

TEC

Compiladores e Intérpretes  
Proyecto #2: Analizador Léxico  
Profesor: Francisco Torres

Dennisse Rojas Casanova  
Treicy Sánchez Gutiérrez

25 de Mayo, 2016

El Análisis Léxico consiste en descomponer un fuente de entrada en categorías léxicas mínimas llamadas tokens. Un programa en Flex consiste básicamente en una lista de expresiones regulares que definen acciones a ejecutar cuando ocurre un match.

```

token next_token( void ) {
    current_token;
}
void match( token t ) {
    if ( nextev_token == t ) {
        nextev_token = scanner( );
        if ( nextev_token == -1 ) nextev_token = scanner( );
        current_token = nextev_token;
    }
}

```

```
}  
else {  
    syntax_error( t ) ;  
}  
}  
void syntax_error( token t) {
```

```
printf ( Sintax error, expecting %s\ n, get_token_name( t ) );  
exit ( 0 ) ;  
}  
void system _ goal( void ) {  
  
    program( ) ;
```

```
    match( SCANEOF) ;  
    finish( ) ;  
}  
void  program( void ) {  
  
    start( ) ;  match( BEGIN) ;  
    statement_list( ) ;  
    match( END) ;  
}
```

```
}  
void statement_list( void ) {  
  
    statement( ) ;  
    while ( true ) {  
        switch ( next_token( ) ) {  
            case ID:  
            case READ:
```

```
    case WRITE:
        statement( ) ;
        break ;
    default :
        ;
}
}
}
void statement( void ) {
```



```

token tok= next_token( ) ;
expr_rec result;    expr_rec p_expr;  switch ( tok) {
    case ID:

        strcpy ( previous_tokenbuffer, token_buffer) ;          match(
ID) ;
        result = process_id( ) ;
        match( ASSIGNOP) ;
        expression( ^ p_expr) ;          assign( result, p_expr) ;
        match( SEMICOLON) ;
        break ;

```

case READ:

```
match( READ) ; match( LPAREN) ;  
id_list( ) ; match( RPAREN) ;  
match( SEMICOLON) ;  
    break ;
```

case WRITE:

```
match( WRITE) ; match( LPAREN) ;  
expr_list( ) ; match( RPAREN) ;
```

```
    match( SEMICOLON ) ;  
        break ;  
default :  
    syntax_error( tok ) ;  
    break ;  
}  
}
```

```
void assign ( expr_rec result, expr_rec p_expr ) {  
    char * salida;  
    salida = calloc ( 1024 , sizeof ( char ) ) ;  
    strcpy ( salida, extractEXPR( p_expr ) ) ;  
    generate ( Store, salida, , ) ;  
}
```

```

void id_list( void ) {

    expr_rec result;
    strcpy ( previous_tokenbuffer, token_buffer );    match( ID );
    strcpy ( , previous_tokenbuffer );
    while ( next_token( ) == COMMA ) {
        strcpy ( previous_tokenbuffer, token_buffer );    match(
COMMA );
        strcpy ( previous_tokenbuffer, token_buffer );    match( ID )
;
        strcpy ( , previous_tokenbuffer );
        read_id( result );
    }
}

```

```

    }
}
void read_id ( expr_rec in_var) {

    generate ( Read, , Integer, );
}
void write_expr ( expr_rec out_expr) {

```

```

    generate ( Write, , Integer, ) ;
}
void expr_list( void ) {

    expr_rec result;
    expression( ^ result ) ;
    write_expr( result ) ;
    while ( next_token( ) == COMMA ) {
        match( COMMA ) ;
    }
}

```

```

    expression( ^ result) ;
    write_ expr( result) ;
}
}
void expression ( expr_rec * result) {
    expr_rec left_ operand, right_ operand;
    op_rec op;
    primary ( ^ left_ operand) ;
    while ( next_token( ) == PLUSOP || next_token( )
== MINUSOP) {

```



```

    add_op ( ^ op) ;
    primary ( ^ right_operand) ;
    left_operand = gen_infix( left_operand, op, right_
operand) ;
}
* result = left_operand;
}
void add_op( op_rec* p_operand) {

```

```
token tok = next_token( ) ;
```

```
if ( tok == PLUSOP || tok== MINUSOP) {  
    previous_currentToken = current_token;    match( tok) ;  
    * p_operand = process_op( ) ;  
} else {  
    sintax_error( tok) ;  
}  
}
```

```

void primary( expr_rec* p_operand) {
    expr_rec result;
    token tok= next_token( );
    switch ( tok) {
        case LPAREN:

            match( LPAREN) ;
            expression( ^ result) ;
            * p_operand = result;
            match( RPAREN) ;

```

```

        break ;
    case ID:

        strcpy ( previous_tokenbuffer, token_buffer) ;           match(
ID) ;
        * p_operand = process_id( ) ;
        break ;
    case INTLITERAL:

        strcpy ( previous_tokenbuffer, token_buffer) ;           match(
INTLITERAL) ;
        * p_operand = process_literal( ) ;

```

```
        break ;
    default :
        sintax_error( tok ) ;
        break ;
    }
}
const char * get_token_name( token t) {
```

```
switch ( t ) {  
  case BEGIN:  
    BEGIN;  
  case END:  
    END;  
  case READ:  
    READ;  
  case WRITE:  
    WRITE;  
  case ID:
```

```
ID;  
case INTLITERAL:  
    INTLITERAL;  
case LPAREN:  
    LPAREN;  
case RPAREN:  
    RPAREN;  
case SEMICOLON:  
    SEMICOLON;  
case COMMA:
```

```
        COMMA;  
    case ASSIGNOP:  
        ASSIGNOP;  
    case PLUSOP:  
        PLUSOP;  
    case MINUSOP:  
        MINUSOP;  
    case SCANEOF:  
        SCANEOF;  
}
```



```
}  
void open_outputFile( ) {  
    output_file = fopen ( , w+ ) ;  
}  
void close_outputFile( ) {  
    fclose ( output_file ) ;
```

```
}  
void start( void ) {  
  
}  
void finish( void ) {  
    generate( Halt, , , ) ;  
}
```

```
void ident( ) {  
    }  
expr_rec process_id( void ) {  
    expr_rec t;  
  
    check_id( previous_tokenbuffer ) ;  
    = IDEXPR;  
    strcpy ( , previous_tokenbuffer ) ;  
    t;
```

```

}
expr_rec process_literal( void ) {
    expr_rec t;

    = LITERALEXPR;
    ( void ) sscanf ( previous_tokenbuffer, % d, ^ );
    t;

```

```
}  
op_rec process_op ( void ) {  
    op_rec o;  
    if ( previous_currentToken == PLUSOP)  
        = PLUS;  
    else  
        = MINUS;
```

```

    o;
}
void check_id( string s) {
    if ( ! lookup( s) ) {
        enter( s) ;
        generate( Declare, s, Integer, );
    }
}

```

```

bool lookup( string s) {
    int i;
    for ( i = 0 ; i < symTable_count; i++ ) {
        if ( strcmp ( symbol_table[ i] , s) == 0 ) {
            true ;
        }
        i++ ;
    }
    false ;
}

```

```

}
void enter( string s) {
    strcpy ( symbol_table[ symTable_count] , s) ;
    symTable_count++;
}
void generate( string op1, string op2, string op3, string op4) {
    if ( ( op1 == Declare) || ( op1 == Store) ) {
        fprintf ( output_file, % s % s % s % s\ n , op1, op2, op3,
op4) ;
    }
}

```



```

    } else if ( op1 == Halt) {
        fprintf ( output_ file, % s % s % s % s\ n , op1, op2, op3,
op4) ;
    }
    else {
        fprintf ( output_ file, % s % s % s % s\ n , op1, op2, op3,
op4) ;
    }
}
char * extractOP( op_ rec p_ operand) {

```

```

    char * minus = Sub;
char * plus = Add;
if ( == MINUS) {
    minus;
} else if ( == PLUS) {
    plus;
}
}
char * extractEXPR( expr_rec p_expr) {

```

```

char * express_buffer;
express_buffer = calloc ( 1024 , sizeof ( char ) );
if ( == IDEXPR) {
    strcpy ( express_buffer, );
    express_buffer;
} else if ( == LITERALEXPRESS) {
    sprintf ( express_buffer, % d, );
    express_buffer;
} else {
    strcpy ( express_buffer, );

```

```

        express_buffer;
    }
}
expr_rec gen_infix ( expr_rec e1, op_rec op, expr_rec e2)
{
    expr_rec e_rec, e_rec1;

    = LITERALEXPR;

```

```
char * resultadoEXPR2;  
resultadoEXPR2 = calloc ( 1024 , sizeof ( char ) ) ;  
char * resultadoEXPR1;  
resultadoEXPR2 = calloc ( 1024 , sizeof ( char ) ) ;  
int resultado;  
  
if ( == LITERALEXPR && == LITERALEXPR) {  
    int numero1 = ;  
    int numero2 = ;  
}
```

```

if ( == PLUS) {
    resultado = numero1 + numero2;
} else {
    resultado = numero1 - numero2;
}
= resultado;
e_rec;
} else {
    expr_rec e_rec1;
    = TEMPEXPR;

```

```

    resultadoEXPR1 = extractEXPR( e1 ) ;
    resultadoEXPR2 = extractEXPR( e2 ) ;
    strncpy( , get_temp( ) , MAXIDLEN ) ;
    generate ( extractOP( op ) , resultadoEXPR1, resultadoEXPR2,
) ;
    e_rec1;
}
}
char * get_temp ( void ) {

```

```
static char tempname[ MAXIDLEN] ;  
max_ temp++;  
sprintf( tempname, Temp % d,  max_ temp) ;  
check_ id ( tempname) ;  
    tempname;  
}  
void open_ file( ) {
```



```

char filename [ 100 ] = ;
printf ( Enter a value : );
scanf( % s, filename);
file = fopen ( filename, r );
}
void read_file( ) {
    open_file( );
    if ( file) {
        len = ftell( file);
    }
}

```

```
    }  
    else {  
        printf ( Problema al abrir el archivo\ n) ;  
        exit ( 0 ) ;  
    }  
}  
void close_ file( ) {
```

```

    fclose ( file ) ;
}
void  buffer_char ( char  c ) {
    token_buffer[ charPos++ ] = c;
}
void  clear_token_buffer( ) {
    memset ( token_buffer,  0 ,  5 ) ;
    charPos =  0 ;
}

```

```

}
token check_reserved( ) {
    int letter, c;
    bool reserved;
    for ( letter= 0 ; letter < len_tb; letter++ ) {
        if ( ' B' == toupper ( token_buffer[ letter] ) ) {
            reserved = true ;
            for ( c = 0 ; c < 5 ; c++ ) {
                if ( begin_buffer[ c] != toupper ( token_buffer[
letter++ ] ) ) {

```

```
        reserved = false ;  
        break ;  
    }  
}  
if ( reserved == true ) {  
    BEGIN;  
} else {  
    ID;  
}
```

```

    break ;
} else if ( ' E' == toupper ( token_ buffer[ letter] ) ) {
    reserved = true ;
    for ( c = 0 ; c < 3 ; c++ ) {
        if ( end_ buffer[ c] != toupper ( token_ buffer[ letter++
] ) ) {
            reserved = false ;
            break ;
        }
    }
    if ( reserved == true ) {

```

```

        END;
    } else {
        ID;
    }
    break ;
} else if ( ' W' == toupper ( token_ buffer[ letter] ) ) {
    reserved = true ;
    for ( c = 0 ; c < 5 ; c++ ) {
        if ( write_ buffer[ c] != toupper ( token_ buffer[
letter++ ] ) ) {
            reserved = false ;

```

```

        break ;
    }
}
if ( reserved == true ) {
    WRITE;
} else {
    ID;
}
break ;
} else if ( ' R' == toupper ( token_buffer[ letter] ) ) {

```



```

reserved = true ;
for ( c = 0 ; c < 4 ; c++ ) {
    if ( read_buffer[ c ] != toupper ( token_buffer[
letter++ ] ) ) {
        reserved = false ;
        break ;
    }
}
if ( reserved == true ) {
    READ;
} else {

```

```

        ID;
    }
    break ;
} else {
    ID;
}
}
}
}
void lexical_error( int character) {

```

```

    printf ( LEXICAL ERROR % d\ n,  character) ;
}
void  print_token_buffer( ) {
    int  i;
    printf ( IMPRIMIENDO TOKEN BUFFER\ n) ;
    for  ( i = 0 ;  i <= len_token_buffer;  i++ ) {
        printf ( % d\ n,  token_buffer[ i]  ) ;
    }
}

```

```
void get_tokens( ) {  
    read_file( ) ;  
    token ejemplo;  
    while ( filePos != len) {  
        ejemplo = scanner( ) ;  
        printf ( token %d\ n, ejemplo ) ;  
    }  
}
```



