Evidencia 1. Resaltador de sintaxis

José Antonio Pacheco Chargoy - A01663815
Eashley Brittney Martínez Vergara - A01663894
Jose Luis Almeida Esparza - A01028493
Carlos Alberto Páez de la Cruz - A01703050

Implementación de métodos computacionales

Gpo 647

17/05/2024

1) Selecciona UN (1) lenguaje de programación que te resulte familiar (por ejemplo, C, C++,
C#, Java, JavaScript) y que desees emular.

El lenguaje seleccionado para este proyecto es C, un lenguaje de programación estructurado y ampliamente utilizado.

2) Analiza el lenguaje de programación que seleccionaste y determina, por lo menos, 30 categorías léxicas que definan: palabras reservadas, operadores, literales, comentarios, expresiones de asignación, funciones, etc. Deberás también definir la categoría léxica principal que reúne a las otras 30. De esa manera, tendrás la estructura del lenguaje a emular.

Para emular el lenguaje C, se han identificado las siguientes categorías léxicas:

1. **Palabras reservadas**: int, char, float, double, void, if, else, for, while, do, return, switch, case, default, break, struct, static, sizeof, const.
2. **Operadores**: +, -, *, /, %, ++, --, =, +=, -=, *=, /=, %=, ==, !=, <, <=, >, >=, &&, ||, !, &, |, ^, ~, <<, >>, ->
3. **Literales**:
    - Números enteros: 123, 0x1A, 075.
    - Números de punto flotante: 1.23, 3.14e-2.
    - Caracteres: 'a', '\n'.
    - Cadenas de texto: "Hello, world!".
4. **Comentarios**:
    - De línea: // Este es un comentario de línea.
    - De bloque: /* Este es un comentario de bloque */.
5. **Identificadores**: main, foo, bar.
6. **Puntuación**: ‘;’, ‘,’, ‘(‘,’)’, ‘{‘,’ }’, ‘[‘,’]’.
7. **Expresiones de asignación**: a = b, c += d, e = f * g.
8. **Funciones**: printf, scanf.

Estatutos

- **Declaración <var_declaration>**
- **Asignación <var_assignment>**
- **declaración + asignación en el mismo renglón <var_declaration_and_assignament>**

Selección

- **<condition>**
- **<if>**
- **<if- else>**
- **<if else if*>**
- **<if else >**

Ciclos

- **<for >**

- **<while >**
- **<do-while>**

Función
Programa

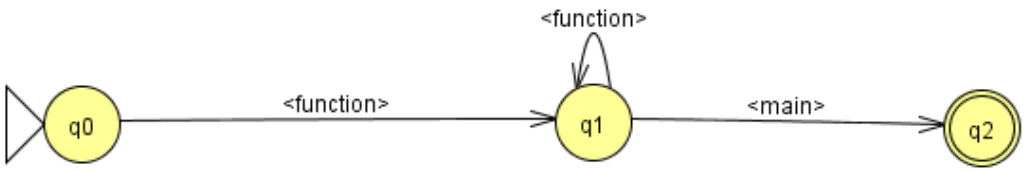
3) Define la BNF (Notación de Backus-Naur) para representar todas las categorías léxicas que seleccionaste.

<program> ::= <statement-list>
<statement-list> ::= <statement> | <statement-list><statement>
<statement> ::= <declaration-statement> | <control-statement>
<block> ::= ‘{‘ <statement-list> ‘}’
<control-statement> ::= <selection-statement> | <cycle-statement> | <function-call>
<selection-statement> ::= ‘if’ ‘(‘ <statement> ‘)’ <block> |
‘if’ ‘(‘ <statement> ‘)’ <block> ‘else’ <block>
<cycle-statement> ::= <for-loop> | <while-loop> | <do-while-loop>
<declaration-statement> ::= <var-declaration> | <fun-declaration>

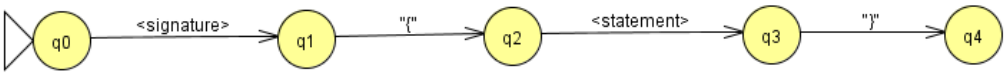
<program> ::= <function>
<function> ::= <signature> { signature { stature> }
<signature> ::= <type><space><fname> (<parameter>)
<statute>::= <statute> | <fun_call> | <var_declaration> | <var assignment> | <loop> | <selection>

4) Realiza la conversión de BNF a AFD (autómata finito determinista). Construye todos los AFDs que necesites usando JFlap.

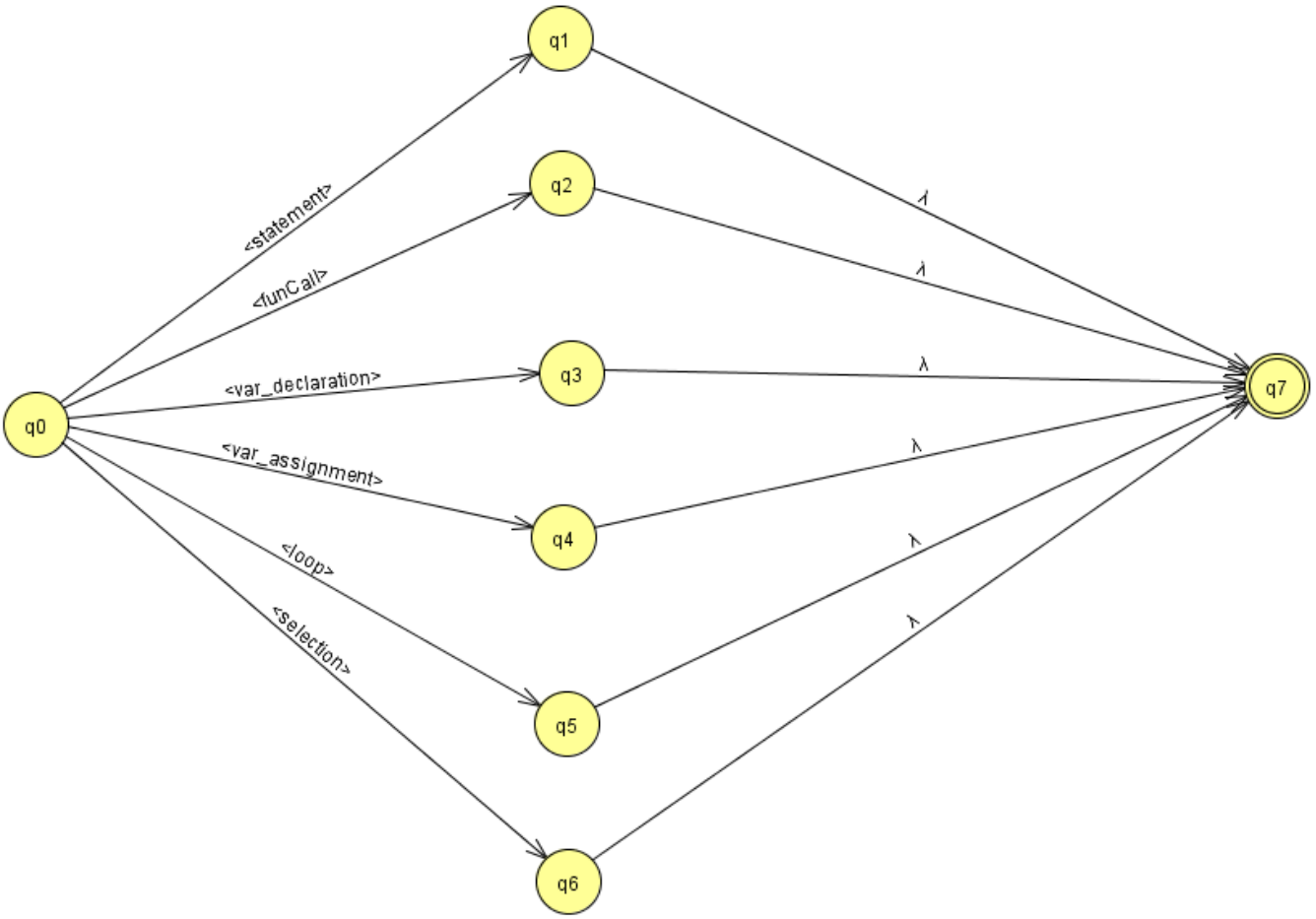
1. <main>



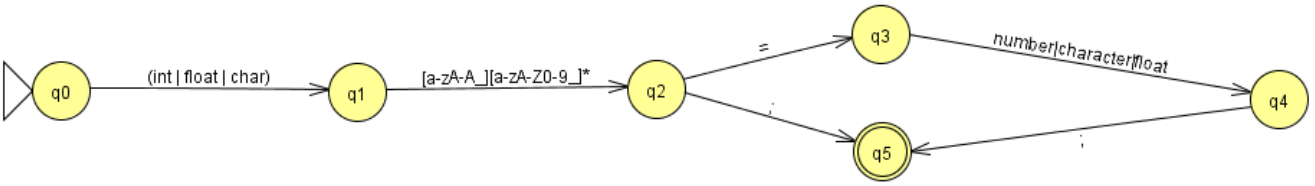
2. <function>



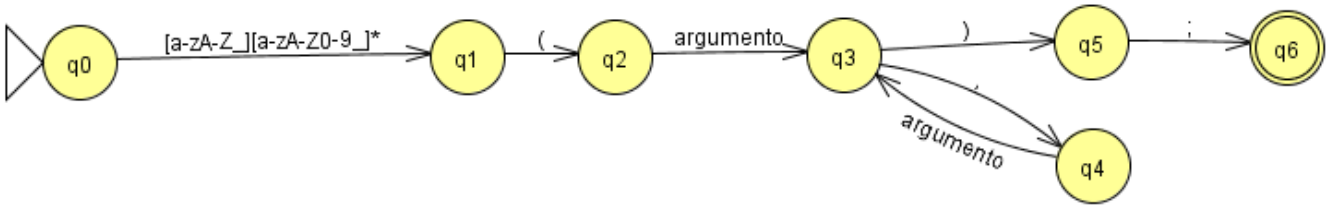
3. Statements



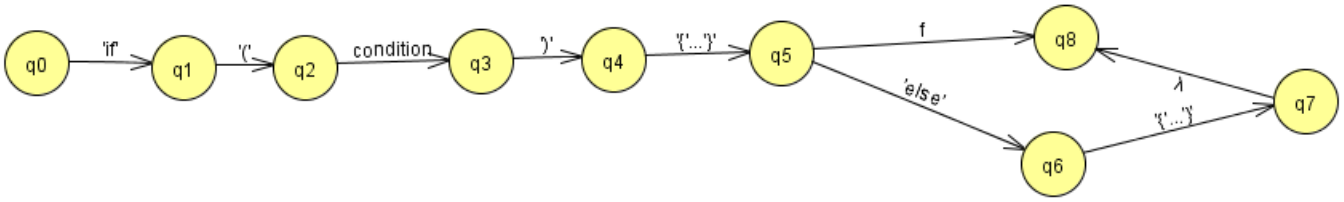
4. Var Declaration and Var assignment



5. Fun Call



6. Selection



5) Usando Racket, implementa un motor de expresiones regulares que sea capaz de leer una cadena de texto conteniendo código fuente escrito en el lenguaje de programación que seleccionaste. Esta cadena de texto corresponde a la entrada (input) a ser analizada. La cadena de entrada puede ser escrita directamente en la interfaz o leída desde un archivo de texto.

Código en Racket del lexer:

```
Unset
#lang racket

(require parser-tools/lex)
;(require parser-tools/lex-sre)
(require (prefix-in : parser-tools/lex-sre))

(define-tokens literal-tokens (NUMBER STRING NULL BOOLEAN))
(define-tokens identifier-tokens (IDENTIFIER))
```

```
(define-empty-tokens keyword-tokens
                     (IF ELSE FOR WHILE BREAK CONTINUE RETURN FUNCTION VAR
LET CONST CLASS))

(define-empty-tokens expression-operator-tokens (PLUS MINUS))

(define-empty-tokens term-operator-tokens
                     (MULTIPLY DIVIDE
                               ASSIGNMENT
                               EQUAL
                               NOT_EQUAL
                               LESS_THAN
                               GREATER_THAN
                               LESS_THAN_OR_EQUAL
                               GREATER_THAN_OR_EQUAL
                               AND
                               OR))

(define-empty-tokens
 punctuation-tokens
 (SEMICOLON COMMA DOT LEFT_BRACE RIGHT_BRACE LEFT_BRACKET RIGHT_BRACKET
LEFT_PAREN RIGHT_PAREN EOF))

(define lex

  (lexer-src-pos
   ;COMMENTS
   [(:seq "//" (:* (:~ #\newline)) #\newline) (return-without-pos (lex
input-port))]
   ;WHITESPACE
   [whitespace (return-without-pos (lex input-port))]
   ;KEYWORDS
   ["if" (token-IF)]
   ["else" (token-ELSE)]
   ["for" (token-FOR)]
   ["while" (token-WHILE)]
   ["break" (token-BREAK)]
   ["continue" (token-CONTINUE)]
   ["return" (token-RETURN)]
   ["function" (token-FUNCTION)]
   ["var" (token-VAR)]
   ["let" (token-LET)]
   ["const" (token-CONST)]
   ["class" (token-CLASS)]
   ;OPERATORS
   ["+" (token-PLUS)]
   ["-" (token-MINUS)]
```

```
   ["*" (token-MULTIPLY)]
   ["/" (token-DIVIDE)]
   ["=" (token-ASSIGNMENT)]
   ["==" (token-EQUAL)]
   ["!=" (token-NOT_EQUAL)]
   ["<" (token-LESS_THAN)]
   [">" (token-GREATER_THAN)]
   ["<=" (token-LESS_THAN_OR_EQUAL)]
   [">=" (token-GREATER_THAN_OR_EQUAL)]
   ["&&" (token-AND)]
   ["||" (token-OR)]
   ;PUNCTUATION
   [";" (token-SEMICOLON)]
   ["(" (token-LEFT_PAREN)]
   [")" (token-RIGHT_PAREN)]
   ["[" (token-LEFT_BRACKET)]
   ["]" (token-RIGHT_BRACKET)]
   ["{" (token-LEFT_BRACE)]
   ["}" (token-RIGHT_BRACE)]
   ["." (token-DOT)]
   ["," (token-COMMA)]
   ;STRING LITERAL
   [(:seq #\" (:* (:~ #\")) #\") (token-STRING lexeme)]
   ;NUMBER LITERAL
   [(:seq (:+ numeric) (:? (:seq #\. (:* numeric)))) (token-NUMBER
(string->number lexeme))]
   ;NULL LITERAL
   ["null" (token-NULL lexeme)]
   ;TRUE LITERAL
   ["true" (token-BOOLEAN lexeme)]
   ;FALSE LITERAL
   ["false" (token-BOOLEAN lexeme)]
   ;IDENTIFIERS
   [(:seq (:or alphabetic #\_) (:* (union alphabetic numeric #\_)))
(token-IDENTIFIER lexeme)]
   ;EOF
   [(eof) (token-EOF)]))

(define src-code (open-input-file "src/test.js"))
(port-count-lines! src-code) ;enable lines and cols nums

;exports
(provide literal-tokens)
(provide identifier-tokens)
(provide keyword-tokens)
(provide expression-operator-tokens)
(provide term-operator-tokens)
(provide punctuation-tokens)
```

```
(provide lex)
```

Ejemplo de ejecución

6) Deberás realizar el análisis sintáctico de la cadena de entrada. Para ello, deberás descomponer la cadena de entrada en la subcadenas que la componen en lexemas y escanearlos para verificar que cumplan la BNF/AFD que definiste. Para este fin, se recomienda usar los paquetes Lexers y LALR Parsers de Racket (https://docs.racket-lang.org/parser-tools/index.html). De esa manera, dada una cadena de entrada, el analizador sintáctico que desarrollarás verificará si el código contenido en dicha cadena cumple o no la sintaxis que definiste.

Código en Racket del parser (analizador de sintaxis):

```
Unset
#lang racket

(require racket/runtime-path)

; Define the relative path to the file
(define-runtime-path source-code-file-path "input/source_code.txt")
(define-runtime-path parsed-code-file-path "output/parsed_code.html")

(require "lexer.rkt")

(require parser-tools/yacc)
(define (increment x)
  (cond
    [(number? x) (+ x 1)]
    [(symbol? x) (list 'post-increment x)]))

(define (decrement x)
  (cond
    [(number? x) (- x 1)]
    [(symbol? x) (list 'post-decrement x)]))

(define error-handler
  (lambda (tok-ok? tok-name tok-value start-pos end-pos)
    (void)))  ; No hacer nada, solo retornar un valor vacío

(define the-parser
  (parser [start c-program]
          [end EOF]
          [src-pos]
```

```
          [error error-handler]
          [tokens
           literal-tokens
           identifier-tokens
           keyword-tokens
           type-tokens
           expression-operator-tokens
           term-operator-tokens
           punctuation-tokens]
          [grammar
           [c-program [(main-function) $1]]
           [main-function
            [(INT IDENTIFIER LEFT_PAREN RIGHT_PAREN block)
             (if (equal? $2 "main") (list 'main-function $5) (error
"Expected 'main' function"))]]
           [block [(LEFT_BRACE statements RIGHT_BRACE) $2]]
           [statements [(statement statements) (cons $1 $2)] [() '()]]
           [statement
            [(declaration SEMICOLON) $1]
            [(expr SEMICOLON) $1]
            [(if-statement) $1]
            [(for-statement) $1]
            [(while-statement) $1]
            [(do-while-statement) $1]
            [(switch-statement) $1]
            [(return-statement) $1]
            [(continue-statement) $1]
            [(function-call SEMICOLON) $1]]
           [if-statement
            [(IF LEFT_PAREN expr RIGHT_PAREN block) (list 'if $3 $5)]
            [(IF LEFT_PAREN expr RIGHT_PAREN block ELSE block) (list
'if-else $3 $5 $7)]
            [(IF LEFT_PAREN expr RIGHT_PAREN statement) (list 'if $3 $5)]
            [(IF LEFT_PAREN expr RIGHT_PAREN statement ELSE statement) (list
'if-else $3 $5 $7)]]
           [for-statement
            [(FOR LEFT_PAREN for-init SEMICOLON for-cond SEMICOLON for-incr
RIGHT_PAREN block)
             (list 'for $3 $5 $7 $9)]
            [(FOR LEFT_PAREN for-init SEMICOLON for-cond SEMICOLON for-incr
RIGHT_PAREN statement)
             (list 'for $3 $5 $7 $9)]]
           [while-statement
            [(WHILE LEFT_PAREN expr RIGHT_PAREN block) (list 'while $3 $5)]
            [(WHILE LEFT_PAREN expr RIGHT_PAREN statement) (list 'while $3
$5)]]
           [do-while-statement
```

```
           [(DO block WHILE LEFT_PAREN expr RIGHT_PAREN SEMICOLON) (list
'do-while $2 $5)]
           [(DO statement WHILE LEFT_PAREN expr RIGHT_PAREN SEMICOLON)
(list 'do-while $2 $5)]]
          [switch-statement [(SWITCH LEFT_PAREN expr RIGHT_PAREN
switch-block) (list 'switch $3 $5)]]
          [switch-block [(LEFT_BRACE case-statements RIGHT_BRACE) $2]]
          [case-statements
           [(case-statement case-statements) (cons $1 $2)]
           [(default-statement case-statements) (cons $1 $2)]
           [() '()]]
          [case-statement [(CASE expr COLON statements break-statement)
(list 'case $2 $4 $5)]]
          [default-statement
           [(DEFAULT COLON statements break-statement) (list 'default $3
$4)]
           [(DEFAULT COLON statements) (list 'default $3)]]
          [break-statement [(BREAK SEMICOLON) 'break]]
          [return-statement [(RETURN expr SEMICOLON) (list 'return $2)]
[(RETURN SEMICOLON) 'return]]
          [continue-statement [(CONTINUE SEMICOLON) 'continue]]
          [for-init [(declaration) $1] [(expr) $1] [() #f]]
          [for-cond [(expr) $1] [() #t]]
          [for-incr [(expr) $1] [() #f]]
          [function-call [(IDENTIFIER LEFT_PAREN args RIGHT_PAREN) (list
'call $1 $3)]]
          [args [(expr) (list $1)] [() '()]]
          [declaration
           [(type-token IDENTIFIER) (list 'declare $1 $2)]
           [(type-token IDENTIFIER ASSIGNMENT expr) (list 'declare-init $1
$2 $4)]]
          [type-token [(INT) 'int] [(CHAR) 'char] [(FLOAT) 'float]
[(DOUBLE) 'double] [(VOID) 'void]]
          [expr
           [(IDENTIFIER ASSIGNMENT expr) (list 'assign $1 $3)]
           [(expr PLUS expr) (list '+ $1 $3)]
           [(expr MINUS expr) (list '- $1 $3)]
           [(expr MULTIPLY expr) (list '* $1 $3)]
           [(expr DIVIDE expr) (list '/ $1 $3)]
           [(expr EQUAL expr) (list '== $1 $3)]
           [(expr NOT_EQUAL expr) (list '!= $1 $3)]
           [(expr LESS_THAN expr) (list '< $1 $3)]
           [(expr GREATER_THAN expr) (list '> $1 $3)]
           [(expr LESS_THAN_OR_EQUAL expr) (list '<= $1 $3)]
           [(expr GREATER_THAN_OR_EQUAL expr) (list '>= $1 $3)]
           [(expr AND expr) (list '&& $1 $3)]
           [(expr OR expr) (list '|| $1 $3)]
           [(expr INCREMENT) (increment $1)]
```

```
            [(expr DECREMENT) (decrement $1)]
            [(IDENTIFIER) $1]
            [(NUMBER) $1]
            [(LEFT_PAREN expr RIGHT_PAREN) $2]
            [(STRING) $1]]
           [term
            [(IDENTIFIER) $1]
            [(NUMBER) $1]
            [(STRING) $1]
            [(LEFT_PAREN expr RIGHT_PAREN) $2]
            [(expr INCREMENT) (increment $1)]
            [(expr DECREMENT) (decrement $1)]]]))

;; Function to generate highlighted HTML
(define (surround1 s1 c)
  (string-append "<" c ">" s1))

(define (surround2 s1 c)
  (string-append "<" c ">" s1 "</" c ">"))

(define (surround3 s1 c)
  (string-append s1 "</" c ">"))

(define (sep str)
  (regexp-split #rx"(?<=\\]|\\[|(\r\n)|[+-/()<>{}=;:
])|(?=\\]|\\[|(\r\n)|[+-/()<>{}=;: ])" str))

(define (surroundRegexp s1)
  (cond
    [(regexp-match #rx"[0-9]+|^int$" s1) (set! s1 (string-append "<span
style=\"color: #0077cc;\">" s1 "</span>"))]
    [(regexp-match #rx"\\." s1) (set! s1 (string-append "<span
style=\"color: #0077cc;\">" s1 "</span>"))]
    [(regexp-match #rx"[><*=/+-]" s1) (set! s1 (string-append "<span
style=\"color: #ff7f00;\">" s1 "</span>"))]
    [(regexp-match #rx"^else$|^if$" s1) (set! s1 (string-append "<span
style=\"color: #a020f0;\">" s1 "</span>"))]
    [(regexp-match #rx"^for$|^while$|^switch$|^do$" s1) (set! s1
(string-append "<span style=\"color: #a020f0;\">" s1 "</span>"))]
    [(regexp-match #rx"\".*\"" s1) (set! s1 (string-append "<span
style=\"color: #00a86b;\">" s1 "</span>"))]
    [(regexp-match #rx"(?<=\")(?=.)" s1) (set! s1 (string-append "<span
style=\"color: #00a86b;\">" s1))]
    [(regexp-match #rx"(?<=.)(?=\")" s1) (set! s1 (string-append s1
"</span>"))]
    [(regexp-match #rx"(?<=')(?=.)" s1) (set! s1 (string-append "<span
style=\"color: #00a86b;\">" s1))]
```

```
    [(regexp-match #rx"(?<=.)(?=')" s1) (set! s1 (string-append s1
"</span>"))]
    [(regexp-match #rx"'.*'" s1) (set! s1 (string-append "<span
style=\"color: #00a86b;\">" s1 "</span>"))]
    [(regexp-match #rx"\\]|\\[|[(]|[)]|[{]|[}]" s1) (set! s1 (string-append
"<span style=\"color: #FE591A;\">" s1 "</span>"))]
    [(regexp-match #rx"^default$|^case$|^break$|^print$" s1) (set! s1
(string-append "<span style=\"color: #F93B08;\">" s1 "</span>"))]
    [(regexp-match #rx"^printf$" s1) (set! s1 (string-append "<span
style=\"color: #3E7900;\">" s1 "</span>"))]
    [(regexp-match #rx"[,]" s1) (set! s1 (string-append "<span
style=\"color: #666666;\">" s1 "</span>"))]
    [(regexp-match #rx" " s1) (set! s1 " ")]
    [(regexp-match #rx"(\r\n|\n)" s1) (set! s1 "<br>")]  ; Manejar tanto
\r\n como \n
    [else s1])  ; Devolver s1 sin cambios si no coincide con ninguna regla
  s1)

(define (resaltar2 x)
  (set! x (sep x))
  (set! x (map (lambda (lst) (surroundRegexp lst)) x))
  (set! x (string-join x ""))
  (set! x (string-append "<pre style=\"background-color: #f4f4f4; padding:
10px;\">" x "</pre>"))
  x)

(define (resultado x)
  (resaltar2 x))

(define (append-parse-result highlighted-code result)
  (define status (if (eq? result 'parse-error) "Sintaxis Incorrecta"
"Sintaxis Correcta"))
  (define status-html
    (string-append
     "<div style=\"background-color: black; color: white; padding: 10px;
margin-top: 20px; font-weight: bold;\">"
     status
     "</div>"))

  (define full-html
    (string-append
     "<!DOCTYPE html>\n<html lang=\"es\">\n<head>\n<meta
charset=\"UTF-8\">\n<title>Código Resaltado</title>\n</head>\n<body>\n"
     highlighted-code
     status-html
     "\n</body>\n</html>"))

  (call-with-output-file parsed-code-file-path
```

```
    (lambda (port)
      (display full-html port))
    #:exists 'replace))

;; Main function to parse the source code and generate highlighted HTML
(define (main)
  (define src-code-string (file->string source-code-file-path))
  (define highlighted-code (resultado src-code-string))

  (define src-code (open-input-string src-code-string))
  (port-count-lines! src-code)
  (define result (with-handlers ([exn:fail? (lambda (e) 'parse-error)])
                   (the-parser (λ () (lex src-code)))))

  (append-parse-result highlighted-code result))

(main)
```

7) El resultado del procesamiento del programa debe convertir la cadena de entrada a una salida (output) en formato HTML+CSS capaz de resaltar el léxico del lenguaje de programación seleccionado. Cada léxico debe tener un color diferente. En la cadena de salida en HTML+CSS, en la parte inferior, debe haber un bloque de fondo negro (simulando una consola), informando si el pedazo de código analizado es léxicamente correcto o no.

A continuación se muestran dos imágenes como ejemplos de la salida generada por el programa. La primera imagen muestra un ejemplo de una sintaxis incorrecta según las gramáticas para C definidas anteriormente, y la segunda imagen muestra un ejemplo de una sintaxis correcta según dichas gramáticas.

```c
int main() {
    printf("Hello, World!");
    int x = 10;
    for (int i = 0; i < 5; i++) {
        x = x + i;
    }
    if (x > 15) {
        printf("x is greater than 15");
    } else {
        printf("x is not greater
    }
}
```

Sintaxis Incorrecta

```c
int main() {
    printf("Hello, World!");
    int x = 10;
    for (int i = 0; i < 5; i++) {
        x = x + i;
    }
    if (x > 15) {
        printf("x is greater than 15");
    } else {
        printf("x is not greater" );
    }
}
```

Sintaxis Correcta

8) Generar 5 archivos de código en el lenguaje de programación seleccionado. Estos archivos de código deben tener tamaños diferentes de números de línea de código: 10 líneas, 20 líneas, 50 líneas, 70 líneas, 100 líneas. Ingresa tus archivos de código en tu programa y analiza los resultados determinando si el analizador funciona correctamente.

Archivo de ejemplo 1:

```C/C++
int main() {
    int day = 4;

    while (day < 7) {
        printf("Day");
        day++;
    }

    printf("Done with days");
    return 0;
}
```

Tiempo de ejecución:

```
Welcome to DrRacket, version 8.12 [cs].
Language: racket, with debugging; memory limit: 128 MB.
184 shift/reduce conflicts
(list
 (position-token 'VOID (position 1 #f #f) (position 5 #f #f))
 (position-token (token 'IDENTIFIER "main") (position 6 #f #f) (position 10 #f #f))
 (position-token 'LEFT_PAREN (position 10 #f #f) (position 11 #f #f))
 (position-token 'RIGHT_PAREN (position 11 #f #f) (position 12 #f #f))
 (position-token 'LEFT_BRACE (position 12 #f #f) (position 13 #f #f)))
Time taken: 0.660888671875 ms
>
```

Archivo de ejemplo 2:

```C/C++
int main() {
    int salary = 1000;

    while (salary < 5000) {
        printf("Salary");
        salary++;
    }

    do {
        printf("Salary");
        salary++;
    } while (salary < 7000);

    for (int i = 0; i < 3; i++) {
        printf("Increment");
    }

    if (salary >= 7000) {
        printf("Reached goal");
```

```
    }

    return 0;
}
```

Tiempo de ejecución:

```
Welcome to DrRacket, version 8.12 [cs].
Language: racket, with debugging; memory limit: 128 MB.
184 shift/reduce conflicts
(list
 (position-token 'VOID (position 1 #f #f) (position 5 #f #f))
 (position-token (token 'IDENTIFIER "main") (position 6 #f #f) (position 10 #f #f))
 (position-token 'LEFT_PAREN (position 10 #f #f) (position 11 #f #f))
 (position-token 'RIGHT_PAREN (position 11 #f #f) (position 12 #f #f))
 (position-token 'LEFT_BRACE (position 12 #f #f) (position 13 #f #f)))
Time taken: 1.06201171875 ms
> |
```

Archivo de ejemplo 3:

```
C/C++
int main() {
    int month = 3;

    switch (month) {
    case 12:
        printf("December");
        break;
    case 1:
        printf("January");
        break;
    default:
        printf("Other Month");
    }

    while (month < 6) {
        printf("Month");
        month++;
    }

    do {
        printf("Month");
        month++;
    } while (month < 9);

    if (month >= 9) {
        printf("Month is greater than or equal to 9");
    }
```

```
    int count = 0;
    while (count < 3) {
        printf("Count");
        count++;
    }

    for (int i = 0; i < 3; i++) {
        printf("i");
    }

    int extra = 0;
    while (extra < 6) {
        printf("Extra");
        extra++;
    }

    do {
        printf("Extra");
        extra++;
    } while (extra < 9);

    if (extra >= 9) {
        printf("Extra is greater than or equal to 9");
    }

    for (int j = 0; j < 3; j++) {
        printf("Loop");
    }

    int k = 0;
    while (k < 2) {
        printf("While Loop");
        k++;
    }

    return 0;
}
```

Tiempo de ejecución:

```
Welcome to DrRacket, version 8.12 [cs].
Language: racket, with debugging; memory limit: 128 MB.
184 shift/reduce conflicts
(list
 (position-token 'VOID (position 1 #f #f) (position 5 #f #f))
 (position-token (token 'IDENTIFIER "main") (position 6 #f #f) (position 10 #f #f))
 (position-token 'LEFT_PAREN (position 10 #f #f) (position 11 #f #f))
 (position-token 'RIGHT_PAREN (position 11 #f #f) (position 12 #f #f))
 (position-token 'LEFT_BRACE (position 12 #f #f) (position 13 #f #f)))
Time taken: 3.203369140625 ms
> 
```

Archivo de ejemplo 4:

C/C++

```c
int main() {
    int grade = 75;

    switch (grade) {
    case 90:
        printf("A");
        break;
    case 80:
        printf("B");
        break;
    default:
        printf("C or below");
    }

    while (grade < 80) {
        printf("Grade");
        grade++;
    }

    do {
        printf("Grade");
        grade++;
    } while (grade < 85);

    if (grade >= 85) {
        printf("Grade is greater than or equal to 85");
    }

    int count = 0;
    while (count < 3) {
        printf("Count");
        count++;
    }
```

```
    for (int i = 0; i < 3; i++) {
        printf("i");
    }

    int a = 0;
    while (a < 5) {
        if (a == 2) {
            a++;
            continue;
        }
        printf("a");
        a++;
    }

    int b = 0;
    do {
        if (b == 1) {
            b++;
            continue;
        }
        printf("b");
        b++;
    } while (b < 5);

    int x = 0;
    while (x < 7) {
        printf("x");
        x++;
    }

    for (int y = 0; y < 7; y++) {
        printf("y");
    }

    int z = 0;
    while (z < 8) {
        printf("z");
        z++;
    }

    for (int m = 0; m < 8; m++) {
        printf("m");
    }

    return 0;
}
```

Tiempo de ejecución:

```
Welcome to DrRacket, version 8.12 [cs].
Language: racket, with debugging; memory limit: 128 MB.
184 shift/reduce conflicts
(list
 (position-token 'VOID (position 1 #f #f) (position 5 #f #f))
 (position-token (token 'IDENTIFIER "main") (position 6 #f #f) (position 10 #f #f))
 (position-token 'LEFT_PAREN (position 10 #f #f) (position 11 #f #f))
 (position-token 'RIGHT_PAREN (position 11 #f #f) (position 12 #f #f))
 (position-token 'LEFT_BRACE (position 12 #f #f) (position 13 #f #f)))
Time taken: 2.789306640625 ms
>
```

Archivo de ejemplo 5:

C/C++

```c
int main() {
    int hours = 40;

    switch (hours) {
    case 50:
        printf("Overtime");
        break;
    case 40:
        printf("Full time");
        break;
    default:
        printf("Part time");
    }

    while (hours < 45) {
        printf("Hours");
        hours++;
    }

    do {
        printf("Hours");
        hours++;
    } while (hours < 50);

    if (hours >= 50) {
        printf("Hours are greater than or equal to 50");
    }

    int count = 0;
    while (count < 3) {
        printf("Count");
        count++;
    }

    for (int i = 0; i < 3; i++) {
        printf("i");
```

```c
}

int a = 0;
while (a < 5) {
    if (a == 2) {
        a++;
        continue;
    }
    printf("a");
    a++;
}

int b = 0;
do {
    if (b == 1) {
        b++;
        continue;
    }
    printf("b");
    b++;
} while (b < 5);

int x = 0;
while (x < 6) {
    printf("x");
    x++;
}

for (int y = 0; y < 6; y++) {
    printf("y");
}

int z = 0;
while (z < 7) {
    printf("z");
    z++;
}

for (int m = 0; m < 7; m++) {
    printf("m");
}

int n = 0;
while (n < 8) {
    printf("n");
    n++;
}
```

```
    for (int p = 0; p < 8; p++) {
        printf("p");
    }

    int q = 0;
    while (q < 9) {
        printf("q");
        q++;
    }

    for (int r = 0; r < 9; r++) {
        printf("r");
    }

    int s = 0;
    while (s < 10) {
        printf("s");
        s++;
    }

    int t = 0;
    while (t < 10) {
        printf("t");
        t++;
    }

    for (int u = 0; u < 10; u++) {
        printf("u");
    }

    int v = 0;
    while (v < 11) {
        printf("v");
        v++;
    }

    for (int w = 0; w < 11; w++) {
        printf("w");
    }

    return 0;
}
```

Tiempo de ejecución

```
Welcome to DrRacket, version 8.12 [cs].
Language: racket, with debugging; memory limit: 128 MB.
184 shift/reduce conflicts
(list
 (position-token 'VOID (position 1 #f #f) (position 5 #f #f))
 (position-token (token 'IDENTIFIER "main") (position 6 #f #f) (position 10 #f #f))
 (position-token 'LEFT_PAREN (position 10 #f #f) (position 11 #f #f))
 (position-token 'RIGHT_PAREN (position 11 #f #f) (position 12 #f #f))
 (position-token 'LEFT_BRACE (position 12 #f #f) (position 13 #f #f)))
Time taken: 2.979736328125 ms
>
```

9) Determinar la complejidad del analizador sintáctico. Con screenshots del sistema mostrando ejemplos de funcionamiento.

La complejidad del algoritmo de análisis sintáctico es de O(N), donde N representa el número de tokens en la entrada. Esta complejidad se justifica por el hecho de que el analizador recorre cada token una sola vez durante el proceso de análisis.

1. Lexical Analysis (Análisis Léxico)

La fase de análisis léxico convierte la secuencia de caracteres en una secuencia de tokens. Esta fase tiene una complejidad de O(n), donde “n” es el número de caracteres en el archivo fuente. Esto se debe a que el lexer necesita escanear cada carácter al menos una vez para identificar tokens.

```
(require "lexer.rkt")


(define src-code (open-input-string src-code-string))
(port-count-lines! src-code)
(define result (with-handlers ([exn:fail? (lambda (e) 'parse-error)])
                 (the-parser (λ () (lex src-code)))))
```

2. Syntactic Analysis (Análisis Sintáctico)

El analizador sintáctico se basa en parser-tools/yacc, que implementa un analizador LALR(1). La complejidad de un analizador LALR(1) es O(n) en términos del número de tokens, debido a que procesa cada token una sola vez en una pasada.

```
(require parser-tools/yacc)
(define (increment x)
  (cond
    [(number? x) (+ x 1)]
    [(symbol? x) (list 'post-increment x)]))

(define (decrement x)
  (cond
    [(number? x) (- x 1)]
    [(symbol? x) (list 'post-decrement x)]))

(define error-handler
  (lambda (tok-ok? tok-name tok-value start-pos end-pos)
    (void)))  ; No hacer nada, solo retornar un valor vacío

(define the-parser
  (parser [start c-program]
          [end EOF]
          [src-pos]
          [error error-handler]
          [tokens
           literal-tokens
           identifier-tokens
           keyword-tokens
           type-tokens
           expression-operator-tokens
           term-operator-tokens
           punctuation-tokens]
          [grammar
```

3. Highlighting HTML (Resaltado HTML)

Las funciones como surroundRegexp, sep, y resaltar2 manipulan y transforman la representación en cadena del código fuente. Estas operaciones tienen una complejidad lineal O(n), ya que procesan cada parte del código fuente una vez.

```
;; Function to generate highlighted HTML
(define (surround1 s1 c)
  (string-append "<" c ">" s1))

(define (surround2 s1 c)
  (string-append "<" c ">" s1 "</" c ">"))

(define (surround3 s1 c)
  (string-append s1 "</" c ">"))

(define (sep str)
  (regexp-split #rx"(?<=\\]|\\[|(\r\n)|[+-/()<>{}=;: ])|(?=\\]|\\[|(\r\n)|[+-/()<>{}=;: ])" str))

(define (surroundRegexp s1)
  (cond
    [(regexp-match #rx"[0-9]+|^int$" s1) (set! s1 (string-append "<span style=\"color: #0077cc;\">" s1 "</span>"))]
    [(regexp-match #rx"\\." s1) (set! s1 (string-append "<span style=\"color: #0077cc;\">" s1 "</span>"))]
    [(regexp-match #rx"[><*=/+-]" s1) (set! s1 (string-append "<span style=\"color: #ff7f00;\">" s1 "</span>"))]
    [(regexp-match #rx"^else$|^if$" s1) (set! s1 (string-append "<span style=\"color: #a020f0;\">" s1 "</span>"))]
    [(regexp-match #rx"^for$|^while$|^switch$|^do$" s1) (set! s1 (string-append "<span style=\"color: #a020f0;\">" s1 "</span>"))]
    [(regexp-match #rx"\".*\"" s1) (set! s1 (string-append "<span style=\"color: #00a86b;\">" s1 "</span>"))]
    [(regexp-match #rx"(?<=\")(?=.)" s1) (set! s1 (string-append "<span style=\"color: #00a86b;\">" s1))]
    [(regexp-match #rx"(?<=.)(?=\")" s1) (set! s1 (string-append s1 "</span>"))]
    [(regexp-match #rx"(?<=')(?=.)" s1) (set! s1 (string-append "<span style=\"color: #00a86b;\">" s1))]
    [(regexp-match #rx"(?<=.)(?=')" s1) (set! s1 (string-append s1 "</span>"))]
    [(regexp-match #rx"'.*'" s1) (set! s1 (string-append "<span style=\"color: #00a86b;\">" s1 "</span>"))]
    [(regexp-match #rx"\\]|\\[|[(]|[)]|[{]|[}]" s1) (set! s1 (string-append "<span style=\"color: #FE591A;\">" s1 "</span>"))]
    [(regexp-match #rx"^default$|^case$|^break$|^print$" s1) (set! s1 (string-append "<span style=\"color: #F93B08;\">" s1 "</span>"))]
    [(regexp-match #rx"^printf$" s1) (set! s1 (string-append "<span style=\"color: #3E7900;\">" s1 "</span>"))]
    [(regexp-match #rx"[,]" s1) (set! s1 (string-append "<span style=\"color: #666666;\">" s1 "</span>"))]
    [(regexp-match #rx" " s1) (set! s1 " ")]
    [(regexp-match #rx"(\r\n|\n)" s1) (set! s1 "<br>")]  ; Manejar tanto \r\n como \n
    [else s1])  ; Devolver s1 sin cambios si no coincide con ninguna regla
  s1)



(define (resaltar2 x)
  (set! x (sep x))
  (set! x (map (lambda (lst) (surroundRegexp lst)) x))
  (set! x (string-join x ""))
  (set! x (string-append "<pre style=\"background-color: #f4f4f4; padding: 10px;\">" x "</pre>"))
  x)
```

4. File Operations (Operaciones de Archivos)

Leer el archivo de entrada (file->string) y escribir el archivo de salida (call-with-output-file) son operaciones lineales O(n) respecto al tamaño del archivo.

```
(define (append-parse-result highlighted-code result)
  (define status (if (eq? result 'parse-error) "Sintaxis Incorrecta" "Sintaxis Correcta"))
  (define status-html
    (string-append
     "<div style=\"background-color: black; color: white; padding: 10px; margin-top: 20px; font-weight: bold;\">"
     status
     "</div>"))

  (define full-html
    (string-append
     "<!DOCTYPE html>\n<html lang=\"es\">\n<head>\n<meta charset=\"UTF-8\">\n<title>Código Resaltado</title>\n</head>\n<body>\n"
     highlighted-code
     status-html
     "\n</body>\n</html>"))

  (call-with-output-file parsed-code-file-path
    (lambda (port)
      (display full-html port))
    #:exists 'replace))

;; Main function to parse the source code and generate highlighted HTML
(define (main)
  (define src-code-string (file->string source-code-file-path))
  (define highlighted-code (resultado src-code-string))

  (define src-code (open-input-string src-code-string))
  (port-count-lines! src-code)
  (define result (with-handlers ([exn:fail? (lambda (e) 'parse-error)])
                   (the-parser (λ () (lex src-code)))))

  (append-parse-result highlighted-code result))

(main)
```

Para ilustrar el funcionamiento del analizador, a continuación se presentan capturas de pantalla del sistema en acción:

Ejemplo 1: Análisis de una expresión aritmética

```
[expr
 [(IDENTIFIER ASSIGNMENT expr) (list 'assign $1 $3)]
 [(expr PLUS expr) (list '+ $1 $3)]
 [(expr MINUS expr) (list '- $1 $3)]
 [(expr MULTIPLY expr) (list '* $1 $3)]
 [(expr DIVIDE expr) (list '/ $1 $3)]
 [(expr EQUAL expr) (list '== $1 $3)]
 [(expr NOT_EQUAL expr) (list '!= $1 $3)]
 [(expr LESS_THAN expr) (list '< $1 $3)]
 [(expr GREATER_THAN expr) (list '> $1 $3)]
 [(expr LESS_THAN_OR_EQUAL expr) (list '<= $1 $3)]
 [(expr GREATER_THAN_OR_EQUAL expr) (list '>= $1 $3)]
 [(expr AND expr) (list '&& $1 $3)]
 [(expr OR expr) (list '|| $1 $3)]
 [(expr INCREMENT) (increment $1)]
 [(expr DECREMENT) (decrement $1)]
 [(IDENTIFIER) $1]
 [(NUMBER) $1]
 [(LEFT_PAREN expr RIGHT_PAREN) $2]
 [(STRING) $1]]
```

Ejemplo 2: Análisis de una estructura de control

```
[default-statement
 [(DEFAULT COLON statements break-statement) (list 'default $3 $4)]
 [(DEFAULT COLON statements) (list 'default $3)]]
[break-statement [(BREAK SEMICOLON) 'break]]
[return-statement [(RETURN expr SEMICOLON) (list 'return $2)] [(RETURN SEMICOLON)
[continue-statement [(CONTINUE SEMICOLON) 'continue]]
[for-init [(declaration) $1] [(expr) $1] [() #f]]
```

Ejemplo 3: Análisis de una declaración de función

```
[default-statement
 [(DEFAULT COLON statements break-statement) (list 'default $3 $4)]
 [(DEFAULT COLON statements) (list 'default $3)]]
[break-statement [(BREAK SEMICOLON) 'break]]
[return-statement [(RETURN expr SEMICOLON) (list 'return $2)] [(RETURN SEMICOLON)
[continue-statement [(CONTINUE SEMICOLON) 'continue]]
[for-init [(declaration) $1] [(expr) $1] [() #f]]
```

Conclusión

Estas capturas de pantalla y descripciones demuestran cómo el analizador sintáctico maneja diferentes tipos de entradas de manera eficiente, siempre respetando la complejidad de O(N). Este rendimiento se logra gracias a la capacidad del algoritmo para procesar cada

token una sola vez, sin necesidad de retrocesos ni reanálisis. La complejidad total del analizador sintáctico es O(n), donde “n” es el tamaño del archivo de entrada, ya que todas las fases principales del análisis (léxico, sintáctico, resaltado HTML y operaciones de archivos) tienen una complejidad lineal respecto al tamaño del archivo.