put arguments of a2
  call a2
  ...
  put arguments of an
  call an
  deallocate
```

# The End

We now understand how the WAM processes complex Prolog programs that contain both facts and rules. Hooray!

To recapitulate: we compile a query into a set of instructions that builds a model of it on the heap.

We compile a fact into a set of instructions that matches it with the query already built on the heap.

To process rules, we treat the head of the rule as a fact and its body goals as queries, and process them accordingly. Stack frames protect permanent variables, and track where to continue execution upon deallocation.

# The End...?

However, something very, *very* big is missing from our implementation of Prolog:

# The End...?

However, something very, *very* big is missing from our implementation of Prolog: **backtracking**.

# The End...?

However, something very, *very* big is missing from our implementation of Prolog: **backtracking**.

Our current implementation of Prolog cannot backtrack. If it encounters failure, it will fail immediately without backtracking to try other options.

Prolog's ability to solve complex problems and function as an (arguably) useful language is largely predicated upon backtracking.

Thus our implementation is pretty useless (but it's cool tho.)

# The End...?

However, something very, *very* big is missing from our implementation of Prolog: **backtracking**.

Our current implementation of Prolog cannot backtrack. If it encounters failure, it will fail immediately without backtracking to try other options.

Prolog's ability to solve complex problems and function as an (arguably) useful language is largely predicated upon backtracking.

Thus our implementation is pretty useless (but it's cool tho.)

Backtracking is conceptually simple. However, it is complex to implement in the WAM and is a topic for another day.

# The End...?!?!!!

We're also missing a number of useful features such as lists, arithmetic, and several important optimizations such as tail call optimization.

Again, these omissions, along with the problem of backtracking, are topics for another day. Think to yourself: how might you solve these problems?

Perhaps you could write a better logical programming language!

# The End!

# Bibliography


- AN ABSTRACT PROLOG INSTRUCTION SET - David H.D. Warren, 1983
- Warren’s Abstract Machine: A Tutorial Reconstruction - Hassan Ait-Kaci, 1999