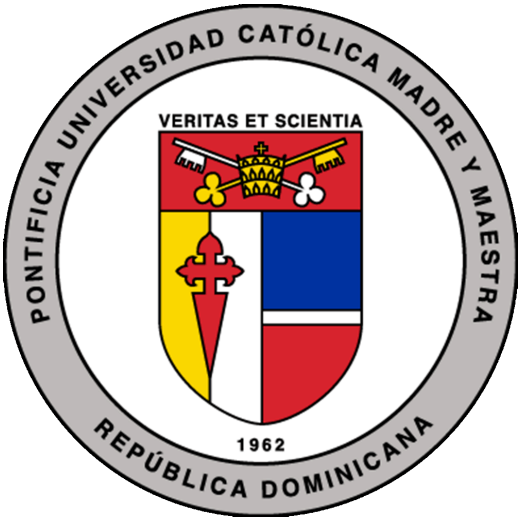
Pontificia Universidad Católica Madre y Maestra



CSTI-1880-6812 - Procesamiento de Lenguaje Compilador

Tema:

Procesamiento Lenguaje y Compilación

Integrantes:

1014-3611 Starlin Frías Luna

Profesor:

Juan Felipe

Santiago de los Caballeros, República Dominicana

Domingo 15 de Febrero del 2025

1. Introducción

Este informe detalla el desarrollo de un analizador léxico para el lenguaje de programación Tiger. El analizador léxico es la primera etapa del proceso de compilación, encargado de transformar el código fuente en una secuencia de tokens que serán utilizados posteriormente por el analizador sintáctico.

1.1 Objetivos

- Implementar un analizador léxico completo para Tiger usando ML-Lex
- Manejar correctamente todos los elementos léxicos del lenguaje Tiger
- Proporcionar manejo de errores robusto
- Crear una base sólida para las siguientes etapas del compilador

1.2 Herramientas Utilizadas

- Standard ML of New Jersey (SML/NJ)
- ML-Lex para la generación del analizador léxico
- Herramientas de construcción CM

2. Estructura del Proyecto

2.1 Archivos Principales

- cm.cm: Archivo de configuración para el sistema de construcción
- driver.sml: Controlador principal que maneja la entrada/salida
- errormsg.sml: Sistema de manejo de errores
- tokens.sig: Definición de la interfaz de tokens
- tokens.sml: Implementación de la estructura de tokens
- tiger.lex: Especificación del analizador léxico

2.2 Jerarquía de Componentes

Analizador Léxico Tiger

├── Sistema de Construcción (cm.cm)

├── Interfaz de Tokens (tokens.sig)

├── Implementación de Tokens (tokens.sml)

├── Manejo de Errores (errormsg.sml)

├── Especificación Léxica (tiger.lex)

└── Driver Principal (driver.sml)

3. Implementación

3.1 Tokens del Lenguaje

El analizador léxico reconoce las siguientes categorías de tokens:

Palabras Reservadas:

- Estructuras de control: while, for, to, break, let, in, end
- Declaraciones: function, var, type, array
- Condicionales: if, then, else
- Otros: do, of, nil

Operadores y Separadores:

- Aritméticos: +, -, *, /
- Comparación: =, <>, <, <=, >, >=
- Lógicos: &, |
- Asignación: :=
- Otros: , : ; ( ) [ ] { } .

3.2 Características Especiales

Manejo de Strings:

```
<INITIAL>\"          => (YYBEGIN STRING;
                    stringStart := yypos;
                    stringBuffer := "";
                    inString := true;
                    continue());
<STRING>\"           => (YYBEGIN INITIAL;
                    inString := false;
                    Tokens.STRING(!stringBuffer, !stringStart, yypos+1));
```

Manejo de Comentarios:

```
<INITIAL>"/*"        => (commentLevel := !commentLevel + 1;
                     YYBEGIN COMMENT;
                     continue());
<COMMENT>"/*"        => (commentLevel := !commentLevel + 1;
                     continue());
<COMMENT>"*/"        => (commentLevel := !commentLevel - 1;
                     if !commentLevel = 0 then YYBEGIN INITIAL else ();
                     continue());
```

3.3 Manejo de Errores

El sistema implementa detección y reporte de:

- Strings no cerrados
- Comentarios no cerrados
- Caracteres ilegales
- Valores numéricos inválidos

4. Pruebas

4.1 Caso de Prueba: Ámbitos Anidados (test01.tig)

Código a analizar:

```
/* test01.tig -- nested scopes and shadowing */
let var x := 5
in let var x := 6
       var y := 7
   in x+y
   end
   + x
   + y
end
```

Salida del lexer:

```
- Parse.parse "test01.tig";
LET 48:51
VAR 52:55
ID(x) 56:57
ASSIGN 58:60
INT(5) 61:62
IN 63:65
LET 66:69
VAR 70:73
ID(x) 74:75
ASSIGN 76:78
INT(6) 79:80
VAR 88:91
ID(y) 92:93
ASSIGN 94:96
INT(7) 97:98
IN 102:104
ID(x) 105:106
PLUS 106:107
ID(y) 107:108
END 112:115
PLUS 124:125
ID(x) 126:127
PLUS 131:132
ID(y) 133:134
END 140:143
EOF 143:143
val it = () : unit
```

4.2 Caso de Prueba: Manejo de Tipos (test02.tig)

Código a analizar:

```
/* test02.tig -- separate namespaces for types and variables/functions */
let type x = {fst:int, snd:int}
    var int := x {fst = 3, snd = 4}
    var x := 5
in
    int.fst +
    x +
    int.snd
end
```

Salida del lexer:

```
- Parse.parse "test02.tig";
LET 74:77
TYPE 78:82
ID(x) 83:84
EQ 85:86
LBRACE 87:88
ID(fst) 88:91
COLON 91:92
ID(int) 92:95
COMMA 95:96
ID(snd) 97:100
COLON 100:101
ID(int) 101:104
RBRACE 104:105
VAR 110:113
ID(int) 114:117
ASSIGN 118:120
ID(x) 121:122
LBRACE 123:124
ID(fst) 124:127
EQ 128:129
INT(3) 130:131
COMMA 131:132
ID(snd) 133:136
EQ 137:138
INT(4) 139:140
RBRACE 140:141
VAR 146:149
ID(x) 150:151
ASSIGN 152:154
INT(5) 155:156
IN 157:159
ID(int) 164:167
DOT 167:168
ID(fst) 168:171
PLUS 172:173
ID(x) 179:180
PLUS 181:182
ID(int) 188:191
DOT 191:192
ID(snd) 192:195
END 196:199
EOF 199:199
val it = () : unit
```

4.3 Caso de Prueba: Función Factorial (test4.tig)

Código a analizar:

```
/* define a recursive function */
let

/* calculate n! */
function nfactor(n: int): int =
                if  n = 0
                        then 1
                        else n * nfactor(n-1)

in
        nfactor(10)
end
```

Salida del lexer:

```
- Parse.parse "test4.tig";
LET 34:37
FUNCTION 58:66
ID(nfactor) 67:74
LPAREN 74:75
ID(n) 75:76
COLON 76:77
ID(int) 78:81
RPAREN 81:82
COLON 82:83
ID(int) 84:87
EQ 88:89
IF 92:94
ID(n) 96:97
EQ 98:99
INT(0) 100:101
THEN 106:110
INT(1) 111:112
ELSE 116:120
ID(n) 121:122
TIMES 123:124
ID(nfactor) 125:132
LPAREN 132:133
ID(n) 133:134
MINUS 134:135
INT(1) 135:136
RPAREN 136:137
IN 139:141
ID(nfactor) 143:150
LPAREN 150:151
INT(10) 151:153
RPAREN 153:154
END 155:158
EOF 159:159
val it = () : unit
```

4.4 Caso de Prueba: Ámbitos Anidados (test04.tig)

Código a analizar:

```
/* test04.tig -- break and loop counter restrictions */
let var y := 3
    var x := "hello"
    var z := 32
in
    3+2;
    break;   /* error, not within loop */
    for x := y to z do    /* loop counter x shadows local x, ok */
      (z := x + 1;  /* ok, but does not affect stopping condition */
       break;       /* ok */
       x := z;      /* error, assigning to loop counter */
       while y < z do
         (x := 31415;  /* error, assigning to loop counter */
          break;       /* ok */
          y := x*x);    /* ok */
       break;       /* still ok */
       let var x := 42
       in x := 88 end;  /* ok */
       y := y - 1); /* ok */
    break;          /* error, not within loop */
    x := "world";   /* ok, local x is a string */
    while y < z do
       break;       /* ok */
    while z >= y do
       let function f() =
          break  /* TECHNICALLY an error, but quite subtle */
       in break    /* ok */
       end
end
```

Salida del lexer:

```
- Parse.parse "test04.tig";
LET 56:59
VAR 60:63
ID(y) 64:65
ASSIGN 66:68
INT(3) 69:70
VAR 75:78
ID(x) 79:80
ASSIGN 81:83
STRING(hello) 84:91
VAR 96:99
ID(z) 100:101
ASSIGN 102:104
INT(32) 105:107
IN 108:110
INT(3) 115:116
PLUS 116:117
INT(2) 117:118
SEMICOLON 118:119
BREAK 124:129
SEMICOLON 129:130
FOR 166:169
ID(x) 170:171
ASSIGN 172:174
ID(y) 175:176
TO 177:179
ID(z) 180:181
DO 182:184
LPAREN 235:236
ID(z) 236:237
ASSIGN 238:240
ID(x) 241:242
PLUS 243:244
INT(1) 245:246
SEMICOLON 246:247
BREAK 305:310
SEMICOLON 310:311
ID(x) 334:335
ASSIGN 336:338
ID(z) 339:340
SEMICOLON 340:341
```

```
END 612:615
SEMICOLON 615:616
ID(y) 634:635
ASSIGN 636:638
ID(y) 639:640
MINUS 641:642
INT(1) 643:644
RPAREN 644:645
SEMICOLON 645:646
BREAK 660:665
SEMICOLON 665:666
ID(x) 709:710
ASSIGN 711:713
STRING(world) 714:721
SEMICOLON 721:722
WHILE 759:764
ID(y) 765:766
LT 767:768
ID(z) 769:770
DO 771:773
BREAK 781:786
SEMICOLON 786:787
WHILE 807:812
ID(z) 813:814
GE 815:817
ID(y) 818:819
DO 820:822
LET 830:833
FUNCTION 834:842
ID(f) 843:844
LPAREN 844:845
RPAREN 845:846
EQ 847:848
BREAK 860:865
IN 919:921
BREAK 922:927
END 947:950
END 951:954
EOF 954:954
val it = () : unit
-
```

5. Características Destacadas

5.1 Manejo de Comentarios Anidados

El analizador maneja correctamente comentarios anidados usando un contador de nivel:

- Incrementa el contador con cada “/*”
- Decrementa con cada “*/”
- Verifica comentarios no cerrados al final del archivo

5.2 Manejo de Strings

Características implementadas:

- Secuencias de escape (\n, \t, ", \)
- Buffer para construcción de strings
- Detección de strings no cerrados
- Tracking de posición inicial del string

5.3 Seguimiento de Posición

El analizador mantiene información precisa sobre:

- Número de línea actual
- Posición en la línea
- Posición absoluta en el archivo

6. Conclusiones

El desarrollo del analizador léxico para el lenguaje Tiger representa un paso fundamental en la construcción de un compilador completo. A través de este proyecto, hemos logrado implementar un sistema robusto que no solo cumple con las especificaciones básicas del lenguaje, sino que también incorpora características avanzadas de manejo de errores y procesamiento de tokens.

La experiencia de desarrollo ha revelado la importancia crucial del análisis léxico en el proceso de compilación. El analizador no solo identifica y clasifica los elementos léxicos del

lenguaje, sino que también proporciona una base sólida para las etapas posteriores del compilador, como el análisis sintáctico y semántico.

Entre los aspectos más destacables de nuestra implementación se encuentran:

La gestión sofisticada de comentarios anidados, que demuestra la capacidad del analizador para manejar estructuras complejas y mantener un estado coherente durante todo el proceso de análisis. Este aspecto es particularmente importante en un lenguaje de programación moderno como Tiger, donde la legibilidad y mantenibilidad del código son fundamentales.

El manejo robusto de cadenas de caracteres, que incluye el procesamiento de secuencias de escape y la detección de errores comunes, como strings no cerrados. Esta característica es esencial para garantizar la integridad de los datos y la seguridad en el procesamiento de texto.

El sistema de seguimiento de posición preciso, que facilita la depuración y mejora significativamente la experiencia del desarrollador al proporcionar información detallada sobre la ubicación de errores en el código fuente.

Recomendaciones para el Futuro

Para continuar mejorando el analizador, se sugieren las siguientes líneas de desarrollo:
La implementación de un sistema de recuperación de errores más sofisticado que permita al compilador continuar el análisis incluso después de encontrar errores léxicos, maximizando así la utilidad de los mensajes de error para el usuario.

La expansión del conjunto de pruebas para cubrir casos más específicos y edge cases, garantizando la robustez del analizador en situaciones menos comunes pero igualmente importantes.

La integración de herramientas de análisis de rendimiento para optimizar el procesamiento de archivos grandes y mejorar la eficiencia general del analizador.

La incorporación de características adicionales de diagnóstico que ayuden a los desarrolladores a identificar y corregir problemas en su código de manera más eficiente.

Reflexión Final

El desarrollo de este analizador léxico no solo ha cumplido con los requisitos técnicos establecidos, sino que también ha proporcionado valiosas lecciones sobre el diseño e implementación de componentes fundamentales de un compilador. La experiencia adquirida en el manejo de estados, procesamiento de tokens y gestión de errores será invaluable para las siguientes fases del desarrollo del compilador.

La modularidad y extensibilidad del diseño implementado permitirán futuras mejoras y adaptaciones, manteniendo al mismo tiempo la robustez y confiabilidad necesarias para un componente tan crítico del proceso de compilación. Este proyecto sienta las bases para el desarrollo continuo y la evolución del compilador Tiger, proporcionando una plataforma sólida para futuras expansiones y mejoras.

7. Referencias

1. Manual de Referencia de Tiger
2. Documentación de ML-Lex
3. Modern Compiler Implementation in ML
4. Standard ML of New Jersey Documentation

8. Guía de Uso

Para utilizar el analizador:

1. Es necesario instalar todo para ingresar al compilador de SML a través de la terminal. Ingresar en la terminal luego:

   ```
   SML
   ```

2. Compilar el proyecto:

   ```
   CM.make("cm.cm");
   ```

```
(base) friasluna@friasluna:~/Desktop/Procesamiento_Lenguaje_Compilador/Tareas Reportes/T2$ sml
Standard ML of New Jersey v110.79 [built: Mon Apr 22 10:14:55 2024]
- CM.make("cm.cm");
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
[library $/pgraph.cm is stable]
[library $smlnj/internal/srcpath-lib.cm is stable]
[library $SMLNJ-BASIS/basis.cm is stable]
[library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
[autoloading done]
[scanning cm.cm]
[library $/ml-yacc-lib.cm is stable]
[loading (cm.cm):tokens.sig]
[loading (cm.cm):errormsg.sml]
[loading (cm.cm):tiger.lex.sml]
[loading (cm.cm):tokens.sml]
[loading (cm.cm):driver.sml]
[New bindings added.]
val it = true : bool
-
```

Debe retornar true para indicar que todo está correcto.

3. Analizar un archivo:

   Parse.parse "nombre_archivo.tig";

   Ejemplo de prueba:

```
- Parse.parse "test4.tig";
LET 34:37
FUNCTION 58:66
ID(nfactor) 67:74
LPAREN 74:75
ID(n) 75:76
COLON 76:77
ID(int) 78:81
RPAREN 81:82
COLON 82:83
ID(int) 84:87
EQ 88:89
IF 92:94
ID(n) 96:97
EQ 98:99
INT(0) 100:101
THEN 106:110
INT(1) 111:112
ELSE 116:120
ID(n) 121:122
TIMES 123:124
ID(nfactor) 125:132
LPAREN 132:133
ID(n) 133:134
MINUS 134:135
INT(1) 135:136
RPAREN 136:137
IN 139:141
ID(nfactor) 143:150
LPAREN 150:151
INT(10) 151:153
RPAREN 153:154
END 155:158
EOF 159:159
val it = () : unit
```