A WAM Prolog for Teaching

Andrew Davison
Dept. of Computer Eng., Prince of Songkla Univ.
Hat Yai, Songkhla 90110, Thailand
Email: `ad@fivedots.coe.psu.ac.th`

July 2020

1. Introduction

BuProlog consists of two JAR files – BuProlog.jar holding a Prolog [6] compiler that generates WAM-like [7] code, and BuWAM.jar, a virtual machine for executing queries against that code.

The system was developed for two reasons. The primary one was to create a smallish compiler and virtual machine suitable for projects in this author's course on compiler design. The second purpose was to make a simple-to-use programming tool for an introductory lab on Prolog. One result of these goals is that the system can print out extensive diagnostic data and, more unusually, generate parse trees and execution graphs showing how queries are evaluated.

This work is a fork of the Java-based Prolog compiler/runtime system written by Stefan Buettcher, available at http://stefan.buettcher.org/cs/wam/. However, the compiler was largely rewritten to employ conventional recursive descent parsing and code generation techniques, and the diagnostics and graphing functionality are also new. Buettcher's virtual machine was refactored and simplified in several ways, while retaining enough features to run programs typical of an introductory Prolog lab.

The main purpose of this report it to give a fairly detailed overview of the internals of the compiler and virtual machine, along with an introduction to the WAM instruction set and its implementation [2]. This guide is not intended to be an introduction to Prolog or compiler design. There are many good books about Prolog[4; 3; 6] and websites such as "Learn Prolog Now" (http://learnprolognow.org), "The Power of Prolog" (https://www.metalevel.at/prolog), and "Adventures in Prolog" (http://www.amzi.com/AdventureInProlog/).

1.1. Using the System

BuProlog comes with many small examples (see Appendix A for a list). A file is compiled like so:

```
> java -jar BuProlog.jar parents.pro
```

This generates a `parents.wam` text file of WAM instructions, which can be loaded by the run-time system:

```
> java -jar BuWam.jar -p parents.wam
```

BuWAM then enters a REPL, where the user can pose queries against the loaded code. For example:

```
?- parentOf(herbert,X).
?- length([a,b,c],L).
?- append(X,Y,[1,2,3]).
```

The "-p" flag included with the BuWAM.jar call causes the evaluation of each query to also be rendered as a graph, which is saved as a PNG image.

For example, consider when the `?-parentOf(herbert,X)` query is applied to the facts:

```
parentOf(kim,holly).      % kim is the parent of holly
parentOf(margaret,kim).
parentOf(margaret,kent).
parentOf(esther,margaret).
parentOf(herbert,margaret).
parentOf(herbert,jean).
```

The first answer is `X=margaret` and after the user requests another answer, `X=jean`. When the "-p" flag is set, the execution will generate the graph shown in Figure 1.

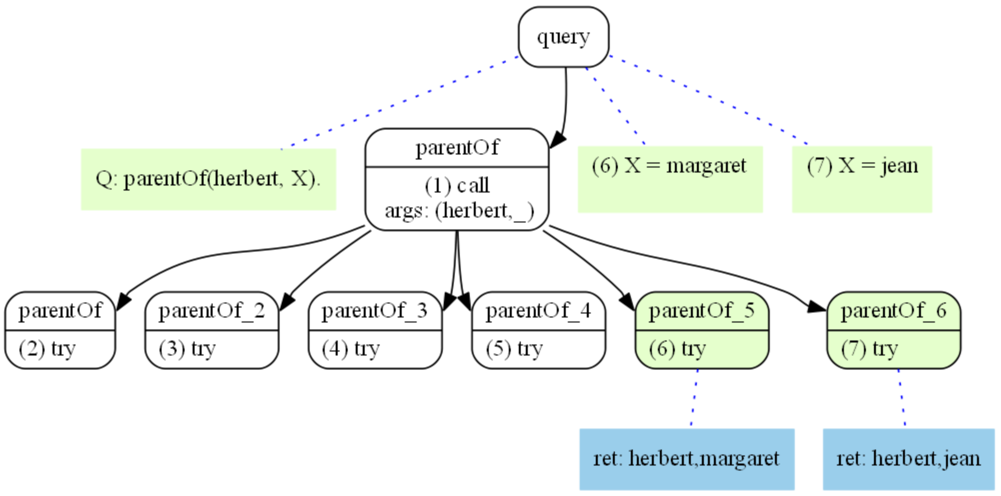


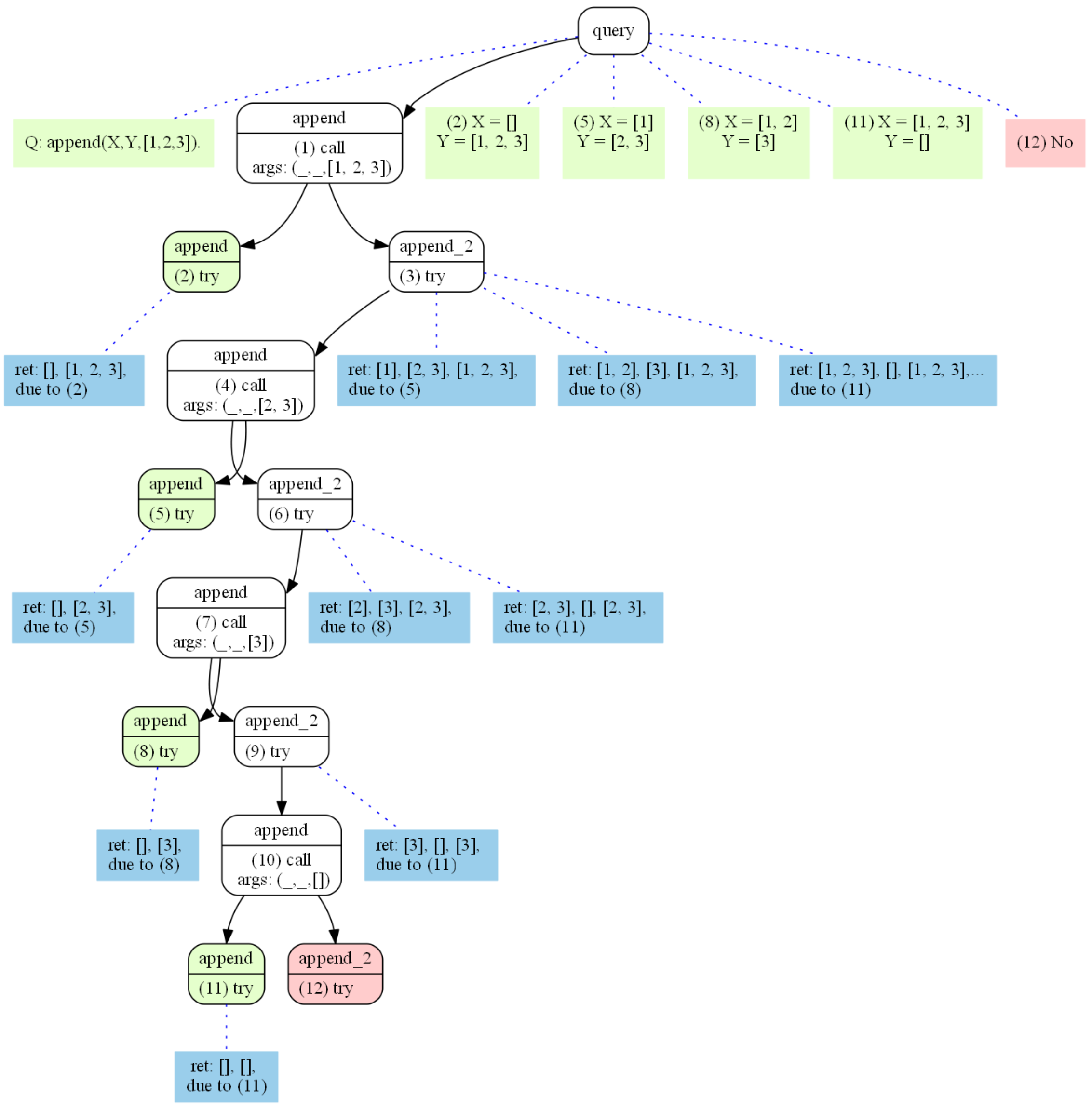
Figure 1. The execution of `?-parentOf(herbert,X).`

An execution graph consists of alternating levels of "call" and "try" nodes. A "call" represents a goal that will be matched against facts and rules in its child "try" nodes. Each "try" is labeled with the predicate name and "_<number>" which denotes which of the predicate's facts or rules is being tested. All of the "call" and "try" nodes are uniquely numbered to help the user follow Prolog's backtracking execution order.

Nodes colored green denote that a query has succeeded and will return an answer. For instance, in Figure 1, the binding of X to margaret occurs in the fifth parentOf/2 fact. When the user requests another answer, execution backtracks to node (6), resumes and a second answer is generated.

Figure 2 shows the execution of `?-append(X,Y,[1,2,3])`, which produced four results through backtracking. append/3 is defined in the standard way:

```
append([],L,L).
append([H|T],L2,[H|L3]) :- append(T,L2,L3).
```



query
Q: append(X,Y,[1,2,3]).
append (1) call args: (_,_,[1, 2, 3])
(2) X = [] Y = [1, 2, 3]
(5) X = [1] Y = [2, 3]
(8) X = [1, 2] Y = [3]
(11) X = [1, 2, 3] Y = []
(12) No
append (2) try
append_2 (3) try
ret: [], [1, 2, 3], due to (2)
append (4) call args: (_,_,[2, 3])
ret: [1], [2, 3], [1, 2, 3], due to (5)
ret: [1, 2], [3], [1, 2, 3], due to (8)
ret: [1, 2, 3], [], [1, 2, 3],... due to (11)
append (5) try
append_2 (6) try
ret: [], [2, 3], due to (5)
append (7) call args: (_,_,[3])
ret: [2], [3], [2, 3], due to (8)
ret: [2, 3], [], [2, 3], due to (11)
append (8) try
append_2 (9) try
ret: [], [3], due to (8)
append (10) call args: (_,_,[])
ret: [3], [], [3], due to (11)
append (11) try
append_2 (12) try
ret: [], [], due to (11)

Figure 2. The execution of ?- append(X,Y,[1,2,3]).

The four results are obtained when the query succeeds in nodes (2), (5), (8), and (11). When the user asks for a fifth answer, all the possibilities are exhausted by node (12), and "No" is printed. A query failure is denoted by coloring the node red.

The execution graph also includes bindings (in blue boxes), which show how a result is constructed. For example, the third result, X = [1,2] and Y =[3], is due to the success of node

(8), but the binding is constructed in stages, represented by the three boxes hanging off nodes (8), (6), and (3).

When the user requests an alternative answer, the system resumes its execution immediately after the node that supplied the previous answer. For example, the third answer to the append/3 query begins at node (9). This tries the second append/3 clause against the append/3 call in node (7).

These graphs are a useful tool for students new to Prolog, but also suffer from some drawbacks. The main one is that they quickly become very large, and consequentially quite confusing. For example, students are advised to run the example N-queens and Towers of Hanoi programs with small input sizes (e.g. a 4x4 board, and three disks respectively) to get an idea of how they work before moving onto larger input (e.g. a 8x8 board and six disks). Also, very large graphs may be too large for the graph renderer, which will report a "graph is too large" error or even crash. A third concern is that BuProlog has no control over how the generated graphs are laid out, and so there's no way to correct the occasionally confusing positioning of arrows (e.g. as seen below nodes (4) and (7) in Figure 2).

This graphing is carried out by the excellent GraphViz tool (https://graphviz.org/) which is installed separately from BuProlog.

1.2. Differences from Prolog

BuProlog only supports a small set of built-in predicates, which are listed in the source code of the Builtin enum (see Builtin.java) and in Appendix B.

The most serious language restriction is that atoms and variable names can only use letters and digits, although a variable may start with "_". This is spelled out by the regular expressions for those token types in Lexer.jflex:

```
Atom     = [a-z]([:letter:] | [:digit:])*

Variable = [A-Z_]([:letter:] | [:digit:])*
```

Also only integers are supported:

```
Integer  = [-+]?[:digit:]+
```

In addition, every functor must be defined as an atom followed by a bracketed list of arguments. The relevant BNF grammar rule is:

```
functor ::= Atom [ '(' args ')' ]
```

In particular this means that tests such as `X < 2`, must be coded as `le(X,2)`, and common Prolog built-ins '=", "\=" and "=.." are referred to by the names `unify`, `nununify`, and `univ`. The is/2 predicate is also written in prefix form, and its expression as a term. For instance, a programmer must write `is(X1, add(X,1))` rather than `X1 is X+1`.

2. The BuProlog Compiler

The compiler utilizes a standard design [1] , as illustrated by Figure 3.

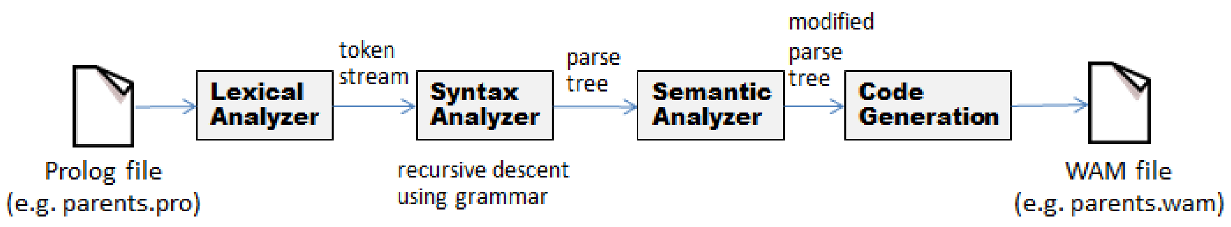


Figure 3. Stages of a Compiler.

Figure 3 is the basis of an augmented UML class diagram shown in Figure 4. The "Semantic Analysis" stage is missing since its functionality mostly consists of extra processing inside the code generator.

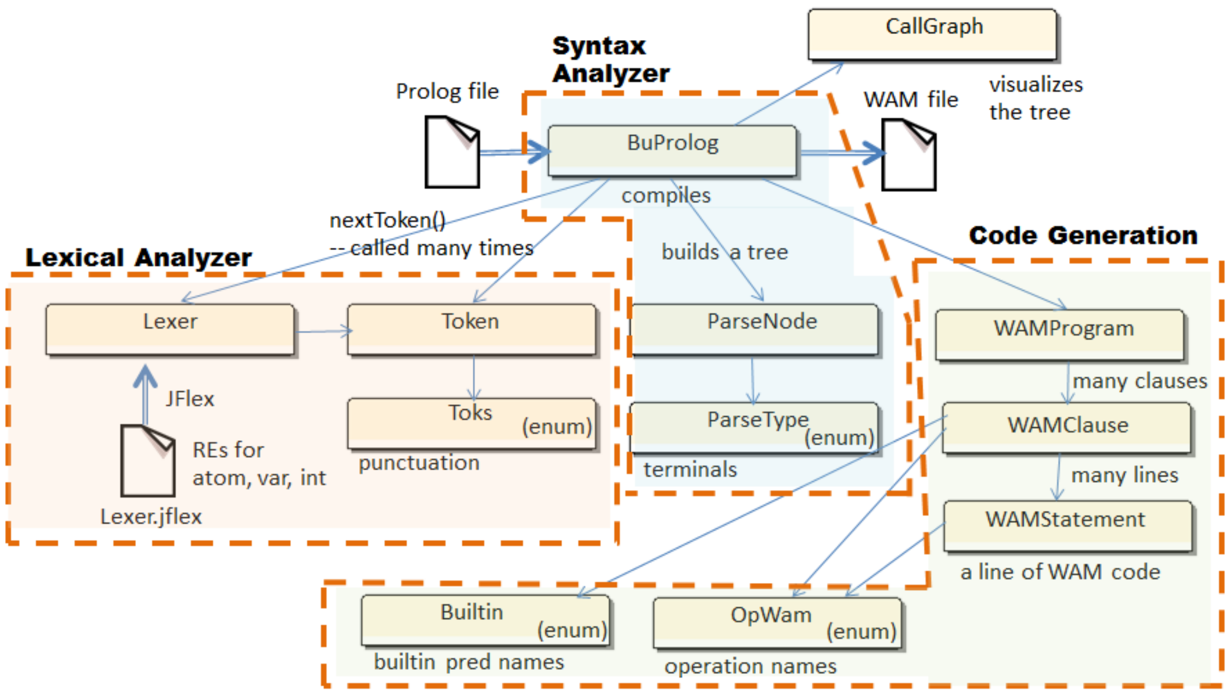


Figure 4. Class Diagram for the BuProlog Compiler.

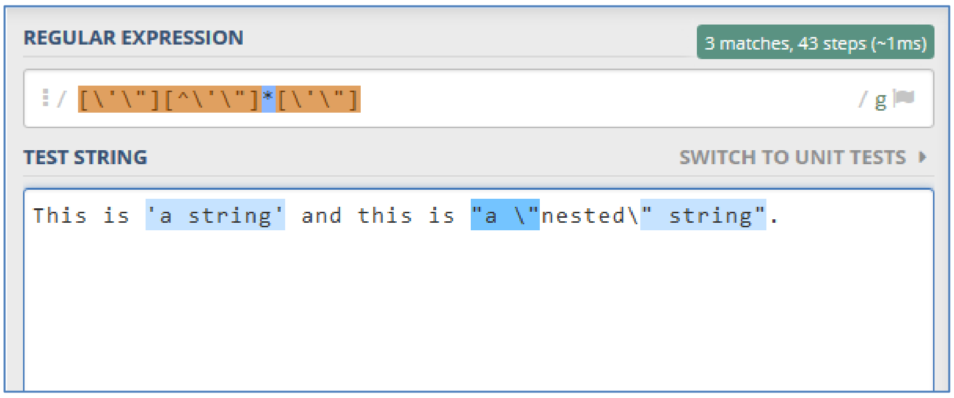
2.1. Lexical Analysis

The lexical analysis phase is defined by regular expressions expressed in a Java version of lex called JFlex (https://www.jflex.de/).

The two most confusing aspects of writing a lexer are probably formulating its regexs and it's interface to the rest of the parser. For the first problem, a regular expression testing site, such as https://regex101.com, is a great help. For instance, one of the hardest regexs in BuProlog matches a quoted string:

```
Str   = [\'\"][^\'\"]*[\'\"]
```

The expression's strengths and weaknesses can be seen by testing it at regex101, as shown in Figure 5.



```
/ [\'\"][^\'\"]*[\'\"] / g
```

```
This is 'a string' and this is "a \"nested\" string".
```

Figure 5. Testing a Regular Expression.

Figure 5 shows that this regex cannot deal with nested strings.

The compiler also includes a simple test-rig (in LexerTest.java) for invoking the Lexer class separately from the compiler:

```
public class LexerTest
{
  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.out.println("Usage: java LexerTest <fnm>");
      return;
    }
    Token tok;
    try {
      Lexer lexer = new Lexer(new FileReader(args[0]), true);
                                  // printing is on
      tok = lexer.nextToken();
      while (tok.symbol() != Toks.EOF) {
        // System.out.print(tok + " ");
        // no need to print if Lexer() called with true arg
        tok = lexer.nextToken();
      }
      System.out.println();
    }
    catch (Exception e)
    { System.out.println(e.getMessage()); }
  }
}  // end of LexerTest class
```

2.2. Syntax Analysis

BuProlog's recursive descent parser is a direct translation of the following EBNF grammar for the language:

```
program ::=  query | clauses

query ::= '?-' body '.'

clauses ::= clause clause*
clause ::= functor [  ':-' body  ] '.'

functor ::= Atom [ '(' args ')' ]

body ::= goal [ ',' body ]
goal ::= functor  |  '!'

structure ::= Atom [ '(' args ')' ]
   // same as functor but useful separation for code generation

args  ::=   term  ( ',' args )
term  ::=   structure | Variable | Integer | list | Str

list  ::=   '[' [  args  [ '|' list_tail ] ]  ']'
list_tail  ::=    Variable | list


// managed by the lexer:
Atom     = [a-z]([:letter:] | [:digit:])*
Variable = [A-Z_]([:letter:] | [:digit:])*
Integer  = [:digit:]+
Str      = [\'\"][^\'\"]*[\'\"]
```

Each rule is encoded by a similarly named method which combines two tasks – the parsing of that part of the grammar, and the construction of its part of a parse tree.

For example, the top-level rule, `program`, becomes:

```
public ParseNode program() throws IOException
// EBNF: program ::=  clauses
{
  int numSps = 1;
  if (debugOn)
    iewriteln(numSps,"program");

  ParseNode tree = new ParseNode();
  clauses(tree, numSps+1);
  if (tok.symbol() == Toks.EOF)  {
    writeln("\n\nFinished parsing.");
    updateNames(tree);
  }
  else {
    writeln("Parsing failed");
    tree = null;
  }
  br.close();
  return tree;
}  // end of program()
```

The call to updateNames() after all the tokens have been processed ensures that the predicate names of each clause are modified to include information about their arity, the clause index, and the total number of clauses. For example: the first clause of `append` will be renamed to `append/3_1/2`, and the second (and final clause) becomes `append/3_2/2`.

The method for parsing a Prolog `list` is:

```
private void list(ParseNode tree, int numSps) throws IOException
/* EBNF: list  ::=   '[' [  args  [ '|' list_tail ] ]  ']'
*/
{ if (debugOn)
    iewriteln(numSps, "list");

  tree.type = ParseType.LIST;
  tree.head = new ParseNode();
  tree.tail = new ParseNode();

  match(tok, Toks.LSQUARE);
  if (tok.symbol() != Toks.RSQUARE) {
    arguments(tree.head, true, numSps+1);   //  list arguments
    if (tok.symbol() == Toks.BAR) {
      match(tok, Toks.BAR);
      listTail(tree.tail, numSps+1);
    }
    else
      tree.tail = null;
    match(tok, Toks.RSQUARE);
  }
  else {   // an empty list
    match(tok, Toks.RSQUARE);
    tree.value = "[]";
    tree.head = null;
    tree.tail = null;
  }
}  // end of list()
```

At the heart of list(), and other parsing methods, are repeated calls to match() which tests the current lexer token and reads in the next one.

The compiler can be invoked with a "-d" option (short for 'diagnostics' or 'debugging') which makes its appearance in the previous methods as code prefaced by a test of debugOn. The diagnostic code in the parser constructs a rudimentary indented list of method calls which parallel the descent over the grammar. For example, the parsing of the first clause of append/2 will be reported as:

```
append            % program
                  %  clauses
                  %   clause
                  %    functor
                  %     atom
([                %     arguments
                  %      term
                  %       list
],L               %      arguments
                  %       term
                  %        var
,L                %       arguments
                  %        term
                  %         var
).
```

One drawback of this approach is the sheer volume of information that pours forth from the compiler. For that reason, all the diagnostics output is sent to the stderr, which can be redirected:

```
>  java -jar BuProlog.jar -d append.pro 2> debug.txt
```

All the normal output from the compiler goes to stdout, and so appears on the command line.

2.3. The Parse Tree

The other task of methods such as list() is to build a parse tree, by linking ParseNode objects together. ParseNode defines a node that can construct a binary tree:

```
// inside ParseNode.java
public ParseType type;
public String value;
public ParseNode head, tail;    // links to the subtrees
```

The generated tree can be quite hard to visualize, the "-p" option of BuProlog.jar causes it to generate a graphical representation:

```
>  java -jar BuProlog.jar -p append.pro
```

The parse tree drawn for append/3 is shown in Figure 6.

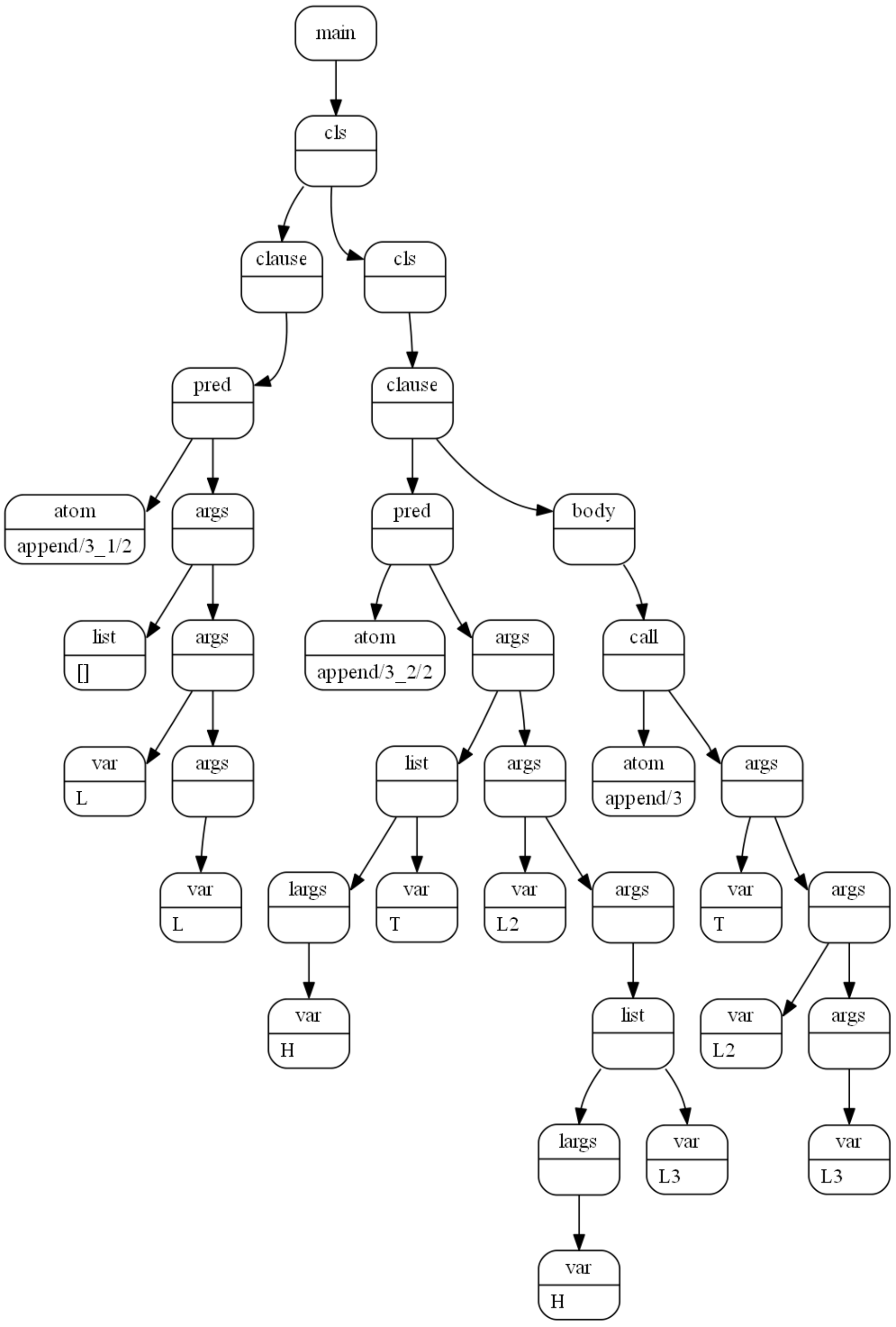


Figure 6. The Parse Tree for append/3.

The text in the top half of each node is a ParseType value, and the bottom half holds either a string (e.g. "append/3_1/2"), or one or two pointers going to subtrees.

As with the execution graphs generated by BuWAM.jar, this approach doesn't scale up to programs much larger than append/3, but it's still useful for getting to grips with how parse trees represent Prolog code.

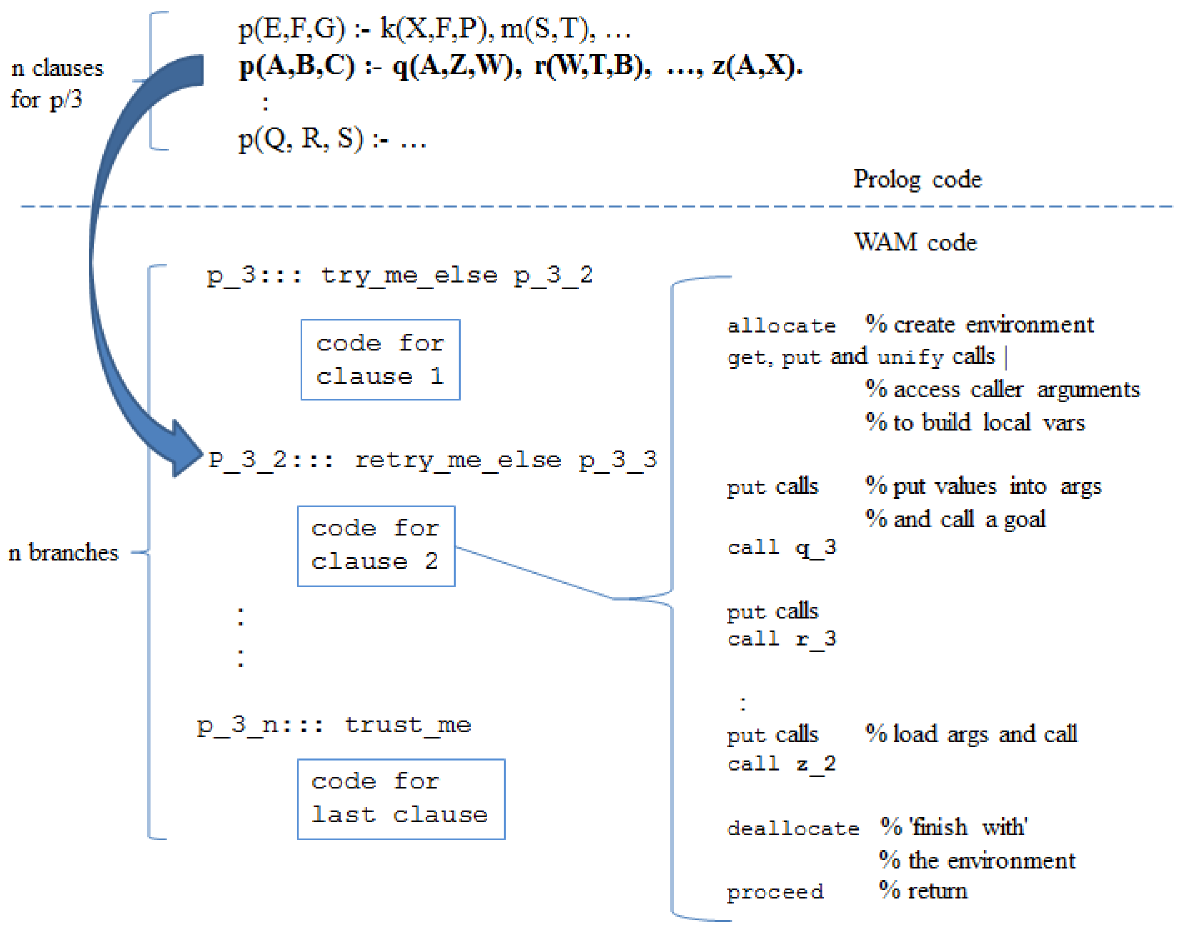
2.4. The Structure of a WAM Program

Before I can give an overview of the compiler's code generation, it's necessary to first explain how typical WAM programs are structured. This section will focus primarily on the syntax of the WAM instruction set, while details about how WAM code is executed being deferred until when the BuWAM is discussed.

After the WAM was proposed by David Warren [7], it soon became the de facto way that Prolog compilers are implemented, albeit with variations in the instruction set. A great introduction to it is the textbook by Hassan Aït-Kaci [2], available online at https://github.com/a-yiorgos/wambook. That site also includes a briefer set of slides which he prepared at around the time the book was published.

Stefan Buettcher's WAM implementation differs in several ways from the WAM described by Warren and Aït-Kaci, which I'll discuss as the presentation progresses.

The template in Figure 7, based on one by Van Roy [5], shows how a collection of clauses for a p/3 predicate is represented in WAM code.



```
p(E,F,G) :- k(X,F,P), m(S,T), …
p(A,B,C) :- q(A,Z,W), r(W,T,B), …, z(A,X).
  :
p(Q, R, S) :- …
```

```
p_3::: try_me_else p_3_2
P_3_2::: retry_me_else p_3_3
  :
  :
p_3_n::: trust_me
```

```
allocate      % create environment
get, put and unify calls |
              % access caller arguments
              % to build local vars

put calls     % put values into args
              % and call a goal
call q_3

put calls
call r_3

  :
put calls     % load args and call
call z_2

deallocate    % 'finish with'
              % the environment
proceed       % return
```

Figure 7. Compiling the p/3 Predicate to WAM Code.

A multi-way switch is defined using a `try_me_else` instruction for the first clause, a `trust_me` for the last clause, and multiple `retry_me_else` instructions for the clauses in between. Each instruction is labeled with "p_3" standing for the predicate's name and arity, and "_number" for the clause index (aside from in the first branch).

The code for each branch has a general structure, based around a call to `allocate` at the start, and `deallocate` and `proceed` at the end. `allocate` creates an environment for the variables local to the clause (the Prolog equivalent of an activation frame) and `deallocate` 'finishes' using that environment at the clause's end.

The code within a clause can be understood in terms of getting, putting and unifying two types of variables, called *permanents* and *arguments*. Permanent variables are local variables created within a clause while it's being executed (they're denoted by a 'Y' and a number). Argument variables are used in two ways within a typical clause: they hold data coming from the calling goal, which is copied into permanents, and secondly they're utilized to store data that's about to be passed to goals called from within the clause. Arguments are denoted by an 'A' and a number.

The easiest way of understanding how the get, put, and unify instructions work together is by comparing a few WAM programs with their Prolog originals.

For example, the Prolog code for append/3:

```
append([],L,L).
append([H|T],L2,[H|L3]) :- append(T,L2,L3).
```

becomes:

```
    append_3::: try_me_else append_3_2          % first clause
                allocate

                get_variable Y0 A0              % 1
                put_constant [] Y1              % 2
                unify_variable Y0 Y1            % 3

                get_variable Y2 A1              % 4

                get_value Y2 A2                 % 5

                deallocate
                proceed

  append_3_2::: trust_me                        % second clause
                allocate

                get_variable Y0 A0              % 1
                unify_list Y3 Y2 Y1             % 2
                unify_variable Y0 Y3            % 3

                get_variable Y4 A1              % 4

                get_variable Y5 A2              % 5
                unify_list Y7 Y2 Y6             % 6
                unify_variable Y5 Y7            % 7

                put_value Y1 A0                 % 8
                put_value Y4 A1                 % 9
                put_value Y6 A2                 % 10
                call append_3

                deallocate
                proceed
```

Extra blank lines have been added, along with numbered comments (after the "%"s).

Consider the query `?-append(A0,A1,A2)`. It will start by matching with the first branch of the WAM code. The resulting sequence of get, put and unify calls are numbered in Figure 8, and correspond to the commented lines in the code above.

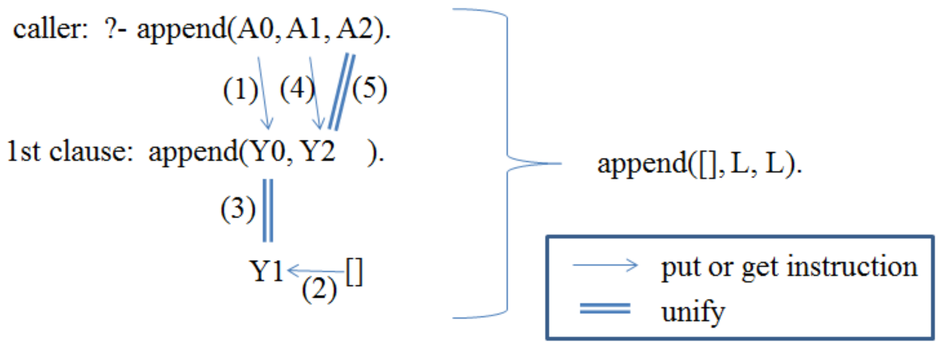
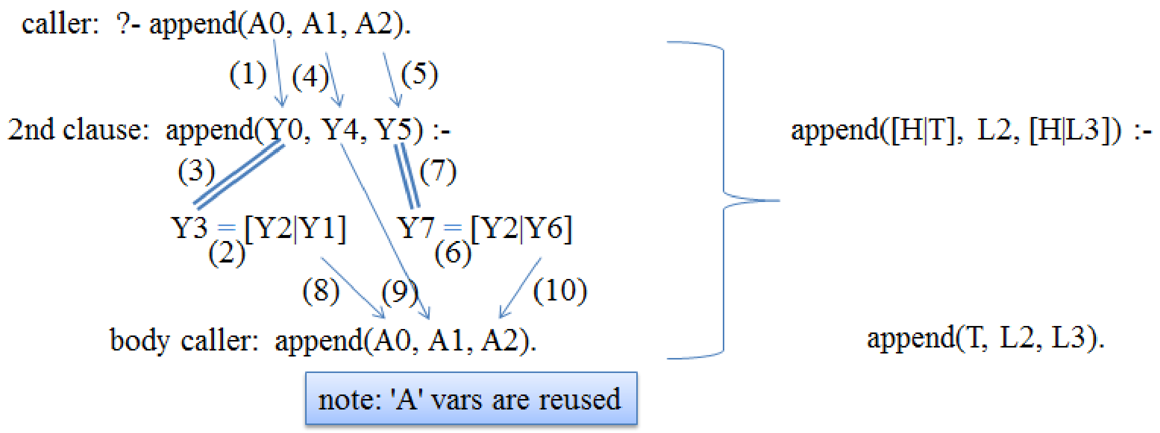


Figure 8. Matching a query to the first append/3 clause.

The numbered instructions 1, 2, and 3 deal with matching the first input argument (A0) with the first permanent variable (Y0) in the clause. The fourth instruction deals with assigning A1 to Y2, and the final argument (A3) is handled by unification with Y2 (there's no need for a Y3).

If this matching fails, then the `try_me_else` instruction will cause execution to jump to the second branch beginning with the label `append_3_2` (i.e. the second clause). The next round of get, put and unify calls are numbered in Figure 9 and in the code above.



caller: ?- append(A0, A1, A2).
(1)
(4)
(5)
2nd clause: append(Y0, Y4, Y5) :-
(3)
(7)
Y3 = [Y2|Y1]
(2)
Y7 = [Y2|Y6]
(6)
(8)
(9)
(10)
body caller: append(A0, A1, A2).
note: 'A' vars are reused
append([H|T], L2, [H|L3]) :-
append(T, L2, L3).

Figure 9. Matching a query to the second append/3 clause.

A similar mix of get, put, and unify instructions is used to match the query's input arguments to permanent variables. However, the clause's call to append/3 means that there's a second phase carried out by the instructions numbered 8-10. They reuse the argument variables A0, A1 , and A2 to store data that needs to be passed to the append_3 call after instruction 10.

This second branch utilizes a variation of the `unify` instruction: `unify_list` which takes three arguments, and specifically supports list unification:

`unify_list List Head Tail` ≡ List = [Head | Tail]

There's also a `unify_struct`, which is employed in the WAM code for the following address/3 fact:

```
address(john, street(19, brooke), liverpool).
```

It becomes:

```
   address_3::: try_me_else address_3_2
                allocate

                get_constant john A0           % 1

                get_variable Y0 A1             % 2
                put_constant street Y1         % 3
                put_constant 19 Y2             % 4
                put_constant brooke Y3         % 5
                put_constant [] Y4             % 6
                unify_list Y5 Y3 Y4            % 7
                unify_list Y6 Y2 Y5            % 8
                unify_struc Y7 Y1 Y6           % 9
                unify_variable Y0 Y7           % 10

                get_constant liverpool A2      % 11

                deallocate
                proceed
```

When the query `?-address(A0,A1,A2)` is matched, the sequence of get, put and unify calls are numbered as in Figure 10, and also in the code above.

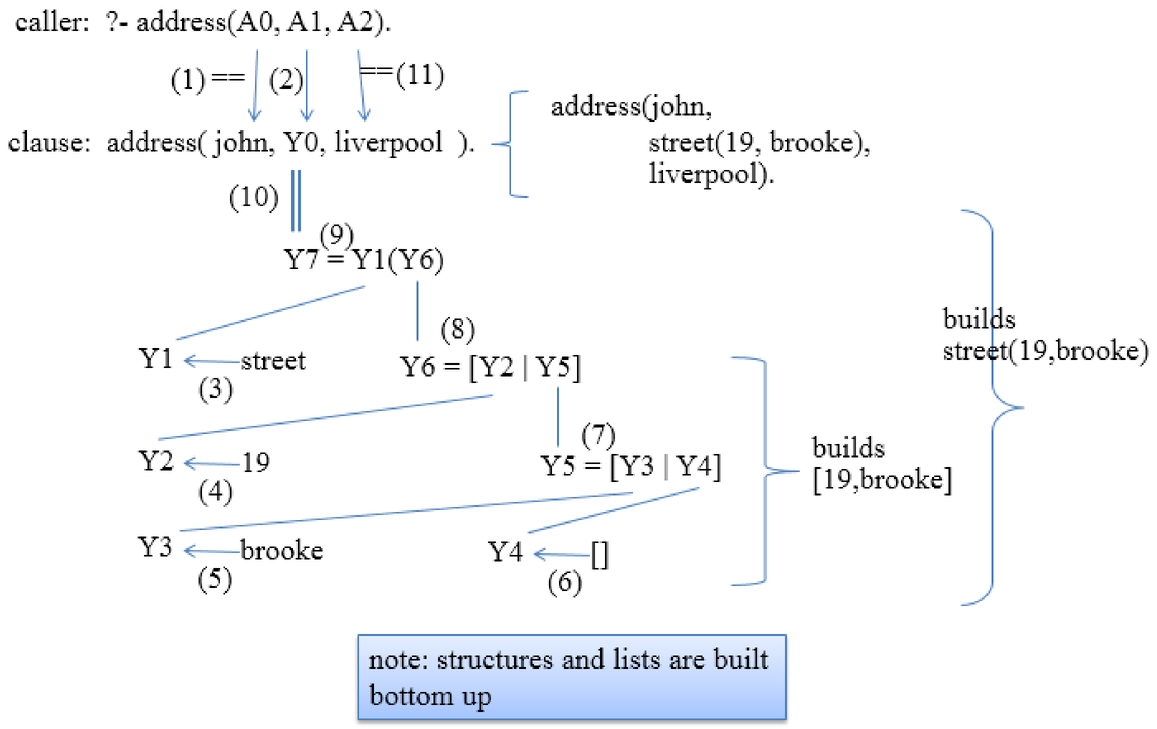


Figure 9. Matching a Query to the address Clause.

Most of the instructions are involved with matching the second argument (A1) of the query with the structure that occupies the second position in the head of the clause (i.e. `street(19,brooke)`). The match is built up from the constituent parts of a structure, and then calls `unify_struc` to deal with term-based unification:

`unify_struc Structure Functor Args_list` ≡ Structure = Functor(Args_list)

Since a structure can have any number of arguments (including none), the BuWAM stores those arguments in a list. The `street` term has two arguments which requires the list to be built up in three stages – first an empty list is stored in Y4, and then two elements are added to its front (Y3 and Y2) by list unification.

2.5. A Summary of the BuWAM Instructions

The preceding section's examples used 14 of the 18 instructions in Stefan Buettcher's WAM. They can be grouped into six categories, as in Table 1.

1. Get	Used when a goal is matched against the head of a clause to load arguments from that goal (its 'A' variables) into the clause variables (Y's).
2. Put	Utilized before calling a goal in the body of a clause to load

	clause variables (Y's) into the goal's arguments (A's).
3. Unification	Employed with get and put instructions to help link variables. The list and structure versions pull apart and construct those data structures.
4. Calling	Used to execute a goal, to return, and to set up the environment for a clause that matches the goal.
5. Branching.	Switch between clauses that might match a goal, and to create choice point information.
6. Cuts	Handles Prolog cuts.

Table 1. Types of WAM Instruction.

A complete list of all the instructions, with a short description of each one, appears in Table 2. (A "// NEW" comment indicates that this instruction isn't in the original WAM.)

1. Get Instructions	
get_variable Y A	Y ← A Only used for the first occurrence of Y inside the clause’s head.
get_value V A	Does Y = A? Used for subsequent occurrences of Y inside the clause’s head.
get_constant c A	Does A == c?
2. Put Instructions	
put_value Y A	A ← Y

put_constant c A
put_constant c Y
	A ← c
	Y ← c

create_variable Y name	Create a new variable with a printable name. Only used for 'Q' variables which appear in a user query. // NEW
3. Unification	
unify_variable v1 v2	v1 = v2
unify_list v1 v2 v3	v1 = [v2 | v3] // NEW

unify_struc v1 v2 v3	v1 = v2(v3) // NEW
	v3 is a *list* of the functor's arguments.

4. Calling	
call label_arity	Execute the predicate with the specified label and arity by jumping to its first clause.
proceed	Return from a clause.
allocate	Create an environment for this clause.
deallocate	Restore the previous environment, but don't delete the current one.
halt	The query's evaluation is terminated. //

	NEW
5. Branching	
(re)try_me_else label	Create a choice point which jumps to the clause starting at label if the code of the current clause following this instruction fails.
trust_me	If the code of the clause following this instruction fails, then the execution of the goal will fail.
6. Cuts	
get_level Y	Y is assigned this clause's choice point.
cut Y	Discard all the choice points from the current one back to the one stored in Y.

Table 2. Stefan Buettcher's WAM Instruction Set.

The four instructions that haven't appeared yet are `create_variable`, `halt`, `cut_level`, and `cut`. The first two are used in the WAM code generated for user queries, and (as you might expect) the cut instructions come into play when Prolog uses "!".

2.6. Other WAMs

For readers familiar with the WAM, Buettcher's version differs in several ways from the one described by Warren [7], Aït-Kaci [2], and Van Roy [5].

The standard WAM utilizes two types of variable inside a clause called *permanents* and *temporaries*. This distinction allows registers to be reused more efficiently.

The original WAM doesn't offer `unify_list` and `unify_struc` instructions. Instead, the functionality is supported by `set` instructions and `get_list` and `get_structure` operations that work with a version of `unify_variable` that only takes a single argument. For example:

```
% in a standard WAM
get_structure p/3 V0     %  X0 = p
unify_variable V1        %    (V1,
unify_variable V2        %     V2,
unify_variable V3        %     V3 )
```

Buettcher's WAM uses explicit arguments in its unification instructions, which is arguably easier to understand, although it means that lists and structures generally require more lines of code to be constructed or pulled apart.

All of the WAM instructions related to optimizations, such as clause indexing, and last call optimization are missing from Buettcher's WAM; BuProlog is primarily intended as a teaching tool.

3. Code Generation in the Compiler

A look back at the class diagram for BuProlog in Figure 4 shows that code generation utilizes three classes: WAMProgram, WAMClause, WAMStatement, and two enumerations: OpWAM, and Builtin. The main() function of BuProlog.java calls WAMProgram to build a list of WAMClause objects, one for each clause in the program. In turn, each WAMClause object holds a list of WAMStatement objects, representing the sequence of instructions that make up the clause.

Another way of understanding the code generation process is that the compiler's parse tree (e.g. the tree for append/3 in Figure 6) is converted into lines of WAM code (e.g. append.wam listed on p.13). To help a compiler design student understand this transformation, BuProlog can be called with the "-d" and "-p" options:

```
> java -jar BuProlog.jar -dp append.pro 2> debug.txt
```

The code generation section of the diagnostic output is shown below:

```
Generating WAM Code:

      % clause
      %   predGen: append/3_1/2
      |   [ append_3::: try_me_else append_3_2 ]
      %     predArgGen: list:[]
      |     [ get_variable Y0 A0 ]
      %       listGen
      |       [ put_constant [] Y1 ]
      |     [ unify_variable Y0 Y1 ]
      %     predArgGen: var:L
      |     [ get_variable Y2 A1 ]
      %     predArgGen: var:L
      |     [ get_value Y2 A2 ]
      | [ allocate ] (1)
      | [ deallocate ]
      | [ proceed ]

      % clause
      %   predGen: append/3_2/2
      |   [ append_3_2::: trust_me ]
      %     predArgGen: list:
      |     [ get_variable Y0 A0 ]
      %       listGen
      %         callArgGen: var:T
      %         listArgsGen
      %           callArgGen: var:H
      |         [ unify_list Y3 Y2 Y1 ]
      |     [ unify_variable Y0 Y3 ]
      %     predArgGen: var:L2
      |     [ get_variable Y4 A1 ]
      %     predArgGen: list:
      |     [ get_variable Y5 A2 ]
      %       listGen
      %         callArgGen: var:L3
      %         listArgsGen
      %           callArgGen: var:H
      |         [ unify_list Y7 Y2 Y6 ]
      |     [ unify_variable Y5 Y7 ]
      %   bodyGen
      %     callGen: append/3
      %       callArgGen: var:T
      |       [ put_value Y1 A0 ]
      %       callArgGen: var:L2
      |       [ put_value Y4 A1 ]
      %       callArgGen: var:L3
      |       [ put_value Y6 A2 ]
      |     [ call append_3 ]
      | [ allocate ] (1)
      | [ deallocate ]
      | [ proceed ]
```

The output is ab interleaving of two kinds of data. Each line either begins with a "%" which denotes that a code generation function was called, while "|" lines lists the generated WAM code. The lines are also indented to reflect the nesting of the function calls inside WAMClause.

3.1. Code Generation inside WAMClause

The WAMClause methods will be explained by focusing on the two parts of the parse tree for append/3 highlighted in Figure 10.

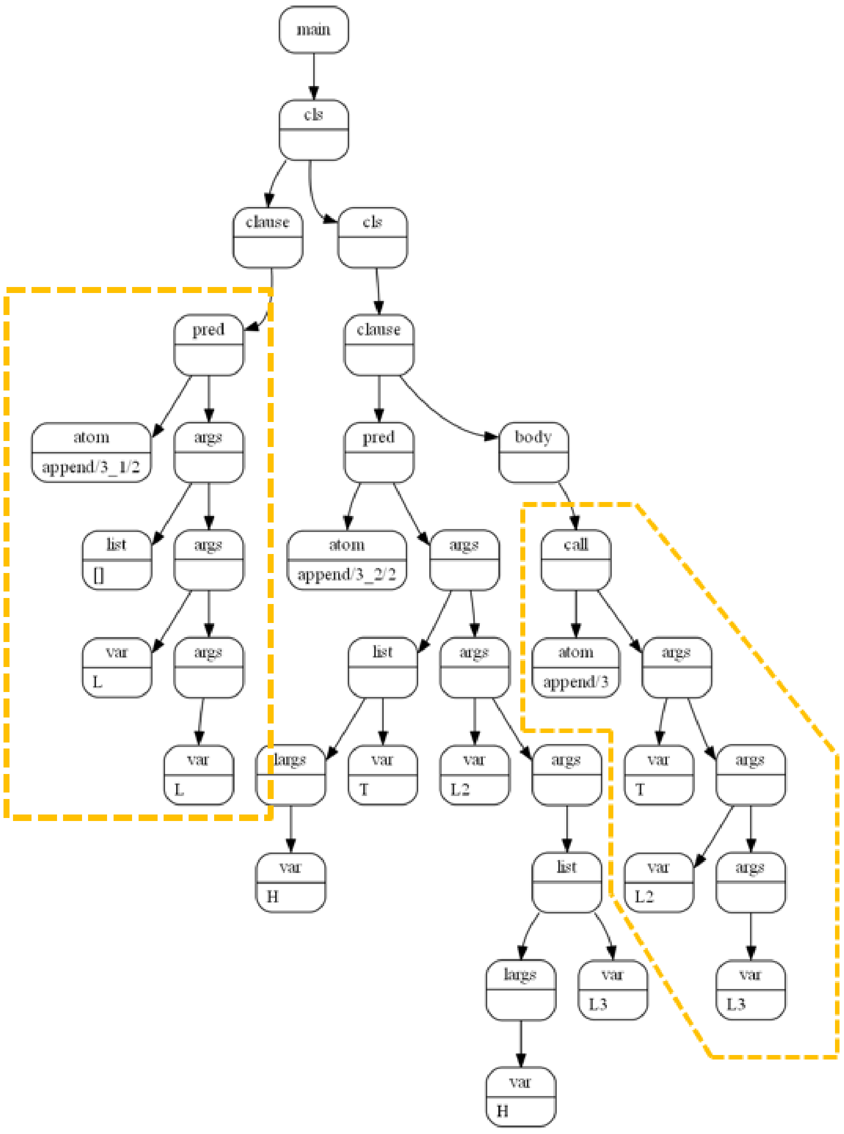


Figure 10. Highlighted Parse Tree for append/3.

3.2. Compiling a Fact

The left-most box in Figure 10 contains the subtree for the first clause of append/3:

```
append([],L,L).
```

It becomes the WAM code:

```
    append_3::: try_me_else append_3_2
                allocate
                get_variable Y0 A0
                put_constant [] Y1
                unify_variable Y0 Y1
                get_variable Y2 A1
                get_value Y2 A2
                deallocate
                proceed
```

The relevant part of the diagnostic output is:

```
% clause
%   predGen: append/3_1/2
|   [ append_3::: try_me_else append_3_2 ]
%     predArgGen: list:[]
|     [ get_variable Y0 A0 ]
%       listGen
|       [ put_constant [] Y1 ]
|     [ unify_variable Y0 Y1 ]
%     predArgGen: var:L
|     [ get_variable Y2 A1 ]
%     predArgGen: var:L
|     [ get_value Y2 A2 ]
| [ allocate ] (1)
| [ deallocate ]
| [ proceed ]
```

This indicates that the WAMClause() constructor (labeled as `clause`) called predGen(), which called predArgGen() three times (to deal with the fact's three arguments). The first of those argument (a list) was processed by listGen().

The relevant lines of WAMClause() are:

```
// global variables
private ArrayList<WAMStatement> stmts = new ArrayList<>();
    // a clause is encoded as a series of WAM statements


public WAMClause(ParseNode t, boolean debugOn)
// parse a CLAUSE or a QUERY
{
  int numSps = 1;
  this.debugOn = debugOn;
  if (t.type == ParseType.CLAUSE) {
    varPrefix = "Y";
    predGen(t.head, numSps+2);   // a predicate head
    int bodyCalls = 0;
    if (t.tail != null)
      bodyCalls = bodyGen(t.tail, numSps+2);  // an optional body

    if ((varNames.size() > 0) || (bodyCalls > 0)) {
      addToStmts(numSps, 1, new WAMStatement(OpWam.ALLOCATE));
      addToStmts(numSps, new WAMStatement(OpWam.DEALLOCATE));
    }
    addToStmts(numSps, new WAMStatement(OpWam.PROCEED));
  }
  else if (t.type == ParseType.QUERY) {
    // this branch is called from BuWAM.java
        :
  }
  else
    System.out.println("Error: expected a CLAUSE or QUERY node");
}  // end of WAMClause()
```

The code generation functions store WAM instructions in a global `stmts` list by calling addToStmts(). The `allocate` instruction is stored as the first line, while `deallocate` and `proceed` are appended to the end.

predGen() does some predicate name conversion (in this case, changing `append/3_1/2` in the parse tree (see Figure 10) into `append_3`, and then enters a loop which moves down through the `ARGS` nodes in the tree, calling predArgGen() for each argument.

predArgGen() generates `GET` and `UNIFY` WAM instructions to copy data from the query's arguments into the head of the clause (e.g. see Figure 8). The function's code:

```
private void predArgGen(ParseNode t, int argCount, int numSps)
// parse the argument subtree of a PRED's ARGS node
{
  if (t.type == ParseType.CONSTANT)
    addToStmts(numSps, new WAMStatement(OpWam.GET_CONSTANT,
                                t.value, "A" + argCount));
  else if (t.type == ParseType.STR)
    addToStmts(numSps, new WAMStatement(OpWam.GET_CONSTANT,
                                "'"+t.value+"'", "A" + argCount));
  else if (t.type == ParseType.VARIABLE) {
    if (isFirstUse(t.value))
      addToStmts(numSps, new WAMStatement(OpWam.GET_VARIABLE,
                         renameVar(t.value), "A" + argCount));
    else
      addToStmts(numSps, new WAMStatement(OpWam.GET_VALUE,
                         renameVar(t.value), "A" + argCount));
  }
  else if (t.type == ParseType.STRUCTURE) {
    String v = newVar();
    addToStmts(numSps, new WAMStatement(OpWam.GET_VARIABLE,
                                v, "A" + argCount));
    String structVar = structGen(t, numSps+2);
    addToStmts(numSps, new WAMStatement(OpWam.UNIFY_VARIABLE,
                                v, structVar));
  }
  else if (t.type == ParseType.LIST) {
    String v = newVar();
    addToStmts(numSps, new WAMStatement(OpWam.GET_VARIABLE,
                                v, "A" + argCount));
    String listVar = listGen(t, numSps+2);
    addToStmts(numSps, new WAMStatement(OpWam.UNIFY_VARIABLE,
                                v, listVar));
  }
  else
    System.out.println("Error: not a valid PRED argument node");
}  // end of predArgGen()
```

predArgGen() utilizes a series of if-tests that respond to the type of the node. For example, the first argument of the append/3 fact is a list constant (`"[]"`), which causes execution to branch to the final else-if. A `get_variable` instruction is stored, then several lines are output for processing the list by listGen(), and then a `unify_variable` line is added. The `get_variable` instruction creates a new local variable by calling newVar(), and generates an argument variable by appending `"A"` to an `argCount` value.

3.3. Compiling a Body Goal

The highlighted region on the right of Figure10 contains the subtree for the body goal in the second clause of append/3:

`append([H|T],L2,[H|L3])` **`:- append(T,L2,L3).`**

It is converted into the WAM code highlighted in the following:

```
  append_3_2::: trust_me
                allocate
                get_variable Y0 A0
                unify_list Y3 Y2 Y1
                unify_variable Y0 Y3
                get_variable Y4 A1
                get_variable Y5 A2
                unify_list Y7 Y2 Y6
                unify_variable Y5 Y7
                put_value Y1 A0
                put_value Y4 A1
                put_value Y6 A2
                call append_3
                deallocate
                proceed
```

The relevant part of the diagnostic output is:

```
      % clause
      %   predGen: append/3_2/2
      |   [ append_3_2::: trust_me ]
            : // lines not shown
      %   bodyGen
      %     callGen: append/3
      %       callArgGen: var:T
      |       [ put_value Y1 A0 ]
      %       callArgGen: var:L2
      |       [ put_value Y4 A1 ]
      %       callArgGen: var:L3
      |       [ put_value Y6 A2 ]
      |     [ call append_3 ]
           : // lines not shown
```

The second clause is also processed by WAMClause() and predGen(), but this time bodyGen() deals with its body goals. bodyGen() executes callGen() for each of the `call` nodes of the tree.

The relevant parts of callGen() are:

```
private void callGen(ParseNode t, int numSps)
// parse the CALL subtree
{
  if (t.type == ParseType.CALL) {
    if (t.head.value.equals("CUT")) {
      // process a cut
           :
    }
    else {  // deal with an atom head and args in the tail
      if (t.tail != null) {    // first the arguments
        ParseNode args = t.tail;
        int argCount = 0;
        while (args != null) {
          if (args.type == ParseType.ARGS) {
            callArgGen(args.head, argCount, numSps+2);
            argCount++;
          }
          else
            System.out.println("Error: expected an ARGS node");
          args = args.tail;
        }
      }
      // now call the atom
      if (!Builtin.contains(t.head.value)) { // user-defined pred
        String[] nameToks = t.head.value.split("/");
                        // format is name/arity
        String name = nameToks[0];
        int arity = Integer.parseInt(nameToks[1]);
        addToStmts(numSps,
              new WAMStatement(OpWam.CALL, name + "_" + arity));
      }
      else
        addToStmts(numSps, new WAMStatement(OpWam.CALL, t.head.value));
    }
  }
  else
    System.out.println("Error: expected a CALL node");
}  // end of callGen()
```

callGen() converts the arguments of the body goal into WAM code before the goal is processed. A while loop invokes callArgGen() for each argument:

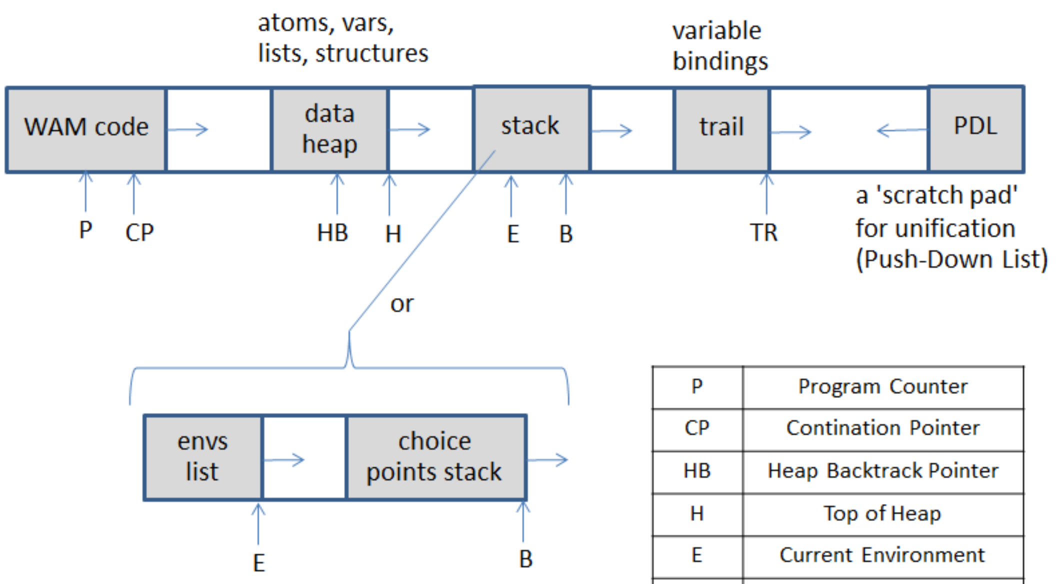
```
private String callArgGen(ParseNode t, int argCount, int numSps)
{  return callArgGen(t, argCount, false, numSps);  }


private String callArgGen(ParseNode t, int argCount,
                           boolean isInner, int numSps)
// parse the argument subtree of a CALL;
// isInner means that the argument being processed
// is inside a list or a structure
{
  String lastVar = null;
  if (t.type == ParseType.CONSTANT)
    if (isInner) {
      lastVar = newVar();
      addToStmts(numSps, new WAMStatement(OpWam.PUT_CONSTANT,
                                 t.value, lastVar));
    }
    else
      addToStmts(numSps, new WAMStatement(OpWam.PUT_CONSTANT,
                               t.value, "A" + argCount));
  else if (t.type == ParseType.STR)
    if (isInner) {
      lastVar = newVar();
      addToStmts(numSps, new WAMStatement(OpWam.PUT_CONSTANT,
                                 "'"+t.value+"'", lastVar));
    }
    else
      addToStmts(numSps, new WAMStatement(OpWam.PUT_CONSTANT,
                              "'"+t.value+"'", "A" + argCount));
  else if (t.type == ParseType.VARIABLE) {
    if ( varPrefix.equals("Q") &&
         isFirstUse(t.value) ) // set Q's name
      addToStmts(numSps, new WAMStatement(OpWam.CREATE_VARIABLE,
                              renameVar(t.value), t.value));
    lastVar = renameVar(t.value);
    if (!isInner)
      addToStmts(numSps, new WAMStatement(OpWam.PUT_VALUE,
                                 lastVar, "A" + argCount));
  }
  else if (t.type == ParseType.STRUCTURE) {
    lastVar = structGen(t, numSps+2);
    if (!isInner)
      addToStmts(numSps, new WAMStatement(OpWam.PUT_VALUE,
                               lastVar, "A" + argCount));
  }
  else if (t.type == ParseType.LIST) {
    lastVar = listGen(t, numSps+2);
    if (!isInner)
      addToStmts(numSps, new WAMStatement(OpWam.PUT_VALUE,
                               lastVar, "A" + argCount));
  }
  else
```

```
    System.out.println("Error: unknown arg type for CALL at pos " +
                       (argCount+1));

  return lastVar;
}  // end of callArgGen()
```

callArgGen() is invoked with an isInner flag, which affects how it generates code. If isInner is true then the argument being processed is inside a list or structure, and so any new variables should be locals rather than "A" variables. However, aside from this complication, callArgGen() is structured in much the same way as predArgGen().

4. The BuWAM Virtual Machine

Before the BuWAM 's Java code can be understood, the main data structures employed in the standard WAM need to be explained. For this purpose, Warren [7] utilized a diagram somewhat similar to the one in Figure 11.



P	Program Counter
CP	Contination Pointer
HB	Heap Backtrack Pointer
H	Top of Heap
E	Current Environment
B	Current Choice Point (Backtrack Pointer)
TR	Top of the Trail

Figure 11. The WAM in Memory.

The WAM employs five (or six) main data structures, stored at various locations in memory. Most of them are stacks which grow in a certain direction as data is added, and use pointers to store the current tops of those structures.

The **heap** holds data in a tag-based format, organized across multiple words of memory. Figure 12 shows the first example of this format from Aït-Kaci's book.

0	STR	1
1	$h/2$	
2	REF	2
3	REF	3
4	STR	5
5	$f/1$	
6	REF	3
7	STR	8
8	$p/3$	
9	REF	2
10	STR	1
11	STR	5

Figure 12. The Heap representation of the term p(Z, h(Z,W),f(W).

The **stack** holds environment and choice point records, which some versions of the WAM encoding it as two data structures, as in Figure 11. The environments roughly correspond to activation records in virtual machines for imperative languages, holding the variables used during the execution of a particular clause, and a return address. The choice point records handle the tricky problem of dealing with Prolog's backtracking capabilities.

The **trail** is a list of variable bindings which must be rolled back when Prolog backtracks to an earlier choice point. The **PDL** is an area used for temporary calculations during unification.

One drawback of this memory-based model is that its associated algorithms naturally have implementations where data fields are denoted by integer offsets from one or more pointers. For instance, Aït-Kaci's high-level code for the `allocate` instruction is expressed as in Figure 13.

```
if E > B
  then newE ← E + CODE[STACK[E + 1] − 1] + 2
  else newE ← B + STACK[B] + 8;
STACK[newE] ← E;
STACK[newE + 1] ← CP;
E ← newE;
P ← P + instruction_size(P);
```

Figure 13. Aït-Kaci's implementation of `allocate`.

Arguably Buettcher's version of the WAM is easier to understand because it manipulates fields inside objects stored in explicit Java stacks and lists. This point can be seen in Figure 14, a class diagram for the BuWAM, which corresponds to the memory model in Figure 11.

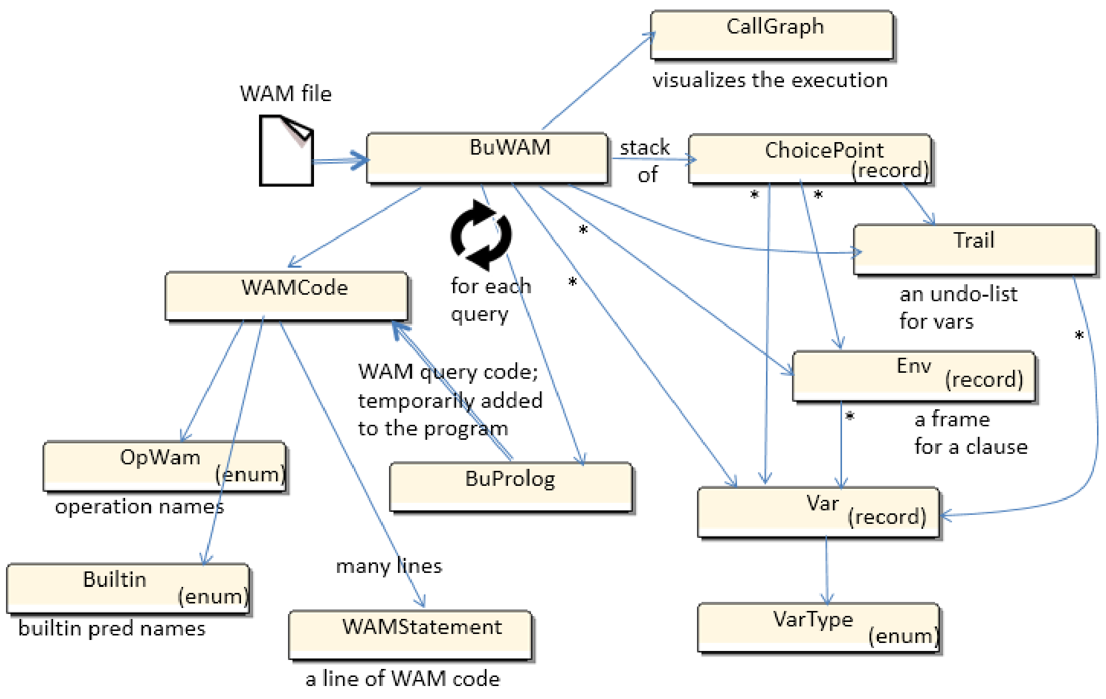


Figure 14. A Class Diagram for the BuWAM.

The BuWAM utilizes a WAMCode object to hold code, several lists of Var objects to point to data, a stack of choice point objects, a list of environments, and a Trail object. Unification calculations are carried out inside the BuWAM object.

4.1. WAM Initialization

The BuWAM is supplied with a WAM filename as a command line argument:

```
> java -jar BuWAM append.wam
```

The file is read in line-by-line, and each line is tokenized and used to instantiate a WAMStatement object. The resulting ArrayList inside WAMCode corresponds to the code area of the original WAM (see Figure 11). In addition, the labels at the start of each clause are stored in a TreeMap along with their 'line' number in the list.

If the BuWAM is invoked with a "-d" option, then the loaded code and the TreeMap labels are printed out, as shown below:

```
----------------- WAM code ----------------
(0000)      append_3::: try_me_else append_3_2
(0001)                  allocate
(0002)                  get_variable Y0 A0
(0003)                  put_constant [] Y1
(0004)                  unify_variable Y0 Y1
(0005)                  get_variable Y2 A1
(0006)                  get_value Y2 A2
(0007)                  deallocate
```

```
(0008)                  proceed
(0009)    append_3_2::: trust_me
(0010)                  allocate
      :   // more lines
(0022)                  deallocate
(0023)                  proceed
---------------------------------------------
Labels=Jumps: {append_3=0, append_3_2=9}
```

4.2. Parsing a Query

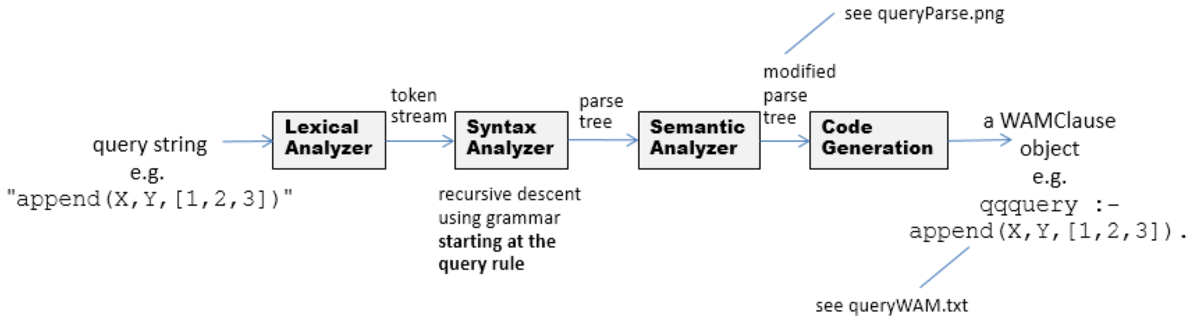
The BUWAM's REPL is in its main() function:

```
// in BUWAM.main()
String line;
do {
  wam.writeln("");
  wam.write("?- ");
  line = wam.readline();
  wam.writeln("");
} while (wam.runQuery(line));
```

runQuery() parses the query string into a WAMClause object, and executes it:

```
// part of runQuery()
WAMClause query = parseQuery(queryStr);
if (query == null)
  return true;   // an error but return to the REPL
executeQuery(query);
```

The stages in parsing a query are summarized by Figure 15.



see queryParse.png
modified parse tree
token stream
parse tree
query string e.g. "append(X,Y,[1,2,3])"
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Code Generation
a WAMClause object e.g. qqquery :- append(X,Y,[1,2,3]).
recursive descent using grammar starting at the query rule
see queryWAM.txt

Figure 15. Parsing a Query.

Most of the steps shown in Figure 15 reuse code from the BuProlog compiler by utilizing three 'tricks': the lexical analyzer is passed a string rather than a filename, the recursive descent parser starts from the `query` grammar rule rather than the `program` rule, and the generated WAM code is treated like the body of a `qqquery/0` clause.

parseQuery() is complicated by code to print diagnostics and generate pictures, but if that's stripped away, there're only three important lines left:

```
// part of parseQuery()
BuProlog qc = new BuProlog(queryStr, debugOn);
ParseNode queryTree = qc.query();
WAMClause  query = new WAMClause(queryTree, debugOn);
```

The BuProlog constructor determines that the query string isn't a filename, and sets up the lexer and parser differently:

```
public BuProlog(String fnm, boolean debugOn) throws IOException
{
  this.debugOn = debugOn;
  if (fnm.endsWith(".pro")) {
    br = new BufferedReader(new FileReader(fnm));
    lexer = new Lexer(br, true); // always show lex output
  }
  else {
    queryParsing = true;
    br = new BufferedReader(new StringReader(fnm));
                // assume the input is a query string
    lexer = new Lexer(br, debugOn);
  }
  getToken();
}
```

The parser is driven by BuProlog.query():

```
public ParseNode query() throws IOException
// EBNF:  query ::=  body '.'
{
  int numSps = 1;
  ParseNode tree = new ParseNode();
  tree.type = ParseType.QUERY;
  tree.head = new ParseNode();
  tree.tail = new ParseNode();

  body(tree.tail, numSps+1);
  match(tok, Toks.PERIOD);
  if (tok.symbol() == Toks.EOF)  {
    tree.head.type = ParseType.PREDICATE;
    tree.head.tail = null;
    tree.head.head = new ParseNode();
    tree.head.head.type = ParseType.ATOM;
    tree.head.head.value = "qqquery/0_1/1";  // /arity_count/total
    tree.head.tail = null;
  }
  else {
    writeln("Parsing failed");
    tree = null;
  }
  br.close();

  return tree;
}  // end of query()
```

The query is processed by BuProlog.body() which builds a tree that's added to nodes representing the head of the clause `qqquery :- body`. Rather confusingly, the name of this predicate must be "qqquery/0_1/1", since the arity, clause index, and total number of clauses must be included. If BuWAM.jar is invoked with the "-p" option, a picture of the parse tree is stored in queryParse.png. Figure 16 shows the tree generated for `?-append(X,Y,[1,2,3]).`

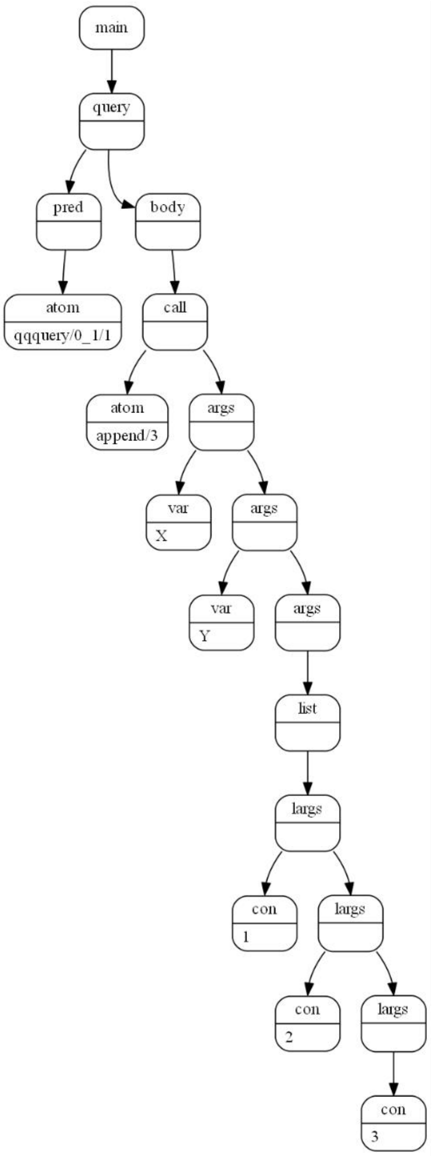


Figure 13. The parse tree for ?- append(X,Y,[1,2,3]).

The top-level "query" node becomes important when the tree is passed to the WAMClause constructor back in BuWAM.parseQuery(). The constructor employs a two-way branch which determines if the node type is ParseType.CLAUSE or ParseType.QUERY:

```
public WAMClause(ParseNode t, boolean debugOn)
{
  int numSps = 1;
  this.debugOn = debugOn;
  if (t.type == ParseType.CLAUSE) {
    // this branch is used by BuProlog.java
    varPrefix = "Y";
        :   // lines not shown
  }
  else if (t.type == ParseType.QUERY) {
    // this branch is called from BuWAM.java
    varPrefix = "Q";
    predGen(t.head, numSps+2);   // a dummy predicate name (qqquery)
    bodyGen(t.tail, numSps+2);   // a body
    addToStmts(numSps, new WAMStatement(OpWam.HALT));
  }
  else
    System.out.println("Error: expected a CLAUSE or QUERY node");
}  // end of WAMClause()
```

WAMClause uses the existing predGen() and bodyGen() functions to generate WAM code, but with the varPrefix global set to "Q" rather than "Y". This means that "Q" (query) variables will be used rather than "Y" (permanent) variables in the WAM statements.

The code generated for `qqquery :- append(X,Y,[1,2,3])` is given below, with the addition of numbered comments:

```
   qqquery_0::: trust_me

                create_variable Q0 X    % 1
                put_value Q0 A0         % 2

                create_variable Q1 Y    % 3
                put_value Q1 A1         % 4

                put_constant 1 Q2       % 5
                put_constant 2 Q3       % 6
                put_constant 3 Q4       % 7
                put_constant [] Q5      % 8
                unify_list Q6 Q4 Q5     % 9
                unify_list Q7 Q3 Q6     % 10
                unify_list Q8 Q2 Q7     % 11
                put_value Q8 A2         % 12

                call append_3
                halt
```

These comment numbers also appear in Figure 14 to label the bindings that match the query to a goal.

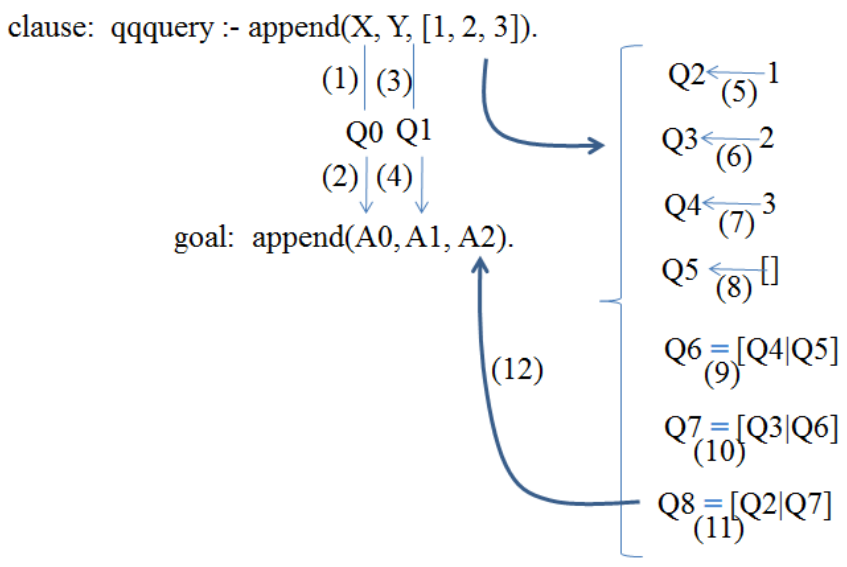


Figure 14. Matching the body of `qqquery:-append(X,Y,[1,2,3]).`

The `create_variable` instruction is utilized on lines (1) and (3) to assigns printable names to Q variables. Q0 is given the name "X", and Q1 gets "Y". These names mean that although there are nine Q variables in Figure 14, only the bindings for Q0 and Q1 will be printed when the query succeeds.There's no `create_variable` in the original WAM, although its `put_variable` is somewhat similar.

4.3. Executing a Query

The query is executed by BuWAM.executeQuery(), which is shown below (but without its copious diagnostics code):

```
private void executeQuery(WAMClause query)
{
  wc.deleteQuery();    // remove any previous query from the program code

  // add the query to the end of the program code
  wc.addQuery(query);

  pc = wc.getQueryPos();
  String answer = "";
  do {
    runWAM();
    if (queryFailed) {
      writeln("No.");
      break;
    }

    // display query variable bindings
    writeln(getQVars());

    // if there are some choicepoints,
    // ask the user if they should be tried via backtracking
    if (!choices.isEmpty()) {
      write("\nMore? (y/n) ");
      answer = readline();
    }
    else
      break;

    // has the user decided to backtrack?
    if (answer.equals("y") ||  answer.equals("yes"))
      backtrack();
  } while ( answer.equals("y") || answer.equals("yes"));
}  // end of executeQuery()
```

The function begins by deleting any earlier query stored with the program code and then adds the current query. The query is added after the program, as reflected by the line numbers in the diagnostic output:

```
(0024)     qqquery_0::: trust_me
(0025)                  create_variable Q0 X
(0026)                  put_value Q0 A0
(0027)                  create_variable Q1 Y
(0028)                  put_value Q1 A1
(0029)                  put_constant 1 Q2
(0030)                  put_constant 2 Q3
(0031)                  put_constant 3 Q4
(0032)                  put_constant [] Q5
(0033)                  unify_list Q6 Q4 Q5
(0034)                  unify_list Q7 Q3 Q6
(0035)                  unify_list Q8 Q2 Q7
(0036)                  put_value Q8 A2
(0037)                  call append_3
(0038)                  halt
--------------------------------------------
Labels=Jumps: {append_3=0, append_3_2=9, qqquery_0=24}
```

The append/3 predicate occupies lines 0 to 23, and the query is at lines 24 to 38.

runWAM() may be called several times for each query if there are choice points in the result and the user requests alternative answers.

runWAM() enters a loop to execute the lines of WAM code in the query, continuing until the program counter is negative (indicating an error) or the query's `halt` instruction is reached. In outline, the function has the form:

```
// globals
private int pc = 0;        // program counter


public void runWAM()
{
  while (pc >= 0) {  // pc < 0 happens on a jump error
    queryFailed = false;
    WAMStatement s = wc.getStatement(pc);

    OpWam op = s.getOp();
    if (op == OpWam.PUT_VALUE)
      put_value(s.getArg(0), s.getArg(1));
    else if (op == OpWam.PUT_CONSTANT)
      put_constant(s.getArg(0), s.getArg(1));
          : // many more lines
          :
    else if (op == OpWam.CALL)
      call(s.getArg(0), s.getJump());
    else if ((op == OpWam.TRY_ME_ELSE) ||
           (op == OpWam.RETRY_ME_ELSE) )
      try_me_else(s.getLabel(), s.getJump());
        :  // more lines
        :
    else if (op == OpWam.HALT)
      break;
    else if (op == OpWam.NO_OP)
      pc++;
    else {
      writeln("Invalid op on line " + String.format("%04d", pc));
      backtrack();
    }
  }  // end of while (pc >= 0)

  if (queryFailed) {
    while (!choices.isEmpty())
      backtrack();
  }
} // end of runWAM()
```

I'll discuss the processing of the instructions by dividing them into the categories used in Table 1: Get, Put, Unification, Calling, Branching, and Cuts.

4.4. Get and Put Instructions

When a goal is matched against the head of a clause, get instructions load data from the goal's variables (A's) into the clause's variables (Y's).

Before calling a goal in the body of a clause, put instruction load data from the clause's variables (Y's) into goal arguments (A's).

AS a consequence, both kinds of instructions utilize the Var class:

```
public class Var
{
  public VarType type;     //   PERM, ARG, QUERY
  public enum ValType {REF, CON, LIST, STRUC}
  public ValType tag;
  public int index;       // number for the var e.g. V0, V1, etc
  public String name = null;
                          // name of var when it's a QUERY variable
  public String con;      // content when this var is a CON
  public Var ref;         // content when it is a REF
  public Var head, tail;
              // used when the var points to a list or structure
  public ChoicePoint cutLevel;
             // used by the cut and get_level WAM instructions

             :  // more fields and methods
}
```

VarType is an enumeration for the three kinds of variables used in this WAM: permanent (i.e. local) variables inside a clause (PERM), goal arguments (ARG), and variables in a user's query (QUERY).

The ValType enumeration ({REF, CON, LIST, STRUC}) corresponds to the data tags used in the original WAM (e.g. see Figure 12).

If a Var object points to another variable rather than containing its own data (i.e., its `tag` field value is REF), then the reference is stored in the `ref` field. For the special case when the variable is unbound, `ref` points to the object itself (i.e. `ref==this`).

If the variable has been assigned a constant, then the `tag` field is set to CON, and the `con` field holds that constant as a string. In fact, a constant can be an atom, integer, or string, but this's determined at runtime by parsing the contents of the field.

A variable referring to a list or structure is implemented as a binary tree of Var objects, with the help of the `head` and `tail` fields. For a structure (the `tag` field is STRUC), then the `head` field will point to a Var object holding the functor name, and `tail` will point to an object linked to a list of the arguments.

Every variable name is a letter and integer (e.g. Y0, A5, Q10), and the letter is mapped to a VarType ('Y' for PERM, 'A' for ARG, and 'Q' for QUERY) in its `type` field, and the integer is stored in the `index` field. The index is also used to position the Var object in its ArrayList.

Only QUERY variables use the Var's `name` field, to hold the user's name for that variable. For example in `?-append(X,Y,[1,2,3])`, the `name` field of the Var object for Q0 will be assigned the name "X", and Q1 will get "Y".

All of the put and get instructions work in a similar way; `put_value` and `get_value` are typical:

```
private void put_value(String sv, String sAv)
// put_value v Av. Copy the contents of v into Av
{ Var v = varLookup(sv);
  Var Av = varLookup(sAv);
  Av.copyValue(v);
  pc++;
}


private void get_value(String sv, String sAv)
// get_value v Av. Does v = Av?
{  unify_variables(sv, sAv);  }
```

varLookup() maps a variable name (e.g. Y0) to a Var object, and then uses a suitable method from the Var class to update its fields. Some of the instruction, such as `get_value`, utilize the unification functions, which are described next.

4.5. Unification

Unification in the original WAM is implemented somewhat differently from the approach used in Buettcher's WAM. It works implicitly on the current structure being examined, and also employs a read/write mode. The read mode is used when a term is being pattern matched against, while write mode is utilized when the term is being constructed. However, Buettcher's WAM uses explicit arguments in its unification instructions, and doesn't utilize modes. The differences can be seen by considering the 'standard' WAM code generated for the second clause of append/3

```
append([H|T1],  L2, [H|T2]) :- append (T1, L2, T2).
```

The code would look something like:

```
append/3: trust_me
      allocate
      get_list A1                 % [
      unify_variable Y4           % H |
      unify_variable A1           % T1],  L2,
      get_list A3                 % [
      unify_value  Y4             % H |
      unify_variable A3           % T2]
      call append/3               % :- append(T1,L2,T2)
      deallocate
```

By comparison, the BuWAM outputs:

```
append_3_2::: trust_me                  % append(A0, A1, A2) :- ...
              allocate
              get_variable Y0 A0        % Y0 :- A0
              unify_list Y3 Y2 Y1       % Y3 = [Y2 | Y1 ]
              unify_variable Y0 Y3      % Y0 = Y3
              get_variable Y4 A1        % Y4 :- A1
              get_variable Y5 A2        % Y5 :- A2
              unify_list Y7 Y2 Y6       % Y7 = [Y2 | Y6]
              unify_variable Y5 Y7      % Y5 = Y7
              put_value Y1 A0           % A0 :- Y1
              put_value Y4 A1           % A1 :- Y4
              put_value Y6 A2           % A2 :- Y6
              call append_3             % ... :- append (A0, A1, A2)
              deallocate
              proceed
```

In the standard WAM, `get_list` loads the first input argument from the heap, and sets the unification mode to read (pattern matching). The subsequent calls to `unify_variable` only include one argument , and implicitly access first the head and then the tail. A similar approach is used to pattern match on the third input argument (A3). In addition, the WAM is clever enough to set up A1 and A3 so they are ready to be passed to the call to `append/3` at the end of the clause without the need for explicit calls to `put_value`.

The BuWAM code is longer, but perhaps easier to understand since each line shows all the variables being used, and the lack of optimizations makes those lines simpler.

As Table 2 indicates, there are three unification operations in the BuWAM: `unify_variable`, `unify_list`, and `unify_struc`. Since they all work in a similar way, I'll only consider `unify_variable` which calls the unify_variables() support function:

```
private void unify_variables(String s1, String s2)
// v1 = v2
{ Var v1 = varLookup(s1);
  Var v2 = varLookup(s2);
  if (unify(v1, v2))
    pc++;
  else
    backtrack();
} // end of unify_variables()
```

The variable names are mapped to their Var objects by varLookup() and then unify() is called. Its success means that execution can continue, which is handled by incrementing the program counter; failure triggers a call to backtrack().

unify() compares the two Var objects:

```
private boolean unify(Var v1, Var v2)
```

```
{
  if ((v1 == null) || (v2 == null))
    return false;

  v1 = v1.deref();
  v2 = v2.deref();
  if (v1 == v2)
    return true;

  if (v1.tag == Var.ValType.REF) {
    trail.add(v1);
    v1.copyValue(v2);
    return true;
  }

  if (v2.tag == Var.ValType.REF) {
    trail.add(v2);
    v2.copyValue(v1);
    return true;
  }

  if ((v1.tag == Var.ValType.CON) && (v2.tag == Var.ValType.CON))
    return (v1.con.equals(v2.con));

  if (((v1.tag == Var.ValType.LIST) && (v2.tag == Var.ValType.LIST)) ||
     ((v1.tag == Var.ValType.STRUC) && (v2.tag == Var.ValType.STRUC)))
    if ((unify(v1.head, v2.head)) &&
       (unify(v1.tail, v2.tail)))
      return true;

  return false;
} // end of unify()
```

A variable may contain data, point to itself (i.e. be unbound), or point to another variable which holds the data. This latter case is handled by calling Var.deref() which follows links until data is found, or a self-reference is detected:

```
// inside the Var class
public Var deref()
{
  if ((tag == ValType.REF) && (ref != this)) {
    Var v = ref;
    while ((v.tag == ValType.REF) && (v.ref != v))
      v = v.ref;
    return v;
  }
  else
    return this;
} // end of deref()
```

Back in unify(), the Var objects' `tag` fields (which will be CON, LIST, or STRUC) are used for matching. The processing of a LIST or STRUC is handled by recursive calls to unify() to deal with the `head` and `tail` fields in the Var objects.

The Trail

If one of the variables is assigned the value of the other in unify(), that variable is added to the WAM's trail:

```
// in unify()
  :
  if (v1.tag == Var.ValType.REF) {
    trail.add(v1);
    v1.copyValue(v2);
    return true;
  }
 :
```

The trail (see Figures 11 and 14) stores a reference to the bound variable in an ArrayList, so that it's easy to undo that binding during backtracking.

Rolling back a binding is straight forward in logic languages since they're *single assignment* – a variable can only be bound once. This means that undoing that binding only requires resetting its `tag` field to REF and making the object point to itself. This is done by rollback() in the Var class:

```
public void rollbackTo(int pos)
{
  for (int i = contents.size()-1; i >= pos; i--) {
    Var v = contents.get(i);
    if (v != null) {
      v.tag = Var.ValType.REF;
      v.ref = v;    // an unbound var points to itself
      v.refersToArg = -1;
    }
  }
}  // end of rollbackTo()
```

rollback() is passed an index into the trail's list, and all the variables between that position and the end of the list are reset. The `v.refersToArg` field is only used by the diagnostics code.

4.6. Calling Predicates

Table 2 lists five instructions related to calling: `call`, `proceed`, `allocate`, `deallocate`, and `halt`. I'll look at the first four here.

The relevant branch in runWAM() that process the `call` instruction is:

```
// in runWAM() in BuWAM.java
  :
    else if (op == OpWam.CALL) {
      nodesCount++;
      call(s.getArg(0), s.getJump());
    }
  :
```

The nodesCount variable is used by the GraphViz drawing code, so isn't relevant to this explanation. The `s` variable points to the current WAMStatement object, which for a `call` statement stores the label of the predicate being called and the line number where the call should jump to.

The call() function, ignoring its diagnostics and graph printing lines, is:

```
private void call(String callLabel, int pos)
{
  if (pos >= 0) {
    cp = pc+1;  // return to next line after current pc location
    if (choices.empty())
      cutPoint = null;
    else
      cutPoint = choices.peek();   // used by a possible future cut
    pc = pos;
  }
  else  // pos < 0 indicates either a built-in predicate or an error
    if (Builtin.contains(pos))
      callBuiltins(pos);
    else {
      // System.out.println("Error: Could not identify call label");
      backtrack();
    }
} // end of call()
```

The crucial line is:

```
pc = pos;
```

which sets the program counter to the line number where the predicate begin. However, if pos is negative, then it may indicate that the call is to a built-in predicate, and callBuiltins() is executed. If the value doesn't match a built-in's opcode then `call` fails, which triggers backtracking.

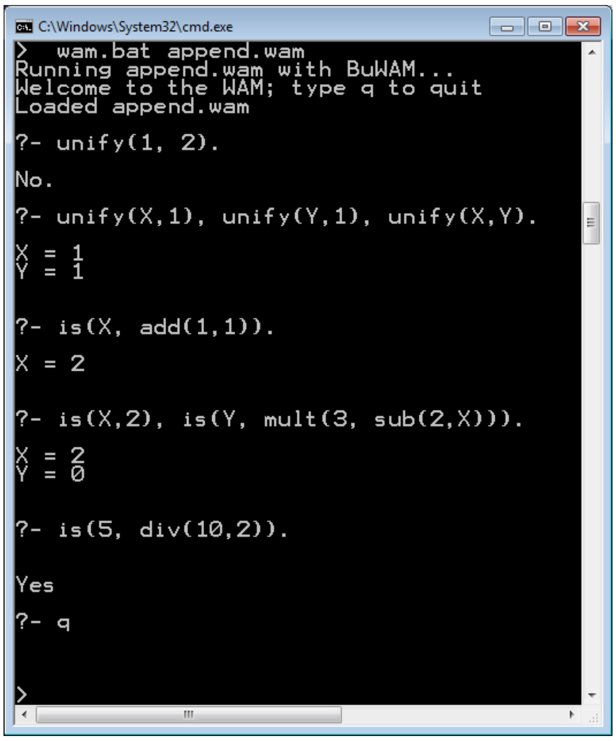
One other feature of the `call` instruction is the setting up of the mechanism used to support cuts, which I'll explain later.

4.6.1. Processing Built-ins

The built-ins supported by the BuWAM are defined in the Builtin enum. Each enumeration has a string-based name and a negative opcode, such as:

```
// in Builtin.java
     :
  VAR("var", -14),
  NONVAR("nonvar", -15),
  FAIL("fail", -16),
  IS("is", -20),
     :
```

Back in the BuWAM class, callBuiltins() consists of a large set of if-branches that test the opcode. I'll look briefly at how the unify/2 and is/2 built-ins are processed. These can be tested inside the BuWAM by typing queries in the REPL, as in Figure 15.



```
>  wam.bat append.wam
Running append.wam with BuWAM...
Welcome to the WAM; type q to quit
Loaded append.wam

?- unify(1, 2).

No.

?- unify(X,1), unify(Y,1), unify(X,Y).

X = 1
Y = 1


?- is(X, add(1,1)).

X = 2


?- is(X,2), is(Y, mult(3, sub(2,X))).

X = 2
Y = 0


?- is(5, div(10,2)).


Yes

?- q



>
```

Figure 15. Using unify/2 and is/3.

The relevant branches for unify/2 and is/2 inside callBuiltins() are:

```
// in callBuiltins() in BuWAM.java
   :
  else if (b == Builtin.IS) {     // is(Result, v).
    is( args.get(0), args.get(1));
  }
   :
  else if (b == Builtin.UNIFY) {
    // unify(v1,v2)
    Var v1 = args.get(0);
    Var v2 = args.get(1);
    if (unify(v1,v2))
      pc++;
    else
      backtrack();
  }
    :
```

unify() is passed two input variables by calling args.get(); args is a global ArrayList of Var objects for the goal arguments. If unify() succeeds then the program counter is incremented, while failure triggers backtracking. These responses to success and failure are used by almost all of the built-ins.

is/2 is handled by is(), which is passed two arguments – the first is the output variable, and the second a term representing the arithmetic expression. The code for is():

```
private void is(Var vRes, Var v)
// vRes is assigned the result of evaluating v
{
  v = v.deref();
  vRes = vRes.deref();

  int result = evalCalc(v);
  if (vRes.tag == Var.ValType.REF) {   // bind vRes to the result
    trail.add(vRes);
    vRes.tag = Var.ValType.CON;
    vRes.con = "" + result;
    pc++;
  }
  else if (vRes.tag == Var.ValType.CON) {
    int val = parseInt(vRes);
    if (val == result)
      pc++;
    else
      backtrack();
  }
  else {
    System.out.println("Error: is/2: result must be var or const");
    backtrack();
  }
}  // end of is()
```

Both of the variables are dereferenced to ensure that if they point to data then its directly accessible. Most of the built-ins start in this way.

eventCalc() evaluates the term argument using a conventional recursive evaluation of the Var object (which may be a constant (with a CON tag) or a structure (with a STRUC tag)). The result is an integer, which is assigned to the vRes variable, and noted on the trail. However, the vRes variable may be bound (as in the last example of Figure 15), which will mean that vRes is tagged as a constant. This case is dealt with by testing if the vRes value is the same as the calculated one.

4.6.2. Proceed

The `proceed` instruction is the WAM equivalent of returning from a function in a conventional language, and is implemented in a similar way. The code for processing `proceed` is 15 lines long, but 14 of those are for generating diagnostics and graphs. If they're ignored, the 'proceed' branch in runWAM() is only:

```
// in runWAM() in BuWAM.java
    :
    else if (op == OpWam.PROCEED)    // return from a clause
      pc = cp;
    :
```

pc is the program counter and cp the continuation pointer.

4.6.3. Allocating and Deallocating Environments

The WAM code for a clause is bracketed by a call to `allocate` at the start and a call to `deallocate` near its end. For example, consider the first clause of append/3:

```
    append_3::: try_me_else append_3_2
                allocate
                get_variable Y0 A0
                put_constant [] Y1
                unify_variable Y0 Y1
                get_variable Y2 A1
                get_value Y2 A2
                deallocate
                proceed
```

`allocate` create an environment for the clause's variables, while `deallocate` restore the previous environment. In this respect, an environment is equivalent to an activation record (or frame) used by virtual machines for imperative languages. There's a major quirk however – `deallocate` (despite what it's name may suggest) doesn't delete the current environment when the previous one is reinstated.

Aït-Kaci [2] explains the issue using the following snippet of Prolog:

```
a :- b(X), c(X).
b(X) :- e(X).
c(1).
e(X) :- f(X).
```

```
e(X) :- g(X).
f(2).
g(1).
```

When the query `?-a` is executed, the stack of environments shown in Figure 16 will be created by the time that f/1 is called. In addition, the choice point stack will have gained a record for e/1 since there are two "e" clauses that can be tried. Amongst the data stored in the choice point is a pointer (called CE usually) that links back to the environment that created the "e" environment.

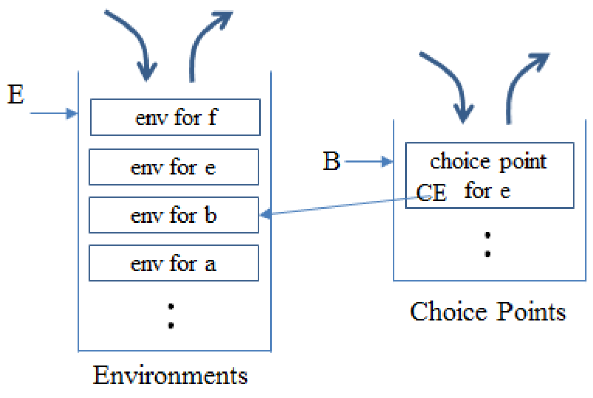


Figure 16. Executing f/1 in the query `?-a`.

As the query continues, X will be bound to 2 inside f/1, and then execution will start returning, causing the "f", "e", and "b" environments to be deallocated until execution is back in a/1, and c/1 is about to be called with X == 2. This situation is shown in Figure 17.

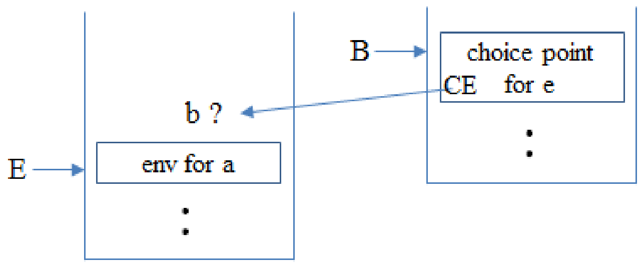


Figure 17. About to execute c/1 in the a/0 clause.

If environments were really deleted when they were deallocated then there's a problem due to the "b" environment's deletion. When the c/1 goal is executed, it fails because X == 2, and the WAM will want to backtrack to the "e" choice point. This requires the resumption of the "b" environment (via the CE pointer) so that b/1 can call the second clause of e/1. Unfortunately, the "b" environment no longer exists!

The BuWAM's solution is to have the `deallocate` instruction move the E pointer down the environments stack as it 'deallocates' the "f"," e", and "b" environments but **not** to delete those records. This situation is shown in Figure 18.

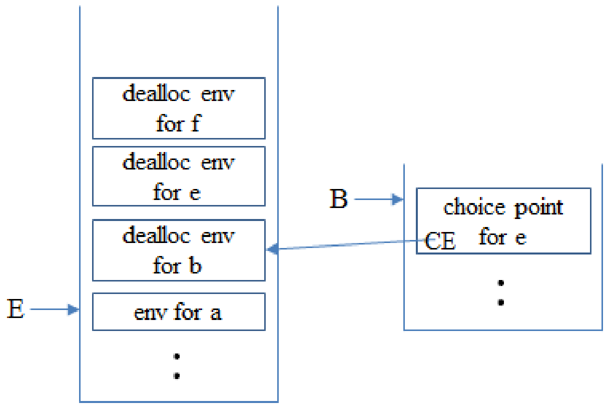


Figure 18. About to execute c/1 in the a/0 clause.

One objection to this approach is the needless retention of the "f" and "e" environments. However, they'll only hang around until the "b" environment is reactivated. Part of its code (which will be shown later) tidies up the environment list, deleting those deallocated records that are above the environment record used by the top-most choice point (i.e. the "e" choice point in Figure 18). This clean-up process results in Figure 19, after the removal of the "f" and "e" records.

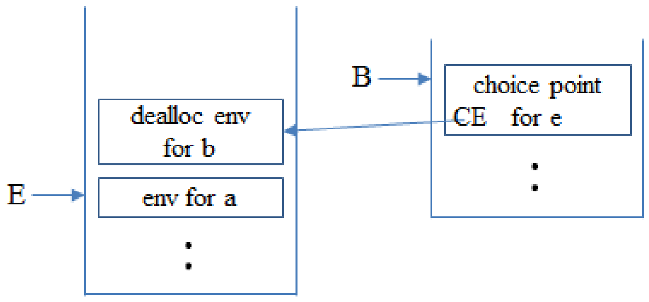


Figure 19. Reactivating the "b" environment.

This approach differs somewhat from the one outlined by Aït-Kaci [2], which combines the environment and choice point stacks into a single data structure. In that situation, there's no need for an explicit CE pointer from the "e" choice point to the "b" environment since the "e" choice point rests directly on top of that environment. Deallocation becomes a matter of deleting an environment record only if there isn't a choice point somewhere above it on the stack.

The `allocate` instruction is processed by allocate() in the BuWAM class:

```
// globals
private ArrayList<Env> envList = new ArrayList<>();
private int envIdx;
private Stack<ChoicePoint> choices = new Stack<>();


private void allocate()
{
  String clauseLabel = getLineLabel(pc-1);
     // the clause's label is on the line before "allocate"

  Env env = new Env(clauseLabel, cp, envIdx, tryNodeNm, nodesCount);

  // really delete old env. records
  if ((envIdx > -1) && (!choices.isEmpty())) {
    int ce = choices.peek().envIdx;
    if (envIdx > ce)   // delete unneeded environments
      envList.subList(envIdx+1, envList.size()).clear();
  }
  envList.add(env);
  envIdx = envList.size()-1;   // set to top of list

  pc++;
}   // end of allocate()
```

A new Environment object is added to the global envList. Rather than employ an "E" pointer as shown in the previous figures, the top of the list is recorded as an integer index, stored in the envIdx global.

The if-test in the middle of allocate() is where real environment deletion is performed. It looks at the top-level choice point and gets its ce value. If envIdx is larger than ce then a situation like Figure 16 exists, and the extra environment objects can be deleted.

The Environment record created inside allocate() takes five arguments, but three of those are for diagnostics reporting. For the virtual machine's execution, the important arguments are the cp (continuation pointer) return address, and the environment index. Each Environment object also stores the clause's permanent variables, which are added to the object as they're encountered in the WAM instructions that follow the `allocate` line.

The `deallocate` instruction is processed by deallocate(), and is quite long due to all the diagnostics and graphing code. If all that is elided, the function becomes:

```
private void deallocate()
{
  cp = envList.get(envIdx).cp;
                // the continuation pointer (the return address)
  envIdx = envList.get(envIdx).ce;
  pc++;
}  // end of deallocate()
```

The continuation pointer and environment index stored by the `allocate` instruction are retrieved, and used to reset those globals in the WAM. Note that deallocate() does not delete the environment object.

4.7. Branching Instructions

The BuWAM has two branching instructions: `try_me_else` and `trust_me`. The WAM described by Aït-Kaci [2] also has `retry_me_else`, but Buettcher folded its functionality into `try_me_else`. A third operation related to branching is backtracking, as implemented by backtrack().

`try_me_else` and backtrack() are closely related – the former creates a choice point, pushing it onto a global `choices` stack, while backtrack() pops the top-most choice point from that stack to let Prolog backtrack.

The choice point is the most complicated data structure in the standard WAM explained by by Aït-Kaci. Table 3 lists its fields, along with B0 for supporting cuts.

A0, A1, …, An	Copy of Arguments
CE	Pointer to previous Environment
CP	Continuation Pointer (return address) at the start of this choice
B	Pointer to **previous** choice point
TR	Pointer to trail (for rollback)
H	Pointer to heap (for garbage collection; used to set HB)
BP	Label of next clause to try if this choice fails
B0	The cut register

Table 3. The Fields of a Choice Point.

The first column of Table 3 are the standard WAM names for the fields, which denote pointers to different data structures stored in memory. If you look back to Figure 11, you'll see CP, H, B, and TR, while CE appears in Figures 16 and 19. BP points to the label of the next clause to try, and so links into the WAM code area of Figure 11. B0 is the cut register which I'll explain in the "cut" section next.

The BuWAM version of a choice point is an object created by the ChoicePoint class, which is given below:

```
public class ChoicePoint
{
  public ArrayList<Var> args;   // the ARG vars

  public int envIdx;            // index into environment list
                                // for the previous clause's PERM vars;
                                // called CE in the original WAM

  public int cp;                // continuation pointer
                                // (i.e. the return address)

  public int trailIdx;          // current trail index for undoing
                                // variable bindings;
                                // called TR in the original WAM

  public int elseLabel;         // address of next clause to try
                                // during backtracking;
                                // called BP in the original WAM

  public ChoicePoint cutPoint;  // used to backtrack over a cut;
                                // called B0 in the original WAM

  // diagnostics data fields
        :

  public ChoicePoint(ArrayList<Var> vargs)
  {
    args = new ArrayList<>();
    for (int i = 0; i < vargs.size(); i++)
      args.add(new Var(vargs.get(i)));
  } // end of ChoicePoint()

} // end of class ChoicePoint
```

There's a fairly straight forward mapping between a choice point in the standard WAM and the BuWAM. The most major change is the absence of the B and H fields. The previous choice point (B) is instead implemented by peeking at the record below this one in the `choices` stack. Also, the BuWAM doesn't have an explicit heap, since Var objects utilize the Java heap to store data. Another difference is that some of the fields are encoded as indices (i.e. envIdx and trailIdx) rather than as pointers.

try_me_else and Backtracking

The `try_me_else` instruction is processed by try_me_else(), which creates a new ChoicePoint object and pushes it onto the global `choices` stack:

```
private void try_me_else(String callLabel, int elseLabelIdx)
{
  // build a choice point
  ChoicePoint choicePt = new ChoicePoint(args);
  choicePt.label = callLabel;
  choicePt.trailIdx = trail.getLength();
  choicePt.cp = cp;
  choicePt.elseLabel = elseLabelIdx;   // address of next clause
  String elseLabel = getLineLabel(elseLabelIdx);
  choicePt.cutPoint = cutPoint;
  choicePt.envIdx = envIdx;

  choicePt.callNodeNm = callNodeNm;
  choicePt.tryCount = nodesCount;

  choices.push(choicePt);   // store the choice point
  pc++;  // start the clause's code
}  // end of try_me_else()
```

The popping of a ChoicePoint object from the stack, and the restoration of its data is dealt with by backtrack() :

```
private void backtrack()
{
  queryFailed = true;
  if (!choices.isEmpty()) {
    // restore earlier state before trying next clause...
    int choicePos = choices.size()-1;
    ChoicePoint choicePt = choices.pop();

    args = choicePt.args;      // reset the clause head's arguments

    int tIdx = choicePt.trailIdx;   // rollback the trail of bindings
    trail.rollbackTo(tIdx);

    cp = choicePt.cp;          // reset the return address
    pc = choicePt.elseLabel;   // set addr of the next clause to try
    callNodeNm = choicePt.callNodeNm;
    cutPoint = choicePt.cutPoint;
    envIdx = choicePt.envIdx;
  }
  else {   // should not happen
    trail.rollbackTo(0);
    pc = -1;
  }
} // end of backtrack()
```

This is not exactly the approach used in Aït-Kaci's WAM which utilizes a simpler backtrack() function which only adjusts the program counter and the cut register. The rest of the resetting is carried out by the WAM's branching instructions, such as `retry_me_else` and `trust_me`. The relocation of this behavior to backtrack() in the BuWAM means that it's possible to remove `retry_me_else` from its instruction set, and utilize `try_me_else` instead. Also, the implementation for `trust_me` is greatly shortened. Ignoring the diagnostics code, `trust_me` is handled by a one-liner in runWAM():

```
  :
else if (op == OpWam.TRUST_ME)
  pc++;
  :
```

4.8. Dealing With Cuts

Cuts weren't part of Warren's original WAM design [7], and are only introduced near the end of Aït-Kaci text [2], but the changes are quite minimal – two new instructions, `get_level` and `cut`, and a global variable called the cut register (usually called B0, but renamed `cutPoint` in the BuWAM). The Var class must also be extended with a new field.

Before each goal is called, the `cutPoint` global is linked to the current choice point. In the BuWAM, this is implemented inside call() which handles the `call` instruction.

```
// in BuWAM.java
// global
private ChoicePoint cutPoint = null;

// in call()
cutPoint = choices.peek();
```

Also, when a ChoicePoint object is created for a clause in try_me_else(), `cutPoint` is stored inside that object's `cutPoint` field. Subsequently, if backtrack() restores that choice point, its `cutPoint` field is used to reset the global `cutPoint`:

```
// in try_me_else()
choicePt.cutPoint = cutPoint;

// in backtrack()
cutPoint = choicePt.cutPoint;
```

All this storing and restoring means that `cutPoint` always points to the choice point existing when the clause containing the cut was first called.

`cutPoint` is actually used by the WAM `get_level` and `cut` instructions, which are best explained by looking at how Prolog code using a "!" is converted into WAM instructions.

The following list merging predicate employs several cuts:

```
merge([X|Xs],[Y|Ys],[X|Zs]) :-
  lt(X,Y), !,
  merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :-
  eq(X,Y), !,
  merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :-
  gt(X,Y), !,
  merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) :- !.
merge([],Xs,Xs) :- !.
```

The first clause is translated into the following WAM code:

```
merge_3::: try_me_else merge_3_2
           allocate
           get_level Y11
           get_variable Y0 A0
           unify_list Y3 Y2 Y1
           unify_variable Y0 Y3
           get_variable Y4 A1
           unify_list Y7 Y6 Y5
           unify_variable Y4 Y7
           get_variable Y8 A2
           unify_list Y10 Y2 Y9
           unify_variable Y8 Y10
           put_value Y2 A0
           put_value Y6 A1
           call lt
           cut Y11
           put_value Y1 A0
           unify_list Y12 Y6 Y5
           put_value Y12 A1
           put_value Y9 A2
           call merge_3
           deallocate
           proceed
```

The compilation of the "!" causes the addition of a call to `get_level` at the start of the clause, and a call to `cut` at the place where the "!" occurred.

`get_level` stores `cutPoint` in a permanent variable (Y11 in the code above). This is dealt with by get_level() in BuWAM.java:

```
private void get_level(String sv)
{
  Var v = varLookup(sv);
  v.cutLevel = cutPoint;
  pc++;
}
```

The subsequent `cut` instruction is handled by cut(), which is passed this variable (i.e. Y11):

```
// globals
private Stack<ChoicePoint> choices = new Stack<>();


private void cut(String sv)
{
  Var v = varLookup(sv);
  ChoicePoint cpt = v.cutLevel;
  // discard all the choicepoints from the top of
  // the stack down to cpt.
  if (cpt != null) {
    ChoicePoint currCpt = choices.peek();
    while (!cpt.equals(currCpt)) {
      choices.pop();
      currCpt = choices.peek();
    }
  }
  else
     choices.clear();
  pc++;
}  // end of cut()
```

The `cutPoint` value stored in the variable is retrieved, and used to rollback the global choice points stack, discarding all the choices made since a goal matched with this clause up to the "!' goal within the clause.

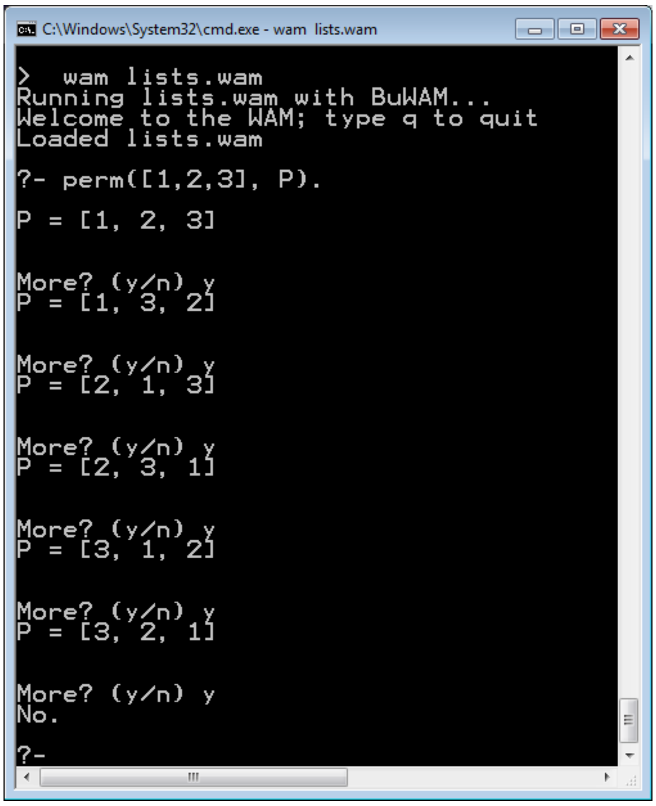
5. Using the Diagnostics output

This section examines how the diagnostics output from BuWAM can be used alongside the graph diagrams to better understand the workings of the virtual machine.

The following two predicates implement list permutation in list.pro:

```
perm([], []).
perm(L, [X|P]) :-
  del(X, L, L1),
  perm(L1, P).

del(A, [A|List], List).
del(A, [B|List], [B|Listl]) :-
  del(A, List, Listl).
```

The query shown in Figure 19 generates all the permutations of the list [1,2,3].



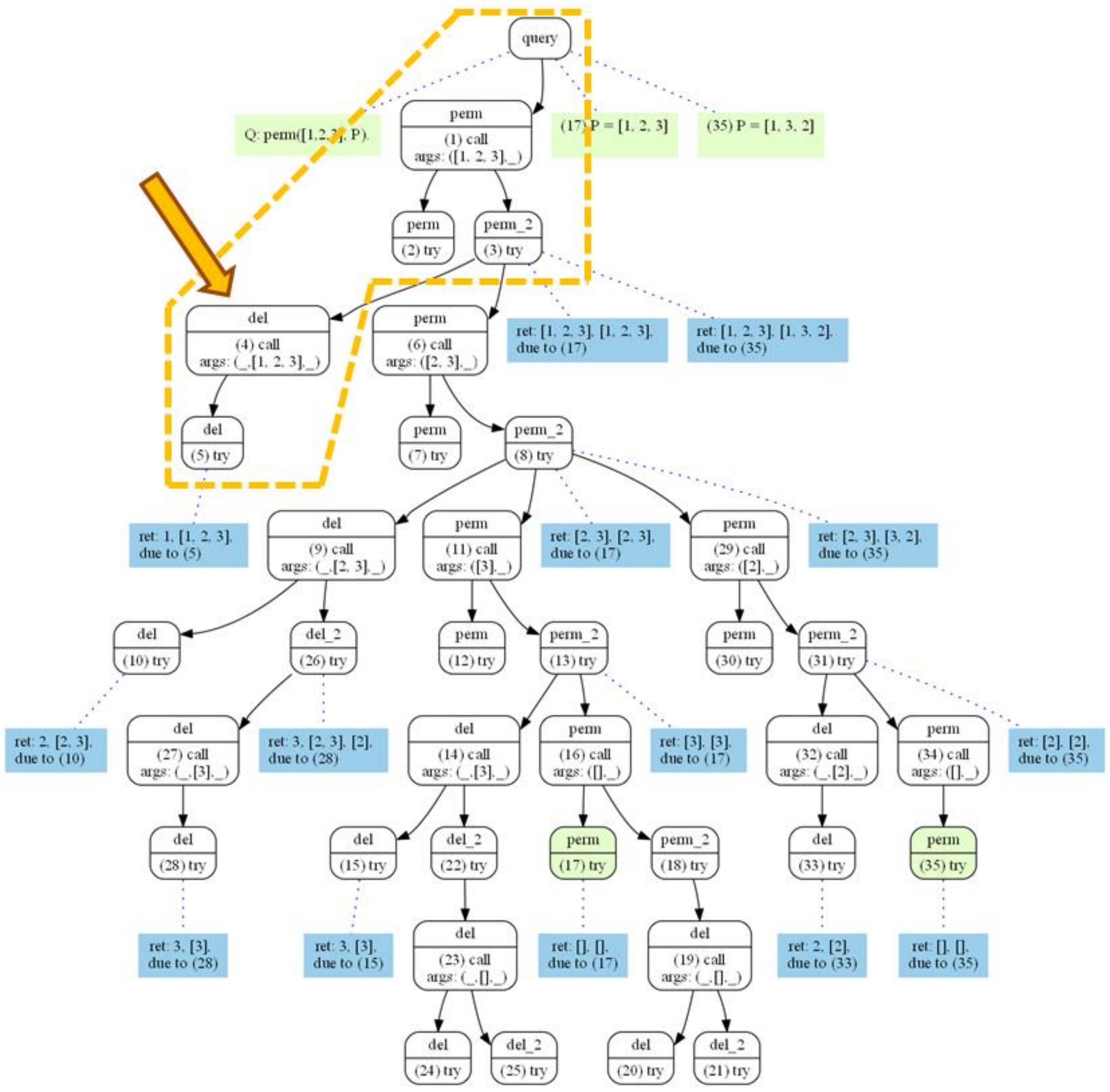
```
>  wam lists.wam
Running lists.wam with BuWAM...
Welcome to the WAM; type q to quit
Loaded lists.wam

?- perm([1,2,3], P).

P = [1, 2, 3]


More? (y/n) y
P = [1, 3, 2]


More? (y/n) y
P = [2, 1, 3]


More? (y/n) y
P = [2, 3, 1]


More? (y/n) y
P = [3, 1, 2]


More? (y/n) y
P = [3, 2, 1]


More? (y/n) y
No.

?-
```

Figure 19. Permutations of [1,2,3].

If BuWAM is instead invoked with the –d and –p flags, then diagnostics and a graph of the execution tree will be created:

```
> java -jar BuWAM.jar -dp lists.wam 2> debug.txt
```

Figure 20 shows the execution tree when `?-perm([1,2,3],P)` is asked to produce two answers.



query
Q: perm([1,2,3], P).
(17) P = [1, 2, 3]
(35) P = [1, 3, 2]
perm (1) call args: ([1, 2, 3],_)
perm (2) try
perm_2 (3) try
del (4) call args: (_,[1, 2, 3],_)
perm (6) call args: ([2, 3],_)
ret: [1, 2, 3], [1, 2, 3], due to (17)
ret: [1, 2, 3], [1, 3, 2], due to (35)
del (5) try
perm (7) try
perm_2 (8) try
ret: 1, [1, 2, 3], due to (5)
del (9) call args: (_,[2, 3],_)
perm (11) call args: ([3],_)
ret: [2, 3], [2, 3], due to (17)
perm (29) call args: ([2],_)
ret: [2, 3], [3, 2], due to (35)
del (10) try
del_2 (26) try
perm (12) try
perm_2 (13) try
perm (30) try
perm_2 (31) try
ret: 2, [2, 3], due to (10)
del (27) call args: (_,[3],_)
ret: 3, [2, 3], [2], due to (28)
del (14) call args: (_,[3],_)
perm (16) call args: ([],_)
ret: [3], [3], due to (17)
del (32) call args: (_,[2],_)
perm (34) call args: ([],_)
ret: [2], [2], due to (35)
del (28) try
del (15) try
del_2 (22) try
perm (17) try
perm_2 (18) try
del (33) try
perm (35) try
ret: 3, [3], due to (28)
ret: 3, [3], due to (15)
del (23) call args: (_,[],_)
ret: [], [], due to (17)
del (19) call args: (_,[],_)
ret: 2, [2], due to (33)
ret: [], [], due to (35)
del (24) try
del_2 (25) try
del (20) try
del_2 (21) try

Figure 20. An Execution Tree for ?- perm([1,2,3], P).

For this example, we'll look more closely at node (4) in Figure 19, a call to del/3.

Using the node numbers in the graph, it's easy to locate the related diagnostics output in debug.txt:

```
    :
  --> Calling del_3 (4)
  --> vars: [ A0 = _; A1 = [1, 2, 3]; A2 = _; ]
Run: (0139)        del_3::: try_me_else del_3_2  % in del_3
  --> Trying del_3 (5); choice recorded
  --> Choices:
      * C0: del_3 (5) in E1
Run: (0140)                  allocate  % in del_3
  --> Envs (current env: E1):
        E0: perm_2 (2); back to query
      * E1: perm_2_2 (3); back to query
Run: (0141)                  get_variable Y0 A0  % in del_3
Run: (0142)                  get_variable Y1 A1  % in del_3
Run: (0143)                  unify_list Y3 Y0 Y2  % in del_3
Run: (0144)                  unify_variable Y1 Y3  % in del_3
Run: (0145)                  get_value Y2 A2  % in del_3
Run: (0146)                  deallocate  % in del_3
  --> ret: 1, [1, 2, 3],
      due to (5)
  --> Envs (current env: E2):
        E0: perm_2 (2); back to query
        E1: perm_2_2 (3); back to query
      * E2: del_3 (5); back to E1
      --> leaving E2 for E1
Run: (0147)                  proceed  % in del_3
  --> Return to perm_2_2
    :
```

The del/3 call is followed by the arguments passed to the clause (A0, A1, and A2). The first clause of del/3 is tried, becoming node (5), and a choice point is created. The entire choices stack is printed, indicating that there's only a single choice point in play at this time.

When the first clause of del/3 is executed, the WAM code is printed as numbered "Run" lines: the clause begins with an `allocate` instruction on line 140, and continues until the clause returns due to the `proceed` on line 147.

When a clause is entered, the current environments list is printed. In this case, it contains environments for the perm/2 clauses run in nodes (2) and (3).

When the `deallocate` line is reached, any bindings and the final status of the environments list are printed. The list now includes an E2 environment representing this del/3 clause. The execution then returns to the second clause of perm/2.

The main drawback of the diagnostics output is its extreme size, but it's possible to use the node numbers in the graph to focus on points of interest. For instance, the two green nodes in the execution tree which signal where top-level results were first created, are numbered (17) and (35). These nodes can be readily located in the diagnostic output to obtain more details.

Perhaps more useful for debugging, nodes which result in failure are colored red in the execution tree, and are also numbered.

6. Possible Extensions

BuProlog was developed for use during an "Introduction to Prolog" lab, and also as a source of programming projects for a course on compiler design. This section describes some extensions to the compiler and virtual machine that could be the basis of project ideas.

It's sometimes a little irritating that the lexical pattern for an atom only uses upper and lowercase letters and digits (i.e. `Atom = [a-z]([:letter:]|[:digit:])*`). Most Prologs allow at least the use of underscores ('_') and a few other non-letter symbols. Unfortunately, adding '_' to the `Atom` regular expression in Lexer.jflex (see Figure 4) will break some of the WAM code generation routines, which use '_' to delimit arity information in the predicate names. In addition, GraphViz is quite temperamental about using non-letter symbols in the names of its nodes.

The restriction of numerical data to integers could be relaxed by adding support for floating point numbers. This would require changes to the lexical analyzer, but the rest of the compiler manipulates its data as strings so would remain mostly unchanged. However, the BuWAM would need extra support for treating its data as floats. For example, the numerical operations offered by is/2 would need to be extended.

The need to write numerical tests, such as '<', '==", and so on as functors (i.e. as lt/2 and eq/2) is burdensome, and adding grammar rules to support infix versions of these tests would undoubtedly be cheerfully greeted. Similarly, it would be pleasing if the expression term in is/2 could be written using infix operations such as '+', '-' and so on. Such changes should probably be restricted to the parser, which would still convert these elements into their current parse tree forms. Also, I'd avoid trying to support operator precedence; brackets for grouping operations is a lot easier to implement.

Another parsing change might be to add support for if-then-else and or, implemented by compilation down to existing parse tree structures that employ '!'.

There's plenty of leeway for adding more built-ins. One approach is to have the BuWAM load a standard library of compiled predicates (e.g. the WAM code for append/3, not/1, member/2, and so on). Another would be add built-ins in the same way as predicates like is/2.

Another useful predicate would be getch/1 for reading in a character, although this could be supported by adding string manipulation predicates that manipulate the input from readln/1. In general, IO should be extended to support streams and/or filenames.

Other helpful built-ins would be similar to SWI-Prolog's recorded database predicates (https://www.swi-prolog.org/pldoc/man?section=recdb) for storing and retrieving terms. This capability would allow the Prolog-level implementation of all-solution predicates such as findall/3 and bagof/3, which in turn would allow more complex types of search predicates to be encoded, such as breadth-first search [3]. Recorded database functionality would be much easier to add than support for assert and retract.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, 1986, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley

[2] Hassan Aït-Kaci, 1991, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press. Out of print. Available at https://github.com/a-yiorgos/wambook

[3] Ivan Bratko,1986, *Prolog Programming for Artificial Intelligence*, Addison-Wesley.

[4] William Clocksin and Christopher S. Mellish, 1984, *Programming in Prolog*, 2nd ed. Springer-Verlag.

[5] Peter Van Roy, 1994, "1983-1993:The Wonder Years of Sequential Prolog Implementation", *The Journal of Logic Programming*, Vol. 19, May, pp. 385–441.

[6] Leon Sterling and Ehud Shapiro, 1986, *The Art of Prolog*, MIT Press.

[7] David H. D. Warren, 1983, "An Abstract Prolog Instruction Set", Technical Note 309, SRI International, August.

Appendix A

The following table lists all of the example Prolog files included with the system, and a short description of each one.

Filename	Description
addrs.pro	Home addresses; a small test of terms
append.pro	append/3
blocks.pro	The Blocks World problem
boxes.pro	The boxes problem
calls.pro	Tests of call/1 and univ/3
colorMap.pro	Two versions of map coloring
crypto.pro	Solving cryptoarithmetic puzzles
cut.pro	Tests of cut
diff.pro	Small test of gt/2 and is/2
errors.pro	Predicates with (deliberate) errors.
family.pro	A family database
flights.pro	A flight route planner
hanoi.pro	Towers of Hanoi
jugs.pro	The Water Jugs problem
lanford.pro	The Lanford sequence
lists.pro	Several list predicates: reverse, permutation, length, etc.
math.pro	Several math predicates: primes, gcd, lcm, ackerman, fibonacci
maze.pro	Find a path through a maze
mc.pro	The missionaries and cannibals problem
movies.pro	A movies database
nqueens.pro	The N queens problem
parents.pro	A parents database; used in an Introduction to Prolog lab
roman.pro	Convert years into roman numerals
solve.pro	A simple Prolog meta-interpreter
sorts.pro	Quicksort and insertion sort
sudoku2.pro	A very slow Sudoku solver; illustrates the need for constraints.
tictactoe.pro	Tic-Tac-Toe
zebra.pro	The "Finding the zebra" problem.

Table A. Prolog Examples.

Appendix B

The following table lists all of the built-in predicates in BuProlog, along with a list of the example files where they are used.

Built-in Predicate	Example File(s)
writeln/1, write/1	colorMap.pro, cut.pro, mc.pro, roman.pro, sudoku2.pro, tictactoe.pro
atom/1	solve.pro
integer/1	solve.pro, sudoku2.pro, tictactoe.pro
var/1, nonvar/1	crypto.pro, sudoku2.pro
fail	calls.pro, math.pro, parents.pro, tictactoe.pro
is/2: add, sub/minus, mult/mul, div, mod	calls.pro, crypto.pro, diff.pro, errors.pro, flights.pro, hanoi.pro, lists.pro, math.pro, mc.pro, nqueens.pro, parents.pro, roman.pro, solve.pro
read/1	--
readln/1	tictactoe.pro
Tests: eq/2 (==), neq/2 (!=),	crypto.pro, cut.pro, solve.pro
Tests: lt/2 (<), le/2 (<=), gt/2 (>), ge/2 (>=)	boxes.pro, calls.pro, cut.pro, family.pro, flights.pro, hanoi.pro, math.pro, mc.pro, movies.pro, nqueens.pro, parents.pro, roman.pro, solve.pro, sorts.pro
! (cut)	calls.pro, crypto.pro, cut.pro, math.pro, parents.pro, solve.pro, sudoku2.pro, tictactoe.pro
unify/2 (=)	solve.pro, sudoku2.pro, tictactoe.pro, zebra.pro
nunify/2 (\=)	maze.pro, mc.pro, parents.pro, solve.pro
call/1	calls.pro, parents.pro, solve.pro, tictactoe.pro
univ/3 (=..)	calls.pro, solve.pro

Table B. Example Files using the Built-ins.