Master's Thesis

# Automatically extracting the correspondence between the natural language and the pseudo-code descriptions of instruction set manuals

2010432 NGUYEN Thi Hai Yen

Supervisor Prof. Mizuhito Ogawa
Examiners Prof. Minh Le Nguyen
Prof. Kazuhiro Ogata
Associate Prof. Naoya Inoue

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2023

## Abstract

Instruction set architecture defines a CPU and its execution. When an automated assistant tool is considered for binary code, the formal semantics of an instruction set is required as the fundamental basis. However, often the instruction set is large, which requires heavy engineering efforts on specifying the formal semantics. For example, Intel x86 is a CISC architecture and has several thousand instructions. ARM (Advanced RISC Machine) is a RISC architecture that has few hundred instructions, but it has Cortex-A, Cortex-M, and Cortex-R series and each has 10 to 20 variations of chipsets. To overcome such a situation, one possibility is to automatically extract formal semantics from an instruction set manual, written in English. For instance, x86 has Intel developer's manual, and each chipset of ARM has a reference manual, which is open to the public.

However, the interpretation of English description to a formal description, e.g., programming language, is not easy to obtain, since there are no explicit data set to train AI-related methods. Fortunately, often an instruction set manual has both English and pseudo-code descriptions for specifying the same execution step of instruction.

This thesis proposes how to automatically find the correspondence between English and pseudo-code descriptions in the instruction set manual, which will be the first step to automatically obtain interpretation rules from English to a programming language. We first limit ourselves to data processing and the load/store instructions since they are quite uniform and cover 90% of instructions.

We first parse the English and the pseudo-code descriptions. For English sentences, we use the Stanford parser. For pseudo-code descriptions, we design 48 grammar rules for ANTLR. Next, we remove explanations in English and default declarations in pseudo-code to extract essentially describing the operation and the flag update. Lastly, we find the correspondence between the extracted parts of English and pseudo-code descriptions.

We mostly work on ARM and collect 2475 instruction descriptions over 39 chipsets, in which 2251 instructions belong to either the data processing or the load/store instruction groups. Among them, we randomly select the 30 results of each group instruction of the detected correspondence and examine them manually. As far as our selection, our method correctly extracts the correspondence.

***Keywords***— ARM Cortex-A, ARM Cortex-R, ARM Cortex-M, Instruction set, Correspondence, ARM architecture

## Acknowledgments

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Professor Mizuhito Ogawa. Thanks for his dedicated guidance and help during my time studying at Japan Advanced Institute of Science and Technology. In particular, during the process of research and writing the thesis, he took the time to give me comments and suggestions to prepare at the best. I greatly admire his careful and deliberate working style.

Next, I would like to thank Professor Tu Minh Phuong is professor at Posts and Telecommunications Institute of Technology, who recommended me the the Collaborative Education Program between PTIT and JAIST. He gave me useful advice while studying at PTIT and at JAIST.

I would like to thank my friends at JAIST. Thank you to everyone who has helped me have interesting experiences in studying and in everyday life.

Most significantly, I would like to thank my family from the bottom of my heart. My family is a great source of motivation that makes me always strive harder and keeps me working hard. They are everything to me.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## Motivation

Instruction set architecture defines a CPU and its execution. When an automated assistant tool is considered for binary code, the formal semantics of an instruction set is required as the fundamental basis. Formal semantics is a method of studying meaning that has its roots in logic, philosophy of language, and linguistics.[1]. However, often the instruction set is large, which requires heavy engineering efforts on specifying the formal semantics. For example, Intel x86 is a CISC architecture and has several thousand instructions. ARM (Advanced RISC Machine) is a RISC architecture that has few hundred instructions, but it has Cortex-A, Cortex-M, and Cortex-R series and each has 10 to 20 variations of chipsets. To overcome such a situation, one possibility is to automatically extract formal semantics from an instruction set manual, written in English. For instance, x86 has Intel developer's manual, and each chipset of ARM has a reference manual, which is open to the public.

Almost all current smartphones and tablets, even some laptop computers use ARM processors. ARM (Advanced RISC Machine) is a company owned by Arm Holdings, Ltd. developed processors based on the RISC (Reduced Instruction Set Computer) architecture. The "R" in ARM stands for RISC - Simplified Instruction Set Computer. There are several ARM variations in various lines, such as Cortex-A, Cortex-M, and Cortex-R.

Extracting ARM formal semantics refers to the process of formally and mathematically capturing the meaning and behavior of ARM instructions to present the semantics of ARM instructions clearly and without ambiguity using defined rules, logic, and mathematical notation. The interpretation of English description to a formal description, e.g., programming language, is not easy to obtain, since there are no explicit data set to train AI-related methods. Fortunately, often an instruction set manual has both English and pseudo-code descriptions for specifying the same execution step of instruction.

This thesis proposes how to automatically find the correspondence between English and pseudo-code descriptions in the instruction set manual, which will be the first step to automatically obtain interpretation rules from English to a programming language. Specifically, the thesis extracts a sentence in the natural language description that corresponds to any statement or block of statements in pseudo-code, and conversely, a pseudo-code statement corresponds to one or more sentences in the natural language description. We first limit ourselves to data processing and the load/store instructions since they are quite uniform and cover 90 % of instructions.

After discovering the correspondence between description and pseudo-code, we can develop research to automatically extract ARM formal semantics, especially Java source. The goal of extracting ARM formal semantics is to reduce the tedious human effort involved in tool development, which includes symbolic execution. In other words, use the generated results of ARM formal semantics extraction to develop a dynamic symbolic execution tool. Symbolic execution is a program analysis technique that allows a systematic study of program behavior. Rather than running a program with specific inputs, symbolic execution uses symbolic values to represent the possible range of input. Symbolic execution can be used for various purposes, including software testing, verification, and vulnerability detection. Using dynamic symbolic execution to reconstruct the control-flow graph (CFG) to investigate indirect jumps and understand malware behavior.

Furthermore, the discovery of the mapping between natural language descriptions and their corresponding pseudo-code representations promises tremendous potential for improving the automation, optimization, and interpretation of instructions in a wide range of computational domains. Since then, there has been a growing desire to equip machines with the ability to read and interpret the complexity of human language.

## Overview

The thesis proposes a method to find the correspondence between the natural language description and pseudo-code of the instruction set, called automatic section extraction. The thesis implemented on processors implementing the Cortex-M, Cortex-A, and Cortex-R series architectures. We first parse the English and the pseudo-code descriptions. For English sentences, we use the Stanford parser. For pseudo-code descriptions, we design 48 grammar rules for ANTLR. Next, we remove explanations in English and default declarations in pseudo-code to extract essentially describing the operation and the flag update. Lastly, we find the correspondence. The main idea of finding the correspondence between the natural language description and the pseudo-code of the instructions is to break the description and pseudo-code into sections, then label those sections, and finally map them together. The first thesis applies techniques in natural language processing to automatically extract the main operation section and the flag update section. From the main operations section, the thesis further extracts the results section and the execution section, and from the execution section, it further extracts some subsections, function names, operators, etc., if possible.

We mostly work on ARM and collect 2475 instruction descriptions over 39 chipsets, in which 2251 instructions belong to either the data processing or the load/store instruction groups. These instructions are collected from the Arm Developer Reference Manuals. Among them, we randomly select the 30 results of the detected correspondence and examine them manually. As far as our selection, our method correctly extracts the correspondence. The work also aims to use a generalization method that can be deployed on other architectures such as x86 and MIPS.

## Thesis Outline

This thesis is composed of 9 chapters. The chapters are summarized as follows:

- **Chapter** 1 Introduction: Introduction to motivation, goals, and solution ideas.
- **Chapter** 2 Instruction set manuals: In this chapter, the thesis presents the definition of the instruction set, types of the instruction set and some processor architectures or instruction set architectures (ISAs), especially ARM processor architecture.
- **Chapter** 3 Natural language processing techniques: This chapter presents some existing natural language processing techniques used including Syntactic parsing, methods to convert text into vectors, similarity measures between sentences, and the library that uses them.
- **Chapter** 4 Observation of automatically extracting the correspondence between the natural language and the pseudo-code descriptions: This chapter presents an observation of automatically extracting the correspondence between the natural language and the pseudo-code descriptions.
- **Chapter** 5 Syntactic parsing: In this chapter, the thesis presents a syntactic analysis of the natural language description of instruction, as well as procedures for the creation of grammar rules for abstract parsing that represent the structure of pseudo-code.
- **Chapter** 6 Automatic section extraction: In this chapter, the thesis describes in detail how sections of natural language description and pseudocode are automatically extracted.
- **Chapter** 7 Automatically extracting the correspondence: In this chapter, the thesis presents a method to identify the correspondence between natural language description and pseudo-code and some manual testing strategies. The thesis also gives a comprehensive example to illustrate the steps of section extraction and corresponding extraction.

- **Chapter** 8 Experiments: This chapter presents the results that the thesis has experimented. This chapter presents the results that the thesis has experimented. The thesis experimented with the introduced method for three ARM Cortex processors.

- **Chapter** 9 Conclusion: In this chapter, the thesis summarizes the main contributions of the thesis, and some future works are also mentioned to suggest some directions to improve and extend our proposed method to other architecture such as x86, MIPS, etc.

# Chapter 2

# Instruction set manuals

In this chapter, the thesis presents the definition of the instruction set, types of instruction set and some processor architectures or instruction set architectures (ISAs), especially ARM processor architecture.

## 2.1 Instruction set

Instruction set architecture (ISA) or computer architecture defines how the CPU is controlled by the software[2]. The ISA specifies the supported data types, registers, how the hardware manages main memory, key features, what instructions can be executed by a microprocessor, and the input/output model of several ISA implementations. The ISA can be expanded to include new instructions or other functions and to support larger addresses and data values. Instruction set architecture refers to the overall design of the CPU, including the instruction set, the registers, the memory model, and the interrupt handling.

Some examples of instruction set architectures are x86, ARM, MIPS, PowerPC, SPARC.

There are many different types of instruction set architectures:

- **Complex Instruction Set Computer (CISC)**: The instruction set in CISC architectures is larger and more complex, with more operands. x86 is an example of a CISC architecture.
- **Reduced Instruction Set Computer (RISC)**: The instruction set in RISC architectures is small and simple, with few operands. ARM and MIPS are examples of RISC architectures.
- **Very Long Instruction Word (VLIW)**: Some examples of VLIW architectures include EPIC and MMX.

- **Explicitly Parallel Instruction Computing (EPIC)**: Some examples of EPIC architectures include Itanium and PA-RISC.
- **Minimal Instruction Set Computers (MISC)**: Some examples of MISC architectures include CLU and PICO

The instruction set is a collection of machine-level instructions that a processor can understand and execute. It defines the operations that the processor can perform, the format of the instructions, the encoding of the instructions, and the behavior of the processor when executing each instruction. Instruction set refers to the specific set of commands that a particular processor or computer architecture supports, including the opcodes (operation codes), operands, and addressing modes.

There are some examples of instructions in the instruction set of x86 architectures [3]:

- Arithmetic instructions:

  ADD: This instruction adds two values together.

  SUB: This instruction subtracts one value from another.

  MUL: This instruction multiplies two values.

- Logic instructions:

  AND: This instruction performs a bitwise logical AND operation between two values.

  OR: This instruction performs a bitwise logical OR operation between two values.

  XOR: This instruction performs a bitwise logical XOR operation between two values.

  NOT: This instruction performs a bitwise logical NOT operation on a value.

- Control Transfer Instructions:

  JUMP: Jumps to a specified memory address or label.

  CALL: Call a procedure or subroutine

- String instructions:

  LODSB/W/D/Q.

  STOSB/W/D/Q

  MOVSB/W/D/Q

  SCASB/W/D/Q

  CMPSB/W/D/Q

## 2.2 ARM instruction sets

ARM architecture (Advanced RISC Machines) is a RISC architecture developed by ARM Limited. The ARM architecture integrates key features commonly found in RISC architectures. These include a spacious and consistent register file, a load/store architecture where data processing operations are performed exclusively on register contents rather than memory contents, and straightforward addressing modes that rely on register values and instruction fields for load/store addresses. Moreover, the ARM architecture offers additional advantages such as instructions that combine shifts with arithmetic or logical operations, addressing modes that automatically increment or decrement to optimize program loops, load and store multiple instructions to enhance data throughput, and the ability to conditionally execute many instructions to maximize execution speed [4].

### 2.2.1 ARM architecture versions, variants and profiles

The ARM instruction set architecture has developed significantly. The instruction set has nine major versions, the number versions 1 to 9. Some examples of ARM architecture variants include ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv5TEJ, ARMv6, ARMv7M, ARMv7A, ARMv7R and more.

Since ARMv7, each profile refers to a subset of features and instructions designed to specific application domains or system requirements. There are three profiles:

- Application profile (A-profile): This profile is designed for high-performance applications running on complex operating systems such as Linux and Android. Arm Cortex-A processor family or Cortex-A series and Cortex-X series implement the A-profile of ARM architecture.
- Real-time profile (R-profile): This profile is designed for high-performance real-time applications such as hard disk controllers and networking equipment media players. Arm Cortex-R processor family or Cortex-R series implements the R-profile of ARM architecture.
- Microcontroller profile (M-profile): This profile is designed for low-power devices embedded systems such as IoT devices. Arm Cortex-M processor family or Cortex-M series implements the M-profile of ARM architecture.

### 2.2.2 Hierachy of ARM Processors

The ARM architecture has had various versions and profiles, each with its own set of features and capabilities. Here is a summary of the different versions and profiles:

- ARMv1, ARMv2, and ARMv3: These are early architectures of the ARM processor.

- ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv5TEJ, and ARMv6: These architectures are implemented by ARM classic processors. Starting with ARMv4, they support the ARM instruction set. ARMv4T, ARMv5T, ARMv5TE, ARMv5TEJ, and ARMv6 additionally support the Thumb instruction set. ARMv6-M is a subset of ARMv7-M. From ARMv6 onwards, these architectures not only support the ARM and Thumb instruction sets but also the Thumb-2 instruction set.[5]
- ARMv7: ARMv7 provides three profiles: ARMv7-A, ARMv7-R, and ARMv7-M. These profiles are implemented by Cortex-A, Cortex-R, and Cortex-M processors, respectively. ARMv7-M does not support the ARM instruction set (An ARMv7E-M implementation is an ARMv7-M implementation that contains the DSP extension).[6]
- ARMv8: Similarly, ARMv8 also provides three profiles: ARMv8-A, ARMv8-R, and ARMv8-M. Currently, ARMv9 provides ARMv9-A. ARMv8-A and ARMv8-R support the A32 and T32 instruction sets. ARMv8-M supports the execution of T32 instructions. Additionally, ARMv8-A and ARMv9-A support the A64 instruction set. Depending on the execution state (AArch64 or AArch32), ARMv8-A can support A64, A32, or T32 instruction sets. The AArch64 state supports the A64 instruction set, while the AArch32 state supports the A32 and T32 instruction sets [7].

The ARM instruction set comprises a collection of 32-bit instructions, while the Thumb instruction set consists of 16-bit instructions and is a subset of the ARM instruction set. The Thumb-2 instruction set is an extension of the original Thumb instruction set with many 32-bit instructions. It combines the advantages of code density of Thumb code with the performance of the full 32-bit ARM instruction set [4]. In previous versions of the ARM architecture, the T32 and A32 instruction sets were referred to as the Thumb instruction set and the ARM instruction set, respectively. The A64 and A32 instruction sets use 32-bit instruction encodings, while the T32 instruction set uses both 16-bit and 32-bit instruction encodings.

There are also instruction set architecture extensions such as Jazelle, ThumbEE, Floating-point, and Advanced SIMD.

The table 2.1 details the processors on which each architecture is implemented. [8] [9], [10]

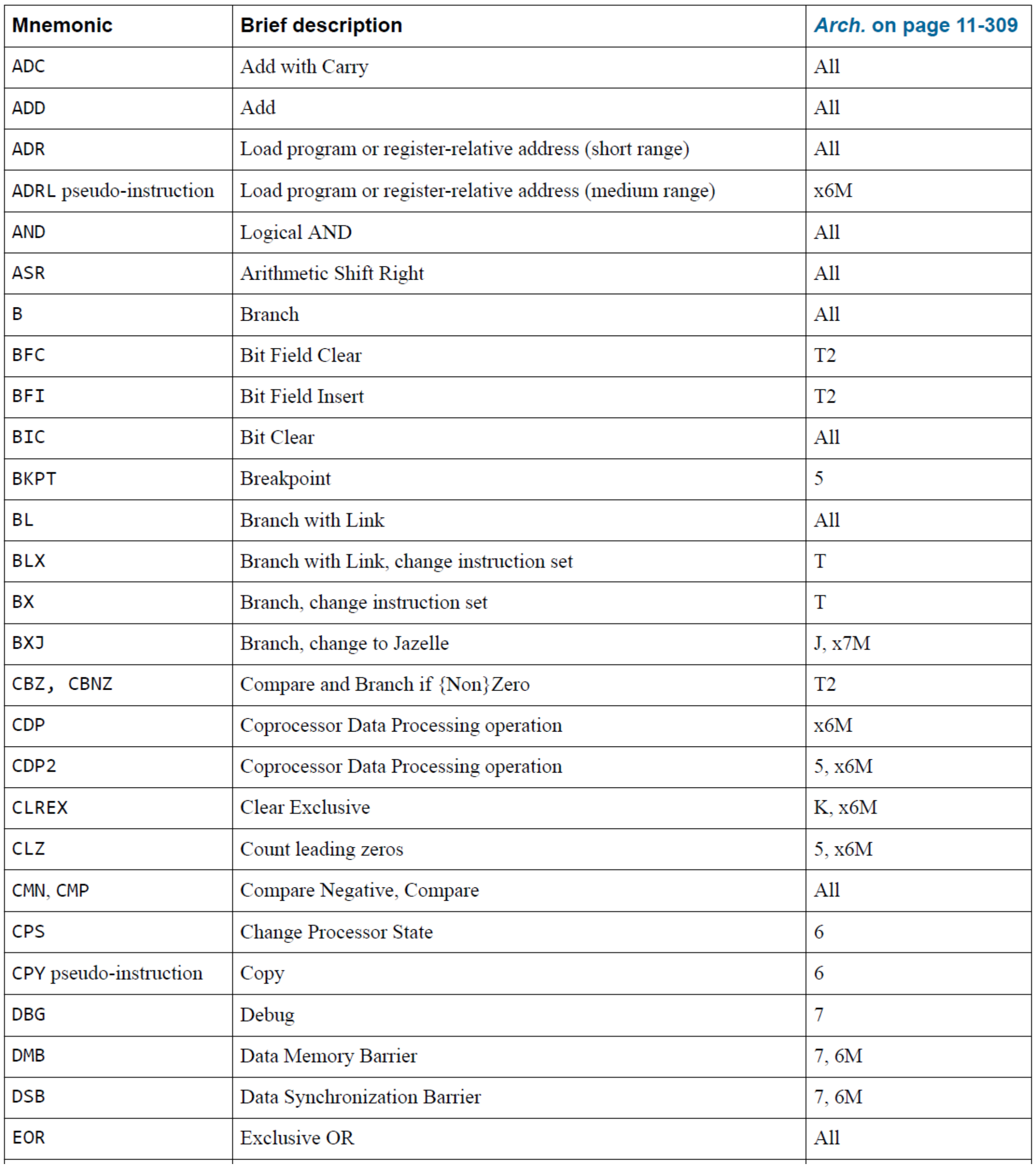
| **ARM architecture** | **Profile** | **Processor** | **Instruction set** |
|---|---|---|---|
| ARMv4 | Classic | | ARM instruction set |

| | | | |
|---|---|---|---|
| ARMv4T | Classic | | ARM instruction set<br>Thumb instruction set |
| ARMv5T | Classic | | ARM instruction set<br>Thumb instruction set |
| ARMv5TE | Classic | | ARM instruction set<br>Thumb instruction set |
| ARMv5TEJ | Classic | | ARM instruction set<br>Thumb instruction set |
| ARMv6 | Classic | | ARM instruction set<br>Thumb instruction set<br>Thumb-2 instruction |
| ARMv6-M | Microcontroller | ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1 | Thumb instruction set |
| ARMv7-A | Application | ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17 | ARM instruction set<br>Thumb instruction set<br>Thumb-2 instruction |
| ARMv7-R | Real-time | ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8 | ARM instruction set<br>Thumb instruction set<br>Thumb-2 instruction |
| ARMv7-M | Microcontroller | ARM Cortex-M3 | Thumb instruction set<br>Thumb-2 instruction |
| ARMv7E-M | Microcontroller | ARM Cortex-M4, ARM Cortex-M7 | Thumb instruction set<br>Thumb-2 instruction |

| | | | |
|---|---|---|---|
| ARMv8-A | Application | Cortex-A32, ARM Cortex-A34, ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72, ARM Cortex-A73 | A32 instruction set<br>T32 instruction<br>A64 instruction set |
| ARMv8.2-A | Application | Cortex-A55, ARM Cortex-A65, ARM Cortex-A75, ARM Cortex-A76, ARM Cortex-A77, ARM Cortex-A78 | A32 instruction set<br>T32 instruction<br>A64 instruction set |
| ARMv8-R | Real-time | ARM Cortex-R52, ARM Cortex-R82 | A32 instruction set<br>T32 instruction<br>A64 instruction set |
| ARMv8-M | Microcontroller | ARM Cortex-M23, ARM Cortex-M33 | T32 instruction |
| ARMv9-A | Application | Cortex-A510, A710 and A715 | A32 instruction set<br>T32 instruction<br>A64 instruction set |
| ARMv9.2-A | Application | Cortex-A520 and A720 | A32 instruction set<br>T32 instruction<br>A64 instruction set |

Table 2.1: Hierarchy of ARM Processors

Figure 2.1 shows a part of the ARM instruction set [11].



| Mnemonic | Brief description | *Arch.* on page 11-309 |
|---|---|---|
| ADC | Add with Carry | All |
| ADD | Add | All |
| ADR | Load program or register-relative address (short range) | All |
| ADRL pseudo-instruction | Load program or register-relative address (medium range) | x6M |
| AND | Logical AND | All |
| ASR | Arithmetic Shift Right | All |
| B | Branch | All |
| BFC | Bit Field Clear | T2 |
| BFI | Bit Field Insert | T2 |
| BIC | Bit Clear | All |
| BKPT | Breakpoint | 5 |
| BL | Branch with Link | All |
| BLX | Branch with Link, change instruction set | T |
| BX | Branch, change instruction set | T |
| BXJ | Branch, change to Jazelle | J, x7M |
| CBZ, CBNZ | Compare and Branch if {Non}Zero | T2 |
| CDP | Coprocessor Data Processing operation | x6M |
| CDP2 | Coprocessor Data Processing operation | 5, x6M |
| CLREX | Clear Exclusive | K, x6M |
| CLZ | Count leading zeros | 5, x6M |
| CMN, CMP | Compare Negative, Compare | All |
| CPS | Change Processor State | 6 |
| CPY pseudo-instruction | Copy | 6 |
| DBG | Debug | 7 |
| DMB | Data Memory Barrier | 7, 6M |
| DSB | Data Synchronization Barrier | 7, 6M |
| EOR | Exclusive OR | All |

Figure 2.1: A part of ARM instruction set

# Chapter 3

# Natural language processing techniques

This chapter presents some existing natural language processing techniques used including Syntactic parsing, methods to convert text into vectors, similarity measures between sentences, and the library that uses them.

## 3.1 Syntactic parsing

Parsing involves analyzing the structure of a symbol sequence. This is possible with simple language, computer languages and data structures. Parsing is essential in many fields, including natural language processing, programming language compilers, and data processing systems. The purpose of parsing is to discover the meaning and syntactic validity of a sequence of symbols using a specific formal grammar. In natural language processing, parsing is used to examine sentence structure and interpret meaning. In programming language compilers, parsing is a technique used to assess the syntactic correctness of programs and examine their overall structure. And in data processing systems, parsing is used to analyze the structure of data and identify its meaning.

Parsing should follow the rules of formal grammar. Context-Free Grammar (CFG) is the most common grammar formalism for expressing language syntax.

**Definition 3.1.1.** A context-free grammar (CFG) G is defined as a four component $G =< \Sigma, N, S, R >$, where: [12]

- $\Sigma$: finite sets of terminal symbols
- N: finite sets of nonterminal symbols
- $S \in N$: the start symbol.
- R: a finite set of production rules of the form $A \rightarrow \alpha$

- $A \in N$: a nonterminal
- $V = N \cup Z$, $\alpha \in V*$: a sequence of symbols

Chomsky Normal Form (CNF) is a normal form of CFGs. It is a context-free grammar consisting of two types of rules: $A \to \alpha$ and $A \to BC$. Any context-free grammar (CFG) can be normalized to Chomsky Normal Form (CNF).
Phrase Structure Grammar is a context-free grammar that describes the structures of natural languages. A syntax tree is a derivation tree generated by a Phrase Structure Grammar.
Syntactic parsing is the determination of the grammatical structure of a sentence. It usually uses a phrase structure grammar. A sentence can have many parse trees that follow a grammar.

There are common parsing algorithms: CKY Algorithm, Chart Parsing Algorithm, Earley Algorithm, GLR Method, etc.

Parsing strategies are divided into 3 groups:

- Trial-and-false (backtracking): top-down, bottom-up
- Dynamic planning strategy: CYK, Earley, etc.
- Deterministic strategies: LL, LR, etc.

**Types of parsers**

Two types of parsing algorithms are top-down parsing and bottom-up parsing. These methods apply various methodologies for analyzing and comprehending the structure of sequences based on formal grammar rules.

Top-down parsing is an approach in which the parser starts with the top-level grammar symbol (often the start symbol) and recursively expands the grammar rules to create a parse tree. The goal is to find leftmost derivatives of an input stream by sequentially applying the production rules from top to bottom. This method uses a recursive descent strategy in which each non-terminal in the grammar corresponds to a parsing procedure that matches and consumes input tokens. If a particular parsing path does not match the input, backtracking can be used to explore alternative paths.[13]

Bottom-up parsing takes a different approach from top-down parsing. In bottom-up parsing, the parse tree is built from the input tokens, starting with the terminal symbols and going down to the non-terminal at the top level. It uses a shift-reduce strategy in which the parser shifts input tokens onto a stack until a sequence of tokens matches the right-hand side of a grammar rule.[13]

**Probabilistic Context-Free Grammars (PCFG)**

Probabilistic Context-Free Grammar (PCFG) is a technique for statistical parsing. In parsing, ambiguity often occurs: A sentence can be parsed into different parse trees. Therefore, we can select a tree using the Probabilistic Context-Free Grammar method.

For no annotated data, the inside-outside algorithm to train can be used. Grammar learning is the concept of training a PCFG. We have a hidden data issue if a parsed training corpus is not accessible. We can use the Inside-Outside approach to train the parameters of PCFG using the unannotated phrases of language.

## Syntax parsing for natural language

Every sentence in every language consists of part of speech. Parsing is a fundamental problem in natural language processing because it has many applications for solving more difficult problems. Several libraries are available for syntax parsing for natural language. Some popular ones include:

- Stanford Parser [1]
- spaCy [2]
- Gensim [3]
- NLTK [4]

**Example 1**

A syntax tree of the sentence *"There is an apple on the table."*

[1] https://nlp.stanford.edu/software/lex-parser.shtml

[2] https://spacy.io/

[3] https://pypi.org/project/gensim/

[4] https://www.nltk.org/

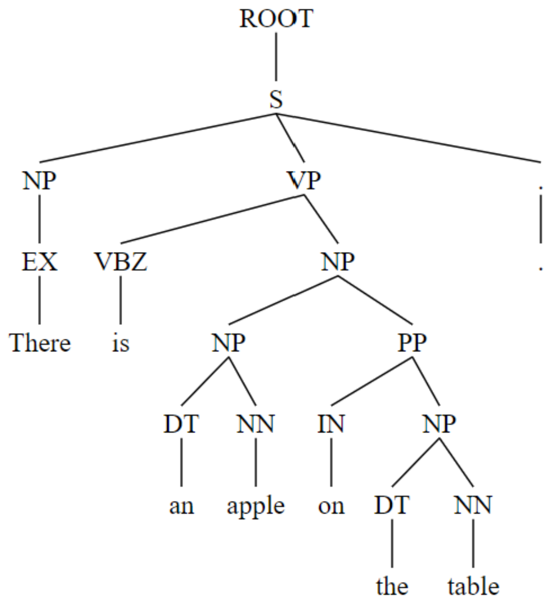


Figure 3.1: Example of a sentence syntax tree

## Syntax parsing for programming language

An Abstract Syntax Tree (AST) is a data structure commonly used in computer programming. It denotes the grammatical structure of programming source code. It is a hierarchical tree-like representation that captures the logical structure of the code but not the grammatical peculiarities. In most cases, Abstract Syntax Tree is used as an intermediate representation during the compilation or interpretation process. Each node in an AST represents a programming language construct, such as a function call, assignment, if statement, loop, and so on. Edges link the nodes, representing the relationships between the structures.

Several libraries are available for syntax parsing. These libraries provide tools to generate parsers for different languages and can also be used to create custom parsers. Some popular ones include:

- ANTLR [5] is a parser generator written in Java that can also generate parsers for many other languages. ANTLR is based on an LL algorithm.
- APG [6]is a Superset Augmented BNF recursive-descent parser generator. ABNF is a BNF version aimed to improve bidirectional communication protocol functionality.
- Lark [7]

[5]https://www.antlr.org/
[6]https://sabnf.com/
[7]https://github.com/lark-parser/lark

- Lrparsing [8]
- PLY [9]

**Example 2**

Figure 3.2 is the Abstract Syntax Tree of this code.

```
if d == 15 then
{
    ALUWritePC(result);
}
else
{
    R[d] = result;
}
```

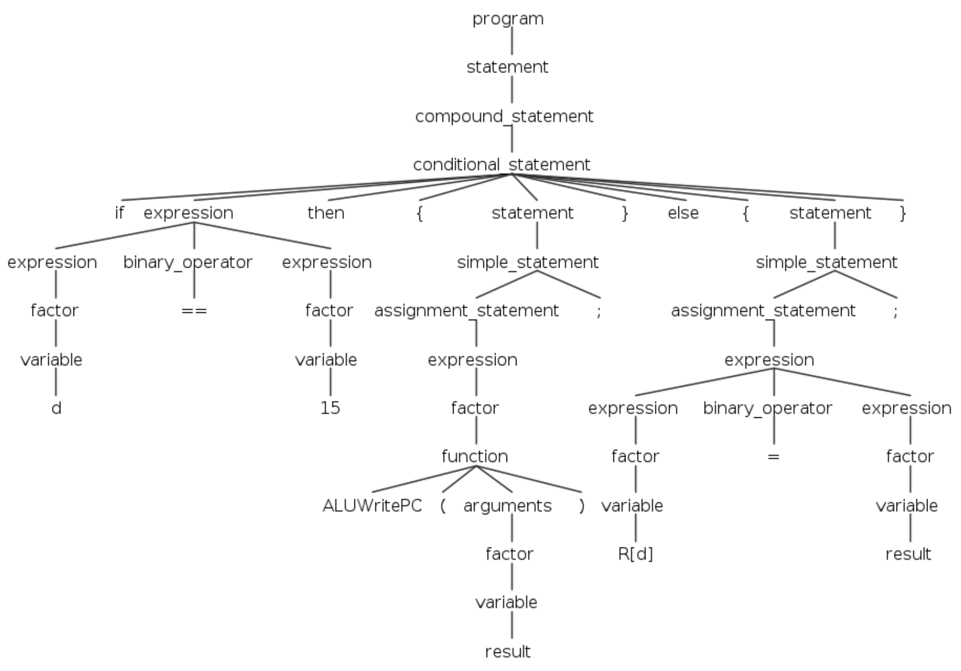


Figure 3.2: Example of Abstract Syntax Tree

[8]https://lrparsing.sourceforge.net
[9]https://github.com/dabeaz/ply

## 3.2 TF-IDF

TF-IDF (Term Frequency Inverse Document Frequency) calculates two things: term frequency and inverse document frequency. TF-IDF weights are used to evaluate the importance of a word in a text.

TF (term frequency) is used to estimate the frequency of occurrence of words in the text. The simplest method to calculate a raw count of instances a word appears in a document.

$$\text{TF}(t,d) = \frac{f(t,d)}{max(f(w,d) : w \in d)}$$

where:

- $TF(t,d)$: frequency of occurrence of word t in the document d
- $f(t,d)$: Number of occurrences of the word t in the document d
- $max(f(w,d) : w \in d)$: Number of occurrences of the word with the highest number of occurrences in the document d

IDF (Inverse Document Frequency) is used to estimate the importance of a word. This means, how common or rare a word is in the entire document set. However, there are some words that are often used but are not important to express the meaning of a passage called stopwords.

$$\text{IDF}(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

where:

- $IDF(t,D)$: IDF value of word t in the set D
- $|D|$: Total number of documents in the set D
- $|d \in D : t \in d|$: represents the number of documents in set D containing the word t.

So, these TF and IDF values of each word for a specific sample are multiplied to obtain the feature vectors for that sample.

The formula for TF-IDF (Term Frequency-Inverse Document Frequency) can be represented as follows:

$$\text{TF-IDF}(t,d,D) = \text{TF}(t,d) \times \text{IDF}(t,D)$$

where:

- $TF - IDF(t, d, D)$ represents the TF-IDF score of term t in document d relative to the document set D.
- $TF(t, d)$ represents the term frequency of term t in document d.
- $IDF(t, D)$ represents the inverse document frequency of term t in the document set D.

## 3.3 Cosine similarity measure

There are many methods used to measure the distance of two vectors, such as:

- 1. Euclidean Distance
- 2. Manhattan Distance
- 3. Chebyshev Distance
- 4. Minkowski
- 5. Cosine Similarity
- 6. Hamming Distance
- 7. Leveshtein Distance

In this thesis, we used the Cosine similarity measure to calculate the similarity of two sentences.

Cosine similarity is often used to solve the problem of Euclidean distance in multidimensional space. It is defined as the cosine of the angle between the two vectors, which can be calculated as the dot product of the vectors divided by the product of their magnitudes. 2 vectors in the same direction will have a cosine similarity of 1 and opposite directions will have a value of -1.

Cosine similarity is frequently used in text analysis to measure the similarity between two documents, where each document is represented as a vector of word frequencies or TF-IDF values. The cosine similarity of the two documents ranges from 0 to 1, where 1 indicates they are identical and 0 indicates they have no words in common. The formula to calculate the cosine similarity between two vectors A and B is as follows:

$$\text{cosine similarity} = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Where:

- n is the dimension of the vectors
- $A_i$ and $B_i$ are the $i$-th elements of vectors $\vec{A}$ and $\vec{B}$, respectively
- $\|\vec{A}\|$ and $\|\vec{B}\|$ are the magnitudes of vectors $\vec{A}$ and $\vec{B}$, respectively

## 3.4 Association Rule Mining

### 3.4.1 Apriori Algorithm

The Apriori algorithm is the algorithm used to compute the association rules between two objects. It denotes the relationship between two or more items. The Apriori is used to locate frequent itemsets in a transaction database which is a collection of items that appear in transactions on a regular basis that exceeds a predetermined minimum support threshold.

Support: The frequency with which an itemset appears in the dataset is measured by support which is determined as the proportion of transactions containing the itemset to total transactions.

Confidence: The possibility that an association rule is true is measured by confidence which is determined as the ratio of X's support to the support of the itemset including both X and Y.

Lift: Lift quantifies how much more probable itemset Y is to be purchased when itemset X is purchased, as opposed to when Y is purchased independently of X.
newline Following the identification of common itemsets, the Apriori method may be used to build association rules. It produces rules based on minimal confidence for each frequent itemset.

### 3.4.2 FP-Growth (Frequent Pattern Growth) Algorithm

Both FP-Growth and the Apriori Algorithm are used for mining frequent itemsets in transactional databases or datasets.

The FP-Growth algorithm employs a unique data structure known as the frequent-pattern tree (FP-tree), which stores item set association information. Each FP-tree node represents an item from the item collection. The item sets are represented by the lower nodes, whereas the root node represents null.

The key concept of the FP-Growth algorithm is its divide-and-conquer strategy. It projects and partitions databases based on the currently discovered frequent patterns and grows such patterns to longer ones in the projected databases[14].

# Chapter 4

# Observation of automatically extracting the correspondence between the natural language and the pseudo-code descriptions

This chapter presents an observation of automatically extracting the correspondence between the natural language and the pseudo-code descriptions.

## 4.1 Data collection

In this section, the thesis provides comprehensive guidance on the essential techniques and methods of the data collection process, as well as a step-by-step explanation of the complexities and intricacies underlying the competent processing of the description and pseudo-code of the instruction set.

The thesis follows three phases to extract instructions from the ARM architecture instruction set:

- Step 1: Collect the data from the PDF file of the document. The PDF file of the document is downloaded from the ARM Developer Website official site.[1]. The thesis collected data on the application level programmer's model and main instruction (Arm instruction and thumb instruction).

  In the documentation for the ARM Architecture Reference Manual, in general, each instruction description has a format that includes: the instruction section title, introduction to the instruction, instruction encoding, assembler syntax, pseudo-code describing how the instruction works and notes. Based on the general format of the

[1]https://developer.arm.com/documentation

document that the thesis has observed, the thesis extracts each block corresponding to each instruction. Each block contains information about the instruction including the instruction section title, introduction to the instruction, instruction encoding, assembler syntax, pseudo-code that describes how the instruction operates and notes. Based on the title and special characters, the thesis has identified the three most important parts for the research: instruction section title, introduction to the instruction, and pseudo-code.

- Step 2: Extract collected raw data and normalize data.
  - Step 2.1: Process the natural language description of the instruction:

    Different from website data extraction, extracting data from PDF files often encounters problems such as encoding issues, complex layouts, missing text, and so on. After automatic data collection, we check random data and perform normalized data such as removing special characters, removing information that the thesis considers redundant, and removing error data. during the collection process.
  - Step 2.2: Process the pseudo-code of instruction:

    Similar to processing the natural language description of the instruction, for the pseudo-code of the instruction, the thesis also removes the error data due to the collection process. However, to facilitate the construction of abstract syntax trees, the thesis has added the braces { } to mark statements belonging to the same conditional or repeating statement.

- Save data: The data from instructions after going through Step 2 has saved the information of each instruction including description and pseudo-code. Each instruction is stored information as in Table 4.1

| | |
|---|---|
| **Mnemonic** | ADC (immediate) |
| **Description** | Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result. |
| **Pseudo-code** | |

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32,
     ↪  APSR.C);
    if d == 15 then           // Can only occur for ARM
     ↪  encoding
        ALUWritePC(result); // setflags is always FALSE
         ↪  here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Table 4.1: The specification of instruction Add with Carry (immediate) .[4]

## 4.2 ARM Specification

### 4.2.1 Components of ARM Architecture

The ARM architecture is characterized as a load/store architecture in which only load and store instructions have direct access to memory. The addressing range varies depending on whether the 32-bit or 64-bit architecture is used. Data processing instructions, on the other hand, work exclusively with register contents. Depending on the execution state AArch64 or AArch32, the code has access to 32-bit general-purpose registers or 64-bit general-purpose registers [15].

A ARM processor contains the following main components:

- **Memory**: The ARM processor uses a load-store architecture, loading data into general-purpose registers before processing and returning it afterward. It supports multiple memory access modes, including byte, half-word, and word access.

- **Registers**: The ARM processor utilizes a large set of general-purpose registers to store temporary data and memory addresses during instruction execution. These registers store processed data and memory addresses for accessing stored data in memory. Some of the registers are reserved in each mode for the specific use of the core. ARM processors provide general-purpose and special-purpose registers including X registers in AArch64 and R registers in AArch32, other registers include the Stack Pointer (SP), Link Register (LR), Program Counter (PC) Application Program Status Register (APSR) and various special-purpose registers for specific functionalities.

- **Flags**: The APSR (Application Program Status Register) in the ARM architecture indeed holds the conditional flags (N, Z, C, V). In some versions of ARM, the APSR also holds the GE (Greater than or Equal) flags and the Q (Saturation) flag. APSR, the naming convention changed to PSTATE (Processor State) with the introduction of the ARMv8-A architecture to reflect the wider range of status bits and the introduction of a 64-bit execution state (AArch64). The condition flags in the APSR are [11]:

  - N flag (Negative condition flag): Set to 1 if the result of the last arithmetic operation was negative, and 0 otherwise.
  - Z flag (Zero condition flag): Set to 1 if the result of the last arithmetic operation was zero, and 0 otherwise.
  - C flag (Carry condition flag): Set to 1 if the last arithmetic operation resulted in a carry, and 0 otherwise.
  - V flag (Overflow condition flag): Set to 1 if the last arithmetic operation resulted in an overflow and 0 otherwise.

  In some specific ARM architectures, the APSR may also include additional flags:

  - Q flag (The Overflow or saturation flag): indicates overflow or saturation. In ARMv5TE, ARMv6, and later, the Q flag is set to 1 when saturation has occurred in saturating arithmetic instructions, or when overflow has occurred in certain multiply instructions.
  - GE (The Greater than or Equal flags): indicate the results from individual bytes or halfwords of an operation.

  These flags can be updated based on the execution of instructions. The flags are used by the processor to determine the outcome of conditional instructions. The flags can also be used by software to track the results of arithmetic and logical operations. It means, that when an instruction is conditionally executed, its effect on the model operation of programmers, memory, and coprocessors is determined by the state of the condition flags (N, Z, C, and V) in the APSR. If the condition specified by the cond field is satisfied by the flags, the instruction performs its normal operation. However, if the flags do not meet the specified condition, the instruction behaves as a NOP (No-Operation). In this case, the execution flow advances to the next instruction as usual, including any checks for exceptions, but the conditional instruction itself has no further effect [4].

### 4.2.2 Instruction group

The thesis groups the instructions based on the functional grouping of the instructions to aid in the identification of sections to be extracted. Different versions or documentation of the ARM architecture may contain or organize information differently. Group the instructions into the following groups [4]:

- **Branch instructions**

  This group of instructions serves several purposes:

  - 1. Branching to subprograms, which allows the program to call and return from functions.
  - 2. Facilitate loop constructions by branching backward.
  - 3. Enable conditional branching forward based on specific conditions.
  - 4. Modify the execution of the following instruction conditionally without actual branching.
  - 5. Switch the processor between A32 and T32 states for flexibility in instruction set usage.

- **Data Processing instructions**

  The Data Processing instructions primarily operate on the general-purpose registers. They provide a variety of operations that use the contents of two registers, such as addition, subtraction, and bitwise logic. The result is then stored in a third register. These instructions can also operate on the value within a single register or combine a register value with an immediate value provided within the instruction. In addition, the Long Multiply instructions provide a 64-bit result distributed across two registers.

  - Standard data-processing instructions.
  - Shift instructions
  - Multiply instructions
  - Saturating instructions
  - Saturating addition and subtraction instructions

  - Packing and unpacking instructions
  - Parallel addition and subtraction instructions
  - Divide instructions
  - Miscellaneous data-processing instructions

- **Load/store instructions**

  Instructions in this group are designed to load or store the value of a single register from or to memory. These operations may involve loading or storing a 32-bit word, a 16-bit half-word, or an 8-bit unsigned byte. In particular, byte and half-word loads can be either sign-extended or zero-extended to fill the 32-bit register. Some specific instructions are designed to facilitate the loading or storing of 64-bit double-word values that span across two 32-bit registers.

- **Load/store multiple instructions**

  Load/store multiple instructions allow the loading or storing of any subset of the general-purpose registers directly from or to memory.

- **Status register access instructions**

  Status register access instructions are responsible for moving the contents of a status register to or from a general-purpose register. These status registers hold important information about the current state of the processor, and these instructions enable access to their contents for further processing or manipulation.

- **Other instructions**: Instructions that do not belong to the above instruction groups

The thesis groups instructions into groups because the groups have different structures for natural language description and the structure of pseudo-code description. A fundamental approach to effectively analyzing and understanding the complexity of different instructions is to systematically classify them into different groups, each with its own unique characteristics. This deliberate categorization is important because it allows for a comprehensive understanding of the underlying patterns within each group and facilitates the identification of correspondence of sections in both the natural language descriptions and the pseudo-code.

## 4.3 Observation of automatically extracting the correspondence between the natural language and the pseudo-code descriptions

### Methodology and Approach

The first step of extracting the correspondence between the natural language description and the pseudo-code description of the instructions is to divide the natural language description and pseudo-code description into sections as templates, then label those sections, and finally map them together. Figure 4.1 illustrates this.

The ARM instruction set is organized into groups of instructions that perform similar operations, such as data processing, branch, or load/store instructions. The thesis recognizes that each group of instructions usually has some of the same natural language description structures. In addition, the order of operations or sequence of steps that the processor follows to execute each instruction is also somewhat similar within each group. For example, the arithmetic instructions all involve performing an arithmetic or logical operation on one or more register values and an optional immediate value and storing the result in a destination register. The natural language description for these instructions explains the operation performed, the operands used, and the result produced. The order of operations for these instructions involves fetching the instruction and its operands, performing the specified operation, and storing the result. The thesis selects the load/-store instructions group, load/store multiple instructions and all sub-groups in the data processing instructions group to implement. The reason is that these groups have quite a large number of instructions, accounting for about 90 % of the instruction of the base instruction of the application level programmer's model.

Next, the thesis extracts the section for both the natural language description and the pseudo-code description. Each section of the natural language description can consist of one or more sentences. Each section of the pseudo-code description can be a statement or a block of statements. For example, in the data processing instructions group, the thesis has extracted the *"Flags update section"* and *"Main operation section"*. After obtaining the main operation section, it proceeds to extract the "result section" and "execution section", etc. Sub-sections are further extracted if possible, e.g. extracting function names, operators, etc.

Once the natural language description and pseudo-code description sections have been independently identified, the mapping process begins, wherein each section from the natural language description is matched with its corresponding counterpart in the pseudo-code description.

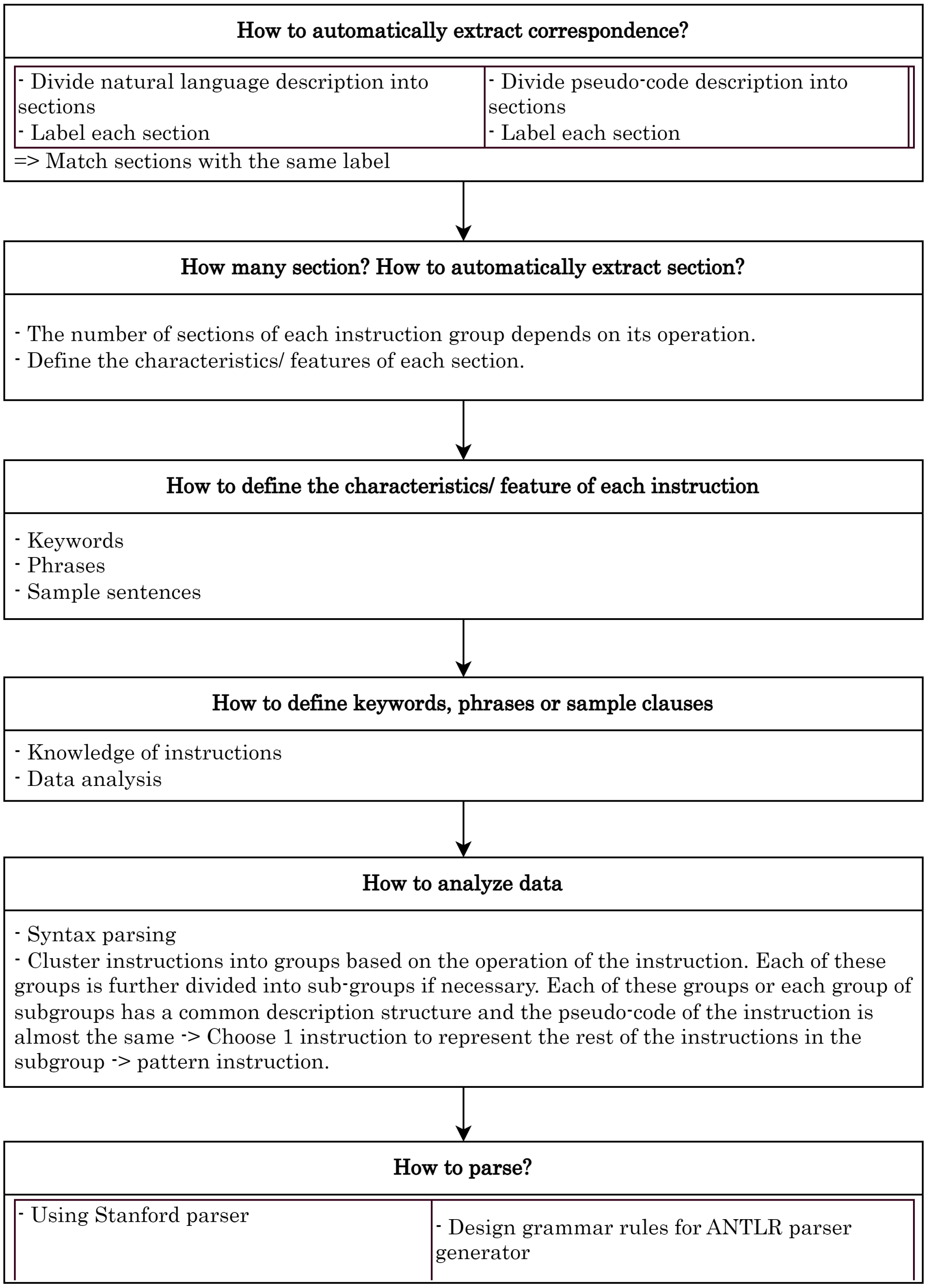

How to automatically extract correspondence?
- Divide natural language description into sections
- Label each section
- Divide pseudo-code description into sections
- Label each section
=> Match sections with the same label
How many section? How to automatically extract section?
- The number of sections of each instruction group depends on its operation.
- Define the characteristics/ features of each section.
How to define the characteristics/ feature of each instruction
- Keywords
- Phrases
- Sample sentences
How to define keywords, phrases or sample clauses
- Knowledge of instructions
- Data analysis
How to analyze data
- Syntax parsing
- Cluster instructions into groups based on the operation of the instruction. Each of these groups is further divided into sub-groups if necessary. Each of these groups or each group of subgroups has a common description structure and the pseudo-code of the instruction is almost the same -> Choose 1 instruction to represent the rest of the instructions in the subgroup -> pattern instruction.
How to parse?
- Using Stanford parser
- Design grammar rules for ANTLR parser generator

Figure 4.1: The approach of automatically extracting correspondence

## Process of automatically extracting correspondence

The following figure 4.2 illustrates the steps to determine the correspondence of natural language description and pseudo-code description. The next three chapters detail the implementation steps using the NLP techniques discussed in Chapter 3.

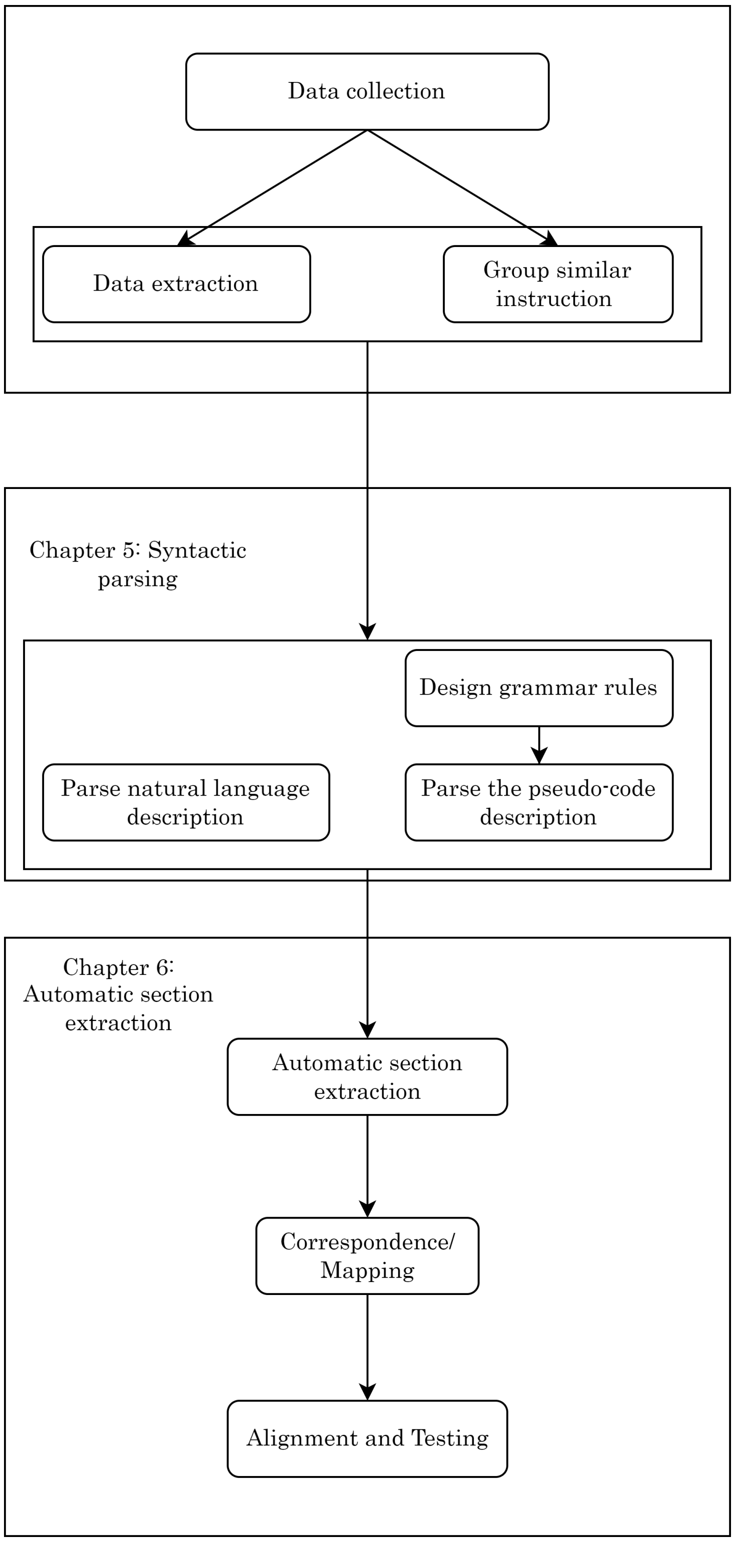

Data collection
Data extraction
Group similar instruction
Chapter 5: Syntactic parsing
Design grammar rules
Parse natural language description
Parse the pseudo-code description
Chapter 6: Automatic section extraction
Automatic section extraction
Correspondence/ Mapping
Alignment and Testing

Figure 4.2: Overview of automatically extracting correspondence process

In the chapter 5, the thesis presents a syntactic analysis of the natural language description of instruction, as well as procedures for the creation of grammar rules for abstract parsing that represent the structure of pseudo-code.

In the chapter 6 Automatic section extraction: In this chapter, the thesis describes in detail how sections of natural language description and pseudo-code are automatically extracted.

In the chapter 7 Automatically extracting the correspondence: In this chapter, the thesis presents a method to identify the correspondence between natural language description and pseudo-code and some manual testing strategies. The thesis also gives a comprehensive example to illustrate the steps of section extraction and corresponding extraction.

# Chapter 5

# Syntactic parsing

In this chapter, the thesis presents a syntactic analysis of the natural language description of instruction, as well as procedures for the creation of grammar rules for abstract parsing that represent the structure of pseudo-code.

## 5.1 Syntax parsing of the natural language description

In this section, the thesis presents the syntactic analysis of the natural language description of instruction.

A sentence can be parsed into different parse trees. In the process of parsing the description, the thesis approaches many different methods and libraries of parsing. The main library used primarily for parsing is the Stanford Parser [1]. The thesis randomly selects the parsed results to check the accuracy of the parse tree.

Through the application of part-of-speech (POS) tags, each word is given a special label that indicates its specific syntactic role in the sentence. These syntactic categories cover a wide range of linguistic functions, including nouns, verbs, adjectives, adverbs, pronouns, conjunctions, prepositions, etc., that form the grammatical structure of sentences. They represent various syntactic categories for words and phrases in a sentence. Such a detailed classification helps to understand the syntactic structure of sentences. Here are some commonly used POS tags:

[1] https://nlp.stanford.edu/software/lex-parser.shtml

| **POS Tag** | **Description** |
|---|---|
| CC | Coordinating conjunction |
| CD | Cardinal number |
| DT | Determiner |
| EX | Existential there |
| FW | Foreign word |
| IN | Preposition or subordinating conjunction |
| JJ | Adjective |
| JJR | Adjective, comparative |
| JJS | Adjective, superlative |
| LS | List item marker |
| MD | Modal |
| NN | Noun, singular or mass |
| NNS | Noun, plural |
| NNP | Proper noun, singular |
| NNPS | Proper noun, plural |
| PDT | Predeterminer |
| POS | Possessive ending |
| PRP | Personal pronoun |
| PRP$ | Possessive pronoun |
| RB | Adverb |
| RBR | Adverb, comparative |
| RBS | Adverb, superlative |
| RP | Particle |
| SYM | Symbol |
| TO | to |
| UH | Interjection |
| VB | Verb, base form |
| VBD | Verb, past tense |
| VBG | Verb, gerund or present participle |
| VBN | Verb, past participle |
| VBP | Verb, non-3rd person singular present |
| VBZ | Verb, 3rd person singular present |
| WDT | Wh-determiner |
| WP | Wh-pronoun |
| WP$ | Possessive wh-pronoun |
| WRB | Wh-adverb |

Table 5.1: Syntax notations in the syntax tree

**Example 3**
According to the description of instruction ADC (immediate), *"Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result."* [4]. Figure 5.1 shows an example of a syntax tree of the description of ADC (immediate) instruction, update flags section, *"It can optionally update the condition flags based on the result."*

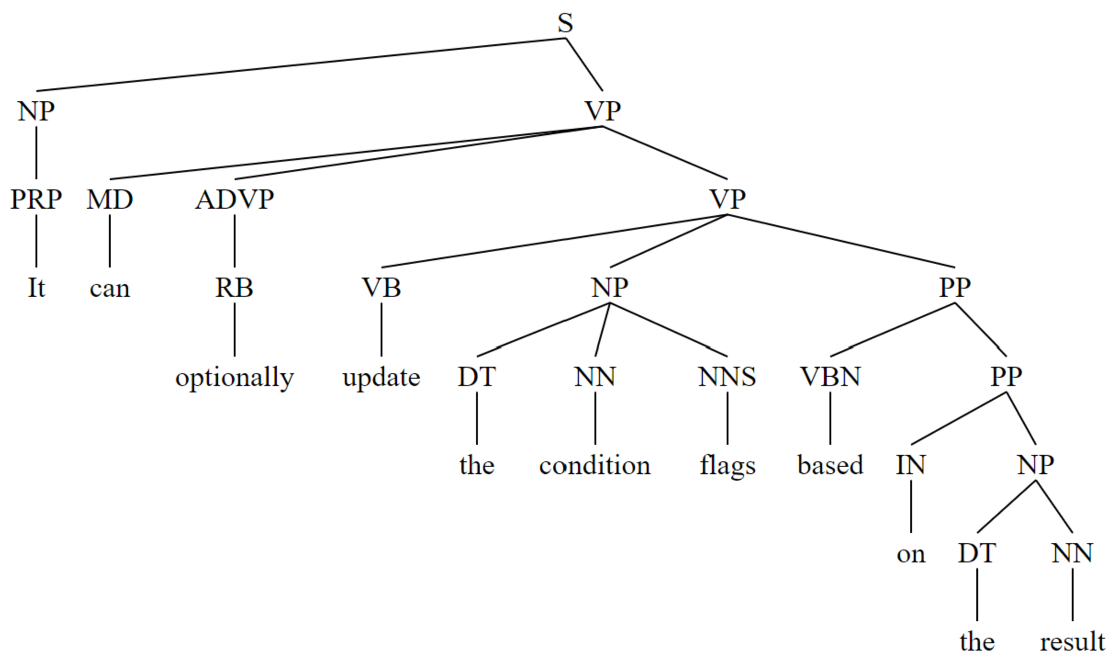


Figure 5.1: Example of syntax tree of the description of ADC (immediate) instruction

## 5.2 Syntax parsing of pseudo-code

In this section, the thesis introduces abstract syntax parsing. The thesis used ANTLR (ANother Tool for Language Recognition) [2] to generate a parser for the pseudo-code of the instruction.

ANTLR (ANother Tool for Language Recognition) is a robust parser generator used to create parsers, interpreters, compilers, and other language processing tools. It is widely used in computer science, especially in the field of language processing, to automatically generate parsers for various programming languages and domain-specific languages. ANTLR takes a formal language grammar as input and generates a parser for it in a

[2]https://www.antlr.org/

variety of programming languages including Java, Python, C, and others. The generated parser can then be used to parse and process input text that conforms to the grammar rules defined in the input. Among the many features ANTLR offers are support for left and right recursive grammars, lexer modes, semantic predicates, and error recovery mechanisms. It is highly customizable and allows developers to define complex grammars for a variety of languages. ANTLR is commonly used in a variety of applications, including building programming language compilers, parsing configuration files, processing domain-specific languages, and a variety of other tasks that require analysis and understanding of structured text input. It is an open-source tool that is actively maintained and improved by the community.

The thesis has created an abstract syntax tree (AST) for pseudo-code for instruction presented in the ARM architecture document. The construction of the AST requires an iterative and rigorous methodology, wherein each pseudo-code representation is methodically parsed and translated into the corresponding abstract syntax tree nodes.

First, the thesis identifies the basic components required of the pseudo-code of the instruction. According to the ARM architecture documentation [4], these include:

- Data types
  - General data type rules
  - Bitstrings
  - Integers
  - Reals
  - Booleans
  - Enumerations
  - Lists
  - Arrays
- Expressions:
  - General expression syntax
  - Operators and functions - polymorphism and prototypes
  - Precedence rules
- Operators and built-in functions
  - Operations on generic types
  - Operations on Booleans and prototypes
  - Bitstring manipulation

    - Arithmetic

- Statements and program structure
    - Simple statements
    - Compound statements
        * if ... then ... else ...

```
if <boolean_expression> then
<statement 1>
<statement 2>
...
<statement n>
elsif <boolean_expression> then
<statement a>
<statement b>
...
<statement z>
else
<statement A>
<statement B>
...
<statement Z>
```

        * repeat ... until ...

```
repeat
<statement 1>
<statement 2>
...
<statement n>
until <boolean_expression>;
```

        * while ... do

```
while <boolean_expression> do
<statement 1>
<statement 2>
...
<statement n>
```

        * for ...

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
<statement 1>
```

```
<statement 2>
...
<statement n>
```

∗ case . . . of . . .

```
case <expression> of
when <constant values>
<statement 1>
<statement 2>
...
<statement n>
... more "when" groups ...
otherwise
<statement A>
<statement B>
...
<statement Z>
```

– Comments

In order to facilitate the construction of the abstract syntax tree, the braces { } have been added to compound statements such as if, for and while statements to mark statements belonging to the same conditional or repeating statement.

The thesis manually deduced and proposed a context-free grammar including 48 rules for parsing pseudo-code to abstract syntax trees as Appendix A.

**Example 4**

Figure 5.2 shows an example of the syntax tree of pseudo-code of ADC (immediate) instruction (execution section): statement

```
(result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
```

```
if ConditionPassed() then
{
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    if d == 15 then           // Can only occur for ARM encoding
    {
         ALUWritePC(result); // setflags is always FALSE here
     }
    else
    {
         R[d] = result;
         if setflags then
         {
             APSR.N = result<31>;
             APSR.Z = IsZeroBit(result);
             APSR.C = carry;
             APSR.V = overflow;
         }
     }
 }
```

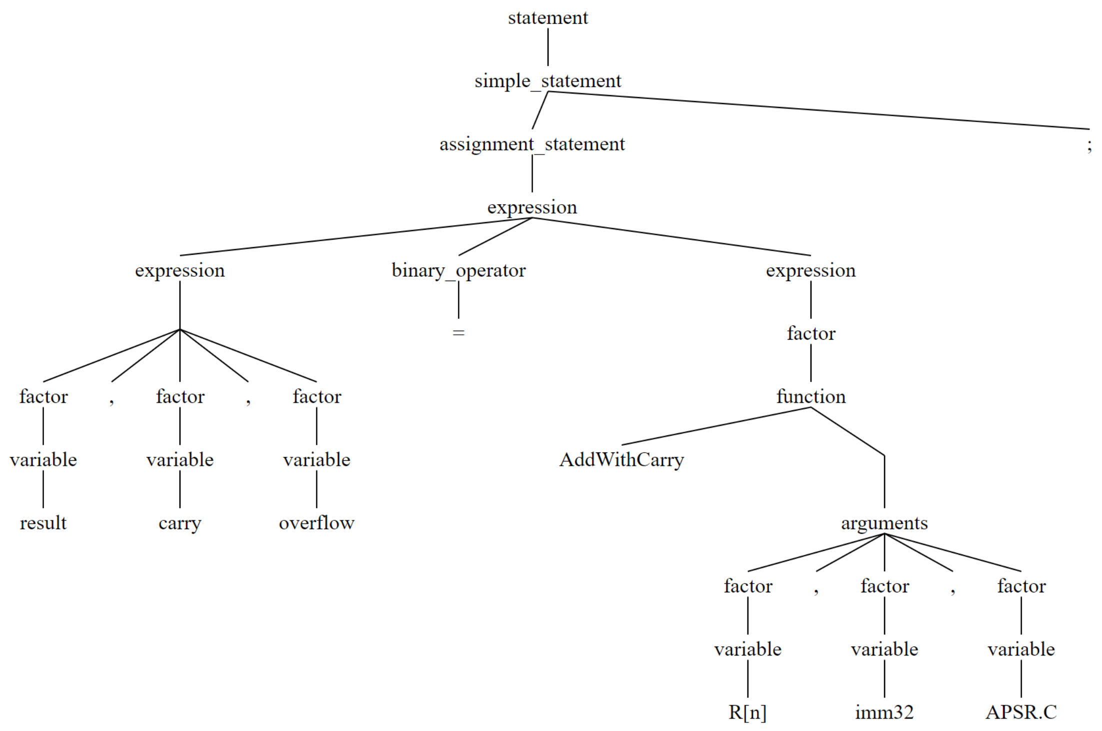


Figure 5.2: Example of the syntax tree of pseudo-code of ADC (immediate) instruction [4] (execution section)

# Chapter 6

# Automatic section extraction

In this chapter, the thesis describes in detail how sections of natural language description and pseudo-code are automatically extracted. The thesis needs to identify the features of each section to extract sections automatically.

## 6.1 Section definition

The groups have different structures for natural language description and the structure of pseudo-code. Instructions belonging to the same group often have similar descriptions and pseudo-code structures. As a result, the thesis divided the instructions into distinct groups to handle each group. The thesis selects the load/store instructions group, load/-store multiple instructions, and all sub-groups in the data processing instructions group to implement.

Figure 6.1 shows the steps for automatically extracting sections.

According to ARM documentation [4], in the data processing instructions group there are many subgroups. They are: ['Standard data-processing', 'Shift instructions', 'Multiply instructions', 'Unsigned multiply instructions', 'Saturating instructions', 'Saturating addition and subtraction instructions', 'Packing and unpacking instructions', 'Parallel addition and subtraction instructions', 'Divide instructions', 'Miscellaneous data-processing instructions'].

According to our preliminary survey, instructions in these same subgroups usually have the same structure. However, to be sure, we will analyze each instruction subgroup to find the general structure of an instruction subgroup. In an instruction subgroup, there is more than one common structure. Figure 6.2 shows the subgroups and sections of instructions.

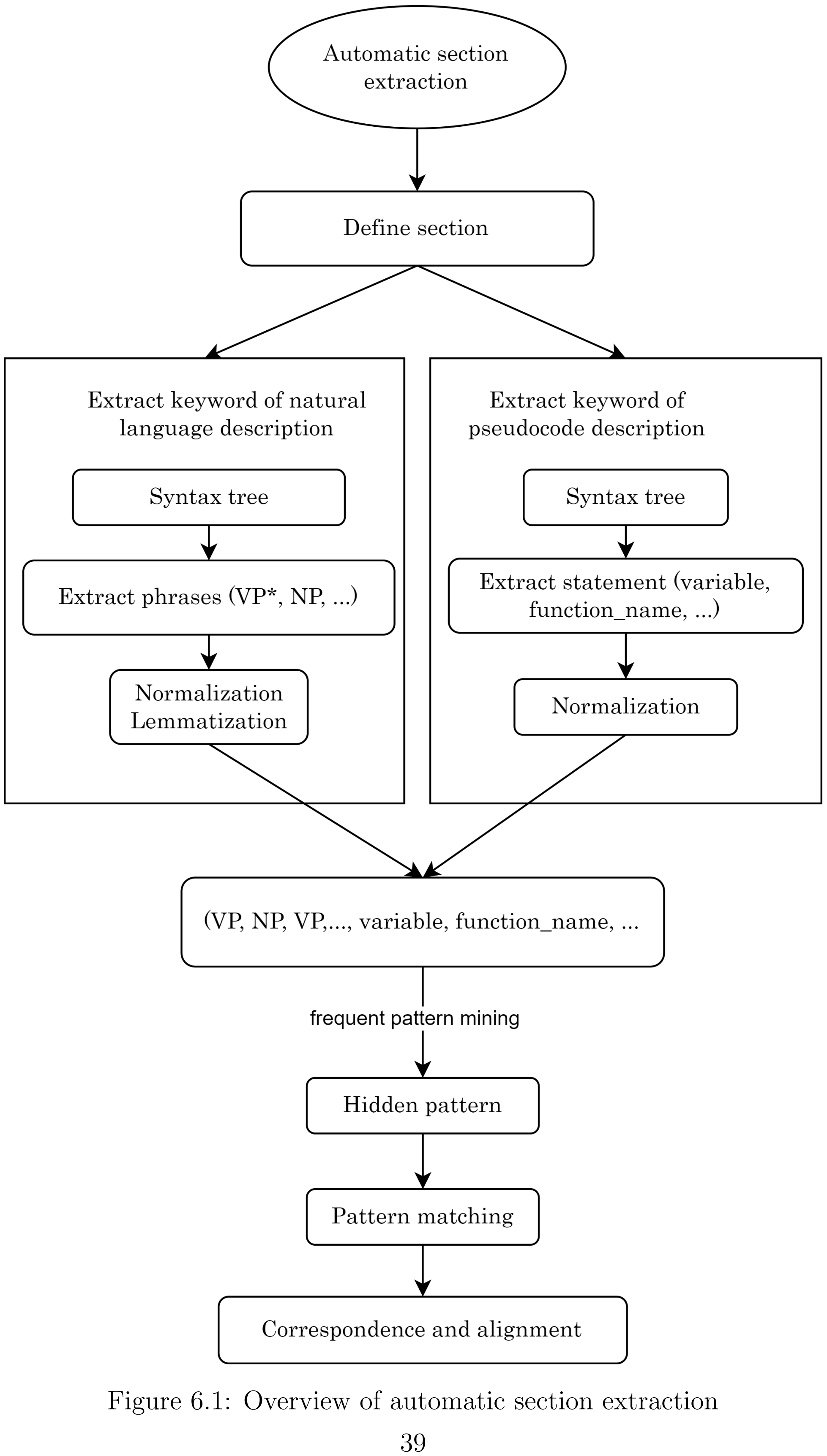

Automatic section extraction
Define section
Extract keyword of natural language description
Syntax tree
Extract phrases (VP*, NP, ...)
Normalization
Lemmatization
Extract keyword of pseudocode description
Syntax tree
Extract statement (variable, function_name, ...)
Normalization
(VP, NP, VP,..., variable, function_name, ...
frequent pattern mining
Hidden pattern
Pattern matching
Correspondence and alignment

Figure 6.1: Overview of automatic section extraction

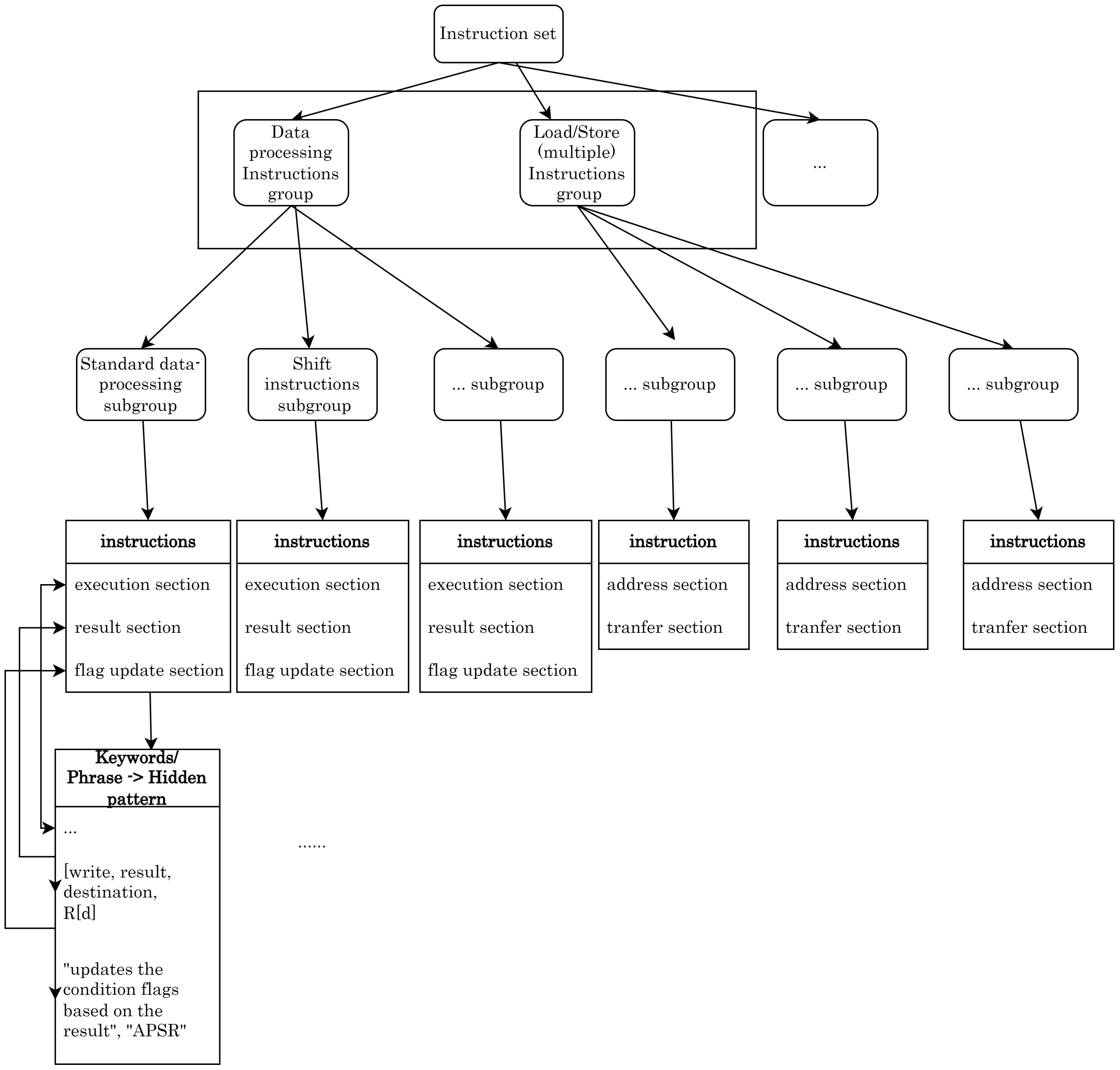


Figure 6.2: Cluster instruction group

## Section definition for data processing groups

As the thesis presented in chapter 5, each instruction group will have different functions, so instructions belonging to different groups will be able to contain different sections. Therefore, the thesis needs to define which sections of each instruction group need to be extracted.

Based on the observation, the thesis defines the 3 main sections for data processing: Execution Section, result section, and flag update section.

**Definition 6.1.1.** Execution Section: This section contains the actual pseudo-code or assembly code that describes the operation performed by the instruction.

**Definition 6.1.2.** Result Section: This section describes the result of the data processing operation and how it is stored or updated in the destination register.

**Definition 6.1.3.** Flag Update Section: This section indicates whether the instruction updates the condition flags (such as N, Z, C, V) in the APSR, CPSR, or PSTATE based on the result of the operation.

**Example 5**

Structure of pseudo-code with 3 sections:

```
if ConditionPassed() then
{
    $ Execution Section

    $ Result Section

    $ Flag Update Section
}
```

**Example 6**

The sections of the natural language description and the pseudo-code of the ADC (immediate) instruction [4] are as follows:

**Execution Section**

The natural language description: *"Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value,"*

The pseudo-code:

```
if ConditionPassed() then
{
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
}
```

### Result section

The natural language description: *"and writes the result to the destination register."*

The pseudo-code:

```
if d == 15 then           // Can only occur for ARM encoding
{
     ALUWritePC(result); // setflags is always FALSE here
 }
else
{
     R[d] = result;
 }
```

### Flag update section

The natural language description: *"It can optionally update the condition flags based on the result."*

The pseudo-code:

```
if setflags then
{
     APSR.N = result<31>;
     APSR.Z = IsZeroBit(result);
     APSR.C = carry;
     APSR.V = overflow;
}
```

### Section definition for load/store instruction groups and Load/store multiple instructions

Based on the observation, the thesis defines the 2 main sections for the load/store instructions group: Addressing Section, transfer section.

**Definition 6.1.4.** Address Section: The memory address is calculated based on the addressing mode used in the instruction.

**Definition 6.1.5.** Transfer Section: These instructions load or store the value of a single register from or to memory or these instructions load or store any subset of the general-purpose registers from or to memory.

### Strategies for extracting sections

The idea of extracting sections of the thesis is to extract the most identifiable sections first, and the complex and ambiguous sections later. Figure 6.3 shows the order of extraction of sections

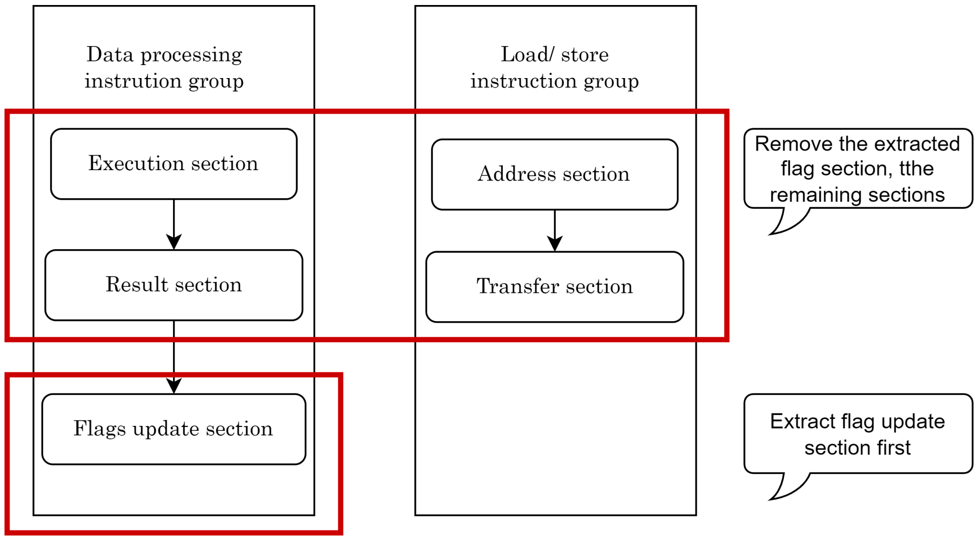

Data processing instrution group
Execution section
Result section
Flags update section
Load/ store instruction group
Address section
Transfer section
Remove the extracted flag section, tthe remaining sections
Extract flag update section first

Figure 6.3: Order of extraction of sections

#### Data processing instructions group

For the data processing instructions group, the first step in the thesis extracts the main operation and the flags update section. After extracting the main operation section, our next goal is to extract the executable code section and the result section. Here is an overview of the steps to automatically extract sections.

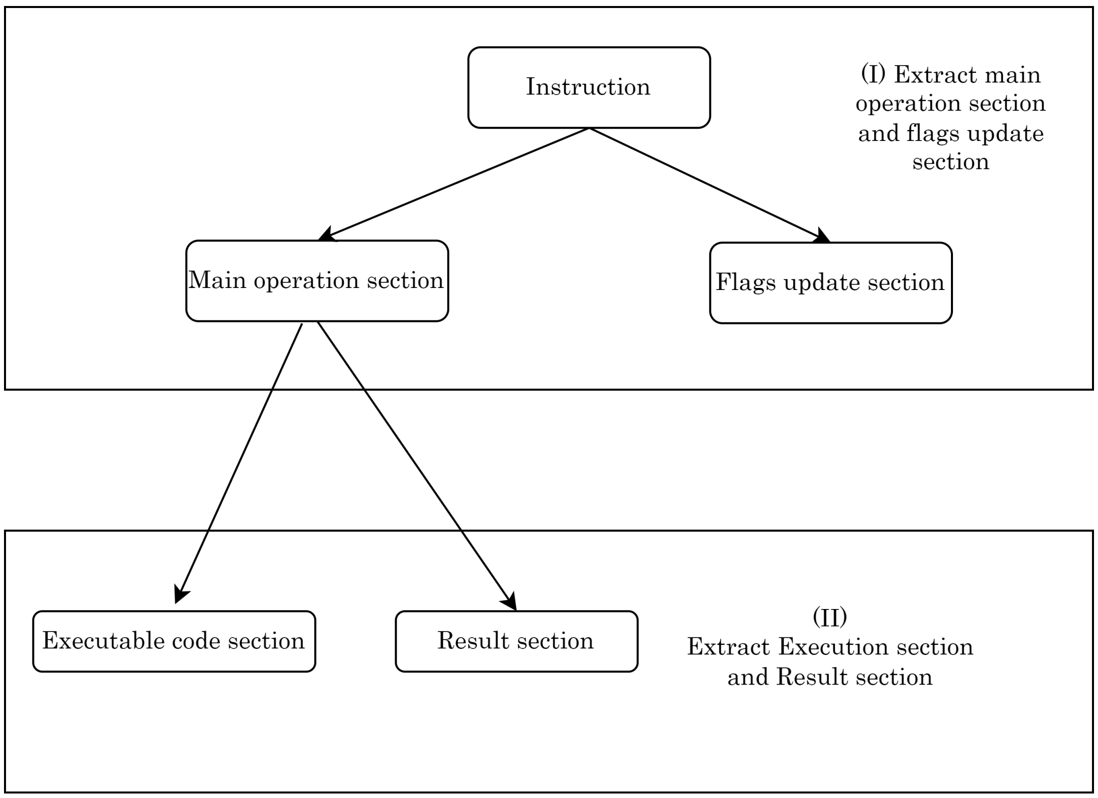

Instruction
(I) Extract main operation section and flags update section
Main operation section
Flags update section
Executable code section
Result section
(II) Extract Execution section and Result section

Figure 6.4: Illustrate the order in which sections are extracted of data processing instructions groups

1. **Extract flag update section**: Flag update section is extracted first
2. **Extract main operation section**: After extracting the flag update section, the remaining clauses of the description will belong to the main operation section. Similarly, the remaining statements and block statements of pseudo-code will also belong to the main operation section.
   (a) **Extract result section**: After extracting two separate sections, the operation section and the flags section, the thesis continues to extract two subsections of the operation section, namely the execution section and the result section from the operation section. Similar to the method of extracting the flag update section and main operation section, the thesis identifies keywords and key phrases of the result section.
   (b) **Extract execution section**: After extracting the result section, the remaining clauses of the description will belong to the execution section. Similarly, the remaining statements and block statements of pseudo-code will also belong to the execution section.

### Load/store (multiple) instructions group

Similarly, For the Load/store (multiple) instructions group, the first step in the thesis extracts the addressing section. After extracting the addressing section, our next goal is to extract the transfer section.

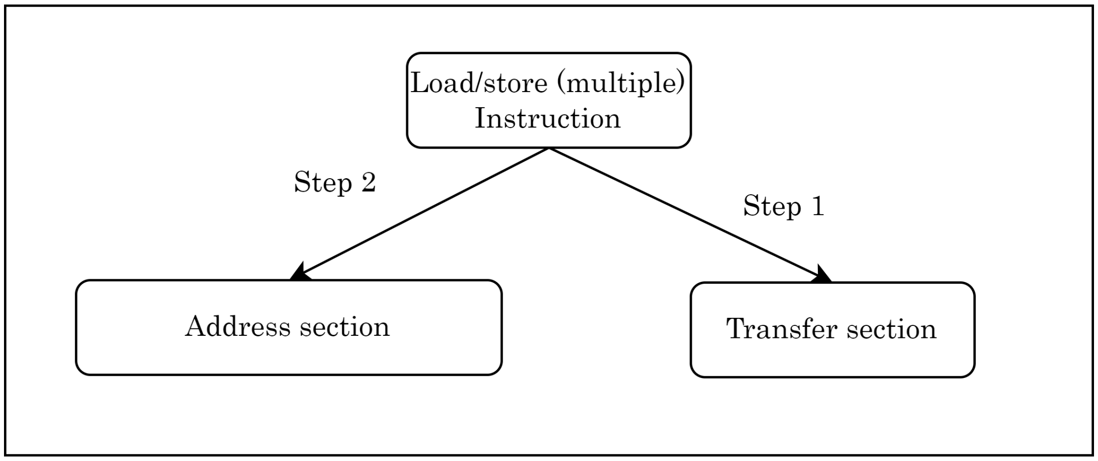


Figure 6.5: Illustrate the order in which sections are extracted of Load/store (multiple) instructions group

In the next section, the thesis focuses on how to identify keywords, phrases, and hidden patterns for the data processing instructions group. The approach to identify keywords and phrases for load/store (multiple) instructions group is similar.

## 6.2 Instruction analysis

### 6.2.1 Natural language analysis

In this section, the thesis presents the steps to normalize the natural language description of the instruction to explore the data and find the feature of each section.

#### Extract simple clauses

The natural language description of the instruction consists of many sentences, in the sentence there are many clauses describing the sequence of operations of the instruction. Therefore, the thesis needs to separate the description into separate clauses. To do this, the thesis uses the parse tree that was built in chapter 5. Besides, the thesis also uses the spaCy [1] library to partially support it. However, the Spacy library only supports splitting paragraphs into sentences. A simple sentence consists of only one clause. A compound

[1]https://spacy.io/

sentence consists of two or more independent clauses. The thesis uses a parse tree to separate sentences into clauses, "sub-sentence". We take the highest nodes of the parse tree labeled VP. If that sub-tree has more than one VP of the same level, we continue to split that VP subtree into smaller subtrees.

**Example 7**

Instruction ADC (immediate) has the natural language description, *"Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result."* [4]

Split this description into:
*"['Add with Carry ( immediate ) adds an immediate value and the Carry flag value to a register value, and .',*
*'writes the result to the destination register',*
*'It can optionally update the condition flags based on the result .']"*

**Lemmatization**

Lemmatization is the analysis of inflected forms of a word, grouped together by the word's lemma, to determine its lemma in computational linguistics. It links words with similar meanings and is similar to stemming but requires correct identification of the intended speech and meaning in a sentence and the larger context.

**Remove the redundant explanations in natural language**

First, we remove the redundant explanations in natural language. We speculate that natural languages containing many clauses often contain the most redundant information (the clause that does not have a corresponding in the pseudo-code). Therefore, for each group, we select instructions whose natural language contains more than 10 clauses to test. Indeed, these introductions all contain redundant information. Redundant clauses often begin with phrases like: "The field descriptions for", "It has encodings from the following instruction sets", or just contain "for more information". The propositions that follow these redundant clauses are also redundant clauses. For example, the natural language of the ADC, ADCS (immediate) instruction[16].

**Identify keywords and phrase of flag update section**

Not all instructions contain the update flags section. In case the description of the instruction contains the update flags section, the thesis defines whether a flag is changed or not. A flag that is changed is called a "modified flags" case. A flag that is not changed is called the "unmodified flags" case. For the "unmodified flags" case, the pseudo-code of the instruction may not have the flag update section. However, there are also cases where the description of instruction does not contain the update flags section, but the pseudo-code of instruction still contains the update flag section, for example, instruction set ADD (SP plus immediate) [4]. Based on observation, the flags section of the description can contain the keyword 'flag'. The thesis extracted the keyword "flag" from the description, then used "Cosine similarity measure" to determine "update flags" and "not update flags". There are two common phrases describing update flags and not update flags "do not affect the condition flags" and "updates the condition flags based on the result". They also often contain sentence patterns such as: "The Q flag is set if the operation saturates", "It updates the condition flags based on the result", "It sets the APSR.GE bits according to the results of the additions" and "The condition flags are not affected".

The table 6.1 shows some common expressions for flags update section:

| Type | Common expression |
|---|---|
| update flag | The Q flag is set if the operation saturates |
|  | It updates the condition flags based on the result |
|  | It sets the APSR.GE bits according to the results of the additions |
| not affect | The condition flags are not affected |

Table 6.1: Common expressions for flags update section

The thesis uses the **TF-IDF** method to convert clauses into vectors. Then use the **Cosine similarity measure** which is mentioned in chapter 3 to determine which clauses belong to the flag update section.

### 6.2.2 Pseudo-code analysis

For example, the multiple instructions groups of the data processing group usually have the form:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand* = ***
    addend   = ***
    result = ***
    R[d] = ***
    if setflags then
        APSR.* = ***
```

#### Remove the default declarations in pseudo-code

First, we remove the default declarations in pseudo-code as "if ConditionPassed()" then and "EncodingSpecificOperations();". From the parse tree of pseudo-code, we remove the subtree whose subtree's root is "function" and the node is "ConditionPassed" and "EncodingSpecificOperations"

**Identify keywords and statement of flag update section**

The APSR (Application Program Status Register) in the ARM architecture indeed holds the flags (version 6, 7) (and alternatively called PTSATE in version 8.9). From this, we deduce that the flag update section of pseudo-code contains the featured keyword: "PSTATE", "APSR", "setflags" which contains syntax patterns such as the statement "APSR.N = * ", "PSTATE.N = * " or condition statement "setflags".

The thesis extracts the keyword "PSTATE", "APSR" and "setflags" from pseudo-code which contains syntax patterns such as the statement "APSR.N = * ", "PSTATE.N = * " or condition statement "setflags". The thesis uses the parse tree built in the chapter 5 and the DFS/BFS algorithm to identify subtree which has node simple statements and variable is "APSR.*" or "PSATE.*" (to the left of the assignment statement) or condition statements containing the keyword "setflags" .

**Identify keywords and phases for other section**

After extracting the flag update section, we remove this section to simplify extracting other sections.

As we mentioned, for natural language description, we take the highest nodes of the parse tree labeled VP. If that sub-tree has more than one VP of the same level, we continue to split that VP subtree into smaller subtrees. Then we extract all lowest level VB*, and NP. For pseudo-code description, we extract all lowset nodes of the parse tree labeled level variable, and function name. We combine these keywords into an array. Then we use some **frequent pattern mining** algorithms to identify sections. For example a hidden pattern: [(write, destination register), (R[d])]. This hidden pattern belongs result section. The result section of the description usually contains the keyword 'write' and 'destination register'. The thesis also uses the **TF-IDF** method to convert clauses into vectors. Then use the **Cosine similarity measure** which is mentioned in chapter 3 to determine which clauses belong to the result section. Besides, the destination register, where the result is located is usually R[d] which contains syntax patterns such as the statement "R[d]* = * ". From this, we deduce that the result section of pseudo-code contains the featured keyword: "R[d]" which contains syntax patterns such as the statement "R[d]* = * ". The thesis uses the parse tree built in the chapter 5 to identify subtrees which has node simple statements and variable is "R[d].*"

# Chapter 7

# Automatically extracting the correspondence

In this chapter, the thesis presents a method to identify the correspondence between natural language description and pseudo-code and some manual testing strategies. The thesis also gives a comprehensive example to illustrate the steps of section extraction and corresponding extraction.

## 7.1 Identify correspondence between the natural language description and pseudo-code

### Correspondence

This section of the thesis aims to identify and align the specific sections of the description that correspond to the relevant sections of the pseudo-code, forming a cohesive and unambiguous link between the two representations. The correspondence process is crucial, establishing correspondences between natural language descriptions and pseudo-code representations. It aligns language constructs with code elements, enabling machines to understand the underlying logic and intentions in both forms. A systematic mapping process is used to establish a clear and coherent correspondence between the textual description of instructions and their associated pseudo-code representations. Once the description and pseudo-code sections have been independently identified, the matching process begins, matching each section from the description to its corresponding pseudo-code counterpart. The figure 7.1 shows the correspondence between description and pseudo-code.

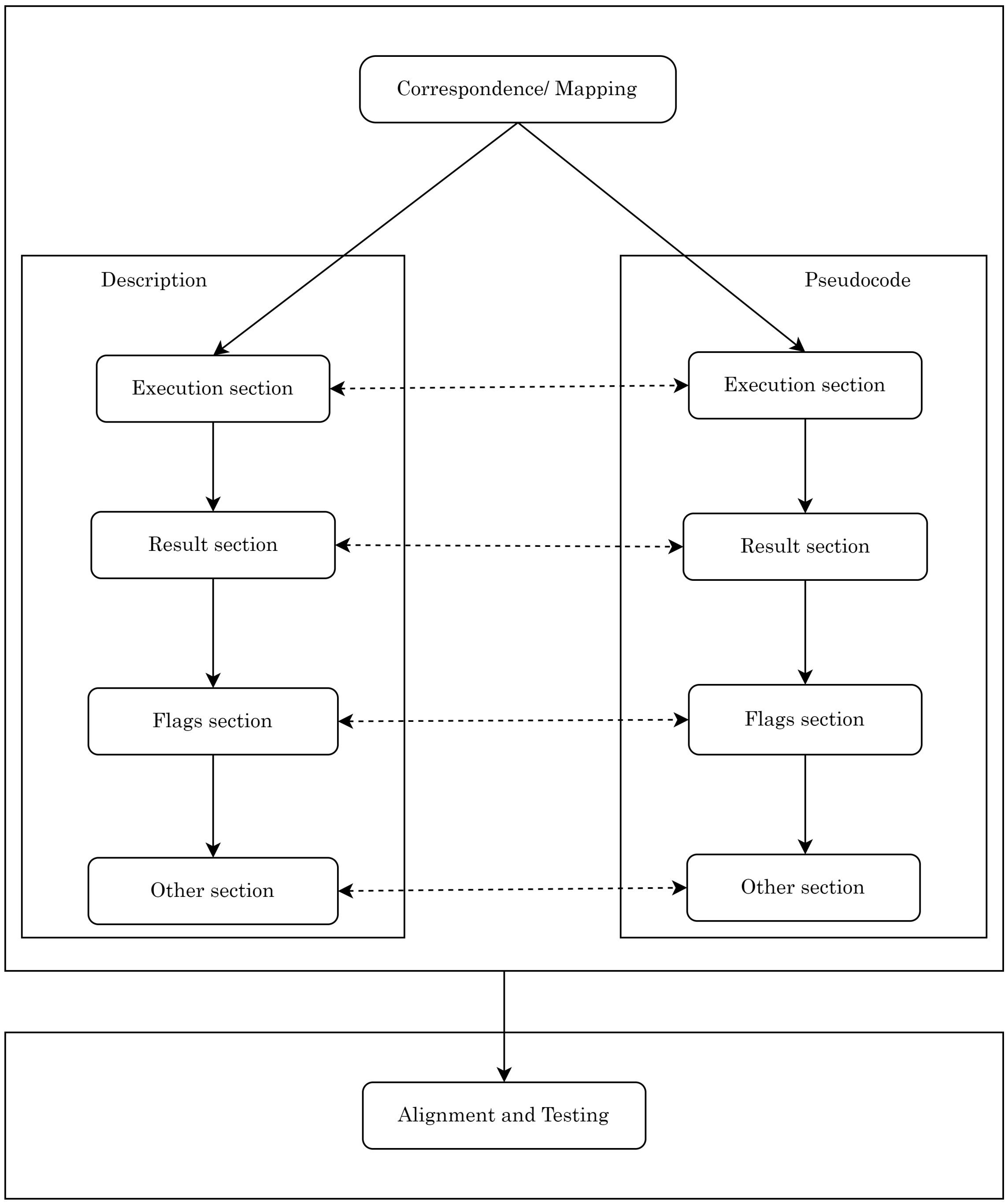


Figure 7.1: Process of correspondence

**Example 8**

### Conformance testing

To accurately assess the precision and effectiveness of the automated extraction process for identifying different sections within the instruction set, the thesis uses a systematic random selection approach, in which a specific number of instructions from different groups are carefully hand-selected for subsequent manual testing. This methodical design ensures a representative sample of the instruction set and thus increases the reliability and validity of the evaluation process.

Some manual testing strategies of the thesis are as follows:

- The thesis chooses instructions that update flags of the natural language description and the pseudo-code are not synchronized. For example, the natural language description does not describe updating flags, but in pseudo-code, the update flags section still appears after performing the "automatic section extraction" step.
- Calculate the average length of phrases in the natural language description as update flags and the average length of statements in the pseudo-code as update flags. To find errors, compare the lengths of the update flags section of instructions to the "average length" accordingly.

## 7.2 Comprehensive example

**Example 9**

Choose instruction ADC (immediate) [4] as a eexample, from which the thesis finds that:

| **Mnemonic** | ADC (immediate) |
|---|---|
| **Description** | Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result. |
| **Pseudo-code** | <pre>if ConditionPassed() then<br>    EncodingSpecificOperations();<br>    (result, carry, overflow) = AddWithCarry(R[n], imm32,<br>    ↪  APSR.C);<br>    if d == 15 then           // Can only occur for ARM<br>    ↪  encoding<br>         ALUWritePC(result); // setflags is always FALSE<br>          ↪  here<br>    else<br>         R[d] = result;<br>         if setflags then<br>             APSR.N = result<31>;<br>             APSR.Z = IsZeroBit(result);<br>             APSR.C = carry;<br>             APSR.V = overflow;</pre> |

Table 7.1: The specification of instruction Add with Carry (immediate) .[4]

1. **Extract main operation section and flag update section**

   (a) **Extract main operation section and flag update section for the natural language description**

      i. Split the description into sentences, and split complex sentences into simple clauses.
      *“['Add with Carry ( immediate ) adds an immediate value and the Carry flag value to a register value, and .', 'writes the result to the destination register', 'It can optionally update the condition flags based on the result .']”*

      ii. Lemmatization
      *“['add with carry ( immediate ) add an immediate value and the carry flag value to a register value, and .', 'write the result to the destination register', 'it can optionally update the condition flag base on the result .']”*

      iii. The thesis uses the TF-IDF method to convert clauses into vectors. Then use the **cosine similarity measure** to determine which clauses belong to the flag update section.
      *“['it can optionally update the condition flag base on the result .']”*

iv. The rest of the sentences are in the main operation section "*['add with carry ( immediate ) add an immediate value and the carry flag value to a register value, and .', 'write the result to the destination register']*"

(b) **Extract main operation section and flag update section for the pseudo-code**

i. Syntax tree of the flag update section of the pseudo-code: From the parse tree of the pseudo-code, get the sub-trees that belong to the flag update section. Based on observation, the thesis extracted sub-trees that have the keyword *APSR* or *PSTATE* in the pseudo-code.

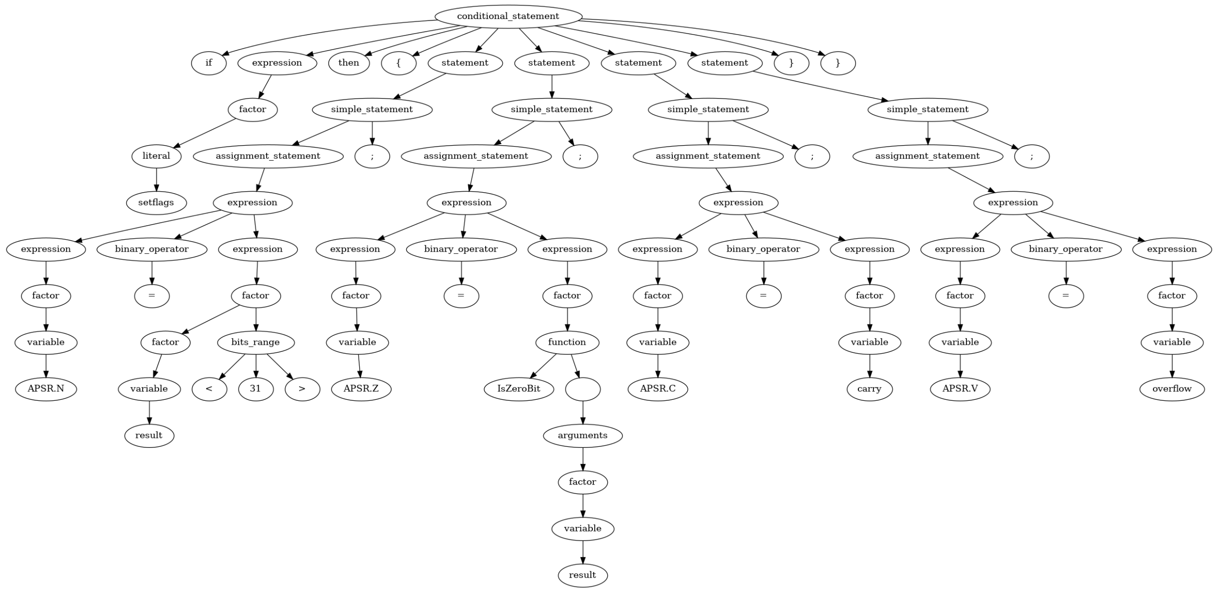


Figure 7.2: Flag update section of pseudo-code of instruction ADC (immediate)

Recovering the pseudo-code from the syntax tree, obtained:

```
if setflags then
{
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
}
```

ii. Syntax tree of the main operation section of the pseudo-code: The rest of the tree are in the main operation section.

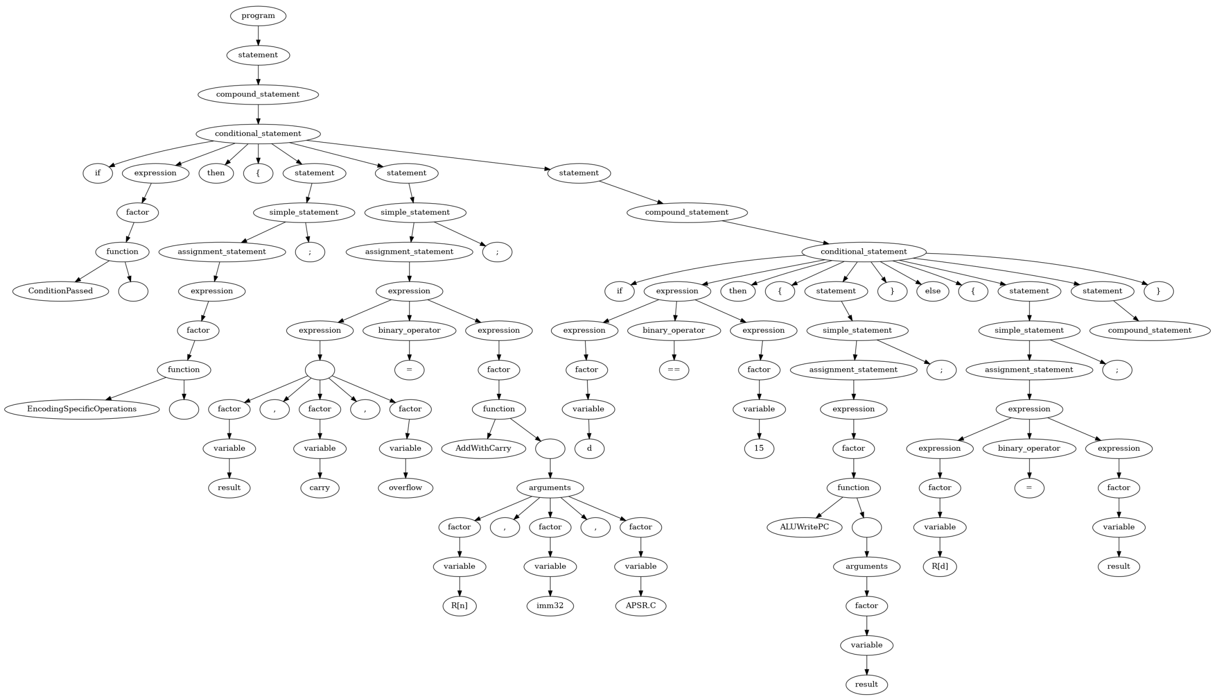


Figure 7.3: Main operation section of pseudo-code of instruction ADC (immediate)

Recovering the pseudo-code from the syntax tree, obtained:

```
if ConditionPassed() then
{
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    if d == 15 then
    {
         ALUWritePC(result);
     }
    else
    {
         R[d] = result;
     }
 }
```

After extracting the main operation section and the flags section of instruction ADC (immediate), the thesis continues to extract the execution section and the result section from the main operation section.

2. **Extract execution section and result section**

   (a) **Extract execution section and result section for the natural language description**

i. Extract result section
"*['write the result to the destination register']*"

ii. Extract execution section
"*['add with carry ( immediate ) add an immediate value and the carry flag value to a register value , and .']*"

(b) **Extract execution section and result section for the pseudo-code**

i. Syntax tree of the result section of the pseudo-code: From the syntax tree of the main operation, get the sub-trees that belong to the result section

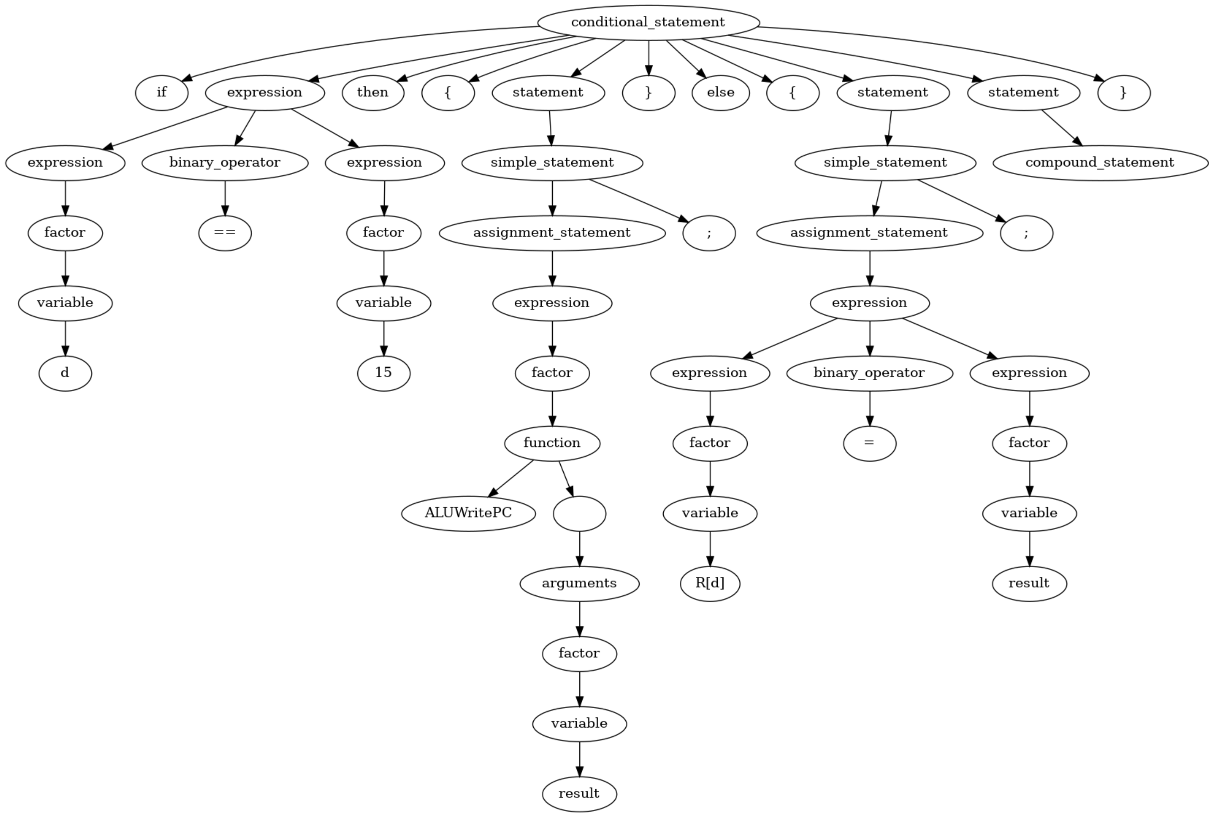


Figure 7.4: Result section of pseudo-code of instruction ADC (immediate)

Recovering the pseudo-code from the syntax tree, obtained:

```
if d == 15 then
{
    ALUWritePC(result);
 }
else
{
    R[d] = result;
 }
```

ii. Syntax tree of the execution section of the pseudo-code: The rest of the tree are in the execution section

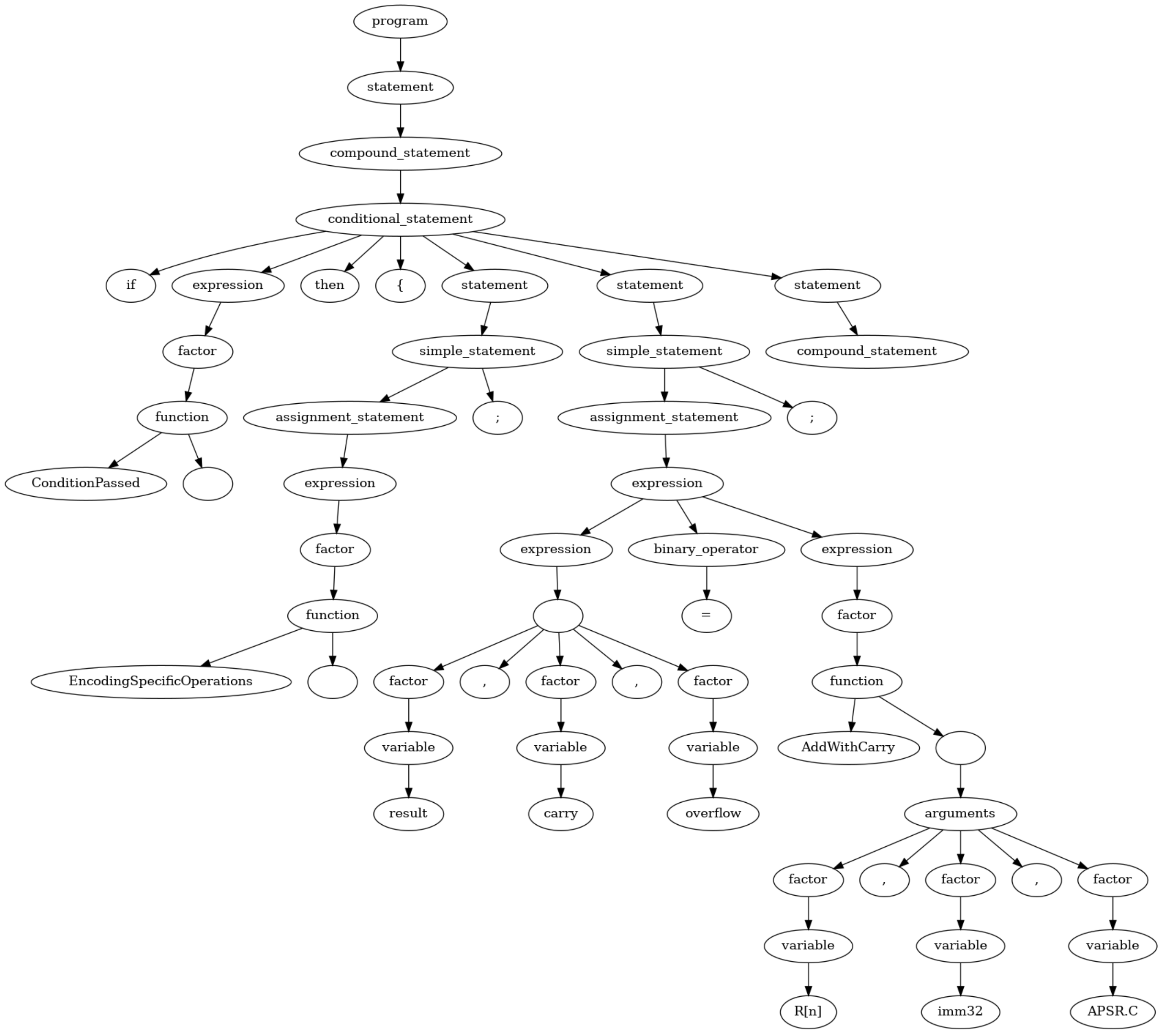
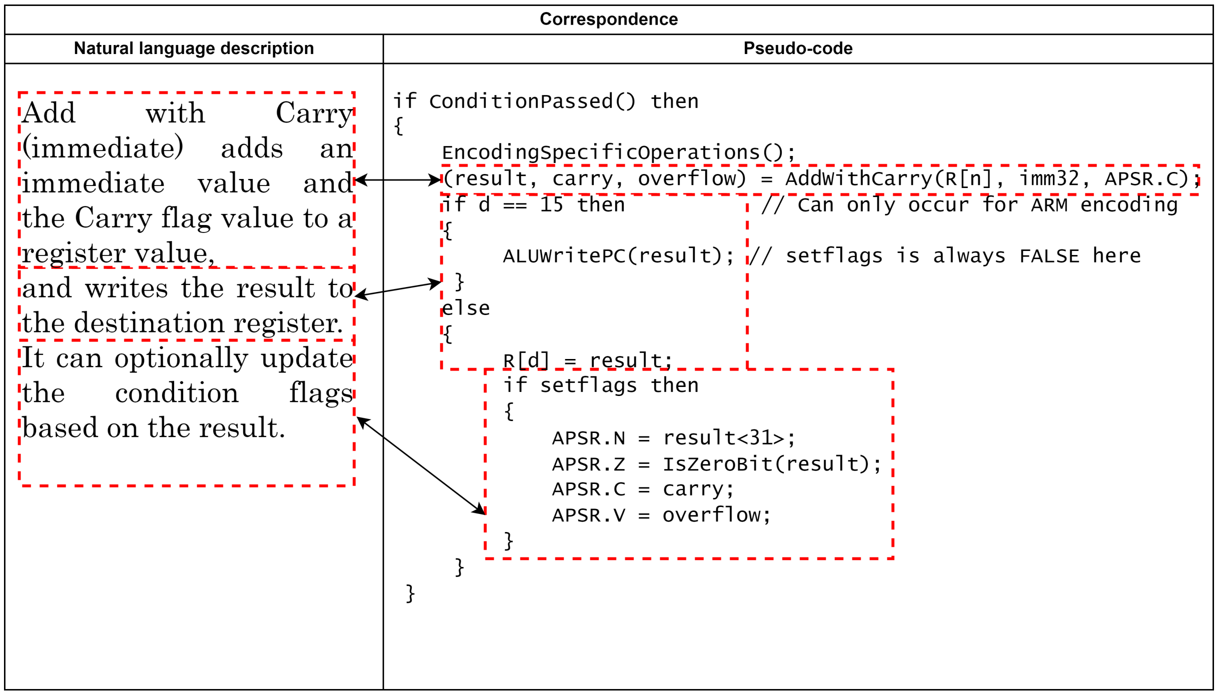


Figure 7.5: Execution section of pseudo-code of instruction ADC (immediate)

Recovering the pseudo-code from the syntax tree, obtained:

```
if ConditionPassed() then
{
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
 }
```

Figure 7.6 illustrates the correspondence between the natural language description and



Add with Carry (immediate) adds an immediate value and the Carry flag value to a register value,
and writes the result to the destination register.
It can optionally update the condition flags based on the result.

```
if ConditionPassed() then
{
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    if d == 15 then           // Can only occur for ARM encoding
    {
        ALUWritePC(result); // setflags is always FALSE here
    }
    else
    {
        R[d] = result;
        if setflags then
        {
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
        }
    }
}
```

Figure 7.6: Example of the correspondence of instruction ADC (immediate)

pseudo-code of instruction ADC (immediate)

# Chapter 8

# Experiments

This chapter presents the results that the thesis has experimented. Thesis experimented with the introduced method for three ARM Cortex processors.

## 8.1 Results

### Collected instructions and Selected instructions

The thesis experimented with the introduced method with the following arm architectures: ARMv6-M [17], ARMv7-M [18], ARMv7E-M, ARMv7-A [4], ARMv7-R [4], ARMv8-M [19], ARMv8-A A32/T32 instruction set [16], ARMv8.2-A A32/T32 instruction set [20], ARMv8-R [21], ARMv9-A A32/T32 instruction set [22], ARMv9.2-A A32/T32 instruction set [7].

Data is collected from the following groups: Branch instructions, Coprocessor instructions, Data processing, Exception-generating and exception-handling instructions, Load-/store instructions, Load/store multiple instructions, Miscellaneous instructions, Status register access instructions, excluding other instructions. However, the thesis selects the load/store instructions group, load/store multiple instructions, and all sub-groups of the data processing instructions group to implement.

Detailed information about the number of instructions that the thesis collects is shown in column **Number of collected instructions**s of table 8.1. The column **Number of selected instructions** represents the number of instructions that the thesis uses for implementation.

- Collected instructions: Total instructions collected from the ARM manuals
- Selected instructions: Total instructions selected from the data processing instructions group and the load/store instructions group.

| ARM architecture | Processor | Number of collected instructions | Number of selected instructions |
|---|---|---|---|
| ARMv6-M | ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1 | 71 | 56 |
| ARMv7-A | ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17 | 263 | 233 |
| ARMv7-R | ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8 | 263 | 233 |
| ARMv7-M | ARM Cortex-M3 | 146 | 115 |
| ARMv7E-M | ARM Cortex-M4, ARM Cortex-M7 | 257 | 184 |
| ARMv8-A | Cortex-A32, ARM Cortex-A34, ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72, ARM Cortex-A73 | 293 | 243 |
| ARMv8.2-A | Cortex-A55, ARM Cortex-A65, ARM Cortex-A75, ARM Cortex-A76, ARM Cortex-A77, ARM Cortex-A78 | 294 | 243 |
| ARMv8-R | ARM Cortex-R52, ARM Cortex-R82 | 294 | 243 |
| ARMv8-M | Cortex-M23, ARM Cortex-M33 | 293 | 213 |
| ARMv9-A | Cortex-A510, A710 and A715 | 301 | 244 |
| ARMv9.2-A | Cortex-A520 and A720 | 301 | 244 |

Table 8.1: Number of collected instructions and selected instructions

## Ignored Cases

The thesis ignores instructions with excessively long descriptions. Instructions with insufficient description but excessive pseudo-code are also ignored. For example, the ignored instruction WFE [4] contains the following data:

| Mnemonic | WFE |
|---|---|
| **Description** | Wait For Event is a hint instruction that permits the processor to enter a low-power state until one of a number of events occurs, including events signaled by executing the SEV instruction on any processor in the multiprocessor system. For more information, see Wait For Event and Send Event on page B1-1199. In an implementation that includes the Virtualization Extensions, if HCR.TWE is set to 1, execution of a WFE instruction in a Non-secure mode other than Hyp mode generates a Hyp Trap exception if, ignoring the value of the HCR.TWE bit, conditions permit the processor to suspend execution. For more information see Trapping use of the WFI and WFE instructions on page B1-1253. |
| **Pseudo-code** | `if ConditionPassed() then`<br>`{`<br>`    EncodingSpecificOperations();`<br>`    if EventRegistered() then`<br>`    {`<br>`        ClearEventRegister();`<br>`    }`<br>`    else`<br>`    {`<br>`        if HaveVirtExt() && !IsSecure() &&`<br>`        ↪  !CurrentModeIsHyp() && HCR.TWE == '1' then`<br>`        {`<br>`            HSRString = Zeros(25);`<br>`            HSRString<0> = '1';`<br>`            WriteHSR('000001', HSRString);`<br>`            TakeHypTrapException();`<br>`        }`<br>`        else`<br>`        {`<br>`            WaitForEvent();`<br>`        }`<br>`    }`<br>`}` |

Table 8.2: Example a ignored instruction

The instructions that are ignored are usually instructions from the ”Miscellaneous instructions” group.

## Inconsistency cases

Inconsistency cases: section of natural language description is not empty while its corresponding pseudo-code section is empty or section of natural language description is empty while its corresponding pseudo-code section is not empty.

Table 8.3 describes the inconsistency of the flag update section. The flag update section of the natural language description and pseudo-code is inconsistent. Check for instructions that fall into this case, for example, the flag update section of the natural language description is not empty while its corresponding pseudo-code update section flag is empty.

The absence of a one-to-one mapping between the description and the pseudo-code poses a dilemma in the mapping process. Based on the testing strategy proposed by the thesis in section 6, there are cases where the correct sections can be extracted but the correspondence cannot be found because the natural language description does not describe how to update flags.

| **Version** | **Inconsistency of natural language flags update section** | **Inconsistency of pseudocode flags update section** |
|---|---|---|
| ARMv6-M | 0 | 8 |
| ARMv7-A | 18 | 2 |
| ARMv7-R | 18 | 2 |
| ARMv7-M | 5 | 2 |
| ARMv7E-M | 14 | 2 |
| ARMv8-A | 27 | 2 |
| ARMv8.2-A | 27 | 2 |
| ARMv8-R | 27 | 2 |
| ARMv8-M | 27 | 2 |
| ARMv9.2-A | 27 | 2 |

Table 8.3: Inconsistency of flags update section

The ADD (SP plus immediate) [4] instruction is an illustrative example.

| **Mnemonic** | ADD (SP plus immediate) |
|---|---|
| **Description** | This instruction adds an immediate value to the SP value, and writes the result to the destination register. |

**Pseudo-code**

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32,
     ↪  '0');
    if d == 15 then           // Can only occur for ARM
     ↪  encoding
        ALUWritePC(result); // setflags is always FALSE
         ↪  here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Table 8.4: Example a inconsistency case

In this instance, the approach presented by the thesis can still extract other portions such as the result section and the execution section.

Another possible confusing case is the MVN (immediate)[4] instruction. In this case, the execution section and the result section both belong to the same clause.

| Mnemonic | MVN (immediate) |
|---|---|
| **Description** | Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value. |
| **Pseudo-code** | <pre>if ConditionPassed() then<br>    EncodingSpecificOperations();<br>    result = NOT(imm32);<br>    if d == 15 then            // Can only occur for ARM<br>    ↪   encoding<br>        ALUWritePC(result); // setflags is always FALSE<br>        ↪   here<br>    else<br>        R[d] = result;<br>        if setflags then<br>            APSR.N = result&lt;31&gt;;<br>            APSR.Z = IsZeroBit(result);<br>            APSR.C = carry;</pre> |

Table 8.5: Example a inconsistency case

These are the instructions that the thesis chooses to test based on the proposed thesis strategy:

- MVN (immediate), MVN (register), MOV (register), MOV (immediate). These instructions have an execution section and a result section of natural language that both belong to the same clause. For, example: MOV (immediate) Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value. [18]

- TST (immediate), TST (register): Does not specify which register the result is written to.

- MLA: In pseudo-code, there is a flags section but in natural language, it is not mentioned.

- SMULL, UMLAL, UMULL, SSAT, SSAT16, USAT, USAT16: No destination register mentioned in natural language

## 8.2 Discussion

When identifying correspondences between the natural language descriptions and the pseudo-code of instruction, Complex problems can be difficult to solve. When the description lacks a direct and unambiguous definition of specific portions or phrases that perfectly match each statement in the pseudo-code, such a difficulty occurs. This disparity might be caused by a variety of variables, including the intrinsic diversity of natural languages, variances in linguistic expressions, and the degree of information provided in the description. Natural language descriptions, as opposed to pseudo-code, can be more flexible, allowing for different modes of expression and conveying information in a more narrative or contextual manner.

### Limitation

1. Ambiguity and Variability: Natural language is not always clear, it also causes ambiguity in understanding. A sentence can have many interpretations. In contrast, many sentences express the same idea by using synonyms. This makes it difficult not only for this research but also for the general field of natural language processing.

2. Incompleteness and redundancy: The natural language descriptions may not always provide all the information commands in pseudo-code. In addition, much of the information in the description is additional descriptive information, which does not describe any commands in the pseudo-code. This can cause no correspondence between the natural language description and the pseudo-code.

# Chapter 9

# Conclusion

In this chapter, the thesis summarizes the main contributions of the thesis, and some future works are also mentioned to suggest some directions to improve and extend our proposed method to other architecture such as x86, MIPS, etc.

## 9.1 Conclusion

In conclusion, the thesis proposed an automatic approach for extracting sections. This thesis also proposed a method to find the correspondence between the natural language description and the pseudo-code of the instruction set. The thesis implements the method on processors that implement the ARM architecture.

The thesis has found correspondence for processors of A-profile, M-profile and R-profile with Arm architectures: ARMv6-M, ARMv7-M, ARMv7E-M, ARMv7-A, ARMv7-R, ARMv8-M, ARMv8-A A32/T32 instruction set, ARMv8.2-A A32/T32 instruction set, ARMv9-A A32/T32 instruction set, ARMv9.2-A A32/T32 instruction set. Among 2475 collected ARM instructions, the thesis focuses on the 2251 instructions of the data processing instructions group and load/store instructions group. Identifying correspondence is an important step in extracting the formal semantics of the ARM architecture. The thesis has contributed the following modules:

1. Thesis has collected and processed data format from pdf file, then extracted natural language description and pseudo-code of ARM instruction set.
2. Thesis has created an abstract syntax tree to analyze the structure and meaning of pseudo-code. The thesis manually deduced and proposed a context-free grammar including 48 rules for parsing pseudo-code to abstract syntax trees.
3. Create a tool to split complex sentences into simple clauses.
4. Automatic section extraction: The thesis has successfully extracted the main operation section and flag update section. From the main operation section, the thesis

continues to extract the result section and execution section, and from the execution section, continues to extract some subsections. Currently, the thesis selects the set of data processing instructions and the set of load/store data instructions for extraction.

5. The thesis has determined the correspondence between the information provided in the natural language description and the operations represented in the pseudo-code of instruction.

### Drawback

For the long and complex instructions, the thesis method cannot be adequately handled. Nevertheless, the number of long and complex instructions represents only a small portion of the entire base instructions.

## 9.2 Future directions

Currently, the thesis has found correspondence for processors of A-profile, M-profile, and R-profile. In the future, we intend to continue this research to increase the capacity of our method and make it able to cover more architectures.

Specifically, some of our future research plans are:

1. We intend to continue researching and improving the method of automatically extracting the formal semantics of the ARM architecture in order to implement other sets of ARM instructions, even though the number of instructions in these groups is not large. From there, we will build dynamic symbolic execution to reconstruct the CFG in order to investigate indirect jumps and understand malware behavior (for the ARM Cortex Architecture).

2. We plan to continue working on automatic extraction for many other architectures and build a generalization system for them.

3. Previously, the correctness of the correspondence results was determined manually. In the future, we will work on a method that can automatically validate the results. In other words, we want to find benchmark and scoring standards, and an automated way to validate the results of a match against those standards.

In conclusion, the future directions of finding promising correspondences between natural language descriptions and pseudo-code

# Bibliography

[1] B. H. Partee, “Formal semantics,” in *The Cambridge Handbook of Formal Semantics* (Cambridge Handbooks in Language and Linguistics), M. Aloni and P. Dekker, Eds., Cambridge Handbooks in Language and Linguistics. Cambridge University Press, 2016, pp. 3–32. DOI: `10.1017/CBO9781139236157.002`.

[2] *What is instruction set architecture (isa)? – arm®*. [Online]. Available: `https://www.arm.com/glossary/isa` (visited on Jul. 8, 2023).

[3] *Intel® 64 and ia-32 architectures software developer manual: Combined volumes 2a, 2b, 2c, and 2d: Instruction set reference, a-z*, Intel Corporation, 2021. [Online]. Available: `https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325383-sdm-vol-2abcd.pdf`.

[4] *Arm architecture reference manual armv7-a and armv7-r edition*, `https://developer.arm.com/documentation/ddi0406/cd/`, Accessed: 2023-07-08.

[5] Arm Developer, “ARM Architecture Reference Manual,” in *ARM Architecture Reference Manual*, Arm Developer, 1996, pp. 1021–1038.

[6] *Armv7-m architecture reference manual*, `https://developer.arm.com/documentation/ddi0403/ee`, Accessed: 2023-07-08.

[7] *Arm architecture reference manual for a-profile architecture*, `https://developer.arm.com/documentation/ddi0487/ja/?lang=en`, Accessed: 2023-07-08.

[8] *Arm cortex-a processor comparison table.* [Online]. Available: `https://developer.arm.com/documentation/102826/latest/` (visited on Jul. 8, 2023).

[9] *Arm cortex-m processor comparison table.* [Online]. Available: `https://developer.arm.com/documentation/102787/latest` (visited on Jul. 8, 2023).

[10] *Arm cortex-r processor comparison table.* [Online]. Available: `https://developer.arm.com/documentation/102788/latest` (visited on Jul. 8, 2023).

[11] Arm Developer, *ARM Compiler armasm User Guide Version 5.06*, Online, 2016. [Online]. Available: `https://developer.arm.com/documentation/dui0473` (visited on Jul. 8, 2023).

[12] N. Indurkhya and F. J. Damerau, *Handbook of Natural Language Processing*, 2nd. Chapman & Hall/CRC, 2010, ISBN: 1420085921, 9781420085921.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006, ISBN: 0321486811.

[14] J. Han and J. Pei, "Pattern-growth methods," in *Frequent Pattern Mining*, C. C. Aggarwal and J. Han, Eds. Springer International Publishing Switzerland, 2014, p. 65. DOI: 10.1007/978-3-319-07821-2_3. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-07821-2_3.pdf.

[15] Arm Developer, "About the Arm architecture," in *Arm Compiler armasm User Guide*, Arm Developer, 2023, p. 71. [Online]. Available: https://www.a.

[16] *Armv8-a reference manual (issue a.k)*, https://developer.arm.com/documentation/ddi0487/ak/?lang=en, Accessed: 2023-07-08.

[17] *Armv6-m architecture reference manual*, https://developer.arm.com/documentation/ddi0419/e/?lang=en, Accessed: 2023-07-08.

[18] *Arm v7-m architecture application level reference manual*, https://developer.arm.com/documentation/ddi0403/ee.

[19] *Armv8-m architecture reference manual*, https://developer.arm.com/documentation/ddi0553/bv/?lang=en, Accessed: 2023-07-08.

[20] *Arm architecture reference manual armv8, for armv8-a architecture profile*, https://developer.arm.com/documentation/ddi0487/b/?lang=en, Accessed: 2023-07-08.

[21] *Arm architecture reference manual supplement - armv8, for the armv8-r aarch32 architecture profile*, https://developer.arm.com/documentation/ddi0568/a/?lang=en, Accessed: 2023-07-08.

[22] *Arm architecture reference manual for a-profile architecture*, https://developer.arm.com/documentation/ddi0487/ha/?lang=en, Accessed: 2023-07-08.

# Appendix A

# Grammar rules

Grammar rules for parsing pseudo-code

```
program: statement+;

statement:
    compound_statement
    | simple_statement
    | special_statement
    ;

simple_statement:
    assignment_statement SEMICOLON
    ;
compound_statement:
    conditional_statement
    | loop_statement
    ;

special_statement:
    KEYWORDS SEMICOLON
     ;

conditional_statement:
          IF (LEFT_PAREN)? expression (LEFT_PAREN)? THEN (LEFT_BRACE)?
           ↪  statement+  (RIGHT_BRACE)? (ELSIF (LEFT_BRACE)?
           ↪  statement+  (RIGHT_BRACE)?)? (RIGHT_BRACE)? (ELSE
           ↪  (LEFT_BRACE)?  statement+  (RIGHT_BRACE)?)?
          | 'case'  expression 'of' (LEFT_BRACE)?  statement+
           ↪  (RIGHT_BRACE)?
          | 'when' expression  (LEFT_BRACE)? statement+ (RIGHT_BRACE)?
          | 'otherwise' (LEFT_BRACE)? statement+ (RIGHT_BRACE)?
          ;

loop_statement:
         FOR expression TO expression (DO)? (LEFT_BRACE)? statement+
          ↪  (RIGHT_BRACE)?
         | 'repeat' (LEFT_BRACE)? statement+ (RIGHT_BRACE)? 'until'
          ↪  simple_statement
         ;

 assignment_statement:
    expression
    ;


expression :
    factor
    | (LEFT_PAREN)? factor(',' factor)* (RIGHT_PAREN)?
    | (LEFT_BRACE)? factor(',' factor)* (RIGHT_BRACE)?
    |  expression binary_operator expression
    |  LEFT_PAREN expression RIGHT_PAREN
    | 'assert' expression
    | '!' (LEFT_PAREN)? expression (RIGHT_PAREN)?
    ;

binary_operator :
    '+' | '-' | '*' | '/'
    | '<=' | '>=' // | '<' | '>'
    | 'IN'
    | '==' | '!='
    | '='
    | '&&' | '||'
    | 'AND' | 'EOR' | 'OR'
    | '>>' | '<<'
    | 'MOD' | 'DIV'
    ;


bits_range:
    '<' (LEFT_PAREN)? INT (('+' | '-'| '*')+ INT)* (RIGHT_PAREN)? (COLON
     ↪  (LEFT_PAREN)? INT (('+' | '-'| '*')+ INT)* (RIGHT_PAREN)? )? '>'
    | '<' (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? (COLON
    ↪  (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? )? '>'
    | '[' (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? (COLON
    ↪  (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? )? ']'
    | '[' (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? (COLON
    ↪  (LEFT_PAREN)? ID (('+' | '-'| '*')+ ID)* (RIGHT_PAREN)? )? ']'
    ;

factor // LHS RHS
    : variable
    | literal
    | function
    | factor bits_range+
    | (DATATYPE)+ factor bits_range*
    | (DATATYPE)* KEYWORDS
    | ID '[' (expression (',' expression)? )? ']'
    | ID '.' ID
    | factor COLON  factor
    | (LEFT_PAREN)? IF expression THEN (LEFT_PAREN)? expression
    ↪  (RIGHT_PAREN)?  (ELSE  (LEFT_PAREN)? expression (RIGHT_PAREN)?)?
    ↪  (RIGHT_PAREN)?
    ;

literal :
    'setflags'
    | 'TRUE'
    | 'FALSE'
    | INT
    | BINARY
    ;

variable:
    ID
    | REGISTER
    | IMMEDIATE
    | FLAGS
    ;

function:
    ID LEFT_PAREN (arguments)? RIGHT_PAREN
    | AARCH LEFT_PAREN (arguments)? RIGHT_PAREN
    | COMMON_FUNCTION LEFT_PAREN (arguments)? RIGHT_PAREN
    | BUILT_IN_FUNCION LEFT_PAREN (arguments)? RIGHT_PAREN
    | MISCELLANEOUS_FUNCTION LEFT_PAREN (arguments)? RIGHT_PAREN
    ;

arguments
    : factor(',' factor)* // Some arguments
    | expression(',' expression)*
    ;
IF : 'if' ;
THEN : 'then' ;
ELSIF : 'elsif';
ELSE : 'else' ;
FOR : 'for' ;
TO : 'to' ;
DO : 'do' ;

LEFT_PAREN : '(' ;
RIGHT_PAREN : ')' ;
LEFT_BRACE : '{' ;
RIGHT_BRACE : '}' ;
SEMICOLON : ';' ;
ANGLE_BRACKETS: '<' | '>';
COLON: ':';

FLAGS :
    'PSTATE.<N,Z,C,V>'
    | 'PSTATE.'ID
    | 'N Flag'
    | 'Z Flag'
    | 'C Flag'
    | 'V Flag'
    | 'APSR.'ID
    ;

REGISTER
    : 'R[d]' // destination
    | 'R[m]' // second
    | 'R[n]' // first
    | 'R[s]'
    | 'R''['ID']'
    ;
```

```
IMMEDIATE
     : 'imm'INT
     ;

KEYWORDS:
    'UNPREDICTABLE'
    | 'UNKNOWN'
    | 'UNDEFINED'
    ;

COMMON_FUNCTION:
    'ConditionPassed'
    | 'EncodingSpecificOperations'
    ;

BUILT_IN_FUNCION:
    'Len'
    | 'Replicate'
    | 'BitCount'
    | 'IsZero'
    | 'IsZeroBit'
    | 'IsOnes'
    | 'IsOnesBit'
    | 'LowestSetBit'
    | 'HighestSetBit'
    | 'CountLeadingZeroBits'
    | 'CountLeadingSignBits'
    | 'ZeroExtend'
    | 'SignExtend'
    | 'Int'
    | 'UInt'
    | 'SInt'
    | 'RoundDown'
    | 'RoundUp'
    | 'RoundTowardsZero'
    | 'Align'
    | 'Max'
    | 'NOT'
    ;

MISCELLANEOUS_FUNCTION:
    'EndOfInstruction'
    | 'Hint_Debug'
    | 'Hint_PreloadData'
    | 'Hint_PreloadDataForWrite'
    | 'Hint_PreloadInstr'
    | 'Hint_Yield'
    | 'IsExternalAbort'
    | 'IsAsyncAbort'
    | 'LSInstructionSyndrome'
    | 'ProcessorID'
    | 'RemapRegsHaveResetValues'
    | 'ResetControlRegisters'
    | 'ThisInstr'
    | 'ThisInstrLength'
    ;

AARCH:
    'AArch32.'ID
    'AArch64.'ID
    ;

DATATYPE:
    'bits''('INT')'
    | 'boolean'
    | 'bit'
    | 'integer'
    | 'MBReqDomain'
    | 'MBReqTypes'
    ;

BINARY:
    '\'' ('0' | '1' | 'x'| ' ')+ '\''
    ;

ID
    : [a-zA-Z_][a-zA-Z0-9_]*
    | [a-zA-Z0-9_]+ ('.' [a-zA-Z0-9_]+)*
    ;

WS
    : [ \t\r\n]+ -> skip
    ;
```

```
INT
    : [0-9]+
    ;

COMMENT
    : '//' ~[\r\n]* -> skip
    ;

BLOCK_COMMENT
    : '/*' .*? '*/' -> skip
;

SKIP_
    :   SPACES -> skip
    ;

fragment SPACES
    :   [ \t]+
    ;
```