Ryan Hutchings

Gabriel Coyote

April 27, 2021,

The Parser

**Introduction**

The parser function has an output pararmeter “error” and a parameter “print”, bot integers set to zero. The parser will be runned twice, on the first run it will not print, instead it will check for errors. If found it will set the output parameter “error” to ‘1’. Else it will set “print” to 1, and actually print to the console the XML-parse Tree.

**Data Structure**

The method that we decided to go with is the two-dimensional array from the textbook. The idea is to start by resetting every value to zero before scanning every token in the text file. When scanning, we create a single-dimensional interer array called *tokens\_stored*, which contains 200 integer values and stores them. The index 0 of *tokens\_stored* is the first integer to be encountered and it continues until the index is 199. For example, if *tokens\_stored[0]* = 16, then the first token it encounters is from state 16, which means from the DFA that the token is an ID. In order to detect the new IDs, which are read and write, we will initialize the states as 21 and 22 respectively.

We created a string array called *tokens\_values*, which stores a max of 200 tokens values and a max of 20 characters per string, to store the values of every token. The first token value would be shown as *tokens\_values[0]*.

Then we created another array called *token\_tab*, which the size of the array is 19, to specify the tokens from Figure 2.12 from the textbook. Looking at Figure 2.12, we see that the divide token is in state 2, so we make the index of the array *token\_tab* 2 and the given value 2. Another example is the plus token. If the plus token is in state 8, then the result would be *token\_tab[8]* = 8 and this continues on for the rest of the tokens. For any indices in the *token\_tab*  array that do not have tokens, there values are set to zero.

In order to scan the tokens from the scanner table in Figure 2.12, we need to use a two-dimension array called *scan\_tab*. The first index goes from 0 to 18 which represents every state (expect index 0 is not used) and the second index goes from 0 to 13. For the second index, index 0 represents the white spaces like space and tab, index 1 represents a newline, index 2 represents a /, and it continues on until index 13. For any integer *i* and *j*, *scan\_tab[i][j]* is a record with field names like *action* and *newState.action* that can take values like *move*, *recognize*, and *error*. If *action* = *moves*, then that means the automata will move to the next state (the next state value is equal to the one stored at *scan\_tab[i][j])*. If *action* = *recognize*, then that means the index i is at a final state and the automata can not move to the next state with the corresponding index j. If the automata stops at the final state, then it recognizes the token. If *move* = *error*, then that means the automata can not get to any state from state i to the corresponding number in j.

**Algorithms**

**Algorithm :** scan

**Input:**

*File\_PTR*: The current pointer of the input file

*cur\_char*: current character

*cur\_state:* holds the current state #

*remembered\_state:* holds state #

*image:* list of characters, used to hold encountered token’s string

**Output:**

*token*: holds encountered token’s state #

**Side Effects:**

Prints “error.” if encountered token is invalid, then terminates the program

**Plan:**

**while** *File\_PTR* is not EOF

read *cur­\_char*

**case** *scan\_tab[cur\_state][ cur\_char ].action*

*move:*

**if** *token\_tab[cur\_state]*  is not empty (!= 0)

*remembered\_state := cur\_state*

*cur\_state* := *scan\_tab[cur\_state][cur\_char].*nextState

*recognize:*

*token* := *token\_tab[cur\_state]*

unread *cur\_char*

**return** *token*

*error:*

print “error.” , then terminate program

append *cur\_char* to *image*

**End of Algorithm**

**Algorithm :** Int\_cur\_char

**Input:**

*ch:* a character

**Output:**

returns a number, for *scan\_tab[][ i ]* array

**Side Effects:**

N/A

**Plan:**

**If** *ch* is a space or tab

return 0

**else if** *ch* is a newline character

return 1

**else if** *ch* is a “/”

return 2

**else if** *ch* is a “\*”

return 3

**. . . //**Repeat for all cur character in the Fig 2.12 table

**else if** *ch* is a digit

return 11

**else if** *ch* is a letter

return 12

**else** return 13

**End of Algorithm**

**Algorithm :** Driver

**Input:**

*File\_PTR*: The current pointer of the input file

*tokens\_stored:* array to hold tokens encounterd (Their State #)

*tokens\_values:* array of strings to hold tokens encountered

**Output:**

*tokens\_stored* elements’ values are set to the token encountered (Their State #)

*tokens\_values* elements’ are set to the token encountered (Their string value)

**Data:**

*i*: number used for accessing *tokens\_stored* indices

*tok:* number used to hold encountered token’s state #

*cur\_char*: current character

*cur\_state:* holds the current state #

*remembered\_state:* holds state #

*image:* list of characters, used to hold encountered token’s string

**Side Effects:**

N/A

**Plan:**

**While** *File\_PTR* is not EOF

*cur\_state* := *start\_state* (1)

*remembered\_state* := 0 //None

*image* := null

//*tok* is the output of *scan* [Algorithm]

*tok* := scan( *File\_PTR, cur\_char, cur\_state, remembered\_state, image)*

**if** *image* is equal to “read”

*tok* := *read’s* state number (21)

**else if** *image* is equal to “write”

*tok* := *write’s* state number (22)

**else**

**//**Leave *tok* as is

*Tokens\_stored[i]* := *tok*

*Tokens\_values[i]* := *image*

increment *i* by one

**End of Algorithm**

**PARSER Algorithms**

**Algorithm : Match**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

Consumed the current *input\_token*

**Plan:**

**If** *input\_token* is equal to token *expected*

Consumed token

**Else**

Error := 1

**End of Algorithm**

**Algorithm : Program**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID or READ or WRITE or $$

Stmt\_list(i*nput\_token*);

Match($$,*Input\_token*:,error);

**Else**

Error := 1

**End of Algorithm**

**Algorithm : Stmt\_list**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID or READ or WRITE

Stmt(*Input\_token*:,);

Stmt\_list(*Input\_token*:,);

**Else If** *input\_token* is equal to $$

Skip – (eplison production)

**Else**

Error

**End of Algorithm**

**Algorithm : Stmt**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID

Match(ID,*Input\_token*:,error);

Match(:=,*Input\_token*:,error);

Expr(*Input\_token*:,);

**Else If** *input\_token* is equal to READ

Match(READ,*Input\_token*:,error);

Match(ID,*Input\_token*:,error);

**Else If** *input\_token* is equal to WRITE

Match(WRITE,*Input\_token*:,error);

Expr(*Input\_token*:,);

**Else**

Error

**End of Algorithm**

**Algorithm : Expr**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID or NUMBER or (

Term(*Input\_token*:,);

Term\_tail(*Input\_token*:,);

**Else**

Error

**End of Algorithm**

**Algorithm : Term\_tail**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to + or -

Add\_op(*Input\_token*:,);

Term(*Input\_token*:,);

Term\_tail(*Input\_token*:,);

**Else If** *input\_token* is equal to ) or ID or READ or WRITE or $$

Skip – (eplison production)

**Else**

Error

**End of Algorithm**

**Algorithm : Term**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID or NUMBER or (

Factor(*Input\_token*:,);

Factor\_tail(*Input\_token*:,);

**Else**

Error

**End of Algorithm**

**Algorithm : Factor\_tail**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to \* or /

Mult\_op(*Input\_token*:,);

Factor(*Input\_token*:,);

Factor\_tail(*Input\_token*:,);

**Else If** *input\_token* is equal to + or – or ) or ID or READ or WRITE or $$

Skip – (eplison production)

**Else**

Error

**End of Algorithm**

**Algorithm : Factor**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to ID

Match(ID,*Input\_token*:,error);

**Else If** *input\_token* is equal to NUMBER

Match(NUMBER,*Input\_token*:,error);

**Else If** *input\_token* is equal to (

Match( **(** ,*Input\_token*:,error );

Expr(*Input\_token*);

Match( **)** ,*Input\_token*:,error );

**Else**

Error

**End of Algorithm**

**Algorithm : Add\_op**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to +

Match(+,*Input\_token*:,error);

**Else If** *input\_token* is equal to -

Match(-,*Input\_token*:,error);

**Else**

Error

**End of Algorithm**

**Algorithm : Mult\_op**

**Input:**

*Input\_token*: The current token

*Expected:* token expected

*error:* integer that sets to 1 if there is an error

**Output:**

**Side Effects:**

**Plan:**

**If** *input\_token* is equal to \*

Match(\*,*Input\_token*:,error);

**Else If** *input\_token* is equal to /

Match(/,*Input\_token*:,error);

**Else**

Error

**End of Algorithm**

**Main Algorithm**

**Input:**

*Filename*: text file name from the command line

**Output:**

N/A

**Data:**

*inputFile:* the file pointer

*tokens\_stored:* array to hold tokens encounterd (their State #)

*tokens\_values:* array of strings to hold tokens encounterd (their string value)

*error*: integer that sets to 1, if there is a parse error

*print:* integer that sets to 1, if there are NO parse error

**Side Effects:**

Prints to console error. if there is any parser error; otherwise Prints the XML-parser tree

**Plan:**

*inputFile :=* open *Filename*

Driver( *inputFile*, *tokens\_stored, tokens\_values* ); //Algorithm

*error* := 0;

*print* := 0;

Program( *tokens\_stored, tokens\_values, \*error, print* ); //Algorithm

**If** ( *error == 1)*

Print to console “Error.”

**Else**

*print := 1;*

Program( *tokens\_stored, tokens\_values, \*error, print* ); //Algorithm

Close *Filename*

**End of Algorithm**

**Test Cases**

1. The test case will be the text file *foo.txt* that reads as :

read

/\* foo

bar \*/

\*

five 5

We chose this one because its simple, and it does not follow the context free grammar. Should print “Error.”

1. The other test case will be the text file *parser.txt* that reads as :

Read A

We chose this one as we know what should be printed, given by the Project 2 PDF.

**Aknoweledgement**

BIG thanks to the book, *Programming Language Pragmatics 4th edition (Micharl L. Scott)*, for helping us understand recursive descent parsing.