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 $$
    Stmt_list(input_token);
    Match($$,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.”

2.) 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.