**William Collins 040652633**

**Course: CST 8152 – Compilers**

**Lab Section: 11**

**Assignment: 2**

**Professor: Svillen Ranev**

**Due Date: June 30, 2011**

**Date: June 30, 2011**

**Contents**

Platypus BNF Grammar

RE – TD - TT

scanner.c

table.h

Test Files

/\* File Name: buffer.h

**Scanner.c**

\* Version: 1.12

\* Author: William Collins (040652633) && Svillen Ranev

\* Course: CST8152 - Compilers

\* Assignment: 2

\* Date: June 30, 2011

\* Professor: Svillen Ranev

\* Purpose: A token driven lexical analyzer used to get tokens from a file

\*/

#include <stdio.h> /\* standard input / output \*/

#include <ctype.h> /\* conversion functions \*/

#include <stdlib.h> /\* standard library functions \*/

#include <string.h> /\* string functions \*/

#include <limits.h> /\* integer type constants \*/

#include <float.h> /\* floating-point type constants \*/

#include <math.h>

/\*#define NDEBUG to suppress assert() call \*/

#include <assert.h> /\* assert() prototype \*/

/\* project header files \*/

#include "buffer.h"

#include "token.h"

#include "table.h"

#define DEBUG /\* for conditional processing \*/

#undef DEBUG

/\*Globals\*/

extern Buffer \* str\_LTBL; /\*String literal table \*/

int line; /\* current line number of the source code \*/

extern int scerrnum; /\* defined in platy\_st.c - run-time error number \*/

/\* Function prototypes \*/

static int char\_class(char c); /\* character class function \*/

static int get\_next\_state(int, char, int \*); /\* state machine function \*/

static int iskeyword(char \* kw\_lexeme); /\*keywords lookup functuion \*/

static long atool(char \* lexeme); /\* converts octal string to decimal value \*/

static int isEOF(unsigned char c); /\*tests a symbol for end of file\*/

/\*Token generation function prototypes. Lazy people unite! \*/

/\* NOTE: I know you don't like function calls, but I do have a reason for using

them. These functions make the code much easier to read, and promote code resuse.

The performance loss in using function calls where I have used them is very

minimal, and would only be a factor in embedded systems. So please don't take away marks. Sincerely, #V

PS. Thanks for reading my long comment

PPS. Also, to make you even more upset with my program, instead of

returning a pointer to a token I am letting it return a copy of the

token. Again, it makes it look clean and simple, and I like clean and simple.

Just be aware that I understand why my code may lose performance.

If n is the number of tokens in a file, then my functions are called once per token.

O(n). It's scalable.

I'll be quiet now. And yes, I am aware of the irony of having a super long ugly

comment telling you how I like my code to look pretty.\*/

Token createErrorToken(unsigned int msg\_len, const char \*err\_msg);

Token createIntValToken(int code, int attribute);

Token createRelOpToken(Rel\_Op attribute);

Token createEmptyToken(int code);

Token createArrOpToken(Arr\_Op attribute);

Token createLogOpToken(Log\_Op attribute);

Token createStringToken(int code, char \*attribute);

Token createFltValToken(float attribute);

/\*

\* Purpose: Initialized the scanner for use

\* Author: Svillen Ranev

\* History/Versions: 1.0

\* Parameters: buf - a pointer to the buffer to initialize the scanner with

\*/

void scanner\_init(Buffer \*buf)

{

ca\_addc(buf, '\0'); /\* in case EOF is not in the buffer \*/

b\_reset(str\_LTBL); /\* reset the string literal table \*/

line = 1; /\*set the source code line number to 1\*/

}

/\*

\* Purpose: Scans the buffer and returns the next token

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: pB - pointer to the buffer to scan

\* Return value: The token found

\*/

Token malpar\_next\_token(Buffer \* sc\_buf)

{

Token t; /\*token to be returned \*/

unsigned char c; /\* input symbol \*/

Buffer \*lex\_buf; /\* temporary buffer for holding lexemes \*/

int accept = NOAS; /\* Not Accepting State \*/

int state = 0; /\* Start state in the Transition Table \*/

int lexstart; /\* current lexeme start offset \*/

static int forward = 0; /\* current input char offset \*/

char \*text = sc\_buf->ca\_head; /\*A shortcut to the data in the buffer\*/

char re\_msg[] = "RUN TIME ERROR: "; /\*runtime error message\*/

/\* An assertion seems most appropriate. No buffer coming in here should ever be NULL\*/

assert(sc\_buf != NULL);

while (1) {

c = text[forward++];

/\*Handle white space\*/

if (c == ' ' || c == '\t'){

continue;

}

/\*Handle the new line\*/

if (c == '\n'){

line++;

continue;

}

/\*Handle Comments\*/

if (c == '!'){

/\*Handle the "Not equals" case\*/

if (text[forward] == '='){

forward++;

return createRelOpToken(NE);

}

/\*If not a comment, then we have an error\*/

if (text[forward] != '<'){

t = createErrorToken(2, text + forward -1);

forward++;

}

/\*It is a comment, skip everything until we reach a new line\*/

while (text[forward] != '\n' && !isEOF(text[forward])){

forward++;

}

/\*Now if we had an error, let's return the token\*/

if (t.code == ERR\_T) {

return t;

}

continue;

}

/\*Check to see if we're at end of file. \*/

if (isEOF(c)){

return createEmptyToken(SEOF\_T);

}

/\*Logical Operators\*/

if (c == '.') {

int i = forward;

/\* We found .AND.\*/

if (text[i++] == 'A' && text[i++] == 'N' && text[i++] == 'D' && text[i++] == '.'){

forward = i;

return createLogOpToken(AND);

}

i = forward; /\*need to reset our offset\*/

/\* We found .OR. \*/

if (text[i++] == 'O' && text[i++] == 'R' && text[i++] == '.'){

forward = i;

return createLogOpToken(OR);

}

/\*We have an error\*/

return createErrorToken(1, text + forward-1);

}

/\*String Literal\*/

if (c == '"'){

int i;

/\*Find out if we have a good string or a bad string\*/

for (i = forward; !isEOF(text[i]) && text[i] != '"'; i++);

/\* The string is a good string, so add it to our literal table\*/

if (text[i] == '"'){

int j;

/\*Add the string to the literal table, taking into account line numbers\*/

for (j = 1; forward < i; forward++, j++){

if (text[forward] == '\n') {

line++;

}

str\_LTBL = ca\_addc(str\_LTBL, text[forward]);

}

forward++;

/\*Every string literal has a null terminator\*/

str\_LTBL = ca\_addc(str\_LTBL, '\0');

/\*catch any run-time errors\*/

if (str\_LTBL == NULL){

scerrnum = 1;

return createErrorToken(strlen(re\_msg), re\_msg);

}

/\*Set up the mark for the next string literal\*/

ca\_setmark(str\_LTBL, str\_LTBL->addc\_offset);

return createIntValToken(STR\_T, ca\_getmark(str\_LTBL) - j);

}

forward--; /\* For a bad string we need to include the front quote\*/

/\*Now we can handle a string literal error \*/

for (i = 0; !isEOF(text[forward]); i++, forward++);

t = createErrorToken(i, text+forward-i);

/\*Now that we have the string, if it is too long we substitute some dots\*/

if (i >= ERR\_LEN){

t.attribute.err\_lex[ERR\_LEN - 3] = '.';

t.attribute.err\_lex[ERR\_LEN - 2] = '.';

t.attribute.err\_lex[ERR\_LEN - 1] = '.';

}

return t;

}

/\* Variable Identifiers, Integer Literals, Octal Literals\*/

if (isalnum(c)){

int i;

ca\_setmark(sc\_buf, forward-1);

/\*Finite state machine\*/

for (i = forward; !accept; i++) {

state = get\_next\_state(state, c, &accept);

c = text[i];

}

/\*The value of ASWR and ASNR will decide how far to go back\*/

forward = i - as\_table[state];

/\*create the buffer to hold the lexeme temporarily\*/

lex\_buf = b\_create(VID\_LEN, 10, 'a');

/\*Add the lexeme to the buffer\*/

for(lexstart = ca\_getmark(sc\_buf) ; lexstart < forward; lexstart++){

lex\_buf = ca\_addc(lex\_buf, text[lexstart]);

}

lex\_buf = ca\_addc(lex\_buf, '\0');

/\*Make sure we didn't have any problems with our buffer\*/

if (lex\_buf == NULL){

scerrnum = 1;

return createErrorToken(strlen(re\_msg), re\_msg);

}

/\*Go to our appropriate accepting state function \*/

t = aa\_table[state](lex\_buf->ca\_head);

b\_destroy(lex\_buf);

return t;

}

/\*Equals sign\*/

if (c == '='){

/\*If followed by another, we have a relational op\*/

if (text[forward] == '='){

forward++;

return createRelOpToken(EQ);

}

/\*It is an assignment operator\*/

return createEmptyToken(ASS\_OP\_T);

}

/\*Some Relational Operators\*/

if (c == '<') {

/\*We have a string catenation operator\*/

if (text[forward] == '<') {

forward++;

return createEmptyToken(SCC\_OP\_T);

}

/\*We have a relational operator\*/

return createRelOpToken(LT);

}

if (c == '>') { return createRelOpToken(GT); }

/\*Parentheses and braces\*/

if (c == '('){ return createEmptyToken(LPR\_T); }

if (c == ')'){ return createEmptyToken(RPR\_T); }

if (c == '{'){ return createEmptyToken(LBR\_T); }

if (c == '}'){ return createEmptyToken(RBR\_T); }

/\*Arithmetic Operators\*/

if (c == '+') { return createArrOpToken(PLUS); }

if (c == '-') { return createArrOpToken(MINUS); }

if (c == '/') { return createArrOpToken(DIV); }

if (c == '\*') { return createArrOpToken(MULT); }

/\*Assorted characters. 50% off\*/

if (c == ',') { return createEmptyToken(COM\_T); }

if (c == ';') { return createEmptyToken(EOS\_T); }

/\*All other characters are not allowed\*/

return createErrorToken(1,(char \*) &c);

}

}

/\*

\* Purpose: Gets the next state in the transition table

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: state - the current state of the DFA. Must be an existing state.

c - The symbol to be analyzed

accept - points to the current type of state we are in

\* Return value: The next state in the DFA

\*/

int get\_next\_state(int state, char c, int \*accept)

{

int col;

int next;

col = char\_class(c);

next = st\_table[state][col];

#ifdef DEBUG

printf("Input symbol: %c Row: %d Column: %d Next: %d \n",c,state,col,next);

#endif

assert(next != IS);

#ifdef DEBUG

if(next == IS){

printf("Scanner Error: Illegal state:\n");

printf("Input symbol: %c Row: %d Column: %d\n",c,state,col);

exit(1);

}

#endif

\*accept = as\_table[next];

return next;

}

/\*

\* Purpose: Determine what column of the transition table a symbol represents

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: c - The symbol to be checked

\* Return value: A column of the transition table

\*/

int char\_class (char c)

{

int val;

if (isalpha(c)){ val = 0; } /\* COLUMN 0: ascii letter \*/

else if (c == '0') { val = 1; } /\* COLUMN 1: zero \*/

else if (c == '8' || c == '9'){ val = 2; } /\* COLUMN 2: 8 or 9 \*/

else if (isdigit(c)) { val = 3; } /\* COLUMN 3: 1-7 \*/

else if (c == '.') { val = 4; } /\* COLUMN 4: . \*/

else if (c == '#') { val = 5; } /\* COLUMN 5: # \*/

else { val = 6; } /\* COLUMN 6: other \*/

return val;

}

/\*

\* Purpose: Accepting state function for variables and keywords

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme produced by the finite state machine

\* Return value: A token representing the variable or keyword

\*/

Token aa\_func02(char \*lexeme){

int i;

/\*Lexeme is a keyword\*/

if ( (i = iskeyword(lexeme)) != -1){

return createIntValToken(KW\_T, i);

}

/\*Lexeme is an SVID\*/

if (lexeme[strlen(lexeme)-1] == '#'){

/\*Make sure we keep the '#' if the variable is longer than VID\_LEN\*/

lexeme[VID\_LEN-1] = '#';

return createStringToken(SVID\_T, lexeme);

}

/\*Lexeme is an AVID\*/

return createStringToken(AVID\_T, lexeme);

}

/\*

\* Purpose: Accepting state function for Integer Literals

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme produced by the finite state machine

\* Return value: A token representing the Integer Literal

\*/

Token aa\_func07(char \*lexeme){

long decimal = atol(lexeme);

/\*In case of overflow or underflow. \*/

if (decimal < 0 || decimal > SHRT\_MAX){

return createErrorToken(strlen(lexeme), lexeme);

}

return createIntValToken(INL\_T, (int)decimal);

}

/\*

\* Purpose: Accepting state function for Floating Point Literals

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme produced by the finite state machine

\* Return value: A token representing the floating point literal

\*/

Token aa\_func08(char \*lexeme){

double flt = atof(lexeme);

/\*Check for float overflow. 4 bytes is all we allow \*/

if (flt < FLT\_MIN && flt > 0 || flt > FLT\_MAX){

return createErrorToken(strlen(lexeme), lexeme);

}

return createFltValToken((float)flt);

}

/\*

\* Purpose: Accepting state function for Octal Literals

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme produced by the finite state machine

\* Return value: A token representing the octal literal

\*/

Token aa\_func09(char \*lexeme){

long octal = atool(lexeme); /\* Just using "a tool" of mine ;) \*/

if ( octal < 0 || octal > SHRT\_MAX){

return createErrorToken(strlen(lexeme), lexeme);

}

return createIntValToken(INL\_T, (int) octal);

}

/\*

\* Purpose: Accepting state function for Errors

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme produced by the finite state machine

\* Return value: An error Token

\*/

Token aa\_func11(char \*lexeme){

return createErrorToken(strlen(lexeme), lexeme);

}

/\*

\* Purpose: Converts an octal number in the form of a string to its integer value

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: lexeme - The lexeme to be converted

\* Return value: The integer value of the octal string

\*/

long atool(char \* lexeme){

/\*The second argument is NULL because we don't need an end pointer\*/

return strtol(lexeme, NULL, 8);

}

/\*

\* Purpose: Determines whether a string is in the keyword table

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: kw\_lexeme - The string to be checked

\* Return value: If the lexeme is a keyword, this returns the position in the array

of the keyword, otherwise -1 for error

\*/

int iskeyword(char \* kw\_lexeme){

int i;

/\* Check the keyword table to see if it contains lexeme \*/

for (i = 0; i < KWT\_SIZE; i++){

if (!strcmp(kw\_lexeme, kw\_table[i])){

return i;

}

}

return -1;

}

/\*

\* Purpose: An easy way of creating an error token from a subset of a long string

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: msg\_len - the number of characters to copy into the error token

err\_msg - the string that contains the error message. Must be longer

than the msg\_len.

\* Return value: An error token with the specified message

\*/

Token createErrorToken(unsigned int msg\_len, const char \*err\_msg){

Token t;

unsigned int i;

/\* Make sure we don't send in a stupid msg\_len\*/

assert(strlen(err\_msg) >= msg\_len);

t.code = ERR\_T;

for (i = 0 ; i < msg\_len && i < ERR\_LEN; i++){

t.attribute.err\_lex[i] = err\_msg[i];

}

t.attribute.err\_lex[i] = '\0';

return t;

}

/\*

\* Purpose: To ease the creation of tokens with integer attributes

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: code - The code to represent the type of token

attribute - An integer attribute for the code

\* Return value: A token with your specified values

\*/

Token createIntValToken(int code, int attribute){

Token t;

t.code = code;

t.attribute.int\_value = attribute;

return t;

}

/\*

\* Purpose: To ease the creation of tokens with rel\_op attributes

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: attribute - A rel\_op attribute for the token

\* Return value: A token with your specified values

\*/

Token createRelOpToken(Rel\_Op attribute){

Token t;

t.code = REL\_OP\_T;

t.attribute.rel\_op = attribute;

return t;

}

/\*

\* Purpose: To ease the creation of empty tokens

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: code - The code to represent the type of token

\* Return value: A token with your specified values

\*/

Token createEmptyToken(int code){

Token t;

t.code = code;

return t;

}

/\*

\* Purpose: To ease the creation of arr\_op tokens

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: attribute - An arr\_op attribute for the token

\* Return value: A token with your specified values

\*/

Token createArrOpToken(Arr\_Op attribute){

Token t;

t.code = ART\_OP\_T;

t.attribute.arr\_op = attribute;

return t;

}

/\*

\* Purpose: To ease the creation of log\_op tokens

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: attribute - A log\_op attribute for the token

\* Return value: A token with your specified values

\*/

Token createLogOpToken(Log\_Op attribute){

Token t;

t.code = LOG\_OP\_T;

t.attribute.log\_op = attribute;

return t;

}

/\*

\* Purpose: To ease the creation of tokens with string attributes

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: code - the code for the token

attribute - the string attribute for the token

\* Return value: A token with your specified values

\*/

Token createStringToken(int code, char \*attribute){

Token t;

int i;

t.code = code;

for (i = 0; i < VID\_LEN && attribute[i] != '\0'; i++){

t.attribute.vid\_lex[i] = attribute[i];

}

t.attribute.vid\_lex[i] = '\0';

return t;

}

/\*

\* Purpose: To ease the creation of tokens with float attributes

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: attribute - A float attribute for the token

\* Return value: A token with your specified values

\*/

Token createFltValToken(float attribute){

Token t;

t.code = FPL\_T;

t.attribute.flt\_value = attribute;

return t;

}

/\*

\* Purpose: Tests a symbol to see if it is end of file.

\* Author: William Collins (040652633)

\* History/Versions: 1.0

\* Parameters: c - the symbol to be tested. EOF will be either 0 or 255

\* Return value: true (1) if the symbol is EOF, false(0) otherwise

\*/

static int isEOF(unsigned char c){

if (c == 0 || c == 255){

return 1;

}

return 0;

}

**Table.h**

/\* File Name: table.h

\* Version: 1.12

\* Author: William Collins (040652633) && Svillen Ranev

\* Course: CST8152 - Compilers

\* Assignment: 2

\* Date: June 30, 2011

\* Professor: Svillen Ranev

\* Purpose: Transition table and functions used by the scanner

\*/

#ifndef TABLE\_H\_

#define TABLE\_H\_

#ifndef BUFFER\_H\_

#include "buffer.h"

#endif

#ifndef NULL

#include <\_null.h> /\* NULL pointer constant is defined there \*/

#endif

#define ES 11 /\* Error state \*/

#define IS -1 /\* Inavalid state \*/

/\* Accepting state function prototypes \*/

Token aa\_func02(char \*lexeme);

Token aa\_func07(char \*lexeme);

Token aa\_func08(char \*lexeme);

Token aa\_func09(char \*lexeme);

Token aa\_func11(char \*lexeme);

/\* State transition table definition \*/

#define TABLE\_COLUMNS 7

/\*transition table - type of states defined in separate table \*/

int st\_table[ ][TABLE\_COLUMNS] = {

/\* State 0 \*/ {1, 4, 3, 3, IS, IS, IS},

/\* State 1 \*/ {1, 1, 1, 1, 2, 10, 2},

/\* State 2 \*/ {IS, IS, IS, IS, IS, IS, IS},

/\* State 3 \*/ {ES, 3, 3, 3, 6, 7, 7},

/\* State 4 \*/ {ES, ES, ES, 5, 6, 7, 7},

/\* State 5 \*/ {ES, 5, ES, 5, ES, ES, 9},

/\* State 6 \*/ {8, 6, 6, 6, 8, 8, 8},

/\* State 7 \*/ {IS, IS, IS, IS, IS, IS, IS},

/\* State 8 \*/ {IS, IS, IS, IS, IS, IS, IS},

/\* State 9 \*/ {IS, IS, IS, IS, IS, IS, IS},

/\* State 10 \*/ {ES, ES, ES, ES, ES, ES, 2},

/\* State 11 \*/ {IS, IS, IS, IS, IS, IS, IS}

};

/\* Accepting state table definition \*/

#define ASWR 2 /\* accepting state with retract \*/

#define ASNR 1 /\* accepting state with no retract \*/

#define NOAS 0 /\* not accepting state \*/

int as\_table[ ] = {

NOAS, /\*0\*/

NOAS, /\*1\*/

ASWR, /\*2\*/

NOAS, /\*3\*/

NOAS, /\*4\*/

NOAS, /\*5\*/

NOAS, /\*6\*/

ASWR, /\*7\*/

ASWR, /\*8\*/

ASWR, /\*9\*/

NOAS, /\*10\*/

ASNR /\*11\*/

};

/\* defining a new type: pointer to function (of one char \* argument)

returning Token\*/

typedef Token (\*PTR\_AAF)(char \*lexeme);

/\* Callback table used to handle accepting states \*/

PTR\_AAF aa\_table[ ] ={

NULL, /\*0\*/

NULL, /\*1\*/

aa\_func02, /\*2\*/

NULL, /\*3\*/

NULL, /\*4\*/

NULL, /\*5\*/

NULL, /\*6\*/

aa\_func07, /\*7\*/

aa\_func08, /\*8\*/

aa\_func09, /\*9\*/

NULL, /\*10\*/

aa\_func11 /\*11\*/

};

/\* A list of our keywords \*/

#define KWT\_SIZE 8 /\* The size of the keyword table \*/

char \* kw\_table []= {

"ELSE",

"IF",

"INPUT",

"OUTPUT",

"PLATYPUS",

"REPEAT",

"THEN",

"USING"

};

#endif

**Test Files**

Just like in my buffer assignment, I created a batch script for testing. Once my batch file decided that my code worked, the batch file was run after every change I made. Here is the batch file I used.

ass2 a2empty.pls > a2empty.out

ass2 ass2r.pls > ass2r.out

ass2 ass2w.pls > ass2w.out

ass2 a2error.pls > a2error.out

diff a2empty.out outputs/a2empty.out

diff ass2r.out outputs/ass2r.out

diff ass2w.out outputs/ass2w.out

diff a2error.out outputs/a2error.out

pause

**a2empty.out**

Reading file a2empty.pls ....Please wait

Printing buffer parameters:

The capacity of the buffer is: 1

The current size of the buffer is: 0

Printing buffer contents:

Scanning source file...

Token Attribute

----------------------------------

SEOF\_T

**ass2r.out**

Reading file ass2r.pls ....Please wait

Printing buffer parameters:

The capacity of the buffer is: 577

The current size of the buffer is: 576

Printing buffer contents:

!< This program calculates the sum of 32767 even numbers.

!< The program is "lexically" and "syntactically" correct

!< and should not produce any error

PLATYPUS {

a=+0.0;

sum008 = 7.87050 ;

INPUT(a,sum008);

USING(i = 0,i < 32767 .OR. i == 077777,i = i + 02 )REPEAT{

a=

a\*i/0.5

;

sum008 = sum008 + a - 1 ;

};

IF(text# == "")THEN

text# = "prog" << "ram";

ELSE {

text# = text# << "ram";

};

OUTPUT("\\* This is a platypus -:)-<-<-- \\*");

OUTPUT(text#);

IF(text# == "program".OR.sum008!=8..AND.i>012)THEN

OUTPUT(sum008);

OUTPUT();

ELSE{};

}

Scanning source file...

Token Attribute

----------------------------------

KW\_T PLATYPUS

LBR\_T

AVID\_T a

ASS\_OP\_T

ART\_OP\_T 0

FPL\_T 0.000000

EOS\_T

AVID\_T sum008

ASS\_OP\_T

FPL\_T 7.870500

EOS\_T

KW\_T INPUT

LPR\_T

AVID\_T a

COM\_T

AVID\_T sum008

RPR\_T

EOS\_T

KW\_T USING

LPR\_T

AVID\_T i

ASS\_OP\_T

IL\_T 0

COM\_T

AVID\_T i

REL\_OP\_T 3

IL\_T 32767

LOG\_OP\_T 1

AVID\_T i

REL\_OP\_T 0

IL\_T 32767

COM\_T

AVID\_T i

ASS\_OP\_T

AVID\_T i

ART\_OP\_T 0

IL\_T 2

RPR\_T

KW\_T REPEAT

LBR\_T

AVID\_T a

ASS\_OP\_T

AVID\_T a

ART\_OP\_T 2

AVID\_T i

ART\_OP\_T 3

FPL\_T 0.500000

EOS\_T

AVID\_T sum008

ASS\_OP\_T

AVID\_T sum008

ART\_OP\_T 0

AVID\_T a

ART\_OP\_T 1

IL\_T 1

EOS\_T

RBR\_T

EOS\_T

KW\_T IF

LPR\_T

SVID\_T text#

REL\_OP\_T 0

STR\_T 0

RPR\_T

KW\_T THEN

SVID\_T text#

ASS\_OP\_T

STR\_T 1 prog

SCC\_OP\_T

STR\_T 6 ram

EOS\_T

KW\_T ELSE

LBR\_T

SVID\_T text#

ASS\_OP\_T

SVID\_T text#

SCC\_OP\_T

STR\_T 10 ram

EOS\_T

RBR\_T

EOS\_T

KW\_T OUTPUT

LPR\_T

STR\_T 14 \\* This is a platypus -:)-<-<-- \\*

RPR\_T

EOS\_T

KW\_T OUTPUT

LPR\_T

SVID\_T text#

RPR\_T

EOS\_T

KW\_T IF

LPR\_T

SVID\_T text#

REL\_OP\_T 0

STR\_T 49 program

LOG\_OP\_T 1

AVID\_T sum008

REL\_OP\_T 1

FPL\_T 8.000000

LOG\_OP\_T 0

AVID\_T i

REL\_OP\_T 2

IL\_T 10

RPR\_T

KW\_T THEN

KW\_T OUTPUT

LPR\_T

AVID\_T sum008

RPR\_T

EOS\_T

KW\_T OUTPUT

LPR\_T

RPR\_T

EOS\_T

KW\_T ELSE

LBR\_T

RBR\_T

EOS\_T

RBR\_T

SEOF\_T

**ass2w.out**

Reading file ass2w.pls ....Please wait

Printing buffer parameters:

The capacity of the buffer is: 949

The current size of the buffer is: 948

Printing buffer contents:

!<This program contains many lexical errors

!<It tests mainly your transition table

!< Use MY output to adjust

!< YOUR error state transitions in YOUR transition table

!<You should create your own test file with more errors

!>Wrong comment

PLATYPUs {

m=00;!< illegal octal literal (illegal zero literal)

i=000; !< illegal octal literal

j=087; !< illegal octal digit

k=2I; !< possible error

k=0O; !< digit 0 followed by a letter O

a% = 0.0L; !< possible error

b. = 0.L !< possible error

FOR> = .0 !< illegal floating point litetral

Read(ElSe)

If=0.l !< letter l not digit 1

o=018a !< possible error

b10 = 1O1.0; !< letter O follows 1

3C=7.0O2; !< digit in VID and letter O precedes 2

name#S = Last# << " S.#" !< wrong string VID

USING(b10 > 00 OR. <> b10.AND a !- 0.0)DO{

a=a+1..;

c=01.0; !< leading zero not allowed in floating point

}

WRITE#("No Luck Today);

WRITE (""My Scanner has gone astray");

}

"There is always one more bug

Scanning source file...

Token Attribute

----------------------------------

ERR\_T !>

AVID\_T PLATYPUs

LBR\_T

AVID\_T m

ASS\_OP\_T

ERR\_T 00

EOS\_T

AVID\_T i

ASS\_OP\_T

ERR\_T 00

IL\_T 0

EOS\_T

AVID\_T j

ASS\_OP\_T

ERR\_T 08

IL\_T 7

EOS\_T

AVID\_T k

ASS\_OP\_T

ERR\_T 2I

EOS\_T

AVID\_T k

ASS\_OP\_T

ERR\_T 0O

EOS\_T

AVID\_T a

ERR\_T %

ASS\_OP\_T

FPL\_T 0.000000

AVID\_T L

EOS\_T

AVID\_T b

ERR\_T .

ASS\_OP\_T

FPL\_T 0.000000

AVID\_T L

AVID\_T FOR

REL\_OP\_T 2

ASS\_OP\_T

ERR\_T .

IL\_T 0

AVID\_T Read

LPR\_T

AVID\_T ElSe

RPR\_T

AVID\_T If

ASS\_OP\_T

FPL\_T 0.000000

AVID\_T l

AVID\_T o

ASS\_OP\_T

ERR\_T 018

AVID\_T a

AVID\_T b10

ASS\_OP\_T

ERR\_T 1O

FPL\_T 1.000000

EOS\_T

ERR\_T 3C

ASS\_OP\_T

FPL\_T 7.000000

AVID\_T O2

EOS\_T

ERR\_T name#S

ASS\_OP\_T

SVID\_T Last#

SCC\_OP\_T

STR\_T 0 S.#

KW\_T USING

LPR\_T

AVID\_T b10

REL\_OP\_T 2

ERR\_T 00

AVID\_T OR

ERR\_T .

REL\_OP\_T 3

REL\_OP\_T 2

AVID\_T b10

ERR\_T .

AVID\_T AND

AVID\_T a

ERR\_T !-

AVID\_T a

ASS\_OP\_T

AVID\_T a

ART\_OP\_T 0

FPL\_T 1.000000

ERR\_T .

EOS\_T

AVID\_T c

ASS\_OP\_T

ERR\_T 01.

IL\_T 0

EOS\_T

RBR\_T

SVID\_T WRITE#

LPR\_T

STR\_T 5 No Luck Today);

WRITE (

STR\_T 31 My Scanner has gone astray

RPR\_T

EOS\_T

RBR\_T

ERR\_T "There is always ...

SEOF\_T

S.#

**a2error.out**

Reading file a2error.pls ....Please wait

Printing buffer parameters:

The capacity of the buffer is: 1155

The current size of the buffer is: 1154

Printing buffer contents:

!<This file contains many lexical errors.

!<It tests mainly the correctness of the accepting functions

!<5 tabs

a1234567

!<legal, but too long (possible semantic error)

a1234567MORE

b1234567MORE#

!<lexical error: # out of context, allowed only in comments and strings

#OR$

!<lexical error: | allowed only in strings and comments

|

!<another lexical error: # out of context

Not#here

!<another lexical error: . out of context

.

!<another lexical error: ~ illegal symbol

~

!<legal FPL (will be rounded to 2.0 when displayed)

1.999999999999999911111111111111111111111111111111111111111111111111111111111111111

!< big float number (error: out of range)

999999999999999999999999999999999999999999999999999999999999999999999999999999999.0

!<small float number (error: out of range)

0.00000000000000000000000000000000000000000000000000000000000000000000000000000000001

!< big decimal int number (error: out of range)

99999

!< big octal int number (error: out of range)

0177777

!< gigantic decimal int number (error: out of range)

123456789012345678901234567890

!< gigantic octal int number (error: out of range)

0123456701234567012345670

Scanning source file...

Token Attribute

----------------------------------

AVID\_T a1234567

AVID\_T a1234567

SVID\_T b123456#

ERR\_T #

AVID\_T OR

ERR\_T $

ERR\_T |

ERR\_T Not#h

AVID\_T ere

ERR\_T .

ERR\_T ~

FPL\_T 2.000000

ERR\_T 99999999999999999999

ERR\_T 0.000000000000000000

ERR\_T 99999

ERR\_T 0177777

ERR\_T 12345678901234567890

ERR\_T 01234567012345670123

SEOF\_T