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

CCF441 – COMPILADORES
LINGUAGEM TAO – ENTREGA 2 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
JULHO 2022

SUMÁRIO

1 INTRODUÇÃO

Na 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.

Na segunda parte do trabalho, o objetivo é realizar a análise sintática, utilizando para isso o gerador de analisador sintático Yacc, integrando com o Lex. Ao final desta parte, nos exemplos, será impresso como saída o programa fonte com linhas enumeradas, o conteúdo da tabela de símbolos e mensagens alertando se o programa está sintaticamente correto ou não. Assim, espera-se obter um maior conhecimento sobre a análise sintática vista de forma teórica na disciplina, mostrada aqui de forma prática.

2 ESPECIFICAÇÃO

Nesta seção é 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* (道, "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* (易經, 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

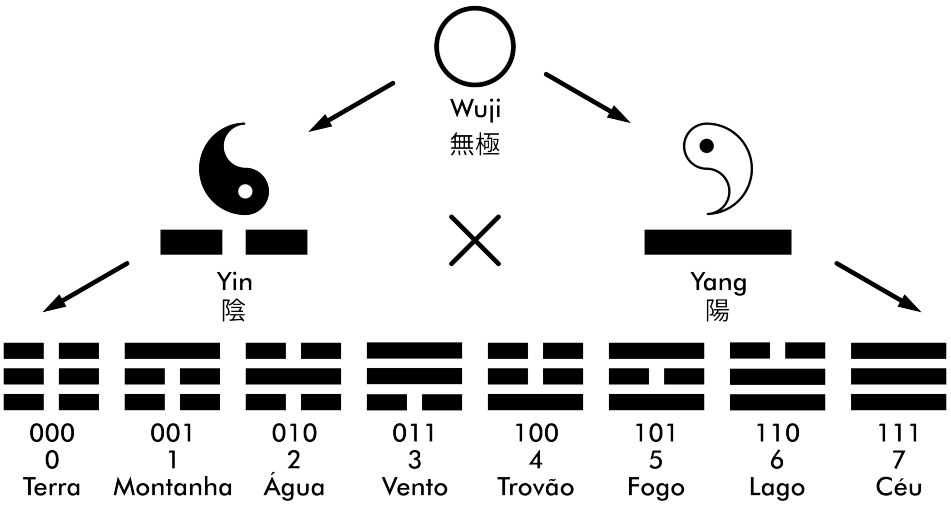


Figura 2 – Os trigramas e sua origem.

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 `:||`.

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.

- Todos os comandos devem ser separados por ponto e vírgula (`;`). Anteriormente, era possível separar com nova linha, mas decidimos retirar essa opção, pois causava problemas na gramática;
- Caracteres arroba (`@`) prefixam nomes de tipo, indicando que trata-se de ponteiro/vetor. Esse caractere pode também ser utilizado para acessar um ponteiro quando utilizado de maneira sufixada. Equivale a acessar um ponteiro utilizado [0]. Anteriormente, esse acesso era prefixado, mas essa mudança simplificou a gramática e a deixou mais intuitiva;
- 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/vetores;
- 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;
- `@Any` – ponteiro genérico.

O tipo genérico `@Any` foi adicionado nesta entrega, e equivale ao `void *` de C. O tipo `Any` existe apenas para ser usado com ponteiro, e não pode ser usado em variáveis sozinho.

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 `|||::|` – para funções – ou `|||:||` – para procedimentos – precedido por uma lista de tipos – referentes aos tipos dos parâmetros – e seguido por um único tipo caso seja uma função, indicando o tipo de retorno. Por exemplo, `(Int, Int) |||::| Bool` denota funções que recebem dois inteiros e retorna um 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 próprio 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`. Nesta entrega foi também adicionado um construtor de tipo especial,

o `Null`, que retorna sempre um ponteiro nulo de tipo `@Any`. Esse é o único construtor de tipo que retorna um ponteiro.

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, condicional, casamento de tipos, repetição (*while* e *repeat*) e liberação de memória. As alterações na sintaxe dos comandos em relação à primeira entrega são mencionadas onde for apropriado. Nota-se também que todos os comandos agora precisam ser separados com ponto-e-vírgula, e que toda expressão pode ser considerada um comando, tornando desnecessário distinguir aqui a chamada de procedimentos e de funções – o que faz sentido, visto que sua sintaxe é idêntica e ter produções diferentes para cada geraria conflitos, como discutido na Seção 4.

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

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

Algoritmo 2.2 – Comando de exportação de módulo.

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.

```
'' 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.

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 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`).

```
1  :::::: String = @Char;
2
3  ::: Person = Person(name: @Char, age: Int);
4
5  yin pi: Real = 3.1415926535;
6
7  yang initial(p: Person): Char = p.name[0];
8
9  '' ::::|| lista parâmetros de tipo para
10 '' a função/procedimento/operador que vier em seguida
11 ::::|| T
12 yang len(list: @List(T)): Int = { ''' expresão '''  }
13
14 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 `|||||:` (equivalente ao clássico `return`), seguido opcionalmente de uma expressão, para encerrar a execução imediatamente e retornar o valor dessa expressão. Fizemos uma correção aqui, pois anteriormente a expressão era obrigatória.

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. Anteriormente, eram usados hexagramas especiais para definir associatividade e precedência dos operadores. No entanto, isso geraria um desafio para a implementação do analisador léxico e sintático. Para simplificar o projeto da linguagem, definimos um padrão léxico para os operadores de forma que os próprios caracteres indicam sua precedência e associatividade, como discutido em detalhes na Seção 2.7.

Veja no Algoritmo 2.6 a implementação do comando condicional. Nele, o trigrama `:|:`

```
'' operador associativo à esquerda, precedência 5
'' prefixo <
yang <#(x: Real, exp: Real): Real = { ''' expressão '''  }

'' operador associativo à esquerda, precedência 1
'' sufixo >>
yang +>>(item: Int, head: @List): @List = { ''' expressão ''' }

'' operador não associativo, precedência 1
'' não possui padrão
yang =!=(s0: @Char, s1: @Char): Bool = { ''' expressão ''' }
```

Algoritmo 2.5 – Comando de definição de operadores.

```
:|: 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 2.6 – 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 `:|::|:`.

No Algoritmo 2.7 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.

No Algoritmo 2.8, temos os comandos responsáveis pela repetição. O trigrama `:||`

```
::: 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 2.7 – Comando de casamento de tipo.

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

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

Algoritmo 2.8 – Comando de repetição com teste antes do laço.

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. A condição é opcionalmente seguida do hexagrama `:|||::`, que indica um bloco de código a ser executado após cada iteração do laço. Por fim, deve ser seguido do hexagrama `:|||||` que define o trecho de código a ser 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 2.9 – Comando de repetição com teste depois do laço.

No Algoritmo 2.9, 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`).

```
1 yin list: @List = '' expressão
2 |:: list;
```

Algoritmo 2.10 – Comando de liberação de memória.

No Algoritmo 2.10, observe que utilizamos o trigrama `|::` para a liberação de memória. Fazendo um paralelo com a linguagem C, funcionaria como uma chamada à função `free()`.

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 esses operadores, 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.

Operadores definidos pelo usuário têm precedência e associatividade atribuída de acordo com o padrão de lexema. Onde aparece `?` na tabela, significa que podem aparecer quaisquer símbolos, dentre os símbolos permitidos, para formar aquele operador. Quaisquer operadores que tenham ambos os prefixos e sufixos simultaneamente seriam ambíguos, e portanto inválidos.

No total, as expressões disponíveis são: condicional, casamento de tipo, atribuição (`=`), acesso a campo (`.comid`), acesso a posição (`[i]`), acesso direto (`@` sufixado), referenciação (`@` prefixado), 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 ou procedimento. Anteriormente, a referenciação era feita com um operador `$` prefixado, porém usamos agora apenas `@` para lidar com ponteiros, sendo o prefixado a referenciação, e sufixado o acesso.

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

Precedência	Operadores	Associatividade
8	`**`	Direita
7	`* / & ^`	Esquerda
6	`+ - |`	Esquerda
5	`<< >>`	Esquerda
5	`<?`	Esquerda
5	`?>`	Direita
4	`== != < <= >= >`	Não associativo
3	`&&`	Esquerda
2	`||`	Esquerda
1	`<<?`	Esquerda
1	`?>>`	Direita
1	`?`	Não associativo

Tabela 1 – Operadores da linguagem.

expressões de negação e de manipulação de ponteiro são os únicos operadores prefixados. Exemplos de cada expressão são mostrados no Algoritmo 2.11.

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. Uma mudança na notação dos tokens para esta entrega é que usamos os mesmos nomes das constantes definidas no arquivo de entrada para o Yacc, com a notação usual de constantes em C, por exemplo, `NOME_DO_TOKEN`.

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

- `INTEGER` – literal de número inteiro;
- `REAL` – literal de número real;
- `CHAR` – literal de caractere;
- `STRING` – literal de string;
- `ENDL` – separador de comando;
- `TRIGN` – trigrama de valor `N`, ex.: `(TRIG6)` é `||:`;
- `HEXNN` – hexagrama de valor `NN`, ex.: `(HEX42)` é `|:|:|:`;
- `(COM_ID)` – identificador comum;
- `(PRO_ID)` – identificador próprio;
- `(SYM_ID_XY)` – identificador simbólico, de associatividade `X` e precedência `Y`, sendo que `X` pode ser `L` (esquerda), `R` (direita) ou `N` (não associativo) e `Y` vai de 1 a 8;

```
'' 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
yin p: @Person = ::| Person;
'' alocação de 10 posições
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 2.11 – Expressões.

- `(QCOM_ID)` – identificador comum qualificado;
- `(QPRO_ID)` – identificador próprio qualificado;
- `(QSYM_ID_XY)` – 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.

No Algoritmo 2.12 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.

```
<program> ::= <module-decl> <top-stmts>

<module-decl> ::= TRIG7 <pro-id> HEX56 <exports> ENDL | ""
<exports> ::= <exports> "," <export-id> | <export-id>
<export-id> ::= COM_ID | PRO_ID | <sym-id->

<top-stmts> ::= <top-stmts> ENDL <top-stmt> | <top-stmt>
<top-stmt> ::= <import>
             | <type-def>
             | <type-alias>
             | <callable-def>
             | <call-type-params> <callable-def>
             | <var-def>
             | ""
```

Algoritmo 2.12 – Início da GLC.

Em seguida, a variável <stmts> (Alg. 2.13) é a que permite quaisquer comandos, e aparecerá no corpo de outras produções mais a frente. Vale ressaltar que as produções com <expr> foram deixadas intencionalmente ambíguas, pois isso simplifica a implementação da gramática no Yacc. Veja mais informações sobre isso na seção 4.

```
<stmts> ::= <stmts> ENDL <stmt> | <stmt>
<stmt> ::= <top-stmt>
         | <while>
         | <repeat>
         | <free>
         | <break>
         | <continue>
         | <return>
         | <expr>
```

Algoritmo 2.13 – Comandos expressos na GLC.

O `stmt`, anteriormente, tinha todas as definições que `top-stmt` tem, nesse caso, existe uma ambiguidade da gramática. Para sanar esse problema o `stmt` pode derivar em `top-stmt`. Nesse caso, a definição de variável, por exemplo, pode ser declarada dentro de um escopo aninhado.

O Algoritmo 2.14 mostra as variáveis que expressam as construções de definição e declaração de tipos, variáveis, funções, etc., além do comando de importação. Nessa excerto acima, renomeamos a definição regular `constructor` para `constr` e `constructors` para `constrs`, pois a ferramenta utilizada na análise sintática possui `constructor` como um apalavra reservada.

No Algoritmo 2.15 temos alguns comandos e expressões que podem ser usadas como comandos. Comandos auxiliares de controle de fluxo também aparecem aqui, com os hexagramas

```
<import> ::= TRIG6 <pro-id>
           | TRIG6 <pro-id> HEX51 <exports>
           | TRIG6 <pro-id> HEX54 PRO_ID

<type-def> ::= TRIG0 PRO_ID <type-param-list> "=" <constrs>
<constrs> ::= <constrs> "," <constr> | <constr>
<constr> ::= PRO_ID "(" <param-list> ")" | PRO_ID

<type-alias> ::= HEX00 PRO_ID <type-param-list> "=" <type-id>

<type-param-list> ::= "(" <type-params> ")" | ""
<type-params> ::= <type-params> "," PRO_ID | PRO_ID

<call-type-params> ::= HEX03 <type-params>

<callable-def> ::= <func-def> | <op-def> | <proc-def>

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

<op-def> ::= "yang" <sym-id-> "(" <param> "," <param> ")" ":" <type-id>
↪  "=" <expr>

<proc-def> ::= "wuji" COM_ID "(" <param-list> ")" <stmt>

<param-list> ::= <params> | ""
<params> ::= <params> "," <param> | <param>
<param> ::= COM_ID ":" <type-id>

<var-def> ::= "yin" COM_ID ":" <type-id>
            | "yin" COM_ID ":" <type-id> "=" <expr>
```

Algoritmo 2.14 – Comandos de definições e declarações na GLC.

que correspondem ao `break`, `continue` e `return`.

Além disso, houve a mudança da produção do `else` com intuito de essa produção aceitar um comando também, ou seja, após o `else` é possível definir um `stmt`. Essa mesma alteração foi feita para o caso da produção do `while`.

As expressões na GLC requerem uma atenção maior (Alg. 2.16), 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. Com a simplificação dos lexemas para operadores, é possível especificar diretamente cada um deles na gramática. Além disso, o operador de subtração (`-`) precisou ser tratado a parte, pois pode tanto aparecer como prefixado como infixado.

```
<if> ::= TRIG2 <expr> HEX20 <stmt> <elif> <else>
<elif> ::= <elif> HEX18 <expr> HEX20 <stmt> | ""
<else> ::= HEX19 <stmt> | ""

<match> ::= TRIG5 <expr> <cases> <default>
<cases> ::= <cases> <case> | <case>
<case> ::= HEX47 <literal> HEX42 <stmt>
         | HEX47 <decons> HEX42 <stmt>
<default> ::= HEX44 <stmt> | ""
<decons> ::= <pro-id> "(" <com-id-list> ")"
<com-id-list> ::= <com-ids> | ""
<com-ids> ::= <com-ids> "," COM_ID | COM_ID

<while> ::= TRIG3 <expr> <step> HEX31 <block>
<repeat> ::= HEX27 <block> HEX25 <expr> <step>
<step> ::= HEX28 <block> | ""

<break> ::= HEX30
<continue> ::= HEX26
<return> ::= HEX62 <expr> | HEX62

<free> ::= TRIG4 <addr>
```

Algoritmo 2.15 – Comandos condicional, repetições e liberação de memória na GLC.

A produção que define o `malloc` também sofreu alterações. Da maneira anterior existia conflitos com o lexema utilizado, há ambiguidade entre o tipo e um construtor de tipo, logo foi necessário adicionar um token para cada definição, um para o tipo (`::|`) e outro para um construtor de um tipo (`::||:|`).

Ao final da gramática (Alg. 2.18), 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.

3 ANÁLISE LÉXICA

Nesta seção, é mostrado a implementação e desenvolvimento do analisador léxico da linguagem em detalhes. Como dito na introdução, a ferramenta utilizada foi o Lex (4). O arquivo-fonte para o Lex está anexado nos arquivos enviados, em `src/lex.l`. Alguns trechos relevantes do analisador são destacados aqui.

Primeiramente, na seção de definições, temos o `YY_USER_ACTION` (Alg. 3.1), que é um recurso do Lex para especificar um trecho de código a ser executado após cada casamento de lexema. Aqui, ele é responsável por fazer a contagem de linhas e colunas, com o objetivo de acompanhar a posição da leitura do arquivo nas mensagens de erro.

```
<expr> ::= "{" <stmts> "}"
         | <assign>
         | <match>
         | <if>
         | <expr> SYM_ID_L1 <expr> | <expr> QSYM_ID_L1 <expr>
         | <expr> SYM_ID_N1 <expr> | <expr> QSYM_ID_N1 <expr>
         | <expr> SYM_ID_R1 <expr> | <expr> QSYM_ID_R1 <expr>
         | <expr> SYM_ID_L2 <expr> | <expr> QSYM_ID_L2 <expr>
         | <expr> SYM_ID_L3 <expr> | <expr> QSYM_ID_L3 <expr>
         | <expr> SYM_ID_N4 <expr> | <expr> QSYM_ID_N4 <expr>
         | <expr> SYM_ID_R5 <expr> | <expr> QSYM_ID_R5 <expr>
         | <expr> SYM_ID_L5 <expr> | <expr> QSYM_ID_L5 <expr>
         | <expr> SYM_ID_L6 <expr> | <expr> QSYM_ID_L6 <expr>
         | <expr> "-" <expr>
         | <expr> SYM_ID_L7 <expr> | <expr> QSYM_ID_L7 <expr>
         | <expr> SYM_ID_R8 <expr> | <expr> QSYM_ID_R8 <expr>
         | "@" <expr> | "~" <expr> | "!" <expr> | "-" <expr>
         | <malloc>
         | <build>
         | <call>
         | <addr>
         | <literal>
         | "(" <expr> ")"
```

Algoritmo 2.16 – Expressões na GLC.

```
<assign> ::= <addr> "=" <expr>
<addr> ::= <addr> "[" <expr> "]"
         | <addr> "." COM_ID
         | <addr> "@"
         | COM_ID

<malloc> ::= TRIG1 <type-id> <malloc-n>
           | HEX13 <expr> <malloc-n>
<malloc-n> ::= HEX11 (LiteralInt) | ""

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

Algoritmo 2.17 – Expressões na GLC.

```
<type-id> ::= <type-ptr> <pro-id> <type-arg-list>
            | <type-ptr> <type-arg-list> HEX57 <type-id>
            | <type-ptr> <type-arg-list> HEX59

<type-ptr> ::= <type-ptr> "@"
             | <type-ptr> "[" INTEGER "]"
             | ""

<type-arg-list> ::= "(" <type-args> ")"
<type-args> ::= <type-args> "," <type-id>
              | <type-id>

<pro-id> ::= <pro-id->
<pro-id-> ::= PRO_ID | QPRO_ID
<com-id> ::= <com-id->
<com-id-> ::= COM_ID | QCOM_ID

<sym-id-> ::= SYM_ID_R8 | SYM_ID_L7 | SYM_ID_L6 | SYM_ID_L5
    | SYM_ID_R5 | SYM_ID_N4 | SYM_ID_L3 | SYM_ID_L2
    | SYM_ID_L1 | SYM_ID_R1 | SYM_ID_N1 | "-"

<literal> ::= CHAR | STRING | INTEGER | REAL
```

Algoritmo 2.18 – Variáveis auxiliares da GLC.

```
#define YY_USER_ACTION \
    yylloc.first_line = yylloc.last_line; \
    yylloc.first_column = yylloc.last_column; \
    if (!started) started = 1, printf("\%3d: ", yylloc.last_line); \
    for (int i = 0; yytext[i]; i++) { \
      putchar(yytext[i]); \
      if (yytext[i] == '\n') \
        yylloc.last_line++, yylloc.last_column = 1, printf("\%3d: ",
        ↪  yylloc.last_line); \
      else \
        yylloc.last_column++; \
    }
```

Algoritmo 3.1 – Definição `YY_USER_ACTION`.

Nesta entrega, na qual temos que integrar o analisador léxico ao analisador sintático, ao invés de imprimir os tokens na saída padrão, seu valor deve ser retornado. Os tokens agora são as constantes definidas no arquivo do Yacc (mais detalhes na Seção 4), de valor inteiro, e retornadas nas ações, como exemplificado no Algoritmo 3.2.

```
1 wuji                          {return WUJI;}
2 yin                           {return YIN;}
3 yang                          {return YANG;}
4 {trigram}                     {return TRIG0 + tri_hex_val();}
5 {hexagram}                    {return HEX00 + tri_hex_val();}
```

Algoritmo 3.2 – Exemplos de tokens simples retornados.

Uma diferença notável no analisador léxico é no tratamento de literais de strings, que são agora lidas usando a diretiva de estados do Lex (Alg. 3.3). Ao encontrar aspas, o analisador entre no estado de leitura de strings e processa caractere por caractere, facilitando o tratamento de sequências de escape, e permitindo mensagens de erro léxico específicas para strings. Neste trecho também é possível ver como os valores de atributo de token são passados para o Yacc (linha 2), usando a variável global `yylval`.

```
1 \"                            {BEGIN(STRLIT); s = buffer;}
2 <STRLIT>\"                    {BEGIN(INITIAL); *s = 0; yylval.string_val =
  ↪  string_val(); return STRING;}
3 <STRLIT>\\{charesc}           {*s++ = unescape(yytext[1]);}
4 <STRLIT>\\x{hexit}{hexit}     {*s++ = (char)((char_int_val(yytext[2]) << 4)
  ↪  + char_int_val(yytext[3]));}
5 <STRLIT>{gap}                 {/* ignore */}
6 <STRLIT>{ws_space}            {lexerror("invalid string literal");}
7 <STRLIT>.                     {*s++ = *yytext;}
8 <STRLIT><<EOF>>               {lexerror("invalid string literal");}
```

Algoritmo 3.3 – Leitura de literal de string.

Por fim, outra mudança importante é quanto aos operadores. Além dos operadores predefinidos, lidos separadamente, temos os operadores disponíveis para o próprio usuário definir, cujos padrões de lexema foram descritos previamente, e sua leitura é apresentada no Algoritmo 3.4. Vale ressaltar que existe um padrão de maior precedência para impedir os operadores ambíguos (linha 1).

4 ANÁLISE SINTÁTICA

Tendo a gramática sido devidamente revisada, sua especificação foi passada para o arquivo `src/parse.y`, usado como entrada para o Yacc, o qual não será mostrado na íntegra

```
{qualify}?\<{symbol}*\>     {lexerror("invalid operator");}
{qualify}?\<\<{symbol}+     {return id_token(SYM_ID_L1);}
{qualify}?{symbol}+\>\>     {return id_token(SYM_ID_R1);}
{qualify}?\<{symbol}+       {return id_token(SYM_ID_L5);}
{qualify}?{symbol}+\>       {return id_token(SYM_ID_R5);}
{qualify}?{symid}           {return id_token(SYM_ID_N1);}
```

Algoritmo 3.4 – Leitura dos operadores definidos pelo usuário.

neste relatório, devido ao fato de que está anexado à entrega. São destacados aqui apenas trechos de maior relevância.

Como a gramática já foi especificada na BNF, foi necessário apenas adequar a sintaxe para o formato do Yacc: as variáveis que antes eram escritas como `<nome-variavel>` passaram a ser escritas como `nome_variave`, trocando os hifens por traços baixos, e as produções devem encerrar sempre com ponto-e-vírgula.

Durante a implementação do analisador sintático foram feitos alguns ajustes na gramática. Isso se deve principalmente por causa de ambiguidades na gramática que não eram previstas.

Durante o desenvolvimento da análise sintática da linguagem proposta, o *yacc* apontou conflitos do tipo *shift/reduce* e do tipo *reduce/reduce*. O primeiro tipo de conflito é resolvido pelo *yacc* automaticamente, realizando o *shift*. Já no segundo caso, há o tratamento automático, reduzindo para a produção que foi definida primeiro na gramática. No entanto, é recomendado, pela documentação da ferramenta, sempre que possível, retirar as ambiguidades da gramática de forma que conflitos *reduce/reduce* não aconteçam.

Para desconsiderar os conflitos de *shift/reduce* e assumir que o *yacc* realizará a análise da forma esperada, é possível definir a precedência dos operadores como foi especificado na Tabela 1.

A precedência de acordo com 4.1 aumenta de baixo para cima, ou seja, o '=' possui a menor precedência e o *SYM_ID_R8* e *QSYM_ID_R8* possui a maior precedência[1]. Com esse recurso foi possível especificar uma gramática ambígua que simplifica as definições das produções. Apesar de a gramática ser ambígua, o *yacc* a partir das precedências consegue distinguir qual operação realizar quando existir conflito. É notável que o '=' tem a menor precedência, pois as expressões precisam ser avaliadas antes de serem atribuídas a um endereço de memória.

Além do conflito desse tipo com os operadores, surgiu o clássico problema do "else pendente", que foi resolvido com a reescrita da gramática para 4.2.

Além de tratar os conflitos *shift/reduce* adicionando a precedência aos operadores,

[1] A diretiva *%nonassoc* significa não associatividade.

```
%right '='
%nonassoc SYM_ID_N1 QSYM_ID_N1
%right SYM_ID_R1 QSYM_ID_R1
%left SYM_ID_L1 QSYM_ID_L1
%left SYM_ID_L2 QSYM_ID_L2
%left SYM_ID_L3 QSYM_ID_L3
%nonassoc SYM_ID_N4 QSYM_ID_N4
%right SYM_ID_R5 QSYM_ID_R5
%left  SYM_ID_L5 QSYM_ID_L5
%left  SYM_ID_L6 QSYM_ID_L6
%left  SYM_ID_L7 QSYM_ID_L7
%right SYM_ID_R8 QSYM_ID_R8
```

Algoritmo 4.1 – Precedência no *yacc*

```
if: TRIG2 expr HEX20 stmt elif else
elif: elif HEX18 expr HEX20 stmt | ;
else: HEX19 stmt | ;
```

Algoritmo 4.2

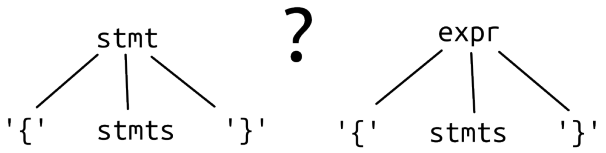


Figura 3 – Exemplo de conflito *reduce/reduce*.

tivemos que tratar os *reduce/reduce*. Porém, nesse caso, é preciso alterar a escrita da gramática. O mais frequente desse último tipo era porque tanto `stmt` quanto `expr` poderiam derivar em uma sequência de comandos envoltos por chaves, e `stmt` poderia derivar em `expr` (Fig. 3). Assim, o analisador ficaria em dúvida de qual decisão tomar: reduzir uma forma sentencial `'{' stmts '}'` para `stmt` ou para `expr`?

Por se tratar de uma funcionalidade intencional do projeto da linguagem – possibilitar que em qualquer lugar que se espera uma expressão, essa possa ser quebrada em vários comandos –, resolvemos esse conflito deixando apenas `expr` podendo ser aberto em chaves, visto que todo comando pode ser derivado numa expressão. Com isso, abrimos mão da própria gramática ser capaz de especificar que toda expressão aberta em chaves tem que encerrar com uma expressão, deixando essa verificação para a análise semântica.

4.1 TABELA DE SÍMBOLOS

Para produzir a outra saída solicitada para o analisador sintático, de exibir a tabela de símbolos, foram implementadas ações semânticas das produções da gramática, junto com algumas estruturas de dados. Definimos uma estrutura de lista encadeada simples nos arquivos

`src/list.h` e `src/list.c`, com operações LIFO de inserção e retirada, e também definimos uma estrutura de tabela hash de tamanho variável nos arquivos `src/hash.h` e `hash.c`, que é utilizada diretamente pela tabela de símbolos. Ambas estruturas armazenam ponteiros genéricos (`void *`), sendo necessário *typecasts* em cada contexto para acessar seus itens.

```
1 typedef struct {
2     HashKey key;
3     ASTNode *node;
4 } SymTableEntry;
5
6 typedef struct symbol_table {
7     struct symbol_table *parent, *child, *sibling;
8     HashTable table;
9 } SymbolTable;
```

Algoritmo 4.3 – Estrutura da tabela de símbolos.

A estrutura da tabela de símbolos (Alg. 4.3) também possui um encadeamento, para tratar dos escopos aninhados dos blocos da linguagem. Normalmente seria suficiente apenas o ponteiro para a tabela anterior (`parent`), mas neste caso queremos exibir a tabela completa, então um encadeado de ida e volta foi inserido: `child` aponta para a primeira tabela aninhada num dado escopo, e `sibling` aponta para a próxima tabela aninhada nesse mesmo escopo.

Como é possível ver na linha 3, cada entrada da tabela de símbolos se refere a um nó da árvore sintática abstrata do programa (`ASTNode`). Decidimos implementar parcialmente a árvore sintática abstrata, criando apenas os nós que introduziriam algum valor na tabela de símbolos, pois cada construção da linguagem possui diferentes formatos do que deve ser registrado na tabela. A estrutura e a manipulação desses nós estão nos arquivos `src/defs.h` e `src/defs.c`. Cada nó específico tem sua própria estrutura, e todas essas estruturas são agrupadas numa `union`, formando o tipo genérico `ASTNode`, e através da enumeração `NodeTag` conseguimos distinguir os subtipos dos nós (Alg. 4.4).

Ao todo, são três tipos de nós que são registrados na tabela de símbolos: nós de definição (`DefNode`), nós de definição de tipo (`TypeDefNode`) e nós de declaração de tipo (`TypeAliasNode`). Esse primeiro tipo de nó é capaz de registrar variáveis, funções, procedimentos e construtores (que nada mais são que funções com uma semântica específica), e sua estrutura é mostrada no Algoritmo 4.5. Todos os `ASTNodes` possuem uma referência da sua localização de aparição no código, representada pelo campo `loc`. Também é importante ressaltar que o tipo da definição é definido por um outro nó, o `TypeNode`, mostrado no Algoritmo 4.6.

As Figuras 4a e 4b mostram exemplos de como o `DefNode` é usado para as construções da linguagem com `yin` e `yang`. É importante ressaltar que, no caso da definição de função, não apenas o nó raiz é registrado na tabela de símbolos, mas também cada um de seus parâmetros (ilustrados por nós retangulares), pois são variáveis no escopo interno da função, que por sua

```
typedef union ast_node ASTNode;

// ...

typedef union ast_node {
    NodeTag tag;
    DefNode def_node;
    IdNode id_node;
    TypeNode type_node;
    PtrTypeNode ptr_type_node;
    ImportNode import_node;
    TypeDefNode type_def_node;
    TypeAliasNode type_alias_node;
    DeconsNode decons_node;
} ASTNode;
```

Algoritmo 4.4 – Estrutura dos nós da árvore sintática abstrata.

```
typedef struct {
    NodeTag tag;
    Loc loc;
    char *id;
    TypeNode *type;
    ASTNode *body;
} DefNode;
```

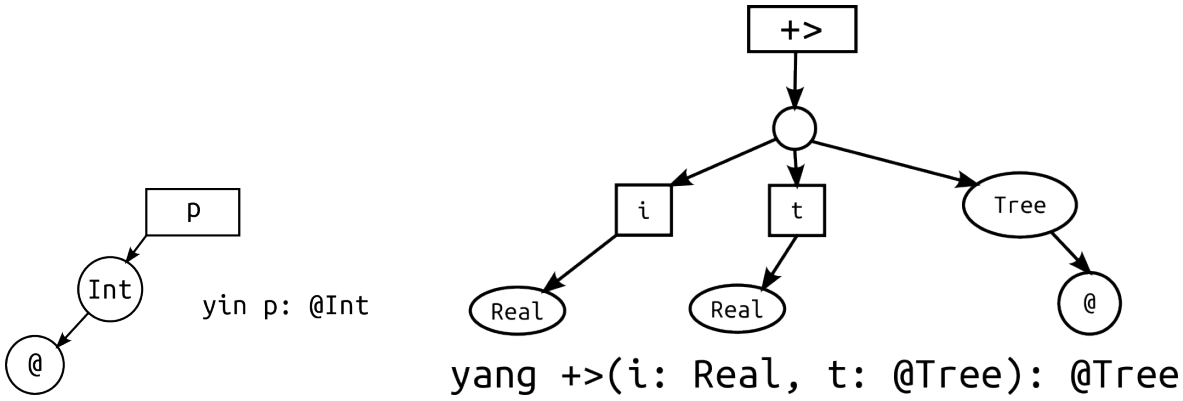
Algoritmo 4.5 – Estrutura do nó de definição.

```
typedef struct ptr_type_node {
    NodeTag tag;
    Loc loc;
    struct ptr_type_node *ptr_t;
    int size;
} PtrTypeNode;

typedef struct {
    NodeTag tag;
    Loc loc;
    PtrTypeNode *ptr_t;
    IdNode *id;
    int arity;
    ASTNode **params;
} TypeNode;
```

Algoritmo 4.6 – Estruturas dos nós de tipo.

vez são `DefNodes`.



(a) `DefNode` para definição de variável.

(b) `DefNode` para definição de função.

Figura 4 – Exemplos do `DefNode`.

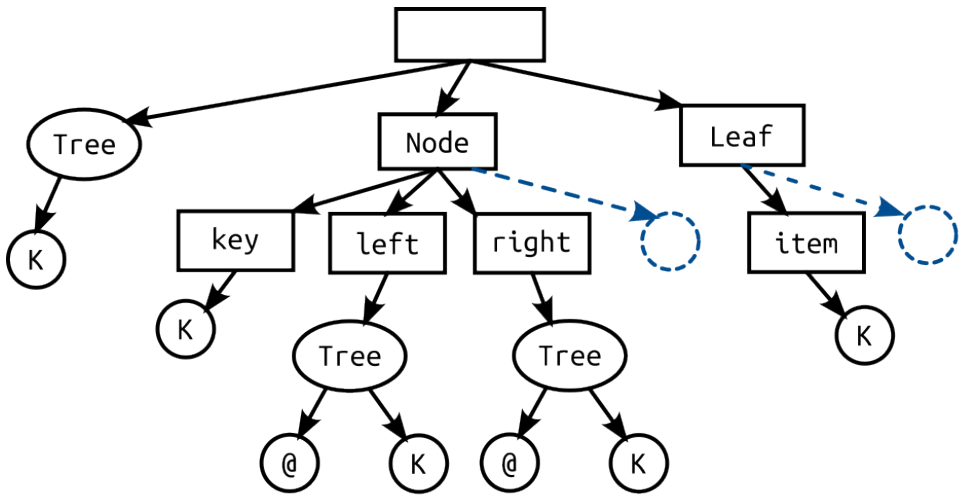
Podemos ver esse processo nas ações semânticas das respectivas produções. No Algoritmo 4.7, apenas a produção `func_def` é mostrada, visto que as produções `op_def` e `proc_def` possuem estrutura muito similar. Como definido na gramática, antes de uma função, é possível que haja parâmetros de tipo. A presença ou ausência desses parâmetros é marcada pela variável global `with_type_params`, alterada nas ações semânticas da produção `call_type_params`, que cria uma novo escopo. Se esse escopo já tiver sido criado, o nome da função que está sendo criada deve ficar no escopo anterior (linha 5). Caso contrário, quando não houve parâmetros de tipo, então se instala a função no escopo atual (linha 7), e um novo escopo deve ser criado (linha 8). Em seguida, com a ajuda da função auxiliar `install_params`, a lista de parâmetros é percorrida, registrando cada um. Vale ressaltar que o argumento para essa função neste contexto é justamente `$4`, o valor da variável `param_list`. Nota-se também o uso de *mid-rules*, ou seja, ações semânticas no meio da produção, visto que é preciso adicionar o escopo antes do corpo da função, e fechar esse escopo ao final (linha 13).

Uma característica comum às definições de variável, funções e procedimentos, é que todas podem ter um "corpo"(campo `body` na estrutura). No caso das variáveis, seria a expressão que a inicializa, e é opcional. Por enquanto, esse valor será sempre preenchido com nulo, pois ainda não estamos criando a árvore sintática abstrata por completo, e portanto as produções `expr` e `stmt` sempre vão retornar nulo. Porém, o código já está estruturado para refletir esse comportamento esperado se e quando a árvore completa for construída.

O próximo tipo de nó registrado na tabela é o `TypeDefNode`. A Figura 5 mostra como seria um exemplo desse nó para alguma construção da linguagem. Aqui novamente deve ser registrado não apenas o identificador da entidade que estamos definindo (o tipo, neste caso, `Tree`) como também cada um de seus construtores. Como supracitado, os construtores são como funções, logo, cada caixa do construtor da figura também é registrada no mesmo nível da tabela de símbolos. O Algoritmo 4.8 mostra esse processo em detalhes. Antes de definir

```
func_def: YANG COM_ID '(' param_list ')' ':' type_id '=' <node>{
        ASTNode *t = Node_fun_type(loc(@4), NULL, List_push($7, $4));
        $$ = Node_yang(loc(@1), $2, t, NULL);
        if (with_type_params) {
            SymTable_install(SymTableEntry_new($$), env->parent);
        } else {
            SymTable_install(SymTableEntry_new($$), env);
            env = SymTable_new(env);
        }
        with_type_params = 0;
        install_params($4);
    }[def] expr[body] {
        env = env->parent;
        $def->def_node.body = $body;
        $$ = $def;
    } ;
```

Algoritmo 4.7 – Produção `func_def` e suas ações semânticas.




```
::: Tree(K) = Node(key: K, left: @Tree(K), right: @Tree(K)),
              Leaf(item: K);
```

Figura 5 – Exemplo do `TypeDefNode`.

os construtores, o nome do tipo já é registrado na tabela (linha 4), pois os tipos podem ser recursivos, como já mostrado na Figura 5. Depois, cada construtor é registrado (linha 9).

É preciso destacar as funções `Node_adj_constr` (linha 8) e `Node_adj_type_def` (linha 11), que ajustam os valores dos nós construtores e do tipo, respectivamente. O primeiro caso é necessário pois os construtores, definidos nas produções de `constrs`, são como funções que retornam um valor do tipo que está sendo definido. No entanto, a variável `constrs` ainda não tem acesso ao tipo, que deveria ser passado como atributo herdado, e não encontramos como implementar atributos herdados de forma mais direta. Assim, recorremos a essas funções auxiliares e o uso de "*mid-rules*"para adicionar o retorno dos construtores, indicados na figura de azul e pontilhado. O segundo caso, da função `Node_adj_type_def`, serve para ligar os construtores efetivamente à definição do tipo, visto que durante a *mid-rule* a variável `constrs` ainda não tinha aparecido.

```
1  type_def: TRIG0 PRO_ID type_param_list '=' <node>{
2          ASTNode *decl = Node_type_decl(loc(@2), $2, $3);
3          $$ = Node_type_def(loc(@1), decl);
4          SymTable_install(SymTableEntry_new($$), env);
5      }[def] constrs[cs] {
6          for (List *c = $cs; c; c = c->tail) {
7              ASTNode *t = (ASTNode *)$def->type_def_node.decl;
8              ASTNode *p = Node_adj_constr(t, (ASTNode *)c->item);
9              SymTable_install(SymTableEntry_new(p), env);
10         }
11         $$ = Node_adj_type_def($cs, $def);
12     };
```

Algoritmo 4.8 – Produção `type_def` e suas ações semânticas.

Já os nós do tipo `TypeAliasNode` apenas conferem um novo nome a um tipo já existente, e portanto sua definição é muito simples, sendo suficiente observar sua estrutura na Figura 6.

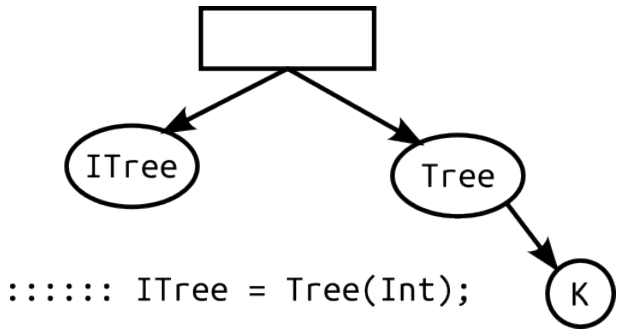


Figura 6 – Exemplo do `TypeAliasNode`.

Existem outros dois pontos importantes no qual a tabela de símbolos é manipulada, que é quando um novo bloco é aberto quando uma expressão se torna um conjunto de comandos, e quando a expressão de casamento de tipos é usada. O primeiro caso é muito simples, bastante

ações semânticas simples nas produções de `expr` (Alg 4.9). Após abrir chaves, um novo escopo é aberto na tabela de símbolos (linha 1) e após fechar chave, esse escopo é encerrado (linha 2). Já o segundo caso, da expressão de casamento de tipos, requer mais atenção.

```
1 expr: '{' { env = SymTable_new(env); } stmts '}' {
2             env = env->parent;
3             $$ = NULL;
4         }
5     | // outras produções...
```

Algoritmo 4.9 – Produção `expr` e abertura de bloco.

Essa expressão é capaz de "desconstruir"os construtores de tipo e acessar seus campos diretamente, executando apenas o trecho de código referente ao construtor correto. Por exemplo, na expressão `|:| tree |:|||| Node(k, l, r) |:|:|: stmt`, o comando só é executado se o resultado da expressão `tree` for um `Node`. E dentro do escopo desse comando, os identificadores `k`, `l`, e `r` ficam disponíveis como variáveis, se referindo aos campos do `Node`. Registrar esses identificadores como variáveis é fácil, e isso já está sendo feito, como mostrado no Algoritmo 4.10, nas linhas 5, 8 e 9. O desafio, porém, é detectar qual tipo deve ser associado a essas variáveis. Deve-se consultar na tabela de símbolos o construtor que está sendo "desconstruído"para obter os tipos de cada um de seus campos (linha 13). O acesso a cada parâmetro do tipo do construtor está abstraído dentro da função `Node_decons`, que cria o respectivo nó. Como ainda não estamos na análise semântica, não verificamos se a quantidade de campos no "desconstrutor"da expressão casa com a aridade do construtor.

Há também o caso das importações de outros módulos como construção que adiciona símbolos na tabela, porém, decidimos omitir esse tipo de registro, pois ainda não temos conhecimento sobre como compilar mais de um arquivo. Assim, a gramática permite essa construção na linguagem, porém nenhum efeito dela se torna visível.

Por fim, é preciso destacar que ao criar o escopo raiz da tabela de símbolos, instalamos manualmente os tipos de dados, operadores e construtores predefinidos (Alg. 4.11). Como nesta entrega nenhuma análise será feita sobre eles, instalamos apenas alguns desses símbolos, para ilustrar que isso é possível de ser feito, e deixamos para a análise semântica realizar essa tarefa com mais cuidado e de forma mais elegante.

4.2 TRATAMENTO DE ERROS

Na especificação do trabalho, foi solicitado que uma mensagem simples fosse reportada caso haja algum erro no código analisado, e que a compilação seja terminada imediatamente. No entanto, o Yacc possui um recurso de recuperação de erros, que consiste em criar produções na gramática que indiquem por onde o analisador sintático pode continuar a percorrer o código

```
match: TRIG5 expr cases default;
cases: cases case | case ;
case: HEX47 literal HEX42 stmt
    | HEX47 decons {
        env = SymTable_new(env);
        DeconsNode *p = (DeconsNode *)$2;
        for (int i = 0; i < p->argc; ++i)
            SymTable_install(SymTableEntry_new((ASTNode*)p->args[i]),env);
    } HEX42 stmt { env = env->parent; }
    ;
default: HEX44 stmt | ;
decons: pro_id '(' com_id_list ')' {
        SymTableEntry *e = SymTable_lookup($1->id_node.id, env);
        $$ = Node_decons(loc(@1), $1, $3, e->node);
    };
com_id_list: com_ids { $$ = $1; } | { $$ = NULL; } ;
com_ids: com_ids ',' COM_ID {$$=List_push(Node_com_id(loc(@3),$3),$1);}
       | COM_ID { $$ = List_push(Node_com_id(loc(@1), $1), NULL); }
       ;
```

Algoritmo 4.10 – Produção `match` e relacionadas.

```
void init_symbol_table() {
    root = SymTable_new(NULL);
    char *types[] = { "Int", "Long", "Char", "Real", "Bool", "Any" };
    ASTNode *p = NULL;
    for (int i = 0; i < (sizeof(types)/sizeof(char *)); ++i) {
        p = Node_type_def(no_loc(), Node_type_decl(no_loc(), types[i],
        ↪  NULL));
        SymTable_install(SymTableEntry_new(p), root);
    }
    p = Node_constructor(no_loc(), "Null", NULL);
    ASTNode *t = Node_type_decl(no_loc(), "Any", NULL);
    t->type_node.ptr_t = (PtrTypeNode *)Node_ptr_type(no_loc(), NULL, 0);
    p = Node_adj_constr(t, p);
    SymTable_install(SymTableEntry_new(p), root);
}
```

Algoritmo 4.11 – Função `init_symbol_table`, chamada logo antes de `yyparse`.

após algum erro. Usar essa funcionalidade requer cuidado, pois é preciso pensar em pontos estratégicos da gramática para delimitar derivações válidas e inválidas.

Assim, tentamos incluir algumas produções de erro, para indicar, por exemplo, a falta de ponto-e-vírgula, mas isso acabou gerando mensagens de erro espúrias e sem sentido. Assim, mantivemos algumas poucas produções de erro, apenas para demonstrar a funcionalidade, como mostra o Algoritmo 4.12, exemplificando possíveis erros na declaração de exportação de módulo.

```
1 module_decl: TRIG7 pro_id HEX56 exports ENDL
2              | TRIG7 error HEX56 exports ENDL
3                 { yyerror_("expected a proper identifier", @2); }
4              | TRIG7 pro_id error exports ENDL
5                 { yyerror_("expected |||:::", @3); }
6              |
7              ;
```

Algoritmo 4.12 – `module_decl` e produções de erro.

Além da função padrão do Yacc `yyerror`, que sempre exibe uma mensagem de erro genérica, definimos também a função `yyerror_` que permitiria as mensagens definidas por nós. Em ambos os casos, seguindo a temática da linguagem, os erros são reportados como `lexical unbalance` ou `syntax unbalance`, indicando um "desequilíbrio"do *tao* no código de entrada. Deixaremos para a próxima entrega, se possível, aperfeiçoar esse mecanismo de recuperação de erros e proporcionar mensagens de erro significativas em diversos níveis da gramática.

5 RESULTADOS

Nesta seção veremos alguns resultados, os quais apresentarão programas corretos e programas com erros sintáticos. Para compilar o programa, basta executar o comando `make` na raiz da pasta. Um executável em `bin/taoc` será gerado. Para executar, é preciso redirecionar o arquivo para a entrada padrão: `$ bin/taoc < entrada.tao`.

No código mostrado em 5.1, temos a representação de algumas características da linguagem. Na linha 3 é possível ver a definição de uma expressão. Na linha 6 há a definição de um novo tipo e na linha 9 há a definição de um apelido ao tipo definido anteriormente. Na linha 11, define-se os parâmetros de tipo K e T. Na próxima linha, há definição de um procedimento que adiciona um nó à árvore binária definida anteriormente utilizando a sintaxe do `match`.

Na linha 30 é definido um procedimento que calcula um fatorial de um número n, utilizando a sintaxe do `while`. Na linha 40 há a demonstração da função principal do programa escrito em tao, utilizando o lexema wuji e o uso de ponteiros com argv.

No código mostrado em 5.2, é possível ver que a sintaxe utilizada no programa 5.1 está correto. Além disso, é mostrado cada linha numerada. No código mostrado em 5.3, a tabela de símbolos é mostrada e é possível observar cada escopo definido no programa e suas definições.

```
yin test: Int = 0
  + 3;

::: Tree(K,T) = Node(k: K, left: @Tree, right: @Tree),
                Leaf(i: T);

:::::: ITree(K) = Tree(K, Int);

::::|| K, T
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) |:|:|:
        :|: 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)
      }
};

yang fatorial(n: Int): Int = {
    yin x: Int = 1;
    yin i: Int = 1;

    :|| i <= n :|||:: i = i+1 ||||| {
        x = x * i
    };
    |||||: x
};

wuji main(argc: Int, argv: @@Char) {

};
```

Algoritmo 5.1 – Código do arquivo `completo.tao`.

```
  1: ::: Tree(K,T) = Node(k: K, left: @Tree, right: @Tree),
  2:                 Leaf(i: T);
  3:
  4: :::::: ITree(K) = Tree(K, Int);
  5:
  6: ::::|| K, T
  7: yang add(item: T, tree: @Tree(K, T)): @Tree(K, T) = {
  8:     yin itemKey: K = getKey(item);
  9:     yin newLeaf: @Tree(K, T) = ::| Tree(K, T);
 10:     |:| @tree
 11:     |:|||| Node(k, l, r) |:|:|:
 12:         :|: itemKey < k :|:|::
 13:             tree.lt = add(item, l)
 14:         :|::||
 15:             tree.rt = add(item, r)
 16:     |:|||| Leaf(i) |:|:|: {
 17:         yin leafKey: K = getKey(i);
 18:         :|: itemKey < leafKey :|:|::
 19:             ::||:| Node(leafKey, newLeaf, tree)
 20:         :|::||
 21:             ::||:| Node(itemKey, tree, newLeaf)
 22:       }
 23: };
 24:
 25: yang fatorial(n: Int): Int = {
 26:     yin x: Int = 1;
 27:     yin i: Int = 1;
 28:
 29:     :|| i <= n :|||:: i = i+1 :||||| {
 30:         x = x * i
 31:     };
 32:     |||||: x
 33: };
 34:
 35: wuji main(argc: Int, argv: @@Char) {
 36:
 37: };
 38:

Syntax is balanced.
```

Algoritmo 5.2 – Saída 1 do analisador sintático `completo.tao`.

```
+==================================+
| [  0,  0]           Long
| [  0,  0]           Int
| [  0,  0]           Char
| [  0,  0]           Any
| [  0,  0]     Null: () |||::| @Any
| [  0,  0]           Real
| [  0,  0]           Bool
    +==================================+
    | [ 35,  1]     main: (Int, @@Char) |||:||
    | [  1, 17]     Node: (K, @Tree, @Tree) |||::| Tree(K, T)
    | [  7,  1]      add: (T, @Tree(K, T)) |||::| @Tree(K, T)
    | [  2, 17]     Leaf: (T) |||::| Tree(K, T)
    | [  1,  1]           Tree(K, T)
    | [  4,  1]           ITree(K)
    | [ 25,  1] fatorial: (Int) |||::| Int
        +==================================+
        | [  6,  8]           K
        | [  7, 19]     tree: @Tree(K, T)
        | [  6, 11]           T
        | [  7, 10]     item: T
            +==================================+
            | [  9,  5]  newLeaf: @Tree(K, T)
            | [  8,  5]  itemKey: K
                +==================================+
                | [ 11, 20]        l: @Tree
                | [ 11, 17]        k: K
                | [ 11, 23]        r: @Tree
                +==================================+
                | [ 16, 17]        i: T
                    +==================================+
                    | [ 17,  9]  leafKey: K
                    +================//================+
        +==================================+
        | [ 25, 15]        n: Int
            +==================================+
            | [ 26,  5]        x: Int
            | [ 27,  5]        i: Int
            // Saída foi comprimida para respeitar o espaço da página
        +==================================+
        | [ 35, 11]     argc: Int
        | [ 35, 22]     argv: @@Char
```

Algoritmo 5.3 – Tabela de símbolo do analisador sintático `completo.tao`.

Caso retiremos o ponto e vírgula, que separa comandos como definição de variável, obteremos um erro especificando que a sintaxe está incorreta e aponta que é necessário um ponto e vírgula no comando da linha 3, assim como visto em 5.4.

```
1  yang fatorial(n: Int): Int = {
2      yin x: Int = 1
3      yin i: Int = 1;
4
5      :|| i <= n :|||:: i = i+1 :||||| {
6          x = x * i
7      };
8      |||||: x
9  };
10
11 3:5: syntax unbalance: missing `;` in previous statement
```

Algoritmo 5.4 – Código do arquivo `erro.tao`.

```
1  +==================================+
2  | [  0,  0]           Long
3  | [  0,  0]           Int
4  | [  0,  0]           Char
5  | [  0,  0]           Any
6  | [  0,  0]     Null: () |||::| @Any
7  | [  0,  0]           Real
8  | [  0,  0]           Bool
9      +================================+
10     | [  1,  1] fatorial: (Int) |||::| Int
11         +================================+
12         | [  1, 15]        n: Int
13             +================================+
14             | [  2,  5]        x: Int
15             | [  3,  5]        i: Int
16                 +================================+
```

Algoritmo 5.5 – Tabela de símbolos com erro de sintaxe `erro.tao`.

6 CONSIDERAÇÕES FINAIS

Em relação à essa segunda parte, ao realizar a construção do gerador de analisador sintático, foi possível fixar a unidade de análise sintática da disciplina de forma consistente, apesar que sua tarefa teve um nível de dificuldade maior que o analisador léxico. Também é importante salientar que muitos detalhes da parte anterior tiveram que ser revisados para correto uso, após o término desta parte, tendo em vista os desafios da análise sintática.

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, ????