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

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 = 8225
```

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.

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 `8225` could be represented as tokens of the form

`<IDENTIFIER, material>`,

`<ASSIGNMENT, =>`, and

`<INTEGER, 8225>`

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.

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.

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

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

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.

3 Results

3.1 NJOY Input Format (`NIF`)

The new input format, NJOY Input Format (NIF), is basically the NJOY input instructions [4] 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.

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_singleton "=" r_value_singleton
             | l_value_pair "=" r_value_pair
             | l_value_triplet "=" r_value_triplet

l_value_singleton ::= l_value
r_value_singleton ::= r_value

l_value_pair ::= l_value "," l_value
r_value_pair ::= r_value "," r_value

l_value_triplet ::= l_value "," l_value "," l_value
r_value_triplet ::= r_value "," r_value "," r_value

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 tuple is assigned to hold the values of a right hand side tuple. A left hand side tuple is an ordered list of elements, where the elements can be an array or identifier. A right hand side tuple 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 value or a character string.

For an example, in algorithm 3, the lines 1 through 7 in 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 }
```

3.2 NJOY Input Format Translator (`nifty`)

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.

The phases in the translation process is shown in figure 1. The lexer implements the lexical analysis and the parser implements the syntax analysis. The organizer implements an order analysis phase. The analyzer implements the semantic analysis. The emitter implements a NJOY input instructions generator.

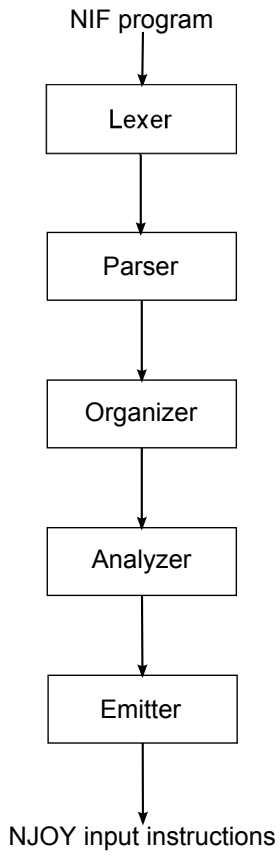

Figure 1: Translation process in nifty

Lexer Lexical analyzer.

- Used a lexical analyzer to generate tokens and detect errors. PLY Lex.
- Reserved words.
- Wrote regular expressions to recognize tokens.

Parser Syntax analyzer.

- Used a parser generator such to detect and report syntax errors. PLY YACC.

- Constructed an syntax tree to represent the structure of NIF.
- Description of syntax tree?

Organizer Order organizer.

- To arrange in order.

Analyzer Semantic analyzer.

- Type checks.
- Detected and reported errors.

Emitter Code generation.

- Massage the syntax tree into NJOY input instructions.
- Flattened the tree structure.

4 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.

4.1 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.