Ryan Hutchings

Gabriel Coyote

April 18, 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**

To store the token type we set an one-dimensional integer array to hold these values;

The integar array, *tokens\_stored,* is of size 200. All values are set to zero before the *scan* function runs. The index 0 of *tokens\_stored* represents the first token encounter. Index 1, the second token encountered and so on. The *tokens\_stored[0] := 16,* for example reads as the first token encounter is of state 16 (from DFA its token ID). New state, that acts as exception for the ID token are *read* and *write,* state 21 and 22 respectfully.

To store the tokens’ values, we set a an array of strings called *tokens\_values,* it holds 200 token values of max string length 20. First token encountered is stored at *tokens\_values*[0].

We have token array, *token\_tab,* which is of size 19 to specify the tokens in Fig 2.12 in the textbook. Looking at the Fig 2.12 we see token *div* has a state number of 2, and so in the token array at index 2 we have the value 2. Token *plus* has a state number of 8, and so at *token\_tab[8] := 8,* and so on for the rest of the tokens*.* For indices in the *token\_tab* array without tokens, their values are set to 0.

For the scanner table in the Fig 2.12 we need a two dimensional array, named *scan\_tab*. The first dimension is index from 0 to 18 (The number of states; index 0 will not be used) and the second dimension is index from 0 to 13 with 0 reprsenting white spaces (space, tab), 1 (newline), 2 ( / ), 3 (\*), and so on. For any integer *i* and *j, scan\_tab[i][j]* is a record with field names *action* and *newState.action* can take values of *move*, *recognize, error,* and if *action = move* means that the automata should move to the next state (the next state value is equal to the one stored at *scan\_tab[i][j] )*. If *action = recognize* means that *i* is a final state and the automata can not move to any other state *i* with the input character corresponding to the number *j.* We recognize a token! If *move = error* means that the automata can not get to any state from state *i* with a character corresponding to the number *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:**

**Output:**

**Side Effects:**

**Plan:**

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

Stmt\_list();

Match($$);

**Else**

Error

Type here

**End of Algorithm**

**Algorithm :**

**Input:**

**Output:**

**Side Effects:**

**Plan:**

**End of Algorithm**

**Algorithm :**

**Input:**

**Output:**

**Side Effects:**

**Plan:**

**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.