Universidade Federal de Viçosa
Campus Florestal
CCF 441 - Compiladores

Trabalho Prático 1 - Linguagem Grimório

Gabriel Moisés Monteiro Bonfim - EF04252
Marcos Biscotto de Oliveira - EF04236
Patrick Oliveira Corrêa de Araújo - EF04217
Tassia Martins Almeida Gomes - EF04247
Thulio Marcus Santos Silva - EF04223

Florestal
2024

Parte 1 ..... **4**
Introdução **4**
Tomadas de decisões **4**
Descrição geral da linguagem 4
Primeiras ideias 4
Obstáculos 5
Verbosidade 5
Palavras Reservadas 5
Utilização de espaços 5
Demarcador de fim de linha 6
Desenvolvimento **6**
Operador de Atribuição 6
Tipos primitivos 6
Int 7
Float 7
Boolean 8
Char 8
Void 8
Operadores Aritméticos 8
Operador de soma (+) e Subtração (-) 9
Operador de multiplicação (*) 9
Operador de divisão (/) 10
Operador de resto de divisão (%) 10
Operadores Relacionais 11
Maior (>) e Menor (<) 11
Maior ou igual (>=) e Menor ou igual (<=) 12
Igualdade (==) 12
Diferente (!=) 13
Operadores Lógicos 13
Ou (or) e E (and) 14
Hierarquia de blocos e separadores 14
Abertura e fechamento de blocos (_/ _) 14
Colchetes 15
Separador de parâmetros 15
Delimitador de string e char 15
Palavras reservadas 15
While 16
For 16
Break 17
If 17
Switch Case 17
Estruturas e Funções 18
Declarar função 19

Return....19
Definição de tipos....19
Operadores unários....20
Declaração de Array....21
Analisador Léxico....21
Definições regulares....22
Gramática....23
Exemplos de códigos em Grimório....30
Exemplo 1: Laço e Condicional....31
Exemplo 2: Conversão de Celsius e Fahrenheit....32
Exemplo 3: Funções....33
Parte 2....35
Atualização do Lex para análise sintática....35
Extensão de reconhecimento léxico....38
Análise Sintática....40
Derivações da Gramática....42
Tabela de Símbolos....50
Estruturas de Apoio....58
Function....59
Expression....59
Types....60
Operators....61
Análise Semântica....61
Implementações Pendentes....63
Conclusão....65
Referências....65

Parte 1

Introdução

Neste trabalho prático, desenvolvemos a linguagem de programação “Grimório”, cujo nome se baseia nos livros os quais magos poderosos escrevem suas magias, refletindo sua estrutura em verbos baseados nas falas utilizadas por esses magos enquanto conjuram feitiços, preparam poções ou realizam rituais. A Grimório foi projetada com um conjunto de tipos de dados primitivos, incluindo inteiros, floats, strings e booleans, com suporte para verificação estática de tipos, que será implementada na análise semântica em uma entrega posterior.

A linguagem adota o paradigma procedural, oferecendo comandos claros e intuitivos que permitem a construção de programas estruturados e eficientes. Essa documentação inclui uma descrição detalhada das palavras-chave e reservadas da linguagem, além da gramática, especificando variáveis, terminais (tokens) e padrões de lexema. Este trabalho visa fornecer uma base sólida para a compreensão e uso da Grimório, explorando aspectos técnicos e funcionais que tornam esta linguagem única.

Tomadas de decisões

Descrição geral da linguagem

A ideia da linguagem é simular a escrita de um mago de fantasia como se ao invés de um código, estivessem sendo preparadas magias em seu livro ou grimório. Com isso, a linguagem faz uso de termos desnecessariamente longos para tornar o código o mais textual e verboso possível, contendo jargões e termos relacionados à magia e fantasia no geral.

Primeiras ideias

A ideia surgiu pelo interesse dos membros do grupo em histórias de fantasia e jogos de RPG, a ideia inicial era transformar os comandos em feitiços e utilizar os jargões e termos comuns do gênero no lugar de palavras chave e comandos, deixando a estrutura do código menos simbólica e mais textual.

Obstáculos

Além de dúvidas sobre quais palavras utilizar para cada comando e organização geral do código.

Logo nas primeiras reuniões para especificação do que poderia ser implementado, alguns obstáculos iniciais para criação das regras para as expressões regulares surgiram:

- **Redução de símbolos:** A redução de símbolos abriu o questionamento de quais termos utilizar em seu lugar, assim como a indagação da possível criação de comandos de múltiplas palavras para criação de sentenças completas.
- **Tempo verbal:** Com a definição de palavras para cada operador, foi necessário encontrar uma forma de conectar o tempo verbal de cada palavra reservada para que as frases fizessem sentido.

Verbosidade

Todos os operadores lógicos e aritméticos foram substituídos por frases que resumem por extenso seus significados, desta forma eles podem ser utilizados entre variáveis ou valores para simular o funcionamento como em linguagens convencionais.

Palavras Reservadas

A linguagem contém palavras reservadas que não podem ser reutilizadas para variáveis. Além disto, cada palavra reservada da linguagem (exceto operadores aritméticos, lógicos e relacionais) começa com uma letra maiúscula para auxiliar na identificação.

Utilização de espaços

Para auxiliar na leitura e compreensão do que está sendo programado, para o reconhecimento correto dos lexemas é preciso haver um espaço em branco no final de cada comando para estimular a escrita de forma concisa, afinal o objetivo é

simular um texto. Desta forma, entre cada operador, variável ou palavra reservada é necessário a utilização de exatamente um espaço antes do próximo lexema.

Demarcador de fim de linha

Para que o compilador seja capaz de identificar o fim de uma sequência de comandos, foi especificado uma sequência de caracteres reservada para o fim de linha. Desta forma, deve-se utilizar "..." no fim de cada cadeia para indicar o fim de uma sentença. Os três pontos auxiliam na ênfase dramática de cada pensamento do mago (programador).

```
Fractal farenheit...
Fractal celsius...

Fractal value será 35...
```

Figura 1: Exemplo de fim de linha

Desenvolvimento

Operador de Atribuição

Na atribuição de valores a variáveis é necessário utilizar o operador de atribuição representado pela pela palavra reservada "será".

```
1   Glifo val1 será 10...
2   Glifo val2 será 15...
3   Glifo val3 será 20...
4   Glifo qnt será 3...
5   Glifo total...
6   Glifo contador...
```

Figura 2: Exemplo de atribuição

Tipos primitivos

Para cada tipo primitivo, foi definido na linguagem uma nova palavra para condensar seu significado. A tabela abaixo pode ser usada para consulta:

Tipo Primitivo	Grimorio
Int	Glifo
Long Int	Arquiglifo
Short Int	Semiglifo
Float	Fractal

Double Float	Arquifractal
Boolean	Axioma
Char	Runa
Void	Nulo

Tabela 1: Tabela de tipos primitivos

Int

O tipo primitivo inteiro é definido pela palavra reservada “Glifo”, que deve ser utilizada na declaração da variável para realizar a verificação de tipo, desta forma é possível declarar inteiros de duas formas:

- Declaração com atribuição de valor:

```
Glifo val1 será 10...
Glifo val2 será 80...
```

Figura 3: Exemplo de declaração de inteiro com atribuição de valor

- Declaração sem atribuição:

```
Glifo total...
Glifo contador...
```

Figura 4: Exemplo de declaração de inteiro sem atribuição de valor

Além do inteiro padrão, existe também uma definição para Long Int e Short Int, sendo possível declará-las utilizando respectivamente os termos “Arquiglifo” e “Semiglifo”.

Float

O tipo primitivo float é definido pela palavra reservada “Fractal”, que remete ao conceito de fratura ou parte não inteira.

```
Fractal farenheit...
Fractal celsius...

Fractal value será 35...
```

Figura 5: Exemplo de declaração de tipo flutuante

Além da definição do float padrão, é possível definir um Float Double por meio da palavra reservada “Arquifractal”.

Boolean

A palavra reservada para o tipo booleano é “Axioma”, remetendo a dualidade máxima representada pelo estado quintessencial da computação de verdadeiro ou falso. Desta forma, a linguagem possui duas palavras reservadas para remeter aos estados possíveis para as variáveis booleanas. Assim temos:

- Veritas: Do latim, representando o estado verdadeiro.
- Falsum: Também do latim, representando o estado falso.

```
Axioma verdadeiro será Veritas...
Axioma falso será Falsum...
Axioma talvez...
```

Figura 6: Exemplo de declaração de tipo booleano

Char

Para definição dos caracteres é utilizada a palavra reservada “Runa”, remetendo à utilização de símbolos capazes de conter significados utilizados por magos. Desta forma é possível realizar a declaração de um caractere com:

```
Runa primeiraLetra será "a"...
Runa ultimaLetra será "z"...
Runa algumaLetra...
```

Figura 7: Exemplo de declaração de tipo char

Void

Para a declaração do tipo vazio, é possível utilizar a palavra reservada “Nulo”.

Operadores Aritméticos

Os operadores aritméticos foram inteiramente substituídos por termos reservados que os representam, colaborando para a estruturação textual da linguagem, abaixo estão suas conversões:

Aritmético	Grimório
(+)	fundido a
(-)	dissolvido de
(*)	replicado de
(/)	fragmentado em
(%)	transmoglifado a

Tabela 2: Tabela de operadores aritméticos

Operador de soma (+) e Subtração (-)

Os operadores de soma e subtração foram substituídos pelo conceito de fusão ou junção e extração ou dissolução respectivamente. Desta forma, simulando uma magia feita para unir duas metades ou dividir algo em duas partes, desta forma seus termos são “fundido a ” e “dissolvido de ”. A definição regular para os operadores são:

```
soma fundido[ ]a[ ]
subtracao dissolvido[ ]de[ ]
```

Figura 8: Definição regular de operadores de soma e subtração

A utilização destes operadores na linguagem fica da seguinte forma:

```
Glifo fragmento1 será 5...
Glifo fragmento2 será 10...

Glifo resultado será fragmento1 fundido a fragmento2...


Glifo monolito1 será 20...
Glifo monolito2 será 10...

Glifo reducao será monolito1 dissolvido de monolito2...
```

Figura 9: Exemplo de uso de operadores de soma e subtração

Operador de multiplicação (*)

O operador de multiplicação é representado pelo conceito de replicação, onde um mago cria um número definido de cópias de algo. Desta forma, o termo reservado é “replicado de ”, representado pela definição regular:

```
multiplicacao replicado[ ]de[ ]
```

Figura 10: Definição regular do operador de multiplicação

Sua utilização na linguagem pode ser obtida através de:

```
Glifo original será 10...
Glifo numReplicas será 5...

Glifo total será original replicado de numReplicas...
```

Figura 11: Exemplo de uso do operador de multiplicação

Operador de divisão (/)

Em oposição ao conceito de multiplicação, a divisão é representada pelo poder do mago de fragmentar algo em um número definido de partes. Assim, o termo reservado é “fragmentado de ”, com a seguinte definição regular:

```
divisao fragmentado[ ]em[ ]
```

Figura 12: Definição regular do operador de divisão

Na linguagem, o uso do termo pode ser exemplificado por:

```
Glifo monolito será 40...
Glifo partes será 4...

Glifo resultado será monolito fragmentado em partes...
```

Figura 13: Exemplo de uso do operador de divisão

Operador de resto de divisão (%)

Em adição ao conceito de divisão, existe também o resto da fragmentação, que é representado pelo que resta das magias destrutivas do mago. Assim, existe o termo reservado “transmoglifado a ” que diz respeito ao resto da divisão, representado pela definição regular:

```
mod transmoglifado[ ]a[ ]
```

Figura 14: Definição regular do operador de resto de divisão

Na linguagem é possível utilizar o termo como:

```
Glifo monolito será 20...
Glifo partes será 4...
Glifo restante...
Se monolito transmoglifado a 4 for equivalente a 0 _/
    Axioma veredito será Veritas...
\_
```

Figura 15: Exemplo de uso do operador de resto de divisão

Operadores Relacionais

O funcionamento geral dos operadores relacionais mantém a forma de utilização de seus respectivos predecessores de linguagens convencionais, sendo essa a de utilização entre dois valores.

Relacional	Grimório
(>)	for superior a
(<)	for inferior a
(>=)	for superequivalente a
(<=)	for infraequivalente a
(==)	for equivalente a
(!=)	for distinto de

Tabela 3: Tabela de operadores relacionais

Maior (>) e Menor (<)

Para o operador relacional de comparação (>) e (<), foi escolhido o lexema “for superior a” e “for inferior a” respectivamente, com exatamente um espaço entre cada uma das palavras. A definição regular para os lexemas é:

```
maior for[ ]superior[ ]a[ ]
menor for[ ]inferior[ ]a[ ]
```

Figura 16: Definição regular do operadores de maior e menor

A ideia destes operadores é assim como o caractere, ser utilizado como intermédio entre dois valores, exemplo:

```
Se 10 for superior a 5 _/
    Glifo num será 10...
\_

Se 15 for inferior a 30 _/
    Glifo num será 15...
\_
```

Figura 17: Exemplo de uso do operadores de maior e menor

Maior ou igual (>=) e Menor ou igual (<=)

Análogos aos operadores de maior ou menor, os operadores de (>=) e (<=) são reservados para as sentenças “for superequivalente a” e “for infraequivalente a” respectivamente, com exatamente um espaço entre cada uma das palavras. A definição regular para os lexemas é:

```
maior_igual for[ ]superequivalente[ ]a[ ]
menor_igual for[ ]infraequivalente[ ]a[ ]
```

Figura 18: Definição regular do operadores de maior ou igual e menor ou igual

Na linguagem, é possível utilizar estes operadores da seguinte forma:

```
Se 10 for superequivalente a 5 _/
    Glifo num será 5...
\_

Se 15 for infraequivalente a 30 _/
    Glifo num será 30...
\_
```

Figura 19: Exemplo de uso do operadores de maior ou igual e menor ou igual

Igualdade (==)

O operador de igualdade pode ser representado pelo termo reservado “for equivalente a”, assim como os demais operadores lógicos, basta utilizá-lo entre dois valores. Sua definição regular é:

```
igualdade for[ ]equivalente[ ]a[ ]
```

Figura 20: Definição regular do operadores de igualdade

Na linguagem, o operador age da seguinte forma:

```
Se variavel1 for equivalente a variavel2 _/
    Axioma booleano será Veritas...
\_

Se 15 for equivalente a 30 _/
    Axioma booleano será Falsum...
\_
```

Figura 21: Exemplo de uso do operadores de igualdade

Diferente (!=)

O operador de diferença pode ser representado pelo termo reservado “for distinto de”, assim como os demais operadores lógicos, basta utilizá-lo entre dois valores. Sua definição regular é:

```
diferente distinto[ ]de[ ]
```

Figura 22: Definição regular do operadores de diferença

Na linguagem o operador pode ser usado como:

```
Se variavel1 for distinto de variavel2 _/
    Axioma booleano será Falsum...
\_
```

Figura 23: Exemplo de uso do operadores de diferença

Operadores Lógicos

Análogo a implementação dos operadores relacionais, os operadores lógicos são sentenças feitas para serem utilizadas entre dois valores, com exceção da negação que se comporta como operador unário.

Lógicos	Grimório
(or)	ou então
(and)	assim como
(not)	contrariando

Tabela 4: Tabela de operadores lógicos

Ou (or) e E (and)

Os operadores lógicos (or) e (and) são representados por sentenças que podem ser utilizadas da mesma forma que seus análogos contrapontos em linguagens convencionais. Suas definições regulares são:

```
ou ou[ ]entao[ ]
e assim[ ]como[ ]
```

Figura 24: Definição regular do operadores Ou e E

Na linguagem, eles podem ser usados da seguinte forma:

```
Se variavel1 for equivalente a 10 assim como variavel 2 for equivalente a 30 _/
    Axioma booleano será Veritas...
\_
```

Figura 25: Exemplo de uso do operadores Ou e E

Hierarquia de blocos e separadores

Para manter a organização e boa visibilidade do código, nem todos os símbolos foram removidos, mantendo assim algumas formas de dividir blocos de comandos em certas situações, como por exemplo declaração de funções, IF's e estruturas de repetição. Abaixo é possível ver as definições regulares destes lexemas.

```
/* hierarquias e blocos */
abre_bloco \_\/
fecha_bloco \\\_
abre \[
fecha \]
divisor \,
delimitadorChar [\" | \"]
```

Figura 26: Definição regular de hierarquia de blocos e separadores

Abertura e fechamento de blocos (_/ _)

No lugar de chaves, o separador de blocos na linguagem Grimório é representado pelas mãos do mago conjurando seus feitiços, indicado pelos símbolos (_/) na abertura do bloco e (_) no fechamento. Assim, é possível utilizá-los como:

```
Se variavel1 for distinto de variavel2 _/
    Axioma booleano será Falsum...
\_
```

Figura 27: Exemplo de uso de fechamento de blocos

Colchetes

Na linguagem, os caracteres utilizados para delimitar prioridade de operações são os colchetes, que cumprem o papel de separar e ordenar a execução de operações aritméticas ou parâmetros passados para funções, eles podem ser usados da seguinte forma:

```
Glifo total será [10 fundido a 15] replicado de 8...
```

Figura 28: Exemplo de uso de colchetes

Separador de parâmetros

Para separar os parâmetros passados para funções, o operador utilizado é a vírgula, já que o símbolo por si só possui um caráter textual, sua utilização colabora para a aparência geral do código. É possível utilizá-las da seguinte forma:

```
Preparar Glifo SomaNum, Componentes: Glifo num1, Glifo num2 _/
    Glifo resultado será num1 fundido a num2...
    Regressus resultado...
\_
```

Figura 29: Exemplo de uso de separador de parâmetros

Delimitador de string e char

Para definir um símbolo com caractere ou string é necessário utilizar as aspas (“ ”) antes e depois da sequência de símbolos para especificar que ela deve ser interpretada como tal. Assim como a vírgula, as aspas já possuem um caráter textual e por isso seu uso foi mantido.

```
Runa char1 será "A"...
Runa char2 será "Z"...
```

Figura 30: Exemplo de uso de delimitador de char

Palavras reservadas

Palavra Reservada	Grimório
while	Dilatar tempo enquanto
for	Fraturar tempo se
if	Ponderar se

else	Ou optar por
elif	Ou optar por ponderar se
switch	Iniciar profecia de
case	Profetizar que se
default	Cumprir profecia
return	Regressus
goto	Translocar para

Tabela 5: Tabela de palavras reservadas

While

Para a construção do comando, a inspiração partiu da capacidade do mago de controlar o tempo como quer. Desta forma, a construção do laço de repetição na linguagem é feito com o uso do termo reservado “Lapso temporal enquanto”, que inicia o comando e espera como parâmetro a condição de parada para encerrar o laço antes da abertura do bloco.

```
Dilatar tempo enquanto i for inferior a 10 _/
  Ponderar val1 transmoglifado a 2 equivalente a 0
  _/
   val1 será val1 fundido a val2...
  \_
  i será i fundido a 1...
\_
```

Figura 31: Exemplo de uso de while

For

Para a definição do comando, a ideia é a capacidade do mago de também de controlar o tempo de maneira mais controlada, ditando quantas vezes o tempo irá se repetir. Assim, para construir um for pode-se usar o termo reservado “Fraturar tempo se” seguido dos parâmetros do laço, como no seguinte exemplo:

```
Glifo contador será 0...

Fraturar tempo se i for inferior a 10, Encantar i, Glifo i será 0 _/
    Encantar contador...
\_
```

Figura 32: Exemplo de uso de for

Break

A ideia do termo utilizado para o break é a capacidade do mago de decidir como dada distorção no tempo irá se encerrar, desta forma é possível utilizar o termo “Que assim seja” para encerrar o while ou for.

```
Dilatar tempo enquanto i for inferior a 10 _/
  Ponderar val1 transmoglifado a 2 equivalente a 0
  _/
    val1 será val1 fundido a val2...
    Que assim seja...
  \_
```

Figura 32: Exemplo de uso de break

If

O bloco condicional usa o conceito de ponderar, onde o mago irá utilizar sua magia para decidir o destino de dada constatação. Assim, é feito o uso da palavra reservada “Ponderar se” seguido da condição para que o bloco seja instanciado. Desta forma, é possível declarar a funcionalidade como:

```
Glifo pergaminho1 será 15...
Glifo pergaminho2 será 5...
Axioma veredito...

Ponderar se pergaminho1 for superior a pergaminho2 _/
    veredito será Veritas...
\_
```

Figura 33: Exemplo de uso de if

Switch Case

Para o bloco de casos, foi utilizado o conceito de profecia usado pelos magos, onde se anunciam as possibilidades para dada ideia. Desta forma, em nossa linguagem é possível criar um bloco condicional com a sequência “Profetizar sob” que será seguido pela variável do case. Desta forma, é possível instanciar os casos com:

```
Iniciar Profecia de val _/
Profetizar que se 1;

    que assim seja...
Profetizar que se 2;

    que assim seja...
cumprir profecia;

    que assim seja...
\_
```

Figura 34: Exemplo de uso de switch case

Cada caso é instanciado dentro do bloco da profecia com o termo reservado “Profetizar que se” seguido pelos operadores lógicos do caso. Por fim, o caso default é definido por “Cumprir profecia”.

Estruturas e Funções

Assim como em outras linguagens, é possível declarar funções para realizar determinadas funcionalidades no código. Em nosso caso, estas funções ou métodos são chamadas de magias, sendo elas criações especiais do mago para situações específicas. O mago precisa, primeiramente, especificar que está preparando uma magia, em seguida especificando os componentes para tal feitiço antes de explicar seu funcionamento.

Funções e estruturas	Grimório
typedef	Transmutar
struct	Componentização em
union	Conjuntura em
unum	Glifos em
Declarar Função	Preparar magia de
Chamar Função	Conjurar
Passagem de parâmetros	Componentes:
return	Regressus
const	Imutável
volatile	Volátil

Tabela 6: Tabela de funções e estruturas

Declarar função

Primeiramente, para criar uma magia, é necessário utilizar o lexema “Preparar magia de” antes de definir o tipo que será retornado. Com o tipo definido, deve-se especificar o nome que a magia terá. Em sequência são especificados os componentes. Exemplo:

```
Preparar magia de Glifo SomaNum, Componentes: Glifo num1, Glifo num2 _/
    Glifo resultado será num1 fundido a num2...
    Regressus resultado...
\_

Preparar magia de Glifo DissolveNum, Componentes: Glifo num1, Glifo num2 _/
    Glifo resultado será num1 dissolvido de num2...
    Regressus resultado...
\_
```

Figura 35: Exemplo de declaração de função

Abaixo pode-se consultar cada uma das definições regulares relacionadas:

```
declFunc Preparar[ ]magia[ ]de[ ]
chamarFunc Conjurar[ ]
parametros Componentes:[ ]
return Regressus[ ]
```

Figura 36: Exemplo de definição regular de declaração de função

Return

Para retornar valores em funções, é possível utilizar a palavra mágica reservada “Regressus”, do latim, retornar. Assim, é possível declarar o valor ou variável que se deseja obter como retorno em uma função.

```
Preparar magia de Glifo add, Componentes: Glifo a, Glifo b _/
    Regressus a fundido a b...
\_

Glifo valor1 será 100...
Glifo valor2 será 50...
Glifo resultado...

resultado será add[valor1, valor2]...
```

Figura 37: Exemplo de uso de return

Definição de tipos

Para auxiliar o mago, é possível criar novos tipos através de poderes de transmutação, agregando múltiplos componentes em algo novo. Em outras

palavras, é possível utilizar um comando análogo ao “typedef” da linguagem C para se construir estruturas. A palavra reservada para começar a construção é “Transmutar” seguido pelo termo que indica o tipo necessário. O exemplo abaixo mostra como esta declaração pode ser feita:

```
Transmutar componentização em Golem _/
  Glifo tamanho...
  Glifo poder...
  Fractal altura...
  Runa nome...
\_
```

Figura 38: Exemplo de definição de um novo tipo

Para acessar valores internos aos tipos é possível utilizar os símbolos reservados “>” para acesso direto ou “>>” para acesso via ponteiros.

```
Transmutar componentização em Golem _/
  Glifo tamanho...
  Glifo poder...
  Fractal altura...
  Runa nome...
\_

golem golem1...
golem1>tamanho será 10...
golem1>poder será 5...
golem1>altura será 10.2...
golem>nome será "A"...
```

Figura 39: Exemplo de acesso de valores dentro do novo tipo

Operadores unários

Para que o mago consiga com facilidade acessar a essência mais profunda de seus poderes, ele pode fazer uso dos operadores de ponteiro. Além disso, também há um operador unário de soma e subtração para auxílio da definição do for.

Unários	Grimório
(++)	Encantar
(--)	Desencantar
(&)	Alma de
(*)	Lembrança de

Tabela 7: Tabela de operadores unários

A definição regular dos operadores unários pode ser vista abaixo:

```
/* Unários */
adicionarUm Encantar[ ]
subtrairUm Desencantar[ ]
derreferenciar Alma[ ]de[ ]
referenciar Lembrança[ ]de[ ]
```

Figura 40: Exemplo de definição regular dos operadores unários

Declaração de Array

Caso o mago sinta a necessidade de criar uma coletânea de tipos primitivos ou estruturas, ele pode utilizar uma magia especial. Desta forma, é possível anunciar a criação de um array com a sequência de símbolos reservada “-|” e “-|”. Para visualização de como tal declaração é feita, pode se consultar o exemplo abaixo.

```
Glifo coletanea-|10|-...

Glifo i será 0...

Fraturar tempo se i for inferior a 10, Encantar i _/
    coletanea-|i|- será i fundido a 1...
\_
```

Figura 41: Exemplo de declaração de arrays

No exemplo acima, o vetor é declarado com suas posições nulas, sendo posteriormente preenchido com o laço. Caso seja necessário criar um vetor já com valores preenchidos, é possível passar como parâmetros valores para as posições se utilizando o símbolo “|”.

```
Glifo coletanea-|5|- será | 1, 2, 3, 4, 5 |...

Glifo i será 0...
```

Figura 42: Exemplo de declaração de arrays com passagem de parâmetros

A definição regular das regras de vetor podem ser vistas abaixo.

```
abrearray \-\|
fechaarray \|\-
definidorarray \|
```

Figura 43: Exemplo de definição regular de arrays

Analisador Léxico

Para o reconhecimento da linguagem, o primeiro passo foi a criação do analisador capaz de reconhecer um arquivo fonte e gerar o fluxo de tokens. Neste trabalho

estão presentes duas versões stand-alone do analisador, sendo a diferença entre elas apenas a maneira como o terminal exibe os lexemas.

- A primeira versão **lex.l** utiliza as expressões regulares para imprimir no terminal linha a linha cada lexema identificado no arquivo de entrada.
- A versão **lex_color.l** utiliza as expressões regulares para imprimir no terminal o código lido na entrada alterando a cor de cada lexema encontrado para um padrão pré-determinado que será explicado abaixo. A ideia deste arquivo é indicar de forma mais visual o que cada palavra na entrada está representando.

Definições regulares

Primeiramente, algumas definições básicas foram criadas para serem utilizadas posteriormente em outras definições ao longo do analisador. Dentre estas definições básicas temos o lexema de fim de linha, leitura de dígitos, caracteres, números positivos e negativos inteiros e flutuantes.

```
end_line  (\.\.\.)

/* identificação */
digito [0-9]
caractere [a-zA-Z]
digito_positivo [+]?{digito}+
digito_negativo [-]{digito}+
float_negat [-]?{digito}+\.{digito}+
float_posit [+]?{digito}+\.{digito}+
palavra [a-zA-Z]+
```

Figura 44: Definições regulares de identificação

Operadores aritméticos possuem como padrão palavras separadas com exatamente um espaço entre elas, contendo apenas palavras compostas por letras minúsculas. A escolha de utilizar todas as letras minúsculas parte do pressuposto que operadores matemáticos são utilizados entre valores, desta forma nunca estando presentes no início de uma linha de código. Assim, para auxiliar no visual textual da linguagem não há necessidade de se haver letras maiúsculas.

```
/* aritmeticos */
soma fundido[ ]a[ ]
subtracao dissolvido[ ]de[ ]
multiplicacao replicado[ ]de[ ]
divisao fragmentado[ ]em[ ]
mod transmoglifado[ ]a[ ]
```

Figura 45: Definições regulares de operadores aritméticos

Operadores lógicos seguem o mesmo padrão de exatamente um espaço em branco entre as palavras e sem letras maiúsculas.

```
/* lógicos */
maior for[ ]superior[ ]a[ ]
menor for[ ]inferior[ ]a[ ]
maior_igual for[ ]superequivalente[ ]a[ ]
menor_igual for[ ]infraequivalente[ ]a[ ]
ou ou[ ]entao[ ]
e assim[ ]como[ ]
igualdade for[ ]equivalente[ ]a[ ]
diferente for[ ]distinto[ ]de[ ]
atribuicao será[ ]
```

Figura 46: Definições regulares de operadores lógicos

Os tipos primitivos são declarados com suas respectivas palavras reservadas iniciadas sempre com letra maiúscula, seguidos por exatamente um espaço.

```
/* Tipos Primitivos */
tipoInt Glifo[ ]
tipoLongInt Arquiglifo[ ]
tipoShortInt Semiglifo[ ]
tipoFloat Fractal[ ]
tipoDouble Arquifractal[ ]
tipoBool Axioma[ ]
tipoChar Runa[ ]
tipoVoid Nulo[ ]
```

Figura 47: Definições regulares de tipos primitivos

Gramática

A partir das definições regulares e da análise léxica já criadas, com a estrutura da linguagem bem definida, foi elaborada uma gramática que demonstra a estrutura sintática que os magos devem seguir. A gramática de Grimório é demonstrada a seguir:

```C/C++
START                ::=  { DECL-STMT | FUNCTION-DECL | IMPORTS}
```

```
IMPORTS             ::=  { 'Tutorar ' STR-CONST '...' }

VARIABLE             ::=  ID [ ACCESSES ] { ( '>' VARIABLE ) | ( '>>'
VARIABLE) }

FUNCTION-DECL                 ::=   'Preparar magia de ' TYPE ID ',
componentes:' ARGS STMT-BLOCK

FUNCTION-SIGNATURE     ::=     'Preparar  magia  de  '  TYPE  ID  ',
componentes:' ARGS '...'

FUNCTION-CALL       ::=  'Conjurar ' ID '[' PARAMS ']'

ARGS                ::=  [ ARG { ',' ARG } ]

ARG                 ::=  TYPE ID

PARAMS              ::=  [ PARAM { ',' PARAM } ]

PARAM               ::=  EXPR

STMT-BLOCK          ::=  '_/' STMTS '\_'

STMTS               ::=  { STMT }

STMT                ::=   NULL-STMT

                        |   RETURN-STMT

                        |   CONTINUE-STMT

                        |   BREAK-STMT

                        |   IF-STMT

                        |   WHILE-STMT

                        |   FOR-STMT

                        |   GOTO-STMT

                        |   SWITCH-STMT

                        |   CASE-STMT

                        |   DECL-STMT

NULL-STMT           ::=  '...'

RETURN-STMT         ::=  'Regressus ' "expr" '...'

CONTINUE-STMT       ::=  'Prossiga' '...'

BREAK-STMT          ::=  'Que assim seja' '...'
```

```
IF-STMT             ::=  'Ponderar se ' "expr" '...' STMT-BLOCK { 'Ou
optar por ' [ 'Ponderar se' "expr" '...' ] STMT-BLOCK }

WHILE-STMT              ::=   'Dilatar tempo enquanto ' "expr" '...'
STMT-BLOCK

FOR-STMT            ::=  'Fraturar tempo se ' "expr" ',' "expr" '...'
STMT-BLOCK

GOTO-STMT           ::=  'Translocar para ' ID '...'

SWITCH-STMT               ::=   'Iniciar profecia de ' "expr" '...'
SWITCH-BLOCK

SWITCH-BLOCK        ::=  '_/' SWITCH-STMTS '\_'

SWITCH-STMTS          ::=   { CASE CASE-STMT } [ 'Cumprir profecia;'
CASE-STMT]

CASE                ::=  'Profetizar que se  ' "expr" ';'

CASE-STMT           ::=  { STMT } BREAK-STMT

DECL-STMT           ::= TYPE-DEFINITION '...'

                      |  ASSIGNMENTS '...'

                      |  FUNCTION-SIGNATURE '...'

PLUS                ::= 'fundido a '

MINUS               ::= 'dissolvido de '

TIMES               ::= 'replicado por '

DIV                 ::= 'fragmentado em '

MOD                 ::= 'transmoglifado por '

GT                  ::= 'for superior a '

LT                  ::= 'for inferior a '

GE                  ::= 'for superequivalente a '

LE                  ::= 'for infraequivalente a '

EQ                  ::= 'for equivalente a '

NE                  ::= 'for distinto de '

AND                 ::= 'assim como '

OR                  ::= 'ou entao '
```

```
NOT                 ::= 'contrariando '
EXPR                ::=  "term" { OPERATOR "term" }
                      |  '|' { "expr" } '|'
TERM                ::=  FACTOR { OPERATOR FACTOR }
FACTOR              ::=  '[' "expr" ']'
                      |  EXPR
                      |  VARIABLE
                      |  INTEGER
                      |  RATIONAL
                      |  BOOL
                      |  LITERAL
                      |  FUNCTION-CALL
                      |  TERNARY-EXPR
                      |  UNARY-EXPR
OPERATOR            ::=  ARIT | REL | LOGIC
ARIT                ::=  PLUS
                      |  MINUS
                      |  TIMES
                      |  DIV
                      |  MOD
REL                 ::=  GT
                      |  LT
                      |  GE
                      |  LE
                      |  EQ
                      |  NE
LOGIC               ::=  AND
                      |  OR
```

```
                        |  NOT
TERNARY-EXPR          ::=  "expr" ' ? ' "term" ' > < ' TERM
UNARY-EXPR            ::= UNARY-POS-MINUS
                        |  UNARY-POS-SUM
                        |  UNARY-DEREF
                        |  UNARY-REF
                        |  UNARY-NOT
UNARY-POS-MINUS           ::= 'desencantar ' VARIABLE
UNARY-POS-SUM       ::= 'encantar' VARIABLE
UNARY-DEREF         ::= 'alma de ' VARIABLE
UNARY-REF                 ::= 'lembranca de ' VARIABLE
UNARY-NOT                 ::= 'contrariando ' VARIABLE
TYPE                  ::=  'Runa'
                        |  'Glifo'
                        |  'Arquiglifo'
                        |  'Semiglifo'
                        |  'Fractal'
                        |  'Arquifractal'
                        |  'SemiFractal'
                        |  'Nulo'
                        |  'Axioma'
                        |  ENUM-DEFINITION
                        |  STRUCT-DEFINITION
                        |  UNION-DEFINITION
                        |  TYPE-ENUM
                        |  TYPE-STRUCT
                        |  TYPE-UNION
TYPE-QUALIFIER        ::=  'imutavel '
```

```
                      |  'volatil '

ENUM-DEFINITION     ::=  'glifos' ' em ' ID '_/' ENUM-FIELDS '\_'

ENUM-FIELDS         ::=  ENUM-FIELD { ',' ENUM-FIELD }

ENUM-FIELD          ::=  ID [ 'sera ' DEC-DIGIT ]

STRUCT-DEFINITION     ::=   'componentizacao'  '  em  '  [  ID  ]  '_/'
STRUCT-FIELDS '\_'

STRUCT-FIELDS       ::=  STRUCT-FIELD { '...' STRUCT-FIELD }

STRUCT-FIELD        ::=  TYPE-DECL ID

UNION-DEFINITION      ::=   'conjuntura'  '  em  ' ID '_/' UNION-FIELDS
'\_'

UNION-FIELDS        ::=  UNION-FIELD { '...' UNION-FIELD }

UNION-FIELD         ::=  TYPE-DECL ID

TYPE-DEFINITION     ::=  'Transmutar ' TYPE ID

TYPE-ENUM           ::=  'glifos' ID

TYPE-STRUCT         ::=  'componentizacao' ID

TYPE-UNION          ::=  'conjuntura' ID

TYPE-DECL           ::=  [ 'essencia de' ] TYPE [ TYPE-QUALIFIER ]

ASSIGNMENTS         ::=  ASSIGNMENT { ', ' ASSIGNMENT }

ASSIGNMENT          ::=  [ TYPE-DECL ] VARIABLE [ 'sera' "expr" ]

ACCESSES            ::=  { ACCESS }

ACCESS              ::=  '-|' "expr" '|-'


NON-ZERO              ::=  '1' | '2' | '3' | '4' | '5' | '6' | '7' |
'8' | '9'

DIGIT               ::=  '0' | NON-ZERO

DEC-DIGIT           ::=  DIGIT { DIGIT }

INTEGER             ::=  DEC-DIGIT

RATIONAL            ::=  DEC-DIGIT '.' DEC-DIGIT
```

```
LOWER-CASE          ::=  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's'
| 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

UPPER-CASE          ::=  'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S'
| 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

SPECIAL-CHAR        ::=  '!' | '@' | '#' | '$' | '%' | '^' | '&' |
'*' | '(' | ')' | '-' | '_' | '+' | '=' | '{' | '}' | '[' | ']' | '|'
| '\' | ':' | ';' | "'" | '<' | '>' | ',' | '.' | '?' | '/' | '~' |
'`'

ACCENT              ::=  'á' | 'é' | 'í' | 'ó' | 'ú' | 'â' | 'ê' |
'î' | 'ô' | 'û' | 'ã' | 'õ' | 'ç'

LETTER              ::=  LOWER-CASE | UPPER-CASE | ACCENT

CHAR                ::=  LETTER | DIGIT | SPECIAL-CHAR

ID                  ::=  ( LETTER | '_' ) { LETTER | DIGIT | '_' }

TRUE                ::=  'veritas'

FALSE               ::=  'falsum'

BOOL                ::=  TRUE | FALSE

NULL-CONST          ::=  'Vazio'

STR-CONST           ::=  '"' { CHAR } '"'

CHAR-CONST          ::=  "'" CHAR "'"

LITERAL             ::=  STR-CONST | CHAR-CONST | NULL-CONST
```

A criação da gramática foi feita seguindo a notação EBNF (Extended Backus-Naur form), muito utilizada para expressar gramáticas de linguagens de programação. Nesse contexto, o símbolo “::=” representa a atribuição das possíveis produções, ao lado direito, à regra, ao lado esquerdo. A utilização das chaves indica que o conteúdo pode se repetir zero ou mais vezes (como um fecho de Kleene). O uso de colchetes denota a utilização opcional, zero ou uma vez. Os parênteses simbolizam o agrupamento.

O primeiro passo para a criação da gramática foi determinar a variável de partida, definida como “START”, é dessa variável que partem as demais produções. Na variável de partida, podemos ver que o programa pode derivar em zero ou mais

declarações, tanto de funções quanto demais tipos, ou em importações de bibliotecas.

As declarações de função seguem o formato demonstrado em “FUNCTION-DECL”. A produção dessa variável envolve as demais variáveis “TYPE” (todos os tipos possíveis, incluindo tipos criados pelo usuário), “ID” (identificador para a função), “ARGS” (parâmetros exigidos pela função) e “STMT-BLOCK” (bloco onde são descritos os comandos executados pela função).

As demais declarações, definidas por “DECL-STMT”, consistem em “TYPE-DEFINITION” (definição de um novo tipo, com struct, union ou enum), “ASSIGNMENTS” (uma ou mais atribuições subsequentes, separadas por vírgula) e “FUNCTION-SIGNATURE” (cabeçalho de função).

Dessa forma, o programa pode ter inúmeras funções e declarações, e dentro de cada função, no seu bloco de execução, podem ser produzidos os comandos: “NULL-STMT”, “RETURN-STMT”, “CONTINUE-STMT”, “BREAK-STMT”, “IF-STMT”, “WHILE-STMT”, “FOR-STMT”, “GOTO-STMT”, “SWITCH-STMT” e “CASE-STMT”, além da possibilidade de derivar em outro “DECL-STMT”. Todos os comandos possíveis e suas produções podem ser facilmente observados na gramática.

A partir dos comandos, podem ser geradas expressões, derivadas da variável “EXPR”. Por essa derivação, podem ser criadas mais expressões com termos em cadeia, envolvendo “VARIABLE”, “INTEGER”, “RATIONAL”, “BOOL”, “LITERAL”, “FUNCTION-CALL”, “TERNARY-EXPR” e “UNARY-EXPR”. Além disso, as expressões envolvem operadores, a partir de “OPERATOR”, que deriva em “ARIT” (aritméticos), “REL” (relacionais) e “LOGIC” (lógicos).

Exemplos de códigos em Grimório

Para mostrar em prática o funcionamento dos comandos criados, foram criados exemplos de código em C com suas respectivas conversões para a linguagem Grimório. Além disso, será mostrado o código para a mesma, reconhecido pelo analisador léxico presente no arquivo **lex_color.l**.

Primeiramente, é preciso consultar os tópicos abaixo para entender o padrão de cores detectados, as cores para cada tipo de token seguem o seguinte padrão:

- **Valores inteiros, flutuantes e valores booleanos:** Roxo.
- **Tipos primitivos:** Verde.
- **Palavras reservadas para blocos e funções:** Amarelo
- **Operadores aritméticos e unários:** Azul.
- **Operadores lógicos:** Ciano.
- **Variáveis e nomes de função:** Vermelho.
- **Fim de linha:** Representado pela mensagem (fim de linha).
- **Abertura e fechamento de blocos:** Mensagem (abre bloco _/) ou (fecha bloco _).

Exemplo 1: Laço e Condicional

O código abaixo é uma implementação de um laço simples, onde três variáveis val1, val2 e val3 são utilizadas um número de vezes igual a variável qnt. Neste loop caso a iteração seja par o valor de val1 será seu próprio valor somado a val2, caso seja uma iteração ímpar será val1 somado a val3.

A diferença chave entre o código padrão em C e na linguagem Grimório está na forma como o laço é instanciado, é possível notar que na nova linguagem é necessário instanciar a variável que será incrementada com o laço (variável i) antes de declará-lo.

```
int val1 = 10;
int val2 = 15;
int val3 = 20;
int qnt = 3;

for (short i = 0; i < qnt; i++) {
  if(val1 % 2 == 0) {
    val1 = val1 + val2;
  } else {
    val1 = val1 + val3;
  }
}
```

```
Glifo val1 será 10...
Glifo val2 será 15...
Glifo val3 será 20...
Glifo qnt será 3...

Glifo i será 0...

Dilatar tempo enquanto i for inferior a qnt, Encantar i _/
    Ponderar se val1 transmoglifado a 2 equivalente a 0
  _/
   val1 será val1 fundido a val2...
  \_
  Ou optar por
  _/
    val1 será val1 fundido a val3...
  \_
\_
```

Figuras 48 e 49: Diferença no código entre as linguagens C e Grimório

Acima é possível observar o mesmo trecho de código nas duas linguagens. Utilizando o analisador léxico para dado código é possível obter a seguinte saída:

```
Glifo  val1 será  10 (fim de linha)
Glifo  val2 se a 15 (fim de linha)
Glifo  val3 será  20 (fim de linha)
Glifo  qnt será  3 (fim de linha)
Glifo  i será  0 (fim de linha)
Dilatar tempo enquanto  i for inferior a  qnt , Encantar  i
abre bloco _/
Ponderar se  val1 transmoglifado a 2 equivalente a  0
abre bloco _/
val1 será  val1 fundido a  val2 (fim de linha)
fecha bloco \_
Ou optar por  val1 transmoglifado por  2 equivalente a  1
abre bloco _/
val1 será  val1 fundido a  val3 (fim de linha)
fecha bloco \_
i será  i fundido a  1 (fim de linha)
fecha bloco \_
```

Figura 51: Resultado do código em Grimório quando passado no analisador léxico

Na versão colorida, podemos analisar passo a passo o tipo de token que cada palavra está gerando. Nas primeiras linhas é possível ver em verde os tipos Glifo (inteiro, em verde) sendo estaticamente definidos para cada uma das variáveis (em vermelho). Em seguida, é possível observar o lexema de atribuição (será, em azul) antes dos valores que receberão, denotados pelos valores em roxo. É possível notar também a mensagem (fim de linha) seguido por uma quebra de linha sempre que o lexema (...) de fim de linha é encontrado, dando maior legibilidade para a saída.

Exemplo 2: Conversão de Celsius e Fahrenheit

O código abaixo é uma conversão simples de valor em duas medidas de temperatura, o valor 35 é passado como parâmetro para duas funções que o converterão de uma medida para a outra. A ideia principal deste código é mostrar como a organização de múltiplas operações aritméticas em uma mesma linha funcionam na linguagem Grimório.

```
double farenheit;
double celsius;

float value = 35;

celsius = (value - 32) * (5/9);
farenheit = (value * 9/5) + 32;
```

Figura 52: Código de conversão de Celsius e Fahrenheit em C

```
Arquifractal farenheit...
Arquifractal celsius...

Fractal value será 35...

celsius será [value extraido de 32] replicado por [5 fragmentado em 9]...
farenheit será [value replicado de 9 fragmentado em 5] fundido a 32...
```

Figura 53: Código de conversão de Celsius e Fahrenheit em Grimório

É possível notar que os símbolos se tornam bem mais extensos, o uso de separadores (que na linguagem são colchetes) também se mostram necessários. Vale ressaltar que tornar os comandos mais verbosos está entre os interesses principais da linguagem, uma vez que, os magos nunca escrevem demais, sempre apenas o necessário. Usando o analisador léxico no código acima, é possível obter a seguinte saída:

```
Arquifractal  farenheit (fim de linha)
Arquifractal  celsius (fim de linha)
Fractal  value será  35 (fim de linha)
celsius será  [ value extraido de  32 ] replicado por  [ 5 fragmentado em  9 ] (fim de linha)
farenheit será  [ value replicado por  9 fragmentado em  5 ] fundido a  32 (fim de linha)
```

Figura 54: Resultado do código de conversão em Grimório quando passado no analisador léxico

Ao analisar a saída é possível observar os operadores aritméticos sendo corretamente reconhecidos e marcados, assim como os valores e variáveis usados nas operações. Os dois tipos primitivos passados estão marcados em verde e todas as marcações de prioridade (colchetes) estão corretamente sendo reconhecidos.

Exemplo 3: Funções

O exemplo abaixo é uma implementação básica de função. A ideia é criar um simples bloco capaz de receber como parâmetros dois inteiros e retornar a soma de seus valores. Após isso, duas variáveis são instanciadas para armazenar os valores que serão passados para a função, e uma terceira é usada para armazenar os resultados finais.

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int valor1 = 100;
    int valor2 = 50;
    int resultado;

    resultado = add(valor1, valor2);
    return 0;
}
```

Figura 55: Exemplo de funções em C

```
Preparar magia de Glifo add, Componentes: Glifo a, Glifo b _/
    Regressus a fundido a b...
\_

Glifo valor1 será 100...
Glifo valor2 será 50...
Glifo resultado...

resultado será Conjurar add[valor1, valor2]...
```

Figura 56: Exemplo de funções em Grimório

Desta forma, torna-se evidente algumas diferenças chave entre as declarações nas linguagens. Primeiramente, a linguagem Grimório não possui uma estrutura de bloco main. Além disso, na criação do bloco de função um termo reservado “Preparar magia de” precisa ser utilizado para iniciar a instanciação. Por fim, após o nome da variável, é utilizado um caractere de vírgula seguido pelo termo reservado “Componentes:” para se iniciar a declaração de parâmetros (separados por vírgula).

```
Preparar magia de  Glifo  add , Componentes:  Glifo  a , Glifo  b
abre bloco _/
Regressus  a fundido a  b (fim de linha)
fecha bloco \_
Glifo  valor1 será  100 (fim de linha)
Glifo  valor2 será  50 (fim de linha)
Glifo  resultado (fim de linha)
resultado será  Conjurar  add [ valor1 , valor2 ] (fim de linha)
```

Figura 57: Resultado do código de funções em Grimório quando passado no analisador léxico

O analisador léxico reconhece e colore as palavras reservadas principais relativas à declaração de funções, colorindo-as de amarelo. É importante notar que o nome dado a função é reconhecido em vermelho assim como uma variável.

Parte 2

Atualização do Lex para análise sintática

Para a segunda parte do trabalho, foi necessário adaptar o analisador léxico para não mais ser um analisador stand-alone, trabalhando agora em conjunto com o Yacc. Para tal, foi necessária a implementação de novas chamadas durante a etapa de leitura de cada sequência de caracteres. As atualizações incluíram a definição de um código de caracteres para o token de cada lexema, além de pequenas atualizações em alguns lexemas para facilitar a derivação durante a etapa sintática.

O primeiro passo foi incluir o arquivo .h gerado pela compilação do Yacc no cabeçalho do arquivo lex, criando uma conexão entre os dois arquivos, que pode ser usada para passar os tokens e outras variáveis comuns que serão usadas posteriormente em algumas funções.

```
#include <stdio.h>
#include "translate.tab.h"

int line_number = 1;
```

Figura 58: inclusão do cabeçalho do yacc no lex

Com o arquivo corretamente importado no Lex, também foi criada uma variável para contagem de linhas, além de uma alteração na regra “ws” do lex que antes informava ao analisador que deveria ignorar quebras de linha. Agora, com o novo sistema de contagem é importante que tais linhas não sejam mais ignoradas, mas sim incrementem o contador para gerar uma mensagem mais completa de erro.

```
int line_number = 1;

%}

/* Name Definition */
delim        [ \t\r]
ws         {delim}+
```

Figura 59: Definição do contador de linhas

O contador “line_number” é instanciado com seu valor igual à um no início da etapa de análise léxica. Abaixo a regra delim agora possui apenas como símbolos reconhecidos tabulações “\t” e espaços em branco “\r”.

```
extern int yyparse();
extern int invalid_found;
extern char invalid_chars[];
extern int line_number;
```

Figura 60: Novas definições para reconhecer contador de linhas

A implementação da contagem de linhas também se estende para o arquivo Yacc, uma vez que é necessário neste arquivo implementar a função de impressão caso haja um erro sintático. É declarado no topo do arquivo a variável “extern” que faz referência a contida no arquivo .lex.

No final do arquivo, a função de impressão é propriamente instanciada, possuindo uma linha formatada que recebe o valor da variável de contagem de linhas e um imprime no terminal um texto em vermelho informando a linha aproximada do erro, além disso as outras variáveis ”extern” são usadas para checar a existência de um erro léxico, caso haja um valor verdadeiro na variável “invalid_found”, será impresso no terminal o erro léxico com a string concatenada de todos os caracteres inválidos encontrados:

```
/* Error reporting function */
void yyerror(const char *s) {
    fprintf(stderr, "\033[0;31mErro Arcano...\033[0m A linha %d do grimório contém problemas.\n", line_number);
}
```

Figura 61: Formato da mensagem de erro do analisador

No terminal, a saída pode ser vista como:

```
1   Glifo num1 será 10...
2   Glifo num2 será 15...
3
4
5
6
7
8
9   Glifo num3 sera 20...
```

```
patrick@ubuntrick:~/Area de Trabalho/Compiladores/TP1/tp-compilador
Erro Arcano... A linha 9 do grimório contém problemas.
```

Figura 62: Erro sintático exibido no terminal

No exemplo acima, o código apresentado possui um erro na linha nove, uma vez que o operador de atribuição “será” está sendo utilizado sem o acento, configurando um erro sintático. É possível notar, que a mensagem emite um aviso apontando um erro na linha nove.

```
1   Glifo num1 será 10...
2   Glifo num2 será 15...
3
4
5
6   -&
7   $%
8   #
9   ##
10
11
12  Glifo num3 sera 20...
```

Figura 63: Exemplo de código com erro léxico e sintático

Neste segundo exemplo, além do erro sintático que agora está na linha 12, existem erros léxicos da linha 6 até a linha 9. Desta forma, no terminal serão impressas mensagens alertando erros em cada uma destas linhas e o respectivo caractere não reconhecido, além do erro sintático.

```
patrick@ubuntrick:~/Area de Trabalho/Compiladores/TP1/tp-compiladores$ ./teste < ./testes/input
Erro de Infusao... A magia foi profanada na linha 6... As seguintes ranhuras sao condenadas: -
Erro de Infusao... A magia foi profanada na linha 6... As seguintes ranhuras sao condenadas: &
Erro de Infusao... A magia foi profanada na linha 7... As seguintes ranhuras sao condenadas: $
Erro de Infusao... A magia foi profanada na linha 7... As seguintes ranhuras sao condenadas: %
Erro de Infusao... A magia foi profanada na linha 8... As seguintes ranhuras sao condenadas: #
Erro de Infusao... A magia foi profanada na linha 9... As seguintes ranhuras sao condenadas: #
Erro de Infusao... A magia foi profanada na linha 9... As seguintes ranhuras sao condenadas: #
Erro Arcano... A linha 12 do grimório contém problemas.
```

Figura 64: Exibição dos erros léxicos e sintáticos

Alterações na detecção de definições regulares

Para retornar os tokens para o yacc, dois tipos de retornos são feitos. Os retornos de valores numéricos e booleanos necessitam de um valor associado à eles. Desta forma, para passar tais valores para o analisador sintático pode ser utilizado a variável yyval.ival que receberá o valor associado do lexema, em caso de inteiros tal

valor é passado com a formatação da função “atoi” que converte o valor para um inteiro e o armazena.

```
{digito_positivo} { yylval.ival = atoi(yytext); return INT; }
{digito_negativo} { yylval.ival = atoi(yytext); return INT; }
{float_posit} { yylval.fval = atof(yytext); return FLOAT; }
{float_negat} { yylval.fval = atof(yytext); return FLOAT; }
{true} { yylval.ival = 1; return TRUE; }
{false} { yylval.ival = 0; return FALSE; }
{atribuicao} { return ASSIGN; }
```

Figura 65: Mudança do yylval.ival para retorno

A mesma lógica pode ser aplicada para floats, com a utilização da função “atof”. Para booleanos, é aplicado uma formatação mais simples, onde true recebe o valor inteiro “1” e false recebe “0”, assim como na linguagem C. Para os demais tokens que não possuem valor associado, é feito apenas o retorno do respectivo token.

```
{soma} { yylval.arOp = PLUS; return AROP; }
{subtracao} { yylval.arOp = MINUS; return AROP; }
{multiplicacao} { yylval.arOp = MULT; return AROP; }
{divisao} { yylval.arOp = DIV; return AROP; }
{mod} { yylval.arOp = MOD; return AROP; }

{adicionarUm} { return PLUSONE; }
{subtrairUm} { return MINUSONE; }
{derreferenciar} { return DEREF; }
{referenciar} { return REF; }
```

Figura 66: Mudança do yylval.ival para retorno

Acima estão os tokens retornados pelos lexemas aritméticos e unários, o nome criado para cada um reflete o termo em inglês e com letras maiúsculas. Os tokens gerados foram associados à suas respectivas contrapartidas em inglês e não os termos próprios da linguagem para facilitar a leitura das derivações no arquivo da análise sintática.

Extensão de reconhecimento léxico

Como um adicional para a parte 2, foi criada uma extensão para o Visual Studio Code que deixa os lexemas coloridos durante a edição de arquivos .grim

(extensão da nossa linguagem). A criação da biblioteca foi feita utilizando um processo parcialmente automatizado usando as bibliotecas chamadas Yeoman e Generator-Code, elas instanciam todas as dependências particulares do VS Code como arquivos de configuração, JSON e outros arquivos que serão utilizados para integrar a biblioteca com o aplicativo.

Com o passo-a-passo concluído e tudo corretamente configurado, é possível definir assim como no Lex, as definições regulares diretamente em um JSON, a sintaxe do lex é a mesma utilizada no Flex, exceto pela presença de alguns símbolos a mais pelo formato particular do JSON.

```
"keywords": {
    "patterns": [{
        "name": "keyword.control.grimorio",
        "match": "\\b(Dilatar tempo enquanto |Fraturar tempo se |Ponderar se |Ou optar por |
    }]
},
```

Figura 67: Definições para extensão Grimório vscode

É possível ver que dentro da string “*match*”, existe unicamente uma definição regular com vários “*or*”, que definem o reconhecimento de diferentes palavras reservadas, o padrão é repetido para todos os tipos de palavra reservada, como tipos primitivos, operadores, constantes e variáveis.

```
teste.grim
1   Glifo teste1 será 10...
2   Glifo num1 será 10...
3   Arquiglifo teste2 será 15...
4   Semiglifo teste3 será 20...
5   Fractal teste4 será 10.0...
6   Arquifractal teste5 será 15.0...
7   Axioma testeBool será Veritas...
8   Runa testeChar será 'C'...
9
10  Preparar magia de Glifo somaNum, Componentes: [Glifo num1, Glifo num2] _/
11      Regressus num1 ...
12  \_
13
14  Transmutar Componentização em construto _/
15      Runa identificador...
16      Glifo tamanho...
17  \_
18
19  Transmutar Conjuntura em conj _/
20      Runa identificador...
21      Glifo tamanho...
22  \_
23
24
25  Preparar magia de Glifo principal, Componentes: [] _/
26      Dilatar tempo enquanto teste1 for inferior a 20 _/
27          Ponderar se teste1 for superior a teste3 _/
28              teste1 será teste1 dissolvido de 10...
29          \_
30      \_
31      Glifo resultado...
32      Glifo i será 0...
33      Fraturar tempo se i for inferior a 10, Encantar i  /
```

Figura 68: Exemplo de código com a extensão do vscode

Com a extensão configurada, é possível visualizar cada uma das palavras reservadas e termos reservados da linguagem colorida com seus respectivos tipos e cores. Vale ressaltar que dependendo do tema utilizado alguns tipos podem ter cores diferentes ou em alguns casos apenas a cor branca padrão.

Devido a natureza incrementava do trabalho, a versão completa da biblioteca será disponibilizada em conjunto com a terceira parte do trabalho.

Análise Sintática

Aprofundando agora no arquivo Yacc denominado "*translate.y*", realizamos a inclusão de algumas bibliotecas padrões de C e também de alguns TADs criados por nós para auxiliar e fomentar a análise sintática. Esses TADs serão devidamente explicados futuramente nessa documentação. Também utilizamos de variáveis

globais para dar suporte às operações que implementamos para deixar nosso compilador mais completo.

```
1   %{
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5   #include <math.h>
6
7   #include "structures/SymbolTable.h"
8   #include "structures/Expression.h"
9   #include "structures/Operators.h"
10
```

Figura 69: Inclusão de arquivos de autoria própria e bibliotecas

```
10
11   extern int yyparse();
12   extern int invalid_found;
13   extern char invalid_chars[];
14   extern int line_number;
15
```

Figura 70: Variáveis globais

Além disso, também declaramos alguns cabeçalhos de funções que estão diretamente relacionadas à tabela de símbolos, estrutura de suma importância para o desenvolvimento e bom funcionamento de um compilador. Abordaremos o funcionamento e a estrutura interna da tabela de símbolos mais adiante nesse documento.

```
SymbolTable *current_table;
Expression *evaluate_arithmetic(Expression left, Expression right, ArOp op);
Expression *evaluate_relational(Expression left, Expression right, RelOp op);
Expression *evaluate_and(Expression left, Expression right);
Expression *evaluate_or(Expression left, Expression right);
Expression *evaluate_not(Expression expr);
Expression *id_to_expression(char *id);
void* perform_arithmetic(Expression left, Expression right, ArOp op);
void* perform_int_arithmetic(void *left, void *right, ArOp op);
void* perform_float_arithmetic(void *left, void *right, ArOp op);
void* perform_double_arithmetic(void *left, void *right, ArOp op);
void* perform_long_arithmetic(void *left, void *right, ArOp op);
void* perform_short_arithmetic(void *left, void *right, ArOp op);
Expression* create_expression(Type type, void *value);
void assign_value_to_symbol(Symbol *symbol, Expression *expr);
void assign_value_to_expression(Symbol *symbol, Expression *expr);
void apply_unary_operation(Expression *result, Symbol *operand, int operation);
```

Figura 71: Cabeçalhos de funções da tabela de símbolos

Derivações da Gramática

A variável de partida para as derivações é "*start",* que pode ser derivada em uma derivação vazia ou nas variáveis "*start start_item",* que possibilitam as demais derivações na raíz do programa.

A variável "start_item" deriva nas seguintes variáveis:

- *"decl_func"* que tem o intuito de declarar funções da forma de tipo de retorno, nome, iniciar definição de parâmetros, abrir parênteses, argumentos da função, fechar parênteses e um bloco de statements das funções;
- "*decl_import"* é para a importação de bibliotecas e arquivos e deriva em definição de importação, um literal e o marcador de fim de linha;
- *"decl_stmt*" é a declaração de statements, pode derivar em uma atribuição, declaração de variável, definição de tipo e definição de função;
- "*unnary_expr ENDLINE*" que permite a derivação em expressões unárias. O token ENDLINE, representado pelo padrão de lexema "...", indica o fim de uma linha de código.

```
105  begin: start
106          ;
107
108  start: /* empty */
109       | start start_item
110       ;
111
112  start_item: decl_stmt
113            | decl_func
114            | decl_import
115            | unary_expr ENDLINE
116            ;
117
118  decl_import: IMPORT LITERALSTRING ENDLINE
119             ;
120
121  decl_func: DECLFUNC type ID PARAMS OPENBRACK arguments CLOSEBRACK stmt_block
122            {
123             Function *func = create_function($2);
124             Param *param = $6;
125             add_parameter_list(func, &param);
126             Symbol *new_symbol = insert_symbol(current_table, $3, TYPE_FUNC, (void*)func);
127            }
128           ;
129
130  decl_stmt: assignment ENDLINE
131           | sign_func ENDLINE
132           | type_def
133           | decl_var ENDLINE
134           ;
```

Figura 72: Derivações das primeiras variáveis da gramática

A variável "*stmt_blocks*", é fundamental para a criação de qualquer bloco. Ela deriva em símbolo de abertura de bloco (_/), "*stmts"* e símbolo de fechamento de bloco (_) .

```
136  stmt_block: OPENBLOCK
137              {
138                SymbolTable *new_table = create_symbol_table(current_table);
139                current_table = new_table;
140              }
141              stmts
142              CLOSEBLOCK
143              {
144                SymbolTable *old_table = current_table;
145                current_table = current_table->parent;
146              }
147            ;
```

Figura 73: Derivação para a criação de um bloco de statements

A variável “*stmts*” deriva em vazio e em “*stmts stmt*”, e é uma das principais variáveis da gramática, derivando em *“decl_stmt*” e outras estruturas de fluxo de programação, que são: “*stmt_if*”, “*stmt_for*”, “*stmt_while*”, “*"stmt_switch"*”, “*stmt_break*”, “*stmt_return*”, “*stmt_continue*”.

- “*"stmt_if"*”: deriva no token “IF” (Ponderar se) “*stmt_block*” e “*stmt_else*”
- “*stmt_else*”: deriva no token “ELSE” (Ou optar por) “*stmt_block”* ou ELSE "stmt_if" (para o else if) ou vazio caso tenha apenas um if sem else;
- *“stmt_for”*: deriva no token “FOR” (Fraturar tempo se) "expr" “COMMA” (que é o token de vírgula) “*unary_expr”* (que incrementa e decrementa um “*expr”*) *“stmt_block”* ;
- “*stmt_while*”: deriva no token “WHILE” (Dilatar tempo enquanto) “*expr*” “*stmt_block*”;
- "*stmt_switch*": deriva no token “SWITCH” (Iniciar profecia de) "*expr*" símbolo de abertura de bloco (_/) “*case_list” “default_case”* símbolo de fechamento de bloco (_);
    - *“case_list”* deriva em vazio ou case_list case_stmt;
    - “*case_stm*t” deriva no token “CASE” (Profetizar que se) "*expr*" token DELIMCASE (;);
    - *“default_case”* deriva no token “DEFAULT” (Cumprir profecia) token DELIMCASE “*stmts*” *“stmt_break”;*
- *“stmt_break”*: deriva no token “BREAK” (Que assim seja) e no marcador de fim de linha;
- “*stmt_return*”: deriva no token “RETURNT” (a letra T foi acrescentada no final do token, pois return é uma palavra reservada então utilizamos “RETURNT” de return token) e marcador de fim de linha;
- *“stmt_continue”*: retorna o token “CONTINUE” (Prossiga) e marcador de fim de linha.

```
stmt_if: IF expr stmt_block stmt_else
       ;

stmt_else: ELSE stmt_block
         | ELSE stmt_if
         | /* empty */
         ;
```

Figura 74: Derivações do “statement” de uma condicional

```
stmt_for: FOR expr COMMA unary_expr stmt_block
        ;

stmt_while: WHILE expr stmt_block
          ;

```

Figura 75: Derivações para o “statement” “for” e “while”

```
stmt_switch: SWITCH expr
             OPENBLOCK
             {
               SymbolTable *new_table = create_symbol_table(current_table);
               current_table = new_table;

             }
             case_list default_case
             CLOSEBLOCK
             {
               SymbolTable *old_table = current_table;
               current_table = current_table->parent;
             }
           ;

case_list: /* empty */
         | case_list case_stmt
         ;

case_stmt: CASE expr DELIMCASE stmts stmt_break
         ;

default_case: /* empty */
            | DEFAULT DELIMCASE stmts stmt_break
            ;

```

Figura 76: Derivação das variáveis que compõem o “Switch-Case”

```
stmt_break: BREAK ENDLINE
            ;

stmt_continue: CONTINUE ENDLINE
               ;

stmt_return: RETURNT expr ENDLINE
             ;

```

Figura 77: Derivações dos “statements” de “break”, “continue” e “return”

A variável “*assignment*” deriva em um identificador para uma variável token “ASSIGN” "*expr*" e a variável “*opt_assignment”* deriva em vazio ou no token “ASSIGN” “*expr*”.

```
assignment: variable ASSIGN expr
            {
             Symbol *symbol = lookup_symbol(current_table, $1->name);
             if (symbol == NULL) {
                 yyerror("\033[0;34mInscricao arcana\033[0m nao encontrada...\n");
             } else {
                 assign_value_to_symbol(symbol, $3);
             }
            }
           ;

opt_assignment: /* empty */
                {
                 Expression *expression = (Expression*)malloc(sizeof(Expression));
                 expression->type = TYPE_VOID;
                 expression->value = NULL;
                 $$ = expression;
                }
               | ASSIGN expr
               {
                $$ = $2;
               }
               ;

```

Figura 78: Derivações das variáveis de atribuição

A variável *“decl_var”* é utilizada para derivar outras 4 formas de declarações de variáveis do código:

- tipo token “ID” “*opt_assignment”;*
- tipo token “CONST” token “ID” “*opt_assignment”;*

- tipo token “VOLATILE” token “ID” “*opt_assignment*”;
- tipo token “VOLATILE” token “CONST” token “ID” “*opt_assignment*”;

```
239 decl_var: type type_qualifier ID opt_assignment
240             {
241              if ($4->type != $1 && $4->type != TYPE_VOID) {
242                yyerror("\033[0;34mInscricao Arcana\033[0m nao inicializada com o tipo correto...\n");
243              }
244              Symbol *new_symbol = insert_symbol(current_table, $3, $1, $4->value);
245             }
246           | type ID opt_assignment
247             {
248              if ($3->type != $1 && $3->type != TYPE_VOID) {
249                yyerror("\033[0;34mInscricao Arcana\033[0m nao inicializada com o tipo correto...\n");
250              }
251              Symbol *new_symbol = insert_symbol(current_table, $2, $1, $3->value);
252             }
253           ;
254
255 type_qualifier: CONST
256                 | VOLATILE
257                 | VOLATILE CONST
258                 ;
```

Figura 79: Derivações de declaração de variáveis e qualificadores de tipo

A variável “*def_type*” deriva nas três formas de definição:

- token “TYPEDEF” tipo token “ID” e marcação de fim de linha;
- token “TYPEDEF” token “STRUCT” token “ID”;
- token “TYPEDEF” token “UNION” token “ID”.

```
428 type_def: TYPEDEF type ID
429           | TYPEDEF enum_def
430           | TYPEDEF struct_def
431           | TYPEDEF union_def
432           ;
```

Figura 80: Derivações da variável de definição de tipo

A variável *“sign_func”* é a que define o cabeçalho de uma função, ela é derivada em um tipo, token de “ID”, token de “PARAMS” e nos argumentos da função.

```
sign_func: type ID PARAMS arguments
        ;
```

Figura 81: Derivação da variável que define funções

A variável *"type_qualifier"* define qual é o qualificador da variável e é derivada em "CONST" ou "VOLATILE" ou "VOLATILE" "CONST".

```
252   type_qualifier: CONST
253                 | VOLATILE
254                 | VOLATILE CONST
255                 ;
```

Figura 82: Derivação da variável de qualificador de tipo

A variável "expr" é essencial para a nossa gramática e possui 6 derivações:

- *"expr"* token "AROP" (operador aritmético para operações aritméticas básicas) *"term"*. Os operadores aritméticos são:
    - soma (fundido a);
    - subtração (dissolvido de);
    - multiplicação(replicado por);
    - divisão(fragmentado em);
    - mod (transmoglifado por);
- *"expr"* token "RELOP" (operador relacionais) *"term"*. Os operadores relacionais são:
    - maior (for superior a);
    - menor (for inferior a);
    - maior igual (for superequivalente a);
    - menor igual (for infraequivalente a);

    - igual (for equivalente a);
    - diferente (for distinto de);

- *“expr”* token “AND” (assim como) *“term”;*
- *“expr”* token “OR” (ou então) *“term”;*
- token “NOT” (contrariando) *“expr”*;
- *“term”;*

```
expr: expr AROP term { $$ = evaluate_arithmetic(*$1, *$3, $2); }
    | expr RELOP term { $$ = evaluate_relational(*$1, *$3, $2); }
    | expr AND term { $$ = evaluate_and(*$1, *$3); }
    | expr OR term { $$ = evaluate_or(*$1, *$3); }
    | NOT expr { $$ = evaluate_not(*$2); }
    | term { $$ = $1; }
    ;
```

Figura 83: Derivações possíveis de *"expr"*

A variável *"term"* que apareceu em várias derivações da variável *“expr”*, representa um termo, um valor. Suas derivações são:

- literal;
- token INT;
- token FLOAT;
- variable;
- bool;
- *“function_call”*;
- *“unary_expr”;*
- token “OPENBRACK” *"expr"* token “CLOSEBRACK”;

```
term: literal { $$ = $1; }
    | INT
    {
      Expression *result = create_expression(TYPE_INT, NULL);
      result->value = malloc(sizeof(int));
      *(int*)result->value = $1;
      $$ = result;
    }
    | FLOAT
    {
      Expression *result = create_expression(TYPE_FLOAT, NULL);
      result->value = malloc(sizeof(float));
      *(float*)result->value = $1;
      $$ = result;
    }
    | variable
    {
      Expression *result = create_expression($1->type, NULL);
      assign_value_to_expression($1, result);
      $$ = result;
    }
    | bool { $$ = $1; }
    | function_call { $$ = $1; }
    | unary_expr { $$ = $1; }
    | OPENBRACK expr CLOSEBRACK { $$ = $2; }
    ;
```

Figura 84: Derivações da variável term

Tabela de Símbolos

Para realizar uma análise completa e funcional da linguagem, foi desenvolvida uma tabela de símbolos. A estrutura em questão armazena os símbolos encontrados durante a análise sintática em uma tabela hash, com uma tabela criada para cada bloco de escopo encontrado.

```
| 1 | Glifo x será 12...
| 2 | Glifo z será 20...
| 3 | Runa runa1 será ...
| 4 | Arquiglifo arquiglifo1...
| 5 | Arquifractal arquifractal1...
| 6 | Fractal fractal1...
| 7 | Semiglifo semiglifo1...
| 8 | Axioma axioma1 será Veritas...
| 9 | x será Encantar z...
| 10 | Runa jaja será ...
| 11 |
| 12 | Transmutar Glifos em Listagem _/
| 13 |   teste,
| 14 |   teste1,
| 15 |   teste2
| 16 | \_
| 17 |
| 18 | Transmutar Componentização em Golemita _/
| 19 |   Glifo a...
| 20 |   Runa b...
| 21 |   Arquiglifo c...
| 22 |   Arquifractal d...
| 23 |   Fractal e...
| 24 |   Semiglifo f...
```

Figura 85: Exemplo de impressão do programa fonte com as linhas numeradas.

```
| 47 |       Que assim seja...
| 48 |   \_
| 49 |
| 50 |   z-|1|- será 20...
| 51 |   z será z...
| 52 | \_
| 53 |
| 54 | Encantar z...
| 55 |
| 56 | Glifo pp será 0...
| 57 | Glifo pa será 10...
| 58 |
| 59 | Encantar pp...
| 60 | pa será Encantar pp...
| 61 |
| 62 |
| 63 | Glifo t será Conjurar Additium, Componentes: [x,z]...
| 64 |
| 65 |
| 65 |
Codigo sintaticamente correto.
```

Figura 86: Exemplo de impressão do programa fonte com as linhas numeradas.

```
Parsing complete
----------------------------
Name: semiglifo1, Type: intValue: 0
Name: golem, Type: struct
Name: pa, Type: intValue: 2
Name: fractal1, Type: floatValue: 0.000000
Name: lista, Type: enum
Name: axioma1, Type: boolValue: Veritas
Name: pp, Type: intValue: 2
Name: Additium, Type: functionFunction: int params:
Name: t, Type: intValue: 0
Name: x, Type: intValue: 21
Name: z, Type: intValue: 21
Name: arquifractal1, Type: floatValue: 0.000000
Name: runa1, Type: charValue: u
Name: jaja, Type: charValue: o
Name: arquiglifo1, Type: intValue: 0
```

Figura 87: Exibição da tabela de símbolos do escopo global no fim da análise do programa.

```
Unset
Glifo x será 12...
Glifo z será 20...
Runa runa1 será 'u'...
Arquiglifo arquiglifo1...
Arquifractal arquifractal1...
Fractal fractal1...
Semiglifo semiglifo1...
Axioma axioma1 será Veritas...
x será Encantar z...
Runa jaja será 'o'...

Transmutar Glifos em Listagem _/
  teste,
  teste1,
  teste2
\_

Transmutar Componentização em Golemita _/
  Glifo a...
  Runa b...
  Arquiglifo c...
  Arquifractal d...
  Fractal e...
  Semiglifo f...
  Axioma g...
\_

Componentização Golemita golem...
Glifos Listagem lista...


Preparar magia de Glifo Additium, Componentes: [Glifo unnus, Glifo annus]
```

```
_/
  Glifo w...
  w será 20...
  Iniciar profecia de z
  _/
    Profetizar que se 1;
      Fractal jota será 1.0...
      z será z fundido a 1...
      Que assim seja...
    Profetizar que se 2;
      z será z fundido a 2...
      Que assim seja...
    Cumprir profecia;
      z será z fundido a 3...
      Que assim seja...
  \_

  z-|1|- será 20...
  z será z...
\_

Encantar z...

Glifo pp será 0...
Glifo pa será 10...

Encantar pp...
pa será Encantar pp...


Glifo t será Conjurar Additium, Componentes: [x,z]...
```

Código Grimório 1: Código sintaticamente correto que exibe a tabela da imagem anterior.

Outras estruturas de apoio foram criadas para representar tipos e enumerações necessários na manipulação das diversas cláusulas possíveis encontradas no código. Essas estruturas serão melhor explicadas nas próximas seções.

```c
#ifndef SYMBOLTABLE_H
#define SYMBOLTABLE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Types.h"
#include "Function.h"
#include "Expression.h"

#define HASH_SIZE 47
#define MAX_LEVEL 10

typedef struct Symbol {
    char *name;
    Type type;
    void *value;
    int volatile_flag;
    int constant_flag;
    struct Symbol *next;
} Symbol;

typedef struct SymbolTable {
    Symbol *table[HASH_SIZE];
    struct SymbolTable *parent;
} SymbolTable;


unsigned int hash(char *name);
SymbolTable *create_symbol_table(SymbolTable *parent);
Symbol* insert_symbol(SymbolTable *table, char *name, Type type, void* value);
Symbol *lookup_symbol(SymbolTable *table, char *name);
void print_table(SymbolTable *table);
void free_symbol_table(SymbolTable *table);
void* allocate_and_initialize(Type type);
char* bool_to_string(int value);


#endif
```

Figura 88: Arquivo "*SymbolTable.h*" representando a estrutura da tabela de símbolos.

- hash

O objetivo principal da função é converter uma string (nome de um símbolo) em um valor numérico (hash) que será utilizado como índice para armazenar ou buscar o símbolo na tabela de símbolos, feita utilizando hash. A função garante que diferentes nomes de símbolos sejam distribuídos ao longo da tabela, minimizando colisões e permitindo buscas rápidas.

```c
unsigned int hash(char *name)
{
  unsigned int hash_value = 0;
  while (*name)
  {
    hash_value = (hash_value << 3) + *name++;
  }
  return hash_value % HASH_SIZE;
}
```

Figura 89: Estrutura de tabela hash para o corpo da tabela de símbolos

- “create_symbol_table”

Essa função é utilizada para criar e inicializar uma nova tabela de símbolos. A função começa alocando dinamicamente uma memória para uma nova estrutura *SymbolTable* usando a função malloc. Isso é necessário, pois a nova tabela precisa de um espaço na memória para armazenar seus dados. Em seguida, há a definição do ponteiro “*parent*” para apontar à tabela de símbolos pai fornecida como argumento. Por fim, inicializa e retorna a tabela hash para permitir a inserção e consulta de símbolos.

```
18  SymbolTable *create_symbol_table(SymbolTable *parent)
19  {
20    SymbolTable *new_table = (SymbolTable *)malloc(sizeof(SymbolTable));
21    new_table->parent = parent;
22    for (int i = 0; i < HASH_SIZE; i++)
23    {
24      new_table->table[i] = NULL;
25    }
26    return new_table;
27  }
```

Figura 90: Função para criar a tabela de símbolos.

- “insert_symbol”

É necessário uma função para inserir símbolos associando o nome do símbolo com seu tipo e valor. Primeiramente, a função calcula um índice na tabela hash chamando a função hash com o nome do símbolo. O índice é usado para determinar em qual posição da tabela hash o símbolo será armazenado. Em seguida, a função aloca dinamicamente uma memória para a nova estrutura Symbol que será inserida na tabela de símbolos com os respectivos valores recebidos por parâmetros. Caso o valor fornecido para o símbolo seja nulo, a função “allocate_and_initialize” é chamada. Após isso, há a inserção e associação dos valores dos símbolos na tabela de símbolos.

```c
Symbol* insert_symbol(SymbolTable *table, char *name, Type type, void* value)
{
  unsigned int index = hash(name);
  Symbol *new_symbol = (Symbol *)malloc(sizeof(Symbol));
  new_symbol->name = strdup(name);
  new_symbol->type = type;

  if (value == NULL) {
    new_symbol->value = allocate_and_initialize(type);
  } else {
    new_symbol->value = value;
  }

  new_symbol->next = table->table[index];
  table->table[index] = new_symbol;
  return new_symbol;
}
```

Figura 91: Função para inserir um símbolo na tabela de símbolos.

- *“allocate_and_initialize”*

Esta função recebe um tipo e o leva para um switch case, alocando o espaço necessário para armazenar seu valor na tabela de símbolos baseado em seu tipo (“INT”, “FLOAT”, “DOUBLE”, “CHAR”, “BOOL”, “TYPE_LONG” ou “TYPE_SHORT”).

```c
void* allocate_and_initialize(Type type) {
    void *defaultValue = NULL;

    switch (type) {
        case TYPE_INT: {
            int *val = (int*)malloc(sizeof(int));
            *val = 0;
            defaultValue = (void*)val;
            break;
        }
        case TYPE_FLOAT: {
            float *val = (float*)malloc(sizeof(float));
            *val = 0.0;
            defaultValue = (void*)val;
            break;
        }
        case TYPE_DOUBLE: {
            double *val = (double*) malloc(sizeof(double));
            *val = 0.0;
            defaultValue = (void*)val;
            break;
        }
        case TYPE_CHAR: {
            char *val = (char*) malloc(sizeof(char));
            *val = '\0';
            defaultValue = (void*)val;
            break;
        }
        case TYPE_BOOL: {
            int *val = (int*)malloc(sizeof(int));
            *val = 0;
            defaultValue = (void*)val;
            break;
        }
        case TYPE_LONG: {
            long *val = (long*)malloc(sizeof(long));
            *val = 0;
            defaultValue = (void*)val;
            break;
        }
        case TYPE_SHORT: {
            short *val = (short*)malloc(sizeof(short));
            *val = 0;
            defaultValue = (void*)val;
            break;
        }
    }

    return defaultValue;
}
```

Figura 92: Função para inicializar e alocar espaço para cada tipo.

- “lookup_symbol”

A função lookup_symbol é projetada para procurar um símbolo, a partir do nome do símbolo, na tabela de símbolos. Se o símbolo não for encontrado na tabela atual, a função procura recursivamente na tabela pai, permitindo a procura de símbolos em escopos mais amplos. Se o símbolo não for encontrado em nenhum escopo, a função retorna NULL, indicando que o símbolo não está definido, caso contrário, retorna o símbolo encontrado.

```c
Symbol *lookup_symbol(SymbolTable *table, char *name)
{
  unsigned int index = hash(name);
  Symbol *current_symbol = table->table[index];
  while (current_symbol)
  {
    if (strcmp(current_symbol->name, name) == 0)
    {
      return current_symbol;
    }
    current_symbol = current_symbol->next;
  }
  if (table->parent)
  {
    return lookup_symbol(table->parent, name);
  }
  return NULL;
}
```

Figura 93: função de procura por símbolo na tabela de símbolos.

- “print_table”

A função print_table percorre a tabela de símbolos e imprime informações detalhadas sobre cada símbolo, incluindo nome, tipo e valor. Essa função faz isso de forma organizada, lidando com diferentes tipos de dados, e se necessário, imprime, de maneira recursiva, as tabelas de símbolos em escopos superiores. Essa função é útil para depuração e verificação do estado da tabela de símbolos em diferentes momentos durante a execução do programa.

```c
118 void print_table(SymbolTable *table)
119 {
120   printf("-----------------------------\n");
121   for (int i = 0; i < HASH_SIZE; i++)
122   {
123     Symbol *current_symbol = table->table[i];
124     while (current_symbol)
125     {
126       printf("Name: %s, Type: %s", current_symbol->name, type_to_string(current_symbol->type));
127       if (current_symbol->type == TYPE_INT)
128       {
129         printf("Value: %d\n", *(int *)current_symbol->value);
130       }
131       else if (current_symbol->type == TYPE_FLOAT)
132       {
133         printf("Value: %f\n", *(float *)current_symbol->value);
134       }
135       else if (current_symbol->type == TYPE_DOUBLE)
136       {
137         printf("Value: %f\n", *(double *)current_symbol->value);
138       }
139       else if (current_symbol->type == TYPE_CHAR)
140       {
141         printf("Value: %c\n", *(char *)current_symbol->value);
142       }
143       else if (current_symbol->type == TYPE_BOOL)
144       {
145         printf("Value: %s\n", bool_to_string(*(int *)current_symbol->value));
146       }
147       else if (current_symbol->type == TYPE_FUNC)
148       {
149         print_function(*(Function*)current_symbol->value);
150       } else {
151         printf("\n");
152       }
153
154       current_symbol = current_symbol->next;
155     }
156   }
157   if (table->parent)
158   {
159     print_table(table->parent);
160   }
161 }
```

Figura 94: Função para imprimir o atual estado da tabela de símbolos

- “free_symbol_table”

Esta função é responsável por liberar o espaço reservado para a tabela de símbolos de dado escopo, sempre que durante a análise sintática é feito uma saída de escopo, o espaço daquela tabela é liberado.

```c
163 void free_symbol_table(SymbolTable *table) {
164     for (int i = 0; i < HASH_SIZE; i++) {
165         Symbol *symbol = table->table[i];
166         while (symbol) {
167             Symbol *temp = symbol;
168             symbol = symbol->next;
169             free(temp->name);
170             free(temp);
171         }
172     }
173     free(table);
174 }
```

Figura 95: função para limpar tabela de símbolos após saída de escopo.

- “bool_to_string”

Essa função possui o objetivo de transformar o valor inteiro recebido por parâmetro para uma string. Se o valor for 0, retorna a string “Falsum”, caso contrário, retorna a string “Veritas”.

```
176  char* bool_to_string(int value) {
177      if (value == 0) {
178          return "Falsum";
179      } else {
180          return "Veritas";
181      }
182  }
```

Figura 96: função de conversão de booleanos de inteiro para string.

Estruturas de Apoio

Estas estruturas foram criadas com o intuito de facilitar o desenvolvimento da linguagem e permitir que variáveis e tokens fossem tipadas e/ou pudessem retornar ou armazenar múltiplos valores.

Function

A estrutura de função é utilizada para armazenar na tabela de símbolos as informações de cada função instanciada na linguagem, cada função possui um campo do tipo “*Type*” que irá armazenar o tipo do retorno da função, seguido por um campo representando o nome da função. Por fim, a função possui um ponteiro para o primeiro parâmetro passado para ela. Desta forma, os parâmetros possuem seu próprio nome e tipo, seguidos por um ponteiro para o próximo parâmetro caso haja mais na função.

```c
#ifndef FUNCTION_H
#define FUNCTION_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Types.h"

typedef struct Param {
    Type type;
    char *name;
    void* value;
    struct Param *next;
} Param;


typedef struct Function {
    Type type;
    int num_params;
    void *value;
    Param *params;
} Function;

Function *create_function(Type returnType);
void print_function(Function func);
void add_parameter_list(Function *func, Param **param);
int param_list_length(Param *param);
Param *create_param(char *name, Type type);
void link_params(Param *param, Param *next);

#endif
```

Figura 97: Formato da struct de funções

Expression

A estrutura responsável por armazenar valores das expressões é amplamente utilizada ao longo do código para realizar a análise sintática, cada expressão terá um tipo e um valor associado, que será manipulado durante a análise para informar se existe alguma inconsistência ou se a expressão é válida.

```c
#ifndef EXPRESSION_H
#define EXPRESSION_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "Types.h"

typedef struct Expression {
  Type type;
  void *value;
} Expression;

#endif
```

Figura 98: Formato da struct de expressões

Types

A estrutura de tipo é um enumerador que possui um identificador inteiro para cada tipo possível na linguagem, começando com os tipos primitivos e avançando até os mais complexos como arrays, strings e ***ponteiros***.

```c
#ifndef TYPES_H
#define TYPES_H

typedef enum {
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_DOUBLE,
    TYPE_CHAR,
    TYPE_BOOL,
    TYPE_FUNC,
    TYPE_VOID,
    TYPE_STRUCT,
    TYPE_UNION,
    TYPE_ENUM,
    TYPE_LONG,
    TYPE_SHORT,
    TYPE_POINTER,
    TYPE_ARRAY,
    TYPE_STRING
} Type;

char *type_to_string(Type type);

#endif
```

Figura 99: Enum com todos os tipos disponíveis

Operators

O arquivo de operadores inclui uma série de enumeradores responsáveis por indicar ao Yacc qual o tipo de operador está sendo passado pelo lex. Desta forma, os retornos não passam um token diferente para um dos operadores, mas sim um único token com um valor relacionado.

```
#ifndef OPERATORS_H
#define OPERATORS_H

typedef enum ArOp {
    PLUS,
    MINUS,
    MULT,
    DIV,
    MOD
} ArOp;

typedef enum RelOp {
    EQ,
    NE,
    LT,
    GT,
    LE,
    GE
} RelOp;
```

Figura 100: Enum de operadores lógicos e aritméticos

```
{maior} { yylval.relOp = GT; return RELOP; }
{menor} { yylval.relOp = LT; return RELOP; }
{maior_igual} { yylval.relOp = GE; return RELOP; }
{menor_igual} { yylval.relOp = LE; return RELOP; }
{diferente} { yylval.relOp = NE; return RELOP; }
{igualdade} { yylval.relOp = EQ; return RELOP; }
```

Figura 101: Atribuição dos operadores no lex

Como pode ser visto acima, cada token está sendo retornado com um valor juntamente com o token RELOP.

Análise Semântica

Após a implementação da tabela de símbolos, algumas verificações semânticas puderam ser implementadas com facilidade. Alguns dos possíveis erros semânticos já detectados são:

- Conflito entre tipos;

```
decl_var: type type_qualifier ID opt_assignment
        {
         if ($4->type != $1 && $4->type != TYPE_VOID) {
            yyerror("\033[0;34mInscricao Arcana\033[0m nao inicializada com o tipo correto...\n");
         }
         Symbol *new_symbol = insert_symbol(current_table, $3, $1, $4->value);
        }
```

Figura 102: O tipo do identificador não bate com o valor atribuído ao mesmo.

- Conflito no número de parâmetros, tipos de parâmetros ou chamada incorreta;

```
function_call: CALLFUNC ID PARAMS OPENBRACK params CLOSEBRACK
        {
          Symbol *symbol = lookup_symbol(current_table, $2);
          if (symbol == NULL) {
            yyerror("\033[0;32mMagia\033[0m nao encontrada...\n");
          }
          if (symbol->type != TYPE_FUNC) {
            yyerror("Simbolo nao e uma \033[0;32mmagia\033[0m...\n");
          }
          Function *func = (Function *)symbol->value;
          Param *param = $5;
          if (func->params != NULL) {
            if (param_list_length(param) != param_list_length(func->params)) {
              yyerror("Numero de componentes incorreto...\n");
            }
            print_function(*func);
            Param *current = func->params;
            while (current != NULL) {
              if (current->type != param->type) {
                yyerror("Tipo de componente incorreto...\n");
              }
              current = current->next;
              param = param->next;
            }
          }
```

Figura 103: Erro na chamada da função, seja por não existir, pelo número ou pelo tipo dos parâmetros.

- Conflito de operação em tipo inválido;

```
void* perform_arithmetic(Expression left, Expression right, ArOp op) {
    switch (left.type) {
        case TYPE_INT:
            return perform_int_arithmetic(left.value, right.value, op);
        case TYPE_FLOAT:
            return perform_float_arithmetic(left.value, right.value, op);
        case TYPE_DOUBLE:
            return perform_double_arithmetic(left.value, right.value, op);
        case TYPE_LONG:
            return perform_long_arithmetic(left.value, right.value, op);
        case TYPE_SHORT:
            return perform_short_arithmetic(left.value, right.value, op);
        default:
            yyerror("Tipo de \033[0;36minscricao arcana\033[0m nao suportado...\n");
    }
}
    You, 5 hours ago • pointer void
```

Figura 104: Operação não suportada para algum tipo.

- Símbolo não encontrado na tabela;

```
assignment: variable ASSIGN expr
        {
        Symbol *symbol = lookup_symbol(current_table, $1->name);
        if (symbol == NULL) {
            yyerror("\033[0;34mInscricao arcana\033[0m nao encontrada...\n");
        } else {
            assign_value_to_symbol(symbol, $3);
        }
        }
    ;
```

Figura 105: Símbolo não encontrado.

Implementações Pendentes

Tendo em vista o tempo disponível para desenvolver a linguagem, algumas implementações mais elaboradas e complexas se mantêm apenas como ideias ou pendências.

A implementação de um tipo “Array”, um tipo “String”, e de aritmética de ponteiros são ideias que pretendemos implementar ainda na terceira iteração do trabalho. Apesar de complexas, o planejamento para a implementação já está pronto, assim como a implementação sintática. Para os novos tipos de dados, estruturas definidas e implementadas em C serão desenvolvidas, e suas funções utilizadas em ações semânticas. Quanto à aritmética de ponteiros, modificações na tabela de símbolos serão feitas para manipular esse tipo de dados.

Estruturas criadas pelo usuário, como enumerações, estruturas e uniões, estão implementadas na análise sintática, mas sua inserção nas tabelas de símbolos ainda não foram feitas. Ainda não pensamos em uma forma de realizar essa implementação, mas é algo que pretendemos fazer.

```
Transmutar Glifos em Listagem _/
  teste,
  teste1,
  teste2
\_

Transmutar Componentização em Golemita _/
  Glifo a...
  Runa b...
  Arquiglifo c...
  Arquifractal d...
  Fractal e...
  Semiglifo f...
  Axioma g...
\_

Componentização Golemita golem...
Glifos Listagem lista...
```

Figura 106: Definição de tipos pelo usuário.

A execução de funções e gerência do escopo interno das mesmas ainda não foram implementadas, sendo assim, linhas de código no interior de funções são lidas no decorrer da análise, modificando as variáveis nestes escopos.

```
Preparar magia de Glifo Additium, Componentes: [Glifo unnus, Glifo annus]
_/
  Glifo w...
  w será 20...
  Iniciar profecia de z
  _/
    Profetizar que se 1;
      Fractal jota será 1.0...
      z será z fundido a 1...
      Que assim seja...
    Profetizar que se 2;
      z será z fundido a 2...
      Que assim seja...
    Cumprir profecia;
      z será z fundido a 3...
      Que assim seja...
  \_

  z-|1|- será 20...
  z será z...
\_
```

Figura 107: Múltiplos escopos no programa.

Conclusão

Este trabalho prático permitiu a criação da especificação de uma nova linguagem de programação, cobrindo desde a definição de seu nome e origem até a especificação dos tipos de dados primitivos e comandos disponíveis. Com a verificação estática de tipos já implementadas, a estruturação da linguagem no paradigma procedural foi fundamental para seu design, garantindo uma sintaxe coerente e funcional.

Durante o processo, a aplicação da análise sintática desempenhou um papel crucial na definição das regras gramaticais, assegurando que a estrutura da linguagem seja coerente e eficiente para ser processada por um compilador. A elaboração da gramática, incluindo variáveis, terminais (tokens) e padrões de lexema, evidenciou a complexidade envolvida no desenvolvimento de uma linguagem, bem como a importância de garantir clareza e consistência.

A integração da fase inicial da análise semântica, apesar de desafiadora, proporcionou maior entendimento dos conceitos apresentados na disciplina, assim como a implementação prática dos mesmos.

Em suma, a criação desta linguagem de programação proporcionou uma valiosa experiência de aprendizado, aprofundando o entendimento dos princípios fundamentais que orientam o design de linguagens e ferramentas computacionais. O grupo está preparado para continuar o projeto com confiança nas próximas etapas.

Referências

GUPTA, Ajay. *The syntax of C in Backus-Naur form*. Disponível em: https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm. Acesso em: 21 ago. 2024.

WIKIPEDIA. *Extended Backus–Naur form*. Disponível em: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form. Acesso em: 21 ago. 2024.

CHUBEK. *pygame-template.py*. 2021. Disponível em: https://gist.github.com/Chubek/52884d1fa766fa16ae8d8f226ba105ad. Acesso em: 21 ago. 2024.

WESTES, Vern Paxson et al. *Flex: The Fast Lexical Analyzer*. Disponível em: https://westes.github.io/flex/manual/. Acesso em: 21 ago. 2024.

UNESP. *Lex & Yacc.* Disponível em: https://www.dcce.ibilce.unesp.br/~aleardo/cursos/compila/lex.html. Acesso em: 21 ago. 2024

Stack Overflow. Stack Overflow Home Page. Disponível em: https://stackoverflow.com/. Acesso em: 7 ago. 2024.

Misra, Bivas. Lex and Yacc Tutorial. Disponível em: https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf. Acesso em: 7 ago. 2024.