UNIVERSIDADE FEDERAL DE VIÇOSA
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CCF441 – COMPILADORES
LINGUAGEM TAO – ENTREGA 1 DO TRABALHO PRÁTICO

GRUPO:
3873 – Germano Barcelos dos Santos
3051 – Henrique de Souza Santana
3890 – Otávio Santos Gomes
3877 – Pedro Cardoso de Carvalho Mundim
3495 – Vinícius Júlio Martins Barbosa

Relatório de trabalho prático da disciplina CCF441 – COMPILADORES, apresentado à Universidade Federal de Viçosa.

FLORESTAL
MINAS GERAIS - BRASIL
JUNHO 2022

SUMÁRIO

1 INTRODUÇÃO

Nesta primeira parte do trabalho prático da disciplina CCF441 - Compiladores, o objetivo é definir uma linguagem de programação própria, não só prática, mas também criativa. Além disso, também é objetivo do trabalho de implementar o seu analisador léxico, utilizando para isso o Lex e a linguagem de programação C. Dessa forma, espera-se obter um conhecimento mais aprofundado da estrutura de um analisador léxico, além de destacar características de linguagens de programação que geralmente só são observadas por projetistas de uma linguagem. Assim, apresentamos a especificação de uma linguagem, bem como seu analisador léxico a seguir.

2 ESPECIFICAÇÃO

Nesta seção será mostrada a especificação da linguagem, apresentando o nome, origem do nome, tipos de dados, comandos, palavras-chave, dentre outros.

2.1 NOME DA LINGUAGEM E ORIGEM

O nome da linguagem é **Tao**, seu logotipo é mostrado na Figura 1, e tem origem na tradição filosófica e religiosa chinesa do taoismo, que enfatiza a harmonia com o *tao* (, "caminho") (1). Devido a características da linguagem que serão descritas nas seções seguintes, a linguagem Tao busca deixar o programador como responsável por melhor "harmonizar"o código e explicitamente gerenciar tipos e memória.

Três conceitos do taoismo foram incorporados no léxico da linguagem. O primeiro deles é o do *yin* e *yang*. Ambos são forças opostas e complementares, que se originam do *wuji* – um "vazio primordial". O segundo conceito é o dos trigramas. Yin e yang se combinam nos chamados oito trigramas, que representam princípios fundamentais da realidade correlacionados. Como mostra a Figura 2, yin é representado por uma barra quebrada, e yang por uma barra completa. Se considerarmos o yin como valor 0, e o yang como valor 1, e se lermos os trigramas de baixo para cima, podemos associar a cada um deles um valor codificado em binário.

Figura 1 – Logotipo da linguagem.

Além disso, o terceiro conceito, emprestado do *I Ching* (, geralmente traduzido como "O Livro das Mudanças") (2), é que esses oito trigramas também se recombinam, formando 64 hexagramas. Os hexagramas igualmente podem ser associados a valores binários, apesar dessa associação não ser usada nos textos clássicos. Cada hexagrama também representa uma ideia, um conceito, e buscamos associá-los a construções de programação.

As palavras `wuji`, `yin` e `yang` se tornaram palavras reservadas, e cada trigrama e hexagrama também, escritos como sequências de, respectivamente, 3 ou 6 símbolos `:` para yin ou `|` para yang. Por exemplo, o trigrama do trovão se tornou `|::` e o do vento se tornou `:||`.

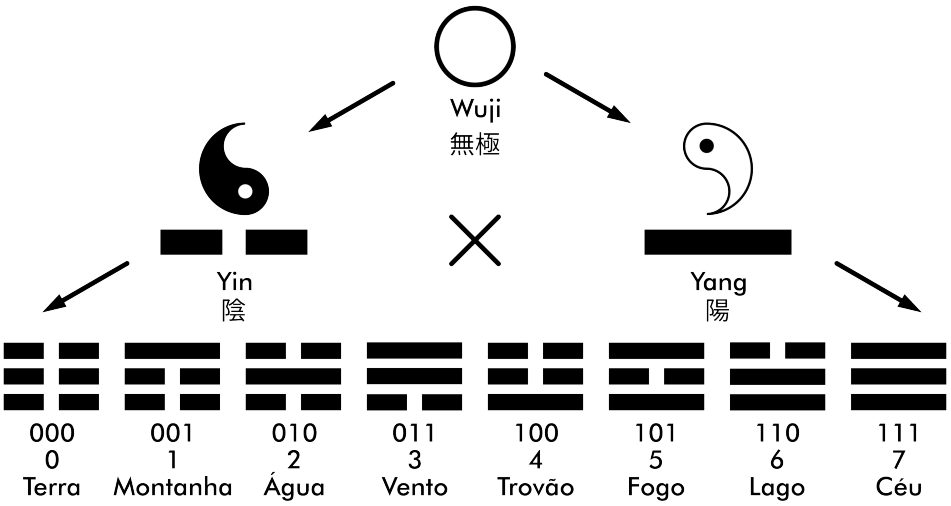

Figura 2 – Os trigramas e sua origem.

2.2 PARADIGMA DE PROGRAMAÇÃO

Como requisitado na especificação do trabalho, o paradigma de programação da linguagem é **procedural**. Além disso, tanto quanto possível, buscamos tornar a linguagem orientada a expressões, de forma que, por exemplo, atribuição e condicional (*if*) podem ser usados como expressões. Deixamos em aberto a possibilidade de definir funções aninhadas e passá-las como parâmetro para outras funções, incorporando aspectos do paradigma funcional. Por fim, pretendemos implementar mecanismos simples de polimorfismo de coerção, de sobrecarga e paramétrico.

2.3 LÉXICO

Um primeiro aspecto importante do léxico da linguagem se refere aos identificadores, que podem ser comuns, próprios ou simbólicos. Identificadores próprios devem começar com uma letra maiúscula, seguida de uma quantidade indefinida de letras, dígitos ou traços baixos (_). Os comuns seguem a mesma regra, mas começam com letra minúscula ou traço baixo. Identificadores simbólicos contém uma ou mais repetições dos caracteres `!$%&*+./<=>?|:#@^~-`.

Todo identificador, seja comum, próprio ou de símbolo, pode ser prefixado com um escopo, caso seja importado de algum módulo. Esse prefixo é uma sequência de um ou mais identificadores próprios seguidos de um ponto final. Por exemplo, uma função chamada `function` importada do módulo `Some.Module` pode ser referida como `Some.Module.function`. Identificadores escritos assim são chamados identificadores qualificados, e são considerados um único lexema. Identificadores comuns nomeiam variáveis, funções e procedimentos, identificadores simbólicos nomeiam operadores, e identificadores próprios nomeiam tipos e módulos.

Por simplicidade e também clareza de código desenvolvido na linguagem, optamos por definir todas as palavras-chave como palavras reservadas, evitando assim ambiguidades. Na verdade, como supracitado, todos os hexagramas e trigramas são palavras reservadas, mas nem todos os hexagramas tem um significado atribuído (ainda). Dessa forma, é possível apontar as seguintes palavras-chave:

- `wuji` – o wuji representa o vazio primordial, então foi escolhido para definir procedimentos, o análogo em C para função de retorno *void*.
- `yin` – o yin é a força negativa/passiva, então escolhemos esta palavra-chave para criar as entidades de programação que armazenam valores, ou seja, definir variáveis.
- `yang` – o yang é a força positiva/ativa, então escolhemos esta palavra-chave para criar as entidades de programação que produzem valores, ou seja, definir funções.

Em seguida, temos os trigramas, que são associados a conceitos mais específicos. Junto aos trigramas temos os hexagramas, que são agrupados de acordo com seu prefixo. Por exemplo, o hexagrama `:|:|::` possui o prefixo do trigrama da água (`:|:`), então será usado em construções relacionadas a este trigrama. Nas Seções 2.6 e 2.7 serão mostrados os hexagramas usados em cada construção específica.

- `:::` – representa receptividade, está relacionado a definição e declaração de tipos;
- `::|` – representa estabilidade, está relacionado a alocação de memória;
- `:|:` – representa movimento, está relacionado a condicionais;
- `:||` – representa pervasividade, está relacionado a repetição;
- `|::` – representa perturbação, está relacionado a liberação de memória;
- `|:|` – representa iluminação, está relacionado a verificação de tipos e valores;
- `||:` – representa abertura, está relacionado a importação de módulos;
- `|||` – representa criação, está relacionado a exportação de módulos.

Além dessas palavras reservadas, alguns símbolos são reservados para controlar a sintaxe da linguagem. Isso é importante, pois, como veremos na Seção 2.7, o programador pode definir novos operadores. Embora `@`, `:`, `=` e `.` possam ser usados como identificadores de operadores – desde que acompanhados de outros símbolos –, os demais símbolos da lista a seguir *nunca* podem ser usados como operadores.

- Todo comando deve encerrar com ponto e vírgula (`;`) ou com nova linha, então esses lexemas representam o mesmo token de fim de linha;
- Caracteres arroba (`@`) prefixam nomes de tipo, indicando que trata-se de ponteiro;
- Dois pontos (`:`) indicam o tipo da variável que o precede;
- Sinal de igual (`=`) é usado em atribuições e definições;

- Ponto final (`.`) é usado para acessar campos em estruturas e módulos;
- Vírgula (`,`) é usada como separador em listas de parâmetro;
- Colchetes (`[]`) são usados para acessar posições em ponteiros;
- Parênteses (`()`) aninham expressões, e chaves (`{}`) aninham blocos.
- Comentários de uma única linha são indicados com dois apóstrofos (`''`), e comentários multilinha são indicados com três apóstrofos no começo e três apóstrofos no final (`'''`).

2.4 TIPOS DE DADOS

A linguagem Tao possui os seguintes tipos primitivos:

- `Int` – valores inteiros de 32 bits com sinal;
- `Long` – valores inteiros de 64 bits com sinal;
- `Real` – valores de ponto flutuante de 64 bits;
- `Char` – valores de caractere de 8 bits;
- `Bool` – valores booleanos;
- `@Type` – ponteiro para `Type`, sendo `Type` qualquer outro tipo.

Uma observação importante é quanto a notação de tipos para funções, não inclusa na lista acima. Quando uma variável na verdade guarda um ponteiro para função, no lugar da anotação de tipo normalmente usada, deve ser usado o hexagrama `|||::|` seguido de uma lista de tipos, e opcionalmente seguido novamente pelo mesmo hexagrama e um único tipo. Quando o hexagrama aparece apenas uma vez, trata-se de um ponteiro para um subprograma que espera como parâmetro os tipos listados. Quando o hexagrama aparece duas vezes, trata-se de um ponteiro para uma função que espera como parâmetro os tipos listados depois do primeiro hexagrama, e retorna valores do tipo mencionado depois do segundo hexagrama. Por exemplo, `|||::| Int, Int |||::| Bool` denota funções que recebem dois inteiros e retorna booleano, `|||::| @Char` denota um subprograma que recebe um ponteiro para caractere.

Além disso, é possível definir tipos compostos, usando uma combinação dos mecanismos de produto cartesiano e união disjunta. Ao definir novos tipos, o programador pode primeiramente definir uma lista de construtores para aquele tipo (união disjunta), e cada construtor pode ter zero ou mais campos nomeados, de tipos arbitrários (produto cartesiano). O Algoritmo 2.1 mostra exemplos dessa definição.

```
'' três construtores, nenhum campo
::: Color = Red, Green, Blue

'' um construtor com dois campos
::: Person = Person(name: @Char, age: Int)

'' três construtores, com campos variados
::: Tree = Node(x: Int, lt: @Tree, rt: @Tree), Leaf(x: Int), Empty
```

Algoritmo 2.1 – Tipos compostos.

2.5 LITERAIS

Cada tipo primitivo possui literais associadas. Valores inteiros são expressados por sequências de dígitos decimais, ou por dígitos hexadecimais, prefixados com `0x`. Valores reais são expressados com dígitos decimais separados com um ponto, seguidos opcionalmente por um expoente em notação científica.

Caracteres são expressados por dois apóstrofos com um caractere ou uma sequência de escape dentro. Sequências de escape começam com contrabarra (`\`) e são seguidas ou de um valor hexadecimal arbitrário entre `00` e `ff`, (ex.: `'\x20'`), ou então são sequências bem conhecidas como `'\n'` de nova linha ou `'\t'` de tabulação. Para expressar o prório apóstrofo também é necessário esse escape (`'\''`). Apesar de não haver strings como tipos primitivos, strings são vetores de caracteres, como em C, que usa ponteiros, e podem ser expressas em literais. As regras são as mesmas para literais de caracteres, porém ao invés de apóstrofos, usa-se aspas, a string pode ser vazia, e ao invés de ser necessário usar escape no apóstrofo, deve ser usado o escape nas aspas.

Literais booleanas não são exatamente literais, mas sim uma expressão de construção de tipo. O tipo `Bool` é definido como `::: Bool = False, True`, e portanto seus valores são usados como `False` e `True`.

2.6 COMANDOS

Os comandos disponíveis são: exportação de módulo, importação, definição de variável, definição de função, definição de operador, definição de procedimento, definição de tipo, declaração de tipo, chamada de procedimento, condicional, casamento de tipos, repetição (*while* e *repeat*) e liberação de memória.

O Algoritmo 2.2 só aparece opcionalmente no topo do arquivo e não se repete. Utilizamos o trigrama `|||` para denotar a exportação e o hexagrama `|||:::` para listar quais procedimentos, funções, variáveis ou tipos serão exportadas.

Com relação à importação de módulos, utilizamos o trigrama `||:` para denotar a importação, o hexagrama `||:::||` para especificar o que se deseja importar de dentro de um módulo. Além disso, o hexagrama `||:||:` pode ser usado para renomear determinado módulo

```
'' arquivo1.tao
||| This.Module.Name |||::: proc1, func2, var4

'' arquivo2.tao
||| Another.Module |||::: var8, proc16, Type32
```

Algoritmo 2.2 – Comando de exportação de módulo.

```
'' equivalente em Python:
'' from src.patricia import *
||: Src.Patricia

'' from math import sin, cos, tan
||: Math ||::|| sin, cos, tan

'' import sqlite3 as sql
||: SQLite3 ||:||: SQL
```

Algoritmo 2.3 – Comando de importação.

importado, facilitando sua utilização posteriormente no código (Alg. 2.3). Nesse último caso, os identificadores pertencentes a esse módulo serão acessados como identificadores qualificados (ex.: `SQL.connect`).

```
:::::: String = @Char

::: Person = Person(name: @Char, age: Int)

yin pi: Real = 3.1415926535

yang initial(p: Person): Char = p.name[0]

'' ::::|| lista parâmetros de tipo para
'' a função/procedimento/operador que vier em seguida
::::|| T
yang len(list: @List(T)): Int = { ''' expresão '''  }

wuji main() { }
```

Algoritmo 2.4 – Comando de declaração de tipo, e comandos de definições: tipos, variáveis, funções e procedimentos.

O Algoritmo 2.4 apresentam os comandos de definição e declaração. A declaração de tipo é dada pelo hexagrama `::::::`, que atribui um novo nome a um tipo já existente. Já com o trigrama `:::`, é possível declarar novos tipos de dados, como especificado na Seção 2.4. `yin` é

utilizado em uma definição de variável. Nessa definição, após o comando `yin` aponta-se o nome da variável seguido de dois pontos e seu tipo, podendo já ser inicializada com um valor.

`yang`, por sua vez, é o comando responsável pela a definição de funções. Utiliza-se o comando `yang` seguido do nome da função, seus parâmetros e tipos dentro de parênteses. Em sequência, coloca-se `:` e o tipo de retorno da função. Por fim, utiliza-se um `=` e o código pode estar em sequência entre chaves, caso seja multilinhas, ou apenas a linha a ser executada logo após o `=`. Por fim, `wuji` pode ser utilizado para definir subprogramas, que são a mesma coisa que funções, porém não retornam nenhum valor e sua chamada não pode ser usada como expressão.

No corpo de subprogramas e funções, é possível usar o comando `|||||:`, seguido de uma expressão, para encerrar a execução imediatamente e retornar o valor dessa expressão.

```
'' operadores associativos à esquerda com :::||:
'' número indica a precedência
yang :::||: 5 <<(x: Real, exp: Real): Real = { ''' expressão '''  }
yang :::||: 5 >>(x: Real, exp: Real): Real = { ''' expressão '''  }

'' operadores associativos à direita com :::|:|
yang :::|:| 5 +>(item: Int, head: @List): @List = {
    ''' expressão '''
}

'' operadores não associativos com :::|::
yang :::|:: 4 ==(s0: @Char, s1: @Char): Bool = {
    ''' expressão '''
}
```

Algoritmo 2.5 – Comando de definição de operadores.

Veja no Algoritmo 2.5 que é possível também definir operadores. Essa definição se dá como na definição de uma função. No entanto, é preciso indicar a precedência do operador. Além disso, deve-se ter exatamente 2 parâmetros. Dessa forma, define-se um operador da seguinte forma: utiliza-se o comando `yang` seguido do hexagrama `:::||:`, `:::|:|` ou `:::|::`, seguido de sua precedência. Esses hexagramas indicam a associatividade do operador (esquerda, direita ou nenhuma respectivamente). Logo em sequência, apresenta-se o nome do operador, seguido de qual o nome e o tipo dos seus 2 parâmetros. Por fim, apresenta-se seu valor de retorno e enfim descreve-se a sua execução no bloco de código.

No Algoritmo 2.6 mostramos que a chamada de um procedimento é bastante simples, podendo-se passar quaisquer parâmetros, conforme estabelecido na definição do procedimento. A sintaxe de chamada de procedimento e de função é exatamente igual, porém futuramente na análise sintática é que deve ser conferido se uma chamada de procedimento não está sendo usada como expressão.

Veja no Algoritmo 2.7 a implementação do comando condicional. Nele, o trigrama `:|:`

```
wuji dfs(node: @Tree) {  }

yin tree: @Tree
dfs(tree)
```

Algoritmo 2.6 – Comando de chamada de procedimento.

```
:|: cond :|:|:: {
    '' executa se cond == True
}

:|: cond :|:|:: {
    '' executa se cond == True
} :|::|| {
    '' executa se cond == False
}

:|: cond :|:|:: {
    '' executa se cond == True
} :|::|: cond2 :|:|:: {
    '' executa se cond == False e cond2 == True
} :|::|| {
    '' executa caso contrário
}
```

Algoritmo 2.7 – Comando condicional.

representa o tradicional comando *se*, e o hexagrama `:|:|::` aparece depois da condição. *Senão* é representado pelo hexagrama `:|::||`, e aparece opcionalmente. Uma quantidade indefinida de *senão-ses* pode aparecer depois da primeira condição, indicados pelo hexagrama `:|::|:`.

```
::: Tree(K) = Node(x: K, lt: @Tree(K), rt: @Tree(K)), Leaf(x: K)

yin tree: @Tree(Int) = '' expressão
|:| @tree
|:|||| Node(x, l, r) |:|:|: { '''executa se for Node''' }
|:|||| Leaf(x) |:|:|: { '''executa se for Leaf''' }
|:||:: { '''default''' }
```

Algoritmo 2.8 – Comando de casamento de tipo.

No Algoritmo 2.8 mostramos a estrutura de um comando similar a um `switch-case` da linguagem C. O comando é definido pelo trigrama `|:|` seguido de qual expressão será utilizada na análise feita pelo comando. O hexagrama `|:||||` serve para realizar o casamento, caso seja verdadeiro, será executado o código presente em sequência, depois do hexagrama `|:|:|:`.

Por fim, o hexagrama `|:|::` representa o caso padrão, que será executado quando nenhuma possibilidade anterior for executada.

```
:|| cond :||||| {
    '' executa enquanto cond == True
    :||:|: '' pula para a próxima iteração, comando "continue"
}

:|| cond :|||:: {'''executa depois do laço'''} :||||| {
    '' executa enquanto cond == True
}
```

Algoritmo 2.9 – Comando de repetição com teste antes do laço.

No Algoritmo 2.9, temos os comandos responsáveis pela repetição. O trigrama `:||` representa o inicio do comando de repetição. Ele é sempre seguido de uma condição. Depois pode ser seguido de diferentes hexagramas. Se seguido de `:|||||`, executará enquanto o comando descrito for verdadeiro. Se, por sua vez, a condição for seguida do hexagrama `:|||::`, este será seguido de um bloco de código a ser executado após a finalização do laço. Por fim, deve ser seguido do hexagrama `:|||||` que terá um trecho de código que será executado enquanto a condição apresentada for verdadeira.

```
:||:||  {
    '' executa até que cond == True
    :||||: '' sai imediatamente do laço, comando "break"
} :||::| cond

:||:||  {
    '' executa até que cond == True
} :||::| cond :|||:: { '''executa depois do laço''' }
```

Algoritmo 2.10 – Comando de repetição com teste depois do laço.

No Algoritmo 2.10, por sua vez, utilizamos o hexagrama `:||:||` seguido de um bloco de código, seguido do hexagrama `:||::|` e da condição de execução desse trecho de código. Esse código será executado até que a condição se torne verdadeira, porém ao contrário da estrutura apresentada anteriormente, executará pelo menos uma vez, sendo checada a condição apenas após a execução.

Em ambos os comandos, é possível usar os hexagramas `:||||:` e `:||:|:` para controlar o fluxo da repetição, sendo que o primeiro é o comando que sai do laço imediatamente (`break`), e o segundo é o que pula para uma próxima iteração do laço (`continue`).

No Algoritmo 2.11, observe que utilizamos o trigrama `|::` para a liberação de memória. Fazendo um paralelo com a linguagem C, funcionaria como uma chamada à função `free()`.

```
yin list: @List = '' expressão
|:: list
```

Algoritmo 2.11 – Comando de liberação de memória.

Precedência	Operadores	Associatividade
8	**	Direita
7	* / & ^	Esquerda
6	+ - |	Esquerda
5	<< >>	Esquerda
4	== != < <= >= >	
3	&&	Esquerda
2	||	Esquerda
1	(nenhum)	

Tabela 1 – Operadores predefinidos.

2.7 EXPRESSÕES

Toda expressão pode ser aberta numa sequência de expressões e comandos, usando chaves (`{}`), e, nesse caso, o resultado da última expressão do bloco se torna o resultado do todo. Expressões também podem ser aninhadas com parênteses.

A proposta da linguagem permite ao programador definir seus próprios operadores, mas já possui operadores predefinidos (que podem inclusive ser sobrecarregados), além de algumas expressões especificadas com os trigramas e hexagramas. A Tabela 1 mostra os operadores predefinidos, suas precedências e suas associatividades. Quanto maior o valor de precedência, maior a prioridade. Os operadores matemáticos `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `>=` e `>` têm seu significado intuitivo. O operador `**` se refere à exponenciação. Os operadores `&`, `|`, `^`, `<<` e `>>` são binários, e se referem respectivamente às operações *and*, *or*, *xor*, *shift-left* e *shift-right*. Por fim, os operadores lógicos `&&` e `||` também têm seu significado intuitivo.

No total, as expressões disponíveis são: condicional, casamento de tipo, atribuição (`=`), acesso a campo (`.comid`), acesso a posição (`[i]`), acesso direto (`@`), referenciação (`$`), negação binária (`~`), negação lógica (`!`), negação numérica (`-`), operações infixadas com ordem de precedência de 1 a 8, literais, variáveis, alocação de memória, construção de tipo composto e chamada de função.

As expressões de casamento de tipo e a condicional seguem a mesma sintaxe de quando são usadas como comandos, com a diferença de que o "else"do condicional é obrigatório. As expressões de negação e de manipulação de ponteiro são os únicos operadores prefixados. Exemplos de cada expressão são mostrados no Algoritmo 2.12.

```
'' expressão de atribuição,  e diferentes níveis de operações infixadas
x = y = 3 | 2 ** 3 >> 0xA + 0xf - 23
'' equivalente à seguinte expressão:
x = (y = ((3 | (2 ** 3)) >> ((0xA + 0xf) - 23)))

'' negações
cond = !(-x < ~y)

'' manipulação de ponteiros
y = @($x + 10)

'' acesso a posição, seguido de acesso a campo
people[10].name = "Alfred Aho"

'' alocação de memória no monte
yin p: @Person = ::| Person
'' alocação de memória na pilha
p = ::||:: Person
'' alocação de 10 posições no monte
p = ::| Person ::|:|| 10
'' alocação de 10 posições na pilha
p = ::||:: Person ::|:|| 10
'' construção de tipo
@p = Person("Fulano", 42)
'' alocação + construção
p = ::| Person("Fulano", 42)
'' alocação + construção de 20 posições
p = ::| Person("", 0) ::|:|| 20

'' chamada de função
x = a + random() * (b - a)
```

Algoritmo 2.12 – Expressões.

2.8 SINTAXE

A gramática da linguagem será expressada na Forma de Backus-Naur (BNF) (3). Nessa notação, as variáveis da gramática são expressas `<desta-forma>`, as produções são indicadas por `::=`, e terminais são indicados enter aspas. Evidentemente, todo terminal seria representado por um token, mas, para facilitar a representação de tokens cujos lexemas podem variar, estendemos essa notação usando parênteses, por exemplo `(LiteralInt)` se refere uma literal de inteiro.

Especificamente, esses são os tokens para os quais usamos essa notação:

- `(LiteralInt)` – literal de número inteiro;
- `(LiteralReal)` – literal de número real;

- `(LiteralChar)` – literal de caractere;
- `(LiteralString)` – literal de string;
- `(Endl)` – fim de linha;
- `(TrigN)` – trigrama de valor N, ex.: `(Trig6)` é `||:`;
- `(HexN)` – hexagrama de valor N, ex.: `(Hex42)` é `|:|:|:`;
- `(ComID)` – identificador comum;
- `(ProID)` – identificador próprio;
- `(SymID)` – identificador simbólico;
- `(QComID)` – identificador comum qualificado;
- `(QProID)` – identificador próprio qualificado;
- `(QSymID)` – identificador simbólico qualificado.

Assim, segue a GLC da linguagem, segmentada em partes relacionadas para facilitar seu entendimento. Como convencional, a primeira variável mostrada (`<program>`) é a variável de partida.

```
1  <program> ::= <module-decl> <top-stmts>
2
3  <module-decl> ::= {Trig7} <pro-id> {Hex56} <exports> {Endl} | ""
4  <exports> ::= <export> "," <export-id> | <export-id>
5  <export-id> ::= {ComID} | {ProID} | {SymID}
6
7  <top-stmts> ::= <top-stmts> {Endl} <top-stmt> | <top-stmt>
8  <top-stmt> ::= <import>
9               | <callable-def>
10              | <def-type-params> <callable-def>
11              | <var-def>
12              | <type-def>
13              | <type-alias>
14              | ""
```

Algoritmo 2.13 – Início da GLC.

No Algoritmo 2.13 são mostradas as construções que devem aparecer no topo de um programa, antes de entrar em qualquer bloco. Como supracitado, a declaração de módulo é opcional e necessariamente deve ser a primeira linha do programa, representada pela variável `<module-decl>`. A variável `<top-stmts>` é usada para permitir que apenas comandos de definição e declaração possam ser usados fora de algum bloco.

Em seguida, a variável <stmts> (Alg. 2.14) é a que permite quaisquer comandos, e aparecerá no corpo de outras produções mais a frente.

```
<stmts> ::= <stmts> (Endl) <stmt> | <stmt>
<stmt> ::= <import>
         | <callable-def>
         | <def-type-params> <callable-def>
         | <var-def>
         | <type-def>
         | <type-alias>
         | <call>
         | <if-stmt>
         | <match>
         | <while>
         | <repeat>
         | <free>
         | <break>
         | <continue>
         | <return>
         | <expr>
         | ""
```

Algoritmo 2.14 – Comandos expressos na GLC.

O Algoritmo 2.15 mostra as variáveis que expressam as construções de definição e declaração de tipos, variáveis, funções, etc., além do comando de importação.

No Algoritmo 2.16 temos alguns comandos e expressões que podem ser usadas como comandos. No caso, o comando condicional, com o detalhe que sua sintaxe é ligeiramente diferente quando usado como expressão. Isso pode ser visto no corpo das produções referentes às variáveis <if-stmt> e <if-expr>. O comando de casamento de tipos não muda a sintaxe quando usado como expressão, então não foi necessário distinguir. Comandos auxiliares de controle de fluxo também aparecem aqui, com os hexagramas que correspondem ao `break`, `continue` e `return`.

As expressões na GLC requerem uma atenção maior (Alg. 2.17), devido aos diferentes níveis de precedência e associatividade. Depois de permitir a abertura de expressões em blocos, depois da possibilidade de usar expressões condicionais, de casamento de tipos, de atribuição, de endereçamento e operações unárias prefixadas, vêm os operadores infixados. As variáveis <expr-N>, <expr-n-N>, <expr-l-N> e <expr-r-N> na verdade representam as variáveis <expr-1>, <expr-n-1>, <expr-l-1>, <expr-r-1> até <expr-8>, <expr-n-8>, <expr-l-8> e <expr-r-8>, e onde aparece `N+1` no nome das variáveis e tokens no corpo de suas produções, deve-se trocar pelo valor de $N + 1$. Por exemplo, a variável <expr-l-8> produz <expr-l-8> `(SymIDl8)` <expr-9>.

Primeiramente, a diferenciação entre <expr-N>, <expr-n-N> e <expr-l-N> serve para

```
<import> ::= (Trig6) <pro-id>
           | (Trig6) <pro-id> (Hex51) <export-list>
           | (Trig6) <pro-id> (Hex54) (ProID)

<def-type-params> ::= (Hex03) <type-params>
<callable-def> ::= <func-def> | <op-def> | <proc-def>

<var-def> ::= "yin" (ComID) ":" <type-id>
            | "yin" (ComID) ":" <type-id> "=" <expr>

<func-def> ::= "yang" (ComID) "(" <param-list> ")" ":" <type-id> "=" <expr>

<op-def> ::= "yang" <op-assoc> <op-prec> (SymID) "(" <param> "," <param>
↪  ")" ":" <type-id> "=" <expr>
<op-assoc> ::= (Hex04) | (Hex05) | (Hex06)
<op-prec> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"

<proc-def> ::= "wuji" (ComID) "(" <param-list> ")" <block>

<type-def> ::= (Trig0) (ProID) <type-param-list> "=" <constructors>
<constructors> ::= <constructors> "," <constructor> | <constructor>
<constructor> ::= (ProID) "(" <param-list> ")"

<type-alias> ::= (Hex00) (ProID) <type-param-list> "=" <type-id>

<param-list> ::= <params> | ""
<params> ::= <params> "," <param> | <param>
<param> ::= (ComID) ":" <type-id>
```

Algoritmo 2.15 – Comandos de definições e declarações na GLC.

indicar a associatividade, fazendo com que a árvore de derivação da gramática siga a associatividade do operador, e também impedindo que associatividades diferentes se misturem num mesmo nível de precedência. Outro detalhe é que os tokens (`SymIDxN`) não aparecem na legenda mostrada no começo desta seção. Isso é porque, pensando numa próxima entrega do trabalho, será necessário que o analisador sintático seja capaz de consultar a tabela de símbolos e verificar, para um dado operador (token (`SymID`)), se possui a associatividade e precedência esperadas.

Chegando na variável `<expr-9>` se encerra os níveis de precedência e enfim a expressão pode se tornar uma literal, um identificador, uma alocação de memória, uma construção de tipo uma chamada de função, ou um aninhamento de expressão com parênteses, recomeçando todo o ciclo. No Algoritmo 2.18 vemos as produções referentes a essas últimas construções.

Ao final da gramática (Alg. 2.19), há algumas variáveis que foram usadas em várias produções não necessariamente relacionadas, com destaque à variável `<type-id>`, que define como um tipo deve ser descrito, e foi uma variável bastante utilizada.

```
<if-expr> ::= (Trig2) <expr> (Hex20) <expr> <elif> <else>
<if-stmt> ::= (Trig2) <expr> (Hex20) <expr> <elif> <else-stmt>
<else-stmt> ::= <else> | ""
<else> ::= (Hex19) <expr>
<elif> ::= <elif> (Hex18) <expr> (Hex20) <expr> | ""

<match> ::= (Trig5) <expr> <cases> <default>
<cases> ::= <cases> <case> | <case>
<case> ::= (Hex47) <case-cond> (Hex42) <expr>
<default> ::= (Hex44) <expr> | ""
<case-cond> ::= <literal> | <decons>
<decons> ::= <pro-id> "(" <com-id-list> ")"
<com-id-list> ::= <com-ids> | ""
<com-ids> ::= <com-ids> "," (ComID) | (ComID)

<while> ::= (Trig3) <expr> <step> (Hex31) <block>
<repeat> ::= (Hex27) <block> (Hex25) <expr> <step>
<step> ::= (Hex28) <block> | ""

<free> ::= (Trig4) <addr>

<break> ::= (Hex30)
<continue> ::= (Hex26)
<return> ::= (Hex62) <expr>
```

Algoritmo 2.16 – Comandos condicional, repetições e liberação de memória na GLC.

3 ANÁLISE LÉXICA

Nesta seção, será mostrado a implementação e desenvolvimento do analisador léxico da linguagem em detalhes. Como dito na introdução, a ferramenta utilizada foi o Lex (4), juntamente com a linguagem C. O arquivo-fonte para o Lex está anexado nos arquivos enviados dentro da pasta /src. Abaixo estão exemplos de entradas e comentários sobre suas respectivas saídas.

As três primeiras linhas do código do Algoritmo 3.2 são os comandos para compilar e executar o analisador lendo o arquivo `match.tao`, e em seguida temos a sua saída. Alternativamente, pode ser executado o comando `make && make run INPUT=match.tao`, usando o arquivo `makefile` preparado.

Primeiramente, é importante observar que organizamos o analisador para fazer a contagem das linhas em que o token analisado está. Com isso, podemos observar que a linguagem ignora o texto que é escrito entre apóstrofos triplos, sendo eles utilizados como comentários, por isso vemos que a linha 1 foi pulada, e, em seguida, foi identificado na linha 2 uma quebra de linha, representada por um `(Endl)`. Em seguida, na linha 3, vemos identificações de termos

```
<expr> ::= "{" <stmts> (Endl) <expr> "}"
         | <if-expr>
         | <match>
         | <assign>
         | <expr-addr>
<expr-addr> ::= <expr-addr> "[" <expr> "]"
              | <expr-addr> "." (ComID)
              | <expr-unary>
<expr-unary> ::= <unary-ops> <expr-1> | <expr-1>
<unary-ops> ::= <unary-ops> <unary-op>
<unary-op> ::= "@" | "$" | "~" | "!" | "-"

<expr-N> ::= <expr-l-N> | <expr-r-N> | <expr-n-N> | <expr-N+1>
<expr-n-N> ::= <expr-N+1> (SymIDnN) <expr-N+1>
<expr-l-N> ::= <expr-l-N> (SymIDlN) <expr-N+1>
<expr-r-N> ::= <expr-N+1> (SymIDrN) <expr-r-N>

<expr-9> ::= <literal>
           | <com-id>
           | <malloc>
           | <build>
           | <call>
           | "(" <expr> ")"
```

Algoritmo 2.17 – Expressões na GLC.

```
<assign> ::= <addr> "=" <expr>
<addr> ::= <addr> "[" <expr> "]"
         | <addr> "." (ComID)
         | <addr-0>
<addr-0> ::= <pointers> <addr-1>
<addr-1> ::= <com-id> | "(" <addr> ")"

<malloc> ::= <malloc-type> <type-id> <malloc-n>
           | <malloc-type> <expr>
<malloc-type> ::= (Trig1) | (Hex12)
<malloc-n> ::= (Hex11) (LiteralInt) | ""

<build> ::= <pro-id> | <pro-id> "(" <exprs> ")"
<call> ::= <com-id> "(" <expr-list> ")"
<expr-list> ::= <exprs> | ""
<exprs> ::= <exprs> "," <expr>
```

Algoritmo 2.18 – Expressões na GLC.

```
<block> ::= "{" <stmts> "}" | <stmt>

<pro-id> ::= (ProID) | (QProID)
<com-id> ::= (ComID) | (QComID)

<type-id> ::= <pointers> <pro-id> <type-param-list>
            | <pointers> (Hex57) <type-param-list> (Hex57) <type-id>
            | <pointers> (Hex57) <type-param-list>
<type-param-list> ::= "(" <type-params> ")" | ""
<type-params> ::= <type-params> "," <type-id> | <type-id>
<pointers> ::= <pointers> "@" | ""
<literal> ::= (LiteralChar) | (LiteralString) | (LiteralInt) |
↪  (LiteralReal)
```

Algoritmo 2.19 – Variáveis auxiliares da GLC.

```
'''A linguagem ignora os comentários
    escritos entre apóstrofes'''
yang add(item: T, tree: @Tree(K, T)): @Tree(K, T) = {
  yin itemKey: K = getKey(item)
  yin newLeaf: @Tree(K, T) = ::| Tree(K, T)
  |:| @tree
  |:|||| Node(k, l, r) |:|:|:
    '' :|: = if
    '' :|:|:: = then
    '' :|::|: = elif
    '' :|::|| = else
    :|: itemKey < k :|:|::
      tree.lt = add(item, l)
    :|::||
      tree.rt = add(item, r)
  |:|||| Leaf(i) |:|:|: {
    yin leafKey: K = getKey(i)
    :|: itemKey < leafKey :|:|::
      ::| Node(leafKey, newLeaf, tree)
    :|::||
      ::| Node(itemKey, tree, newLeaf)
  }
}
```

Algoritmo 3.1 – Código do arquivo `match.tao`.

```
$ flex lex.l
$ gcc lex.yy.c
$ ./a.out < match.tao
2@(Endl)
3@(Yang)
3@(ComId, `add`)
3@(Special, `(`)
3@(ComId, `item`)
...
```

Algoritmo 3.2 – Compilando o arquivo `lex.l` e trecho da saída para `match.tao`.

como `(Yang)`, que define funções, e add, que é um identificador comum, *(ComID)*.

```
...
6@(Trig, 5)
6@(SymPointer)
6@(ComId, `tree`)
6@(Endl)
7@(Hex, 47)
7@(ProId, `Node`)
7@(Special, `(`)
7@(ComId, `k`)
7@(Special, `,`)
7@(ComId, `l`)
7@(Special, `,`)
7@(ComId, `r`)
7@(Special, `)`)
7@(Hex, 42)
7@(Endl)
...
```

Algoritmo 3.3 – Trecho da saída de `match.tao` sobre as linhas 6 e 7.

Analisando o trecho referente às linhas 6 e 7 do arquivo *match.tao* (Alg. 3.3), é possível notar a identificação do trigrama relacionado à verificação de tipos e valores (`|:|`, `(Trig3)`), seguido de um símbolo de ponteiro (`@`, `(SymPointer)`) e um identificador comum. Na linha debaixo é identificado o hexagrama `|:||||`, identificador próprio, símbolos especiais de parênteses e vírgulas, identificadores comum, e por fim outro hexagrama (`|:|:|:`) e quebra de linha.

Observando outro exemplo (Alg. 3.4), podemos ver como é útil a contagem de linha também para apontar erros no código. Primeiramente, podemos destacar que, na linha 1, a literal de inteiro foi devidamente convertida para seu valor numérico. Depois, na linha 4, na tentativa de definir um operador, foi usado um símbolo não previsto no léxico, um acento grave (`` ` ``).

```
yin x: Int = 0xaff

'' operador estranho
yang :::|:: ``(x: Real, y: Real): Int = 2/(1/x + 1/y)
```

Algoritmo 3.4 – Código do arquivo `error.tao`.

```
1@(Yin)
1@(ComId, `x`)
1@(SymTypeTag)
1@(ProId, `Int`)
1@(SymAssign)
1@(LiteralInt, 2815)
1@(Endl)
2@(Endl)
4@(Yang)
4@(Hex, 4)
Lexical unbalance at line 4: unknown symbol ```
Lexical unbalance at line 4: unknown symbol ```
4@(Special, `(`)
...
```

Algoritmo 3.5 – Trecho da saída para `error.tao`.

4 ALTERAÇÕES

5 YACC

6 CONSIDERAÇÕES FINAIS

Consideramos que a especificação de uma linguagem foi uma tarefa interessante uma vez que nos colocou no lugar de projetistas de uma linguagem e nos confrontamos com problemas e dificuldades que muitas vezes observamos em outras linguagens. Percebemos que, muitas vezes, ao facilitar o desenvolvimento da linguagem para o projetista, o desenvolvedor final que utilizará a linguagem ganha mais responsabilidade e pode ter mais dificuldade com o seu uso. Por sua vez, o contrário também é válido, uma vez que a implementação e estruturação de processos mais complexos por parte dos projetistas pode facilitar a vida do programador final.

Por fim, além da especificação da linguagem, a construção do analisador léxico foi relevante para a fixação do conteúdo próprio da disciplina, apresentando-se como uma tarefa desafiadora, mas interessante.

REFERÊNCIAS

1 TAMOSAUSKAS, T. Filosofia chinesa: pensadores chineses de todos os tempos. **Kindle Edition**, 2020.

2 LEGGE, J. **The I Ching**. [S.l.]: Courier Corporation, 1963. v. 16.

3 MCCRACKEN, D. D.; REILLY, E. D. Backus-naur form (bnf). In: **Encyclopedia of Computer Science**. [S.l.: s.n.], 2003. p. 129–131.

4 NIEMANN, T. **A Compact Guide to Lex & Yacc**. [S.l.]: ePaper Press, ????