Universidad de Ingeniería y Tecnología

Compiladores

Profesor: José Fiestas

2021-0

Proyecto Final

Esteban Villacorta 201910

Daniela Abril Vento Bustamante 201910331

Jean Paul Huby Tuesta 201910194

Marzo del 2021

Índice general

1. Introducción

Si bien usamos mayoritariamente el mismo sistema de numeración, la forma en la que se escriben los nombres de los números varía entre cada idioma. El caso del alemán resulta interesante, pues es un poco enredado. Para visualizar esta idea, se considera la gramática en forma normal de Chomsky, con sus respectivas anotaciones por regla, mostrada a continuación:

$\mathbf{R_1} : \mathbf{Z_2} \rightarrow \mathbf{Z_3}+ \mathbf{zehn}$ // builds 1X, where X is a digit \e [3, 9]
$\mathbf{R_2} : \mathbf{Z_7} \rightarrow \mathbf{Z_4}+ \mathbf{zig}$ // builds X0, where X is a digit \e [2] U [4, 9]
$\mathbf{R_3} : \mathbf{Z_7} \rightarrow \mathbf{drei} + \mathbf{ssig}$ // builds 30
$\mathbf{R_4} : \mathbf{Z_8} \rightarrow \mathbf{U} + \mathbf{Z_7}$ // builds XY, where X, Y are digits \e [1, 9]
$\mathbf{R_5} : \mathbf{Z_9} \rightarrow \mathbf{Z_1} + \mathbf{Z_5}$ // builds X00, where X is a digit\e [1, 9]
$\mathbf{R_6} : \mathbf{Z_{10}} \rightarrow \mathbf{Z_2} + \mathbf{Z_5}$ // builds 1000, 1100, 1200
$\mathbf{R_7} : \mathbf{Z_{11}} \rightarrow \mathbf{Z_{5,9}} + \mathbf{Z_{1,2,7,8}}$ // builds XYZ, where X, Y, Z are digits \e [1,9]
$\mathbf{R_8} : \mathbf{Z_{12}} \rightarrow \mathbf{Z_{10}} + \mathbf{Z_{1,2,7,8}}$ // builds 10XY, 11XY, 12XY, where X, Y are digits \e [1, 9]
$\mathbf{R_9} : \mathbf{Z_{13}} \rightarrow \mathbf{Z_{1,2,7,8,9,11}} + \mathbf{Z_6}$ // builds X000, where X is a number \e [1, 999]
$\mathbf{R_{10}} : \mathbf{Z_{14}} \rightarrow \mathbf{Z_{6,13}} + \mathbf{Z_{1,2,7,8,9,11}}$ // builds XY, where X, Y are numbers \e [1, 999]
$\mathbf{R_{11}} : \mathbf{U} \rightarrow \mathbf{Z_1}+\mathbf{und}$ // builds "X und", where X is a digit \e [1, 9]

Donde:
$Z_1 = \{\text{ein, zwei, drei, vier, fünf, sechs, sieben, acht, neun}\}$
$Z_2 = \{\text{zehn, elf, zwölf }\}$
$Z_3 = \{\text{drei, vier, fünf, sech, sieb, acht, neun }\}$
$Z_4 = \{\text{zwan, vier, fünf, sech, sieb, acht, neun }\}$
$Z_5 = \{\text{hundert}\}$
$Z_6 = \{\text{tausend}\}$
$\text{drei} = \{\text{drei}\}$
$\text{zehn} = \{\text{zehn}\}$
$\text{zig} = \{\text{zig}\}$
$\text{ssig} = \{\text{ssig}\}$
$\text{und} = \{\text{und}\}$

Figura 1.1: Reglas originales provistas en el enunciado

Este proyecto tiene como objetivo hacer el front-end de un analizador para la construcción de los nombres de números en alemán, es decir, programar un scanner y un parser optimizado que lea, acepte y rechace (produciendo los mensajes de error que correspondan) cadenas de números escritos en alemán.

2. Elección del Método

Existen diversas formas de implemntar el analizador, cada una con sus propias ventajas y desventajas. En particular, diferentes métodos pueden ser más o menos eficientes que otros para reconocer una gramática dada.

La figura [2.1] muestra los diferentes métodos que se pueden utilizar para reconocer Context Free Grammars (CFG's). Como se puede ver, existen dos acercamientos generales al problema: un top-down parser comienza desde el no terminal inicial y trata de acoplarlo al input mediante las reglas provistas. En contraste, un bottom-up parser comienza desde el input y lo intenta reducir mediante las reglas hasta llegar al no terminal inicial.

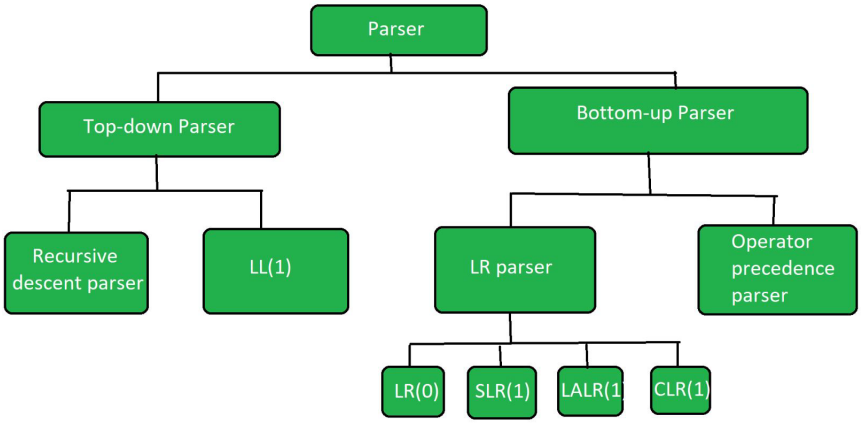

Figura 2.1: Diferentes categorías de parsers

Se consideraron diferentes posibilidades para implementar el programa. Se empezó por el método más fácil antes de pasar a opciones más complicadas, pero que permiten cubrir las necesidades de la gramática sugerida, además de optimizar el proceso.

2.1. Descenso recursivo no predictivo

Este es el método más simple de implementar, además del más general. Simplemente prueba todas las reglas en cada paso hasta llegar a derivar la cadena.

Este método puede parsear efectivamente la gramática dada, y es simple de implementar. Sin embargo, es bastante ineficiente para inputs largos, puesto que su complejidad escala exponencialmente en caso se requiera volver a un estado anterior. Debido a esto, no se utilizó para la implementación del proyecto.

2.2. LL(1)

El método LL(1) asume que cada no terminal tiene una forma única de reconocerse tras leer un token de lookahead por la izquierda del input stream. De esta manera, elimina la posibilidad de backtracking, incrementando así la eficiencia del parsing con respecto al descenso recursivo regular.

Al revisar la gramática, se puede rechazar de manera trivial que es LL(1) debido a que tenemos reglas como **z14** $\rightarrow$ **z6 z9** y **z14** $\rightarrow$ **z6 z11** que incumplen el método, puesto que los primeros coinciden (`Primero(z6 z9)` = `Primero(z6 z11)` = `Primero(z6)` != $\emptyset$).

La gramática podría alterarse para cumplir con los requerimientos. Sin embargo, dado que los atributos (valor retornado) son atributos sintetizados (es decir, se transmiten de abajo hacia arriba), no tiene mucho sentido utilizar este procedimiento, puesto que se requeriría realizar varios passes para transmitir información en el árbol sintáctico, eliminando la razón principal para usarlo (eficiencia). Además, el algoritmo puede expandir la gramática de forma significativa. Por tanto, esta implementación tampoco se utilizó en el proyecto.

2.3. LR(0)

El parsing por LR(0) asume que las reglas de derivación pueden reducirse poco a poco hasta llegar al estado inicial por la derecha. Este método no realiza ningún lookahead, por lo que es bastante eficiente. Sin embargo, es menos poderoso que otros métodos.

Debido a que el estado inicial va a contener sí o sí a los estados **z1** y **z3**, estos van a tener una misma transición con el terminal `drei` a otro estado. Por tanto, leer `drei` puede reducir por medio de dos reglas: **z1** $\rightarrow$ **drei.** y **z3** $\rightarrow$ **drei.**. Esto implica que existen conflictos al intentar reducir la gramática (por reducción-reducción). Por lo tanto, el parser LR(0) no funciona directamente con la gramática.

De la misma forma que para LL(1), se puede cambiar la gramática para cumplir con LR(0). Sin embargo, la gramática todavía puede expandirse mucho con respecto a su estado actual, y el proceso de transformación no es trivial. Por lo tanto, se decidió no utilizar este método.

2.4. SLR(1)

SLR(1) utiliza el proceso de LR(0) y lo combina con predicción mediante el set de Siguientes para realizar las reducciones. Esto asegura que, en el caso encontrar conflictos LR(0), el parser pueda hacer la elección correcta entre dos reducciones. Esto incrementa significativamente su efectividad.

No se encontraron propiamente conflictos para el parser SLR(1) con la gramática dada, por lo que no se requiere cambiar la gramática. Puesto que los atributos a utilizar son sintetizados, el parsing en bottom-up es también util. Debido a ello, se decidió utilizar este parser para el desarrollo del proyecto. Cabe mencionar, que es la opción más óptima.

2.5. Análisis de la gramática

Debido a que la no específica cual es el estado inicial por lo cual propondremos una nueva gramática que acepte la mayor cantidad de numeros cardinales alemanes.

Como podemos ver en la gramática original [1.1], los no terminales z1 al z2 sirven para referenciar directamente a los tokens terminales. Z2 especificamente se refiere a numeros unicos (10, 11 y 12). los tokens en estos estados pueden iniciar una cadena. si consideramos otros no terminales que contienen el grupo de tokes terminales como iniciales, tendríamos un caso de ambiguedad. Usando un método parecido con los demás tokens no terminales hicimos una nueva gramatica con el estado inicial definido (ver Anexo).

3. Desarrollo

Para poder reconcer/parsear una gramatica se necesita hacer uso de tres fases:

1. **Léxico:** se encarga de ver si cada **Token** es legal
2. **Semántico:** ve si la estructura de los tokens tenga sentido
3. **Sintáctico:** ve el significado de la oracion en su totalidad

3.1. Gramática

Cabe mencionar que para poder realizar las operaciones, es necesario que la gramática se encuentre en el código. Para hacer esto, lo que se hizo fue establecer un namespace **Grammar**, en el que se especifican las reglas, literales y tokens. Esta implementación se encuentra en el archivo **scanner.cpp**.

3.2. Léxico

El lexer es un programa que lee el input, agrupa sus caracteres en lexemas y, luego, con ayuda de la tabla de símbolos, emplea estos lexemas para producir una secuencia de tokens en el programa fuente. Esta implementación se encuentra en el archivo **scanner.cpp**. El fragmento de código de la función principal del lexer se muestra a continuación:

```
static std::vector<Item> lex (View stream)
{
    std::vector<Item> lexemes;
    unsigned position = 0;
    while (true) {
        auto item = lexToken(stream.substr(position));
        if (item.token == Grammar::$END)
        {
            lexemes.push_back(item);
            return lexemes;
        }
        if (item.token == Grammar::$ERROR)
        {
            position++;
            if (lexemes.empty() || lexemes.back().token != Grammar::$ERROR)
                lexemes.push_back(item);
            else
            {
                auto& back = lexemes.back().strVal;
                back = Grammar::View{ back.data(), back.size() + 1 };
            }
        }
        else
        {
            lexemes.push_back(item);
            position += item.strVal.size();
        }
    }
}
```

Como se puede ver, declara un vector de items, las cuales son estructuras que contienen un token, el string leído y su valor. En cada iteración, la función llama a lexToken, que devuelve el primer match entre todos los tokens posibles, o un error. Si llega al final del input, se detiene y retorna el vector. Si encuentra un error, mira si se encontraba anteriormente lexando un error, y agrega la sección del fault a este. En caso que no se trate del final ni de un error, simplemente se añade el elemento al final y se avanza a la siguiente posición.

A continuación se muestra la función que asigna los tokens al input leído:

```
static auto lexToken (View s)
{
    using namespace std;

    if (s.empty()) return Item{ $END, strOf($END), 0 };

    auto greater_than_s = [s = View(s)] (const auto& a) { return a > s; };
    auto gt_eq = greater_equal<>();
    auto noMatch = s.substr(0, 0);

    auto rtBegin = rbegin(TOKEN_STR), rtEnd = rend(TOKEN_STR) - 1;
    auto tBegin = begin(TOKEN_STR) + 1, tEnd = end(TOKEN_STR);

    auto rev = lower_bound(rtBegin, rtEnd, s, gt_eq);
    auto matchFst = decltype(TOKEN_STR)::const_iterator((rev + 1).base());
```

```
    auto matchLst = find_if(matchFst, tEnd, greater_than_s);

    if (rev == rtEnd || (*rev)[0] != s[0])
        matchFst++, matchLst++;

    auto matchFunc = matcher(s);
    auto match = accumulate(matchFst, matchLst, noMatch, matchFunc);

    auto fs = lower_bound(tBegin, tEnd, match);
    if (fs != tEnd && *fs == match)
    {
        auto token = static_cast<Token>(distance(begin(TOKEN_STR), fs));
        auto strVal = s.substr(0, match.size());
        auto item = Item{ token, strVal, valueOf(token) };
        return item;
    }
    return Item{ $ERROR, s.substr(0, 1) };
}
```

La función hace longest prefix matching entre los strings correspondientes a los tokens y la cadena, buscando primero un rango válido de candidatos y revisando el prefijo común con el input. En caso el prefijo mas largo corresponda a un token válido, retorna el token. Caso contrario, retorna un error. Dado que los tokens de error se concatenan en `lex`, esto permite retornar todos los tokens.

3.3. Semántico

El parser es un programa que recibe la cadena de tokens producida durante el análisis léxico y verifica que esta sea válida acorde a la gramática del lenguaje fuente. El pseudocódigo correspondiente es el siguiente (reducido del código original por legibilidad).

```
auto parse(const vector<Lexer::Item> &input)
{
    if (hasLexicErrors()) return false;
    stack.push(state{0});
    for (int index = 0; index < input.size(); )
    {
        if(stack.top().isState) {
            if (table.action[state][token] == $error) // try to recover from error;
            else if (action() == accepted) return errors.empty();
            if (action() == shift) stack.push(nextState);
            else if (action() == reduce) {doSyntaxAnalysis(); reduce(rule);}
            else return false;
        }
        else Goto(stack.top(), stack.belowTop());
    }
    return false;
}
```

Para ver el funcionamiento más detallado de código ver Anexo.

3.4. Sintáctico

El análisis sintáctico sucede al paralelo que el análisis semántico, ya que el valor del string es un atributo sintetizado que calcula al momento de hacer un reemplazamiento en el stack. Esto se observa esta sección especifica del código del parser en (ver Anexo).

3.5. Optimización

La primera optimización realizada fue utilizar el parser SLR(1) para el reconocimiento del input. Esto permite que el parser sea eficiente en cuanto a lectura de la cadena, además de para la síntesis de atributos.

Otra optimización agregada con respecto al enunciado original fue la capacidad de observar la construcción y la síntesis de atributos del árbol sintáctico (como se puede observar en los resultados). Como parte de los atributos, además, se agregó la recuperación del valor del input al convertirse en numero entero (e.g., drei = 3).

Una de las decisiones que se tomó fue convertir a las reglas y tokens en valores numéricos. Esto se hizo con el objetivo de hacer una comparación rápida entre valores. Además, permite identificar rápidamente el valor numérico de cada token durante la síntesis de valores. Esto es superior a la comparación con strings, que aunque posible, es significativamente más lenta, y no permite un mapping natural al valor.

El valor que toma los lexemas es decidido en **scanner.hpp**, mientras que el valor numérico de una regla es decidido en **json.h** cuando lee el archivo **rules.txt**, donde el numero de línea donde se encuentra la regla como su valor numérico.

3.6. Manejo de errores

El manejo de errores se divide en dos partes:

3.6.1. Errores Léxicos

Como fue aclarado en La sección 3.2 el análisis léxico nos devuelve un vector de items. Este después se utiliza en el parser de la siguiente forma:

```
auto parse(const vector<Lexer::Item> &input)
{
    using Entry = struct
    {
        bool isState;
        Grammar::Type stored;
        Grammar::View strVal = "";
        int val = 0;
    };

    auto lexicErrors = readLexicErrors(input);
    if (lexicErrors)
        return false;
.
.
.
}
```

Cuando entra a **readLexicErrors** lee todo los items de input y si encuentra un error imprime un mensaje y sugiere el token legal mas similar. Esto se hace haciendo uso de la siguiente función:

```
string test(std::string str)
{
    int tempRes = 0;
    int temp = INT_MAX;
    for (int i = 0; i < Grammar::NUM_TOKENS; i++)
    {
        auto res = levDistance(str, static_cast<const string>(Grammar::TOKEN_STR[i]));
        if (temp > res)
        {
            temp = res;
            tempRes = i;
        }
    }

    return string(Grammar::TOKEN_STR[tempRes]);
}
```

La función levDistance determina la diferencia relativa entre un string y otro (no incluida por motivos de legibilidad). Por tanto, la función test determina el mejor match para un input dado. De esta forma, se puede determinar el mejor reemplazo para el caso de un input erróneo.

3.6.2. Errores semánticos

Los errores semánticos suceden cuando en el análisis semántico no hay una acción legal desde un estado usando un token como transición. En caso de que suceda eso se tiene que seguir una serie de instrucciones para decidir si explotar o extraer de la siguiente manera:

```
if (action.first.empty()) {
    //push error index
    errors.push_back(index);
    auto& entry = table.goTo[state];
    auto nToken = entryOf(token, entry);
    //explore
    if (nToken != Grammar::$ERROR) {
        symbolStack.push(Entry{ false, indexOfProduction(nToken),
            PRODUCTION_STR[nToken], 0});
       symbolStack.push(Entry{ true, entry[nToken]});
        }
    //extract
    else index++;
}
```

A penas se detecta alguno de estos casos se guarda la posicion del error en el input para luego printear todos los errores encontrados.

4. Resultados

Se testeo el funcionamiento del parser usando los siguientes tests:

4.1. Test 1

Input	Valor Numérico	resultado
zweihundert zweiundzwanzig tausendvierhundert siebzehng	222417	aceptado

```
C:\Users\esteb\Desktop\asdfasdf>a.exe
zweihundertzweiundzwanzigtausendvierhundertsiebzehn
2[zwei]=2[z1]
100[hundert]=100[z5]
100[z5]*2[z1]=200[z9]
2[zwei]=2[z1]
0[und]+2[z1]=2[u]
2[zwan]=2[z4]
10[zig]*2[z4]=20[z7]
20[z7]+2[u]=22[z8]
22[z8]+200[z9]=222[z11]
1000[tausend]=1000[z6]
1000[z6]*222[z11]=222000[z13]
222000[z13]=222000[z613]
4[vier]=4[z1]
100[hundert]=100[z5]
100[z5]*4[z1]=400[z9]
7[sieb]=7[z3]
10[zehn]+7[z3]=17[z2]
17[z2]+400[z9]=417[z11]
417[z11]=417[z1278911]
417[z1278911]+222000[z613]=222417[z14]
222417[z14]=222417[s]
Accepted: true
```

Figura 4.1: Test 1 cadena aceptada

4.2. Test 2

Input	Valor Numérico	resultado
zweitausend neunhundert sechsundsiebzig	2976	aceptado

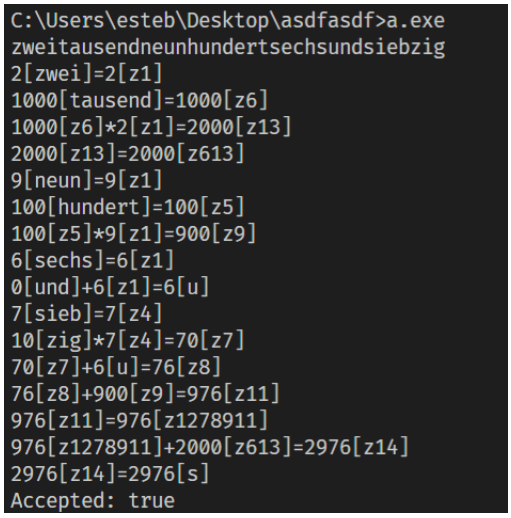

Figura 4.2: Test 2 cadena aceptada

4.3. Test 3

Input	Valor Numérico	resultado
fuenftausendzwei hundertneunund fuenfzig	222417	aceptado

```
C:\Users\esteb\Desktop\asdfasdf>a.exe
fuenftausendzweihundertneunundfuenfzig
5[fuenf]=5[z1]
1000[tausend]=1000[z6]
1000[z6]*5[z1]=5000[z13]
5000[z13]=5000[z613]
2[zwei]=2[z1]
100[hundert]=100[z5]
100[z5]*2[z1]=200[z9]
9[neun]=9[z1]
0[und]+9[z1]=9[u]
5[fuenf]=5[z4]
10[zig]*5[z4]=50[z7]
50[z7]+9[u]=59[z8]
59[z8]+200[z9]=259[z11]
259[z11]=259[z1278911]
259[z1278911]+5000[z613]=5259[z14]
5259[z14]=5259[s]
Accepted: true
```

Figura 4.3: Test 3 cadena aceptada

4.4. Test 4

Input	Valor Numérico	resultado
eineinhundert	NONE	Rechazado: Error semántico

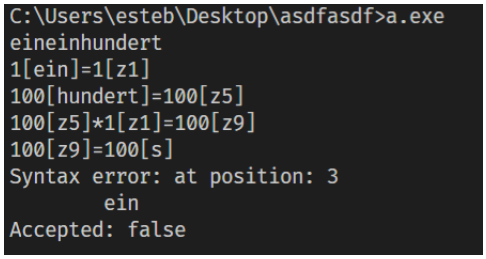

Figura 4.4: Test 4 error semántico

4.5. Test 5

Input	Valor Numérico	resultado
oineundgis	NONE	Rechazado: Error léxico

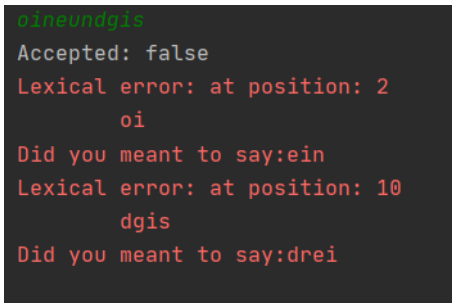

Figura 4.5: Test 5 error léxico

5. Conclusiones

En principio, cabe mencionar que el objetivo del proyecto fue alcanzado, dado que se pudo realizar la implementación de un scanner y parser optimizado para la lectura de cadenas de números redactados en alemán.

Al llevar a cabo este proyecto, se pudo apreciar la interacción entre los programas requeridos para hacer que el front-end funcione. A partir de ello se observó que, con respecto al análisis sintáctico del lenguaje, este caso es evidencia de que no todos los lenguajes usan las operaciones en el mismo orden.

En idiomas como el alemán la estructura para construir números es distinta comparado con otros lenguajes. Por ejemplo, en el español se puede analizar de izquierda a derecha, donde a la izquierda están las expresiones de mayor valor, mientras que en el alemán hay expresiones como **zweiundzwanzig**, donde la unidad está a la izquierda y la decena a la derecha teniendo la magnitud de los números de manera más mezclada al que se está acostumbrado. Por ello se necesita su propia lógica o gramática para poder ser entendido por una computadora.

Asimismo, la estructura de los números en lenguajes tienen forma normal de Chomsky. Con el análisis del lenguaje dado, nos dimos cuenta de que el parser SLR(1), y sus mejoras son mucho más óptimas para lidiar con este tipo de casos debido a que son menos susceptibles a ser ambiguos.

5.1. Mejoras

En vez de utilizar la gramática en su forma actual (sin reglas de error), se podría cambiar la gramática para que la misma controle los errores. De esta forma, se mejoraría la flexibilidad del parser, y se mejoraría la calidad de los mensajes de error, que actualmente se manejan de forma manual.

Otra mejora que se puede hacer es usar mejores parser como LALR(1) para reducir el número de estados del DFA para analizar la cadena.

A. Anexos

A.1. Repositorio del Proyecto

GitHub (ver rama PARSER).

A.2. Gramatica de los numeros cardinales alemanes

S' ->S
S ->z1 | z2 | z7 | z8 | z9 | z10 | z11 | z12 | z13 | z14
z2 ->z3 zehn
z7 ->z4 zig | drei ssig
z8 ->u z7
z9 ->z1 z5
z10 ->z2 z5
z11 ->z5 z1 | z5 z2 | z5 z7 | z5 z8 | z9 z1 | z9 z2 | z9 z7 | z9 z8
z12 ->z10 z1 | z10 z2 | z10 z7 | z10 z8
z13 ->z1 z6 | z2 z6 | z7 z6 | z8 z6 | z9 z6 | z11 z6
z14 ->z613 z1278911
u ->z1 und
z613 ->z6 | z13
z1278911 ->z1 | z2 | z7 | z8 | z9 | z11
z1 ->ein | zwei | drei | vier | fuenf | sechs | sieben | acht | neun
z2 ->zehn | elf | zwoelf
z3 ->drei | vier | fuenf | sech | sieb | acht | neun
z4 ->zwan | vier | fuenf | sech | sieb | acht | neun

z5 ->hundert

z6 ->tausend

A.3. Parser

```
auto parse(const vector<Lexer::Item> &input) {
    using namespace Grammar;
    if (readLexicErrors(input)) return false;
    stack<Entry> symbolStack;
    symbolStack.emplace(Entry{true, 0});
    for (int index = 0; index < input.size(); ) {
        if (symbolStack.top().isState) {
            auto state = symbolStack.top().stored;
            auto token = input.at(index).token;
            auto action = table.action[state][token];
            if (action.first.empty()) {
                errors.push_back(index);
                auto& entry = table.goTo[state];
                auto nToken = entryOf(token, entry);

                if (nToken != Grammar::$ERROR) {
                    symbolStack.push(Entry{ false, indexOfProduction(nToken),
                        PRODUCTION_STR[nToken], 0});
                    symbolStack.push(Entry{ true, entry[nToken]});
                }
                else index++;

            }
            else if (action.first == "acc") return errors.empty();
            if (action.first == "s") {
                symbolStack.push(Entry{false, token, input.at(index).strVal,
                    input.at(index).value});
                symbolStack.push(Entry{true, action.second});
                index++;
            }
            else if (action.first == "r") {
                Doreplacement();
                syntaxAnalysis();
            }
            else return false;
        }
        else // goto
        {
            auto rule = symbolStack.top();
            symbolStack.pop();
            auto state = symbolStack.top().stored;
            symbolStack.push(rule);
            symbolStack.push(Entry{true, table.goTo[state][rule.stored]});
        }
    }
    return false;
}
```

A.4. Análisis sintáctico

```
// syntaxAnalysis
// repl -> rhs decomposition
auto [repl, rhs] = table.rules[action.second];
     string strTemp = {};
    int valTemp = {};
    bool firstRead = true;

    using namespace Grammar;
    vector<Grammar::Type> multipication{Z7, Z9, Z10, Z13};
    // begin replacement
    while (!rhs.empty()){

        symbolStack.pop();
        // if replacement fails return false
        if (symbolStack.top().stored != rhs.back())
            return false;

        auto item = symbolStack.top();
        bool f = (State is not in multiplication);

        if (firstRead || (f)){
            valTemp += item.val;
            firstRead = false;
        }
        else valTemp *= item.val;

        symbolStack.pop();
        rhs.pop_back();
    }
    symbolPush(Replacement);
```

Bibliografía

[1] Levenshtein Distance. 24/3/2021, de wikipedia web: https://en.wikipedia.org/wiki/Levenshtein_distance

[2] Louden, K. (1997). Compiler Construction Principles and Practice. PWS Publishing Company