FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Interpreter for a Prolog Subset using Warren Abstract Machine
Student:	Jiří Skotal
Supervisor:	Ing. Jan Liam Verter
Study program:	Informatics
Branch / specialization:	Computer Science 2021
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Get acquainted with the Prolog programming language [1,2], and the Warren Abstract Machine (WAM) [2] for the execution of Prolog.
Implement interpreter and REPL for a subset of Prolog using WAM.
Your subset must support at least Horn clauses and cuts.
Verify the functionality of your implementation by testing with a relevant set of sample codes.

[1] STERLING, Leon; SHAPIRO, Ehud Y. The art of Prolog: advanced programming techniques. MIT press, 1994.
[2] KOGGE, Peter M. The architecture of symbolic computers. McGraw-Hill, Inc., 1990.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 12 February 2024 in Prague.

Bachelor's thesis

INTERPRETER FOR A PROLOG SUBSET USING WARREN ABSTRACT MACHINE

Jiří Skotal

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: Ing. Jan Liam Verter
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Skotal Jiří. *Interpreter for a Prolog Subset using Warren Abstract Machine.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

List of Acronyms ix

Introduction 1

1 Logic programming 2
- 1.0.1 Fact . . . 2
- 1.0.2 Query . . . 2
- 1.0.3 Rule . . . 2
- 1.1 First-order predicate logic . . . 3
- 1.2 Horn clause . . . 3
- 1.3 PROLOG . . . 4
  - 1.3.1 Unification . . . 4
  - 1.3.2 Backtracking . . . 4
  - 1.3.3 Query satisfaction . . . 4

2 Warren Abstract Machine 6
- 2.1 Program structure . . . 6
  - 2.1.1 An Individual Code Section . . . 6
- 2.2 Data structures and state registers . . . 7
  - 2.2.1 Code Area . . . 7
  - 2.2.2 Heap . . . 8
  - 2.2.3 Stack . . . 8
  - 2.2.4 Trail . . . 8
  - 2.2.5 Push Down List . . . 8
  - 2.2.6 Mode flag . . . 9
- 2.3 Memory word format . . . 9
- 2.4 Choice Point . . . 9
  - 2.4.1 Choice point structure . . . 10
- 2.5 Instruction set . . . 11
  - 2.5.1 Indexing instructions . . . 13
  - 2.5.2 Procedural instructions . . . 16
  - 2.5.3 Get instructions . . . 17
  - 2.5.4 Put instructions . . . 20
  - 2.5.5 Unify instructions . . . 21
  - 2.5.6 Cut instruction . . . 24
- 2.6 Prolog-to-WAM compiler . . . 24
  - 2.6.1 Procedure-Level Compilation . . . 24
  - 2.6.2 Clause-Level Compilation . . . 25

3 Implementation strategy **29**
3.1 Lexer . . . 29
3.2 Parser . . . 29
3.2.1 Wildcard variables . . . 30
3.2.2 Abstract Syntax Tree . . . 31
3.3 Bytecode representation . . . 32
3.4 Bytecode generation . . . 32
3.4.1 Query compilation . . . 34
3.4.2 Procedure name encoding . . . 34
3.5 Major data structures . . . 35
3.5.1 Argument registers . . . 35
3.5.2 Heap, Trail, and Push-down list . . . 35
3.5.3 Stack and Choice Point . . . 35
3.6 Interpretation . . . 36
3.6.1 Multiple answers . . . 36
3.7 Unification operator . . . 36
3.8 Is operator . . . 37
3.9 Preprocessor . . . 38
3.9.1 Naming convention . . . 38

4 Implementation **40**
4.1 Variable and reference word . . . 40
4.2 List representation . . . 41
4.3 Nested complex objects pre-calculation . . . 41
4.4 Right-hand side bytecode generation . . . 42
4.5 Infinite terms . . . 42

Conclusion **47**

Contents of the attachment **49**

List of Figures

2.1 Major memory areas of WAM [2, p. 492] 7
2.2 The choice point structure [2, p. 496] 11

List of Tables

2.1 Memory Word Combinations 9
2.2 getv cases 18
2.3 Heap state prior to `call` instruction 28

List of code listings

1.1 DFS pseudocode 4

2.1 Sample source code/bytecode 10
2.2 Example program 12
2.3 Example bytecode 12
2.4 Instructions for the unification operator 19
2.5 Put instructions bytecode 20
2.6 Bytecode for a fact with complex arguments 22
2.7 Bytecode generated for `s(x)` 26
2.8 Bytecode generated for list 26
2.9 Bytecode generated the whole query 27

3.1 LL(1) grammar in EBNF 30
3.2 Code and bytecode example for RHS code generation 33
3.3 Bytecode for q 34
3.4 Bytecode for q with arity encoded 34
3.5 Bytecode for id 36
3.6 Bytecode for is operator 37
3.7 Factorial using **is** operator 38

4.1 `VariableWord` class 40
4.2 `bound()` method implementation 41
4.3 ListNode constructor 44
4.4 Nested complex objects pre-calculation 45
4.5 Bytecode generation for RHS 45
4.6 unifyArguments method implementation 46

I would like to thank my supervisor, Ing. Jan Liam Verter, for his guidance, patience, valuable advice, and availability.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity

In Prague on May 16, 2024

Abstract

This thesis explores an implementation of the Warren Abstract Machine as an interpreter for a subset of Prolog. It delves into the theoretical model of the Warren Abstract Machine and its inner workings, offering detailed examples to serve as a reference. Additionally, the compiler from Prolog to Warren Abstract Machine bytecode and an interactive Read-eval-print loop are implemented to execute Prolog programs.

Keywords logic programming, Prolog, interpreter, abstract machine, C++

Abstrakt

Tato práce se zabývá implementací interpreteru pro podmnožinu jazyka Prolog pomocí Warren Abstract Machine. Podrobně popisuje teoretický model Warren Abstract Machine, její vnitřní fungování a nabízí detailní příklady jako referenci. Dále je naimplementován překladač z Prologu do bytecodu Warren Abstract Machine, společně s Read-eval-print prostředím pro spouštění Prolog programů.

Klíčová slova logické programování, Prolog, interpreter, abstract machine, C++

List of Acronyms

AST	Abstract Syntax Tree
CPU	Central Processing Unit
LHS	Left-hand Side
PDL	Push Down List
PC	Program Counter
RHS	Right-hand Side
REPL	Read-eval-print loop
I/O	Input/Output
WAM	Warren Abstract Machine

Introduction

Although very different from standard procedural programming, the logic programming paradigm has found many uses in various fields, such as database systems, proof assistants, and Artificial Intelligence.

Logic programming is what is known as declarative programming, which, intuitively, means that the programmer declares what is to be done, leaving the language interpreter to focus on determining how to achieve the desired outcome. This allows for a more high-level expression of problem-solving logic, where the programmer describes the problem's constraints and relationships rather than prescribing explicit procedural steps [1]. Because of this abstraction, understanding how these languages are implemented and function may not be immediately intuitive.

The thesis is divided into the following chapters: Chapter 1 introduces logic programming and the fundamentals of first-order predicate logic. Attention is also given to the Prolog programming language and its theoretical concepts.

Chapter 2 describes the Warren Abstract Machine model, starting with a high-level overview of the memory model and bytecode structure, continuing with a detailed analysis of individual instructions and the WAM bytecode compilation process.

Chapter 3 is dedicated to the design of the implementation itself. It describes the structure of individual classes and the implementation choices made, addressing the challenges encountered and their solutions.

Finally, chapter 4 then delves into the implementation, describing interesting algorithms.

Goals of the thesis

The primary goal of this thesis is to serve as a practical reference for students or individuals interested in comprehending and implementing a logic language. It aims to offer a comprehensive understanding of the concepts of implementing a logic programming language, going beyond a mere high-level overview.

It can serve as a reference, providing additional explanation when reading about the WAM in other literature, such as *The Architecture of Symbolic Computers* [2].

Chapter 1
Logic programming

A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of the consequences of the program. A program defines a set of consequences, which is its meaning. The basic constructs of logic programming are facts, rules, and queries. There is a single data structure: the ***logical term****. [3, pp. 9, 11]*

1.0.1 Fact

Fact is the simplest kind of statement that can be made in a logic program. It represents a relation that holds between objects.

An example of a fact is:

```
bigger(elephant,mouse).
```

Which states that the elephant is bigger than a mouse, or, in other words, that the `bigger` relation holds between the `elephant` and the `mouse`.

Another name for a relation is *predicate*. The animals in the example are known as *atoms*. It is a convention that the names of both predicates and atoms begin with a lowercase letter to distinguish them from logical variables. [3, pp. 11, 12]

1.0.2 Query

Queries are a means of retrieving information from the logic program. A query asks whether a certain relation holds between objects. Syntactically, queries and facts look the same; however, context can distinguish them.

A query in the form of

```
?> bigger(mouse,bug)
```

asks whether the `bigger` relation holds between `mouse` and `bug`.

Answering a query with respect to a program is determining whether the query is a logical consequence of the program. [3, p. 12]

1.0.3 Rule

Another important statement in logic programming is a rule. It enables us to define new relations using already existing relationships.

A rule is a statement in the form of:

```
q:- A1,A2,...,An, n ≥ 0
```

where `q` is called the **head**, and the conjunction of goals `A1,...,An` is known as the **body** of the rule. Note that fact is a special case of a rule for `n = 0`.

The rules are means to express new or complex queries in terms of simple queries. For example, say we have a query in the form of `father(james,X),male(X)`. This query asks whether `X` is James's son. We can express this using the following rule: `son(X,Y) :- father(Y,X), male(X)`. [3, pp. 18, 19]

1.1 First-order predicate logic

Let's describe the first-order predicate logic language as outlined in [4]. This language consists of logical connectives: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, variables (e.g., x, y, z), quantifiers $\forall$ and $\exists$, parentheses, constants, predicate symbols, and function symbols.

A sequence of symbols in the language of predicate logic is a **term** if it was created by following rules in finitely many steps:

1. It is a variable or a constant.
2. If $t_1, ..., t_n$ are terms and f is a n-ary function symbol, then $f(t_1, ..., t_n)$ is a term.

A sequence of symbols in the language of predicate logic is a **formula** if it was created by following rules in finitely many steps:

1. p is a n-ary predicate symbol and $t_1, ..., t_n$ are terms, then $p(t_1, ..., t_n)$ is a formula. Such a formula is called the **atomic formula** or an **atom**.
2. Let A, B be formulas, then $\neg A$,$(A \wedge B), (A \vee B), (A \Rightarrow B), (A \Leftrightarrow B)$ are formulas.
3. Let x be a variable and let A be a formula. Then $(\forall x)A$ and $(\exists x)A$ are formulas.

1.2 Horn clause

A **Horn clause** is a clause C, such that, at most, one atom in the clause is not negated. With this constraint, a clause can take exactly three forms:

- Exactly one unnegated atom and one or more negated ones:

$$C = (\neg q_1 \vee \cdots \vee \neg q_n \vee p) \equiv ((q_1 \wedge \cdots \wedge q_n) \Rightarrow p)$$

- Exactly one unnegated atom and no negated ones:

$$C = p$$

  Since $p = \top \vee p = \neg\bot \vee p = \top \Rightarrow p$, the whole formula holds if p holds when $\top$ holds. Since $\top$ is a tautology, it is always true, so the p must always hold. Hence, it represents a **fact**.

- No unnegated atoms and one or more negated atoms:

$$C = (\neg q_1 \vee \cdots \vee \neg q_n)$$

  This represents a **query**.

[2, p. 407]

1.3 PROLOG

For our purposes, we consider a *pure* version of Prolog, where the source program is a set of Horn clauses. The key aspects of a Prolog program interpretation are **unification** and **backtracking**.

1.3.1 Unification

The unification algorithm can be described as follows. Assume we have two terms, t_1 and t_2.

1. If t_1 and t_2 are both constant terms, they unify only if they have the same value.
2. If t_1 is a logical variable, then t_1 and t_2 unify, and t_1 is **instantiated** to t_2 (if it hasn't been instantiated yet). If both t_1 and t_2 are variables, they are instantiated to each other and share the value.
3. If t_1 and t_2 are both complex terms, they unify only if their functor and arity match and also if all of their arguments unify.

It is worth noting that an uninstantiated variable unifies with any arbitrary term. If a variable is instantiated, unification succeeds only if the value of the instantiated variable is the same as the term it is unifying with.

This unification algorithm is a simplified version of the algorithm described in [5].

1.3.2 Backtracking

The basic idea of backtracking is that of Depth-First graph traversal. The Depth-First search algorithm can be written in pseudocode as seen in the code listing 1.1.

Code listing 1.1 DFS pseudocode

```
procedure DFS(G,v):
    set v as discovered
    For all successors w of the vertex v:
        If state(w) = undiscovered
            call DFS(G,w)
    set v as closed
```

Described verbally, whenever the algorithm finds yet unprocessed vertex, it marks it as discovered and calls itself recursively on all of the vertex's successors. After that, he closes the vertex and returns from recursion. [6]

This corresponds to the act of backtracking. When a goal is reached where the unification fails, the algorithm traverses back (returns from recursion) until another decision can be made (successor, which the algorithm hasn't called itself on yet) and attempts to find the answer using a different path.

Since the backtrack traverses back the choices made, an instantiated variable can become uninstantiated again.

1.3.3 Query satisfaction

To satisfy a query, the following algorithm is used:

- Find the first clause that matches the query's predicate name and arity. The clauses are scanned in the order they are present in the source code.
- If a corresponding clause is found, unify the formal arguments with the actual arguments found in the query.

- If the clause is a fact and unification was successful, the query has been satisfied.
- If the clause is a rule, try to satisfy goals, going left to right. If all goals have been satisfied, the query has also been satisfied.
- In case of any unification failure, initiate backtracking to try a different path.

Chapter 2
Warren Abstract Machine

> *An abstract machine is a semantic model of how a program carries on some computation. The* ***Warren Abstract Machine*** *is an example of this for logic programming. [2, p. 486]*

The following chapter describes the inner workings of the Warren Abstract Machine as described in the book *Architecture of Symbolic Computers* in chapter 17. [2]

2.1 Program structure

Now, we'll describe the structure of a compiled program. "*The general structure of a compiled program closely mirrors the original PROLOG program. For each of the original PROLOG statements, there is a corresponding section of* ***WAM*** *instructions that handles the head unification for that clause and the sequencing through goals that appear on the statement's right-hand side. All such code sections for clauses with the same name are chained together in the order they have been entered in the original PROLOG source code.*" The chaining is done via a *retry-me-else* instruction, discussed in 2.5.1.2 at the beginning of each such code section. "*This permits the computer to rapidly find the next clause to try if the current clause fails.*"

"*Together, each linkage of sections acts as a single procedure, handling appropriate goals (goals with the same predicate name as the clause for which the code has been generated). The internals of the procedure step through the appropriate statements in the expected PROLOG order, with calls to other such procedures as goals are processed.*" This description can be found in section 17.1 in [2].

2.1.1 An Individual Code Section

This subsection will provide an overview of a code section for a single clause. The first few instructions of the code section serve to set up the machine's data structures to permit trying the clause. This includes saving a pointer to the next clause with the same name to try next in case of backtracking, creating a choice point (discussed in 2.4), and allocating space for local variables.

After the initialization, some instructions check that the formal arguments of the clause are unifiable with the actual arguments. These instructions fall under the **Get instructions** category and are discussed in detail in 2.5.3.

The actual arguments can be found in the argument registers. The unification is performed one argument at a time. "*A fail in unification causes this code sequence to be aborted, and control is transferred to the next appropriate code section (as set up by the initialization code). This is*

*called **shallow backtracking**, as only a small amount of information must be reset.*" [2, pp. 489, 490]

After all arguments have been successfully unified, the control is passed to a sequence of instructions for the clause's body. The body contains instructions for each goal, and the sections appear in the order in which they are written in the source code.

Each goal found in the clause's body is treated as a call. Say we have a rule:

```
q :- a(x,y).
```

The method, informally said, to try to find if the goal `a(x,y)` is satisfiable is to load the argument registers with constant `x` and `y`, then branch to the code section created for the predicate `a`. The `put` instructions handle the loading of the argument registers with the actual arguments, and the control is transferred to the entry code for the goal's predicate.

Successfully executing the code for the current goal leads control back to the code that called it. This process of constructing arguments and passing control is then repeated for the subsequent goal. If no matching clauses are found at all, a backtrack is initiated into the caller's code to look for an alternative solution for the prior goal. This type of backtrack, where most of the state information needs resetting, is called a **deep backtrack**.

2.2 Data structures and state registers

The memory of **WAM** is an array of individually addressed locations. The **WAM** CPU contains registers for addressing the memory and executing instructions one by one out of this memory. The memory is divided into several major areas that are used and manipulated by dedicated registers and instructions. To give an example, the `backtrack` instruction (discussed in detail in 2.5.1.3) uses a register labeled as **B** to retrieve older choice point to permit trying different solution for a prior goal. [2, pp. 491, 492]

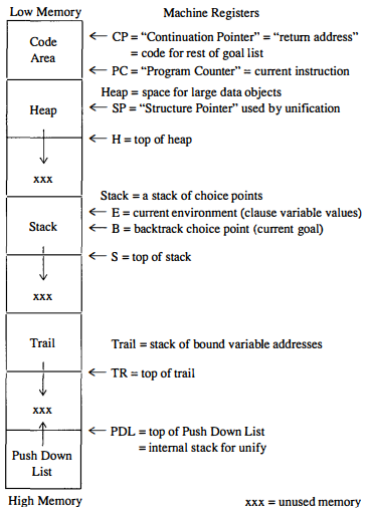

Figure 2.1 Major memory areas of WAM [2, p. 492]

2.2.1 Code Area

The code area is a memory area from which the instructions are fetched one at a time, specified by the **Program Counter** (PC) register. These are the instructions found in the code section for a clause within a specific procedure.

Completing the code section for a clause means that a goal for some clause's right-hand side has succeeded, and the machine should return to and resume execution from that point. This location is indicated by the **Continuation Pointer** (**CP**). [2, pp. 491, 492]

2.2.2 Heap

"*The heap contains structures and lists built during the unification process. These objects are typically too large to fit in either an argument register or a single cell. The **Structure Pointer (SP)** steps through the components of such objects during unification. Storage here is allocated dynamically. The **H register** points to the top of the allocated part of the heap.*" [2, p. 493]

2.2.3 Stack

The stack stores call, return, and environment information for sequencing through the clause code segments. This information for each attempt to solve a goal is called a **choice point** and will be discussed more in detail in section 2.4. The registers associated with the stack are a **B register** (B for backtrack), which points to the most recently created choice point, and a **E register (E for environment)**, pointing to the choice point created when the clause code currently indicated by the program counter was entered. In practice, this means that if we have the following clause:

```
p :- a,b,c
```

and the control has just been passed back from the **b**'s code, the **E** register points to the **p**'s choice point, and the **B** register points to the **b**'s choice point. The **S register** points to the current stack top from which new choice points will be built.

The relationship between **PC**, **B**, and **E** registers is as follows: "*At any point, **PC** points to a code for some clause, and the **E** register gives access to the current values for variables in the clause. The **B** register points to the most recently created choice point and may be equal to or greater than **E**. It is equal to **E** just as the code for the right-hand side of some clause is entered and is greater as goals in that clause's body are solved successfully. All the choice points between **E** and **B** reflect goals that have been solved while handling the right-hand side of **E**'s code.*" [2, p. 493]

2.2.4 Trail

"*The trail is a stack containing references to variables that have received values during the execution, i.e., became bound/instantiated and may have to be unbound at some point in the backtracking process. The **TR register** points to the top of this stack, where new trails can be pushed.*" [2, p. 493]

The act of recording that a variable has been bound is called **trailing**, and it consists of pushing the variable's address to the trail stack.

If a backtrack occurs, the unbinding process, called trail "unwinding," checks for the difference between the **TR** register and the **BTR** field at **B**'s choice point. This difference signals how many new bindings were made during the attempt to satisfy the current goal and have to be reset to attempt to try a different clause.

2.2.5 Push Down List

The Push Down List is a small stack utilized by unification instructions while unifying complex objects. The **PDL register** points to the top of this stack [2, p. 493]. The contents of the **PDL** are triples, consisting of the starting addresses of the actual and formal objects, and the number of consecutive cells to compare, and its exact use will be shown in section 2.5.3.4.

2.2.6 Mode flag

A one-bit flag register that signals whether WAM is in **read mode** or **write mode**. This register is set and used by instructions during head-goal unification. Its purpose is shown in the later section 2.5.5, describing the unify instructions. [2, p. 493]

2.3 Memory word format

As mentioned, the memory in the WAM model is an array of individually addressable locations. Each location is divided into two parts, a **tag** and a **value**. The tag identifies how to interpret the value stored there. The argument registers store the arguments in the same format. The possible combinations are in table 2.1.

Table 2.1 Memory word combinations

Tag	Value
constant	constant value (number, character)
variable	address of this cell
reference	pointer to another cell
list	pointer to list car element
structure	function name and number of arguments
structure-pointer	pointer to a structure

The choice of having the variable contain its own address simplifies the binding/unbinding process. Whenever a variable is bound, the memory location is overwritten with the value it is being bound to. Since the trail contains the memory address of the bound variables, the contents of the memory location pointed to by that address can be just rewritten with a variable word pointing to that address, making it unbound again.

The value of a word with the **list** tag points to a memory location containing the list's **car**, and the subsequent location always contains the list's **cdr**. Both of these can be an arbitrary object.

A structure in Prolog consists of a function symbol and its arguments. As this can not always be encoded in a single memory word, a structure also takes up multiple consecutive words in memory, similar to lists. The memory word tagged **structure** encodes the structure's functor and arity n. The next n subsequent memory cells contain memory words representing the structure's arguments. As with lists, these can be arbitrary objects.

A word with the **reference** tag is a pointer to another memory cell. This can be used to bind objects together (e.g., by assigning variable `X` a reference to the location for variable `Y`, binding them together). In some implementations, the tag used for reference is identical to that for variables, and the distinction is made based on their value fields; the value field pointing to itself signifies an unbound variable. Otherwise, it is a reference. This approach is taken in the actual implementation, discussed in section 4.1. [2, pp. 494, 495]

2.4 Choice Point

The choice point is a major data structure controlling the program execution. It is a set of contiguous locations on the main stack. It contains copies of various machine registers needed to restart a clause's code in case of a return or backtrack.

These objects are built by the `mark` instruction, described in section 2.5.1.1, which is the first instruction in the code sequence for a predicate symbol, modified by an entry code for each clause in that chain, and discarded during backtracking.

To show this more clearly, assume we have the following source code, with its bytecode being under it (listing 2.1).

Code listing 2.1 Sample source code/bytecode

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

a:     mark
       retry-me-else a1
       ...
a1:    retry-me-else quit
       ...
b:     mark
       retry-me-else quit
       ...
c:     mark
       retry-me-else c1
       ...
c1:    retry-me-else quit
       ...
quit:  backtrack
```

As can be seen in listing 2.1, the `mark` instructions are found as the first instructions for each predicate's code section. For `a` and `c`, the last clause's code section for that predicate is labeled `a1` and `c1`, respectively. For `b`, the section labeled `b` is both the first and last clause. If the last clause fails, the goal can't be solved, and backtracking must occur, trying to find a different solution for the previous goals.

"*At any point in a program's execution, each active goal has one choice point, piled up in linear order on the stack. When a successful solution is found, the stack contains a complete history of the individual resolutions used to derive it.*" [2, p. 496]

2.4.1 Choice point structure

The information found in any choice point includes the following:

- A copy of the argument registers `A1,...,An` as they were when the choice point was constructed. These are the arguments that the goal was called with, and storing them enables us to reload the argument registers to their original values if backtracking back to this goal occurs.
- Where to return to if the goal is satisfiable, specifically:
  - The instruction to return to is called the Backtrack Continuation Pointer (**BCP**).
  - The value of the **E** register at the time the goal was called. This field is termed the Backtrack Continuation Environment (**BCE**).
  - To give an example, assume the rule `q(X):- a(X),b(X)` and assume that the choice point has been built for `q`. This choice point holds a local variable `X`, specific to this rule. When the goal `a` is called, the **CP** points to the first instruction after it (in our case, instruction to build the argument for the goal `b`). When the choice point for `a` is constructed, its **BCP** and **BCE** fields are equal to the current values found in the **CP** and **E** registers, respectively. Executing the code for `a` then modifies the **E** and **CP** registers. When the

execution of the code for `a` succeeds, the execution is continued at the instruction found at **BCP** (the first instruction for `b` goal, as expected), and the environment can be restored by the **BCE** field. This allows us to build the arguments for `b` correctly, as it uses a `q`'s local variable `X`, which can only be accessed in this environment.

- The code address for the next clause with the same predicate name to try if the current clause fails. This is called the **Failure Address** (**FA**).
- "*The state of the main memory data structures at the time of the choice point creation, namely, the top of the trail and heap, and a prior choice point in effect before this one, stored in* ***Backtrack Trail*** *(**BTR**),* ***Backtrack Heap*** *(**BH**) and* ***Backtrack B*** *(**BB**) respectively. This is the primary information for the* ***deep backtracking*** *if no clause exists that satisfies the current goal. It represents how to get back to the last goal solved before the current one.*" [2, p. 497]
- The **environment**, which are memory locations for local variables, one such cell of each variable. These are used to store and retrieve current values for the variables.

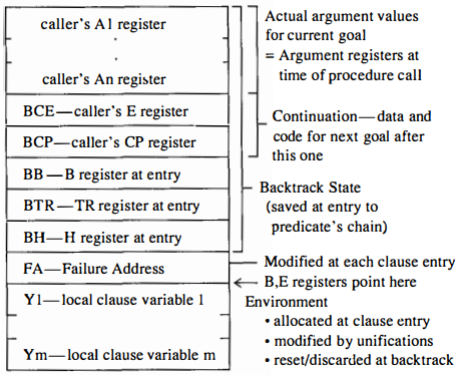

Figure 2.2 The choice point structure [2, p. 496]

2.5 Instruction set

The Architecture of Symbolic Computers [2, pp. 497, 498] divides instructions into five classes based on their intended function:

- **Indexing instructions** to control sequencing through the chain of code sections associated with one procedure (one head predicate symbol)
- **Procedural instructions** to control choice point and environment setup and transfer of control from one chain to another
- **Get instructions** to verify that the formal arguments in a clause unify with current actual arguments (as recorded in the argument registers) and to record the appropriate unifying substitutions
- **Put instructions** to load the argument registers for the next goal on the right-hand side of a clause
- **Unify instructions** to handle get and put instructions of complex objects such as lists and structures

In general, the instructions treat the WAM machine registers in a certain formalized fashion, namely:

- The B register always points to the topmost choice point on the stack
- Once inside the code for some particular clause, the **E** register points to the choice point, which was created when the procedure containing that clause was entered. The only time this may be the same as **B** is just as the code for a clause is entered and before any goals on the right-hand side of that clause are tried. After that, **E** choice point is "buried" under choice points used to solve the right-hand-side goals.
- At the entry to a code segment for a predicate symbol, the **CP** register contains the instruction address to return to if a clause is found in the new procedure that unifies with the current goal and has all of its right-hand-side goals fully satisfiable. The **E** register at this time points to the environment needed to continue execution at **CP**.
- Unless specified otherwise, each instruction increments the **PC** register to point to the next sequential instruction.

To provide an example of how the individual instructions manipulate the state of WAM during execution, assume we have the following Prolog program (code listing 2.2) and a query `c(X)`:

Code listing 2.2 Example program

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

After compiling the source code, bytecode, as can be seen in listing 2.3, will be produced. The query code can also be seen there. This approach is described more in detail in section 3.4.1.

Code listing 2.3 Example bytecode

```
0 a:      mark
1         retry-me-else a1
2         allocate 0
3         get-constant a A1
4         return
5 a1:     retry-me-else quit
6         allocate 0
7         get-constant b A1
8         return
9  b:     mark
10        retry-me-else quit
11        allocate 0
12        get-constant b A1
13        return
14 c:     mark
15        retry-me-else quit
16        allocate 1
17        getv X A1
18        putv X A1
19        call a
20        putv X A1
21        call b
22        return
23 quit:  backtrack
24 query: mark
25        retry-me-else quit
```

```
26          allocate 1
27          putv X A1
28          call c
29          return
```

The state of the machine before execution is as follows:

- The mode flag is set to **write mode**. This choice is arbitrary, as it could be set to the **read mode** by default.
- The environment and backtrack registers, along with the continuation pointer, are not yet set, which will be signaled by their value equaling **xxx**.
- The stack, trail, heap, and push-down list are all empty, which is signaled by the **S**, **TR**, **H**, and **PDL** register equaling 0, as they are pointing to the beginning of their designated memory arrays. The **SP** pointer is also set to 0. This indexing is tied to the representation used in the actual implementation, described in 3.5.
- The program counter is set to 24, as that is the address of the first instruction of the query.

2.5.1 Indexing instructions

"*Indexing instructions chain together and control the code sections for different clauses with the same predicate symbol in their head.*" [2, p. 500]

2.5.1.1 Mark instruction

The *mark* instruction is the first instruction encountered right after the call; it builds a choice point from the current contents of the machine's register. After execution, the **B** register points to the new choice point. [2, pp. 500, 501]

To demonstrate its functionality, assume the WAM is in the state before any instructions are executed, as described above. The building process of a choice point consists of pushing the contents of the machine's registers to stack, which means that the current contents of argument registers, **E** register, **CP**, **B** register, **TR** register, **H** register, and **PC** + 1 are all pushed to the stack.

In our example, that would mean that after executing the first `mark` instruction found in the `query` code section, the stack would look like this (the corresponding choice point fields can be found on the right):

0	xxx	BCE
	xxx	BCP
	xxx	BB
	0	BTR
	0	BH
	25	FA

The instruction also sets the **B** register value to 0, making it point to the just created choice point. The **S** pointer now equals 1, indicating that the next choice point can be constructed on position 1 in its respective memory array.

As this first `mark` instruction was in the query code section, its choice point differs from the choice points constructed by the actual code sections by containing some unset registers and lacking the contents of the argument registers (as they are empty in the initial state). To demonstrate the more common case, let's assume the query's code section has been carried out and the *call c* has been carried out. The WAM state differs from the initial state in the following:

- The environment and backtrack registers are both equal to 0. The continuation pointer is set to 29. The program counter is set to 14. Also, the A1 argument register has been filled with the address of variable X.
- The stack has been modified by the query code instructions and now looks like this:

0	xxx	BCE
	xxx	BCP
	xxx	BB
	0	BTR
	0	BH
	23	FA
	var X	Environment variable at index 0

Now, let's execute the *mark* instruction found in the code section for clause c.

0	xxx	BCE
	⋮	
1	reference to variable X	Contents of A1
	0	BCE
	29	BCP
	0	BB
	0	BTR
	0	BH
	15	FA

The **B** register's value is now 1 and the **S** pointer is now set to 2.

2.5.1.2 Retry-me-else instruction

The `retry-me-else` instruction modifies the **FA** entry in **B**'s choice point to point to the start of the code section for the next possible clause with the same predicate name. The address is provided as an argument to the instruction.

The address set by the `retry-me-else` instruction in a choice point is used if a unification failure occurs in the code for the current clause, or if its goals are not satisfiable. [2, p. 501]

For example, assume that the machine just executed the `mark` instruction as shown in the previous section. After executing the `retry-me-else quit` instruction in the `c`'s code section (line 15), the FA field of the second choice point will change from 15 to 23, and the stack will look like this:

0	⋮	
1		
	⋮	
	23	FA

Now, if a fail sequence is executed while in the code section for clause `c`, the program execution will continue at the address found in the **FA** field, 23 in this case.

2.5.1.3 Backtrack instruction and the fail sequence.

The `backtrack` instruction is the target of the `retry-me-else` for the last clause of a chain. If the program reaches the `backtrack` instruction, none of the clauses satisfy the current goal, and deep backtracking to the predecessor of the current choice point is necessary.

In practice, this is done by backing up one choice point (i.e., reloading **B** from the current choice point) and then initiating the **fail sequence**:

1. Reload the argument registers from **B**'s choice point.
2. Reset the heap top to what it was when **B**'s choice point was built (indicated by the **BH** entry)
3. Use **B** to recompute the top of the main stack, i.e., S = B + 1
4. **Unwind** the trail stack by popping entries (addresses of clause variables) until the **TR** register equals the **BTR** entry in **B**'s choice point.

   For each such entry popped off, the memory location corresponding to that variable is reset to a **variable** entry, indicating that it has been **unbound**.
5. Branch to the code specified by the **FA** field in **B**'s choice point. This is the next possible clause that might satisfy the goal signified by the choice point.

"*After executing the* ***backtrack*** *instruction, all storage associated with the failing goal has been deleted, namely the choice point, values stored on the heap during the execution of the failed goal's code, and any variable assignments made by that code to variables in other choice points. The machine has now been reset to the next deeper choice point on the stack, from which it can try a different clause for the prior goal.*" [2, pp. 501, 502]

In our example, the backtracking occurs in rule `c(X) :- a(X),b(X)` when calling the `b(X)` predicate for the first time. The `X` is now bound to the constant `a` from the first fact, `a(a)`, so the `get-constant b A1` instruction on line 12 will initiate a fail sequence. The code will branch to the `quit` label, where the backtrack instruction can be found. Currently, the values of the relevant machine registers are **S** = 4, **E** = 3, **TR** = 3, **PC** = 23, and finally, argument register **A1** holds the constant `a`. The stack contains the following choice points:

0	⋮	`query`'s choice point
1	⋮	`c`'s choice point
2	1	BCE
	31	BCP
	1	BB
	2	BTR
	0	BH
	5	FA
3	`constant a`	A1 argument register
	1	BCE
	33	BCP
	2	BB
	3	BTR
	0	BH
	23	FA

The last choice point on index 3 is the `b`'s choice point, and the one on index 2 is the `a`'s choice point.

Let's now execute the `backtrack` instruction:

- The previous choice point is reloaded by setting the **B** register to the **BB** entry of the current choice point, i.e., **B** = 2. Then, the fail sequence is initiated again.
- The argument registers are reloaded from the **B**'s choice point. In this case, the **A1** register stored on **B**'s choice point contains a reference to the X variable, currently bound to constant `a`.
- The stack top is recomputed, meaning **S** = **B** + 1 = 3.
- **TR** register and **BTR** differ by 1, so variable unbinding will occur. The trail contains the following values:

0	address of the `X` in the 0th choice point environment, bound to constant `a`
1	address of the `X` in the 1st choice point environment bound to the previous variable
3	address of the `X` from the 0th choice point, bound to constant `a`

  The last entry is popped off, and its address is filled with the variable word containing the same address, making the X variable from the 0th choice point unbound again :

0	address of the `X` in the 0th choice point environment, which is unbound
1	address of the `X` in the 1st choice point environment, bound to the previous variable

- **PC** register is set to the **FA** field, i.e., 5.

After this process, the second clause for predicate `a` can be tried, this time, binding the `X` variable to constant `b`.

2.5.2 Procedural instructions

"*Procedural instructions handle the management of environments and the transfer of control between clauses.*" [2, p. 502]

2.5.2.1 Allocate instruction

The `allocate` instruction is the first instruction of the code section for a clause after the `mark` and `retry-me-else` instructions. It allocates space for all the clause's variables, the size of which is indicated by its argument.

This allocation happens on the stack for the current choice point (designated by **B** register). All locations in the environment are initialized to entries with tag **variable** and value equaling their own memory address, making them unbound variables. These are the variables used in the execution of the following clause. The instruction also sets the **E** register to point to this choice point. [2, p. 502]

2.5.2.2 Call instruction

"*The `call` instruction is used right after loading the argument registers with argument values for a goal in the body of the current clause. It saves the address of the next instruction in the **CP** register and branches off to the clause's entry point corresponding to the predicate symbol in that goal.*" [2, p. 502]

2.5.2.3 Return instruction

The return instruction is the last instruction in a clause segment. Its execution indicates a successful satisfaction of all goals in the body of the clause. The control is then passed back to the continuation address in the caller, found in the **BCP** field of **E**'s choice point. The **E** register is then set to the **BCE** field of the **E**'s choice point.

"*Unlike a return in many conventional computers, the return instruction in WAM does not pop anything off the stack. The choice points built by the procedure being returned from are left intact, pending a possible backtrack that might restart one of them.*" [2, p. 502]

To demonstrate how the WAM state changes when a `return` instruction is executed, consider that we just successfully executed the `get-constant a A1` instruction, as can be found on line 7 in listing 2.3. The **E** register is set to 2, and the **BCP** in **E**'s choice point has the value of 20. This will pass the control back to the code section for clause `c`. The **E** is then set to **BCE**, which is 1. That corresponds to the choice point created for `c`.

2.5.3 Get instructions

"***Get Instructions*** *carry out the unification check between the actual arguments for the current goal and the formal arguments in the head of a clause. At this point, both **B** and **E** point to the same location in the same choice point.*"

"*There is one `get` instruction per formal argument in the clause's head literal, with the code for the k-th formal argument referencing the actual argument in register `Ak`. The form of the `get` depends on the formal argument type.*" [2, p. 503]

2.5.3.1 Get-constant instruction

Get-constant instruction dereferences the argument register indicated by its argument. If the result of dereference has a tag of **constant**, their values are compared. In case of a match, the unification succeeds. In the opposite case, the actual and formal arguments are not unifiable, and the clause cannot be used to satisfy the current goal. The fail sequence is initiated to try another potential clause.

If the dereference result has a tag of **variable**, then the **constant**'s value gets bound to the variable. This is done by trailing the variable and storing a copy of the constant into the variable's memory cell (overwriting the **variable** tag in the process). In our example, this can be seen during the execution of the instruction `get-constant a A1` in the a's code section for example. Right before execution, the **A1** register contains the address of the `X` variable from the 1st environment, which is bound to the `X` variable from the 0th environment. When executed, the variable is trailed, and a constant word with the value *a* overwrites the variable word at the address found in **A1**.

If the result of dereference has any other tag than **constant** or **variable**, the fail sequence is initiated. [2, p. 507]

2.5.3.2 Get-list instruction

The *get-list* instruction dereferences the argument register indicated by its argument and checks whether the result has the tag of **list** or **variable**. If it has any other tag, the fail sequence is executed.

If the result of the dereference has a tag of **list**, then the main part of the instruction has been carried out; the instruction only checks whether the argument is of the correct type. Whether the **car** and **cdr** of the list actually unify with the formal arguments is checked by a pair of unify instructions, described in a later section. To set up WAM for this check, the instruction sets up the mode flag to **read mode** and sets the **SP** register to point to the memory location containing the **car** of the list.

If the tag of the result is a **variable** word, its memory cell is overwritten with a **list** word pointing to the value found in the **H** register. This points the cell to the next available location on the heap, where the subsequent unify instructions will build the actual list. In this case, the mode flag is set to **write mode**, signaling the `unify` instructions to build such a list. [2, p. 507]

2.5.3.3 Get-structure instruction

The `get-structure` instruction is similar to the `get-list` instruction. After the argument register dereference, a tag of **structure-pointer** or **variable** is expected. Otherwise, a fail sequence is initiated.

If the tag encountered is that of a **structure-pointer**, it is dereferenced to access the **structure** word it is pointing to. Then, the functor name and arity of this **structure** word are compared to that stored in the instruction. In case of a mismatch, a fail sequence is initiated.

As with `get-list` instruction, the main part of the instruction is done in the case of a match. The actual and formal arguments have the same function symbol and the same number of arguments, and the following `unify` instructions will check whether the components match. This is signaled by setting the mode flag to **read mode** and setting the SP to point to one memory cell beyond the actual argument's structure cell.

If the actual argument is an unbound variable, the variable is trailed, and its contents are overwritten by the tag of **structure-pointer**, with the value equaling the current **H** register value. The memory cell with the **structure** tag, with its value being the functor and arity from the `get-structure` instruction, is pushed to the heap, and the mode flag is set to **write mode**. [2, p. 507]

2.5.3.4 Getv instruction

The `getv` instruction handles the case where the formal argument is a variable. This is a bit more complex than the already mentioned `get` instructions, as it is often impossible to know whether the variable might have a value at a certain point or what kind of value that might be. This instruction is designed to handle all such cases. The possible combinations are in table 2.2.

Table 2.2 Getv cases

X tag \ Y tag	reference	variable	constant	list	structure-pointer
reference	1	1	1	1	1
variable	2	3	5	5	5
constant	2	4	6	`F`	`F`
list	2	4	`F`	7	`F`
structure-pointer	2	4	`F`	`F`	8

To demonstrate this instruction's functionality, let's look at how it looks in the bytecode:

```
getv Yj Ai
```

where `Yj` is an index of the variable in the current environment.

The operation is labeled **Clear PDL**. First, the PDL is emptied. The `Ai` register is dereferenced, and the result is saved to `X`. The memory contents of the variable at index `Yj` are saved into `Y`. Then, a loop is started, and a case is selected from the table 2.2 using the tags of `X` and `Y`. If the PDL is empty afterward, the instruction is complete. Else do the following:

- Get the first element of the triplet found on the top of PDL and save it into `XA`. Do the same for the second element of the triplet, but save it into `YA`.

- Save the memory contents found at addresses stored `XA` and `XY` into `X` and `Y`, respectively.
- Save the third element of the triplet found on the top of PDL minus 1 into `N` (i.e., `N` = PDL.top[3] - 1).
- Pop the PDL.
- If `N` > 0, push {`XA` + 1, `YA` + 1, `N`} to PDL.
- Repeat the loop.

The individual cases are as follows:

1. Dereference `X` and save it to `X`, then repeat the loop
2. Dereference `Y` and save it to `Y`, then repeat the loop
3. Trail both `X` and `Y`
4. Trail `Y` and bind a copy of `X` to `Y`
5. Trail `X` and bind a copy of `Y` to `X`
6. If values don't match, initiate a fail sequence.
7. Push {value of `X`, value of `Y`, 2} to PDL.
8. If the memory contents at `X` don't match those at `Y`, fail.
   Else push {value of `X` + 1, value of `Y` + 1, arity(mem(`X`))} to PDL, where arity(mem(`X`)) refers to the arity of a structure found in memory at the address stored in `X`.

If the case is `F`, initiate a fail sequence. [2, p. 506]

To showcase how the **PDL** works, assume we have a query in the following form:

```
s(g(x,y)) = s(g(x,y)).
```

Assume the arguments have been already built (discussed in detail in section 2.6.2.2) and the unification is about to be carried out (implemented as `id(A,A)`, discussed in 3.7). The argument registers `A1` and `A2` are loaded with a **structure-pointer** to heap address 3 and **structure-pointer** to heap address 8, respectively. The heap at this moment looks like this:

0	structure g/2
1	constant x
2	constant y
3	structure s/1
4	structure-pointer pointing to heap address 0
5	structure g/2
6	constant x
7	constant y
8	structure s/1
9	structure-pointer pointing to heap address 5

The instructions carrying out the unification can be seen in listing 2.4

Code listing 2.4 Instructions for the unification operator

```
getv A A1
getv A A2
```

The first *getv* will bind the `A` variable to a **structure-pointer**, pointing to the heap address 3. When executing the second `getv`, the A is already bound, so the 8. case in the **Clear PDL** operation is chosen. First, the functor and arity found at the **structure-pointer**'s address are checked for equality. In this case, there is a `s/1` on both addresses 3 and 8, so the following triplet is pushed to the **PDL**: {`3 + 1, 8 + 1, 1`}.

Since the **PDL** is not empty now, the first two elements, 4 and 9 of the triplet are stored in `XA` and `YA`, respectively, and the heap contents found at these addresses are stored into `X` and `Y`. In our case, `X` is a **structure-pointer** pointing to the heap address 5, and `Y` is a **structure-pointer** pointing to the heap address 8. The calculated `N` is 0, so nothing else is being pushed to the **PDL**. Then, the loop repeats.

As both `X` and `Y` are **structure-pointers** again, case 8 is chosen again, and a triple {`6, 1, 2`} is pushed to the **PDL**. This triplet checks the first argument, in our case, a constant `x`. As the **N** calculated for this triple is non-zero, another triple, {`7, 2, 1`}, will be pushed to the **PDL**, this time, checking the second argument, constant `y`.

2.5.4 Put instructions

Put instructions are used to load the argument registers with the actual arguments for the goals in the clauses's body. There is a sequence of `put` instructions before the `call` instruction for each goal, one `put` instruction per top-level argument. Their operation consists solely of copying something and involves no possibility of a failure or backtrack. To give an example, bytecode for the query `a(x,y),b(X,Y)` can be seen in listing 2.5.

Code listing 2.5 Put instructions bytecode

```
put-constant x A1
put-constant y A2
call a
putv X A1
putv Y A2
call b
```

`Put` instructions for complex argument objects such as lists and structures handle only the start of the object and set the **mode flag** to **write mode**, so the subsequent `unify` instructions build the object on the heap. In this case, the argument register is loaded with a pointer to the first component on the heap. [2, p. 510]

2.5.4.1 Put-constant instruction

`Put-constant` instruction loads the argument register indicated by its argument with a word consisting of the tag **constant** and its value, which is also passed as an argument, for example, `put-constant elephant`. [2, p. 510]

2.5.4.2 Putv instruction

The `putv` instruction has the following form in the bytecode

```
putv Y Aj
```

Firstly, the `Y` is dereferenced from the current environment. If the `Y` has a tag of **variable**, the register is loaded with a reference to `Y`. If any other tag is encountered, the register is loaded with the dereferenced result. [2, p. 510]

2.5.4.3 Put-list and put-structure instruction

The `put-list` and `put-structure` instructions handle loading the argument register with a start of the respective object, i.e., a memory cell with the tag of **list** or **structure-pointer**, with the value of the cell equaling the current value found in the **H** register. The mode flag is set to **write mode**, so the following `unify` instructions build the object on the heap.

In the case of the `put-structure`, the cell with the tag of **structure** and the value containing the functor and arity encoded in the instruction is pushed to the heap just after loading the argument register. [2, p. 510]

2.5.5 Unify instructions

Unify instructions are used after a `get` instruction for either a list or a structure to handle their components. They operate in one of two modes as set by the **mode register**, either *read* or *write*, which was set by the prior `get`/`put` instructions.

"*In **read** mode, they attempt to unify the next component of the objects (as pointed to by the **SP** register) with the variable or constant specified in the instruction. A successful match may cause variables to be trailed and bound as in `get` instructions, increasing **SP** to point to the next component. Any mismatch causes a **fail sequence**. Only `get` instructions can set the machine to read mode.*"

"*In **write mode**, these instructions copy the specified constant or variable to the object being built up on the heap. The initial `get` or `put` instructions have earlier given either a register or a variable reference to the start of this object. The **SP** register is not needed in this case.*"
[2, p. 510]

2.5.5.1 Unify-constant instruction

The `unify-constant` instruction is in the form of

```
unify-constant C
```

If the WAM is in write mode, the instruction only pushes the cell with the tag **constant** and value `C` to the heap.

If the machine is in read mode, the memory where the **SP** is pointing is dereferenced, and the **SP** is incremented. If the result of the dereference is a variable, the variable is trailed, and its contents are overwritten with a cell of the tag **constant** and the value `C`.

If the result of the dereference has the tag of **constant**, the values are compared. A fail sequence is initiated in case of a mismatch or if the tag has any other value. [2, p. 511]

2.5.5.2 Unifyv instruction

This instruction can be found in the following form in the bytecode

```
unifyv Y
```

Firstly, the `Y` is dereferenced from the current environment. In case the machine is in the **write mode**, a copy of the `Y` is pushed to the heap

If the WAM is currently in the **read mode**, a memory where the **SP** is pointing to is dereferenced, the `Y` is unified with the dereferenced result, and the **SP** is incremented. The unification process is the same as the one described for the `getv` instruction in the section 2.5.3.4, only this time, the second value is not taken from an argument register but from the heap address pointed to by the **SP** register. [2, p. 511]

2.5.5.3 Procedure for unifying lists and structures

In the WAM model, there are no `unify-list` or `unify-structure` instructions to handle the cases where a formal argument is either a list or a structure and where one or more components are themselves complex terms. The procedure for handling such situations [2, p. 511] is as follows:

1. For each list or structure used as a component of a complex formal argument in the head of a clause, allocate an extra local clause variable not to be used anywhere else
2. When the place in the code is reached where either `unify-list` or `unify-structure` is needed, it is replaced by a `unifyv` instruction with an argument that specifies the new allocated variable.
3. After completion of the top-level code for that formal argument, a `putv` instruction is generated to load some currently unused argument register with the contents of this variable.
4. This is followed by either a `get-list` or `get-structure` instruction against the aforementioned argument register.
5. The components of the structure are handled either with `unifyv` or `unify-constant` instructions or with the procedures described above.

As this process might seem a little unintuitive, let's provide an example. Assume we have a fact in the form of

```
p(s(g(f(x),h(x)), [1,2,s(x)])).
```

The bytecode generated for this fact can be seen in listing 2.6. The `Tx` marks the x-th extra local clause variable. Note that the `allocate` instruction accounts for all of them.

Code listing 2.6 Bytecode for a fact with complex arguments

```
p: mark
   retry-me-else quit
   allocate 7
   get-structure s/2 A1
   unifyv T0
   unifyv T1
   putv T0 A1
   get-structure g/2 A1
   unifyv T2
   unifyv T3
   putv T2 A1
   get-structure f/1 A1
   unify-constant x
   putv T3 A1
   get-structure h/1 A1
   unify-constant x
   putv T1 A1
   get-list A1
   unify-constant 1
   unifyv T4
   putv T4 A1
   get-list A1
   unify-constant 2
   unifyv T5
   putv T5 A1
   get-list A1
   unifyv T6
   unify-constant []
```

```
putv T6 A1
get-structure s/1 A1
unify-constant x
return
```

First, the `get-structure` instruction for `s` is encountered. It has two arguments, a structure `g(f(x),h(x))` and a list `[1,2,s(x)]`. If the arguments were constants or variables, the standard `unify-constant` and `unifyv` instructions would be generated. However, as they are complex terms, the steps above are taken. For both arguments, the `unifyv` specifying the extra variables is generated. This corresponds to step 2. After that, the `T0` variable is put into the `A1` register (step 3), and `get-structure` for `g` is generated (step 4). Then, the structure components are handled (step 5); in this case, they are both complex objects, and the same steps are repeated. When the code is being generated for the two inner complex objects, `f(x)` and `h(x)`, the components of those are both constants, so they are handled by `unify-constant x` instruction.

Now assume that we have received a query in the form of

```
?> p(s(g(f(x),h(x)), [1,2,s(x)])).
```

and that the **A1** register is filled with a **structure-pointer**, pointing to the address 15 on the heap. The exact process of building this heap will be described in the section 2.6.2.2. For now, let's assume the heap looks like this:

0	structure s/1
1	constant x
2	structure-pointer to heap address 0
3	constant []
4	structure h/1
5	constant x
6	constant 2
7	list word pointing to heap address 2
8	structure f/1
9	constant x
10	structure g/2
11	structure-pointer to heap address 8
12	structure-pointer to heap address 4
13	constant 1
14	list pointing to heap address 6
15	structure s/2
16	structure-pointer to heap address 10
17	list pointing to heap address 13

Let's now step through the generated bytecode (listing 2.6) and see how it behaves.

After the `mark, retry-me-else`, and `allocate` instructions, the `get-structure s/2 A1` is encountered. This instruction expects to find either a variable or a structure-pointer for a structure whose functor and arity match `s/2`. The structure pointer found in `A1` satisfies this, so the WAM mode flag is set to read mode, and the **SP** register is set to its address + 1, i.e., 16. Since the **read mode** is set, the next instruction, `unifyv T0`, will dereference the heap at **SP**, unify the `T0` with the contents of the dereference and increment the **SP**. As `T0` is a variable and the dereferenced cell contains a **structure-pointer**, the case 5 of the **Clear PDL** operation (2.5.3.4) is executed. The result of this is that the `T0` is now bound to a **structure-pointer**, pointing to heap address 10. The next instruction, `unifyv T1`, will do the same, this time for a list word pointing to heap address 13. The `T0` is then put into `A1`, and the process is repeated, this time for the more nested components.

2.5.6 Cut instruction

The instructions described so far support pure Prolog, i.e., Prolog without predicates that have side effects, such as cuts.

The cut in Prolog is a predicate that always succeeds on its initial execution but prohibits backtracking back through it. It is denoted by the `!` symbol.

Say we have a clause in the form of:

```
p(...):=q1(...),...,qn,!,r1(...),...,rm(...)
```

When the program execution reaches the cut code, the **B** register points to the most recent choice point built (`qn`'s choice point), and the **E** register points to the one created for `p`.

If backtracking occurs through the cut, it should be as if the `q1` through `qn` choice points never existed and that this clause is the last one possible for the `p` predicate.

This can be achieved by reloading the **B** register from the **E**'s choice point's **BB** register, which is exactly what happens when the cut predicate is first encountered. This choice point is found directly beneath that for `p`. A failure in `r1` will cause a backtrack directly to the desired choice point. [2, pp. 517, 518]

2.6 Prolog-to-WAM compiler

The compiler from Prolog to WAM code comprises two major sections. First, an outer loop that cycles through the clauses and chains them into procedures. The second is compiling a single clause into a code section for the procedure chains.

In addition, we use a symbol table containing an entry for each symbol used as the predicate symbol of the head of some clause. Such entry contains:

- The name of the symbol.
- The memory address of the `mark` instruction.
- The memory address of the last `retry-me-else` instruction compiled for that symbol.
- A list of places where this predicate symbol has been referenced as a goal.

  [2, p. 512]

2.6.1 Procedure-Level Compilation

The program clauses are processed in the order they appear in the source code. For each clause, the following steps are carried out:

1. The next unprocessed clause is selected, and its predicate symbol from the head is obtained.
2. If the head has no code generated for it yet:
   a. The symbol is marked in the symbol table as having code generated for it.
   b. The initial address is recorded as the next available memory location.
   c. `mark` instruction is generated, followed by a `retry-me-else` instruction with no label yet.
   d. The address of the `retry-me-else` instruction is recorded in the symbol table.
3. If the head already has code generated for it:
   a. The next available memory location is stored in the last recorded `retry-me-else` instruction address.

   b. `retry-me-else` instruction is generated with no label.
   c. The address of the `retry-me-else` instruction is recorded in the symbol table.
4. Code is generated for the clause as described in the following section 2.6.2
5. If there are more clauses, go back to step 1.
6. After all clauses have been compiled:
   a. `backtrack` instruction is generated to the next available location, and its address is stored.
   b. For each symbol table entry, the last `retry-me-else` instruction is pointed to this address.
7. The query is compiled.
8. The first instruction of the query's code is the program's starting point.

   [2, pp. 512, 513]

2.6.2 Clause-Level Compilation

The first step in compiling a clause is to calculate the number of local variables needed, including the allowances for temporary variables used for complex objects nested in other complex objects. Then, an `allocate` instruction is generated.

After generating the code for the clause's head and **RHS**, which are described in detail in sections 2.6.2.1 and 2.6.2.2, a `return` instruction is generated. [2, p. 515]

2.6.2.1 Head compilation

During the clause's head compilation, the arguments are processed in the order in which they appear. Assume the k-th argument is being processed:

1. If it is a constant, a `get-constant` instruction is generated, encoding the value of the constant and k-th argument register.
2. If it is a variable, a `getv` instruction is generated, encoding the variable offset and the k-th argument register.
3. If it is a list, a `get-list` instruction is generated, encoding in the k-th variable. Then, for the **car** and **cdr** of this list, either a `unify-constant` or `unifyv` is generated as appropriate. If either **car** or **cdr** is a complex object, a `unifyv` instruction is generated, encoding in a temporary variable, as described in the 2.5.5.3 section.
4. If it is a structure, a `get-structure` instruction is generated, encoding in the functor and structure's arity. Then, the exact same process is carried out for each argument for the list's **car** and **cdr**.

   [2, p. 515]

2.6.2.2 Right-Hand Side compilation

As with the head compilation, the goal literals and their arguments are processed in the order in which they appear in the body. Assume the i-th argument of the current goal is being processed:

1. If the argument is a constant, a `put-constant` instruction is generated, encoding in the constant's value and i-th argument register.

2. If it is a variable, a `putv` instruction is generated, encoding the i-th argument register and the variable offset.
3. If it is a list or structure whose components are either variables or constants, a `put-list` or a `put-structure` instructions are generated as appropriate. Then, for each argument, `unify-constant` or `unifyv` is generated, as required.
4. If it is a list or structure that includes a nested list or structure, the code generation is as follows:
   a. The most deeply nested component is selected.
   b. Currently, unneeded argument register Au is marked
   c. Instructions are generated as described in the previous step for the lists/structures with no complex objects, but results are targeted to the Au register.
   d. A `getv` instruction is generated to place Au into a temporary variable.
   e. Process is then repeated for the next most nested component. When a complex object is encountered, it must already have been processed, so a `unifyv` instruction can be generated, encoding in the temporary variable into which the complex objects have been stored earlier.
   f. Au register is marked as free again.

Then, a `call` is generated as the final instruction. [2, pp. 515, 516]

It is worth highlighting that the processing order for the complex objects for the head is the opposite of the one for the clause's right-hand side, more specifically, outside in and inside out. The reason for this is outlined in the step 4e of the previous algorithm. As all of the nested complex objects have been processed and the complex object with them as its argument is being put into the argument register, it is already completely built.

To provide an example of this inside-out process, assume we have a query `p(s(g(f(x),h(x)), [1,2,s(x)]))`, the same query used in the example in section 2.5.5.3, and let's rewrite the second argument, a list, to a following form `[1|[2|[s(x)|[]]]]`, to demonstrate the nesting more clearly.

The most nested components in the first argument are `f(x)` and `h(x)`, both have depth 3. In the second argument, it is the list `[s(x)|[]]`. The list has a depth of 3, and the nested `s(x)` has a depth of 4, making it the first complex object to be processed. As its argument is a variable, step 3 is taken, and the following bytecode is generated:

Code listing 2.7 Bytecode generated for `s(x)`

```
put-structure s/1 A1
unify-constant x
getv __T0 A1
```

Where the `__T0` is a temporary variable mentioned in step **4c**. After these instructions, the variable `__T0` is bound to a **structure-pointer** pointing to heap address 0, and the heap looks like this:

0	structure s/1
1	constant x

As the `s(x)` has now been processed, the following instructions for loading the list `[s(x)|[]]` can be generated:

Code listing 2.8 Bytecode generated for list

```
put-list A1
```

```
unifyv __T0
unify-constant []
getv __T1 A1
```

Step 3 is taken again, this time starting with a `put-list` instruction. As mentioned in step **4e**, since the variable `__T0` corresponds to the processed nested structure, a `unifyv` instruction referencing the temporary variable can be generated, followed by the `unify-constant` for an empty list. After executing these instructions, the heap contains the following:

0	structure s/1
1	constant x
2	structure-pointer to heap address 0
3	constant []

The steps taken are the same for the rest of the nested complex objects. The final bytecode can be seen in listing 2.9.

Code listing 2.9 Bytecode generated the whole query

```
put-structure s/1 A1
unify-constant x
getv __T0 A1
put-list A1
unifyv __T0
unify-constant []
getv __T1 A1
put-structure h/1 A1
unify-constant x
getv __T2 A1
put-structure f/1 A1
unify-constant x
getv __T3 A1
put-list A1
unify-constant 2
unifyv __T1
getv __T4 A1
put-list A1
unify-constant 1
unifyv __T4
getv __T5 A1
put-structure g/1 A1
unifyv __T3 A1
unifyv __T2 A1
getv __T6 A1
put-structure s/1 A1
unifyv __T6 A1
unifyv __T5 A1
getv __T7 A1
call p
```

After executing all instructions up to the `call` instruction, the heap will look like in the table 2.3.

The `A1` register now contains a **structure-pointer** to heap address 15, which corresponds to the `s/2` structure found as its argument. By going after the addresses in the heap, you can verify that they correspond to the actual arguments. For example, the first argument of `s/2` at address 15 is a **structure-pointer** to address 10. At address 10, you can find a structure `g/2`, and in the two following cells, two **structure-pointers** pointing to `h/1` and `f/1`.

Table 2.3 Heap state prior to call

0	structure s/1
1	constant x
2	structure-pointer to heap address 0
3	constant `[]`
4	structure h/1
5	constant x
6	structure f/1
7	constant x
8	constant 2
9	list word pointing to heap address 2
10	structure g/2
11	structure-pointer to heap address 6
12	structure-pointer to heap address 4
13	constant 1
14	list word pointing to heap address 8
15	structure s/2
16	structure-pointer to heap address 10
17	list word pointing to heap address 13

The above-mentioned procedure mentions `Au` as an unneeded argument register. As you can notice, the bytecode generated here only contains the register `A1`. This is because once the value is stored in the temporary variable, the register `A1` no longer needs to store the value. If the `p` predicate had more arguments, each argument's bytecode uses a different register.

Chapter 3

Implementation strategy

In this chapter, we will outline the implementation and discuss the design decisions taken in the individual modules in the order they are being used when executing a program. First, we describe the source code's parsing and **AST** representation and how the code generation process turns it into bytecode. Attention is also given to the design of the machine's data structures and the interpretation process. Finally, we discuss the arithmetic capabilities of our implementation.

3.1 Lexer

The lexical analyzer splits the source code into a stream of tokens. The individual tokens are represented by a C++ enum `Token`.

The lexer provides an interface to get the next token "on demand" with the `get()` method. A `peek()` method is also available, which returns the current token but does not lex the next token. For some tokens, information about the numeric value or identifier of the token is needed. Methods `identifier()` and `numericValue()` are implemented to provide such functionality.

To offer the functionality of checking whether the currently lexed token matches the expected token during the parsing process, a `match(Token tok)` method is implemented, testing whether the two tokens are equal and lexing the next token.

3.2 Parser

The syntactic analyzer used for parsing the tokenized input is implemented as a predictive parser, which is a type of recursive-descent parsing. Recursive-descent parsing is a top-down approach to syntax analysis where a set of recursive procedures is utilized to process the input. Each nonterminal symbol of the grammar is associated with one procedure. In its basic form, this method may involve trial and error, potentially leading to unsuitable production rule attempts and backtracking.

To mitigate this issue, predictive parsing employs a parsing table. This table is utilized to definitively determine the control flow through the procedure body for each nonterminal symbol. Consequently, the source code can be parsed without the necessity for backtracking. [7, pp. 63, 64] Such a parser can be constructed for a class of grammars called LL(1), which scans the input left to right, producing the leftmost derivation. The 1 means that it uses one input symbol of lookahead at each step to make parsing action decisions. [7, pp. 218, 219]

For parsing our Prolog source code, we've constructed an LL(1) grammar, which can be seen written in EBNF form in listing 3.1. The parsing table for our grammar has been built using an online **LL(1) Parsing Table generator** tool [8].

Code listing 3.1 LL(1) grammar in EBNF

```
Start = [ lower , Predicates, Start ] ;
Predicate = "." | ":-" , Body , "." ;
Predicates = Predicate | "(" , Terms , ")" , Predicate ;
Operator = "=" | "is" ;
Body = Expr2 , BodyOperator , BodyCont | "!" , BodyCont ;
BodyOperator = [ Operator , Expr2 ] ;
BodyCont = [ "," , Body ] ;
List = "[" , ListInner , "]" ;
ListInner = [ Terms, ListCons ] ;
ListCons = [ "|" , Expr2 ] ;
Terms = Expr2 , TermsCont ;
TermsCont = [ "," , Terms ];
TermLower = [ "(" , Terms , ")" ] ;
Expr2 = Expr1 , Expr2R
Expr2R = [ "+" , Expr1, Expr2R ] | [ "-" , Expr1, Expr2R ] ;
Expr1 = Expr , Expr1R ;
Expr1R = [ "*" , Expr, Expr1R ] | [ "/" , Expr, Expr1R ] ;
Expr = lower , TermLower | number
       | List | variable | "(" , Expr2 , ")" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
number = digit , { digit } ;

uppercaseLetter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                | "V" | "W" | "X" | "Y" | "Z" ;

lowercaseLetter = "a" | "b" | "c" | "d" | "e" | "f" | "g"
                | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                | "o" | "p" | "q" | "r" | "s" | "t" | "u"
                | "v" | "w" | "x" | "y" | "z" ;

variable = uppercaseLetter ,
           { uppercaseLetter | lowercaseLetter | digit } ;
lower = { "_" } , lowercaseLetter ,
        { uppercaseLetter | lowercaseLetter | digit } ;
```

The parsing process comprises two parts: the initialization and subsequent parsing. The method `parse()` first prompts the Lexer to get the first token and create the root of the abstract syntax tree. Then, the parsing is started by running the `Start()` method, corresponding to the first rule found in the grammar listing 3.1.

After the process, the abstract syntax tree is completely built and can be accessed through the previously created root node.

3.2.1 Wildcard variables

In Prolog, programmers can utilize wildcard variables, typically denoted by the _ (underscore) symbol. These variables indicate that the result is irrelevant.

During the parsing process, encountering such a symbol triggers the generation of a name for it in the form of `___n`, where `n` represents the number of encountered wildcards thus far (e.g., `___1`). Subsequently, the parser treats the variable as a regular variable.

3.2.2 Abstract Syntax Tree

Abstract Syntax Tree is a representation of the hierarchical syntactic structure of the source program [7, p. 41]. This section will describe individual classes used to represent the nodes.

All classes used to represent the nodes of the AST are all derived from the abstract class `Node`, which offers the `codegen()` method. As each type of node has its own `codegen()` method implemented, the bytecode can be generated just by calling the `codegen()` method on the AST root.

The AST root is always a `ProgramNode`. As the Prolog source code is just a set of clauses, the `ProgramNode` represents that by having a vector of clauses represented by `ClauseNode`. The `ClauseNode` class keeps track of the individual arguments and goals in the body by storing the nodes in their respective vectors and the predicate name.

Arguments and all terms are represented by a `TermNode` class, serving as a base for classes of individual term types. The first class is `ConstNode`, representing either a lowercase atom (e.g., elephant), a natural number, or an empty list (`[]`). Next, `VarNode` represents a logical variable. Lastly, there is the `ComplexNode` class, acting as a base class for `StructNode` and `ListNode`, representing structures and lists, respectively. The `TermNode` class also contains the `codegen_arithmetic` method used by the **is** operator, described in 3.8.

Similarly, a `GoalNode` class serves the same purpose for the goals as the `TermNode` does for terms. Four types of statements can be found on the right-hand side of a clause. Assume a clause in the following form:

```
p :- s(X,b),!, X = c, Y is 1 + 2.
```

The `s(X,b)` is a predicate `s` call, and it is represented by the `CallNode` in the AST. The next goal, `!`, is a cut predicate, represented by the `CutNode` class. Finally, the infix expressions `X = c` and `Y is 1 + 2` are a unification operator and `is` operator, respectively. They are represented by the `UnificationNode` and `IsNode` classes.

To give an example of the AST representation, let us take a look at how this rule would be represented. Note that the `IsNode` has a few `s/1 StructNode`s, even though numbers `1` and `2`

```
ProgramNode
└─ClauseNode p
  ├─CallNode s/2
  │ ├─VarNode X
  │ └─ConstNode b
  ├─CutNode
  ├─UnificationNode
  │ ├─VarNode X
  │ └─ConstNode c
  └─IsNode
    ├─VarNode Y
    └─StructNode +/2
      ├─StructNode s/1
      │ └─ConstNode 0
      └─StructNode s/1
        └─StructNode s/1
          └─ConstNode 0
```

are present on its RHS. This is because we represent natural numbers with their Peano number counterpart and will be discussed in 3.8.

3.3 Bytecode representation

As the bytecode is a series of instructions, a single instruction is represented by the abstract class `Instruction`. The derived classes serve to represent a specific instruction, e.g., `Mark`, `GetConstant`, etc. There is a dedicated class for each instruction mentioned in section 2.5. However, the classes `Call` and `RetryMeElse` differ from the rest. They are not directly subclasses of `Instruction` but of a more specific class called `BranchInstruction`. This class provides an `address` attribute and a `setAddress()` method.

Every instruction class has an execute method, which takes in a reference to the WAM state object and manipulates it according to the type of instruction. This allows the interpreter to execute the program just by cycling through the instructions and running the method.

Now that we have a representation for a single instruction, we describe a class `WAMCode`, representing the code section of WAM's memory. The main methods provided by this class are `addInstructions()`, `getInstruction()`, `popInstructions()`, and `merge()`. The former two are self-explanatory from their name; however, the latter two may deserve more explanation. Both of those instructions are used when working with query code; more specifically, the `popInstructions` removes the last `n` instructions, `n` given by its argument. This is used to remove the query code after the execution is done, and no more answers can be extracted (or the user doesn't ask for more answers). The `merge()` instruction adds the query's bytecode to the source code's bytecode. The class also provides the following methods to work with labels (e.g., `quit`): `addLabel(), removeLabel(), getLabelAddress()`.

Lastly, `addVariable` and `getVariables` methods are present. They store/retrieve all (non-wildcard) variables found in the bytecode. This method only retrieves all user variables entered in a query so their value can be printed out.

3.4 Bytecode generation

The bytecode itself is generated by traversing the AST, starting in its root, using the `codegen()` method. Since some information needs to be shared between the AST nodes during compilation, the `codegen` method expects a reference to a context object. This context is represented by the `CompilationContext` class.

The first major part of the class is a symbol table. This is a data structure used to store information about the identifiers. In our implementation, this entry is represented by a `TableEntry` class and contains information about how many clauses have been already processed for a clause, as it is needed for label numbering.

Context also stores all generated code so far, providing `addInstruction` and `addInstructions` methods, and `code()` method to provide means of retrieving the generated code.

During compilation, there are several occasions where an extra local variable has to be generated, as can be seen in section 2.5.5.3, for example. This can be done by the `generateTempVar()` method, which returns a string in the form `__Tn`, where `n` is the number of variables encountered so far during compilation. There are also two methods, `addVariable()` and `noteVariable()`, which function as follows:

- The `addVariable` adds information about a variable to the generated code. As mentioned before, this is used to extract variables the user enters for their subsequent printing. The wildcard variables or variables generated by the `generateTempVar()` method are not added, as they should not be printed.
- `noteVariable` notes every variable and is mainly used to calculate the `N` argument for the `allocate` instruction and get the variable offsets for their respective environment.

When generating code, some instructions expect an argument register number to know where to load or from where to retrieve a value tied to it. The register allocation process for each clause

is as follows:

- Initially, instructions for head unification are generated. Each instruction for individual arguments utilizes a different argument register. Before any code generation occurs, the available register is set to 1 and is incremented by 1 each time an instruction is generated.
- For right-hand side unification, the available register is reset to 1 before bytecode generation begins for each goal. This is necessary because when unifying the head, the argument registers being dereferenced start from 1. Thus, the instructions for building and loading the registers must also start from 1.

  As the RHS code generation is not as straightforward as the head code generation, let us provide an example:

Code listing 3.2 Code and bytecode example for RHS code generation

```
a(w,x).
b(y,z).
c:-a(w,x),b(y,z).

a:    mark
      retry-me-else quit
      allocate 0
      get-constant w A1
      get-constant x A2
      return
b:    mark
      retry-me-else quit
      allocate 0
      get-constant y A1
      get-constant z A2
      return
c:    mark
      retry-me-else quit
      allocate 0
      put-constant w A1
      put-constant x A2
      call a
      put-constant y A1
      put-constant z A2
      call b
      return
quit: backtrack
```

As can be seen, the `a` and `b` process the head arguments from argument registers 1 and 2, which are the exact registers the predicate `c` loads prior to executing the call instruction.

As described above, the compilation process for a clause can be divided into two parts: bytecode generation for the head of a clause and body generation. This is exactly what the last part of the `CompilationContext` class handles. There is an enum `CodeGenerationMode`, housing two values, **HEAD** and **BODY**. The class variable `m_CGMode` keeps track of these two modes, which can be set by `setHeadGenerationMode` and `setBodyGenerationMode` to **HEAD** and **BODY** respectively. Method `mode()` then provides the information about the state of the class variable. Its exact use will be shown in examples in Chapter 4.

3.4.1 Query compilation

The compilation process described in section 2.6 does not have a special case for queries. Instead, when a query is entered into the REPL in the form of `q(a1,...,an)`, where `ak, k` $\in \{1...n\}$ are its arguments, it is transformed into a clause by prepending it with a special `query` identifier:

```
query :- q(a1,...,an)
```

and then compiled as if it were a standard clause. The address of the initial code segment for this clause is then the starting point of the program execution.

3.4.2 Procedure name encoding

As described in the WAM theoretical model, the code for clauses with the same predicate name is linked into a single chain, regardless of the number of arguments. This can cause a problem with evaluating some query as true, even though it should fail. As an example, let's assume we only have one fact in the source code in the form of:

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

The query

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

should fail, as there is no `q(a)` fact in the knowledge base.

Bytecode, as can seen in 3.3, will be generated for the program and the query. As the first call of the `q` loads, the argument registers `A1` and `A2` with `a` and `b` constants, respectively, and there is no instruction that unloads the registers after the successful execution of the code section; the second call, which should fail, succeeds, as the `A2` register is still loaded with the `b` constant.

Code listing 3.3 Bytecode for q

```
q:      mark
        retry-me-else quit
        get-constant a A1
        get-constant b A2
        return
quit:   backtrack
query:  mark
        retry-me-else quit
        put-constant a A1
        put-constant b A2
        call q
        put-constant a A1
        call q
        return
```

The solution to this problem is to encode the predicate's arity into the name, as they essentially are different predicates. After doing so, the bytecode will look like in the code listing 3.4, and the call to `q/1` will fail as expected, as no address is linked to such procedure.

Code listing 3.4 Bytecode for q with arity encoded

```
q/2:    mark
        retry-me-else quit
        get-constant a A1
        get-constant b A2
        return
quit:   backtrack
query:  mark
```

```
retry-me-else quit
put-constant a A1
put-constant b A2
call q/2
put-constant a A1
call q/1
return
```

3.5 Major data structures

The WAM memory is an array of individually addressed locations. As implementing a memory model is not the main goal of this thesis, this model is partially abandoned in the actual implementation. The individual data structures (e.g. heap, stack) are still implemented as an array of addresses, there is however, no singular continuous chunk as depicted in figure 2.1. Each major data structure has a corresponding array in the `WAMState` class, with methods that perform operations on it. Namely, these operations consist of `Xpush`, `Xpop`, `Xtop`, and `Xat`, where `X` is the name of the data structure, e.g., `heapPush()`. There is also a method `XReg()` for each machine register, where `X` is the name of the register, e.g. `HReg()`.

A single memory word, similar to the instruction representation, is represented as an abstract class `Word`, with classes derived from it serving as an instance of a specific memory word, e.g., `ConstantWord`. Every word class contains a tag and then its value as depicted in table 2.1. An additional method common to all classes is a `toString()` method used when printing variables.

The `VariableWord` design deserves further discussion, as the approach taken in the implementation is that the reference and variable words are represented by the same class, and the contents distinguish whether it's the former or the latter. The exact process of making this distinction will be shown in 4.1.

3.5.1 Argument registers

The **ArgumentRegisters** class encapsulates all of the individual argument registers and provides an interface for their handling. Each register only holds a memory word (or is empty), so it can be represented by an array; a C++ vector is used in the actual implementation. As the registers in WAM are conventionally labeled from 1 to n and the arrays in C++ are indexed from 0 to $n-1$, the method for filling the registers takes this into account and handles the register number adjustment.

The class also offers a method for dereferencing the argument registers. If a reference, or even a chain of references, is encountered, the method repeatedly dereferences until something other than a reference tag is found, as expected.

3.5.2 Heap, Trail, and Push-down list

The Heap, Trail, and Push-down list are the simplest in design. The only required functionality is the ability to insert a new element, provide random access to the elements, and pop the elements from the top so they are all represented as a C++ vector, which satisfies all of the above requirements. The Push-down list differs from the former two by not holding memory words but a triple of values, where the triple is represented as a C++ tuple.

3.5.3 Stack and Choice Point

The Stack performs similar operations to the earlier structures but differs because its contents are more intricate. As the Choice Point in the Warren Abstract Machine comprises a set of

contiguous locations, a `ChoicePoint` class is designed to store each choice point as a single element on the stack.

The `ChoicePoint` class is represented as a C++ struct, encapsulating all of the needed information, like a copy of the argument and machine registers and a vector of words, serving as the space for local variables for a specific clause.

3.6 Interpretation

The `Interpreter` class is designed to shelter all moving parts of the **WAM** and provide a simple interface for evaluating queries.

The class holds an instance of the `WAMState` class and an instance of the `WAMCode` class, where the former represents the **WAM**'s major data structures and state registers, and the latter is the **WAM**'s code area. There is also one more `WAMCode` instance, which holds the currently evaluated query. This comes in handy when asking for multiple answers for a query, which will be discussed more in detail in the following section.

To handle queries, the `compileQuery()`, `evaluateQuery()`, `nextAnswer()`, `setQuery()` and `clearQuery()` are present. The `compileQuery()` compiles the query and adds it to the instructions of the compiled source code. Queries can be evaluated by the `evaluateQuery()` method, which runs a fetch-execute cycle on the instructions, returning a pair of boolean and mapping of variables to the values (if there were any). The `nextAnswer()` method prompts the WAM to look for another possible solution. The `setQuery()` and `clearQuery()` manipulate the current query stored in the `Interpreter` class.

Finally, all of this is encapsulated by the `run()` method, which connects all the parts together and provides a REPL environment for entering and evaluating queries.

3.6.1 Multiple answers

The user may often want all the possible proof sequences for the given query, not just the first one.

In WAM, enabling multiple proof sequences for a given query can be implemented as follows: Upon satisfying the query, the user is prompted whether they desire more answers. If the user inputs the semi-colon `;`, the system initiates a fail sequence, backtracking to the topmost choice point, and execution resumes, attempting to find another suitable solution. If any input other than `;` is provided, the execution for this query concludes, expecting a new query.

3.7 Unification operator

The unification operator = can be interpreted as calling an identity predicate, i.e.

```
id(A,A).
```

In WAM bytecode, this can be expressed as seen in the code listing 3.5

Code listing 3.5 Bytecode for id

```
id:    mark
       retry-me-else quit
       getv A A1
       getv A A2
       return
quit:  backtrack
```

As we want the unification operator to work in our programs, the identity predicate is always inserted into the source code before compilation. Whenever the unification `=` operator is used in the program, it is generated as a call to this predicate. If the backtracking occurs, it is treated as any other predicate and does not require any special handling while providing the expected functionality.

3.8 Is operator

The **is** operator in Prolog serves to evaluate arithmetic expressions; for example, `X is 1 + 2` will yield the result `X = 3`. If the standard unification operator were used, the result would be `X = 1 + 2`. The WAM model described doesn't support arithmetic on its own. For this purpose, the `escape` instruction can be used, which permits the machine to communicate with some other processor that is capable of arithmetic functions.

To allow our implementation to have some arithmetic capabilities without using the `escape` instruction, the **is** operator has also been implemented, using only the instructions described so far. The main idea behind our implementation is to parse and translate the infix arithmetic operations into a sequence of instructions, which will result in the desired outcome. For the purposes of the `is` operator, we parse the infix expression with `+, -, *` and `/` like a structure, for example, an expression `1 + 1` is parsed as `+(1,1)`. It is also worth mentioning that the expressions are parsed as being left associative. This means that an expression `1 + 1 - 1` would yield the
result -1 (`1 - (2)`), and the result would be a **fail**, as -1 is not a natural number.

Whenever the **is** operator is parsed, during code generation, we attempt to generate these instructions for both sides of the operation by the `codegen_arithmetic` method. For our purposes, we encode each natural number present in the source code with its Peano number counterpart, i.e., 1 is encoded as `__s(0)`, 2 as `__s(__s(0))`, etc.

Whenever the *codegen_arithmetic* encounters a list, variable, constant, or a structure that doesn't represent a binary operation (e.g., `+(1,1)`), a call instruction to `__id` is generated, corresponding to the unification operator. The unification target is a variable allocated specially for this, labeled `__arithmeticn`, where `n` depends on the number of variables allocated like this. If a structure representing a binary operation is encountered, the *codegen_arithmetic* method is run on its LHS and RHS.

Let's show this in an example and describe the steps taken. Say we have a query in the form of `X is 1 + 1` and its bytecode as in listing 3.6.

Code listing 3.6 Bytecode for is operator

```
0  putv __arithmetic0 A1
1  putv X(1) A2
2  call __id/2
3  putv __arithmetic1 A1
4  put-structure __s/1 A2
5  unify-constant 0
6  getv __T3 A2
7  call __id/2
8  putv __arithmetic2 A1
9  put-structure __s/1 A2
10 unify-constant 0
11 getv __T5 A2
12 call __id/2
13 putv __arithmetic1 A1
14 putv __arithmetic2 A2
15 putv __arithmetic3 A3
16 call __add/3
```

```
17 putv __arithmetic0 A1
18 putv __arithmetic3 A2
19 call __id/2
```

First, the code for the LHS of the **is** operator is generated. In our case, there is only the variable `X`, so the unification operation is generated for it, corresponding to the first three instructions in the bytecode 3.6. On the RHS, there is an expression in the form of `1 + 1`, which is represented as `+(1,1)`. As this structure represents a binary operation, the process is run again on its LHS and RHS, corresponding to the lines $3 - 12$ in the bytecode. As mentioned before, the Peano number representation can be seen on lines $4 - 6$ and $9 - 11$, as instead of loading the register with the constant 1, it is loaded with a structure `__s(0)`. After generating the code for both sides, the `__add` predicate is called, which carries out the `+` operation. Finally, the variable that has the result of the operation is unified with the variable that holds the `X` variable, yielding the result `X = 2`.

As currently implemented, the **is** operator can handle cases where the arithmetic expressions are known during compile time, like the query `X is 1 + 1` or `X is Y + 1, Y = 2`, or even a more complex one, like the factorial program in listing 3.7.

Code listing 3.7 Factorial using **is** operator

```
fact(0, 1).
fact(N, Res) :-
    N1 is N - 1,
    fact(N1, Tmp),
    Res is Tmp * N.
```

However, assume we have a program with the rule `addOne(X,Y):- Y is X + 1.` and a query `addOne(1+1,Y)`. The variable `X` gets bound to the structure `+(1,1)` during runtime; however, at this point, the code has already been generated. Since there was no arithmetic expression during the compile time in the place of `X`, when the code reaches the `Y is X + 1` goal, it attempts to add `+(1,1)` and 1, resulting in a **fail**.

It is worth noting that since we replace each natural number with its Peano number counterpart, the goal in the form of `1`, for example, will get parsed as a valid goal since it is interpreted as a call to a predicate with the name of `__s`. However, since the names starting with two underscores are reserved (discussed in 3.9.1) and such a predicate is not present in the standard library, a query or a rule with this goal will always fail.

In addition, the structure representation of the binary operations like `+(1,1)` is used strictly as the inner representation, and such structure cannot be parsed directly (`+,-,*,/` are not valid symbols outside the expressions).

3.9 Preprocessor

The process of inserting a predicate into the source code, as mentioned in the section 3.7, is done by a `Preprocessor` class. The main idea of this class is to provide a way to include a standard library of sorts for the programmer. In the current implementation, there is a predicate linked for the unification operator, discussed in 3.7, and for the arithmetic operations `+,-,*,/` used by the **is** operator, discussed in 3.8.

3.9.1 Naming convention

To mitigate potential name collisions between predicates from the standard library and those defined by the programmer, we adopt a naming convention wherein names starting with two underscores (`__`) are reserved for internal usage in the predicate and variable naming. This convention mirrors a similar approach used in the C programming language.

It is worth highlighting that we parse everything, starting with an underscore as a predicate name. The aforementioned use case for variable naming is used in the code generation process for the temporary variables, not in the parsing process.

To give an example, the aforementioned identity predicate is inserted into the source code as

```
__id(A,A).
```

before the actual compilation.

Chapter 4
Implementation

This chapter delves into the implementation details of more complex algorithms and methods, primarily focusing on bytecode generation for nested complex objects and handling infinite terms. Additionally, attention is given to the implementation of the `VariableWord` and the preprocessing and representation of lists. The details described here primarily address parts that may not be immediately intuitive and warrant further explanation. The full implementation with comments can be accessed at `https://github.com/Skotuson/bp`.

4.1 Variable and reference word

In our implementation, the **variable** and **reference** words are represented by a single tag and are distinguished only by the value they store. If the address stored is the same as the address of the word, its tag is a **variable**; otherwise, it is a **reference**.

Listing 4.1 `VariableWord` class

```
class VariableWord : public Word
{
public:
    VariableWord(std::shared_ptr<Word> *ref, std::string name = "");
    void print(std::ostream &os) const override;
    std::shared_ptr<Word> clone(void) const override;
    std::string toString(void);
    TAG tag(void) override;

    std::shared_ptr<Word> dereference(void) const override;
    void bind(std::shared_ptr<Word> w);
    std::shared_ptr<Word> *ref(void) const;

private:
    bool bound(void) const;

    std::shared_ptr<Word> *m_Ref;
    std::string m_Name;
};
```

The `VariableWord` class holds a double pointer to the `Word`, which points to the memory location for some local variable of some clause. Skipping over the constructor and first four methods seen in listing 4.1, let's describe the four methods unique to this class. The first three are self-explanatory. The `ref()` method returns a pointer to the memory location the `VariableWord` is pointing to. The `bind()` method simply writes the argument word w into this memory location. The `dereference()` method returns the word found at this memory location.

The fourth one, `bound()`, serves to check whether the *VariableWord* is actually an unbound variable or a reference, and its implementation can be seen in listing 4.2.

Listing 4.2 `bound()` method implementation

```
bool VariableWord::bound(void) const
{
    std::shared_ptr<VariableWord> vw
        = std::static_pointer_cast<
            VariableWord>(dereference());
    return ref() != vw->ref();
}
```

Described verbally, the method compares the result of the `ref()` method with the result of the dereference of the same address. If the addresses are equal, it means that there is a cycle, and the word represents an unbound variable.

4.2 List representation

In Prolog, a list can be encountered either as an enumeration of its elements, e.g., `[1,2,X]`, or a special notation in the form of `[H|T]`, where `H` unifies with the head and `T` unifies with the rest of the list. Some Prolog implementations also offer a syntactic sugar in the form of `[H1,..,Hn|T]`, where the `H1...,Hn` are the first `n` elements of the lists, and the `T` is the rest of the list. This expression, however, can also be expressed like this `[H1|[H2|[...[Hn|T]...]`.

In our implementation, we support both types of notation. The building of the list happens in a constructor of a list node (listing 4.3).

When parsing the list, it encounters the decomposition or the whole list at once. Let's describe the decomposition case, as the second case uses the same principle. This is represented by the `tail` argument in the constructor not being a `nullptr`. If the decomposition is simple, i.e., `[H|T]`, the `H` and `I` are simply assigned to the list's head and tail, respectively. If multiple head elements are present in the decomposition, only the first one is assigned to the head, and the tail is assigned a new `ListNode`, which recursively constructs the rest of the list.

4.3 Nested complex objects pre-calculation

In section 2.6.2.2, we mention selecting the most nested component when generating code for the right-hand side of a clause. This process can be pre-calculated during the parsing process, as our predictive parser first creates AST nodes for the most nested component. We represent both the list and structure by the `ListNode` and `StructNode`, respectively, which are both derived from a `ComplexNode` class. This node class contains a `NestedPairing`, which maps a `ComplexNode` pointer to a number, which specifies its depth. We can see how the pre-calculation is done for structures in the following listing 4.4. The pre-calculation process is the same for the list node.

The individual arguments of the complex term are iterated over, and a check for their type is carried out. If the type of an argument is that of a structure or a list, the `NestedPairing` of that

argument is fetched, and all its contents are inserted into the current node's `NestedPairing`, with the depth of each entry being increased by 1.

Finally, the node itself is inserted with depth 0. This is done so that the less nested node (the one that has this node as an argument) can add it to its own `NestedPairing` with increased depth and also that the least nested node (the one that houses all the already processed ones) gets its code generated without any need for a special case (shown in the following section).

4.4 Right-hand side bytecode generation

In the previous section, we described the process of pre-calculating the depth of nested complex objects within another complex object. With that knowledge, we can show the actual implementation of the generation process. The process itself is split into two methods, `unifyRHS()` and `unifyArguments()`, and their implementation can be seen in listing 4.5 and 4.6, respectively.

Let's describe the `unifyRHS` method. This method uses the pre-calculated `NestedPairing` `m_Complex` attribute and inserts its contents into a vector, sorting it by depth in descending order. This now means that the most nested component can be found at index 0. Now, this vector is iterated, and a `unifyArguments` method is executed, generating the put and unification instructions for that nested component. After that, the `getv` instruction is generated, which mirrors the approach described in section 2.6.2.2. Finally, the component is inserted into a `ProcessedComplex` structure, which maps the component's pointer to a variable name into which the result was saved. This is important for all the less nested components since their arguments are not only constants and variables but also complex structures. When a complex object is encountered as an argument for the less nested components, it must have already been processed, and a `unifyv` instruction referencing that variable can be used. This can be clearly seen in listing 4.6 in the `else` branch.

As mentioned in the previous section, the least nested node itself is present in the vector with other nested objects (with depth 0), and its code is generated last when all its arguments are already processed and are present in the `ProcessedComplex` structure.

4.5 Infinite terms

In Prolog, it's possible to write a query or goal in the form `X = f(X)`. Intuitively, a variable `X` cannot be bound to a structure that contains `X`, as it creates a cyclic reference. This issue could be addressed by employing an "occurs check," which would cause such unifications to fail. However, this check has not been utilized in our implementation and will not be discussed further.

As we are not using an occurs check in our implementation, only printing the variable's value would lead to an infinite loop. For this reason, we use a flag in each `VariableWord` class to determine whether it already has been visited during printing. If we stumble upon a variable that has already been visited, only its name will be printed, preventing further cycling. For example, printing the result of the aforementioned example will result in `X = f(X)`, as expected.

Let's assume a more intricate scenario: `X = foo(Y), Y = X`. This results in the following variable assignments: `X = foo(foo(Y))` and `Y = foo(Y)`. The value of `Y` aligns with expectations. However, `X` presents a second instance of the `foo` structure. This occurs because during the binding of `Y` to `Y`, `X` is already bound to the structure `foo(Y)`, causing `Y` to be bound to the value of `X`, not to the variable itself. Consequently, when displaying the value of `X`, `Y` needs to be independently accessed for flagging it as visited since they are not directly linked.

Finally, let us show this behavior on an example with a rule `a(Y):- X = foo(X), X = Y` and a query `a(Y)`. The result of such a query is `Y = foo(foo(__1))`, where `__1` marks a local variable for the rule (Such naming is used for all non-user-entered variables to avoid potential name collisions). The situation is essentially the same as described above. In the rule, The `Y` gets bound to the value of `X`, not `X` itself, causing the doubled `foo` occurrence. The reason that the

clause's local variable gets printed is that when the query variable `Y` gets bound to the rule's `Y`, it is bound directly to a structure pointer, which points to a structure that has the local variable as its argument. This local variable has no connection to the query variable from the point of view of WAM, so this is the expected behavior.

Listing 4.3 ListNode constructor

```
// tail is nullptr if no decomposition into [H|T] is happening
ListNode::ListNode(const std::vector<std::shared_ptr<TermNode>> &head,
                   std::shared_ptr<TermNode> tail)
    : ComplexNode("[]")
{
    // List is decomposed using pipe
    if (tail)
    {
        // Simple decomposition in the form of [H|T]
        if (head.size() == 1)
        {
            m_Head = head;
            m_Tail = tail;
        }
        // Decomposition in the form of [X1...Xn|T]
        else
        {
            m_Head = {head.front()};
            std::vector<
                std::shared_ptr<TermNode>> newHead
                    = {head.begin() + 1, head.end()};
            m_Tail = std::make_shared<ListNode>(newHead, tail);
        }
    }

    else
    {
        m_Head = {head.front()};
        std::vector<
            std::shared_ptr<TermNode>> tail = {head.begin() + 1, head.end()};
        if (tail.empty())
        {
            m_Tail = std::make_shared<ConstNode>("[]");
        }
        else
        {
            m_Tail = std::make_shared<ListNode>(tail);
        }
    }
}
```

Listing 4.4 Nested complex objects pre-calculation

```
    for (const auto &arg : m_Args)
    {
        if (arg->type() == STRUCT || arg->type() == LIST)
        {
            ComplexNode *cn = static_cast<ComplexNode *>(arg.get());
            NestedPairing p = cn->getNestedComplex();
            for (const auto &[complexNode, depth] : p)
            {
                m_Complex.insert({complexNode, depth + 1});
            }
        }
    }
    m_Complex.insert({this, 0});
};
```

Listing 4.5 Bytecode generation for RHS

```
void StructNode::unifyRHS(CompilationContext &cctx)
{
    std::vector<ComplexNode *> nested;
    ProcessedComplex processedComplex;
    for (const auto &c : m_Complex)
    {
        nested.push_back(c.first);
    }

    // Sort the nested structures from the most nested to the least nested
    std::sort(nested.begin(), nested.end(), [&](ComplexNode *&a, ComplexNode *&b)
              { return m_Complex[a] > m_Complex[b]; });

    for (const auto &n : nested)
    {
        n->unifyArguments(cctx, processedComplex);
        std::string tempVariable = cctx.generateTempVar();
        cctx.noteVariable(tempVariable);
        cctx.addInstruction(
            std::make_shared<GetVariable>(
                tempVariable, cctx.availableReg(), cctx.getVarOffset(tempVariable)));
        processedComplex.insert({n, tempVariable});
    }
}
```

Listing 4.6 unifyArguments method implementation

```
void StructNode::unifyArguments(CompilationContext &cctx,
                                ProcessedComplex &processedComplex)
{
    cctx.addInstruction(
        std::make_shared<PutStructure>(
            name(), cctx.availableReg(), arity()));
    for (const auto &arg : m_Args)
    {
        TermType type = arg->type();
        if (type == TermNode::CONST)
        {
            cctx.addInstruction(std::make_shared<UnifyConstant>(arg->name()));
        }

        else if (type == TermNode::VAR)
        {
            cctx.noteVariable(arg->name());
            cctx.addVariable(arg->name());
            cctx.addInstruction(
                std::make_shared<UnifyVariable>(
                    arg->name(), cctx.getVarOffset(arg->name())));
        }

        else
        {
            std::string var = processedComplex[arg.get()];
            cctx.addInstruction(
                std::make_shared<UnifyVariable>(
                    var, cctx.getVarOffset(var)));
        }
    }
}
```

Conclusion

This thesis describes the Warren Abstract Machine model, its inner structure, instructions, and evaluation strategy. A documented implementation in C++ of a Prolog compiler and interpreter is created to showcase how a logic language can be implemented using WAM.

The implementation can parse and compile a subset of Prolog language to the WAM bytecode and evaluate it using the interpreter. A preprocessor is implemented to allow linking with a set of standard predicates. It also enables arithmetic operations by replacing natural numbers with their Peano number representations and transpiling mathematical expressions into a chain of predicates.

The WAM model implemented in this thesis is relatively straightforward. Further optimization techniques can be applied to enhance the efficiency of evaluating intricate programs, reducing both runtime and memory requirements. There's room for improving arithmetic capabilities beyond supporting only natural numbers, along with support for I/O operations. Moreover, a very simple stepper has been implemented to view the machine's state during program execution. The stepper could be expanded upon by enhancing its graphical user interface. This improvement could offer another valuable reference when implementing a logic programming language.

Bibliography

1. GABBRIELLI, Maurizio; MARTINI, Simone. Programming Languages: Principles and Paradigms. In: Springer, 2010, pp. 423, 424. ISBN 9781848829138.
2. KOGGE, Peter M. *The Architecture of Symbolic Computers*. McGraw-Hill, 1991. ISBN 9780070355965.
3. STERLING, Leon; SHAPIRO, Ehud Y. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994. ISBN 9780262193382.
4. HILBERT, D.; ACKERMANN, W.; LUCE, R.E. *Principles of Mathematical Logic*. American Mathematical Society, 1999. AMS Chelsea Publishing Series. ISBN 9780821820247.
5. MARTELLI, Alberto; MONTANARI, Ugo. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 1982, vol. 4, no. 2. ISSN 0164-0925. Available from DOI: 10.1145/357162.357169.
6. MAREŠ, Martin; VALLA, Tomáš. Průvodce labyrintem algoritmů. In: 2nd ed. CZ.NIC, 2022, pp. 119, 120. ISBN 9788088168645.
7. AHO, Alfred V. *Compilers: Principles, Techniques and Tools*. 2nd ed. Pearson/Addison Wesley, 2007. ISBN 9780321486813.
8. PECKA, Tomáš. *LL1 Parsing Table* [https://pages.fit.cvut.cz/peckato1/parsingtbl/]. 2022. [Online; accessed 14-May-2024].

Contents of the attachment

- `README.md` ..... brief description of the attachment's contents
- `src`
  - `impl` ..... implementation source code
  - `thesis` ..... source code of the thesis in LaTeX
- `thesis.pdf` ..... text of the thesis in PDF