CSC4180 Assignment 2: Implement a Scanner for Oat v.1 Language

Release Date: February 29, 2024
Due Date: March 20, 2024

1 Introduction

In this assignment, you are required to **design and implement a scanner (lexical analyzer) for Oat v.1 Programming Language** which takes the **Oat program** into **a series of Oat tokens**. You need to implement two scanners for the same Oat v.1 Language:

- **Scanner implemented by flex**: Using a lexical generator like flex makes things a lot easier, and it can produce ground-truth token outputs to verify the scanner implemented by hand.
- **Scanner implemented by hand**: This is the main challenge of this assignment since you need to implement all the details of lexical analysis by yourself.
  - Design appropriate data structures for DFA and NFA.
  - Implement how to construct NFA from regular expressions of language specification.
  - Implement how to convert NFA to DFA by subset construction algorithm.
  - [**Bonus**] Implement how to minimize the number of states in DFA.
  - Implement lexical analysis with DFA and output ¡token, lexeme¿ pairs.

Any plagiarism will not be tolerated and will get 0 point!!!

2 Review of Lexical Analysis

A lexical analyzer (or called scanner) converts a lexical specification consisting of a list of regular expressions and corresponding actions and breaks the input source program into tokens. This is done in five stages and the flowchart is shown below:

- **Define Lexical Specification**: The specification of a programming language often includes a set of rules, the lexical grammar, which defines the lexical syntax.
- **Form Regular Expression**: Since the lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions. They define the set of possible character sequences (lexemes) of a token.
- **Recognize Regular Expression with NFA**: An NFA differs from a DFA in that each state can transition to zero or more other states on each input symbol, and a state can also transition to others without reading a symbol. One of the most significant advantages of NFA is that they can easily recognize regular languages.
- **Convert NFA to DFA**: [Optional] Due to the non-deterministic property, NFA is usually hard to maintain and manipulate for lexical analysis, and it needs to be converted into a corresponding DFA. However, it is also possible to directly do lexical analysis on NFA.
- **Table-Based Scanning with DFA**: DFA clearly defines deterministic state transition from state to state, and lexical analysis can be done character by character.

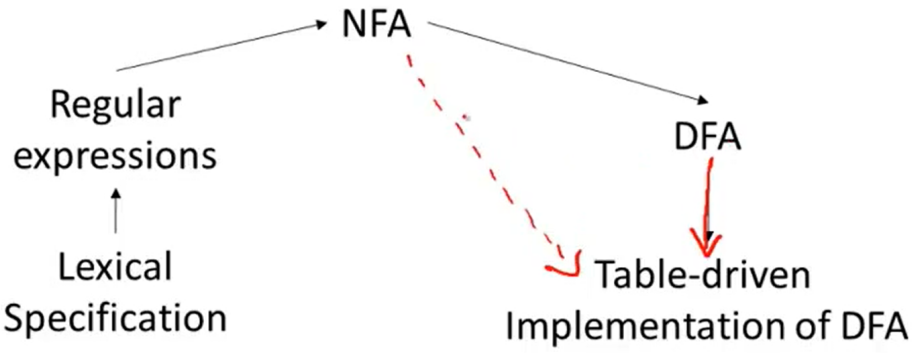

Figure 1: Flowchart of 5-Stage Lexical Analysis
(Lecture P16 Slides Screenshot from Stanford University CS143: Compiler by Prof.Alex Aiken)

2.1 Pesudo Code of Converting NFA to DFA (Subset Construction)

The following algorithm constructs a transition table *Dtran* for DFA. Each DFA state is a set of NFA states and we construct *Dtran* so that DFA will simulate "in parallel" all possible moves NFA can make on a given input string. The following operations are used to keep track of sets of NFA states (s represents an NFA state and T represents a set of NFA states).

OPERATIONS	DESCRIPTIONS
$\epsilon - closure(s)$	Set of NFA states reachable from NFA state s on $\epsilon - transition$ alone
$\epsilon - closure(T)$	Set of NFA states reachable from a set of NFA states in T on $\epsilon - transition$ alone
$move(T, a)$	Set of NFA states ti which there is a transition on input character a of the alphabet from some NFA state s in T

Table 1: Operations on NFA States

Algorithm 1 Subset Construction

Initially, $\epsilon - closure(s_0)$ is the only state in *Dstates* and it is unmarked;
while there is an unmarked state T in *Dstates* **do**
  mark T;
  for each input symbol a in alphabet **do**
    $U := \epsilon - closure(move(T, a))$
    if U is not in *Dstates* **then**
      add U as an unmarked state to *Dstates*;
    end if
    $Dtran[T, a] := U$
  end for
end while

2.2 Useful Online Learning Materials for Finite Automata

- Implementing Lexical Generators (Cornell University CS4120: Introduction to Compilers, Spring 2020)
- GeeksforGeeks Tutorial on Finite Automata
- Textbook Section of Lexical Analysis (University of Ottawa)

3 Oat Language Specification

3.1 Language Specification

Oat is a popular programming language for compiler course teaching, which is adopted by top universities like University of Pennsylvania and University of Princeton. It has two versions of language specification: v.1 and v.2. All assignments starting from this time (A2) adopts the Oat v.1 for simplicity. For those of you who want to challenge yourself and do something more advanced, you are welcomed to adopt Oat v.2, which is more cmplicated in grammar and semantic rules.

- Oat v.1 Language Specification Document (Reserved Words Highlighted)
- Oat v.1 Detailed Explanation by Prof.Steve Zdancewic, COS320: Compiling Techniques, University of Princeton
- Oat v.2 Language Specification Document (Reserved Words Highlighted)
- Oat v.2 Detailed Explanation by Prof.Steve Zdancewic, COS320: Compiling Techniques, University of Princeton

Oat supports multiple base-types of data: int, bool, and string, as well as arrays of such data. The Oat language specification contains a definition of the language syntax. Oat concrete syntax does not require a local variable declaration to include a type definition, instead it uses the keyword var, as shown in the example above. Oat mostly sticks with C or Java-like syntax, except for some quirks: null requires a type annotation, and bit-wise arithmetic operators have their own notation (so there is no overloading).

Oat v.1 will not be safe: it is possible to access an array out of bounds or to call a function with incorrectly typed arguments. The next version of Oat **(v.2, which is more complicated and not compulsotry for this course)** will address these issues and add some other missing features. In particular, although the grammar gives a syntax for function types, this version of Oat does not need to support function pointers; these are included in anticipation of the next project.

3.2 Oat Syntax Highlighter Extension in VSCode

To better deal with the Oat language, you are recommended to install the VSCode extension named **Oat Intellisense (Extension ID: tlcyr4.oat)** implemented by Tigar Cyr. It can highlight the syntax of Oat program (file ended with .oat) and gives you an intuitive understanding of this language.

3.3 Some Regular Expression Formulation

The language specification document already defines all the reserved words, CFGs, and precedence for various operators. Here only shows the regular expressions of some important token classes, that are not covered in the language specification document.

ID(Identifier)	= [a−zA−Z][a−zA−Z0−9_]*
INTLITERAL	= [0−9]+
STRINGLITERAL	= "[^"]*"
COMMENT	= \/*([^*]|*+[^*/])**+\/

Examples

- **ID (Identifier)**: main, program, print_string
- **INTLITERAL (Integer)**: $0, -1, 5$
- **STRINGLITERAL (String)**: "hello_world"
- **COMMENT**: $/ * Comment * /$

4 Sample Output

Similar to assignment 1, the scanner should take a source program as input (.oat file) and outputs a series of $< tokenclass, lexeme >$ pairs for later manipulation of parser.

4.1 Sample Program 1

Input Oat Program

```
int main() {
        var str = "hello world!";
         print_string (str);
        return 0;
}
```

Output Token and Lexeme Pairs

```
TINT int
ID main
LPAREN (
RPAREN )
LBRACE {
VAR var
ID str
ASSIGN =
STRINGLITERAL "hello world!"
SEMICOLON ;
ID print_string
LPAREN (
ID str
RPAREN )
SEMICOLON ;
RETURN return
INTLITERAL 0
SEMICOLON ;
RBRACE }
```

4.2 Sample Program 2

Input Oat Program

```
int fact(int x) {
        var acc = 1;
        while (x > 0) {
                acc = acc * x;
                x = x - 1;
        }
        return acc;
}

int program(int argc, string [] argv) {
         print_string ( string_of_int (fact (5)));
        return 0;
}
```

Output Token and Lexeme Pairs

```
TINT int
ID fact
LPAREN (
TINT int
ID x
```

```
RPAREN )
LBRACE {
VAR var
ID acc
ASSIGN =
INTLITERAL 1
SEMICOLON ;
WHILE while
LPAREN (
ID x
GREAT >
INTLITERAL 0
RPAREN )
LBRACE {
ID acc
ASSIGN =
ID acc
STAR *
ID x
SEMICOLON ;
ID x
ASSIGN =
ID x
MINUS -
INTLITERAL 1
SEMICOLON ;
RBRACE }
RETURN return
ID acc
SEMICOLON ;
RBRACE }
TINT int
ID program
LPAREN (
TINT int
ID argc
COMMA ,
TSTRING string
LBRACKET [
RBRACKET ]
ID argv
RPAREN )
LBRACE {
ID  print_string
LPAREN (
ID  string_of_int
LPAREN (
ID fact
LPAREN (
INTLITERAL 5
RPAREN )
RPAREN )
RPAREN )
SEMICOLON ;
RETURN return
INTLITERAL 0
SEMICOLON ;
RBRACE }
```

4.3 Bonus (Extra Credits 10%)

If you are interested and want to challenge yourself for more advanced compiling techniques, you may try the following options:

- Implement DFA minimization algorithm and apply it in your scanner to make it more efficient
- Improve your scanner for more complicated Oat v.2 Language
- Add error handling features for your scanner
- Any other you can think about...

The extra credits are not going to be added to your final grade since this is an elective course and many of you are interested in compiler and want to do some advanced work. In other words, the extra credits you earn cannot be added to the grade of any programming assignment to make it higher. However, these credits can serve as a proof of your interest and hard work for this course, and I will try my best to offer as high A- rate as I can, probably higher than 40%, by consulting with Prof.CHUNG.

5 Evaluation and Submission

5.1 Grading Scheme

- **Scanner by Flex**: 60%

  Learning how to use lexical generator is an easier and more convenient way of implementing a scanner. Students who are capable of using such tools get the basic 60 points.

- **Scanner by hand**: 30%
  - Construct NFA from Regular Expression 10%
  - Convert NFA to DFA by Subset Construction 10%
  - Scanning with DFA 10%
- **Technical Report**: 10%

  The report doesn't need to be very long, but it needs to clearly answers the following questions:
  - Briefly explain how did you design and implement this assignment?
  - Why we choose regular expression to represent lexical specification?
  - Why NFA is more suitable than DFA to recognize regular expression, and how did you enable NFA to recognize different kinds of regular expression?
  - How did you enable your scanner to always recognize the longest match and the most precedent match?
  - Why do we still need to convert NFA into DFA for lexical analysis in most cases?
- **Bonus:** 10%

  Refer to section 4.3 for more details. The grading of this part will be very flexible and highly depend on the TA's judgement. Please specify clearly what you have done for the bonus part so that he do not miss anything.

5.2 Policy of Late Submission

- Late submission within 10 minutes after then DDL is tolerated for possible network issues during submission.
- 10 Points deducted for each day after the DDL (11 minutes late will be considered as one day, so be careful)
- Zero point if you submitted your project late for more than two days

5.3 Submission with Source Code

If you want to submit source C/C++ program that is executable in our RISC-V docker container, your submission should look like:

```
csc4180-a2-118010200.zip
|-
|--- csc4180-a2-118010200-report.pdf
|-
|--- testcases
|-
|--- src
                |-
                |---Makefile
                |---lexer.l (Scanner by Flex)
                |---scanner.cpp
                |---scanner.hpp
                |---tokens.cpp
                |---tokens.hpp
                |---Other possible files
```

5.4 Submission with Docker

If you want to submit your program in a docker, your submission should look like:

```
csc4180-a2-118010200.zip
|-
|--- csc4180-a2-118010200.Dockerfile
|-
|--- csc4180-a2-118010200-report.pdf
|-
|--- src
        |-
        |---Makefile
        |-
        |---run_compiler.sh
        |-
        |---Other possible files
```