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 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 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 [4], was analyzed separately such that a general structure and common language features could be extracted and used for further analysis.

As stated in [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 [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 [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 an 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 [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 [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] 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 Results

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. It was also designed to resemble the NJOY input instructions such that the documentation in [4] can be used when specifying NJOY jobs in NIF.

3.1.1 Grammar Definition

The resulting 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 [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.

3.1.2 NIF Examples

For an example, in algorithm 3, the lines 1 through 7 from algorithm 1 are expressed in NIF.

Algorithm 3 NIF version of *Algorithm 1*, lines 1-7

```
1  acer {
2      card_1 {
3          endf_input = 20;
4          pendf_input = 21;
5          multigroup_photon_input = 0;
6          ace_output = 31;
7          mcnp_directory_output = 32;
8      }
9      card_2 {
10         acer_run_option = 1;
11         print_control = 0;
12         ace_output_type = 1;
13     }
14     card_3 {
15         description = "proton + 7-n-14 apt la150
               njoy99 mcnpx";
16     }
17     card_5 {
18         material = 725;
19         temperature = 0.0;
20     }
21     card_6 {}
22     card_7 {}
23 }
```

Descriptive names have been specified for the identifiers. Algorithm 4 illustrates how lines 8 through 14 from algorithm 1 are expressed in NIF. Note the difference of the identifier names compared to algorithm 3. The

names in algorithm 4 has been chosen to reflect the documentation in [4]. The `stop` instruction on line 14 in algorithm 1 does not have to be specified, the translator will automatically append it in the translation process.

Algorithm 4 NIF version of *Algorithm 1*, lines 8-14

```
1  acer {
2      card_1 {
3          nendf = 0;
4          npend = 31;
5          ngend = 33;
6          nace = 34;
7          ndir = 35;
8      }
9      card_2 {
10         iopt = 7;
11         iprint = 1;
12         ntype = 2;
13     }
14     card_3 {
15         hk = "proton + 7-n-14 apt la150 njoy99 mcnpx
               ";
16     }
17 }
18 viewr {
19     card_1 {
20         infile = 33;
21         nps = 36;
22     }
23 }
```

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.

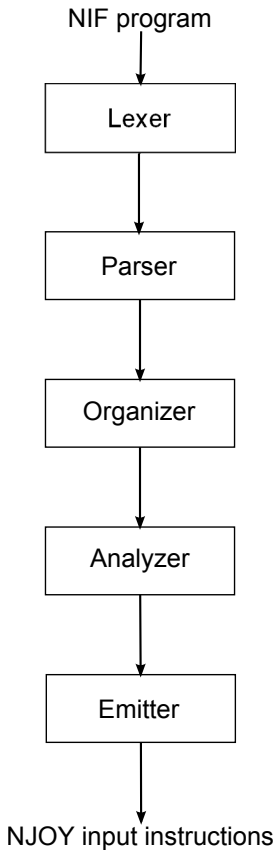

Figure 1: Translation process in nifty

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 which is not found in a translator as described in [5]. The organizer is a phase where the order of the statements in a card are analysed 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.

3.2.2 Reserved Words

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.

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 [4], thus enforcing the use of a specific set of cards and modules as mentioned previously.

Parser The parser is responsible of enforcing the structure of the NIF grammar and constructing the syntax tree. As input, the parser expects a token

stream 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 analyses 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 [4]. The modules and cards still needs to be given in the correct order though.

Each module requires its own organizer implementation since each module has its specific set of rules as described in [4]. Since the identifier names are hard coded into 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 translator.

Analyzer The analyzer expects a syntax tree as its input. Like in the organizer phase, the modules needs to be analysed separately since each

module has its specific set of rules as described in [4]. As such, each module 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 the ordered nature in [4] and since the cards and the identifiers have reserved names. As such, it can in a simple fashion determine whether a card or an identifier has been defined or not and whether an identifier has been defined as the expected type, et cetera.

The analyzer does not alter the syntax tree, it just analyses 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 semantical 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.

4 Discussion

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.

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 modules expect and how they operate on it. 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 of the NJOY modules.

The testing that was conducted within this work is not rigorous enough. NJOY is a large and complex program[2]. The NJOY test problems which were used to test the translation functionality is a very small set of possible NJOY jobs. Hence, there is a lot of possible scenarios that has not been tested.

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

5 Conclusions

- Improvement on the existing situation?
  - Readable?
- Challenges?
  - Constructing a decent input format.
  - The physics (even if it's not within the scope). Documentation is full of it, kind of.
- Usable?
  - Production: not advisable. Grammar not verified. "Toy" translator.

5.1 Future Work

- Possible improvements?
  - Recognize more data-types. E.g. materials, temperatures, etc.
- Complete context-free grammar?
  - Expand grammar. More tokens (`TEMPERATURE`, `MATERIAL`, etc).
- GUI editor?
- Efficient implementation, e.g. C?

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.