Universidad de Ingeniería y Tecnología

CIENCIA DE LA COMPUTACIÓN

COMPILADORES

ANALIZADOR PREDICTIVO PARA PROCESO DE TEXTILES

Autores:

Mayra Díaz Tramontana 201910147

Joaquín Elías Ramírez Gutiérrez 201910277

25 de marzo de 2021

ÍNDICE GENERAL

1. Introducción

En el marco de la innovación dentro de una empresa textil, esta decide implementar un sistema automatizado de diseño de flujos de producción optimizados. Para la correcta ejecución del sistema, este cuenta con reglas que deben ser cumplidas para garantizar un ahorro adecuado de tiempo y recursos. Con el fin de lograrlo, la compañía textil propone la construcción de un compilador que se encargue del manejo del lenguaje de producción y de la determinación de rutas de materia prima adecuadas en una secuencia de operaciones.

La empresa textil solicita que la producción sea rápida y a bajo costo, por lo que cada flujo de producción debe atravesar máximo 5 etapas. Para garantizar la integridad del programa, se construirá un lexer y un parser.

El flujo de producción textil aceptado está definido por la siguiente gramática. Donde cada no terminal representa una etapa de producción y el tipo de esta.

A: Almacén
Sh: Tejido horizontal
Sv: Tejido vertical
N: Teñido
Ac: Acabado
C: Confección

Figura 1.1: Etapas

S → ‘A’ A
A → ‘Sh1’ SH1 | ‘Sh2’ SH2 | ‘Sh3’ SH3 | ‘Sh4’ SH4 | ‘Sh5’ SH5
SH1 → ‘Sv1’ SV1 | ‘Sv2’ SV2 | ‘Sv3’ SV3
SH2 → ‘Sv1’ SV1 | ‘Sv2’ SV2
SH3 → ‘Sv1’ SV1 | ‘Sv2’ SV2 | ‘Sv3’ SV3
SH4 → ‘Sv3’ SV3 | ‘Sv4’ SV4 | ‘Sv5’ SV5
SH5 → ‘Sv3’ SV3 | ‘Sv5’ SV5
SV1 → ‘N1’ N1 | ‘N2’ N2 | ‘N3’ N3
SV2 → ‘N1’ N1 | ‘N2’ N2 | ‘N3’ N3
SV3 → ‘N4’ N4 | ‘N5’ N5
SV4 → ‘N4’ N4 | ‘N5’ N5
SV5 → ‘N5’ N5
N1 → ‘Ac1’ AC1 | ‘Ac2’ AC2
N2 → ‘Ac1’ AC1 | ‘Ac2’ AC2 | ‘Ac3’ AC3
N3 → ‘Ac2’ AC2 | ‘Ac3’ AC3 | ‘Ac4’ AC4
N4 → ‘Ac2’ AC2 | ‘Ac3’ AC3 | ‘Ac4’ AC4
N5 → ‘Ac4’ AC4
AC1 → ‘C1’ C1 | ‘C2’ C2 | ‘C3’ C3
AC2 → ‘C1’ C1 | ‘C2’ C2 | ‘C3’ C3
AC3 → ‘C1’ C1 | ‘C2’ C2 | ‘C3’ C3
AC4 → ‘C4’ C4
C1 → ‘C2’ C2
C2 → ‘C4’ C4
C3 → ‘A’
C4 → ‘A’

Figura 1.2: Gramática

2. Método y Desarrollo

El método utilizado consiste en un parser LL(1).

2.1. Gramática

La gramática original fue ligeramente cambiada para que sea LL(1) y contenga terminales. La gramática en sí no asegura que cada flujo atraviese como máximo 5 etapas. El único caso donde esto ocurre es cuando pasa por confección$_3$, de lo contrario atraviesa más etapas. Por lo tanto, esta condición es verificada en el código.

Cadena: A Sh1 Sv2 N3 Ac2 C2 C4 A

Derivación mi:

S $\rightarrow$ 'A' A

A $\rightarrow$ 'Sh1' SH1

SH1 $\rightarrow$ 'Sh1' SV2

SV2 $\rightarrow$ 'N3' N3

N3 $\rightarrow$ 'Ac2' AC2

Ac2 $\rightarrow$ 'C2' C2

C2 $\rightarrow$ 'C4' C4

C4 $\rightarrow$ 'A'

Figura 2.1: Ejemplo de más de 5 etapas

2.1.1. Conjuntos de Primeros y Siguientes

Los conjuntos de primeros y siguientes son creados en el programa mismo. Dichos conjuntos son:

S = {'A'}	
A = {'Sh1', 'Sh2', 'Sh3', 'Sh4', 'Sh5'}	S = {$}
	A = {$}
SH1 = {'Sv1', 'Sv2', 'Sv3'}	SH1 = {$}
SH2 = {'Sv1', 'Sv2'}	SH2 = {$}
SH3 = {'Sv1', 'Sv2', 'Sv3'}	SH3 = {$}
SH4 = {'Sv3', 'Sv4', 'Sv5'}	SH4 = {$}
SH5 = {'Sv3', 'Sv5'}	SH5 = {$}
SV1 = {'N1', 'N2', 'N3'}	SV1 = {$}
SV2 = {'N1', 'N2', 'N3'}	SV2 = {$}
SV3 = {'N4', 'N5'}	SV3 = {$}
SV4 = {'N4', 'N5'}	SV4 = {$}
SV5 = {'N5'}	SV5 = {$}
N1 = {'Ac1', 'Ac2'}	N1 = {$}
N2 = {'Ac1', 'Ac2', 'Ac3'}	N2 = {$}
N3 = {'Ac2', 'Ac3,' Ac4'}	N3 = {$}
N4 = {'Ac2', 'Ac3', 'Ac4'}	N4 = {$}
N5 = {'Ac4'}	N5 = {$}
AC1 = {'C1', 'C2', 'C3'}	AC1 = {$}
AC2 = {'C1', 'C2', 'C3'}	AC2 = {$}
AC3 = {'C1', 'C2', 'C3'}	AC3 = {$}
AC4 = {'C4'}	AC4 = {$}
C1 = {'C2'}	C1 = {$}
C2 = {'C4'}	C2 = {$}
C3 = {'A'}	C3 = {$}
C4 = {'A'}	C4 = {$}
(a) Primeros	(b) Siguientes

Figura 2.2: Conjuntos Primeros y Siguientes

2.1.2. Tabla M[n,t] - LL(1)

	$	a	sh1	sh2	sh3	sh4	sh5	sv1	sv2	sv3	sv4	sv5	n1	n2	n3	n4	n5	ac1	ac2	ac3	ac4	c1	c2	c3	c4
S		S → a A																							
A			A → sh1 SH1	A → sh2 SH2	A → sh3 SH3	A → sh4 SH4	A → sh5 SH5																		
SH1								SH1 → sv1 SV1	SH1 → sv2 SV2	SH1 → sv3 SV3															
SH2								SH2 → sv1 SV1	SH2 → sv2 SV2																
SH3								SH3 → sv1 SV1	SH3 → sv2 SV2	SH3 → sv3 SV3															
SH4										SH4 → sv3 SV3	SH4 → sv4 SV4	SH4 → sv5 SV5													
SH5										SH5 → sv3 SV3		SH5 → sv5 SV5													
SV1													SV1 → n1 N1	SV1 → n2 N2	SV1 → n3 N3										
SV2													SV2 → n1 N1	SV2 → n2 N2	SV2 → n3 N3										
SV3																SV3 → n4 N4	SV3 → n5 N5								
SV4																SV4 → n4 N4	SV4 → n5 N5								
SV5																	SV5 → n5 N5								
N1																		N1 → ac1 AC1	N1 → ac2 AC2						
N2																		N2 → ac1 AC1	N2 → ac2 AC2	N2 → ac3 AC3					
N3																			N3 → ac2 AC2	N3 → ac3 AC3	N3 → ac4 AC4				
N4																			N4 → ac2 AC2	N4 → ac3 AC3	N4 → ac4 AC4				
N5																					N5 → ac4 AC4				
AC1																						AC1 → c1 C1	AC1 → c2 C2	AC1 → c3 C3	
AC2																						AC2 → c1 C1	AC2 → c2 C2	AC2 → c3 C3	
AC3																						AC3 → c1 C1	AC3 → c2 C2	AC3 → c3 C3	
AC4																									AC4 → c4 C4
C1																							C1 → c2 C2		
C2																									C2 → c4 C4
C3		C3 → a																							
C4		C4 → a																							

2.2. Estructura

El método de implementación del compilador se hará con C++. Para hacer una buena implementación se dividieron las funcionalidades en la siguiente estructura de clases:

1. **Lexer:**
   La clase lexer se encarga del análisis léxico en base a un string. Para que esto sea posible utiliza una estructura **TOKEN**, la cual almacena los datos más importantes de un token.
2. **CFGHandler:**
   El handler se encarga de todas las operaciones respecto a la gramática. Esta clase lee la gramática y la transforma para que pueda ser utilizada. Además, crea los conjuntos de los primeros y siguientes.
3. **Parser:**
   El parser crea la tabla LL(1), haciendo uso del handler para la gramática, y la utiliza para analizar cadenas haciendo uso del lexer.
4. **Textiles:**
   Es una función que crea un pequeño menú interactivo en consola para el usuario que desee información sobre el flujo de producción o quiera verificar un flujo determinado.

2.3. Implementación

A continuación se explicarán las funciones principales de cada parte del código.

2.3.1. Lexer

TOKEN

El token es una estructura que almacena la información necesaria para el análisis:

- **type:** representa la etapa del flujo de producción, además, cuenta con 3 tipos adicionales:
  - *ERROR:* errores léxicos encontrados al leer el input, estos pueden ser ignorados para analizar la cadena y que pueda ser aceptada con errores.

  - *FATAL_ERROR:* errores por condiciones externas que ocasionan un rechazo automático de la cadena. Por ejemplo, para controlar el máximo de etapas.
  - *$:* representa el fin de la cadena, este token es añadido al final.

- **id:** terminal.
- **description:** descripción del terminal, en este caso se utiliza para almacenar el nombre de la etapa y la descripción de los errores.

```
struct TOKEN {
    enum Type {
        A, SH, SV, N, AC, C, ERROR, $, FATAL_ERROR
    };
    Type type;
    std::string id;
    std::string description;

    TOKEN(Type type, std::string id);

    TOKEN(Type type, std::string id, std::string e);

    friend std::ostream &operator<<(std::ostream &os, const TOKEN &
   token);

private:
    std::string get_description() const;
};
```

Constructor

En el constructor se analiza toda la cadena, leyendo el input caracter por caracter. Cuando encuentra un caracter que inicia un terminal, analiza este y lo deriva en un TOKEN. Si logra reconocer dicho terminal, coloca el type y id correspondientes y llama a la funcion get_-description() para colocar la descripción. Si el TOKEN no es el terminal lo deriva en un TOKEN de error con descripción Fase inválida. Al encontrar caracteres desconocidos los deriva en un TOKEN de error con descripción Caracter inválido.
Cada vez que incluye una fase en el vector de tokens aumenta el contador. Si al final del análisis este es mayor a 5, se inserta al inicio de la cadena un TOKEN de error fatal para que la cadena sea automáticamente rechazada.

```
Lexer::Lexer(std::string input) {
    int i = 0, phases = 0;
```

```
 while (i < input.size()) {
     std::string id;
     if (input[i] == ' ')
         ++i;
     if (input[i] == 'A') {
         if (input[++i] == 'c') {
             bool isd = std::isdigit(input[++i]);
             if (isd && input[i] - '0' < 5 && input[i] - '0' > 0)
{
                 tokens.push_back(TOKEN{TOKEN::Type::AC, "ac" +
std::string(1, input[i++])});
                 ++phases;
             } else {
                 id = isd ? "Ac" + std::string(1, input[i++]) : "
Ac";
                 tokens.push_back(TOKEN{TOKEN::Type::ERROR, id, "
Invalid phase"});
             }
         } else {
             tokens.push_back(TOKEN{TOKEN::Type::A, "a"});
         }
     } else if (input[i] == 'S') {
         if (input[++i] == 'h' || input[i] == 'v') {
             auto type = input[i] == 'h' ? TOKEN::Type::SH : TOKEN
::Type::SV;
             id = type == TOKEN::Type::SH ? "sh" + std::string(1,
input[++i]) : "sv" + std::string(1, input[++i]);
             bool isd = std::isdigit(input[i]);
             if (isd && input[i] - '0' < 6 && input[i++] - '0' >
0) {
                 tokens.push_back(TOKEN{type, id});
                 ++phases;
             } else {
                 id = isd ? "S" + std::string(1, input[i - 1]) +
std::string(1, input[i++]) : "S" + std::string(1,

                                                 input[-1]);
                 tokens.push_back(TOKEN{TOKEN::Type::ERROR, id, "
Invalid phase"});
             }
         } else
```

```
                tokens.push_back(TOKEN{TOKEN::Type::ERROR, "S", "
    Invalid phase"});
            } else if (input[i] == 'N') {
                bool isd = std::isdigit(input[++i]);
                if (isd && input[i] - '0' < 6 && input[i] - '0' > 0) {
                    tokens.push_back(TOKEN{TOKEN::Type::N, "n" + std::
    string(1, input[i++])});
                    ++phases;
                } else {
                    id = isd ? "N" + std::string(1, input[i++]) : "N";
                    tokens.push_back(TOKEN{TOKEN::Type::ERROR, id, "
    Invalid phase"});
                }
            } else if (input[i] == 'C') {
                bool isd = std::isdigit(input[++i]);
                if (isd && input[i] - '0' < 6 && input[i] - '0' > 0) {
                    tokens.push_back(TOKEN{TOKEN::Type::C, "c" + std::
    string(1, input[i++])});
                    ++phases;
                } else {
                    tokens.push_back(TOKEN{TOKEN::Type::ERROR, "C", "
    Invalid phase"});
                }
            } else {
                tokens.push_back(TOKEN{TOKEN::Type::ERROR, std::string(1,
     input[i++]), "Invalid character"});
            }
        }
        if (phases > 5) {
            tokens.insert(tokens.begin(), TOKEN{TOKEN::Type::FATAL_ERROR,
     "ERROR", "More than 5 phases."});
        }
}
```

2.3.2. CFGHandler

Constructor

Condiciones de la gramática a recibir:

1. Todos los terminales y no terminales deben estar separados por un único espacio en blanco.
2. Cada conjunto de producciones de un no terminal debe estar separado por el caracter '$'.
3. Si una regla tiene varias producciones, estas deben estar en una sola regla y separadas por el caracter '|'.
4. Al escribir una regla se debe poner el no terminal al inicio y el caracter *':'* al costado sin ninguna separación.

Además, el handler recibe un string con los terminales y otro con los no terminales, cada uno separado por un espacio.

grammari: "S: n NUM | n $ NUM: 1 | 2 | 3 | 4"
terminalsi: "n 1 2 3 4"
non_terminalsi: "S NUM"

Figura 2.3: Ejemplo de input para el CFGHandler

```
CFGHandler::CFGHandler(std::string grammari, std::string terminalsi,
   std::string non_terminalsi,
                       std::string starti) : initial(std::move(starti
   )) {
    read_grammar(std::move(grammari));
    read_elements(std::move(terminalsi));
```

Primeros y Siguientes

Pasos para crear los conjuntos:

1. Inicializar los conjuntos de primeros y siguientes para todos los terminales. Cada conjunto es de tipo *Map<no_terminal, set de strings (representan los terminales)>*.

```
void CFGHandler::generate_firsts_follows() {
    for (const auto &nterminal: non_terminals)
        Firsts.insert({nterminal, SetS_t()});
    for (const auto &nterminal: non_terminals)
        Follows.insert({nterminal, SetS_t()});
```

2. Se inserta el caracter '$' en los Siguientes de la regla inicial.

```
Follows[initial].insert("$");
```

3. Creación de los conjuntos Primeros. Para esto se utiliza un contador para contabilizar los cambios realizados. Se itera sobre las reglas hasta que no se realicen cambios, lo cual nos indica que no se han añadido nuevos primeros y hemos acabado.
Las iteraciones consisten en revisar el primer elemento de cada producción de los no_-terminales. Si este es un terminal y no está dentro de sus primeros, lo añade. De lo contrario, itera en todos los primeros de dicho no_terminal y si no se encuentran en sus primeros lo añade.

```
int changes = 1;
while (changes != 0) {
    changes = 0;
    for (const auto &rule: grammar) {
        for (auto production: rule.second) {
            if (terminals.find(production[0]) != terminals.
end()) {
                if (Firsts[rule.first].find(production[0])
== Firsts[rule.first].end()) {
                    Firsts[rule.first].insert(production[0])
;
                    ++changes;
                }
            } else {
                for (const auto &first: Firsts[production
[0]]) {
                    if (Firsts[rule.first].find(first) ==
Firsts[rule.first].end()) {
                        Firsts[rule.first].insert(first);
                        ++changes;
                    }
                }
            }
        }
    }
}
```

4. Creación de los conjuntos Siguientes. Para esto se utiliza un contador para contabilizar los cambios realizados. Se itera sobre las reglas hasta que no se realicen cambios, lo cual nos indica que no se han añadido nuevos primeros y hemos acabado.
Las iteraciones consisten en revisar los primeros $n-1$ elementos de cada producción de los no_terminales. Si este es un terminal, continúa con el siguiente elemento. De

lo contrario, verifica si el siguiente el elemento es un terminal. Ssi esto es verdadero lo añade a su conjunto Siguientes, de lo contrario itera en todos los primeros de dicho no_terminal y si no se encuentran en sus siguientes lo añade.
Para terminar, verifica si el elemento n de dicha producción es un no terminal. Si esto es verdadero añade todos los siguientes, que no pertenezcan ya a su conjunto Siguientes, del no terminal que lo produce.

```
 changes = 1;
 while (changes != 0) {
     changes = 0;
     for (const auto &rule: grammar) {
         for (auto production: rule.second) {
             for (int i = 0; i < production.size() - 1; ++i)
{
                 if (non_terminals.find(production[i]) !=
non_terminals.end()) {
                     auto next = production[i + 1];
                     auto current = production[i];
                     if (terminals.find(next) != terminals.
end()) {
                         if (Follows[current].find(next) ==
Follows[current].end()) {
                             Follows[current].insert(next);
                             changes++;
                         }
                     } else {
                         for (const auto &t: Firsts[next]) {
                             if (Follows[current].find(t) ==
Follows[current].end()) {
                                 Follows[current].insert(t);
                                 changes++;
                             }
                         }
                     }
                 }
                 if (non_terminals.find(production[production.
size() - 1]) != non_terminals.end()) {
                     auto last = production[production.size() -
1];
                     if (non_terminals.find(last) !=
non_terminals.end()) {
                         for (const auto &t: Follows[rule.first])
```

```
    {
                            if (Follows[last].find(t) == Follows
   [last].end()) {
                                Follows[last].insert(t);
                                changes++;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Cabe resaltar que el handler no maneja gramáticas con símbolos ε.

2.3.3. Parser

Parser

El Parser es generado a partir de la gramática brindada, especificando los terminales y no terminales, a la vez del punto de partida de dicha gramática.

```
Parser::Parser(string_t grammari, string_t terminalsi, string_t
   non_terminalsi, string_t start) :
        handler(std::move(grammari), std::move(terminalsi), std::move
   (non_terminalsi), std::move(start)) {
    initialize();
    fill_table_MNT();
}
```

La función *initialize* se encarga de inicializar la tabla LL(1), asignando índices a cada terminal y no terminal, y viceversa.
Para llenar la tabla M[n,t], analizamos todas las reglas de la gramática $A \rightarrow \alpha$. En cada una de estas, prestamos atención al primer token α y revisamos los Primeros(α). Para cada token *a* dentro de este conjunto, se añade la regla $A \rightarrow \alpha$ en la celda M[A,a].

Tabla LL(1)

```
void Parser::fill_table_MNT() {
    for (auto nter : handler.grammar) {
        for (auto production : nter.second) {//cada regla, '|', rule
   es un vector
            //rule[0] es al q le revisare los primeros
            auto first_conj = get_First(production[0]);
            for (const auto &first : first_conj) {//llegada es un
   token
                //agregar en M[A,a] la regla A --> alpha
                table[nter_int[nter.first]][ter_int[first]] =
   production;
            }
        }
    }
}
```

Análisis predictivo

Una vez la tabla está llena, el análisis LL(1) de una determinada cadena se puede realizar. Para lograr esto, nos hemos guiado del pseudocódigo 1. Primero se toman en cuenta los resultados obtenidos en base al *lexer*, y se procede a ejecutar el algoritmo. Es importante denotar que para el manejo de errores cuando un no temrinal A está en el tope de la pila y el token del input no está en Primeros(A), tomamos en consideración los métodos *explorar* y *extraer*. El primero de ellos se utiliza si es que el token de entrada es $ o si es que está en los

Siguientes(A), mientras que el segundo se usa en caso contrario.

Algorithm 1: Pseudocódigo - Análisis LL(1)

definir ip apuntando al inicio de w
definir X en el tope de la pila
while $X \neq \$$ **do**
  if $X = a$ **then**
    pop en la pila
    avanza ip
  else if *X es terminal* **then**
    error
  else if *M[X,a] es error* **then**
    error
  else if *M[X,a]* $= X \rightarrow Y_1 Y_2 ... Y_k$ **then**
    extraer la producción $X \rightarrow Y_1 Y_2 ... Y_k$
    pop en la pila
    push $X \rightarrow Y_k Y_{k-1} ... Y_1$ en la pila, con Y_1 en el tope de la pila
  definir X en el tope de la pila
end

```
result_t Parser::analyze_lexeme(string_t input) {
    Lexer lexer(std::move(input));
    auto tokens = lexer.get_tokens();
    std::stack<string_t> stack;
    string_t X, a;
    bool fatal_error = tokens[0].type == TOKEN::Type::FATAL_ERROR,
   lex_error = false, syntax_error= false;
    int ip = fatal_error ? 1 : 0;
    tokens.emplace_back(TOKEN::Type::$, "$", "ACCEPTED");
    stack.push("$");
    stack.push(handler.initial);
    X = stack.top();

    std::cout << "STACK\t\t\t\t||\t\t\t\tINPUT\t\t\t\t||\t\t\t\
   tACTION\n";
    std::cout << "
   ------------------------------------------------------------------------
   \n";

    while (X != "$") {
```

```
        print_stack(stack);
        std::cout << "\t\t\t\t";
        print_input(tokens, ip);
        std::cout << "\t\t\t\t\t\t\t\t";
        a = tokens[ip].id;
        if (tokens[ip].type == TOKEN::Type::ERROR) {
            std::cout << "lex_error ( " << tokens[ip].description <<
" )\n";
            lex_error = true;
            ++ip;
        } else if (tokens[ip].type == TOKEN::Type::FATAL_ERROR) {
            std::cout << "error ( " << tokens[ip].description << " )\
n";
            fatal_error = true;
            ++ip;
        } else if (X == a) {
            stack.pop();
            ip++;
            std::cout << "matching ( " << a << " )\n";
        } else if (table[nter_int[X]][ter_int[a]].empty()) {
            if (a == "$" || handler.Follows[X].find(a) != handler.
Follows[X].end()) {
                stack.pop();
                syntax_error = true;
                std::cout << "extraer ( error )\n";
            } else {
                ip++;
                syntax_error = true;
                std::cout << "explorar ( error )\n";
            }
        } else if (!table[nter_int[X]][ter_int[a]].empty()) {
            auto rule = table[nter_int[X]][ter_int[a]];
            stack.pop();
            for (int i = rule.size() - 1; i >= 0; --i)
                stack.push(rule[i]);
            std::cout << X << " --> ";
            for (int i = 0; i < rule.size(); ++i)
                std::cout << rule[i] << " ";
            std::cout << "\n";
        }
        X = stack.top();
    }
```

```
    std::cout << "$\t\t\t\t\t$\n";
    string_t acceptance;
    if (fatal_error && syntax_error)
        acceptance = "REJECTED: " + tokens[0].description + ", Syntax
    error";
    else if (fatal_error)
        acceptance = "REJECTED: " + tokens[0].description;
    else if (syntax_error)
        acceptance = "REJECTED: Syntax error";
    else {
        std::string w;
        for (int i = 0; i < tokens.size()-1; ++i) {
            if (tokens[i].type != TOKEN::Type::ERROR && tokens[i].
   type != TOKEN::Type::FATAL_ERROR)
                w += tokens[i].id + " ";
        }
        acceptance = lex_error ? "ACCEPTED WITH ERRORS, Lexical Error
    \n\tACCEPTED VERSION:\t" + w : "ACCEPTED";
    }
    if (ip != tokens.size() - 1)
        acceptance += ", unnecessary tokens";
    std::cout << "\t" << acceptance << "\n\n";
    return result_t{fatal_error, acceptance};
}
```

2.3.4. Textiles

Programa

El programa crea el parser, pasándole la gramática ya definida. Y para realizar operaciones utiliza dicho parser. Nótese que parte del código fue obviado por la extensión del mismo. Las líneas obviadas representas strings que son pasados al constructor del parser.

```
void TextilesFlujoProduccion(){
    std::string grammar_s = "S: a A $ A: sh1 SH1 | sh2 SH2 | sh3 SH3
   | sh4 SH4 | sh5 SH5 $ "
    Parser parser(grammar_s, terminals, non_terminals, "S");
```

Menú interactivo

El menú brinda 4 opciones al usuario:

1. Información sobre la gramática.
2. Ver la tabla LL.
3. Verificar un flujo.
4. Revisar los errores léxicos.

El programa termina cuando el usuario ingresa 0. Si entra a la opción 3 o 4, el programa pide cadenas de flujos hasta que el usuario ingrese -1.

```
    int option = print_menu();
    while (option!=0){
        switch (option) {
            case 1:{
                parser.print_grammar_info();
                break;
            }
            case 2:{
                parser.print_LL_table();
                break;
            }
            case 3: {
                analyze(parser, true, "la verificaci[U+FFFD]n de flujo");
                break;
            }
            case 4:{
                analyze(parser, true, "la verificaci[U+FFFD]n l[U+FFFD]xica");
                break;
            }
            default:
                break;
        }
        option = print_menu();
    }
}
```

2.4. Optimización

2.4.1. Manejo de errores

El programa detecta errores de tipo:

- Léxicos
- Sintácticos
- Externos (restricciones externas a la gramática)

2.4.2. Extraer y explorar

El código realiza extraer y explorar para poder seguir evaluando la cadena, sin embargo arroja un error sintáctico. Esta implementación puede ayudar a más adelante proveer una corección de la cadena.

2.4.3. Errores léxicos y corrección del programa

Al encontrarse con errores léxicos, ignora dichos tokens y evalúa la gramática. Si esta no posee errores sintácticos, el programa provee una corección de la cadena.

2.4.4. Conjuntos de Primeros y Siguientes

El *handler* contiene un algoritmo que se encarga de hallar los conjuntos de primeros y Siguientes de los no terminales y terminales de cualquier gramática dada.

2.4.5. Generalización del código

Para la facilidad de cambios en el código, utilizamos tipos no nativos de estructuras de datos. El *handler* está abierto a aceptar cualquier gramática, siempre y cuando sea especificada en el código. Asimismo, el *parser* está generalizado a cualquier gramática también, siempre y

cuando el usuario implemente un *lexer* para corroborar los errores léxiso (partiendo del código implementado en este proyecto es sencillo).

3. RESULTADOS

La implementación del compilador y sus optimizaciones han sido probadas con distintos flujos de producción (cadenas de entrada) para corroborar el correcto funcionamiento del sistema.

En el caso de la Figura 3.1, la cadena utilizada es aquella brindada como base del proyecto. Se observa que el procedimiento LL(1) es exitoso, haciendo *match* de terminales y reemplazando las producciones en base a la tabla M[n,t], resultando en que sea aceptada por la gramática.

```
ASh1Sv3N5Ac4C4A
STACK              ||            INPUT            ||            ACTION
_____________________________________________________________________________
$ S                a sh1 sv3 n5 ac4 c4 a $                        S --> a A
$ A a              a sh1 sv3 n5 ac4 c4 a $                        matching ( a )
$ A                sh1 sv3 n5 ac4 c4 a $                      A --> sh1 SH1
$ SH1 sh1              sh1 sv3 n5 ac4 c4 a $                      matching ( sh1 )
$ SH1              sv3 n5 ac4 c4 a $                    SH1 --> sv3 SV3
$ SV3 sv3              sv3 n5 ac4 c4 a $                  matching ( sv3 )
$ SV3              n5 ac4 c4 a $                  SV3 --> n5 N5
$ N5 n5                n5 ac4 c4 a $                  matching ( n5 )
$ N5               ac4 c4 a $                 N5 --> ac4 AC4
$ AC4 ac4              ac4 c4 a $                 matching ( ac4 )
$ AC4              c4 a $                 AC4 --> c4 C4
$ C4 c4                c4 a $                 matching ( c4 )
$ C4               a $                    C4 --> a
$ a                a $                    matching ( a )
$                  $
                        ACCEPTED
```

Figura 3.1: ASh1Sv3N5Ac4C4A

Para la cadena ASh1Sv3N5Ac4C4AAAA (Figura 3.2), la cual es una variante de la previamente analizada, el compilador la analiza de una manera particular. Ejecuta el análisis LL(1) correctamente, pero una vez que se hace *match($)*, detecta que aún hay tokens restantes en el input, por lo que la cadena es *aceptada con tokens innecesarios*.

```
ASh1Sv3N5Ac4C4AAAA
STACK               ||              INPUT              ||              ACTION
_____________________________________________________________________________________________
$ S                 a sh1 sv3 n5 ac4 c4 a a a a $                                S --> a A
$ A a               a sh1 sv3 n5 ac4 c4 a a a a $                                matching ( a )
$ A                 sh1 sv3 n5 ac4 c4 a a a a $                                  A --> sh1 SH1
$ SH1 sh1               sh1 sv3 n5 ac4 c4 a a a a $                                  matching ( sh1 )
$ SH1               sv3 n5 ac4 c4 a a a a $                                  SH1 --> sv3 SV3
$ SV3 sv3               sv3 n5 ac4 c4 a a a a $                                  matching ( sv3 )
$ SV3               n5 ac4 c4 a a a a $                                  SV3 --> n5 N5
$ N5 n5                 n5 ac4 c4 a a a a $                                  matching ( n5 )
$ N5                ac4 c4 a a a a $                                 N5 --> ac4 AC4
$ AC4 ac4               ac4 c4 a a a a $                                 matching ( ac4 )
$ AC4               c4 a a a a $                                 AC4 --> c4 C4
$ C4 c4                 c4 a a a a $                                 matching ( c4 )
$ C4                a a a a $                                C4 --> a
$ a                 a a a a $                                matching ( a )
$                   $
                        ACCEPTED, unnecessary tokens
```

Figura 3.2: ASh1Sv3N5Ac4C4AAAA

En la Figura 3.3 se puede ver nuevamente una variante de la cadena base del proyecto, pero presenta un cambio: contiene un token no existente en la gramática. Lo particular del *lexer* es que a pesar de encontrarse con un error, continúa con la lectura de los tokens restantes de la cadena. Una vez acabado este proceso, el *parser* empieza a analizar la secuencia y cuando se percata de la existencia de un token desconocido, en vez de detener la ejecución, genera un *warning*. De esta manera, ignora aquel fallo léxico y continúa con el algoritmo LL(1), y en este caso, la cadena es *aceptada con errores léxicos*.

```
A#Sh1Sv3N5Ac4C4A
STACK               ||              INPUT              ||              ACTION
_____________________________________________________________________________________________
$ S                 a # sh1 sv3 n5 ac4 c4 a $                                    S --> a A
$ A a               a # sh1 sv3 n5 ac4 c4 a $                                    matching ( a )
$ A                 # sh1 sv3 n5 ac4 c4 a $                                      lex_error ( Invalid character )
$ A                 sh1 sv3 n5 ac4 c4 a $                                    A --> sh1 SH1
$ SH1 sh1               sh1 sv3 n5 ac4 c4 a $                                    matching ( sh1 )
$ SH1               sv3 n5 ac4 c4 a $                                    SH1 --> sv3 SV3
$ SV3 sv3               sv3 n5 ac4 c4 a $                                    matching ( sv3 )
$ SV3               n5 ac4 c4 a $                                    SV3 --> n5 N5
$ N5 n5                 n5 ac4 c4 a $                                    matching ( n5 )
$ N5                ac4 c4 a $                                   N5 --> ac4 AC4
$ AC4 ac4               ac4 c4 a $                                   matching ( ac4 )
$ AC4               c4 a $                                   AC4 --> c4 C4
$ C4 c4                 c4 a $                                   matching ( c4 )
$ C4                a $                                  C4 --> a
$ a                 a $                                  matching ( a )
$                   $
    ACCEPTED WITH ERRORS, Lexical Error
    ACCEPTED VERSION:   a sh1 sv3 n5 ac4 c4 a
```

Figura 3.3: A#Sh1Sv3N5Ac4C4A

Se puede apreciar que la cadena de la Figura 3.4 representa otro caso particular del análisis del compilador. En esta ocasión, la cadena contiene más tokens de lo normal, pero el *lexer* y *parser* no detectan ningún error. Sin embargo, a pesar de que sintácticamente debería ser aceptada, por el criterio del proyecto, cada artículo no debe de atravesar más de 5 etapas, y en

este caso viola dicho criterio.

```
A Sh1 Sv1 N1 Ac1 C1 C2 C4 A
STACK                ||              INPUT              ||              ACTION
_____________________________________________________________________________________________
$ S                  a sh1 sv1 n1 ac1 c1 c2 c4 a $                                S --> a A
$ A a                a sh1 sv1 n1 ac1 c1 c2 c4 a $                                matching ( a )
$ A                  sh1 sv1 n1 ac1 c1 c2 c4 a $                                  A --> sh1 SH1
$ SH1 sh1                sh1 sv1 n1 ac1 c1 c2 c4 a $                                  matching ( sh1 )
$ SH1                sv1 n1 ac1 c1 c2 c4 a $                                  SH1 --> sv1 SV1
$ SV1 sv1                sv1 n1 ac1 c1 c2 c4 a $                                  matching ( sv1 )
$ SV1                n1 ac1 c1 c2 c4 a $                                  SV1 --> n1 N1
$ N1 n1                  n1 ac1 c1 c2 c4 a $                                  matching ( n1 )
$ N1                 ac1 c1 c2 c4 a $                                  N1 --> ac1 AC1
$ AC1 ac1                ac1 c1 c2 c4 a $                                  matching ( ac1 )
$ AC1                c1 c2 c4 a $                                  AC1 --> c1 C1
$ C1 c1                  c1 c2 c4 a $                                  matching ( c1 )
$ C1                 c2 c4 a $                                  C1 --> c2 C2
$ C2 c2                  c2 c4 a $                                  matching ( c2 )
$ C2                 c4 a $                                  C2 --> c4 C4
$ C4 c4                  c4 a $                                  matching ( c4 )
$ C4                 a $                                  C4 --> a
$ a                  a $                                  matching ( a )
$                    $
                        REJECTED: More than 5 phases
```

Figura 3.4: ASh1Sv1N1Ac1C1C2C4A

Similar a la cadena anterior, en la Figura 3.5, se viola con la restricción de no exceder 5 etapas por artículo. No obstante, en este caso, la cadena aún así no sería aceptada, pues se presentan errores sintácticos. En el manejo de errores se utilizó la opción de *explorar*, lo que permitió que la secuencia de tokens pueda ser aceptada, pero debe ser rechazada pues la gramática no lo permite.

```
A Sh1 Sv3 N5 Ac4 C2 C4 A
STACK                ||              INPUT              ||              ACTION
_____________________________________________________________________________________________
$ S                  a sh1 sv3 n5 ac4 c2 c4 a $                                S --> a A
$ A a                a sh1 sv3 n5 ac4 c2 c4 a $                                matching ( a )
$ A                  sh1 sv3 n5 ac4 c2 c4 a $                                  A --> sh1 SH1
$ SH1 sh1                sh1 sv3 n5 ac4 c2 c4 a $                                  matching ( sh1 )
$ SH1                sv3 n5 ac4 c2 c4 a $                                  SH1 --> sv3 SV3
$ SV3 sv3                sv3 n5 ac4 c2 c4 a $                                  matching ( sv3 )
$ SV3                n5 ac4 c2 c4 a $                                  SV3 --> n5 N5
$ N5 n5                  n5 ac4 c2 c4 a $                                  matching ( n5 )
$ N5                 ac4 c2 c4 a $                                  N5 --> ac4 AC4
$ AC4 ac4                ac4 c2 c4 a $                                  matching ( ac4 )
$ AC4                c2 c4 a $                                  explorar ( error )
$ AC4                c4 a $                                  AC4 --> c4 C4
$ C4 c4                  c4 a $                                  matching ( c4 )
$ C4                 a $                                  C4 --> a
$ a                  a $                                  matching ( a )
$                    $
                        REJECTED: More than 5 phases, Syntax error
```

Figura 3.5: ASh1Sv3N5Ac4C2C4A

La cadena de la Figura 3.6 no contiene ningún token identificado en la gramática, por lo que claramente presenta errores léxicos y sintácticos. Sin embargo, el *parser* trata de ignorar los tokens hasta encontrar uno con el que sí se pueda trabajar, y eso se da hasta el final. Asimismo,

para que la cadena pueda ser aceptada se realiza la acción *extraer* casi al último, por lo que debe ser rechazada.

Figura 3.6: #######$$$$$$$$$$(((((((((

En el caso de la Figura 3.7, la cadena contiene tokens de la gramática, pero no es posible que sea derivada. Para el manejo de errores se realizan acciones de *explorar* y *extraer*. Por ende, debe ser rechaza.

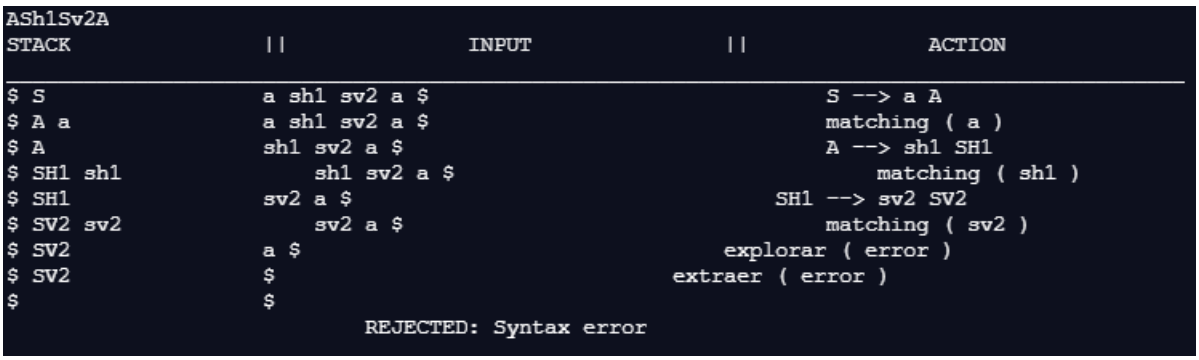

Figura 3.7: ASh1Sv2A

4. CONCLUSIONES

- Es necesario revisar la gramática y asegurarse que esta sea compatible con el tipo de parser a utilizar. En un primer momento tuvimos un acercamiento distinto a la gramática y esta no era LL. Sin embargo, en un segundo acercamiento logramos definir la gramática de tal manera que sea LL.
- Es importante el manejo de errores, pues éste nos permite identificar en dónde debe ser modificada la cadena de entrada para que el compilador pueda aceptarla. Además, reconocer el tipo de error ayuda a poder encontrar una corrección de la misma rápidamente. Por lo tanto, es importante diferenciar entre errores léxicos, sintácticos y los definidos por el creador.
- El lexer nos puede ayudar a definir errores fatales que causan un rechazo automático de la cadena. Estos errores pueden ser externos a la gramática, como en el caso de verificar las 5 etapas.
- Las buenas prácticas en la implementación nos ayudan a mejorar el código rápidamente y a poder separar las funcionalidades. De esta manera podemos reutilizar el código cuando sea necesario.

Appendices

A. Enlaces de interés

- GitHub