COMPILERS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

DEPARTMENT OF INFORMATION AND COMPUTER ENGINEERING

PART A2

AUTOMATIC FINITE-STATE ENCODING VIA FSM

TEAM / WORK DETAILS

TEAM NUMBER: 2
STUDENT DETAILS 1: Athanasiou Vasileios Evangelos (UNIWA-19390005)
STUDENT DETAILS 2: Theocharis Georgios (UNIWA-19390283)
STUDENT DETAILS 3: Tatsis Pantelis (UNIWA-20390226)
STUDENT DETAILS 4 : Iliou Ioannis (UNIWA-19390066)
STUDENT DETAILS 5: Dominaris Vasileios (UNIWA-21390055)

WORKING DEPARTMENT: B1 Wednesday 12:00-14:00
LAB INSTRUCTOR: Iordanakis Michael

COMPILERS

CONTENTS

COMPILERS

Introduction

The writing was undertaken by the student

Athanasiou Vasileios Evangelos (UNIWA-19390005)

Reference

Part "A2 Coding of finite state automata via FSM" is the 1st part of the work "Creating an Independent Parser with the FLEX generator" and presents, through steps that are thoroughly analyzed in this report, the coding of finite state automata for the verbal units of the language Uni-C (the Uni-C language is a subset of the C programming language). Finally, the single automaton is also simulated.

Team organization

Through the climate of cooperation, the team members were organized and the tasks were divided equally so that the workload was fair and beneficial for everyone with the ultimate goal that everyone acquires to a satisfactory degree the knowledge and skills provided by the implementation of this part. The group was divided into sub-groups, each of which separately undertook the simulation of the corresponding Uni-C sub-word units. Finally, the entire group is regrouped for the simulation of the single automaton, with the ultimate goal of covering any difficulties encountered during the simulation of the individual units.

Reference content

The presentation involves performing a series of finite steps that describe the Uni-C language units and are as follows:

1. Reference of the identification model
2. Drawing regular expressions
3. Design of automatic finite states
4. Design of transition tables
5. Encoding automatic finite states via the FSM metatool
6. Annotating control cases through exhaustive test runs

In addition, for each word unit are written:

1. Which subgroup performed the simulation
2. Annotating the control cases
3. Any difficulties encountered and ways to deal with them
4. Reporting any deficiencies and code execution correctness

Analysis of responsibilities/roles of all team members

Athanasiou Vasileios Evangelos (UNIWA-19390005)

In the role of general coordinator, the student participated in the tasks:

- Introduction
- 3. Integers
- 4. Floating point numbers
- Final

He was the coordinator in the organization of the group for the design of the finite automata and for the writing of the report.

Theocharis Georgios (UNIWA-19390283)

In the role of developer, the student participated in the tasks:

- 1. Identifiers
- 3. Integers
- 5. Comments
- Final

He was the coordinator in organizing the team to code the automatic finite states through the FSM metatool and to design the transition tables.

Tatsis Pantelis (UNIWA-20390226)

In the role of tester, the student participated in the tasks:

- 2. Strings
- 3. Integers
- 6. White_spaces characters
- Final

He was a coordinator in organizing the team to select appropriate control cases for the reliability of automatic finite coding, as well as in commenting on the results.

Iliou Ioannis (UNIWA-19390066)

In the role of commentator, the student participated in the tasks:

- 1. Identifiers
- 2. String
- 4. Floating point numbers

- Final

He was the coordinator in the organization of the team to comment on the results of the control cases, as well as their appropriate selection for the reliability of the coding of the automatic finites.

Dominaris Vasileios (UNIWA-21390055)

In the role of analyst, the student participated in the tasks:

- 2. Strings
- 5. Comments
- 6. White_spaces characters
- Final

He was a coordinator in the regular expression design organization

COMPILERS

1. Identifiers (names)

The simulation was undertaken by the students

Theocharis Georgios (PADA-19390283) – Iliou Ioannis (PADA-19390066)

1.1 Identification standard

The function of the automaton is to recognize identifiers supported by the Uni-C language based on an identifier pattern.

The recognition pattern recognizes lexemes consisting of one or more lowercase/uppercase Latin letters or an underscore (_) and, in addition to the initial character , numbers.

Additionally, they are returned as tokens with the identifier name ***identifier*** .

1.2 Regular Expression

The regular expression for lexical units corresponding to alphanumeric variables is:

```
[a-zA-Z_][a-zA-Z0-9_]{0.31}
```

This regular expression is used to match an alphanumeric string consisting of a letter (uppercase or lowercase), underscore, or numbers, with at least one character and up to 32 characters.

Specifically, this expression means:

`[a-zA-Z_]`

A letter (upper or lower case) or the _ (underscore) character.

`[a-zA-Z0-9_]{0.31}`

Followed by zero to 31 characters, which can be letters (upper or lower case), numbers, or the _ (underscore) character.

1.3 Finite state automaton or DM

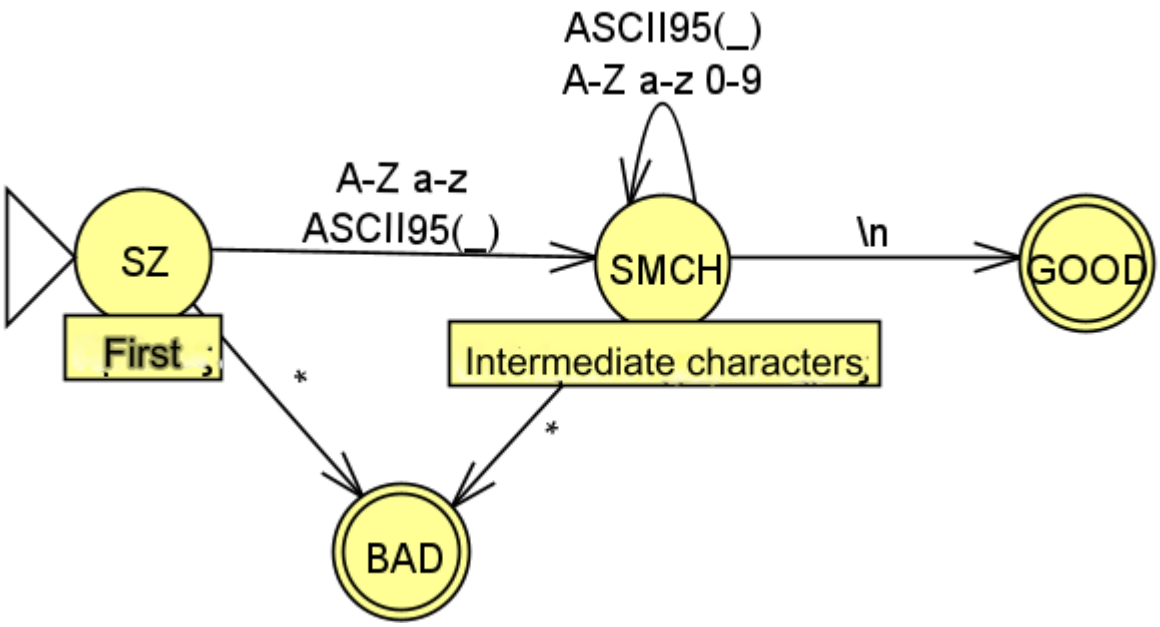


Figure 1.3.1 Uni-C finite automaton of identifiers

Initial state (SZ) for the first character:

If the next character is a letter (upper or lower case) or underscore (_), go to SMCH state.

If next character is any other character, go to BAD exit state.

SMCH status for intermediate characters:

If the next character is a letter (uppercase or lowercase), an underscore (_), or a digit, we stay in the same state (SMCH).

If the next character is a newline (\n), the identifier is valid, so go to the GOOD exit state.

If the next character is any other character, the identifier is invalid, so go to the BAD exit state.

Exit Status GOOD:

ID recognition successful.

1.4 Transition Table

	az	AZ	0-9	-	/n	EXIT STATUS
SZ	SMCH	SMCH	BAD	SMCH	BAD	NO
SMCH	SMCH	SMCH	SMCH	SMCH	GOOD	NO
GOOD						NAI
BAD						YES

Table 1.4.1 Transition table of Uni-C language identifiers

Status Acronym	Status Name	Status Information
SZ	State Z	The status for the 1st character
SMCH	State Mid Characters	The status for intermediate characters

Table 1.4.2 Status table of Uni-C language identifiers

1.5 Encoding via FSM

The FSM encoding of the Uni-C identifier automaton is in the source file **1_identifiers.fsm**

```
// ================== 1_IDs ==================


START = SZ // ==== Initial state is SZ ====


SZ: // --- [State Z] State for 1st character ---

AZ az -> SMCH // Lower or uppercase Latin letter

_ -> SMCH // Underscore

* -> BAD // The identifier is invalid
```

```
SMCH: // --- [State Mid Characters] State for mid
characters ---

AZ az -> SMCH // Lower or uppercase Latin letters

_ -> SMCH // Underscore

0-9 -> SMCH // Digits

\n -> GOOD // The recognition is valid

* -> BAD // ID is invalid


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

1.6 Audit Cases

#1 The automaton accepts this particular string because it starts with a Latin letter (az) and then contains a number. That is, it follows the rules of identifiers in the Uni-C language.

```
./fsm 1_identifiers.fsm

a1

YES
```

#2 The string is accepted by the automaton because it starts with an accepted character (underscore) and then contains a number.

```
./fsm 1_identifiers.fsm

_1

YES
```

#3 The automaton accepts the following string, as all characters (az, AZ) are acceptable for naming an identifier in the Uni-C language.

```
./fsm 1_identifiers.fsm
```

```
Spectacular

YES
```

```
#4    The character “a” is accepted by the Uni-C language as an identifier as it starts with a Latin letter (az) and does not contain any unacceptable special character. There is no character limit preventing it from being invalid.

./fsm 1_identifiers.fsm

a

YES
```

```
#5    The following input is accepted by the automaton because the string starts with a Latin letter (AZ). The string does not contain any special character that would be unacceptable in Uni-C identifier naming.

./fsm 1_identifiers.fsm

A1

YES
```

```
#6    The string is accepted by the automaton because it starts with a Latin letter (AZ) and contains only accepted characters (numbers, Latin letters, underscore).

./fsm 1_identifiers.fsm

SPEctacular_15_

YES
```

```
#7    The string is accepted by the automaton because the Uni-C language allows an identifier to begin with the special underscore character (_). There is no invalid character in this string.

./fsm 1_identifiers.fsm

_a

YES
```

#8 The string is accepted by the automaton, as the Uni-C language allows an identifier to begin with the special character (_). As in the previous example there is no difference in the result because we gave an uppercase character instead of a lowercase character.

```
./fsm 1_identifiers.fsm

_W

YES
```

#9 The following string is not accepted by automaton to set an identifier because the blank character cannot be given in a Uni-C identifier.

```
./fsm 1_identifiers.fsm

_ _A

NO
```

#10 The string is accepted by the automaton as it starts with a Latin letter (az) as defined by the Uni-C language in names and at the same time all the characters it contains are acceptable.

```
./fsm 1_identifiers.fsm

a_b_c_1_2_3

YES
```

#11 The string is acceptable for identifier naming because it starts with a Latin letter (AZ) and does not contain any character not acceptable for identifier definition.

```
./fsm 1_identifiers.fsm

AA4_F

YES
```

#12 The string below we see starts with a number, which is not compatible with the Uni-C rules for defining identifiers. For this reason the automaton rejects the specific series of characters.

```
./fsm 1_identifiers.fsm
```

```
1_id

NO
```

#13 The following sequence of characters cannot be accepted by the automaton for the identifier name, as it contains the special character '/'. Any special character given to an identifier other than '_' makes the string not acceptable.

```
./fsm 1_identifiers.fsm

/id

NO
```

#14 The following string of characters is not accepted by the automaton because it contains the dot special character '.'. This character is not accepted in identifier naming in the Uni-C language.

```
./fsm 1_identifiers.fsm

id.student

NO
```

#15 The following string is not accepted by automaton for setting an identifier because it contains blank characters ' ' and the blank character cannot be given in a Uni-C identifier.

```
./fsm 1_identifiers.fsm

id student

NO
```

#16 The string is not accepted by the automaton, as the character 'E' is a Greek character. Greek characters cannot be used to define a name in the Uni-C language.

```
./fsm -trace 1_identifiers.fsm

E 2

sz \316 -> smch

fsm: in 1_identifiers.fsm, state 'smch' input \225 not accepted
```

```
#17   The following string is acceptable because a Uni-C language identifier can start with the special character '_' and be followed by Latin letters, numbers or even an underscore. Any other special character is not accepted.

./fsm 1_identifiers.fsm

_user

YES
```

```
#18   The string is not accepted by the automaton because it starts with the empty character ' ', this is forbidden based on the identifier naming rules in the Uni-C language.

./fsm 1_identifiers.fsm

user

NO
```

```
#19   The string is not accepted as the newline character '\n' was not used to enter it into the automaton, instead EOF was used and for this reason we do not get a positive result from the automaton.

./fsm -trace 1_identifiers.fsm

hi sz h -> smch

smch i -> smch

smch EOF -> bad

NO
```

1.7 Problems and solutions

We encountered a problem when the team tried to use Greek characters for login.

```
./fsm -trace 1_identifiers.fsm

Hi

sz \316 -> smch

Segmentation fault
```

The ASCII encoding of the Greek characters created complications in character recognition by the FSM meta-tool, as the letters Γ, K, λ had the same ASCII code (\316). Even on unaccepted inputs with Latin letters the exit status was NO and not as shown in these two examples.

```
./fsm -trace 1_identifiers.fsm

Good morning

sz \316 -> smch

smch \232 -> bad

fsm: in 1_identifiers.fsm, state 'bad' input \316 not accepted
```

A solution to the problem was not found, since anyway the Uni-C standard does not recognize identifiers with Greek characters, but neither does the language's alphabet include Greek characters in its verbal units.

Another problem we faced was the character limit. In the automaton and therefore in the FSM encoding we found no way to represent/encode accordingly the character limit proposed by the Uni-C standard (32 characters). Therefore, the automaton accepts matches from 1 to infinite characters as opposed to the regular expression which matches strings of up to 32 characters.

1.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 1_identifiers.fsm

./fsm -list 1_identifiers.fsm

./fsm -trace 1_identifiers.fsm
```

2. Strings

The simulation was undertaken by the students

Dominaris Vasileios (UNIWA-21390055) – Iliou Ioannis (UNIWA-19390066) – Tatsis Pantelis (UNIWA-20390226)

2.1 Identification standard

The function of the automaton is to recognize verbal literals supported by the Uni-C language based on a recognition pattern.

The recognition pattern recognizes strings that are enclosed in double quotes (“) and include any character other than backslash (\), newline (\n), or double quotes that require an escape sequence to be used.

Additionally, they are returned as tokens with the identifier name ***string*** .

2.2 Regular Expression

This regular expression describes a pattern that matches a combination of characters that can contain strings within double quotes, escaped using the backslash (\) character. In other words, the expression recognizes valid literals enclosed in double quotes.

```
"([^"\\]*(\\[\\n"][^"\\]*)*)"
```

Here is the breakdown of the parts of the regular expression:

```
"
```

It corresponds to a double quotation mark that marks the beginning of the verbal.

```
([^"\\]*(\\[\\n"][^"\\]*)*)
```

This part matches an escaped string.

```
[^"\\]*
```

Matches zero or more characters that are not double quotes (" or \\).

```
(\\[\\n"][^"\\]*)*
```

This part corresponds to an escape sequence.

```
\\
```

Matches a backslash (\) character.

```
[\\n"]
```

Matches a character that can be followed by an escape sequence (\, n, or ").

```
[^"\\]*
```

Matches zero or more characters that are not double quotes (" or \\).

In summary, this regular expression matches a string containing escaped double quotes and the characters \, n, " inside them.

2.3 Finite state automaton or DM

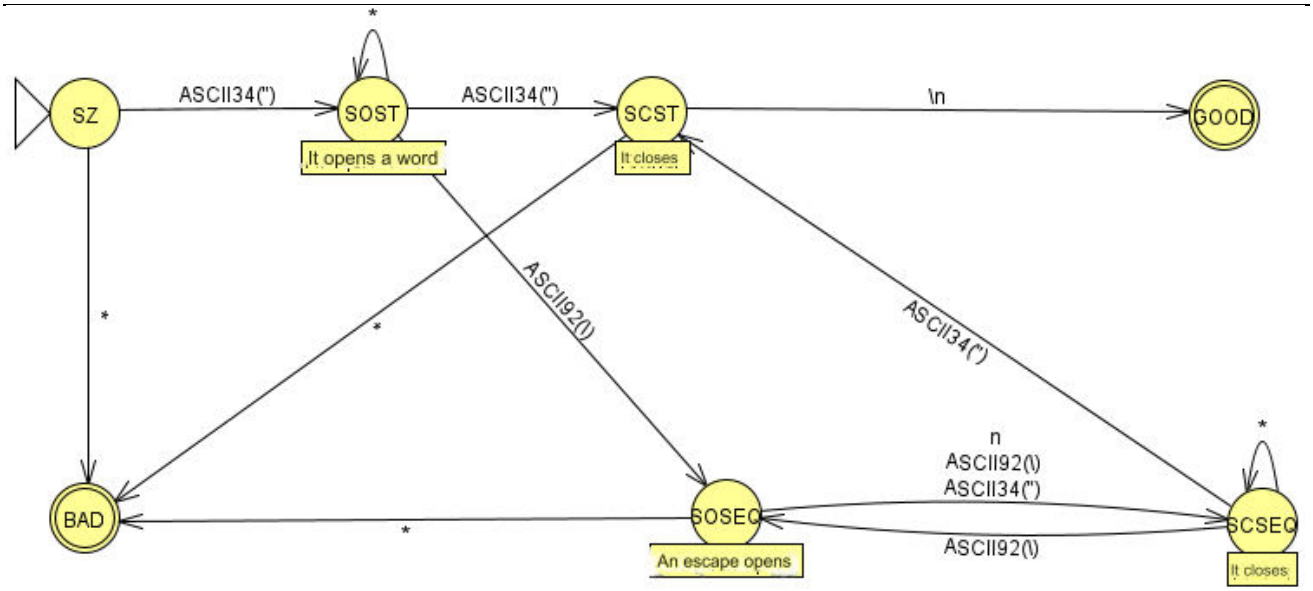


Figure 2.3.1 Finite automaton of the Uni-C language

The automaton starts from the initial state SZ, where the verb must start with double quotes ".

If a double quote is read, the automaton goes into SOST state. In this state, if a double quote is read, it goes to SCST state, if \ is read, it goes to SOSEQ state, otherwise it stays in the same SOST state.

In SCST, if \n is read, the literal is considered valid and the automaton goes to the GOOD exit state, while any other character goes to the BAD exit state.

In the SOSEQ, SCSEQ states, the automaton reads the characters that can appear in escape sequence. If ", \, or n is read, the automaton goes to the SCSEQ state, while any other character goes to the BAD exit state.

In SCSEQ, if " is read, the automaton goes to the SCST state, if \ is read, it goes back to SOSEQ, while at any other character it remains in the same SCSEQ state.

The exit status is GOOD when the literal is correctly enclosed in double quotes and contains no invalid characters.

2.4 Transition Table

	"	n	\	\n	EXIT STATUS
SZ	SOST	BAD	BAD	BAD	NO
SOST	SCST	SOST	SOSEQ	SOST	NO
SOSEQ	SCSEQ	SCSEQ	SCSEQ	BAD	NO
SCSEQ	SCST	SCSEQ	SOSEQ	SCST	NO
SCST	BAD	BAD	BAD	GOOD	NO
GOOD					YES
BAD					YES

Table 2.4.1 Transition table of Uni-C verbal literals

Status Acronym	Status Name	Status Information
SZ	State Z	The situation for the opening of the verbal
SOST	State Open String	The situation for the content of the verbal
SOSEQ	State Open Sequence	The condition for opening an escape sequence within the lexicon
SCEQ	State Close Sequence	The condition for the character followed by escape sequence within the lexicon
SCST	State Close String	The condition for closing the verbal

Table 2.4.2 State table of Uni-C lexicals

2.5 Encoding via FSM

The FSM encoding of the automaton of the Uni-C language literals is in the source file **2_strings.fsm**

```
// ================== 2_Lexuals ==================


START = SZ // ==== Initial state is SZ ====


SZ: // --- [State Z] State to open the verbal ---

" -> SOST // The lexical is opened with a double quote

* -> BAD // The literal is invalid


SOST: // --- [State Open String] State for the content of the
verb ---

" -> SCST // The verb is enclosed in double quotes

\\ -> SOSEQ // Escape sequence follows

* -> SOST // Any other character follows


escaped sequence characters ---


\\ -> SCSEQ // Followed by backslash character (\)

" -> SCSEQ // Double quote follows

n -> SCSEQ // Follows n for newline

* -> BAD // The literal is invalid


SCSEQ: // --- [State Close Sequence] State to close verbal or
escape sequence ---

" -> SCST // The verb is enclosed in double quotes

\\ -> SOSEQ // Return to state for escaped characters

* -> SCSEQ // Any other character follows


SCST: // --- [State Close String] State to close the
verbal ---

\n -> GOOD // The literal is valid
```

```
* -> BAD // The literal is invalid


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

2.6 Audit Cases

```
#1    It is acceptable because it opens and closes with double quotes (“) and is followed by a newline character (\n).

./fsm 2_strings.fsm

"KALHMERA"

YES
```

```
#2    It is acceptable because in “SCS” state, when completion is followed by a newline character (\n), then the state proceeds to an exit state set to “YES”.

./fsm 2_strings.fsm

""

YES
```

```
#3    It's acceptable because it opens with a double quote (“), then we go into SOS mode where we encounter a double quote (“) and then it encounters (\n). The double-quote character is correctly used within the lexical with an escape sequence (\”), as is the newline (\n).

./fsm 2_strings.fsm

"hello, \"world!\"\n"

YES
```

```
#4 It is acceptable because it opens and closes with a double quote (“”). Escape characters (\, “, newline) in the Uni-C language are used correctly within the dictionary.

./fsm -trace 2_strings.fsm

"\"\\n\"\"\\n\""

sz " -> sost

sost \ -> soseq

soseq " -> scseq

scseq \ -> soseq

soseq \ -> scseq

scseq n -> scseq

scseq \ -> soseq

soseq " -> scseq

scseq \ -> soseq

soseq " -> scseq

scseq \ -> soseq

soseq \ -> scseq

scseq n -> scseq

scseq \ -> soseq

soseq " -> scseq

scseq " -> scst

scst \n -> good

YES
```

```
#5 It's wrong because the verb doesn't start with a double quote ("").

./fsm 2_strings.fsm

f"number of loops is {counter}"

NO
```

#6 It is acceptable because it opens with a double quote (“), and proceeds to the state where the characters "6", "/", "3", "=", "2" follow. The verb is correctly closed with double quotation marks ("").

```
./fsm 2_strings.fsm

"6/3=2"

YES
```

#7 It is not accepted because the literal is opened and closed with a double quote (“) and the Uni-C language accepts the use of the double quote (“) as a character within the literal, only by using an escape sequence (\”).

```
./fsm 2_strings.fsm

"He said 'Why Brutus?'"

NO
```

#8 It is not acceptable because the Uni-C language accepts the use of the backslash character (\) as a character within the literal, only using an escape sequence (\\).

```
./fsm -trace 2_strings.fsm

"6\3=2"

sz " -> sost

sost 6 -> sost

sost \ -> soseq

soseq 3 -> bad

bad = -> bad

bad 2 -> bad

bad " -> bad

bad \n -> bad

bad EOF -> bad

NO
```

#9 It is valid because the literal is opened and closed with double quotes and the backslash character (\) is used within the literal in escape order.

```
./fsm -trace 2_strings.fsm

"6\\3=2"

sz " -> sost

sost 6 -> sost

sost \ -> soseq

soseq \ -> scseq

scseq 3 -> scseq

scseq = -> scseq

scseq 2 -> scseq

scseq " -> scst

scst \n -> good

YES
```

#10 Invalid because Uni-C does not recognize the escaped d character within a dictionary.

```
./fsm -trace 2_strings.fsm

"\n\"\\\d"

sz " -> sost

sost \ -> soseq

soseq n -> scseq

scseq \ -> soseq

soseq " -> scseq

scseq \ -> soseq

soseq \ -> scseq

scseq \ -> soseq

soseq d -> bad

bad " -> bad
```

```
bad \n -> bad

bad EOF -> bad

NO
```

```
#11   It is not valid because we have not defined transition rules for Greek letters.

./fsm 2_strings.fsm

"Good morning people"

Segmentation fault (core dumped)
```

```
#12   Invalid because the literal is opened and closed with a single quote (') instead of a double ("").

./fsm 2_strings.fsm

'La vida loca'

NO
```

```
#13   It is valid because the literal starts and ends with double quotes(“)
./fsm 2_strings.fsm

"Hi Mark"

YES
```

```
#14   It is valid because it starts with double quotes(“) and the literal closes with double quotes(“)
./fsm 2_strings.fsm

"Tests"

YES
```

```
#15   It is valid because it starts with double quotes (“), contains a newline (\n), and the literal is closed with double quotes (“)
./fsm 2_strings.fsm
```

```
"Mark said, \"Boo!\"\n"

YES
```

```
#16   It is valid because it opens and closes with double quotes(“), and the newline is used with an escape sequence within the literal (\n)

./fsm 2_strings.fsm

"Hi/n\n"

YES
```

```
#17   Invalid because it breaks reading characters with EOF instead of a newline

./fsm -trace 2_strings.fsm

"hi" sz " -> sost

sost h -> sost

sost i -> sost

sost " -> scst

scst EOF -> bad

NO
```

2.7 Problems and solutions

We faced a problem again in the recognition of Greek characters, as there are literally words with Greek letters. No workaround was found and the result is similar which is detailed here . Therefore, Uni-C supports word literals only with Latin letters, digits, special characters and whitespace characters.

We also ran into a problem while coding the FSM, as when we went to run the code it gave us an error message saying the automaton is not deterministic.

```
./fsm 2_strings.fsm

fsm: in 2_strings.fsm, Non-Deterministic, state='sost', no input and input='"'
```

The FSM metatool only recognizes deterministic automata, however, the cause of the problem was not that we had more than one transition from one state to another with the same character. The cause was in the SOST (State Open String) state, which had these transitions when the problem occurred.

```
SOST: // --- Status for the content of the verbal ---

" -> SCST // The verb is enclosed in double quotes

\ -> SOSEQ // Escape sequence follows

* -> SOST // Any other character follows
```

The backslash character (\) indicates an escape sequence and obviously we needed the character itself to jump to SOSEQ (State Open Sequence). With the single backslash we simply declared an empty escape (no input), which caused the automaton to be non-deterministic.

The problem was addressed with the double backslash, because its use, like the escape characters (newline, tab, space), is done in escape order.

```
SOST: // --- Status for the content of the verb ---

" -> SCST // The verb is enclosed in double quotes

\\ -> SOSEQ // Escape sequence follows

* -> SOST // Any other character follows
```

2.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with Windows Subsystem Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 2_strings.fsm

./fsm -list 2_strings.fsm
```

```
./fsm -trace 2_strings.fsm
```

3. Integers

The simulation was undertaken by the students

Athanasiou Vasileios Evangelos (UNIWA-19390005) – Theocharis Georgios (UNIWA-19390283) – Tatsis Pantelis (UNIWA-20390226)

3.1 Identification standard

The function of the automaton is to recognize unsigned integers.

Specifically, it recognizes integers that belong to the decimal, hexadecimal, or octal numbering system. The decimal integer starts with a non-zero digit (1-9) and is optionally followed by any number of digits (0-9). 0 is recognized as the only exception to a decimal integer starting with 0. A hexadecimal integer is recognized when it starts with 0x or 0X followed by one or more hexadecimal digits (0-9 or AF). Finally, an octal integer is recognized when it starts with 0 and is followed by one or more octal (0-7) digits.

The template identifies them with the identifier name ***integer*** and they are returned as tokens.

3.2 Regular Expression

The regular expression for Uni-C integer literals is:

```
([1-9][0-9]*|0[x|X][0-9A-F]+|0[0-7]+|0)
```

The expression recognizes input strings that correspond to decimal, hexadecimal, and octal integers.

- **Decimals:** `[1-9][0-9]*|0`

  The expression recognizes strings that start with a character in the range “1-9” and may be followed by a character in the range “0-9” zero or more times. Additionally, it also recognizes 0 as the only 0-starting decimal integer exception.

- **Hexadecimal:** `0[x|X][0-9A-F]+`

  The expression recognizes strings starting with the character “0”, followed by either the character “x” or “X” and ending with a character in the range “0-9” or in the range “AF” one or more times.

- **Octal:** `0[0-7]+`

The expression recognizes strings starting with the character “0”, followed by the character in the range “0-7” one or more times.

3.3 Finite state automaton or DM

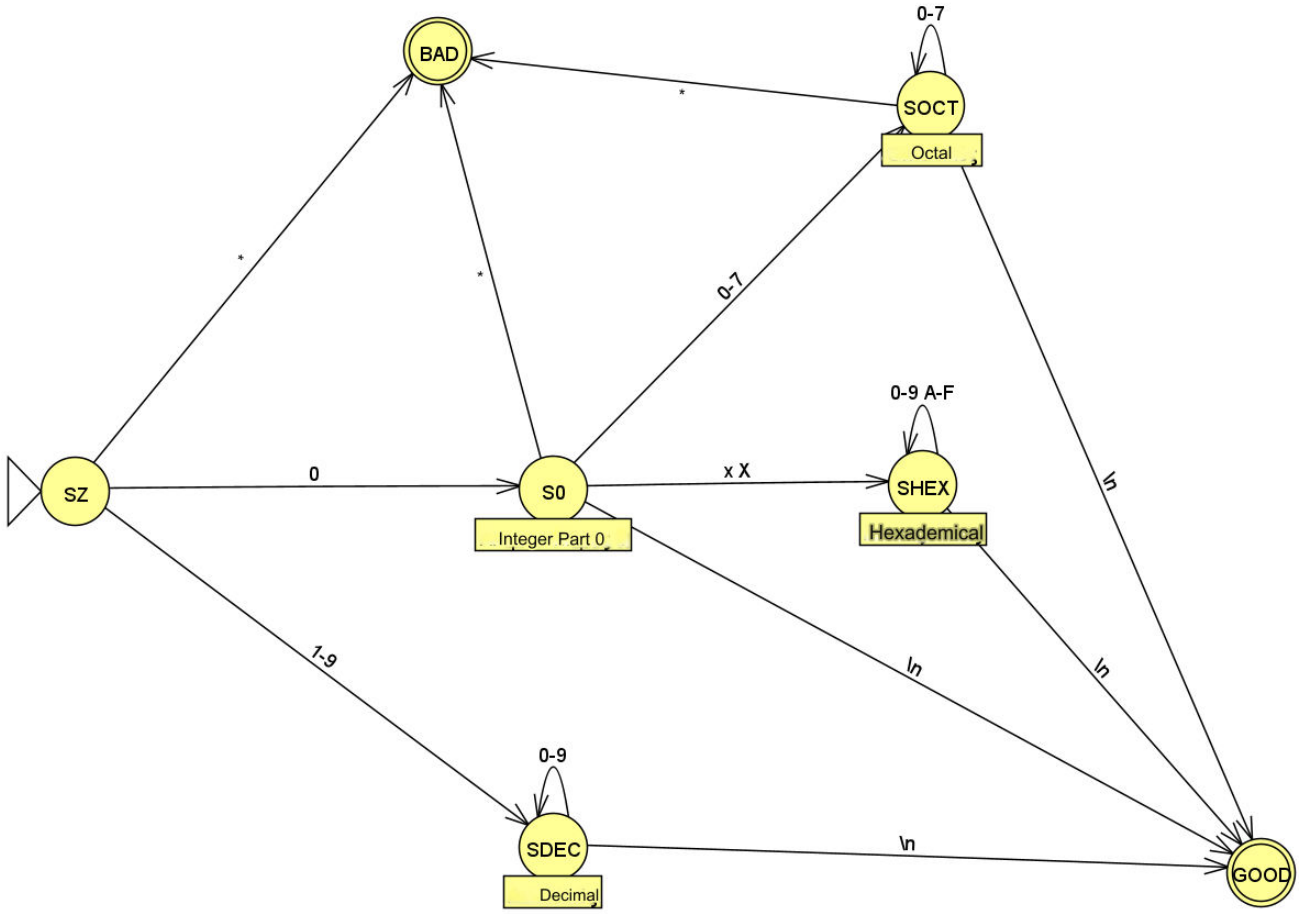


Figure 3.3.1 Uni-C finite integer automaton

This automaton recognizes valid integers that can be decimal, octal, or hexadecimal.

It is initially in the SZ state, waiting for the first symbol of the integer.

If the first symbol is 0, the automaton goes to state S0, checking subsequent symbols to determine whether the integer is octal or hexadecimal.

If the first character is from 1 to 9, the automaton goes into SDEC mode, since it is a decimal integer. It then waits for more decimal places or the end of the line to accept the integer.

In S0 states, SDEC checks the following symbols:

In the S0 state, if the next symbol is from 0 to 7, the number is octal. If the next symbol is x or X, then the number is hexadecimal. If it is the end of line, the number 0 is valid as the only decimal integer exception starting with 0, and the automaton goes to the GOOD state. In any other case, the number is not valid (BAD).

In SDEC mode, if the next symbol is from 0 to 9, the number continues to be decimal. If it is the end of line, the number is valid and the automaton goes to the GOOD state.

In SHEX and SOCT states, the automaton checks the following symbols:

In the SHEX mode of hexadecimal integers, if the next symbol is from 0 to 9 or the letters A to F, the automaton remains in the same state. If it is the end of line, the number is valid and the automaton goes to the GOOD state.

In SOCT state of octal integers, if the next symbol is from 0 to 7, the automaton remains in the same state. If it is the end of line, the number is valid and the automaton goes to the GOOD state.

3.4 Transition Table

	0	1-7	8-9	x X	AF	\n	EXIT STATUS
SZ	S0	SDEC	SDEC	BAD	BAD	BAD	NO
S0	SOCT	SOCT	BAD	SHEX	BAD	GOOD	NO
SDEC	SDEC	SDEC	SDEC	BAD	BAD	GOOD	NO
SHEX	SHEX	SHEX	SHEX	BAD	SHEX	GOOD	NO
SOCT	SOCT	SOCT	BAD	BAD	BAD	GOOD	NO
GOOD							YES
BAD							YES

Table 3.4.1 Transition table of Uni-C language integers

Status Acronym	Status Name	Status Information
SZ	State Z	The status for the 1st digit
S0	State 0	The condition that the integer starts with 0
SDEC	State Decimal	The condition that recognizes integers in the decimal system

SHEX	State Hexadecimal	The state that recognizes integers in hexadecimal
SOCT	State Octal	The state that recognizes integers in octal

Table 3.4.2 State table of Uni-C language integers

3.5 Encoding via FSM

The FSM encoding of the Uni-C integer automaton can be found in the source file **3_integers.fsm**

```
// ================== 3_Integers ==================

START = SZ // ==== Initial state is SZ ====

SZ: // --- [State Z] State for integers ---
0 -> S0 // Integer starting with 0
1-9 -> SDEC // Positive decimal integer
* -> BAD // The integer is invalid

S0: // --- [State 0] State for integers starting with 0 ---
0-7 -> SOCT // Octal integer
x X -> SHEX // Hexadecimal integer
\n -> GOOD // Valid integer 0
* -> BAD // The integer is invalid

SDEC: // --- [State Decimal] Positive Decimal Integer State ---
0-9 -> SDEC // More digits from 0-9 follow
\n -> GOOD // Integer is valid
* -> BAD // The integer is invalid

SHEX: // --- [State Hexadecimal] State Hexadecimal ---
```

```
0-9 AF -> SHEX // More digits from 0-9 and letters from AF follow

\n -> GOOD // Integer is valid

* -> BAD // The integer is invalid


SOCT: // --- [State Octal] State Octal Integer ---

0-7 -> SOCT // More digits from 0-7 follow

\n -> GOOD // Integer is valid

* -> BAD // The integer is invalid


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

3.6 Audit Cases

```
#1 0 is an accepted integer state in the Uni-C language, it is entered with \n
so the automaton correctly accepts the following input.

./fsm 3_integers.fsm

0

YES
```

```
#2 3 is an accepted integer state in Uni-C, it is entered with \n so the
automaton correctly accepts the following input.

./fsm 3_integers.fsm

3

YES
```

```
#3 The following input we see has only an integer part and no decimal, so it is correctly accepted by the automaton once it is the representation of an integer in the Uni-C language.

./fsm 3_integers.fsm

214748

YES
```

```
#4 In the Uni-C language, numbers are unsigned, so the automaton correctly does not accept the input -50.

./fsm 3_integers.fsm

-50

NO
```

```
#5 Because the automaton recognizes strings that start with 0 followed by the character x and are in the range [0-F], it correctly accepts this particular input representing a hexadecimal number.

./fsm 3_integers.fsm

0x4F

YES
```

```
#6  Because the automaton recognizes strings that start with 0 followed by the X character and are in the range [0-F], it correctly accepts this particular input representing a hexadecimal number.

./fsm 3_integers.fsm

0X88AA

YES
```

```
#7 The automaton accepts the following string as it is an integer in the 8-bit numbering system. We notice that all the digits entered are within [0-7], i.e. they follow the rules of the 8-digit system.

./fsm 3_integers.fsm
```

```
063

YES
```

```
#8 The machine, as we saw in the previous example, accepts numbers of the 8-digit numbering system. For this reason the input '00' is acceptable.

./fsm -trace 3_integers.fsm

00

sz 0 -> s0

s0 0 -> soct

soct \n -> good

YES
```

```
#9 The input '01' is again an 8-digit number system integer, as long as there is no sign or digit outside the range [0-7], the input is valid.

./fsm -trace 3_integers.fsm

01

YES
```

```
#10 Input '09' is not accepted by the automaton as the number 9 is not contained in the numbers [0-7].

./fsm -trace 3_integers.fsm

09

sz 0 -> s0

s0 9 -> bad

bad \n -> bad

bad EOF -> bad

NO
```

#11 The automaton rejects the following input because the 'G' entered is outside the bounds of the hexadecimal numbering system, which contains numbers from [0-9] or letters from [AF]. It does not represent a hexadecimal number in the Uni-C language.

```
./fsm -trace 3_integers.fsm

0XFGA9

sz 0 -> s0

s0 X -> shex

shex F -> shex

shex G -> bad

bad A -> bad

bad 9 -> bad

bad \n -> bad

bad EOF -> bad

NO
```

#12 The following string is rejected by the automaton due to the input of the number '8', because it is not a number within [0-7] to be a valid octal integer.

```
./fsm 3_integers.fsm

01578

NO
```

#13 The string starts with 2 zeros, so it does not correctly represent a 16-bit number in the Uni-C language. For this reason the string is automatically rejected.

```
./fsm 3_integers.fsm

00xAFB1

NO
```

#14 The Uni-C language in the integer representation does not accept signed numbers, so the following string will not be accepted by automaton because of the sign.

```
./fsm 3_integers.fsm

-001

NO
```

#15 The Uni-C language in the integer representation does not accept signed numbers, so the following string will not be accepted by automaton because of the sign.

```
./fsm 3_integers.fsm

-0xAF01

NO
```

#16 The f numbers in the string below are written in lowercase letters. For the correct representation of a 16-digit number they should be written in capital letters. Therefore the string is not accepted by automaton.

```
./fsm 3_integers.fsm

0xff23

NO
```

#17 In the Uni-C language there is no representation of signed integers, so due to the sign the following input will be rejected by the automaton.

```
./fsm 3_integers.fsm

+01

NO
```

#18 In the Uni-C language there is no representation of signed integers, so due to the sign the following input will be rejected by the automaton.

```
./fsm 3_integers.fsm

+56
```

```
NO
```

```
#19 There cannot be an integer representation when the string contains a sign. For this reason, the automaton rejects the following string.

./fsm 3_integers.fsm

0x-FF

NO
```

```
#20 The string is not accepted as the newline character '\n' was not used to enter it into the automaton, instead EOF was used and therefore we do not get any positive result from the automaton.

./fsm -trace 3_integers.fsm

619 sz 6 -> sdec

sdec 1 -> sdec

sdec 9 -> sdec

sdec EOF -> bad

NO
```

3.7 Problems and solutions

We encountered a problem in the design of the regular expression, as the initial estimate was this.

```
(0|[1-9][0-9]*|0[x|X][0-9A-F]+|0[0-7]+)
```

The recognition results from the Regexpal tool were as follows:

```
0

0 0 7

10

0 x 10
```

A regular expression is essentially 4 individual regular expressions in a logical expression with an OR operator. The order in which the rules check the input string is from left to right.

Therefore, the regular expression did not recognize hexadecimal and octal integers because the first rule they followed was the rule of 0 as the only exception for a decimal integer starting with 0.

The problem was fixed by changing the regular expression.

```
([1-9][0-9]*|0[x|X][0-9A-F]+|0[0-7]+|0)
```

The rule of 0 as the only decimal integer exception starting with 0 is checked last, so that the additional rules that apply to hexadecimal and octal integers respectively are checked first.

3.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

Bad outputs (BAD) were not plotted in the transition diagram in all states (specifically they were not plotted in the SDEC, SHEX states) for the sake of diagram readability. However, illegal exits were encoded in the FSM and noted in the transition table in all states.

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 3_integers.fsm

./fsm -list 3_integers.fsm

./fsm -trace 3_integers.fsm
```

4. Floating point numbers

The simulation was undertaken by the students

Athanasiou Vasileios Evangelos (UNIWA-19390005) – Iliou Ioannis (UNIWA-19390066)

4.1 Recognition Standard

Auto recognizes floating-point literals of the Uni-C language based on a recognition pattern.

A decimal number consists of the integer part and the decimal part separated by a dot '.'. Both the integer part and the decimal part are defined according to the rules described here for the simple integers of the decimal number system. It is also possible to define forces using the character 'e' or 'E'. In this case the integer and/or decimal part is raised to the integer power following the character 'e' or 'E'.

The template identifies them with the identifier name ***float*** and they are returned as a token.

4.2 Regular Expression

The regular expression for Uni-C floating-point number literals is:

```
(?:[1-9][0-9]*|0)(?:\.(?:[1-9][0-9]*|0*[1-9]+))?(? :[eE](?:-?[1-9][0-9]*|0))?
```

The expression recognizes the following number formats:

- An integer with one or more numeric digits (the rule defined in the template also applies integers).
- An integer that may include a decimal, followed by one or more numeric digits.
- An integer in the role of the base that may include an exponent (integer exponent with the same rule as defined in the pattern of of integers) for the force representation.
- An integer that can include a decimal part, followed by one or more numeric digits, and the decimal part can include an exponent to represent a power.

The expression is parsed as follows:

```
(?:[1-9][0-9]*|0)
```

This part corresponds to the integer part of the number. It is broken down as follows:

```
?:
```

This part indicates a non-recursive group, that is, a group without storing its result.

```
[1-9][0-9]*
```

This corresponds to a non-zero digit followed by zero or more numeric digits.

```
|
```

This indicates switching between the two parts of the expression. In this case, either the number is positive or it is zero.

```
0
```

It corresponds to zero.

```
(?:\.(?:[1-9][0-9]*|0*[1-9]+))?
```

This part corresponds to the decimal part of the number. It is broken down as follows:

```
?:
```

Again, this indicates a non-recursive group.

```
\.
```

Matches the decimal point.

```
?:
```

Non-recursive group.

```
[1-9][0-9]*
```

This corresponds to a non-zero digit followed by zero or more numeric digits.

```
|
```

This indicates switching between the two parts of the expression. In this case, either the number is positive or it is zero.

```
0*[1-9]+
```

This corresponds to the digit 0 zero or more times followed by a digit from 1-9 one or more times.

```
?
```

Matches zero or one time indicating that the presence of a decimal part is optional.

```
(?:[eE](?:-?[1-9][0-9]*|0))?
```

This part corresponds to the scientific form exponent (if any). It is broken down as follows:

```
?:
```

Again, non-recursive group.

```
[e]
```

It corresponds to the letter "e" or "E", which is used for the scientific form of the representation of the power exponent.

```
(?:-?[1-9][0-9]*|0)
```

Matches the number after "e" or "E". It is broken down as follows:

```
-?
```

Matches zero or a negative sign if present.

```
[1-9][0-9]*
```

Matches a non-zero digit followed by zero or more numeric digits.

```
|
```

Toggle between the two options.

```
0
```

It corresponds to zero.

In summary, this regular expression corresponds to numbers that can be integers, decimals, or powers in scientific form.

4.3 Finite state automaton or DM

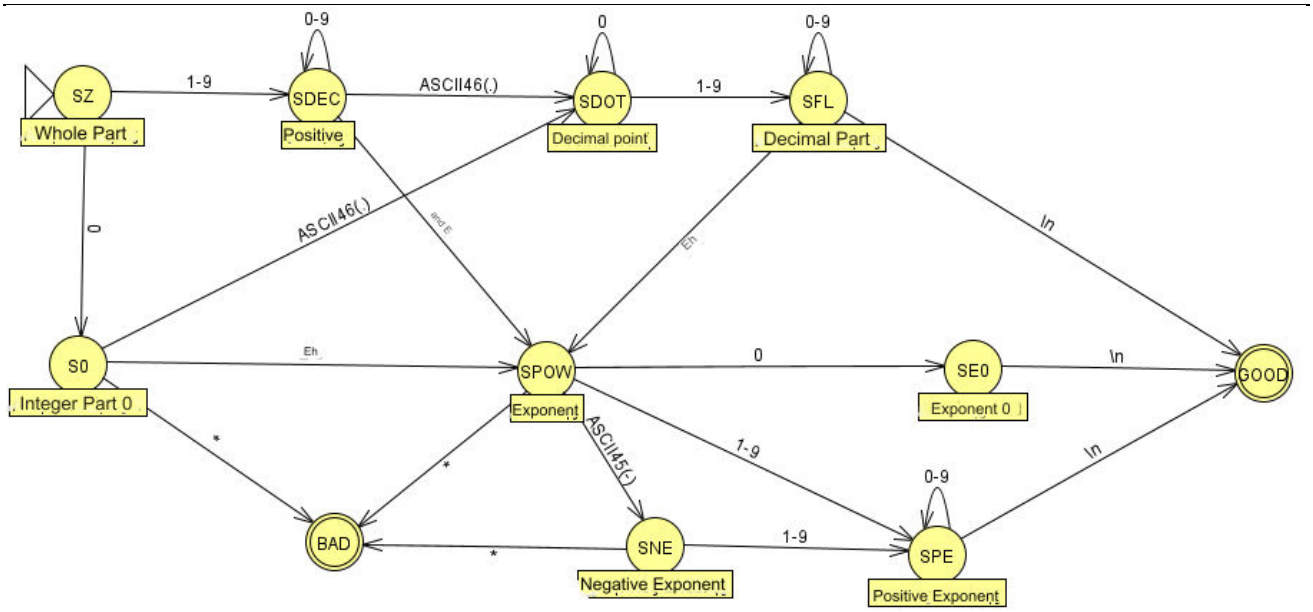


Figure 4.3.1 Finite floating-point automaton of the Uni-C language

The automaton recognizes the structure of a number, which can consist of three parts: the integer part, the decimal part and the power. The sequence of states is illustrated as follows:

- The initial state is SZ, which corresponds to the integer part of the number.
- If it starts with "0", we go to state S0.

- If it starts with a digit from 1 to 9, we go to the S DEC state for a positive integer.
- The integer may be followed by the decimal part, which is recognized by the SDOT and SFL states.
- If followed by a dot, we go to SDOT status.
- If "E" or "e" follows, we go into SPOW mode to recognize the power of the number.
- SDOT and SFL states recognize the decimal part.
- Power is identified by the SPOW, SE0, SNE and SPE states.

The number is valid when it reaches the GOOD exit state, meaning that the integer part, the decimal part (if any), and the power of the number have been recognized, and then the end of line (\n) follows. If any invalid symbol is encountered, the number is considered invalid.

4.4 Transition Table

	-	0	1-9	.	Eh	\n	EXIT STATUS
SZ	BAD	S0	S DEC	BAD	BAD	BAD	NO
S0	BAD	BAD	BAD	SDOT	SPOW	BAD	NO
S DEC	BAD	S DEC	S DEC	SDOT	SPOW	BAD	NO
SDOT	BAD	SDOT	SFL	BAD	BAD	BAD	NO
SFL	BAD	SFL	SFL	BAD	SPOW	GOOD	NO
SPOW	SNE	SE0	SPE	BAD	BAD	BAD	NO
SNE	BAD	BAD	SPE	BAD	BAD	BAD	NO
SPE	BAD	SPE	SPE	BAD	BAD	GOOD	NO
SE0	BAD	BAD	BAD	BAD	BAD	GOOD	NO
GOOD							YES
BAD							YES

Table 4.4.1 Transition table of Uni-C floating point numbers

Status Acronym	Status Name	Status Information

SZ	State Z	The status for the 1st digit
S0	State 0	The condition that the number starts with 0 in the integer part
SDEC	State Decimal	The condition that recognizes integers in the decimal system for the integer part
SDOT	State Dot	The state that recognizes the decimal point
SFL	State Float	The condition that recognizes integers in the decimal system for the decimal part
SPOW	State Power	The condition that recognizes the exponent in scientific form (E or e) for the force reading
SNE	State Negative Exponential	The condition that recognizes a negative sign in the exponent
SPE	State Positive Exponential	The condition that recognizes a positive integer in the exponent
SE0	State Exponential 0	The condition that recognizes 0 in the exponent

Table 4.4.2 State table of Uni-C floating point numbers

4.5 Encoding via FSM

The FSM encoding of the Uni-C floating-point automaton can be found in the source file **4_floats.fsm**

```
// ================= 4_Floating Point Numbers ==================
```

```
START = SZ // ==== The initial state is SZ ====


SZ: // --- [State Z] State for the integer part ---

0 -> S0 // The integer part is 0

1-9 -> SDEC // The integer part starts from 1-9

* -> BAD // The number is invalid


S0: // --- [State 0] State for 0 as integer part ---

. -> SDOT // The integer part is 0 followed by a decimal point for
the decimal part

E e -> SPOW // The power base is 0 followed by the
scientific form of the exponent

* -> BAD // The number is invalid


SDEC: // --- [State Decimal] State for the integer part ---

. -> SDOT // Decimal point follows

0-9 -> SDEC // More digits in the integer part from 0-9 follow

E e -> SPOW // The scientific form of the exponent follows

* -> BAD // The number is invalid


SDOT: // --- [State Dot] State for the decimal point ---

0 -> SDOT // Digit 0 follows

1-9 -> SFL // Digits for the decimal part from 1-9 follow

* -> BAD // The number is invalid


SFL: // --- [State Float] State for the decimal part ---

0-9 -> SFL // More digits follow for the decimal part from 0-9

E e -> SPOW // The scientific form of the exponent follows

\n -> GOOD // The number is valid

* -> BAD // The number is invalid


SPOW: // --- [State Power] State for powers ---
```

```
- -> SNE // Negative sign for the exponent follows

0 -> SE0 // Exponent is 0

1-9 -> SPE // Followed by positive exponent

* -> BAD // The number is invalid


SNE: // --- [State Negative Exponential] State for negative
exponent ---

1-9 -> SPE // Followed by digits 1-9 for the negative exponent

* -> BAD // The number is invalid


SPE: // --- [State Positive Exponential] State for Positive Exponential
---

0-9 -> SPE // Followed by digits 1-9 for the positive exponent

\n -> GOOD // The number is valid

* -> BAD // The number is invalid


SE0: // --- [State Exponential 0] State for digit 0 for
exponent to power ---

\n -> GOOD // The number is valid

* -> BAD // The number is invalid


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

4.6 Audit Cases

```
#1    It is valid because the integer part starts from 1-9 (SDEC), the exponent
form follows (SPOW), and the exponent is 8 (SPE). Follows the rules of integers
in both base and exponent.

./fsm 4_floats.fsm
```

9e8

YES

#2 It is valid because the integer part starts from 1-9 (SDEC), the exponent form follows (SPOW), contains a negative sign for the exponent (SNE) and the negative exponent is 8. It follows the rules of integers and in base and exponent.

./fsm 4_floats.fsm

9E-8

YES

#3 It is valid because the integer part starts from 1-9 (SDEC), followed by a decimal point for the decimal part (SDOT), and followed by digits for the decimal part from 0-9 (SFL). Follows the integer rules in both the integer and decimal parts.

./fsm 4_floats.fsm

3.14

YES

#4 It is not valid because 10 can be represented as an integer, so representing it in decimal form is unnecessary.

./fsm -trace 4_floats.fsm

10.0

sz 1 -> sdec

sdec 0 -> sdec

sdec . -> sdot

sdot 0 -> sdot

sdot \n -> bad

bad EOF -> bad

NO

#5 It is valid because the integer part is 10 (SDEC), there is a decimal point for the decimal part (SDOT), and the decimal part contains the digits 001 (SFL). Follows the integer rules in both the integer and decimal parts.

```
./fsm -trace 4_floats.fsm

10.0001

sz 1 -> sdec

sdec 0 -> sdec

sdec . -> sdot

sdot 0 -> sdot

sdot 0 -> sdot

sdot 0 -> sdot

sdot 1 -> sfl

sfl \n -> good

YES
```

#6 It is valid because the integer part starts from 1-9 (SDEC), the exponent form follows (SPOW), contains a negative sign for the exponent (SNE), and the negative exponent is 15 (SPE). Follows the rules of integers in both base and exponent.

```
./fsm -trace 4_floats.fsm

5e-15

sz 5 -> sdec

sdec e -> spow

spow -> sne

sne 1 -> spe

spe 5 -> spe

spe \n -> good

YES
```

#7 It is valid because the integer part starts from 1-9 (SDEC), the exponent form follows (SPOW), the positive exponent follows (SPE), and the exponent is 100 (SPE). Follows the rules of integers in both base and exponent.

```
./fsm 4_floats.fsm

1e100

YES
```

#8 It is valid because the integer part starts from 1-9 (SDEC), followed by a decimal point for the decimal part (SDOT), the decimal part contains the number 1 (SFL), followed by the exponent form (SPOW) and the exponent is 0 (SE0). Follows the rules of integers in both the whole part and the decimal part.

```
./fsm -trace 4_floats.fsm

3.1E0

sz 3 -> sdec

sdec . -> sdot

sdot 1 -> sfl

sfl E -> spow

spow 0 -> se0

se0 \n -> good

YES
```

#9 It is valid because the integer part is 0 (S0), the exponent form follows (SPOW), and the exponent is 0 (SE0). Follows the integer rules in both the integer and decimal parts.

```
./fsm 4_floats.fsm

0e0

YES
```

#10 Invalid because the integer part does not support signed integers according to the Uni-C integer recognition standard. However Uni-C only supports signed integers in the power exponent (e-100).

```
./fsm 4_floats.fsm

-5.1e-100

NO
```

```
#11   Invalid because 0 is not a floating point number.

./fsm 4_floats.fsm

0

NO
```

```
#12   Invalid because the integer part does not support signed integers according to the Uni-C integer recognition standard.

./fsm 4_floats.fsm

-0.5

NO
```

```
#13   It is not valid because it does not follow the rule for the decimal part, as there is no decimal point (SFL) after the decimal point (SDOT). It also does not follow the rule for the form of the exponent, as the power base does not have a digit from 0-9 (SPOW).

./fsm 4_floats.fsm

5.e1

NO
```

```
#14   Not valid even though the power format looks compatible with a digit in the base and a digit in the exponent. The reason is because the scientific form of the exponent, i.e. E, is a Greek character and not a Latin one. The problem with the Greek characters is analyzed here .

./fsm -trace 4_floats.fsm

1 E 1

sz 1 -> sdec
```

```
sdec \316 -> bad

Segmentation fault (core dumped)
```

#15 It is not valid because the integer part does not follow the rule of integers starting with a digit from 1-9. 0 is the only exception for a decimal integer starting with 0, so the input 00 is not compatible with the Uni-C standard.

```
./fsm 4_floats.fsm

00.01

NO
```

#16 is not valid because the exponent does not follow the integer rule of starting with a digit from 1-9. 0 is the only exception for a decimal integer starting with 0, so the input 01 is not compatible with the Uni-C standard.

```
./fsm -trace 4_floats.fsm

0e01

sz 0 -> s0

s0 e -> spow

spow 0 -> se0

se0 1 -> bad

bad \n -> bad

bad EOF -> bad

NO
```

#17 Invalid because Uni-C does not support such force formats. After an exponent character in scientific form (SPOW) only a decimal integer can follow (SPE), or negative (SNE) or 0 (SE0).

```
./fsm 4_floats.fsm

5e1.5e1

NO
```

#18 Invalid because it does not follow the rule for the exponent, as Uni-C only supports integer decimals and not floating point numbers.

```
./fsm 4_floats.fsm

1E1.2

NO
```

#19 Invalid because the character that leads to the allowed exit state is EOF (End-of-File) and not newline (\n).

```
./fsm -trace 4_floats.fsm

0.1 sz 0 -> s0

s0. -> sdot

sdot 1 -> sfl

sfl EOF -> bad

NO
```

#20 It is not valid because it does not follow the rule for the decimal part, as instead of the character “.” “,” is used (SDOT) and there is no continuation after the decimal point “,” to give the decimal part (SFL).

```
./fsm 4_floats.fsm

6.20

NO
```

#21 It is not valid because 10 can be represented as an integer, so representing it in decimal form is unnecessary.

```
./fsm -trace 4_floats.fsm

0.0

sz 0 -> s0

s0. -> sdot

sdot 0 -> sdot
```

```
sdot \n -> bad

bad EOF -> bad

NO
```

4.7 Problems and solutions

We encountered a problem in the design of the regular expression, as the initial estimate was this.

```
(?:[1-9][0-9]*|0)(?:\. \d +)?(?:[eE](?:-?[1-9][0-9]*| 0))?
```

The recognition results from the Regexpal tool were as follows:

```
0

0.0

0.00

5

5.0

5.00
```

The regular expression also correctly recognized integers as defined by the integer standard, however, 0 does not exist to be recognized in decimal form (0.0). The problem was addressed in the **\d expression** which recognizes any digit from 0-9. We replaced it with this regular expression.

```
(?:[1-9][0-9]*|0*[1-9]+)
```

The recognition results from the Regexpal tool were as follows:

```
0

0 . 0

5

5 . 0

0.01

5.01
```

Now, the regular expression does not recognize 0.0 but also any integer in decimal form (5.0). We chose not to extend the regular expression to recognize any integer in decimal form other than 0, so as not to make the regular expression too large. Additionally it would create a problem

with the readability of the transition diagram. In short, integers are recognized, just not in their decimal form.

4.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

Bad Outputs (BAD) were not plotted in the transition diagram in all states (specifically they were not plotted in SZ, SDEC, SDOT, SFL, SE0, SPE states) for the sake of diagram readability. However, illegal exits were encoded in the FSM and noted in the transition table in all states.

FSM code that does not recognize integers (5, 5.0 etc), as it is covered in the single .

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 4_floats.fsm

./fsm -list 4_floats.fsm

./fsm -trace 4_floats.fsm
```

5. Comments

The simulation was undertaken by the students

Theocharis Georgios (UNIWA-19390283) – Dominaris Vasileios (UNIWA-21390055)

5.1 Recognition Standard

The function of the automaton is to recognize Uni-C comments based on a recognition pattern.

The standard recognizes two kinds of comments: single-line comments and multi-line comments. A single-line comment begins with two consecutive '/' characters (ie //), which are not part of the comment string, and ends with the end-of-line (\n). A multi-line comment starts with /* characters and ends only with */ characters, regardless of the number of intervening lines.

comment identifier but they are not returned as tokens (they are ignored by the parser).

5.2 Regular Expression

The regular expression for Uni-C language comment units is:

```
(?:\/\/.*\n|\/\*[\s\S]*?\*\/)
```

The regular expression is parsed as follows:

- `(?: ... )`

  Non-Recursive Grouping. This means it does not create a group record.

- `\/\/.*\n`

  This part matches comments starting with // and extending to the end of the line (\n).

- `|`

  It symbolizes the alternation, that is, either one or the other part can match.

- `\/\*[\s\S]*?\*\/`

This part matches comments starting with /* and ending with */. [\s\S] matches any character, including spaces and newlines, so it can match multiple lines of comments.

In summary, this regular expression matches comments in code, whether they are single lines starting with //, or multiple lines starting with /* and ending with */.

5.3 Finite state automaton or DM

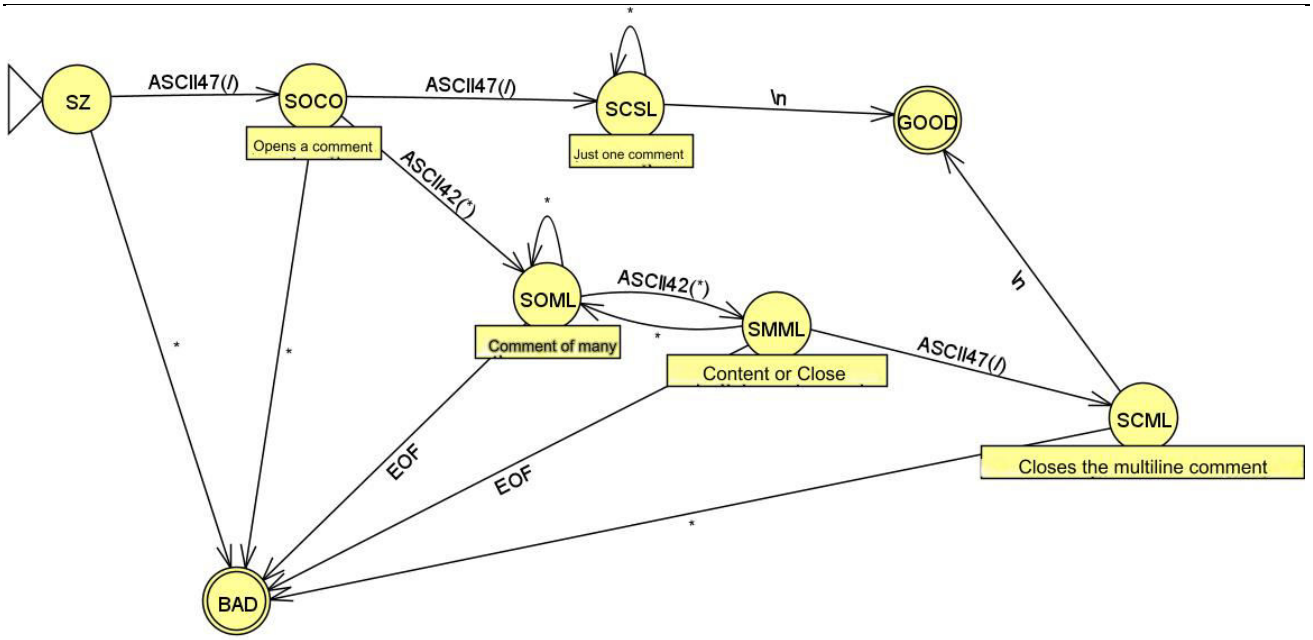


Figure 5.3.1 Uni-C finite comment automaton

The automaton above describes how a program that detects comments in Uni-C code works. Initially, the initial state is SZ, which represents waiting for a comment to be opened. When the / symbol is encountered, the automaton goes into SOCO state to check whether it is a single-line comment or a multi-line comment. If any other symbol is found in the initial state, the automaton switches to the BAD state, replacing the comment as invalid.

The SOML and SMML states control multiline comments, switching from SOML to SMML when another * is found, and from SMML to SCML when a / after a * is found, indicating the end of the multiline comment.

The SCML and SCSL states represent the end of a valid multi-line or single-line comment respectively. The automaton returns to the initial state after a valid comment.

Finally, the GOOD status indicates that a comment has been successfully detected.

5.4 Transition Table

	/	*	EOF	\n	Other	EXIT

					Character	STATUS
SZ	SOCO	BAD	BAD	BAD	BAD	NO
SOCO	SCSL	SOML	BAD	BAD	BAD	NO
SOML	SOML	SMML	BAD	SOML	SOML	NO
SMML	SCML	SOML	BAD	SOML	SOML	NO
SCML	BAD	BAD	BAD	GOOD	BAD	NO
SCSL	BAD	BAD	BAD	GOOD	BAD	NO
GOOD						YES
BAD						YES

Table 5.4.1 Transition table of Uni-C language comments

Status Acronym	Status Name	Status Information
SZ	State Z	The status for opening a comment
SOCO	State Open Comment	The status for opening a single-line or multi-line comment
SOML	State Open Multiple Line (Comment)	The state for opening a multi-line comment and reading content
SMML	State Mid Multiple Line (Comment)	The state for closing a multi-line comment or reading content
SCML	State Close Multiple Line (Comment)	The status for closing the multiline comment
SCSL	State Close Single Line (Comment)	The status for closing a single line comment or reading content

Table 5.4.2 State table of Uni-C language comments

5.5 Encoding via FSM

The FSM encoding of the Uni-C comment automaton is in the source file **5_comments.fsm**

```
// ================== 5_Comments ==================


START = SZ // ==== The initial state is SZ ====


SZ: // --- [State Z] State for opening comment ---

/ -> SOCO // Opens comment

* -> BAD // Comment is invalid


SOCO: // --- [State Open Comment] State Open Comment ---

/ -> SCSL // Opens a single line comment

\* -> SOML // Opens a multiline comment

* -> BAD // Comment is invalid


SOML: // --- [State Open Multiple Line (Comment)] State comment many
                                                                    lines ---

    \* -> SMML // Intent to close the comment

* -> SOML // Comment content

EOF -> BAD // Comment is invalid


SMML: // --- [State Mid Multiple Line (Comment)] State comment many
                                                                   lines ---

    / -> SCML // Closes multiline comment

* -> SOML // Return to state for comment content

EOF -> BAD // Comment is invalid


SCML: // --- [State Close Multiple Line (Comment)] Status valid
                                                                     comment many lines
---

    \n -> GOOD // Comment is valid

* -> BAD // Comment is invalid
```

```
SCSL: // --- [State Close Single Line (Comment)] Status valid comment
                                                              one line ---

    \n -> GOOD // Comment is valid

* -> SCSL // Comment is invalid


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

5.6 Audit Cases

```
#1    The input is accepted by the automaton as it is a single comment, starts with '//' and ends with '\n'.

./fsm 5_comments.fsm

// hello world!!

YES
```

```
#2    Since the string does not start with '//' or '/*' it cannot be a comment in the Uni-C language.

./fsm 5_comments.fsm

/hello world!!/

NO
```

```
#3    Comments in the Uni-C language must start with either '//' or '/*', so the following string is not accepted by automaton. The following comment is supported in the Python language.

./fsm 5_comments.fsm

# hello world!
```

```
NO
```

#4 The following string is not accepted by automaton because the character 'o' inserted before '\n' is a Greek character. Uni-C does not support Greek characters.

```
./fsm -trace 5_comments.fsm

// hell o

sz / -> soco

soco / -> scsl

scsl \s -> scsl

scsl h -> scsl

scsl e -> scsl

scsl l -> scsl

scsl l -> scsl

scsl \316 -> soml

soml \277 -> soml

soml \n -> soml

soml EOF -> bad

NO
```

#5 This input is not accepted by the automaton, as there is no special character to indicate a comment. The string does not exist as a comment.

```
./fsm 5_comments.fsm

hello world

NO
```

#6 The following string is enclosed by the special characters '/* ' *and* '* /' before '\n' is given, so it is a comment and is automatically accepted.

```
./fsm 5_comments.fsm
```

```
/* hello world!! */

YES
```

```
#7    The string is accepted by automaton as a comment as it starts with the special characters '//' and is followed by no Greek letter or missing '\n'.

./fsm 5_comments.fsm

//hello//world//!!

YES
```

```
#8    The string is not accepted by automaton because instead of the special characters '//' that define a comment there are wrongly the special characters '\\'.

./fsm 5_comments.fsm

\\ hello world!!

NO
```

```
#9    The string is enclosed by the special characters '/*' and '*/' therefore it is a comment and is therefore accepted by the automaton.

./fsm 5_comments.fsm

/*

*hello

* world

* !!

 */

YES
```

```
#10   The string starts with the special characters '/*' and ends with the special characters '*/' and the special character '/n'. For this reason it is a comment and is automatically accepted.

./fsm -trace 5_comments.fsm
```

```
/*

sz / -> soco

soco * -> soml

soml \s -> soml

soml \n -> soml

hi

soml h -> soml

soml i -> soml

soml \n -> soml

*/

soml * -> smml

smml / -> scml

scml \n -> good

YES
```

#11 Instead of the special character '/' in this string the special character '\' is used. For this reason, it is not a valid comment quote and is automatically rejected.

```
./fsm 5_comments.fsm

Hello world! *\

NO
```

#12 The '/*' special characters in the following string are not followed by the '*/' special characters so there is no proper comment quotation. The string is automatically discarded.

```
./fsm 5_comments.fsm

/*//

//hi

NO
```

#13 As long as the string starts with the special characters '//', does not contain any forbidden character in between and ends with '\n' it is a valid Uni-C single line comment. Auto accepts the string.

```
./fsm 5_comments.fsm

//* hi *//

YES
```

#14 The string does not have any special character found in Uni-C comments so it is not a comment and is automatically rejected. Commenting is supported in the Matlab language.

```
./fsm 5_comments.fsm

%% hi

NO
```

#15 The string despite being a single line comment is rejected by the automaton. This is because while it starts with the special characters '//', it does not end with '\n' so that it was a comment. Instead, it ends with the last character EOF.

```
./fsm -trace 5_comments.fsm

// hi sz / -> soco

soco / -> scsl

scsl \s -> scsl

scsl h -> scsl

scsl i -> scsl

scsl EOF -> scsl

NO
```

#16 The following string is accepted by the automaton because it starts with the special characters '//' and ends with '\n' without any forbidden character occurring in between.

```
./fsm 5_comments.fsm

// hi */

YES
```

```
#17   The string starts with the special characters '/*' and to be a comment it must also end with the special characters '*/'. This is not the case and the string is rejected by automaton.

./fsm 5_comments.fsm

/* hi //

NO
```

```
#18 The string is not accepted by the automaton to be a Uni-C comment, as the comment content is in Greek letters. The problem with Greek characters is described here .

./fsm -trace 5_comments.fsm

// This is a comment

sz / -> soco

soco / -> scsl

scsl \s -> scsl

scsl \316 -> soml

fsm: in 5_comments.fsm, state 'soml' input \221 not accepted
```

5.7 Problems and solutions

We faced a problem again in the recognition of Greek characters, as there are comments with Greek letters. No workaround was found and the result is similar to what is detailed here . Therefore, Uni-C only supports comments with Latin letters, digits, special characters and whitespace characters.

Also, we encountered a problem in the coding through FSM and mainly in the BAD output state, as, from the beginning, we did not have an outgoing metabase from BAD and in the automata of the aforementioned Uni-C word units. Therefore, we did not see the result of NO on all disallowed strings, but the following result from an illustrative example.

```
./fsm 5_comments.fsm
```

```
this is a comment

fsm: in 5_comments.fsm, state 'bad' input h not accepted
```

The problem was solved by adding the following transition to the FSM code.

```
BAD: // ==== Exit status: OXI ====

* -> BAD
```

So now the result of the disallowed string is the following.

```
./fsm 5_comments.fsm

this is a comment

NO
```

5.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 5_comments.fsm

./fsm -list 5_comments.fsm

./fsm -trace 5_comments.fsm
```

6. White_spaces characters

The simulation was undertaken by the students

Dominaris Vasileios (UNIWA-21390055) – Tatsis Pantelis (UNIWA-20390226)

6.1 Recognition Standard

The function of the automaton is to recognize Uni-C white_spaces characters based on a recognition pattern.

The standard recognizes sequences of separator characters (spaces, tabs or combinations thereof), used to separate lexicons.

The template also recognizes them with the identifier name ***white_spaces*** but they are not returned as tokens (they are ignored by the parser).

6.2 Regular Expression

The regular expression for the whites_spaces Uni-C character lexical units is:

```
\s+
```

Uni-C whitespace characters are space (\s), tab (\t), newline character (\n), and end of file (EOF). A regular expression that recognizes these whitespace characters is \s+. The regular expression represents a space character, tab symbol, or any other whitespace character, and is repeated one or more times. In other words, it matches any number of one or more whitespace characters, regardless of how many whitespace characters are together.

6.3 Finite state automaton or DM

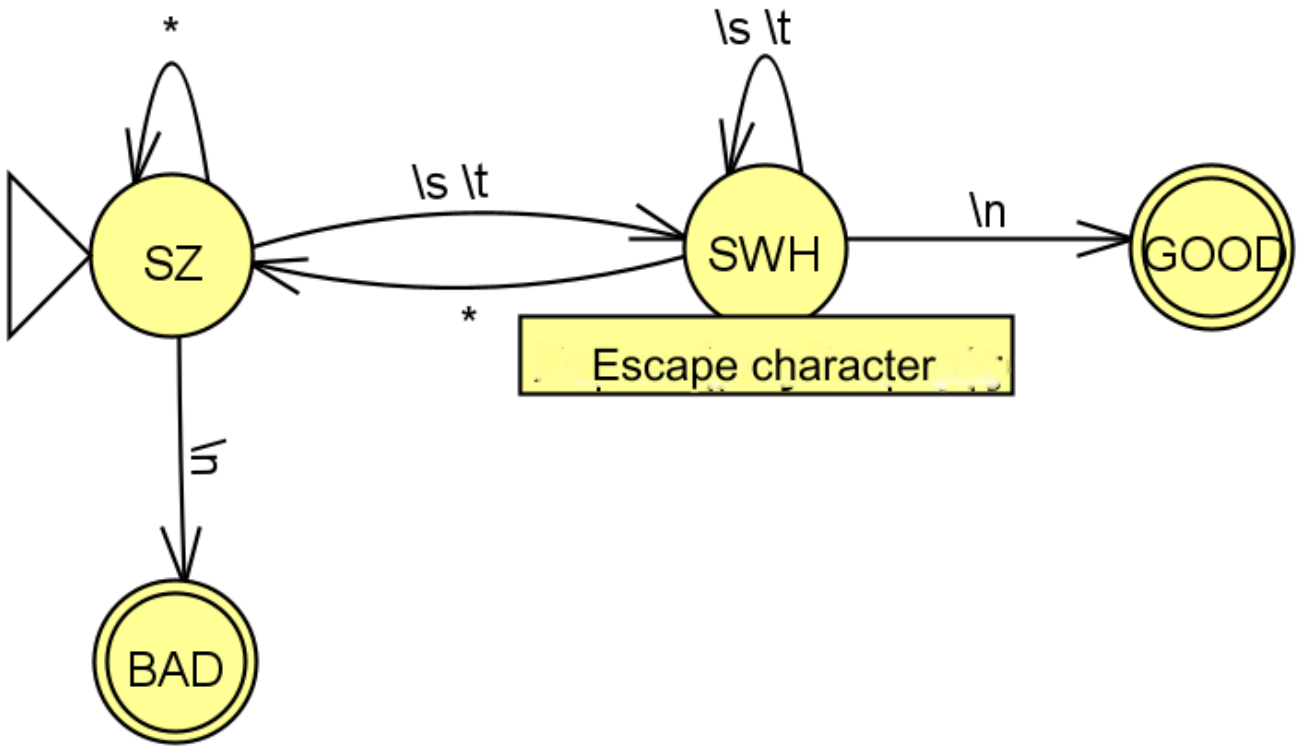


Figure 6.3.1 Uni-C finite escape character automaton

The transition diagram starts from SZ. If any white space character is read from space, tab (\s \t) it goes to SWH state. If any character is read beyond the previously mentioned white space characters, it remains in the SZ state. If a line break is read from SZ, it goes to the BAD exit state. For SWH as long as white space characters are read, the situation remains as it is in SWH, otherwise if any other character is read besides white spaces it goes back to SZ. If a line change is read from SWH, it goes to the GOOD output state.

6.4 Transition Table

	\s	\t	Another	EOF	\n	EXIT

			character			STATUS
SZ	SWH	SWH	SZ	BAD	BAD	NO
SWH	SWH	SWH	SZ	BAD	GOOD	NO
GOOD						YES
BAD						YES

Table 6.4.1 Uni-C escape character transition table

Status Acronym	Status Name	Status Information
SZ	State Z	The status for reading the 1st escape character
SWH	State Whitespaces	The condition for reading any other character along with an escape character

Table 6.4.2 Status table of Uni-C language escape characters

6.5 Encoding via FSM

The FSM encoding of the Uni-C escape character machine is in the source file **6_white_spaces.fsm**

```
// ================== 6_White_spaces ==================


START = SZ // ==== The initial state is SZ ====


SZ: // --- [State Z] State to read escape character ---

\s \t -> SWH // Recognizes escape character

* -> SZ // Recognizes any other character it reads

\n -> BAD // No end-of-file escape character read
beyond newline


SWH: // --- [State Whitespace] State to read
escape character ---
```

```
* -> SZ // Recognizes any other character read,
return to initial state

\n -> GOOD // Ends with escape character followed by newline

\s \t -> SWH // Followed by escape character, stay in same state


BAD: // ==== Exit status: OXI ====

* -> BAD


GOOD(OK): // ==== Exit status: YES ====
```

6.6 Audit Cases

```
#1    It is valid because before the newline followed by EOF is read, the tab
(SWH) is followed which is a whitespace character in the Uni-C language.

./fsm -trace 6_white_spaces.fsm


sz \s -> swh

swh \t -> swh

swh \n -> good

YES
```

```
#2    It is not valid because it does not include tabs or spaces or combinations
thereof before reading the newline followed by EOF. Although newline and EOF are
whitespace characters, in the Uni-C language they are recognized and returned as
tokens with different identifier names.

./fsm -trace 6_white_spaces.fsm

hi

sz h -> sz

sz i -> sz

sz \n -> bad
```

```
bad EOF -> bad

NO
```

#3 It is not valid because there are no spaces, tabs or combinations thereof and it does not end with a newline but with EOF. Although EOF is a whitespace character, in the Uni-C language it is recognized and returned as a token with a different identifier name.

```
./fsm -trace 6_white_spaces.fsm

sz EOF -> sz

NO
```

#4 It is not valid because there are no spaces, tabs or combinations thereof before the newline followed by EOF is read.

```
./fsm -trace 6_white_spaces.fsm

        hi

sz \t -> swh

swh h -> sz

sz i -> sz

sz \n -> bad

bad EOF -> bad

NO
```

#5 It is valid because it reads blank before the newline and end of file EOF is read.

```
./fsm -trace 6_white_spaces.fsm

Hi Mark

sz h -> sz

sz i -> sz

sz \s -> swh
```

```
swh M -> sz

sz a -> sz

sz r -> sz

sz k -> sz

sz \s -> swh

swh \n -> good

YES
```

#6 It is valid because it follows the rule and goes from SZ to SWH when the character (\t) is read, it also stays in SWH state when the escape character (\t) is read, and then when the newline character is read it goes to GOOD.

```
./fsm -trace 6_white_spaces.fsm


sz \t -> swh

swh \t -> swh

swh \n -> good

YES
```

#7 It is not valid because it does not start with the expected escape character (\t) to enter SWH state, is not followed by an escape character (\t) after "h" to remain in SWH state, and does not read (\n) to give GOOD.

```
./fsm -trace 6_white_spaces.fsm

hi mark sz h -> sz

sz i -> sz

sz \s -> swh

swh m -> sz

sz a -> sz

sz r -> sz

sz k -> sz
```

```
sz \s -> swh

swh EOF -> sz

NO
```

6.7 Problems and solutions

We encountered a problem in the transition diagram design, as in all the transition diagrams of the Uni-C language modules mentioned above, we used the newline character (\n) to transition to the GOOD exit state. The original transition diagram also included the newline character in addition to tabs and spaces for the intermediate states (as this is also a whitespace character), while the final character for the transition to GOOD was EOF.

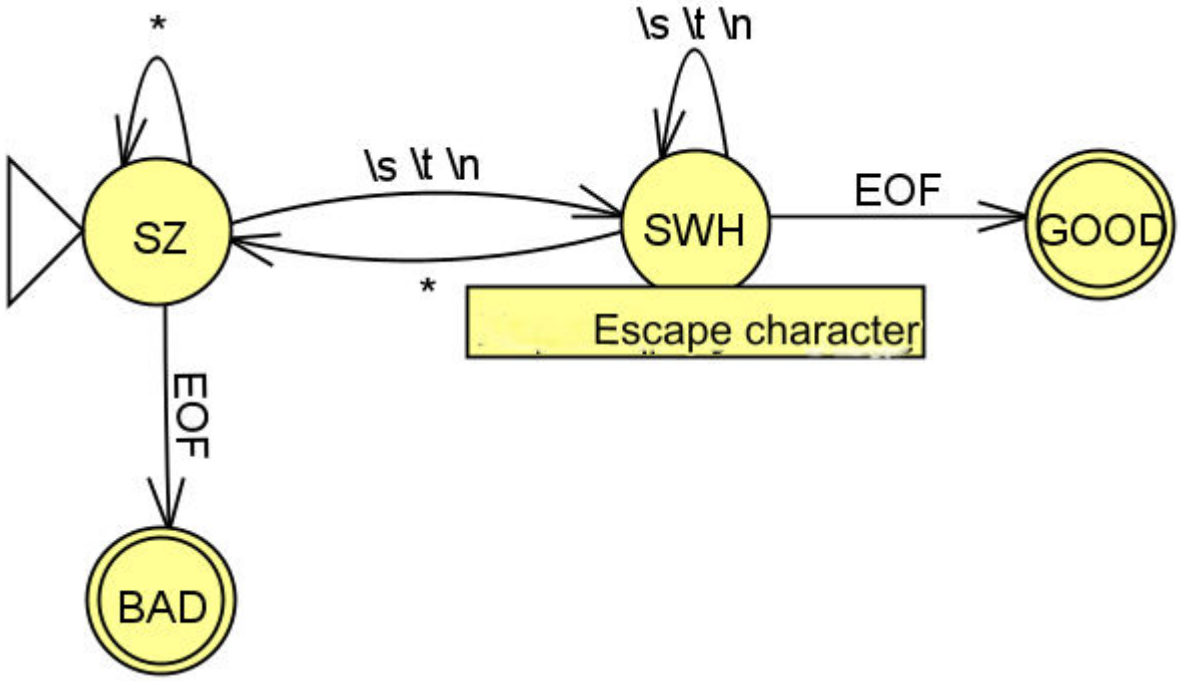


Figure 6.7.1 Initial finite automaton of the Uni-C language escape characters

After another meeting of the team we realized that EOF is also a valid whitespace character so the reflection remained intense for the final diagram.

By better reading the Uni-C alphabet and the basic identifier classes used in pattern recognition in Uni-C, we established that although the newline (\n) and end-of-file (EOF) characters are recognized as whitespaces characters, they are returned as tokens with their own identifier names (***newline*** for line break and ***eof*** for end of file). Therefore, the problem was solved by leaving the newline character for the transitions to the GOOD, BAD output states as we did in the above-mentioned Uni-C language unit diagrams. The final diagram is here .

6.8 Express reporting of deficiencies

The report includes all the requests required:

- Identification pattern
- Regular expression with description
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm 6_white_spaces.fsm

./fsm -list 6_white_spaces.fsm

./fsm -trace 6_white_spaces.fsm
```

Final

The simulation was undertaken by the students of the whole group

Athanasiou Vasileios Evangelos (UNIWA-19390005) – Theocharis Georgios (UNIWA-19390283) – Tatsis Pantelis (UNIWA-20390226) – Iliou Ioannis (UNIWA-19390066) – Dominaris Vasileios (UNIWA-21390055)

Single finite state automaton or FSM

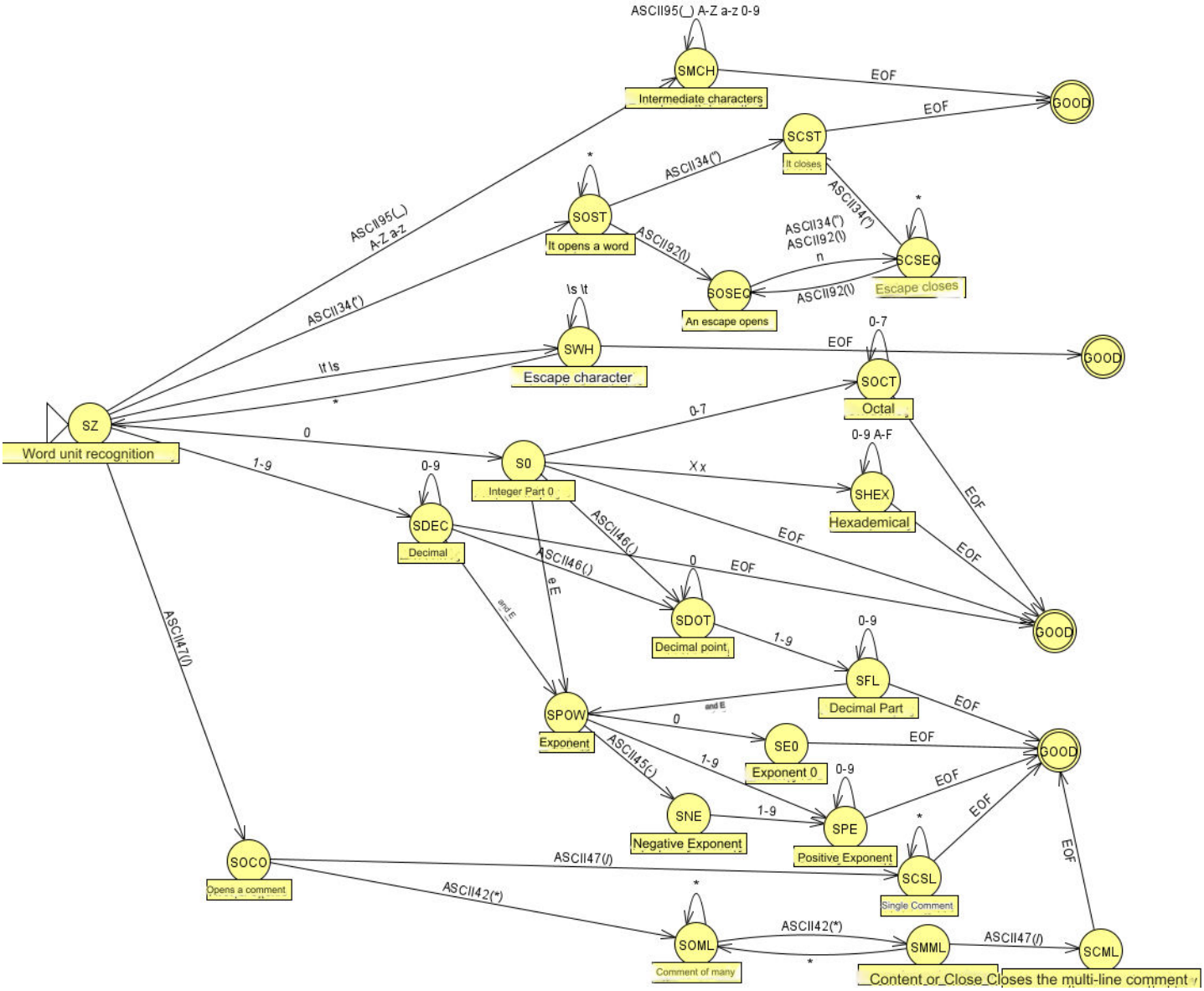


Figure 1. Unitary finite automaton of the Uni-C language units

Uni-C's unified word recognition machine is designed to recognize and separate different kinds of words, numbers, and comments from a text. Here's how it works:

Initial Status (START = SZ):

Recognizes Latin characters, underscores and digits.

Recognizes double quotation marks as an indication of the beginning of a word.

Recognizes numbers starting with 0 as octal, hexadecimal or decimal.

Recognizes comments.

Identifiers (1_Identifiers - SMCH):

It continues to recognize Latin characters, underscores, and digits.

Once it encounters double quotes, it enters verbose (SOST) state.

Vocabulary (2_Vocabulary - SOST):

Recognizes literals until it encounters a second double quote or escape character.

When an escape character is encountered, it enters a read escape character (SOSEQ) state.

Integers & Numbers (3_Integers & 4_Floating Point Numbers):

Recognizes integers, hexadecimal, octal, and floating point numbers.

Can recognize negative numbers, powers and exponent in scientific form.

Comments (5_Comments - SOCO):

Recognizes comments, either single-line or multi-line.

White Spaces (6_White_spaces - SWH):

Recognizes white space characters (space, tab, new line).

Each state has predefined behavior for each character it reads. Depending on the character it reads and the current state, the automaton decides where to go next. In the final state (GOOD) the read is completed successfully.

Single transition table

The single transition table is on a separate sheet in the file **Final_Array.xlsx** for readability reasons.

The single state table is on a separate sheet in the file **Final_Array.xlsx** for readability reasons.

Encoding via FSM

The FSM encoding of the Uni-C language unit recognizer can be found in the source file **Final.fsm**

```
START = SZ


// ================== 0_Initial state ==================


SZ: // --- [State Z] Speech unit recognition state ---

AZ -> SMCH // 1_IDs

_ -> SMCH // 1_IDs

az -> SMCH // 1_IDs


" -> SOST // 2_Vocabulary


0 -> S0 // 3_Integers or 4_Floating Point Numbers

1-9 -> SDEC // 3_Integers or 4_Floating Point Numbers


    / -> SOCO // 5_ Comments


\t -> SWH // 6_White_spaces characters

\s -> SWH // 6_White_spaces characters


EOF -> GOOD // Done file


    * -> BAD // Invalid character


// ================== 1_IDs ==================


SMCH: // --- [State Mid Characters] State for mid
characters ---

AZ az -> SMCH // Lower or uppercase Latin letters

_ -> SMCH // Underscore

0-9 -> SMCH // Digits

\n -> SZ // Acknowledgment is valid and reading continues

EOF -> GOOD // Acknowledgment is valid and reading is terminated

* -> BAD // ID is invalid
```

```
// ================== 2_Lexuals ==================


SOST: // --- [State Open String] State for the content of the
verb ---

" -> SCST // The verb is enclosed in double quotes

\\ -> SOSEQ // Escape sequence follows

* -> SOST // Any other character follows


escaped sequence characters ---


\\ -> SCSEQ // Followed by the slash character (\)

" -> SCSEQ // Double quote follows

n -> SCSEQ // Follows n for newline

* -> BAD // The literal is invalid


SCSEQ: // --- [State Close Sequence] State to close verbal or
escape sequence ---

" -> SCST // The verb is enclosed in double quotes

\\ -> SOSEQ // Return to state for escaped characters

* -> SCSEQ // Any other character follows


SCST: // --- [State Close String] State to close the
verbal ---

\n -> SZ // The literal is valid and reading continues

EOF -> GOOD // The literal is valid and terminates reading

* -> BAD // The literal is invalid


// ================== 3_Integers & 4_Floating Point Numbers ==================


S0: // --- [State 0] State for 0 as integer part ---

0-7 -> SOCT // Octal integer

x X -> SHEX // Hexadecimal integer
```

```
. -> SDOT // The integer part is 0 followed by a decimal point for
the decimal part

E e -> SPOW // The power base is 0 followed by the
scientific form of the exponent

\n -> SZ // Integer 0 is valid and reading continues

EOF -> GOOD // Integer 0 is valid and terminate reading

* -> BAD // The integer is invalid



SDEC: // --- [State Decimal] Positive Decimal Integer State ---

0-9 -> SDEC // More digits from 0-9 follow

. -> SDOT // Decimal point follows

E e -> SPOW // The scientific form of the exponent follows

\n -> SZ // Decimal integer is valid and
reading continues

EOF -> GOOD // Decimal integer is valid and end
reading

* -> BAD // The integer is invalid



SHEX: // --- [State Hexadecimal] State Hexadecimal ---

0-9 AF -> SHEX // More digits from 0-9 and letters from AF follow

\n -> SZ // Hex integer is valid and
reading continues

EOF -> GOOD // Hex integer is valid and end
reading

* -> BAD // The integer is invalid



SOCT: // --- [State Octal] State Octal Integer ---

0-7 -> SOCT // More digits from 0-7 follow

\n -> SZ // Octal integer is valid and
reading continues

EOF -> GOOD // Octal integer is valid and end
reading

* -> BAD // The integer is invalid



// ================== 4_Floating Point Numbers ==================
```

```
SDOT: // --- [State Dot] State for the decimal point ---

0 -> SDOT // Digit 0 follows

1-9 -> SFL // Digits for the decimal part from 1-9 follow

* -> BAD // The number is invalid


SFL: // --- [State Float] State for the decimal part ---

0-9 -> SFL // More digits follow for the decimal part from 0-9

E e -> SPOW // The scientific form of the exponent follows

\n -> SZ // Number is valid and reading continues

EOF -> GOOD // Number is valid and end reading

* -> BAD // The number is invalid


SPOW: // --- [State Power] State for powers ---

- -> SNE // Negative sign for the exponent follows

0 -> SE0 // Exponent is 0

1-9 -> SPE // Followed by positive exponent

* -> BAD // The number is invalid


SNE: // --- [State Negative Exponential] State for negative
exponent ---

1-9 -> SPE // Followed by digits 1-9 for the negative exponent

* -> BAD // The number is invalid


SPE: // --- [State Positive Exponential] State for Positive Exponential
---

0-9 -> SPE // Followed by digits 1-9 for the positive exponent

\n -> SZ // Number is valid and reading continues

EOF -> GOOD // Number is valid and end reading

* -> BAD // The number is invalid


SE0: // --- [State Exponential 0] State for digit 0 for
exponent to power ---

\n -> SZ // Number is valid and reading continues
```

```
EOF -> GOOD // Number is valid and end reading
* -> BAD // The number is invalid


// ================== 5_Comments ==================


SOCO: // --- [State Open Comment] State Open Comment ---
/ -> SCSL // Opens a single line comment
\* -> SOML // Opens a multiline comment
* -> BAD // Comment is invalid


SOML: // --- [State Open Multiple Line (Comment)] State comment
                                                                  many lines ---
    \* -> SMML // Intent to close the comment
* -> SOML // Comment content
EOF -> BAD // Comment is invalid


SMML: // --- [State Mid Multiple Line (Comment)] State comment
                                                                 many lines ---
    / -> SCML // Closes multiline comment
* -> SOML // Return to state for comment content
EOF -> BAD // Comment is invalid


SCML: // --- [State Close Multiple Line (Comment)] Status valid
                                                                   comment many
                                                                   lines ---
    \n -> SZ // Comment is valid and reading continues
EOF -> GOOD // Comment is valid and end reading
* -> BAD // Comment is invalid


SCSL: // --- [State Close Single Line (Comment)] Status valid
                                                                 comment one line
---
    \n -> SZ // Comment is valid and reading continues
EOF -> GOOD // Comment is valid and end reading
* -> SCSL // Comment is invalid
```

```
// ================= 6_White_spaces =================


SWH: // --- [State Whitespace] State to read
escape character ---

* -> SZ // Recognizes any other character read,
return to initial state

\n -> GOOD // Ends with escape character followed by newline

\s \t -> SWH // Followed by escape character, stay in same state


// ================= Exit status: OXI =================


BAD:

* -> BAD


// ================= Exit status: YES =================


OK:
```

Control cases

#1 The automaton accepts the following inputs, as the first input is an identifier, the second is a character string, the third is an integer, the fourth is a decimal number, the fifth is a comment, and the last input is an identifier even though there is a whitespace character (\ t).

```
./fsm -trace Final.fsm

x

sz x -> smch

smch \n -> sz

"y"

sz " -> sost
```

```
sost y -> sost

sost " -> scst

scst \n -> sz

1

sz 1 -> sdec

sdec \n -> sz

1.1

sz 1 -> sdec

sdec . -> sdot

sdot 1 -> sfl

sfl \n -> sz

// z

sz / -> soco

soco / -> scsl

scsl \s -> scsl

scsl z -> scsl

scsl \n -> sz

   hi

sz \t -> swh

swh h -> sz

sz i -> smch

smch \n -> sz

sz EOF -> good

YES
```

#2 The automaton rejects the following string as an identifier must not start with a number.

```
./fsm -trace Final.fsm
```

```
1_user

sz 1 -> sdec

sdec _ -> bad

bad u -> bad

bad s -> bad

bad e -> bad

bad r -> bad

bad \n -> bad

bad EOF -> bad

NO
```

```
#3    The following string is accepted by the automaton as the characters are enclosed by the special characters (“ ”).

./fsm Final.fsm

"user

one

"

YES
```

```
#4    By default the character sequence is valid. The string is an identifier that meets the Uni-C language rules, which state that an identifier must start with a letter [A-Za-z].

 ./fsm -trace Final.fsm

E012

sz E -> smch

smch 0 -> smch

smch 1 -> smch

smch 2 -> smch

smch \n -> sz
```

```
sz EOF -> good

YES
```

```
#5    The automaton accepts the following comments. This is because comments are defined with '//' characters.

./fsm Final.fsm

// hi

// mark

// how

// are

// you?

YES
```

```
#6    It is valid from the automatic. The characters “/*” indicate the beginning of comments and the characters “*/” their end.

./fsm Final.fsm

/*


this is a comment


*/

YES
```

```
#7    The automaton rejects the following input, since in the Uni-C language signed integers are undefined.

./fsm Final.fsm

-0

NO
```

#8 The '“' character at the beginning and end of the literal makes the string a valid string accepted by the automaton.

```
./fsm Final.fsm

"-0"

YES
```

#9 The characters “//” indicate the beginning of the comment. Therefore the following string is accepted as a comment by the automaton.

```
./fsm Final.fsm

//-0

YES
```

#10 The following string is incorrect because it is not a comment in the Uni-C language. This is because “this is a string” is outside the special characters (“ ”). The string is not accepted by automaton.

```
./fsm Final.fsm

"\"this is a string""

NO
```

#11 The following string is not accepted by the automaton because the Uni-C language accepts the use of the backslash character (\) as a character within the literal, only with the use of an escape sequence (\\).

```
./fsm Final.fsm

"this is a string\0"

NO
```

#12 The following inputs are not accepted as identifiers and are automatically rejected, as the special character “-” cannot be used to set an identifier.

```
./fsm Final.fsm

1-Id
```

```
2-Str

3-Int

4-Fl

5-Com

6-Wh

NO
```

```
#13   The following inputs are whitespace characters of the Uni-C language and are therefore accepted by the automaton.

./fsm -trace Final.fsm

  tab

sz \t -> swh

swh t -> sz

sz a -> smch

smch b -> smch

smch \n -> sz

space

sz \s -> swh

swh s -> sz

sz p -> smch

smch a -> smch

smch c -> smch

smch e -> smch

smch \n -> sz

newline

sz n -> smch

smch e -> smch

smch w -> smch
```

```
smch l -> smch

smch i -> smch

smch n -> smch

smch e -> smch

smch \n -> sz

sz EOF -> good

YES
```

#14 The first three tests are valid. The last test with the characters "0xAf" is wrong because the character "f" is a lowercase letter, if it was an uppercase it would be valid.

```
./fsm -trace Final.fsm

007

sz 0 -> s0

s0 0 -> soct

socket 7 -> socket

soct \n -> sz

0

sz 0 -> s0

s0 \n -> sz

123

sz 1 -> sdec

sdec 2 -> sdec

sdec 3 -> sdec

sdec \n -> sz

0xAf

sz 0 -> s0

s0 x -> shex

shex A -> shex
```

```
shex f -> bad

bad \n -> bad

bad EOF -> bad

NO
```

```
#15   The first input is accepted as an integer, the second input is accepted as an 8-bit number, the third is accepted as an integer, the fourth is a 16-bit number. The fifth input is accepted as a decimal number, the sixth input is accepted as an exponential number, the seventh is also accepted as an exponential number as is the last input.

./fsm Final.fsm

0

007

123

0xAF01

3.14

1E-1

1.45E0

0e0

YES
```

```
#16   The third input is not accepted by the automaton, as comments in Uni-C do not start with the “#” character.

./fsm Final.fsm

// C

/* C */

# Python

NO
```

#17 Are the following inputs correct? The first and second are comments while the third input is a string.

```
./fsm Final.fsm

// C

/* C */

"#Python"

YES
```

#18 The following string is not accepted by automaton, because comments should not be written between the special characters "* *\" but between the characters "/* */".

```
./fsm Final.fsm

\*

*

*\

NO
```

#19 The automaton rejects the first input, as -0 is not accepted as an exponent. The rest of the inputs are correct.

```
./fsm Final.fsm

4e-0

1.5

0

4e0

NO
```

#20 The first input is acceptable because the escape character (\") is properly used to properly ignore the (" "). The second input does not make proper use of the escape character (\"), because the backslash (\) is not inside the string. The input is not accepted by the automaton. The third input is not accepted by the automaton as 6\3=2 is not enclosed in double quotes (").

```
./fsm Final.fsm

"he said \"this is a string\"\n"

"he said "\"this is a string\"\""

"he said "6\3=2""

NO
```

```
#21   The literals are valid, because the characters "//" indicate a comment, tab is a whitespace character, and "/* */" are also used for comments.

./fsm Final.fsm

// comments? tab

//

/*

*/

YES
```

```
#22   The second string is accepted as an exponential number. The first string is not accepted because after e the only accepted characters are real numbers.

./fsm Final.fsm

4exp0

4e0

NO
```

```
#23   The dot at the end does not correspond to the automaton standard for identifiers in the Uni-C language. The first 2 entries are correct as identifiers and are accepted.

./fsm Final.fsm

_us_

er_

_service.
```

```
NO
```

#24 The string is not accepted because it has single quotes (' ') instead of double quotes (“ ”) so it is a wrong string definition.

```
./fsm Final.fsm

"\n\n \'This is a string\'"

NO
```

#25 The following inputs are accepted by the automaton as strings are enclosed in double quotes (“ ”).

```
./fsm Final.fsm

"\n\n"

"

"

YES
```

#26 The string is not accepted as the whitespace character after “Hello” does not make hello world an identifier in the Uni-C language.

```
./fsm Final.fsm

Hello World

NO
```

#27 The first 2 literals are not accepted by the automaton, as comments must start with (//) instead of (/). The last entry is a correct comment.

```
./fsm Final.fsm

/

/

// yes

NO
```

#28 The first word is accepted as a number in the 8-digit numbering system. The second literal is not accepted as an 8-digit number because the numbers 8 and 9 are not within the number range [0-7].

```
./fsm Final.fsm

0017

089

NO
```

#29 Verbs are accepted as numbers in the Uni-C language, the first being an 8-bit number and the second being an integer.

```
./fsm Final.fsm

0017

0

YES
```

#30 The first literal is accepted by the automaton as a 16-odd number. The second is unacceptable because hexadecimal numbers must start with "0x" or "0X". The third word is accepted as an identifier.

```
./fsm Final.fsm

0xAF

01EF

A356

NO
```

#31 The following literals are accepted by the automaton as they are 16-even numbers. All literals correctly start with "0x" and contain numbers within the range [0-F].

```
./fsm Final.fsm
```

```
0xAF

0X01EF

0xA356

YES
```

```
#32  The following literal is correct as a string as it is enclosed in double quotes (“ ”), therefore it is accepted by the automaton.

/fsm Final.fsm

"-0x1135

1-0

001

1.56

1.56

"

YES
```

```
#33  The “\” character is not accepted for quoting comments, so the second input is not accepted by the automaton.

./fsm Final.fsm

//////////////////////////

\\\\\\\\\\\\\\\\\\\\\\\\

NO
```

```
#34  The “-” character is not accepted for identifier definition in Uni-C language. For this reason the first verb is not accepted by the automaton, while the second verb is accepted.

./fsm Final.fsm

uni-c

uni_c
```

```
NO
```

#35 Whitespace characters are accepted by the automaton as well as the “uni_c” identifier.

```
./fsm Final.fsm

tab

tab

tab

uni_c

tab

YES
```

#36 The use of the escape character (“\) is correct in the first verb accepted by the automaton. The second literal is also acceptable because it is a string enclosed in double quotes.

```
./fsm Final.fsm

"\\\"\"\n"

"

\\\\"

YES
```

#37 The following words are wrong because instead of the character (/) the backslash (\) is used. The backslash is not used to define comments.

```
./fsm Final.fsm

\\ hey

\* *\

NO
```

#38 The text is not valid, since in order to be accepted as a comment, the characters "*/" should be used instead of the special characters "/*" at the end.

```
./fsm Final.fsm

/*

this

is

a

comment

/*

NO
```

#39 The first literal is not accepted, as the backslash (\) is used for commenting. The second literal is enclosed in quotation marks and is therefore not acceptable, because quotation marks are not used in the definition of a literal in the Uni-C language.

```
./fsm Final.fsm

\*._.

123456910

*\

(.)

NO
```

#40 The following literals are accepted by the automaton, except the fifth literal because the semicolon (?) is not used to define identifiers in the Uni-C language.

```
./fsm Final.fsm

Okay

12345

3.45

3E-1
```

```
SURE?

YES

NO
```

Problems and solutions

We ran into a problem with the single automaton and encoding via FSM , as we had initially decided that it would only accept one valid word unit in one code run. Therefore, in the original version of the code the indicative control cases were as follows:

```
./fsm Final.fsm

identifier

YES
```

```
./fsm Final.fsm

identifier

34

NO
```

From the original transition diagram, when we reached a transition where, with the input of the newline character , we went to the GOOD state , we did not find a way for the automaton to continue to recognize more word units with one code execution.

In conclusion, we programmed in the FSM but did not design in the automaton for readability reasons, an extra transition for each such state that was one input away from the transition to the output state GOOD . The transition takes as input the newline character (\ n) and returns back to the initial state SZ , to continue reading strings. We had ensured that the previous string was valid since, in the transition before GOOD , the only thing missing was this character for the transition to GOOD . The input character for going to GOOD has been replaced with EOF (End - of - File).

```
./fsm Final.fsm

identifier

YES
```

```
./fsm Final.fsm
```

```
identifier

34

YES
```

The concern about whitespace characters remained intense until the completion of the single, however the conclusion is reported here .

For reasons of readability, we decided to write the single table of transitions in an excel file.

Explicit reporting of deficiencies

The report includes all the requests required:

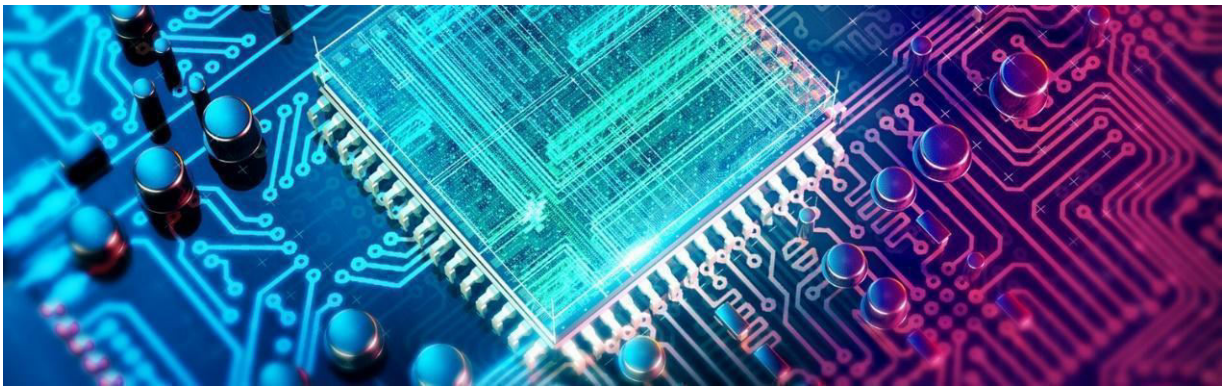
- Finite state automaton with description
- Transition table with state table
- Coding via FSM with code commenting
- Test cases with exhaustive test runs and commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with Windows Subsystem for Linux (WSL) installed. The terminal the team ran the code on was on Ubuntu 22.04 LTS, so the commands we used to run the code are as follows:

```
gcc -o fsm fsm.c

./fsm Final.fsm

./fsm -list Final.fsm

./fsm -trace Final.fsm
```

COMPILERS

Thank you for your attention.



COMPILERS