PLDI

David Halliday

March 8, 2005

Contents

1 Lexical Analysis and Parsing **3**
- 1.1 Deterministic Finite State Machine . . . 3
- 1.2 Implementation of DFSM in Shell parser . . . 4
  - 1.2.1 Aims & changes . . . 4
  - 1.2.2 Code changes . . . 4
- 1.3 Implementation of LR-parsing table . . . 7
  - 1.3.1 Code changes . . . 7
  - 1.3.2 test a - Input File . . . 9
  - 1.3.3 test a - Output . . . 9
  - 1.3.4 test a - Conclusion . . . 10
  - 1.3.5 test b - Input File . . . 10
  - 1.3.6 test b - Output . . . 10
  - 1.3.7 test b - Conclusion . . . 11
- 1.4 Adding new rules to the grammar . . . 12
  - 1.4.1 Extending the LR-parser rules . . . 12
  - 1.4.2 Implementation of new rules . . . 13

2 Bison grammar - pl1/2 **17**
- 2.1 GOTO statemeant . . . 17
  - 2.1.1 changes to syn.y . . . 17
  - 2.1.2 changes to lex.l . . . 17
- 2.2 FOR statemeant . . . 17
  - 2.2.1 changes to syn.y . . . 17
  - 2.2.2 changes to lex.l . . . 17

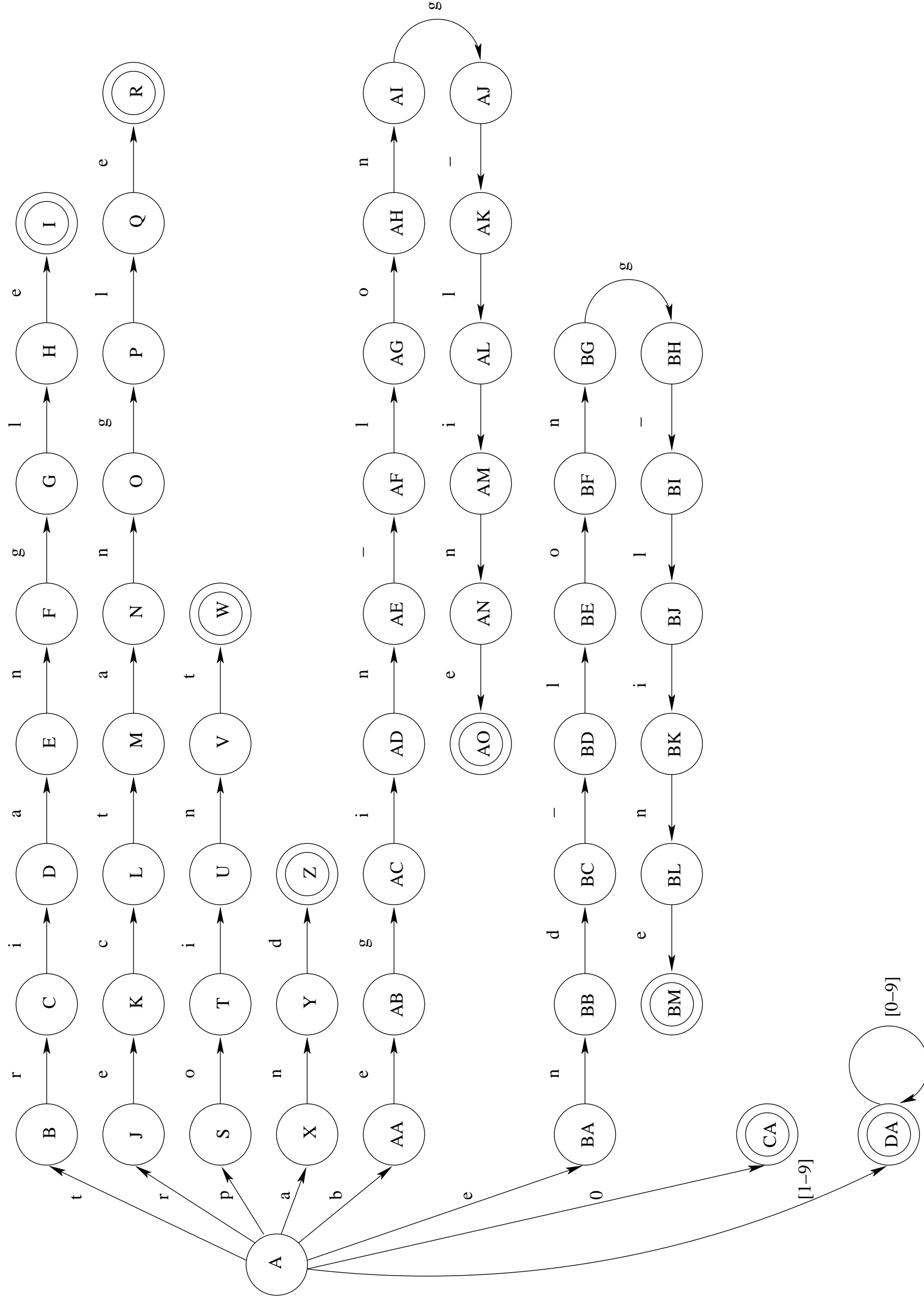
3 Programming Languages Study - Lua **18**
- 3.1 History & Influences . . . 18
- 3.2 Description of language . . . 19
- 3.3 Code Example . . . 20
  - 3.3.1 Code Listing . . . 20
  - 3.3.2 Breakdown of code . . . 20
- 3.4 The reason for creation . . . 21
- 3.5 Estimate of use . . . 21
- 3.6 Influencing Languages . . . 21
- 3.7 Comparison with Similar languages . . . 22
  - 3.7.1 Python . . . 22
  - 3.7.2 UnrealScript . . . 22

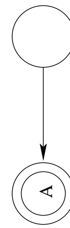
3.7.3 Guile 22
3.7.4 Summary 23
3.8 influences it has had on other languages 23

4 Appendices **25**
4.1 pl11/2 test programmes 25
4.1.1 lex.l code 25
4.1.2 syn.y code 27
4.2 Lua 5.0 license 30

1 Lexical Analysis and Parsing

1.1 Deterministic Finite State Machine





When given a space at the end of a command word the FSM returns to state ‘A’

1.2 Implementation of DFSM in Shell parser

1.2.1 Aims & changes

The aim was to implement the DFSM on page 3 into the lexical analyser so that the shell parser would recognise the tokens in the DFSM.

This was achieved by extending 3 functions within the application

setup_lexer_state_table ()

This function uses a table created as a 2 dimensional array, It Sets every state and input combination to "ERROR_STATE" before taking the valid combinations (such as in STATE_A being given a 't' is a valid combination) to point to the next state for that combination. allowing the table to be tested saying I am in STATE_A and i have been given a 't' what state do i go to or is this an error?

setup_lexer_output_table ()

Similar to the lexer_state_table this function uses a table created as a 2 dimensional array, The use of this table is not to find the next valid state but to only return a non-error value when the state and input combination deliminates a valid input such as being given a space after receiving the characters 'p' 'o' 'i' 'n' 't' deliminating a valid 'point' token.

setup_terminal_names ()

This uses only a 1 dimensional array to link the numerical values attached to the tokens in the lexical analyser to strings so that for example if TRIANGLE (remembering that this is an enumerated value so in a compiled state this points to a number) is found it can be sent to the table and return a string "triangle" that a user can understand.

1.2.2 Code changes

Note: Some coded changes for this area such as setting of enumerations and defining constants to the compiler have been omitted.

```
/* create the state table for mapping the current state and input character
 * to a new state */
void setup_lexer_state_table()
{
    int i, j;

    /* First, initialise the table so every transition is an error */
    for (i=0; i<num_lexer_states; i++)
        for (j=0; j<num_chars; j++)
            lexer_state_table[i][j] = ERROR_STATE;

    /*
     * a mapping of the DFSM into the code with each valid input combination
     * returning the next valid state
     */
  //STATE A (starting state
```

```
  lexer_state_table[STATE_A][T] = STATE_B;
  lexer_state_table[STATE_A][R] = STATE_J;
  lexer_state_table[STATE_A][P] = STATE_S;
  lexer_state_table[STATE_A][A] = STATE_X;
  lexer_state_table[STATE_A][B] = STATE_AA;
  lexer_state_table[STATE_A][E] = STATE_BA;
  lexer_state_table[STATE_A][NUM0]       = STATE_CA;
  lexer_state_table[STATE_A][NUM]        = STATE_DA;
  lexer_state_table[STATE_A][END_CHAR]   = END_STATE;
  lexer_state_table[STATE_A][WHITESPACE] = STATE_A;
//Triangle
  lexer_state_table[STATE_B][R] = STATE_C;
  lexer_state_table[STATE_C][I] = STATE_D;
  lexer_state_table[STATE_D][A] = STATE_E;
  lexer_state_table[STATE_E][N] = STATE_F;
  lexer_state_table[STATE_F][G] = STATE_G;
  lexer_state_table[STATE_G][L] = STATE_H;
  lexer_state_table[STATE_H][E] = STATE_I;
  lexer_state_table[STATE_I][WHITESPACE] = STATE_A;
  lexer_state_table[STATE_I][END_CHAR]   = END_STATE;
//Rectangle
  lexer_state_table[STATE_J][E] = STATE_K;
  lexer_state_table[STATE_K][C] = STATE_L;
  lexer_state_table[STATE_L][T] = STATE_M;
  lexer_state_table[STATE_M][A] = STATE_N;
  lexer_state_table[STATE_N][N] = STATE_O;
  lexer_state_table[STATE_O][G] = STATE_P;
  lexer_state_table[STATE_P][L] = STATE_Q;
  lexer_state_table[STATE_Q][E] = STATE_R;
  lexer_state_table[STATE_R][WHITESPACE] = STATE_A;
  lexer_state_table[STATE_R][END_CHAR]   = END_STATE;
//Point
  lexer_state_table[STATE_S][O] = STATE_T;
  lexer_state_table[STATE_T][I] = STATE_U;
  lexer_state_table[STATE_U][N] = STATE_V;
  lexer_state_table[STATE_V][T] = STATE_W;
  lexer_state_table[STATE_W][WHITESPACE] = STATE_A;
  lexer_state_table[STATE_W][END_CHAR]   = END_STATE;
//and
  lexer_state_table[STATE_X][N] = STATE_Y;
  lexer_state_table[STATE_Y][D] = STATE_Z;
  lexer_state_table[STATE_Z][WHITESPACE] = STATE_A;
  lexer_state_table[STATE_Z][END_CHAR]   = END_STATE;
//Begin-Long-Line
  lexer_state_table[STATE_AA][E] = STATE_AB;
  lexer_state_table[STATE_AB][G] = STATE_AC;
  lexer_state_table[STATE_AC][I] = STATE_AD;
  lexer_state_table[STATE_AD][N] = STATE_AE;
  lexer_state_table[STATE_AE][HYPHEN] = STATE_AF;
  lexer_state_table[STATE_AF][L] = STATE_AG;
  lexer_state_table[STATE_AG][O] = STATE_AH;
  lexer_state_table[STATE_AH][N] = STATE_AI;
  lexer_state_table[STATE_AI][G] = STATE_AJ;
  lexer_state_table[STATE_AJ][HYPHEN] = STATE_AK;
```

```
    lexer_state_table[STATE_AK][L] = STATE_AL;
    lexer_state_table[STATE_AL][I] = STATE_AM;
    lexer_state_table[STATE_AM][N] = STATE_AN;
    lexer_state_table[STATE_AN][E] = STATE_AO;
    lexer_state_table[STATE_AO][WHITESPACE] = STATE_A;
    lexer_state_table[STATE_AO][END_CHAR]   = END_STATE;
  //End-Long-Line
    lexer_state_table[STATE_BA][N] = STATE_BB;
    lexer_state_table[STATE_BB][D] = STATE_BD;
    lexer_state_table[STATE_BD][HYPHEN] = STATE_BE;
    lexer_state_table[STATE_BE][L] = STATE_BF;
    lexer_state_table[STATE_BF][O] = STATE_BG;
    lexer_state_table[STATE_BG][N] = STATE_BH;
    lexer_state_table[STATE_BH][G] = STATE_BI;
    lexer_state_table[STATE_BI][HYPHEN] = STATE_BJ;
    lexer_state_table[STATE_BJ][L] = STATE_BK;
    lexer_state_table[STATE_BK][I] = STATE_BL;
    lexer_state_table[STATE_BL][N] = STATE_BM;
    lexer_state_table[STATE_BM][E] = STATE_BN;
    lexer_state_table[STATE_BN][WHITESPACE] = STATE_A;
    lexer_state_table[STATE_BN][END_CHAR]   = END_STATE;
  //NUM0
    lexer_state_table[STATE_CA][WHITESPACE] = STATE_A;
    lexer_state_table[STATE_CA][END_CHAR]   = END_STATE;
  //NUM
    lexer_state_table[STATE_DA][NUM] = STATE_DA;
    lexer_state_table[STATE_DA][NUM0] = STATE_DA;
    lexer_state_table[STATE_DA][WHITESPACE] = STATE_A;
    lexer_state_table[STATE_DA][END_CHAR]   = END_STATE;

}

/* the output table stores the name of the terminal symbol to be output
 * on successfully completing a state
 */
void setup_lexer_output_table()
{
    /* clear out all the array first */
    int i,j;

    for (i=0; i<num_lexer_states; i++)
        for (j=0; j<num_chars; j++)
            lexer_output_table[i][j] = NO_TERMINAL;

    //set the required states and input combinations to assign a token
    lexer_output_table[STATE_I] [WHITESPACE] = TRIANGLE;
    lexer_output_table[STATE_R] [WHITESPACE] = RECTANGLE;
    lexer_output_table[STATE_W] [WHITESPACE] = POINT;
    lexer_output_table[STATE_Z] [WHITESPACE] = AND;
    lexer_output_table[STATE_AO][WHITESPACE] = BEGINLONGLINE;
    lexer_output_table[STATE_BN][WHITESPACE] = ENDLONGLINE;
    lexer_output_table[STATE_CA][WHITESPACE] = NUMBER;
    lexer_output_table[STATE_DA][WHITESPACE] = NUMBER;
    lexer_output_table[STATE_A] [END_CHAR]   = END_TOKEN;
```

```
}

void setup_terminal_names ()//only used for output of found tokens
{
    //fills the array full of error statemeants
    //(note: error outputs should never bee seen)
    int i;
    for(i = 1;i < num_terminals; i++)
        terminal_names[i] = "!!ERROR!!";

    terminal_names[TRIANGLE]      = "Triangle";
    terminal_names[RECTANGLE]     = "Rectangle";
    terminal_names[POINT]         = "Point";
    terminal_names[AND]           = "And";
    terminal_names[BEGINLONGLINE] = "Begin-Long-Line";
    terminal_names[ENDLONGLINE]   = "End-Long-Line";
    terminal_names[NUMBER]        = "Number";
    terminal_names[ERROR_STATE]   = "Error State";
    terminal_names[END_TOKEN]     = "End State";
}
```

For the testing output of these changes see section 1.3.2 on page 9 and section 1.3.5 on page 10.

1.3 Implementation of LR-parsing table

This was a theoretically simple implementation, most problems here were caused by incorrect enumeration and definition settings causing arrays to be incorrectly constructed and parsed.

setup_action_table ()

The action table is a two dimensional array containing action objects (which hold an action type and a numerical value). Similar to the lexer_state_table in that it checks a state and an input to see if they return an error or a valid next state.

setup_goto_table ()

This is similar to the action_table although it is used for the goto aspect of once something has been found, so for example if you are in STATE_6 and you have just finished a reduce on a point (reduce by rule 8) the table will be tested with 'STATE_6' and '8'and return '10' which is the next valid state to go to.

1.3.1 Code changes

Note: Some coded changes for this area such as setting of enumerations and defining constants to the compiler have been omitted.

```
/* ********** handle the action and goto tables ********** */
void setup_action_table ()
{
    int i, j;
```

```
    /* clear the action table */
    for (i=0; i<num_parser_states; i++)
        for (j=0; j<num_terminals; j++)
            action_table[i][j] = make_action( ERROR_ACTION, ERROR_STATE );

    action_table[STATE_0] [TRIANGLE]  = make_action( SHIFT_ACTION,  STATE_7 );
    action_table[STATE_0] [RECTANGLE] = make_action( SHIFT_ACTION,  STATE_6 );
    action_table[STATE_1] [END_TOKEN] = make_action( ACCEPT_ACTION, STATE_0 );
    action_table[STATE_2] [END_TOKEN] = make_action( REDUCE_ACTION, 1 );
    action_table[STATE_3] [END_TOKEN] = make_action( REDUCE_ACTION, 2 );
    action_table[STATE_3] [AND]       = make_action( SHIFT_ACTION,  STATE_9 );
    action_table[STATE_4] [AND]       = make_action( REDUCE_ACTION, 5 );
    action_table[STATE_4] [END_TOKEN] = make_action( REDUCE_ACTION, 5 );
    action_table[STATE_5] [AND]       = make_action( REDUCE_ACTION, 4 );
    action_table[STATE_5] [END_TOKEN] = make_action( REDUCE_ACTION, 4 );
    action_table[STATE_6] [POINT]     = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_7] [POINT]     = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_8] [NUMBER]    = make_action( SHIFT_ACTION,  STATE_12);
    action_table[STATE_9] [TRIANGLE]  = make_action( SHIFT_ACTION,  STATE_7 );
    action_table[STATE_9] [RECTANGLE] = make_action( SHIFT_ACTION,  STATE_6 );
    action_table[STATE_10][POINT]     = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_11][POINT]     = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_12][NUMBER]    = make_action( SHIFT_ACTION,  STATE_16);
    action_table[STATE_13][END_TOKEN] = make_action( REDUCE_ACTION, 3 );
    action_table[STATE_14][END_TOKEN] = make_action( REDUCE_ACTION, 6 );
    action_table[STATE_14][AND]       = make_action( REDUCE_ACTION, 6 );
    action_table[STATE_15][POINT]     = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_16][POINT]     = make_action( REDUCE_ACTION, 8 );
    action_table[STATE_16][END_TOKEN] = make_action( REDUCE_ACTION, 8 );
    action_table[STATE_16][AND]       = make_action( REDUCE_ACTION, 8 );
    action_table[STATE_17][AND]       = make_action( REDUCE_ACTION, 7 );
    action_table[STATE_17][END_TOKEN] = make_action( REDUCE_ACTION, 7 );
}

void setup_goto_table ()
{
    int i, j;

    /* clear the goto table */
    for (i=0; i<num_parser_states; i++)
        for (j=0; j<num_non_terminals; j++)
            goto_table[i][j] = LR_ERROR_STATE;

    goto_table[STATE_0] [1]  = STATE_1;
    goto_table[STATE_0] [2]  = STATE_2;
    goto_table[STATE_0] [3]  = STATE_2;
    goto_table[STATE_0] [4]  = STATE_3;
    goto_table[STATE_0] [5]  = STATE_3;
    goto_table[STATE_0] [6]  = STATE_4;
    goto_table[STATE_0] [7]  = STATE_5;
    goto_table[STATE_6] [8]  = STATE_10;
    goto_table[STATE_7] [8]  = STATE_11;
    goto_table[STATE_9] [2]  = STATE_13;
    goto_table[STATE_9] [3]  = STATE_13;
```

```
    goto_table[STATE_9] [4]  = STATE_3;
    goto_table[STATE_9] [5]  = STATE_3;
    goto_table[STATE_9] [6]  = STATE_4;
    goto_table[STATE_9] [7]  = STATE_5;
    goto_table[STATE_9] [9]  = STATE_22;
    goto_table[STATE_10][8]  = STATE_14;
    goto_table[STATE_11][8]  = STATE_15;
    goto_table[STATE_15][8]  = STATE_17;
}
```

1.3.2 test a - Input File

```
rectangle point 0 0 point 10 5
```

1.3.3 test a - Output

```
 ***** Running the Lexical Analyser *****

Token: Rectangle
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End State

VALID sequence
Current action 0 6
Shift 6
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 1 6
Current action 1 5
Current action 1 2
Current action 1 1
Current action 3 0see section~\ref{sec:lexcode} on page~\pageref{sec:lexcode}.
LR Parser accepted input
```

1.3.4 test a - Conclusion

The test completed successfully. the parser was also tested with incorrect input showing that errors would be generated if the input was not “Valid code”

1.3.5 test b - Input File

```
triangle point 0 0 point 2 2 point 0 3 and
rectangle point 5 5 point 0 2
```

1.3.6 test b - Output

```
 ***** Running the Lexical Analyser *****

Token: Triangle
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: And
Token: Rectangle
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End State

VALID sequence
Current action 0 7
Shift 7
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
```

```
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 1 7
Current action 1 4
Current action 0 9
Shift 9
Current action 0 6
Shift 6
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 1 6
Current action 1 5
Current action 1 2
Current action 1 3
Current action 1 1
Current action 3 0
LR Parser accepted input
```

1.3.7 test b - Conclusion

The test completed successfully. the parser was also tested with incorrect input showing that errors would be generated if the input was not “Valid code”

1.4 Adding new rules to the grammar

1.4.1 Extending the LR-parser rules

Rules

1	S	→	D
2	D	→	I
3	D	→	I and D
4	I	→	T
5	I	→	R
6	R	→	rectangle P P
7	T	→	triangle P P P
8	P	→	point num num
9	L	→	begin-long-line P P L^1
10	L^1	→	end-long-line
11	L^1	→	P L^1
12	I	→	L

LR-parse table

state	and	rect	tri	point	num	$	beg-l-l	end-l-l	S	D	I	R	T	P	L	L^1
0		s6	s7				s18		1	2	3	4	5		22	
1						acc										
2						r1										
3	s9					r2										
4	r5					r5										
5	r4					r4										
6				s8										10		
7				s8										11		
8					s12											
9		s6	s7				s18			13	3	4	5		22	
10				s8										14		
11				s8										15		
12					s16											
13						r3										
14	r6					r6										
15				s8										17		
16	r8			r8		r8										
17	r7					r7										
18				s8										19		
19				s8										20		
20				s8				s21						20		23
21	r9					r9										
22	r12					r12										
23	r10					r10										

1.4.2 Implementation of new rules

Code changes

Note: Some coded changes for this area such as setting of enumerations and defining constants to the compiler have been omitted.

```
/* ********** handle the action and goto tables ********** */
void setup_action_table ()
{
    int i, j;

    /* clear the action table */
    for (i=0; i<num_parser_states; i+ ***** Running the Lexical Analyser *****

Token: Begin-Long-Line
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End-Long-Line
Token: And
Token: Rectangle
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End State

VALID sequence
Current action 0 18
Shift 18
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
```

```
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 2 57
LR Parsing Error
+)
        for (j=0; j<num_terminals; j++)
            action_table[i][j] = make_action( ERROR_ACTION, ERROR_STATE );

    action_table[STATE_0] [BEGINLONGLINE] = make_action( SHIFT_ACTION,  STATE_18 );
    action_table[STATE_9] [BEGINLONGLINE] = make_action( SHIFT_ACTION,  STATE_18 );
    action_table[STATE_18][POINT]         = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_19][POINT]         = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_20][POINT]         = make_action( SHIFT_ACTION,  STATE_8 );
    action_table[STATE_20][ENDLONGLINE]   = make_action( SHIFT_ACTION,  STATE_23 );
    action_table[STATE_21][AND]           = make_action( REDUCE_ACTION, 9);
    action_table[STATE_21][END_TOKEN]     = make_action( REDUCE_ACTION, 9);
    action_table[STATE_22][AND]           = make_action( REDUCE_ACTION, 12);
    action_table[STATE_22][END_TOKEN]     = make_action( REDUCE_ACTION, 12);
    action_table[STATE_23][AND]           = make_action( REDUCE_ACTION, 10);
    action_table[STATE_23][END_TOKEN]     = make_action( REDUCE_ACTION, 10);
}

void setup_goto_table ()
{
    int i, j;

    /* clear the goto table */
    for (i=0; i<num_parser_states; i++)
        for (j=0; j<num_non_terminals; j++)
            goto_table[i][j] = LR_ERROR_STATE;

    goto_table[STATE_0] [1]  = STATE_1;
    goto_table[STATE_0] [2]  = STATE_2;
    goto_table[STATE_0] [3]  = STATE_2;
    goto_table[STATE_0] [4]  = STATE_3;
    goto_table[STATE_0] [5]  = STATE_3;
    goto_table[STATE_0] [6]  = STATE_4;
    goto_table[STATE_0] [7]  = STATE_5;
    goto_table[STATE_0] [9]  = STATE_22;
    goto_table[STATE_6] [8]  = STATE_10;
    goto_table[STATE_7] [8]  = STATE_11;
    goto_table[STATE_9] [2]  = STATE_13;
    goto_table[STATE_9] [3]  = STATE_13;
    goto_table[STATE_9] [4]  = STATE_3;
    goto_table[STATE_9] [5]  = STATE_3;
    goto_table[STATE_9] [6]  = STATE_4;
    goto_table[STATE_9] [7]  = STATE_5;
    goto_table[STATE_9] [9]  = STATE_22;
    goto_table[STATE_10][8]  = STATE_14;
    goto_table[STATE_11][8]  = STATE_15;
    goto_table[STATE_15][8]  = STATE_17;
    goto_table[STATE_18][8]  = STATE_19;
```

```
    goto_table[STATE_19][8]  = STATE_20;
    goto_table[STATE_20][8]  = STATE_20;
    goto_table[STATE_20][10] = STATE_23;
    goto_table[STATE_20][11] = STATE_23;

}
```

Test c - Input File

```
begin-long-line point 0 0 point 1 1 point 1 0 end-long-line and
rectangle point 5 5 point 2 5
```

Test c - Output

```
 ***** Running the Lexical Analyser *****

Token: Begin-Long-Line
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End-Long-Line
Token: And
Token: Rectangle
Token: Point
Token: Number
Token: Number
Token: Point
Token: Number
Token: Number
Token: End State

VALID sequence
Current action 0 18
Shift 18
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 1 8
```

```
Current action 0 8
Shift 8
Current action 0 12
Shift 12
Current action 0 16
Shift 16
Current action 2 57
LR Parsing Error
```

Test c Conclusion

As can be seen from the "LR Parsing Error" the code implementation has not been successfull, the application errors when trying to reduce the last point before processing the 'end-long-line'.

2 Bison grammar - pl1/2

lex.l code listing see section 4.1.1 on page 25.

syn.y code listing see section 4.1.2 on page 27.

2.1 GOTO statemeant

2.1.1 changes to syn.y

added to the top:

```
%token GOTOSY
```

added to stat:

```
    GOTOSY IDENT     { printf("found: GOTO stat\n"); }
|
    IDENT ':' stats  { printf("found: LABEL stat\n"); }
|
```

2.1.2 changes to lex.l

added to the end:

```
goto        { return GOTOSY; }
```

2.2 FOR statemeant

2.2.1 changes to syn.y

added to the top:

```
%token FORSY STEPSY TOSY
```

added to stat:

```
    FORSY IDENT ASSIGNSY expr TOSY expr STEPSY expr DOSY stats ENDSY
    { printf("found: FOR stat - step\n"); }
|
    FORSY IDENT ASSIGNSY expr TOSY expr DOSY stats ENDSY
    { printf("found: FOR stat - no step\n"); }
|
```

2.2.2 changes to lex.l

added to the end:

```
for         { return FORSY; }
step        { return STEPSY; }
to          { return TOSY; }
```

3 Programming Languages Study - Lua

Lua means moon in Portuguese and is pronounced LOO-ah[Lua-website, 2005].

> Lua is a powerful light-weight programming language designed for extending applications. Lua is also frequently used as a general-purpose, stand-alone language. Lua is free software.

Lua is designed and implemented by a team at Tecgraf, the Computer Graphics Technology Group of PUC-Rio (the Pontifical Catholic University of Rio de Janeiro in Brazil). [Lua-website, 2005]

3.1 History & Influences

Below is an extract from the Abstract of [Roberto Ierusalimschy, 2001] Summarising the history of the language.

> "Since its inception, in 1993, the Lua programming language has gone far beyond our most optimistic expectations. In this paper, we describe the trajectory of Lua, from its creation as an in-house language for two specific projects, until Lua 4.0, released in November 2000. We discuss the evolution of some of its concepts and the main landmarks in its implementation."

The language was devised by a committee, however unlike Algol 68, and Ada (also devised by committees) the committee was very small (only three people) and they aimed small. One of the general design aims of the language is simplicity. The small size, portability and its speed are all a product of this.

The language came about after combining the requirements for two far simpler languages used for very different applications called DEL (data entry language) and SOL (Simple Object Language also meaning Sun in Portuguese). The uses for these (very simple) languages were expanding and so they were combined with more coming from Sol (it being the more complex language).

The development of the language whilst always being Open Source was not always free software in the GNU sense. Starting with Lua 5.0, Lua is licensed under the terms of the MIT license and summarised in the appendix, see section 4.2 on page 30.

The language remained relatively unheard of and therefore only used by a very small number of people. This was partly due to the restrictive license on version 1 (free for academic uses, but not for commercial uses). It was only with version 2 of Lua (released in 1995) that the license changed allowing for companies to use the language to develop commercial applications with Lua that interest really started to increase. This meant there was more “”value” in learning the language. Also with the introduction of version 2 there were a number of changes that meant that the language was not fully compatible with previous versions (meaning that previously written code would not run with that version of the language). This change was done so that the language would not be held back by legal issues preventing new ideas.

In May 1996 Lua 2.4 was released, the major change with this edition was the addition of Luac a Lua compiler. [Roberto Ierusalimschy, 2001]

> This program pre-compiles Lua code and saves bytecode and string tables to a binary file. The format of this file was chosen to be easily loaded and portable across different platforms. With Luac, programs can avoid passing and code generation at run-time, which can be costly, especially for large, static programs such as graphical metafiles.

The big improvements offered by the Luac compiler were in faster loading, off-line syntax checking and code protection.

In 1996 Lua "took off" after being published in two very different publications.

Software: Practice & Experience An academic paper about Lua

Dr. Dobb's A magazine featured an article about the language

Soon after these were published interest in the language grew. One of the biggest companies to take interest in Lua was LucasArts who wanted to replace the existing scripting language (SCUMM) with Lua in the game "Grim Fandango" which was released in 1997. apparently "A TREMENDOUS amount of this game is written in Lua" - Bret Mogilefsky (LucasArts)

The use of Lua in a computer game led other games developers to take an interest in the language and now games development companies are interested in people who are familiar with the Lua language.

The major developments with Lua from version 2 - 5 have been the addition of new features and many changes in the way it can be extended with "Tag methods".

3.2 Description of language

Lua is designed as an interpreted language with dynamic type checking. However there is Luac which is a compiler that does the type checking but does not produce a binary file that can be executed, the file still needs to be used with the Lua programme.

The largest common use of Lua is building it into other programmes to extend them. There is a powerfull C API that allows programmes to use Lua for scripting in larger applications. Lua has a large following in the Games development industry for its scripting uses. It can also be used to read INI like config files as Lua programme files that specify assignmeant of values to objects look like INI files. A text file can be created and passed by Lua with the values passed to the application, this means that an application developer does not have to write a full in depth class for input from an ASCII "options" file.

3.3 Code Example

Below is some code that I wrote to familiarise myself with the Lua programming language[1]. it outputs the lyrics to the old "Ten Green bottles" children's song. I got the lyrics from the [unknown, 2005, KiDiddles website].

3.3.1 Code Listing

```
01 action = " hanging on the wall"
02 run = 1
03 function greenb(x)
04   for i = x,0,-1 do
05     if i == 1 then
06       output = (i .. " green bottle" .. action)
07     elseif i == 0 then
08       output = ("no green bottles" .. action)
09     else
10       output = (i .. " green bottles" .. action)
11     end
12
13     if i>0 then
14       if run ~= 1 then
15         print(output)
16         print()
17       end
18
19       run = run + 1
20
21       print(output)
22       print(output)
23       print("If one green bottle should accidently fall")
24       print("There'll be...")
25     else
26       print(output)
27     end
28   end
29 end
30
31 greenb(10)
```

3.3.2 Breakdown of code

01 assignmeant Here is an assignmeant of a string to a variable. In this case the string is " hanging on the wall" with the variable 'action'.

02 assignmeant This is the assignment of an integer to the variable 'run'. Note that there is not type definitions in the language.

03 - 29 Function declaration This is the declaration of the function 'greenb' which takes one argument which is referred to within the function as 'x'.

[1] One of the beauties of the language is that unlike many other languages semi colons are optional so the compiler will never complain about missing semi colons ;-).

04 - 28 for statemeant This is a for statemeant that repeats from the value of 'x' counting down till 'x' is equal to 0.

05 - 11 if statemeant This is an if statement with simple boolean comparisons featuring 'elsif' and 'else' statements

06 + 08 + 10 Concatenation of strings These lines (selected with the previous If statement) all feature another assignment of a string however this string is made up of concatenating the number stored in 'i' to a new string and the original 'action' string one useful function in the language is this concatenation using two dots '..' which is fast and easy to programme.

21 print() statement print is used to output a line of text (or a value) to the screen. It is very flexible and can be used for simple mathematical functions for example 'print(3 + 4)' would give the output '7' but 'print("3 + 4")' would allow you to output the text '3 + 4' if you wanted (note the addition of the quotes to signify it is a string).

31 calling a function In this instance I have included a call to the function within the code that calls the 'greenb() function giving it the argument 10' often you would define the functions in the file then call them from inside the interpreter or from a C programme.

3.4 The reason for creation

The language was created to fill the niche of a small, highly profitable language that is easy for non programmers to learn. Based on the SOL language more details are available in section 3.1 on page 18. And it has expanded as users have requested more functionality.

3.5 Estimate of use

In 2001 it was estimated that there were 500 active programmers who used the language regularly and about 10,000 programmers who were familiar with the language. The list of companies who use Lua in their applications is ever growing including big names in Games developmeant both open source and commercial applications.

3.6 Influencing Languages

The language itself draws from a number of languages, taking the features and ideas that it needs. Because the use of the language is rather unique the language itself is very unique. Some of the languages it draws from are:

BiBTeX Big influence on the original SOL language for layout and syntax

UIL (User Interface Language) Big influence on the original SOL language for layout and syntax

C++ The idea of allowing a local variable to be declared only where it is needed

Modula Syntax influence (while, if, and repeat until)

CLU Multiple assignment and multiple returns from function calls

3.7 Comparison with Similar languages

Whilst there are a large number of scripting languages around the two big names that are often compared with Lua are Python and UnrealScript.

When it comes to comparing languages the most common method is benchmarking (running similar programmes under different languages in the same conditions and comparing execution times, filesizes memory usage etc...) and in view of this I would like to quote [Unknown, 2005] "there are lies, damn lies, and benchmarks".

3.7.1 Python

Most of this information is from [luausers wiki, 2005b] and [Unknown, 2005].

In [Unknown, 2005] where benchmark results are listed for comparison on a craps style score card. There are 25 different tests the average positions on the score card are listed below.

Benchmark Type	Lua	Lua5	Python
CPU	22	20	23
Memory Usage	16	13	25
Lines Of Code	14	13	12

These results show that version 4 of Lua was a very close match for Python but the updates for version 5 of Lua push it far ahead of Python.

As far as usage Python has a much larger user base and a more extensive range of libraries where as Lua is regarded for its small size, ease of use and high performance.

In 1997 the only reason why Lua was not used for a particular project (which Python was chosen for) was the fact that it was "young, and lacked character and maturity" [Logajan, 1997]. Perhaps now after 7 years the opinion would be different.

3.7.2 UnrealScript

Most of this information is from [luausers wiki, 2005c] which uses the same testing methods as [Unknown, 2005].

The results are that again Lua has better performance than UnrealScript. The reason that this comparison is interesting / important is that UnrealScript is a scripting language designed and written specifically for computer game scripting by a computer game development company [EpicGames, 2005]. UnrealScripting is known to be poor on performance in comparison to normal C code whereas Lua is renowned for its performance.

3.7.3 Guile

This is possibly the closest competitor to Lua. In [Unknown, 2005] where benchmark results are listed for comparison on a craps style score card. There are 25 different tests the average positions on the score card are listed below.

Benchmark Type	Lua	Lua5	Guile
CPU	22	20	33
Memory Usage	16	13	28
Lines Of Code	14	13	14

Again the benchmarks show Lua as being superior in performance.

The Lua users Wiki FAQ [luausers wiki, 2005a] actually says this about the comparison:

> For those comfortable with Scheme, Guile may be a better solution. Lua has a bent towards scripting and configuration by "non-programmers", where infix syntax may be more appealing. Lua is also often used as a stand-alone language, which in the Scheme world is more the domain of scsh than Guile. Guile is significantly larger than Lua.

3.7.4 Summary

Lua ranks highly in benchmarks and from a technical point of view is very fast, small and powerful for scripting. It has a very straightforward code syntax so it is easy for people to learn who are not necessarily programmers. What it lacks is the large number of libraries that many languages have and currently only a comparatively small number of people are familiar with the language.

3.8 influences it has had on other languages

Lua is a relatively new language and has only been particularly popular in small circles for only a few years. Because of this it has yet to have significant influence on any new languages, however it has changed the way that some programmes handle scripting. Also languages like Lua that are good for scripting are good for Graphic Designers to learn so they can easily script actions when testing and demonstrating computer game graphics. Some people recommend to all people aiming to work in computer graphics design that they need to learn a scripting language as it is becoming more common for companies to require it when people are doing graphic design for computer games.

References

[EpicGames, 2005] EpicGames (2005). *EpicGames.* `<http://www.epicgames.com/>`. The epic games website.

[Logajan, 1997] Logajan, J. (1997). *A Tale of Five Languages.* `<http://sunsite.bilkent.edu.tr/pub/languages/pub/doc/langquest.html>`. A comparison of 5 scripting languages for a particular task.

[Lua-website, 2005] Lua-website (2005). *Lua Website.* `<http://www.lua.org/>`. The Lua project website.

[luausers wiki, 2005a] luausers wiki (2005a). *Lua Faq.* `<http://lua-users.org/wiki/LuaFaq>`. A wiki article comparing Lua with Unreal Script.

[luausers wiki, 2005b] luausers wiki (2005b). *Lua Versus Python.* `<http://lua-users.org/wiki/LuaVersusPython>`. A wiki article comparing Lua with Python.

[luausers wiki, 2005c] luausers wiki (2005c). *Lua Versus Unreal Script.* `<http://lua-users.org/wiki/LuaVersusUnrealScript>`. A wiki article comparing Lua with Unreal Script.

[Roberto Ierusalimschy, 2001] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, W. C. (2001). The evolution of an extension language: A history of lua. In *SBLP 2001 invited paper*, `<http://www.lua.org/history.html>`. Lua Team, TeCGraf, Department of Computer Science, PUC-Rio. A paper on the history of the Lua programming language.

[Unknown, 2005] Unknown (2005). *The Great Win32 Computer Language Shootout.* `<http://dada.perl.it/shootout>`. A comparison of scripting languages under windows.

[unknown, 2005] unknown (2005). *KIDiddles: Song Lyrics: Ten Green Bottles.* `<http://www.kididdles.com/mouseum/t050.html>`. Lyrics to 'Ten Green Bottles' on the KIDiddles website.

4 Appendices

4.1 pl11/2 test programmes

4.1.1 lex.l code

```
%{
#include "syn.tab.h"
%}

DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9]*


%%
"//".*\n    printf("skipping comment\n");
[ \t\n]+    /* eat up whitespace */


{DIGIT}+    {  yylval = atoi(yytext);
               return NUMBER;
            }
begin       { return BEGINSY; }
do          { return DOSY; }
end         { return ENDSY; }
program     { return PROGSY; }
if          { return IFSY; }
while       { return WHILESY; }
then        { return THENSY; }
else        { return ELSESY; }
var         { return VARSY; }
const       { return CONSTSY; }
write       { return WRITESY; }
read        { return READSY; }

repeat      { return REPEATSY; }
until       { return UNTILSY; }

goto        { return GOTOSY; }

for         { return FORSY; }
step        { return STEPSY; }
to          { return TOSY; }

{ID}        { return IDENT; }
":="        { return ASSIGNSY; }
"<="        { return LEQ; }

"+"|"-"|"*"|"/"   { return yytext[0]; }
"."|"("|")"|";"   { return yytext[0]; }
```

```
"="|"#"|"<"|">"   { return yytext[0]; }
":"|","           { return yytext[0]; }

.           printf( "Unrecognized character: %s\n", yytext );
%%
```

4.1.2 syn.y code

```
%{
/* pl 1/2

    this is compiler for a very little language called pl1/2
    (pronounced pl-a-half).
    this is a syntax checker version
*/
#include <stdio.h>
extern int fatalerror;
%}

%start prog
%token IDENT NUMBER
%token PROGSY BEGINSY IFSY WHILESY THENSY ELSESY DOSY
%token ENDSY ASSIGNSY CONSTSY VARSY WRITESY READSY
%token REPEATSY UNTILSY
%token GOTOSY
%token FORSY STEPSY TOSY
%nonassoc '<' '>' '=' '#' LEQ
%left '+' '-'
%left '*'
%left NEG  /* used for unary minus - */
%%
//  beginning of grammar rules
//  ==========================
prog:
      PROGSY ';' decls BEGINSY stats ENDSY '.'
          { /* found a program, */
     if (fatalerror > 0 )
    printf("errors detected\n");
else
           printf("program found\n");
  }
   ;

decls:
      /* empty */ { printf("found: decls\n"); }
   |
decls decl { printf("found: decls decl\n"); }
   ;

stats:
      stat { printf("found: stat\n"); }
   |
      stats ';' stat  { printf("found: stats ; stat\n"); }
   ;

decl:
```

```
      CONSTSY IDENT '=' NUMBER ';' { printf("found: CONST decl\n"); }
   |
      VARSY idlist ':' type ';' { printf("found: VAR decl\n"); }
   |
      error ';' { fatal("error in declarations",""); }
   ;

idlist:
      IDENT  { printf("found: IDENT in idlist\n"); }
   |
      idlist ',' IDENT { printf("found: idlist , IDENT\n"); }

stat:
      /* empty */
   |
      IDENT ASSIGNSY expr   { printf("found: ASSIGNment\n"); }
   |
      WHILESY expr DOSY stats ENDSY  { printf("found: WHILE stat\n"); }
   |
      IFSY expr THENSY stats ENDSY { printf("found: IF THEN stat\n"); }
   |
      IFSY expr THENSY stats ELSESY stats ENDSY
      { printf("found: IF THEN ELSE stat\n"); }
   |
      READSY '(' IDENT ')'  { printf("found: READ stat\n"); }
   |
      WRITESY '(' expr ')'    { printf("found: WRITE stat\n"); }
   |
      REPEATSY stats UNTILSY expr { printf("found: REPEAT UNTIL stat\n"); }
   |
      GOTOSY IDENT     { printf("found: GOTO stat\n"); }
   |
      IDENT ':'        { printf("found: LABEL stat\n"); }
   |
      FORSY IDENT ASSIGNSY expr TOSY expr STEPSY expr DOSY stats ENDSY
      { printf("found: FOR stat - step\n"); }
   |
      FORSY IDENT ASSIGNSY expr TOSY expr DOSY stats ENDSY
      { printf("found: FOR stat - no step\n"); }
   |
      error { fatal("error in statement",""); }
   ;

expr:
      expr '+' expr   { printf("found: expr + expr\n"); }
   |
      expr '*' expr   { printf("found: expr * expr\n"); }
   |
      expr '-' expr     { printf("found: expr - expr\n"); }
   |
```

```
      '(' expr ')'   { printf("found: (expr) \n"); }
   |
      IDENT        { printf("found: IDENT \n"); }
   |
      NUMBER       { printf("found: NUMBER \n"); }
   |
      '-' expr %prec NEG       { printf("found: - expr\n"); }
   |
      expr '<' expr        { printf("found: expr < expr\n"); }
   |
      expr LEQ expr        { printf("found: expr <= expr\n"); }
   |
      expr '>' expr        { printf("found: expr > expr\n"); }
   |
      expr '=' expr          { printf("found: expr = expr\n"); }
   |
      expr '#' expr      { printf("found: expr # expr\n"); }
   |
      error
         { fatal("error in expression",""); }
   ;

type:
      IDENT      { printf("found: a type identifier\n"); }
   |
      error
         { fatal("error in type",""); }
   ;

%%

int fatalerror=0;

int fatal(char *s){
    fatalerror++;
fprintf(stderr,"%s\n",s);
}
int yyerror(char *s){
fprintf(stderr,"%s\n",s);
}

int main() {
    printf("starting...\n");
    yyparse();
}
```

4.2 Lua 5.0 license

Copyright 1994-2004 Tecgraf, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

`<http://www.lua.org/license.html>`