Instituto Tecnológico de Costa Rica

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

Dennisse Rojas Casanova
Treicy Sánchez Gutiérrez

25 de Mayo, 2016

Análisis Léxico y Flex

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 {
        sintax_ error( t) ;
    }
}
void    sintax_ 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 :
      sintax_ 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, in_ , Integer, ) ;
}
void  write_expr ( expr_rec out_expr) {
```

```
  generate ( Write, out_ , 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   ( output_ , 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   ( p_   ==   MINUS) {
       minus;
  }  else   if   ( p_   ==   PLUS) {
       plus;
  }
}
char *   extractEXPR( expr_ rec   p_ expr) {
```

```
char *   express_ buffer;
express_ buffer  =   calloc ( 1024 ,   sizeof ( char ) ) ;
if   ( p_   ==   IDEXPR) {
  strcpy ( express_ buffer, p_ ) ;
     express_ buffer;
}  else   if   ( p_   ==   LITERALEXPR) {
  sprintf  ( express_ buffer,  % d, p_ ) ;
     express_ buffer;
}  else   {
  strcpy ( express_ buffer, p_ ) ;
```

```
        express_ buffer;
    }
}
expr_ rec  gen_ infix  ( expr_ rec  e1,   op_ rec  op,   expr_ rec  e2)
{

    expr_ rec  e_ rec,   e_ rec1;

    e_   =   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;
  }
  e_ = resultado;
     e_ rec;
} else {
  expr_ rec  e_ rec1;
  e_ = TEMPEXPR;
```

```
    resultadoEXPR1 = extractEXPR( e1) ;
    resultadoEXPR2 = extractEXPR( e2) ;
    strncpy( e_ , get_ temp( ) , MAXIDLEN) ;
    generate ( extractOP( op) , resultadoEXPR1, resultadoEXPR2,
e_ ) ;
        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] ) ) {
```

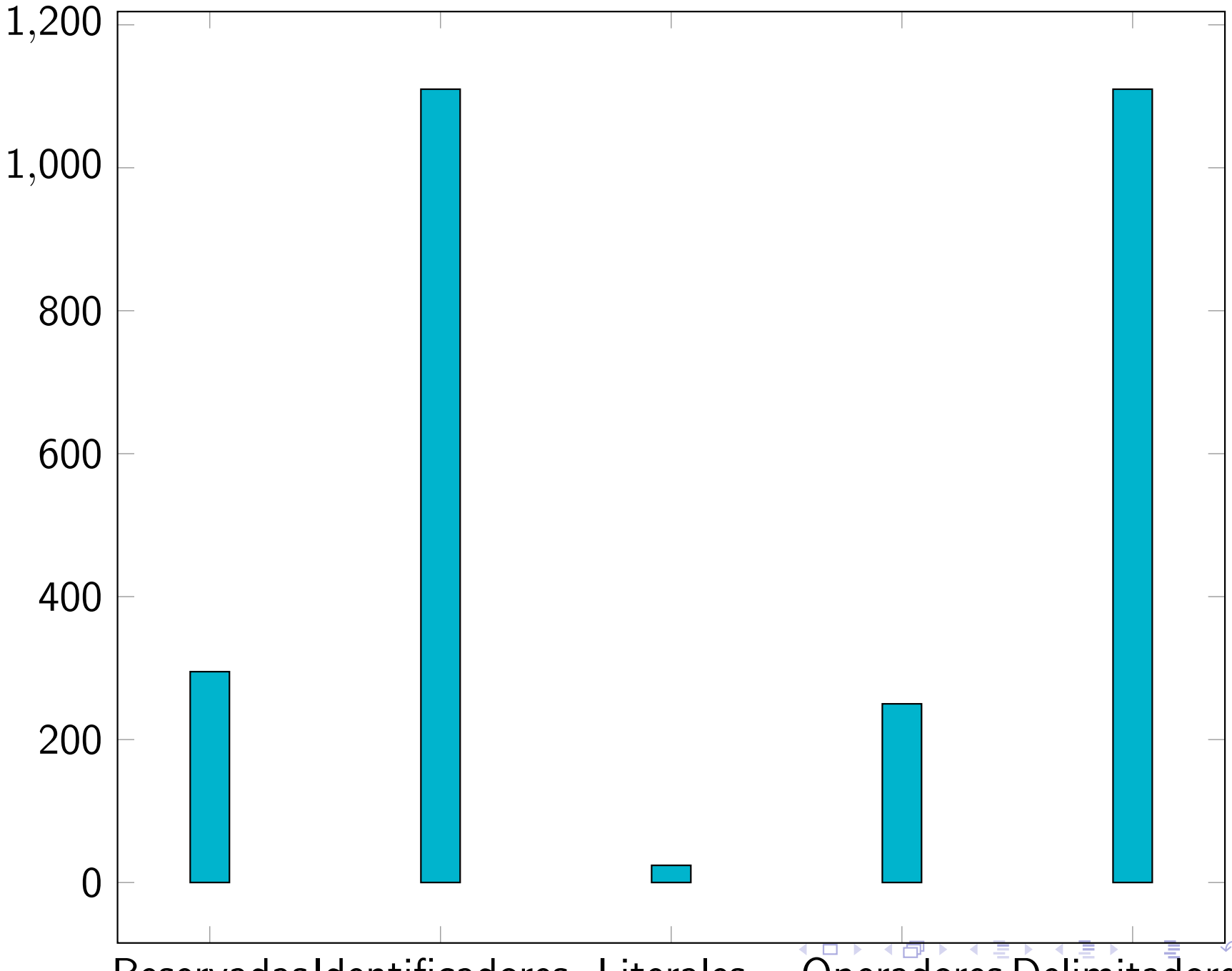
```
      reserved = true;
      for (c = 0; c < 4; c++) {
        if (read_buffer[c] != toupper(token_buffer[
letter++])) {
          reserved = false;
          break;
        }
      }
      if (reserved == true) {
          READ;
      } else {
```

```
            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  ) ;
  }
}
```

Histograma


1,200
1,000
800
600
400
200
0
Reservadas
Identificadores
Literales
Operadores
Delimitadores