Proyecto Final - Compiladores

Integrantes
Claudia Noche
Mauricio Bernuy
Miguiel Yurivilca

December 2020

1 Introducción

El poder leer una línea de texto bajo cierto contexto e interpretarla de forma debida, es un proceso que ha permitido el avance de la computación de alto nivel, permitiendo paralelamente el avance tecnológico. Los compiladores, por ende, han sido desarrollados como los traductores entre los códigos de los lenguajes programación y un código objetivo, generalmente siendo código de máquina.

Dentro del proceso de compilación, se encuentran varios procesos igualmente importantes que el compilador debe realizar previo a devolver un código objetivo, con el fin de asegurar la calidad y efectividad de dicho código. Entre estas, las dos primeras etapas son las que tienen como fin la lectura y la *tokenización* del código. El **análisis léxico** constituye la primera fase, donde este lee y "tokeniza" el código fuente en base a ciertas reglas definidas en el "lexer". Posterior a esto, se revisa que los tokens estén puestos en un orden lógico, tarea del análisis sintáctico. Se prueba en esta etapa si la línea de tokens pertenece o no a la gramática delimitada. "Lex" y "Yacc" son dos tipos de analizadores que, respectivamente, se encargan de estas tareas.

En este trabajo, se realizará un análisis léxico y sintáctico de la gramática de números cardinales en alemán, utilizando Lex y Yacc para cada paso respectivamente.

2 Marco teórico

2.1 Análisis Léxico

Como ya se mencionó previamente, el análisis léxico es el cual lee el código fuente, y, en base a reglas definidas, devuelve un stream de tokens. Para realizar esta tarea, usaremos el analizador "Lex". En Lex, se pueden definir dichas descripciones, conocidas como "lex specifications", de forma simple, creando así

un eficiente analizador léxico que scannea los tokens a través de una rutina en C.

2.2 Análisis Sintáctico

Luego de tokennizar el código fuente, se requiere llevar a cabo el análisis sintáctico. En este, se analiza la cadena de tokens de acuerdo a la gramática especificada. Al estar siendo dividido el código en tokens, se requiere establecer las relaciones entre ellos, pues se requiere identificar las expresiones, declaraciones, bloques y procedimientos establecidos en el programa. Este proceso es conocido como "parsing" y depende de una serie de reglas, establecidas en la gramática, para definir dichas relaciones. Yacc es un parser que permite realizar dicha tarea de forma eficiente, detectando automáticamente cuando una secuencia de tokens no pertenece a la gramática.

Ambos programas serán utilizados para el desarrollo de este trabajo.

3 Método y Desarrollo

Como ya se mencionó previamente, se analizará la gramática de números cardinales en alemán en la forma normal de Chomsky. Esta se puede apreciar en la figura 1.

Figure 1:

$$\mathbf{R_1 : Z_2 \rightarrow Z_3 + zehn}$$
$$\mathbf{R_2 : Z_7 \rightarrow Z_4 + zig}$$
$$\mathbf{R_3 : Z_7 \rightarrow drei + ssig}$$
$$\mathbf{R_4 : Z_8 \rightarrow U + Z_7}$$
$$\mathbf{R_5 : Z_9 \rightarrow Z_1 + Z_5}$$
$$\mathbf{R_6 : Z_{10} \rightarrow Z_2 + Z_5}$$
$$\mathbf{R_7 : Z_{11} \rightarrow Z_{5,9} + Z_{1,2,7,8}}$$
$$\mathbf{R_8 : Z_{12} \rightarrow Z_{10} + Z_{1,2,7,8}}$$
$$\mathbf{R_9 : Z_{13} \rightarrow Z_{1,2,7,8,9,11} + Z_6}$$
$$\mathbf{R_{10} : Z_{14} \rightarrow Z_{6,13} + Z_{1,2,7,8,9,11}}$$
$$\mathbf{R_{11} : U \rightarrow Z_1 + und}$$

Donde:

Figure 2:

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

4 Implementación

Implementamos la gramática como fue presentada en el enunciado. En las siguientes imágenes, se puede ver nuestra implementación en LEX y YACC. Se puede encontrar este código en el siguiente link: `https://github.com/Mauricio-Bernuy/COMPILATORS-FINAL-PROJECT`.

Figure 3:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
ein          return EIN;
zwei         return ZWEI;
drei         return DREI;
vier         return VIER;
fünf         return FUNF;
sechs        return SECHS;
sieben       return SIEBEN;
acht         return ACHT;
neun         return NEUN;
zehn         return ZEHN;
elf          return ELF;
zwölf        return ZWOLF;
zwan         return ZWAN;
sech         return SECH;
sieb         return SIEB;
hundert      return HUNDERT;
tausend      return TAUSEND;
zig          return ZIG;
ssig         return SSIG;
und          return UND;
"\n"         return NUMBEREND;{return(0);};
.            return yytext[0];
<<EOF>>      yyterminate();
[ \t]+       /* ignore whitespace */;
%%
```

En la Figura 3, se puede ver el LEX implementado para esta gramática. Se aceptan todos los tokens que conforman los números en alemán en minúscula y se les asigna un token dependiendo de la cadena recibida.

Figure 4:

```
%{
#include <stdio.h>
#include <string.h>

int yydebug=1;

void yyerror(const char *str)
{
    fprintf(stderr, "\n\nString contains error: %s\n\n", str);
}

int yywrap()
{
    return(1);
}

main()
{
    return(yyparse());
}

%}

%start S
%token EIN ZWEI DREI VIER FUNF SECHS SIEBEN ACHT NEUN ZEHN ELF ZWOLF ZWAN SECH SIEB HUNDERT TAUSEND ZIG SSIG UND NUMBEREND
%%
```

Posterior a eso, en el YACC, denominado "german.y", se crea una función que detecta errores y devuelve un mensaje cuando las cadenas o están mal escritas, o en un orden incorrecto. Se definen los tokens previamente creados en el LEX y se denomina S como el estado de inicio. Con el fin de revisar que una cadena esté entera, se revisa si la producción final tiene un token al final, .

Figure 5:

```
//S: /*empty*/ | S productions {printf("S productions");};
S: /*empty*/ | productions NUMBEREND {printf("\n\nProduction reduced successfully!\n\n"); return(0);};
//productions: Z2 | Z7 | Z8 | Z9 | Z10 | Z11 | Z12 | Z13 | Z14 | U
productions: Z1 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | R11;

// productions
R1: Z2_2;
R2: Z7_1;
R3: Z7_2;
R4: Z8;
R5: Z9;
R6: Z10;
R7: Z11;
R8: Z12;
R9: Z13;
R10: Z14;
R11: U;
```

Se denominan las producciones al igual que en el ejemplo establecido. Se crea el estado "productions" el cual recibe las producciones válidas como producciones finales.
En la Figura 6, se van conformando las reglas con los terminales provistas por el ejemplo en la Figura 1.

Como se va a ir reduciendo en producciones mientras se lean los tokens, se crearon unas reglas extra para poder abarcar todos los casos.

Figure 6:

```
// terminals
Z1: EIN | ZWEI | DREI | VIER | FUNF | SECHS | SIEBEN | ACHT | NEUN;
Z2_1: ZEHN | ELF | ZWOLF;
Z3: DREI | VIER | FUNF | SECH | SIEB | ACHT | NEUN;
Z4: ZWAN | VIER | FUNF | SECH | SIEB | ACHT | NEUN;
Z5: HUNDERT;
Z6: TAUSEND;
drei: DREI;
zehn: ZEHN;
zig: ZIG;
ssig: SSIG;
und: UND;

// rules
Z2_2: Z3 zehn;
Z2: Z2_1 | Z2_2;
Z7_1: Z4 zig;
Z7_2: drei ssig;
Z7: Z7_1 | Z7_2;
Z8: U Z7;
Z9: Z1 Z5;
Z10: Z2 Z5;
Z11: C1 C2;
Z12: Z10 C2;
Z13: C3 Z6;
Z14: C4 C3;
U: Z1 und;

C1: Z5 | Z9;
C2: Z1 | Z2 | Z7 | Z8;
C3: Z1 | Z2 | Z7 | Z8 | Z9 | Z11;
C4: Z6 | Z13;
```

5 Resultados

En esta sección, se prueban 3 números en alemán. Se muestra una imagen del debugger para mostrar el resultado.

Test 1

Cadena: zweitausendneunhundertsechsundsiebzig

Resultado: SUCCESS

```
Starting parse
Entering state 0
Reading a token: fnftausendzweihundertneunundfnfzig
Next token is token FUNF ()
Shifting token FUNF ()
Entering state 5
Reading a token: Next token is token TAUSEND ()
Reducing stack by rule 30 (line 47):
   $1 = token FUNF ()
-> $$ = nterm Z1 ()
Stack now 0
Entering state 31
Next token is token TAUSEND ()
Reducing stack by rule 80 (line 76):
   $1 = nterm Z1 ()
-> $$ = nterm C3 ()
Stack now 0
Entering state 52
Next token is token TAUSEND ()
Shifting token TAUSEND ()
Entering state 17
Reducing stack by rule 53 (line 52):
   $1 = token TAUSEND ()
-> $$ = nterm Z6 ()
Stack now 0 52
Entering state 84
Reducing stack by rule 71 (line 70):
   $1 = nterm C3 ()
   $2 = nterm Z6 ()
-> $$ = nterm Z13 ()
Stack now 0
Entering state 48
Reading a token: Next token is token ZWEI ()
Reducing stack by rule 87 (line 77):
   $1 = nterm Z13 ()
-> $$ = nterm C4 ()
Stack now 0
```

```
Entering state 53
Next token is token ZWEI ()
Shifting token ZWEI ()
Entering state 2
Reducing stack by rule 27 (line 47):
   $1 = token ZWEI ()
-> $$ = nterm Z1 ()
Stack now 0 53
Entering state 85
Reading a token: Next token is token HUNDERT ()
Shifting token HUNDERT ()
Entering state 16
Reducing stack by rule 52 (line 51):
   $1 = token HUNDERT ()
-> $$ = nterm Z5 ()
Stack now 0 53 85
Entering state 57
Reducing stack by rule 67 (line 66):
   $1 = nterm Z1 ()
   $2 = nterm Z5 ()
-> $$ = nterm Z9 ()
Stack now 0 53
Entering state 88
Reading a token: Next token is token NEUN ()
Reducing stack by rule 75 (line 74):
   $1 = nterm Z9 ()
-> $$ = nterm C1 ()
Stack now 0 53
Entering state 51
Next token is token NEUN ()
Shifting token NEUN ()
Entering state 9
Reading a token: Next token is token UND ()
Reducing stack by rule 34 (line 47):
   $1 = token NEUN ()
-> $$ = nterm Z1 ()
Stack now 0 53 51
Entering state 66
Next token is token UND ()
Shifting token UND ()
Entering state 56
Reducing stack by rule 58 (line 57):
   $1 = token UND ()
-> $$ = nterm und ()
Stack now 0 53 51 66
Entering state 58
Reducing stack by rule 73 (line 72):
   $1 = nterm Z1 ()
   $2 = nterm und ()
-> $$ = nterm U ()
```

```
Stack now 0 53 51
Entering state 73
Reading a token: Next token is token FUNF ()
Shifting token FUNF ()
Entering state 77
Reducing stack by rule 47 (line 50):
   $1 = token FUNF ()
-> $$ = nterm Z4 ()
Stack now 0 53 51 73
Entering state 34
Reading a token: Next token is token ZIG ()
Shifting token ZIG ()
Entering state 61
Reducing stack by rule 56 (line 55):
   $1 = token ZIG ()
-> $$ = nterm zig ()
Stack now 0 53 51 73 34
Entering state 62
Reducing stack by rule 62 (line 62):
   $1 = nterm Z4 ()
   $2 = nterm zig ()
-> $$ = nterm Z7_1 ()
Stack now 0 53 51 73
Entering state 69
Reducing stack by rule 64 (line 64):
   $1 = nterm Z7_1 ()
-> $$ = nterm Z7 ()
Stack now 0 53 51 73
Entering state 82
Reducing stack by rule 66 (line 65):
   $1 = nterm U ()
   $2 = nterm Z7 ()
-> $$ = nterm Z8 ()
Stack now 0 53 51
Entering state 72
Reducing stack by rule 79 (line 75):
   $1 = nterm Z8 ()
-> $$ = nterm C2 ()
Stack now 0 53 51
Entering state 83
Reducing stack by rule 69 (line 68):
   $1 = nterm C1 ()
   $2 = nterm C2 ()
-> $$ = nterm Z11 ()
Stack now 0 53
Entering state 89
Reducing stack by rule 85 (line 76):
   $1 = nterm Z11 ()
-> $$ = nterm C3 ()
Stack now 0 53
```

```
Entering state 90
Reducing stack by rule 72 (line 71):
   $1 = nterm C4 ()
   $2 = nterm C3 ()
-> $$ = nterm Z14 ()
Stack now 0
Entering state 49
Reducing stack by rule 24 (line 43):
   $1 = nterm Z14 ()
-> $$ = nterm R10 ()
Stack now 0
Entering state 29
Reducing stack by rule 13 (line 31):
   $1 = nterm R10 ()
-> $$ = nterm productions ()
Stack now 0
Entering state 19
Reading a token: Next token is token NUMBEREND ()
Shifting token NUMBEREND ()
Entering state 55
Reducing stack by rule 2 (line 29):
   $1 = nterm productions ()
   $2 = token NUMBEREND ()


Production reduced successfully!
```

Test 2

Cadena: fünftausendzweihundertneunundfünfzig

Resultado: SUCCESS

```
Starting parse
Entering state 0
Reading a token: zweitausendneunhundertsechsundsiebzig
Next token is token ZWEI ()
Shifting token ZWEI ()
Entering state 2
Reducing stack by rule 27 (line 47):
   $1 = token ZWEI ()
-> $$ = nterm Z1 ()
Stack now 0
Entering state 31
Reading a token: Next token is token TAUSEND ()
Reducing stack by rule 80 (line 76):
   $1 = nterm Z1 ()
```

```
-> $$ = nterm C3 ()
Stack now 0
Entering state 52
Next token is token TAUSEND ()
Shifting token TAUSEND ()
Entering state 17
Reducing stack by rule 53 (line 52):
   $1 = token TAUSEND ()
-> $$ = nterm Z6 ()
Stack now 0 52
Entering state 84
Reducing stack by rule 71 (line 70):
   $1 = nterm C3 ()
   $2 = nterm Z6 ()
-> $$ = nterm Z13 ()
Stack now 0
Entering state 48
Reading a token: Next token is token NEUN ()
Reducing stack by rule 87 (line 77):
   $1 = nterm Z13 ()
-> $$ = nterm C4 ()
Stack now 0
Entering state 53
Next token is token NEUN ()
Shifting token NEUN ()
Entering state 9
Reading a token: Next token is token HUNDERT ()
Reducing stack by rule 34 (line 47):
   $1 = token NEUN ()
-> $$ = nterm Z1 ()
Stack now 0 53
Entering state 85
Next token is token HUNDERT ()
Shifting token HUNDERT ()
Entering state 16
Reducing stack by rule 52 (line 51):
   $1 = token HUNDERT ()
-> $$ = nterm Z5 ()
Stack now 0 53 85
Entering state 57
Reducing stack by rule 67 (line 66):
   $1 = nterm Z1 ()
   $2 = nterm Z5 ()
-> $$ = nterm Z9 ()
Stack now 0 53
Entering state 88
Reading a token: Next token is token SECHS ()
Reducing stack by rule 75 (line 74):
   $1 = nterm Z9 ()
-> $$ = nterm C1 ()
```

```
Stack now 0 53
Entering state 51
Next token is token SECHS ()
Shifting token SECHS ()
Entering state 6
Reducing stack by rule 31 (line 47):
   $1 = token SECHS ()
-> $$ = nterm Z1 ()
Stack now 0 53 51
Entering state 66
Reading a token: Next token is token UND ()
Shifting token UND ()
Entering state 56
Reducing stack by rule 58 (line 57):
   $1 = token UND ()
-> $$ = nterm und ()
Stack now 0 53 51 66
Entering state 58
Reducing stack by rule 73 (line 72):
   $1 = nterm Z1 ()
   $2 = nterm und ()
-> $$ = nterm U ()
Stack now 0 53 51
Entering state 73
Reading a token: Next token is token SIEB ()
Shifting token SIEB ()
Entering state 81
Reducing stack by rule 49 (line 50):
   $1 = token SIEB ()
-> $$ = nterm Z4 ()
Stack now 0 53 51 73
Entering state 34
Reading a token: Next token is token ZIG ()
Shifting token ZIG ()
Entering state 61
Reducing stack by rule 56 (line 55):
   $1 = token ZIG ()
-> $$ = nterm zig ()
Stack now 0 53 51 73 34
Entering state 62
Reducing stack by rule 62 (line 62):
   $1 = nterm Z4 ()
   $2 = nterm zig ()
-> $$ = nterm Z7_1 ()
Stack now 0 53 51 73
Entering state 69
Reducing stack by rule 64 (line 64):
   $1 = nterm Z7_1 ()
-> $$ = nterm Z7 ()
Stack now 0 53 51 73
```

```
Entering state 82
Reducing stack by rule 66 (line 65):
   $1 = nterm U ()
   $2 = nterm Z7 ()
-> $$ = nterm Z8 ()
Stack now 0 53 51
Entering state 72
Reducing stack by rule 79 (line 75):
   $1 = nterm Z8 ()
-> $$ = nterm C2 ()
Stack now 0 53 51
Entering state 83
Reducing stack by rule 69 (line 68):
   $1 = nterm C1 ()
   $2 = nterm C2 ()
-> $$ = nterm Z11 ()
Stack now 0 53
Entering state 89
Reducing stack by rule 85 (line 76):
   $1 = nterm Z11 ()
-> $$ = nterm C3 ()
Stack now 0 53
Entering state 90
Reducing stack by rule 72 (line 71):
   $1 = nterm C4 ()
   $2 = nterm C3 ()
-> $$ = nterm Z14 ()
Stack now 0
Entering state 49
Reducing stack by rule 24 (line 43):
   $1 = nterm Z14 ()
-> $$ = nterm R10 ()
Stack now 0
Entering state 29
Reducing stack by rule 13 (line 31):
   $1 = nterm R10 ()
-> $$ = nterm productions ()
Stack now 0
Entering state 19
Reading a token: Next token is token NUMBEREND ()
Shifting token NUMBEREND ()
Entering state 55
Reducing stack by rule 2 (line 29):
   $1 = nterm productions ()
   $2 = token NUMBEREND ()


Production reduced successfully!
```

Test 3

Cadena: zweihundertzweiundzwanzigtausendvierhundertsiebzehn

Resultado: SUCCESS

```
Starting parse
Entering state 0
Reading a token: zweihundertzweiundzwanzigtausendvierhundertsiebzehn
Next token is token ZWEI ()
Shifting token ZWEI ()
Entering state 2
Reducing stack by rule 27 (line 47):
   $1 = token ZWEI ()
-> $$ = nterm Z1 ()
Stack now 0
Entering state 31
Reading a token: Next token is token HUNDERT ()
Shifting token HUNDERT ()
Entering state 16
Reducing stack by rule 52 (line 51):
   $1 = token HUNDERT ()
-> $$ = nterm Z5 ()
Stack now 0 31
Entering state 57
Reducing stack by rule 67 (line 66):
   $1 = nterm Z1 ()
   $2 = nterm Z5 ()
-> $$ = nterm Z9 ()
Stack now 0
Entering state 44
Reading a token: Next token is token ZWEI ()
Reducing stack by rule 75 (line 74):
   $1 = nterm Z9 ()
-> $$ = nterm C1 ()
Stack now 0
Entering state 51
Next token is token ZWEI ()
Shifting token ZWEI ()
Entering state 2
Reducing stack by rule 27 (line 47):
   $1 = token ZWEI ()
-> $$ = nterm Z1 ()
Stack now 0 51
Entering state 66
Reading a token: Next token is token UND ()
Shifting token UND ()
Entering state 56
Reducing stack by rule 58 (line 57):
```

```
   $1 = token UND ()
-> $$ = nterm und ()
Stack now 0 51 66
Entering state 58
Reducing stack by rule 73 (line 72):
   $1 = nterm Z1 ()
   $2 = nterm und ()
-> $$ = nterm U ()
Stack now 0 51
Entering state 73
Reading a token: Next token is token ZWAN ()
Shifting token ZWAN ()
Entering state 13
Reducing stack by rule 45 (line 50):
   $1 = token ZWAN ()
-> $$ = nterm Z4 ()
Stack now 0 51 73
Entering state 34
Reading a token: Next token is token ZIG ()
Shifting token ZIG ()
Entering state 61
Reducing stack by rule 56 (line 55):
   $1 = token ZIG ()
-> $$ = nterm zig ()
Stack now 0 51 73 34
Entering state 62
Reducing stack by rule 62 (line 62):
   $1 = nterm Z4 ()
   $2 = nterm zig ()
-> $$ = nterm Z7_1 ()
Stack now 0 51 73
Entering state 69
Reducing stack by rule 64 (line 64):
   $1 = nterm Z7_1 ()
-> $$ = nterm Z7 ()
Stack now 0 51 73
Entering state 82
Reducing stack by rule 66 (line 65):
   $1 = nterm U ()
   $2 = nterm Z7 ()
-> $$ = nterm Z8 ()
Stack now 0 51
Entering state 72
Reducing stack by rule 79 (line 75):
   $1 = nterm Z8 ()
-> $$ = nterm C2 ()
Stack now 0 51
Entering state 83
Reducing stack by rule 69 (line 68):
   $1 = nterm C1 ()
```

```
   $2 = nterm C2 ()
-> $$ = nterm Z11 ()
Stack now 0
Entering state 46
Reading a token: Next token is token TAUSEND ()
Reducing stack by rule 85 (line 76):
   $1 = nterm Z11 ()
-> $$ = nterm C3 ()
Stack now 0
Entering state 52
Next token is token TAUSEND ()
Shifting token TAUSEND ()
Entering state 17
Reducing stack by rule 53 (line 52):
   $1 = token TAUSEND ()
-> $$ = nterm Z6 ()
Stack now 0 52
Entering state 84
Reducing stack by rule 71 (line 70):
   $1 = nterm C3 ()
   $2 = nterm Z6 ()
-> $$ = nterm Z13 ()
Stack now 0
Entering state 48
Reading a token: Next token is token VIER ()
Reducing stack by rule 87 (line 77):
   $1 = nterm Z13 ()
-> $$ = nterm C4 ()
Stack now 0
Entering state 53
Next token is token VIER ()
Shifting token VIER ()
Entering state 4
Reading a token: Next token is token HUNDERT ()
Reducing stack by rule 29 (line 47):
   $1 = token VIER ()
-> $$ = nterm Z1 ()
Stack now 0 53
Entering state 85
Next token is token HUNDERT ()
Shifting token HUNDERT ()
Entering state 16
Reducing stack by rule 52 (line 51):
   $1 = token HUNDERT ()
-> $$ = nterm Z5 ()
Stack now 0 53 85
Entering state 57
Reducing stack by rule 67 (line 66):
   $1 = nterm Z1 ()
   $2 = nterm Z5 ()
```

```
-> $$ = nterm Z9 ()
Stack now 0 53
Entering state 88
Reading a token: Next token is token SIEB ()
Reducing stack by rule 75 (line 74):
   $1 = nterm Z9 ()
-> $$ = nterm C1 ()
Stack now 0 53
Entering state 51
Next token is token SIEB ()
Shifting token SIEB ()
Entering state 15
Reading a token: Next token is token ZEHN ()
Reducing stack by rule 42 (line 49):
   $1 = token SIEB ()
-> $$ = nterm Z3 ()
Stack now 0 53 51
Entering state 33
Next token is token ZEHN ()
Shifting token ZEHN ()
Entering state 59
Reducing stack by rule 55 (line 54):
   $1 = token ZEHN ()
-> $$ = nterm zehn ()
Stack now 0 53 51 33
Entering state 60
Reducing stack by rule 59 (line 60):
   $1 = nterm Z3 ()
   $2 = nterm zehn ()
-> $$ = nterm Z2_2 ()
Stack now 0 53 51
Entering state 67
Reducing stack by rule 61 (line 61):
   $1 = nterm Z2_2 ()
-> $$ = nterm Z2 ()
Stack now 0 53 51
Entering state 68
Reducing stack by rule 77 (line 75):
   $1 = nterm Z2 ()
-> $$ = nterm C2 ()
Stack now 0 53 51
Entering state 83
Reducing stack by rule 69 (line 68):
   $1 = nterm C1 ()
   $2 = nterm C2 ()
-> $$ = nterm Z11 ()
Stack now 0 53
Entering state 89
Reducing stack by rule 85 (line 76):
   $1 = nterm Z11 ()
```

```
-> $$ = nterm C3 ()
Stack now 0 53
Entering state 90
Reducing stack by rule 72 (line 71):
   $1 = nterm C4 ()
   $2 = nterm C3 ()
-> $$ = nterm Z14 ()
Stack now 0
Entering state 49
Reducing stack by rule 24 (line 43):
   $1 = nterm Z14 ()
-> $$ = nterm R10 ()
Stack now 0
Entering state 29
Reducing stack by rule 13 (line 31):
   $1 = nterm R10 ()
-> $$ = nterm productions ()
Stack now 0
Entering state 19
Reading a token: Next token is token NUMBEREND ()
Shifting token NUMBEREND ()
Entering state 55
Reducing stack by rule 2 (line 29):
   $1 = nterm productions ()
   $2 = token NUMBEREND ()


Production reduced successfully!
```

Test 4

Cadena: einein

Resultado: FAILURE

```
Starting parse
Entering state 0
Reading a token: einein
Next token is token EIN ()
Shifting token EIN ()
Entering state 1
Reducing stack by rule 26 (line 47):
   $1 = token EIN ()
-> $$ = nterm Z1 ()
Stack now 0
Entering state 31
Reading a token: Next token is token EIN ()
Reducing stack by rule 3 (line 31):
```

```
   $1 = nterm Z1 ()
-> $$ = nterm productions ()
Stack now 0
Entering state 19
Next token is token EIN ()


String contains error: syntax error

Error: popping nterm productions ()
Stack now 0
Cleanup: discarding lookahead token EIN ()
Stack now 0
```

---

6 Conclusión

En conclusión, se puede ver que esta gramática para formar números en alemán, aunque funciona con números grandes y complejos, no tiene la opción de escribir los números del 1 al 10. Cada cifra requiere un acompañante, lo que limita la cantidad de números que se pueden escribir en esta gramática. Se pretendió arreglar la gramática en nuestra implementación para que estos casos sean aceptados.

Sin embargo, la gramática logra simplificar de forma simple la forma de contar en alemán, la cual difiere mucho de la forma en inglés o español, por lo que se puede considerar una gramática exitosa pese a su fallo. Para un trabajo futuro, se podría investigar la forma de hacer una gramática que admita todos los números en alemán.