Contents

1 Introduction

1.1 Background

The NJOY Nuclear Data Processing System [1] is a software system used for nuclear data management. In particular, it is used to convert Evaluated Nuclear Data Files (ENDF) [2] into different formats, as well as performing operations on the nuclear data.

NJOY is currently being used within the MACRO project [3] at the Division of Applied Nuclear Physics, at the Department of Physics and Astronomy at Uppsala University.

1.2 Problem Description

The NJOY input instructions [4] are complex and hard to read compared to e.g. a high-level programming language. For example, algorithm 1 on the following page is a *short* and *simple* NJOY job which illustrates what the input instructions look like.

Algorithm 1 NJOY Test Problem 14

```
1  acer
2  20 21 0 31 32
3  1 0 1/
4  'proton + 7-n-14 apt la150 njoy99 mcnpx'/
5  725 0./
6  /
7  /
8  acer
9  0 31 33 34 35
10 7 1 2/
11 'proton + 7-n-14 apt la150 njoy99 mcnpx'/
12 viewr
13 33 36/
14 stop
```

Without consulting the documentation, one might guess that line 4 and 11 are some kind of descriptive titles, which is correct. However, it is not obvious that line 2 denotes input and output files (each number indicates a specific file) that the `acer` module will operate on. It is also hard to deduce that the first number on line 5 denotes the material to be processed, and that the second number denotes the desired temperature in kelvin.

The input instructions can be annotated with descriptive comments, but even then, working with a large and complex job easily becomes a daunting and error-prone task.

1.3 Objective

The NJOY input instructions is not an optimal input format. Therefore, the scope of this thesis has been to design and implement a more user friendly, and readable input format. The design of the new input format had to be based on some commonly known existing format that is fitting to the task. The basis could for example be a programming language.

In order to make the new input format useable with NJOY, it has to be translated into the original NJOY input instructions. As such, the scope of this work also included developing an accompanying translator for the new input format.

2 Methodology

2.1 Introduction

The NJOY input instructions had to be understood in order to design the new input format. Each module of the NJOY software system, as described in reference [4], was analyzed separately such that a general structure and common language features could be extracted and used for further analysis.

As stated in reference [5], a translator (*compiler*) is a program that can read a program in one language and translate it into an equivalent program in another language. In the following subsections, principles and techniques for constructing a translator presented in reference [5], is described.

2.2 Designing the New Input Format

2.2.1 Syntax Definition

The syntax definition of the input format was specified in a notation called context-free grammar [6]. A context-free grammar is a convenient, and natural method of specifying the syntax of a programming language. For instance, the assignment (*declaration*) of an identifier can have the form

```
material = 9237
```

which can be expressed in a context-free grammar as the production

```
assignment ::= l_value "=" r_value
```

where `l_value` and `r_value` are other productions expressing the structure of the left and right hand side of the assignment, respectively.

2.3 Building the Translator

In reference [5], the translation process is described as a sequence of phases. Each phase inspects and transforms a representation of the source program to another. Phases such as lexical analysis, syntax analysis, and semantic analysis has been used throughout this work.

The translator, which is supposed to translate the input format into NJOY input instructions, was partly constructed using a lexical-analyzer generator [7] and a parser generator [8].

The translator was written in the Python programming language [9], in a Unix-like environment.

2.3.1 Lexical Analysis

Lexical analysis is the process of dividing the source program into sequences of characters, called tokens [10]. Each token describes a group of characters in the source program as an abstract type.

For example, the identifier `material`, the assignment character, `=`, and the integer `9237` could be represented as tokens of the form

`<IDENTIFIER, material>`,

`<ASSIGNMENT, =>`, and

`<INTEGER, 9237>`

PLY Lex [11] was used to generate a lexical analyzer (*lexer*) for the input format. The method of identifying the tokens was implemented by using the notation of regular expressions [12] in PLY Lex.

2.3.2 Syntax Analysis

Syntax analysis is the process of creating a tree-like representation, an abstract syntax tree, composed of the tokens generated by the lexical analyzer [13]. The syntax tree is used to describe the grammatical structure of the source program.

PLY Yacc [11] was used to generate a syntax analyzer (*parser*) for the grammar definition of the input format. The method of building the syntax tree was implemented by using the facilities provided by the PLY tools.

2.3.3 Semantic Analysis

Semantic analysis is the process of checking the syntax tree for errors that have to do with the *meaning* of the program [14].

For example, according to reference [4], card 1, 2 and 3 in module `acer` must always be defined, and they must be defined in sequential order. The translator should report an error if these rules are violated; such as when card 1 has not been defined or when card 3 has been defined prior to card 2.

Type checking is another important part of the semantic analysis where the translator checks that each operator has valid operands.

For example, the identifier `hk`, in card 3 module `acer`, is used to denote a descriptive character string. According to reference [4], `hk` must be declared as a character string and must not exceed 70 characters in length. The translator should report an error if these rules are violated; such as when `hk` has been declared as an integer, or when the character string contains more than 70 characters.

2.4 Testing

Testing was carried out continuously during the design and implementation of the input format and the translator. The NJOY test problems[1] [1] was used to test the functionality of both the input format and the translator.

The NJOY test problems was manually translated into equivalent NJOY jobs in the new input format, which were run through the translator. The resulting output was compared with the expected output, to verify that the translator was working appropriately.

The Python unit testing framework [9] was utilized to set up the testing environment.

[1]The NJOY Test Problems are test runs which are used to test the functionality of the NJOY software system. See `http://t2.lanl.gov/codes/njoy99/`

3 Implementation

3.1 NJOY Input Format (`NIF`)

The new input format, NJOY Input Format (NIF), is basically the NJOY input instructions which have been annotated with a syntax to make it easier to read and express. NIF has been designed to appear more like a high-level programming language.

3.1.1 Grammar Definition

The proposed NJOY Input Format (`NIF`) is illustrated as a context-free grammar definition in algorithm 2. The structure of the grammar is simple. Just like in reference [4], a NIF program is an ordered sequence of modules. Each module is composed by an ordered sequence of cards. A card is an ordered sequence of value definitions.

Algorithm 2 NJOY Input Format (NIF) Grammar Definition

```
program ::= module_list

module_list ::= module module_list
              | empty
module      ::= MODULE "{" card_list "}"

card_list ::= card card_list
            | empty
card      ::= CARD "{" stmt_list "}"

stmt_list  ::= statement stmt_list
             | empty
statement  ::= expression ";"
expression ::= assignment

assignment ::= l_value_list "=" r_value_list

l_value_list ::= l_value
               | l_value "," l_value_list
r_value_list ::= r_value
               | r_value "," r_value_list

l_value ::= array
          | ident
array   ::= IDENTIFIER "[" INTEGER "]"
ident   ::= IDENTIFIER

r_value ::= FLOAT
          | INTEGER
          | NULL
          | STRING
```

The start symbol is `program`. The capitalized terminals, such as `MODULE` and `CARD`, are token classes specified by the lexer. Special symbols are denoted within double quotes. `empty` denotes the empty string.

An assignment denotes that a left hand side is assigned to hold the values of a right hand side. A left hand side is an ordered list of elements, where the elements can be an array or identifier. A right hand side is an ordered list of elements, where the elements can be a float, integer, null or a string. As such, a value definition is an array or identifier that has been declared to hold the value of either a floating-point number, natural number, empty string or a character string.

As indicated by the grammar, NIF supports multiple assignment. That is, multiple identifiers can be assigned in the same expression. For example, the expression

```
material, temp = 9237, 300.0;
```

denotes that the identifier `material` holds the integer 9237, and the identifier `temp` holds the float 300.0.

Note that the number of elements on the left hand side of an assignment does not have to be equal to the number of elements on the right hand side. According to the grammar, an assignment such as

```
material = 128, 9237;
```

is allowed even though it does not make sense. However, the syntax analysis in the parser enforces that the number of elements on both sides are the same.

3.2 NJOY Input Format Translator (`nifty`)

3.2.1 Structure of the Translator

The translator, NJOY Input Format Translator (`nifty`), was constructed as a set of modules where each module implements a specific phase in the translation process. Five phases have been implemented as part of the translation process and are shown in figure 1.

The first phase is the lexical analysis which is implemented by the lexer module. The second phase, syntax analysis, is implemented by the parser module.

The third phase, implemented by the organizer module, is a special phase where the order of the statements in a card are analyzed and possible rearranged.

The fourth phase is the semantic analysis which is implemented by the module named analyzer. The fifth, and final, phase of the translator is the emitter module which implements a NJOY input instructions generator.

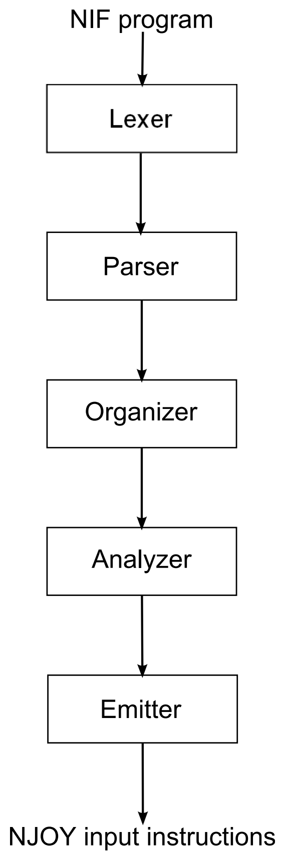

Figure 1: Translation process in nifty

3.2.2 Reserved Keywords

An important design choice is that the translator will enforce the use of reserved keywords to specify NIF programs. It will not only consider card and module names as reserved keywords, but also identifier names. As such, it is not possible to use an identifier name until it has been defined as an identifier in the translator. Similarly, it is not possible to use a card or module name which has not been defined in the translator. This restricts the expressiveness of the input format, but allows detailed analysis of the semantics in the organizer and analyzer modules. As a consequence, it also forces the user to write consistent and readable input files – which has been the objective of this work.

3.2.3 The Modules

Lexer The lexer is responsible for recognizing character patterns and generating the appropriate NIF tokens. As input, the lexer expects a NIF program and will generate a token stream as its output unless the lexer detects a lexical error. If a lexical error is detected, an error message will be reported and the translation process will stop at this phase. The lexer will only recognize card and module names which are specified in reference [4], thus enforcing the use of a specific set of cards and modules as mentioned previously. The lexer also recognizes comments in the input program. The comments will be discarded during the lexical analysis and thus won't be passed on to the next phase in the translation process. Note that the lexer only recognizes real numbers that starts with a digit. Numbers which has a leading dot, such as `.005` or `.5e-2`, which are allowed as input to the NJOY software system are thus not allowed by the lexer.

Parser The parser is responsible for enforcing the structure of the NIF grammar and constructing the syntax tree. As input, the parser expects a stream of tokens generated by the lexer. The parser will produce a syntax tree as its output, which represents the structure of the NIF program. If the parser detects a syntax error, an error message will be reported and the translation process will stop at this phase.

Organizer The organizer analyzes the syntax tree produced by the parser. Its purpose is to rearrange the statements in a card such that they appear in the expected, working order. As such, it should be possible to write a NIF program without having to list the statements in a card in the expected order as indicated by reference [4].

The NJOY modules and the cards within the modules still needs to be given in the correct order though. This is due to the fact that the number of possible NJOY jobs is infinite (all may not be functional in the NJOY software system, though). An infinite number of NJOY jobs can simply be created by just appending another module specification to an existing NJOY job in order to create a new one. Simply stated, the translator can not guess the intention of the job due to the number of possible combinations the modules may be listed in. Hence, the modules must be provided in the expected order by the user. Cards are not arrangeable either, since they also are prone to be repetitive. It is not possible to determine which card should go first from a set of cards (with the same name) which e.g. only contains a descriptive title. The cards must also be provided in the expected order by the user.

Each NJOY module requires its own organizer implementation since each module has its specific set of rules as described in reference [4]. Since the identifier names are hardwired in the translator, the organizer is able to do a detailed analysis of the syntax tree and easily detect if a specific identifier has been defined out of order.

If any statements have been provided out of order in a card, and the organizer is able to arrange the statements, a new syntax tree is returned where the statements have been ordered in the expected sequence. If the organizer somehow fails, it will return the original syntax tree as produced by the parser and pass it on to the next phase in the translation process.

Analyzer The analyzer expects a syntax tree as its input. Like in the organizer phase, the NJOY modules needs to be analyzed separately since each module has its specific set of rules. As such, each module also requires its own analyzer implementation.

The analyzer basically visits every node in the order they appear in the syntax tree and checks if it is the expected one. The analysis can be made very detailed since the translator can, to some extent, predict the next card or identifier due to the ordered nature described in reference [4]. Since the cards and the identifiers have reserved names, the analyzer is able to easily determine whether a card or an identifier is the expected one. Using reserved names also makes type checking easy, since a reserved identifier in a specific card may be associated with a specific type, range, size, length, et cetera.

The analyzer does not alter the syntax tree, it just analyzes it. The input syntax tree will be the output of the analyzer if the syntax tree is semantically correct according to the translator. If the analyzer detects a semantic error in the syntax tree, an error message will be reported and the translation process will stop at this phase.

Emitter The emitter expects a syntax tree as its input and it is responsible for generating NJOY input instructions from the syntax tree. The emitter simply flattens the tree structure and formats the instructions to their corresponding counterparts in the NJOY input instructions format. The emitter returns a string with the resulting NJOY input instructions. Each card in the resulting output has been annotated with a descriptive comment, indicating which card it is, to make it easier to find errors.

4 Results

4.1 NJOY Input Format Examples

The result of the proposed grammar described in section 3.1 on page 9 is best illustrated with examples. Algorithm 3 illustrates NJOY input instructions (slightly modified to make it shorter for illustrational purposes) from NJOY Test Problem 2 [1]. In algorithm 4 on the next page, lines 1 through 9 from algorithm 3 are expressed in NIF.

Algorithm 3 Modified subset of NJOY Test Problem 2

```
 1 moder
 2 20 -21/
 3 reconr
 4 -21 -22/
 5 'pendf tape for pu-238 from endf/b-iv tape 404'/
 6 1050 1/
 7 0.005/
 8 '94-pu-238 from endf/b tape t404'/
 9 0/
10 broadr
11 -21 -22 -23/
12 1050 3 0 1/
13 0.005/
14 300.0 900.0 2100.0/
15 0/
16 stop
```

Algorithm 4 NIF version of Algorithm 3 on the previous page, lines 1 through 9

```
moder {
    card_1 {
        pendf_input = 20;
        pendf_output = -21;
    }
}

reconr {
    card_1 {
        nendf = -21;
        npend = -22;
    }

    card_2 {
        tlabel =  "pendf tape for pu-238 from endf/b
           -iv tape 404";
    }

    card_3 {
        mat = 1050;
        ncards = 1;
    }

    card_4 {
        err = 0.005;
    }

    card_5 {
        cards = "94-pu-238 from endf/b tape t404";
    }
    /* Card 6 not defined since 'ngrid' defaults to
       0 in first card 3. */
    card_3 { mat = 0; } // Terminate reconr.
}
```

Descriptive names for the identifiers on line 3 and 4 have been specified in the translator. The other identifier names has been chosen to reflect the documentation in reference [4] (the identifier names are hardwired in the translator). Line 30 and 31 shows how comments are expressed in NIF. Line 30 illustrates the structure of multiline comments while line 31 illustrates the structure of single line comments.

Algorithm 5 on the following page is a NIF version of the lines 10 through 16 from algorithm 3 on page 17. It shows how arrays are expressed in NIF (lines 24 through 26). The `stop` instruction on line 16 in algorithm 3 on page 17 does not have to be specified in NIF, the translator will automatically append it in the translation process.

When combined, algorithm 4 on the previous page and algorithm 5 on the following page forms the complete NJOY job as listed in algorithm 3 on page 17.

Algorithm 5 NIF version of Algorithm 3 on page 17, lines 10 through 16

```
broadr {
    card_1
    {
        nendf = -21;
        nin = -22;
        nout = -23;
    }

    card_2
    {
        mat1 = 1050;
        ntemp2 = 3;
        istart = 0;
        istrap = 1;
    }

    card_3
    {
        errthn = 0.005;
    }

    card_4
    {
        temp2[0] = 300.0;
        temp2[1] = 900.0;
        temp2[2] = 2100.0;
    }

    /* Terminate execution of broadr with mat1 = 0
       as usual. */
    card_5
    {
        mat1 = 0;
    }
}
```

4.2 NJOY Input Format Translator Implementation

Table 1 shows the implementation status for the NJOY modules. Each column entry indicates the completeness of a translator phase for a given NJOY module.

NJOY module	Lexer	Parser	Organizer	Analyzer	Emitter
`acer`			100%	90%	
`broadr`			100%	90%	
`ccccr`			0%		
`covr`			100%	90%	
`dtfr`			0%		
`errorr`			70%	20%	
`gaminr`			0%		
`gaspr`			0%		
`groupr`			100%	90%	
`heatr`			100%	90%	
`leapr`					
`matxsr`	100%		0%		100%
`mixr`					
`moder`			100%	95%	
`plotr`			100%	90%	
`powr`			0%		
`purr`			0%		
`reconr`			100%	90%	
`resxsr`			0%		
`thermr`			100%	90%	
`unresr`			0%		
`viewr`			100%	10%	

Table 1: Implementation status for the NJOY modules

The completeness of the implementation has been rated in a grading scale with percentage. The grades has been set with respect to whether the functionality of the phases presented in section 3.2.3 on page 13 (also see section 2.3 on page 5) has been fulfilled or not. 100% indicates that the functionality has been finished. 0% indicates that the implementation of the functionality has not been started. The other percentages are rough approximations of how much functionality that has been implemented.

4.3 Testing

All test problems listed in Appendix B on page 33 passed all the phases in the translation process, i.e. they were successfully translated (no lexical, syntax, nor semantic errors were found)[2]. No differences between the expected output and the resulting output were detected for the test problems.

[2]Note that the organizer's ability to arrange statements in the correct order has not been tested for the test problems, since the instructions in the test problems have been provided in the expected order.

5 Discussion

5.1 NJOY Input Format (`NIF`)

The proposed grammar does not differ much from the NJOY input instructions since it basically is an annotated version of them. The NIF grammar could have been expanded to include more complex programming idioms, such as an `if` expression to allow flow control in a NIF program. Although, the structure of NIF was designed to be simple and to closely resemble the original input instructions such that a user does not need to learn a completely new programming language to specify NJOY jobs. Another intention of this design choice is that the NJOY input instructions documented in reference [4] can be used to specify NJOY jobs in NIF.

As indicated by the examples listed in section 4.1 on page 17, a typical NIF program is vertically long compared to the compact notation of the NJOY input instructions. NIF programs can of course be specified in a compact form as well, e.g. on a single line, but this is not the intended usage of NIF. The purpose of NIF is to make NJOY jobs readable. The readability would be limited if the jobs were expressed on a single line.

5.2 NJOY Input Format Translator (`nifty`)

An organizer and analyzer has not been provided for all modules in the NJOY software system due to time constraints of this thesis. As such, the important semantic analysis of the translator is incomplete. However, much of the needed functionality and structure is provided by the existing implementation such that both the organizer and the analyzer should be easy to complete. Even though the organizer and analyzer phase has not been implemented for all NJOY modules in the translator, NIF programs which include these modules can still be translated into functional NJOY input instructions.

The implementation of the analyzer module has been the most time consuming task when designing the translator. It requires detailed analysis of what kind of input the NJOY modules expect and how they operate on it. The documentation in reference [4] was the main resource used while implementing the semantic analysis in the analyzer. It was evident that this was not a sufficient resource for the task at hand. It does not clearly indicate the expected type for all identifiers, nor the expected integer ranges or length of the character strings. In some cases, it has also been hard to deduce which cards that must be supplied by just reading the documentation in reference [4]. To fully check the semantics of a NIF program, the source code for the NJOY software system must be studied in greater detail. The ENDF formats must also be studied in greater detail in order to understand the semantics and what kind of values that the NJOY modules accept.

5.3 Testing

The testing that was conducted within this work is not rigorous enough due to time constraints of this thesis. NJOY is a large and complex program[3] with many possible combinations of input within each NJOY module and card.

The NJOY test problems [1] which were used to test the translation functionality is a very small set of possible NJOY jobs. Hence, there is a lot of scenarios within each NJOY module that has not been tested.

Note that modified versions of the original NJOY test problems had to be used as the expected output when when comparing the output from the translator. The floating-point numbers had to be specified such that the lexer in the translator could recognize them, as described in section 3.2.3 on page 13, and thus be changed into this form in the expected output as well. Since the emitter appends descriptive comments to every card (which are not present in the original test problems) as described in section 3.2.3 on page 13, these comments also had to be appended to the expected output such that the comparison could be performed.

[3]The source files for the NJOY software system consists of more than 100 000 lines.

6 Conclusions

In this thesis, a new input format, NJOY Input Format (`NIF`), has been proposed. A translator which is able to translate NIF into NJOY input instructions has been implemented.

It is possible to specify basic NJOY jobs in NIF with a syntax to make them easier to read and express. The resulting NIF programs can be translated into NJOY input instructions, which can be run by the NJOY software system. Production use is although not advisable, since it has been challenging to conduct rigorous and complete testing. It has also been evident that analysing the NJOY input instructions is not enough to design a new input format for the NJOY software system. Analysing the ENDF formats and the source code for the NJOY software system is required in order to build a translator which can conduct a complete semantic analysis for a NJOY job.

7 Future Work

Future work includes completing the semantic analysis and the organizer feature for all modules in the NJOY software system. The NJOY Input Format and the translator also needs to be systematically evaluated and verified by a complete software quality assurance process as described in reference [15].

A spin-off project, that is related to developing a user friendly and readable input format, is to construct a graphical user interface editor which can display and produce NJOY input instructions in a user friendly fashion.

References

[1] R. E. MacFarlane, "NJOY99 – code system for producing pointwise and multigroup neutron and photon cross-sections from ENDF/B data", Los Alamos Nat. Laboratory, Los Alamos, NM, Rep. RSIC PSR-480, 2000.

[2] M. B. Chadwick *et al.*, "ENDF/B-VII.0: Next Generation Evaluated Nuclear Data Library for Nuclear Science and Technology," *Nuclear Data Sheets*, vol. 107, no. 12, pp. 2931-3060, Dec. 2006.

[3] C. Gustavsson *et al.*, "Massive Computation Methodology for Reactor Operation (MACRO)," in *European Nuclear Conference*, 2010 © European Nuclear Society. ISBN: 978-92-95064-09-6

[4] A. C. Kahler and R. E. MacFarlane. (2010, Mar. 31). *User Input for NJOY99, updated through version 364* [Online]. Available: `http://t2.lanl.gov/codes/njoy99/Userinp.364`

[5] A. V. Aho *et al.*, *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007.

[6] A. V. Aho *et al.*, "Syntax Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 4, sec. 4.2, pp. 197-206.

[7] A. V. Aho *et al.*, "Lexical Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 3, sec. 3.5, pp. 140-146.

[8] A. V. Aho *et al.*, "Syntax Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 4, sec. 4.9, pp. 287-297.

[9] F. L. Drake, Jr., *et al.* (2011, Apr. 16) *Python v2.7.1 documentation* [Online]. Available: `http://docs.python.org/`

[10] A. V. Aho *et al.*, "Lexical Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 3, sec. 3.1, pp. 109-114.

[11] D. M. Beazley. (2011, Apr. 16). *PLY (Python Lex-Yacc)* [Online]. Available: `http://www.dabeaz.com/ply/ply.html`

[12] A. V. Aho *et al.*, "Lexical Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 3, sec. 3.3, pp. 116-124.

[13] A. V. Aho *et al.*, "Syntax Analysis" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 4, sec. 4.1, pp. 192-196.

[14] A. V. Aho *et al.*, "Introduction" in *Compilers: Principles, Techniques, & Tools*, Second Edition. Boston: Pearson Educ., 2007, ch. 1, sec. 1.2, pp. 8-9.

[15] C. Kaner *et al.*, *Testing Computer Software*, Second Edition. New York: John Wiley and Sons, Inc., 1999.

A Users Manual

A.1 Structure of `nifty`

The `nifty` directory structure is organized as shown in figure A.1.

```
nifty/
    bin/
        analyzer
        emitter
        lexer
        nifty
        organizer
        parser
        test
    data/
        ...
        test_problems/
    nifty/
        analyzer/
        emitter/
        environment/
        lexer/
        organizer/
        parser/
        tests/
    [ply/]
```

Figure A.1: Directory Structure of `nifty`

The `nifty/bin/` directory includes all executable Python scripts which are used for running and testing the translator. The `nifty` executable in the `nifty/bin/` directory runs the complete translation process on an input NIF program. The `test` executable runs the test suite. The other executables runs their corresponding named phase in the translation process (and all the successive phases that they depend on).

The test problems are located in the `nifty/data/test_problems/` directory. The `nifty/nifty/` directory contains the source code for the translator. The optional directory `ply/` indicates where PLY can be placed such that the translator is able to locate it.

A.2 Installation

PLY [11] is required to run the translator. It is sufficient to download PLY and put it in the `nifty/` top directory as indicated by figure A.1 on the previous page.

A.3 Running the Translator

The translator has been implemented as a command-line based interface. To run the entire translation process, the `nifty` executable in the `nifty/bin` directory should be used. Issuing the command

```
bin/nifty -h
```

in the `nifty/` top directory, will print the help message shown in figure A.2 on the following page.

```
usage: nifty [options] [input_file] [output_file]
options:
    -h, --help  show this help message and exit
    -a          don't analyze the input
    -o          don't organize the input
```

Figure A.2: Running `bin/nifty`

The `options` flag(s) are optional. The `input_file` and `output_file` are also optional. If no input file is given, standard input (`stdin`) will be used as the input source. If no output file is given, the result will be redirected to standard output (`stdout`).

As an example, the command

```
bin/nifty input.nif output
```

will simply run the translator on a file named `input.nif` and output the resulting NJOY input instructions on a file named `output`. The analyzer and organizer phase can be skipped by giving the `-a` and `-o` flag

`bin/nifty -a input.nif output`, to skip the analyzer phase

`bin/nifty -o input.nif output`, to skip the organizer phase

To skip both the organizer and analyzer phase, run `nifty` with both flags specified

```
bin/nifty -ao input.nif output
```

B Test Problems