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

CCF441 – COMPILADORES
LINGUAGEM TAO – ENTREGA 3 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

Com este trabalho prático, definimos uma linguagem de programação própria, não só prática, mas também criativa. Na primeira entrega, apresentamos a proposta e implementamos seu analisador léxico, utilizando para isso o Lex e inicialmente a linguagem de programação C. Na segunda entrega, refinamos a gramática da linguagem e implementamos seu analisador sintático utilizando o software Bison/Yacc, integrando com o Lex (1).

Nesta entrega final, o objetivo é realizar a análise semântica e geração de código de código intermediário. Para isso, optamos por mudar para a linguagem C++ e construir a árvore sintática abstrata como representação intermediária, para que pudessem ser emitidas posteriormente as instruções da LLVM (2).

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") (3). 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") (4), 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

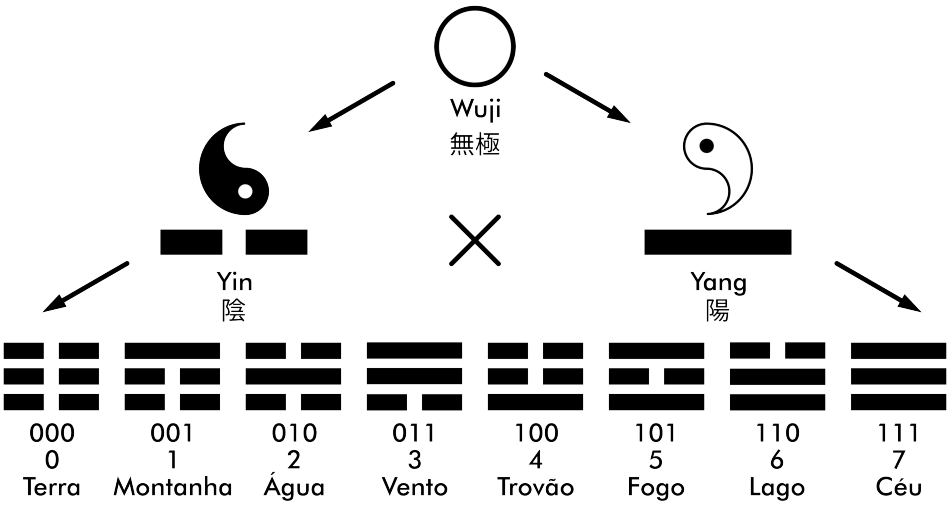


Figura 2 – Os trigramas e sua origem.

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;
- `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`).

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.

```
:::::: 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.

`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.

```
'' 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.

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.

```
:|: 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.

Veja no Algoritmo 2.6 a implementação do comando condicional. Nele, o trigrama `:|:` 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 2.7 – Comando de casamento de tipo.

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.

```
:|| 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.

No Algoritmo 2.8, 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. 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`).

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()`.

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

Algoritmo 2.10 – Comando de liberação de memória.

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.

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

```
'' 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.

2.8 SINTAXE

A gramática da linguagem será expressada na Forma de Backus-Naur (BNF) (5). 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;
- `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.

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.

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.

```
<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.

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

Algoritmo 2.13 – Comandos expressos na GLC.

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 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 sim-

```
<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.

plificação 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. Também optamos por reforçar na própria gramática que o operador de referência (`@` prefixado, equivalente ao `&` prefixado de C) seja seguido de `addr` e não `expr`. Assim garantimos que seu operando será um *l-value*, facilitando a análise semântica.

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 (`::||:|`). Também passamos a permitir uma expressão arbitrária para definir a quantidade de posições de memória a ser alocada.

Ao final da gramática (Alg. 2.18), há algumas variáveis que foram usadas em várias

```
<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.

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 (1). O arquivo-fonte para o Lex está anexado nos arquivos enviados, em `src/lex.l`. Alguns trechos relevantes do analisador são destacados aqui. Devido à mudaça da linguagem, de C para C++, boa parte da implementação foi adaptada para usar classes quando possível, porém a lógica para o analisador léxico se manteve a mesma.

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. Agora não mais fazemos a impressão do arquivo com suas linhas numeradas na saída padrão, então esse comportamento foi retirado, e a implementação foi passada para o método da classe `Lexer`. Essa classe apenas reuniu as funções que antes estavam soltas no escopo global do arquivo `lex.l`.

Ao integrar o analisador léxico ao analisador sintático, ao invés de imprimir os tokens

```
<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>
         | "@" <addr> | "~" <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 <expr> | ""

<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 | QPRO_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 lexer.update_yyloc();
// ...
void Lexer::update_yyloc() {
  yylloc.first_line = yylloc.last_line;
  yylloc.first_column = yylloc.last_column;
  for (int i = 0; yytext[i]; i++)
    if (yytext[i] == '\n')
      yylloc.last_line++, yylloc.last_column = 1;
    else
      yylloc.last_column++;
}
```

Algoritmo 3.1 – Definição `YY_USER_ACTION`.

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 lexer.trig_token();}
5 {hexagram}                    {return lexer.hex_token();}
```

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 entra 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. Aqui também vemos como as regras do analisador léxico ficaram mais legíveis, com os detalhes de implementação passados para métodos.

```
1 \"                            {lexer.str_lit_begin();}
2 <STRLIT>\"                    {return lexer.string_token();}
3 <STRLIT>\\{charesc}           {lexer.str_lit_push_hex();}
4 <STRLIT>\\x{hexit}{hexit}     {lexer.str_lit_push_esc();}
5 <STRLIT>{gap}                 {/* ignore */}
6 <STRLIT>{ws_space}            {lexer.error("invalid string literal");}
7 <STRLIT>.                     {lexer.str_lit_push();}
8 <STRLIT><<EOF>>               {lexer.error("invalid string literal");}
```

Algoritmo 3.3 – Leitura de literal de string.

```
1 {qualify}?\<{symbol}*\>       {lexer.error("invalid operator");}
2 {qualify}?\<\<{symbol}+       {return lexer.id_token(SYM_ID_L1);}
3 {qualify}?{symbol}+\>\>       {return lexer.id_token(SYM_ID_R1);}
4 {qualify}?\<{symbol}+         {return lexer.id_token(SYM_ID_L5);}
5 {qualify}?{symbol}+\>         {return lexer.id_token(SYM_ID_R5);}
6 {qualify}?{symid}             {return lexer.id_token(SYM_ID_N1);}
```

Algoritmo 3.4 – Leitura dos operadores definidos pelo usuário.

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. Há 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, e que não é mostrado na íntegra neste relatório, por ser muito extenso e por já estar anexado à entrega. São destacados aqui apenas trechos de maior relevância. A gramática em si quase não foi alterada, porém nesta entrega, além da adaptação das implementações para C++, as produções foram completadas com ações semânticas para produzir por completo a árvore sintática abstrata.

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_variavel`, trocando os hifens por traços baixos, e as produções devem encerrar sempre com ponto-e-vírgula. Porém, 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.

```
%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.

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

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

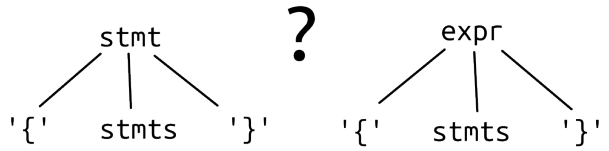


Figura 3 – Exemplo de conflito *reduce/reduce*.

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 como mostra o Algoritmo 4.2. Aqui também são mostradas as ações semânticas adicionadas, discutidas na Seção 5.

```
1  if: TRIG2 expr HEX20 stmt elif else {
2          $5->insert($5->begin(), new IfNode($2,$4));
3          $5->push_back($6);
4          for (auto it = $5->begin(); it != $5->end() - 1; ++it)
5              dynamic_cast<IfNode*>(*it)->set_else(*(it + 1));
6          $$ = *$5->begin();
7          delete $5;
8      }
9  elif: elif HEX18 expr HEX20 stmt { VPUSH($$,$1,new IfNode($3,$5)); }
10     | { $$ = VEMPTY; } ;
11 else: HEX19 stmt { $$ = $2; } | { $$ = NULL; } ;
```

Algoritmo 4.2 – Produções para a expressão condicional.

Além de tratar os conflitos *shift/reduce* adicionando a precedência aos operadores, 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 árvore sintática abstrata, foram implementadas ações semânticas das produções da gramática, junto com algumas estruturas de dados. Usando C++, pudemos utilizar algumas estruturas de dados prontas de sua biblioteca padrão, como o `vector` e o `map`, simplificando a implementação.

```
1  class SymTableEntry {
2  public:
3      Loc loc;
4      ASTNode *node;
5      SymTableEntry(Loc loc, ASTNode *node);
6  };
7
8  class SymTable {
9  public:
10     SymTable *prev;
11     std::map<std::string,std::vector<SymTableEntry>> table;
12     SymTable();
13     SymTable(SymTable *prev);
14     void install(std::string key, const SymTableEntry &entry);
15     std::vector<SymTableEntry> *lookup(std::string key);
16     std::vector<SymTableEntry> *lookup_all(std::string key);
17 };
```

Algoritmo 4.3 – Estrutura da tabela de símbolos.

A estrutura da tabela de símbolos (Alg. 4.3) possui um encadeamento, para tratar dos escopos aninhados dos blocos da linguagem. Agora não sendo mais necessário imprimir a tabela completa, mantemos apenas o encadeamento da tabela atual para a tabela prévia, e assim recursivamente até a tabela raiz.

Como é possível ver na linha 4, cada entrada da tabela de símbolos se refere a um nó da árvore sintática abstrata do programa (`ASTNode`). Além disso, visando implementar o polimorfismo por sobrecarga, cada chave na tabela está associada a uma pilha de entradas (linha 11). Assim, é possível conferir múltiplas implementações de uma mesma função e usar a que for mais adequada aos tipos dos argumentos, por exemplo.

Os nós registrados na tabela são justamente os nós que definem alguma nova entidade de programação (uma nova variável, uma nova função, um novo tipo, etc.). Cada nó específico tem sua própria estrutura, e com o recurso de orientação a objetos, criamos uma hierarquia de tipos para os nós, usando a classe `ASTNode` como base (Alg. 4.4).

A escolha dos nós específicos e de sua estrutura foi revisada, fazendo um mapeamento mais direto com a gramática. Ao todo, registramos na tabela de símbolos os seguintes tipos de nós: `TypeDefNode`, `ConstrNode`, `TypeAliasNode`, `TypeParamNode`, `YinNode`, `YangNode`,

```
1  class ASTNode {
2  public:
3      virtual ~ASTNode() = default;
4      virtual bool declares_type() { return false; };
5      virtual TypeNode *get_type() { return NULL; };
6      virtual std::ostream& show(std::ostream &out);
7      virtual llvm::Value *codegen(CodeGenerator &generator) { return
       ↪  generator.codegen(this); };
8  };
```

Algoritmo 4.4 – Estrutura dos nós da árvore sintática abstrata.

`WujiNode` e `ParamNode`. Na Seção 5 sua estrutura será melhor analisada.

Para registrar corretamente cada um desses nós, é necessário abrir novos escopos na tabela de símbolos em pontos específicos das produções da gramática. Podemos ver um exemplo disso no Algoritmo 4.5, com a produção `func_def` – define novas funções –, sendo 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 não tiver sido criado ainda, isso é feito na linha 5. 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). Usamos os macros `ENVPUSH` (linha 1) e `ENVPOP` (linha 2) para tornar mais legível o código.

```
1  #define ENVPUSH env = new SymTable(env)
2  #define ENVPOP  SymTable *t = env; env = env->prev; delete t
3  // ...
4  func_def: YANG COM_ID {
5              if (!with_type_params) ENVPUSH;
6          } '(' param_list ')' ':' type_id '=' <cnode>{
7          with_type_params = false;
8          $$ = new YangNode(*$2, *$5, $8); delete $5;
9          env->prev->install(*$2, SymTableEntry(Loc(@2), $$));
10         delete $2;
11         procs.push_back($$);
12     }[def] expr[body] {
13         ENVPOP;
14         procs.pop_back();
15         $def->set_body($body);
16         $$ = $def;
17     } ;
```

Algoritmo 4.5 – Produção `func_def` e suas ações semânticas.

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, bastando ações semânticas simples nas produções de `expr` (Alg 4.6). 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: '{' { ENVPUSH; } stmts '}' {
2           ENVPOP; $$ = new BlockNode(*$3); delete $3;
3       }
4       | // outras produções...
```

Algoritmo 4.6 – 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`. Trataríamos essa construção da linguagem, porém, durante a análise semântica, não conseguimos terminar de estabelecer toda a lógica para lidar com os tipos compostos. Assim, optamos por apenas construir a árvore sintática para essa essa expressão, sem fazer manipulações ou verificações na tabela de símbolos.

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. O Algoritmo 4.7 ilustra esse processo, com vários trechos omitidos, por se tratar de uma função muito longa.

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 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,

```
SymTable::SymTable() {
    this->prev = NULL;
    // Tipos básicos
    std::vector<ASTNode*> empty;
    std::vector<std::string> types = { "Char", "Int", "Real", "Any" };
    std::vector<std::string> numeric_types = { "Char", "Int", "Real" };
    std::vector<std::string> integer_types = { "Char", "Int" };
    for (auto it = types.begin(); it != types.end(); ++it)
        this->install(*it, SymTableEntry(Loc(), new TypeDefNode(*it,
        ↪  empty)));
    // ...
    // Operadores
    std::vector<std::string> numeric_ops = { "**", "*", "/", "+", "-" };
    std::vector<std::string> bitwise_ops = { "&", "^", "|", "<<", ">>" };
    std::vector<std::string> logical_ops = { "&&", "||" };
    std::vector<std::string> compare_ops = { "==", "!=", "<", "<=", ">=",
    ↪  ">" };
    for (auto it = numeric_ops.begin(); it != numeric_ops.end(); ++it) {
        for (auto it2 = numeric_types.begin(); it2 != numeric_types.end();
        ↪  ++it2) {
            std::vector<ASTNode*> params = {
                new ParamNode("x", new VarTypeNode(*it2, empty)),
                new ParamNode("y", new VarTypeNode(*it2, empty))
            };
            ASTNode *ret = new VarTypeNode(*it2, empty);
            this->install(*it, SymTableEntry(Loc(), new YangNode(*it,
            ↪  params, ret)));
        }
    }
    // ...
    // Negação numérica
    for (auto &it : numeric_types) {
        std::vector<ASTNode*> params = { new ParamNode("x", new
        ↪  VarTypeNode(it, empty)) };
        ASTNode *ret = new VarTypeNode(it, empty);
        this->install("-", SymTableEntry(Loc(), new YangNode("-", params,
        ↪  ret)));
    }
}
```

Algoritmo 4.7 – Construtor padrão de `SymTable`, chamado logo antes de `yyparse`.

mantivemos algumas poucas produções de erro, apenas para demonstrar a funcionalidade, como mostra o Algoritmo 4.8, exemplificando possíveis erros na declaração de exportação de módulo.

```
module_decl: TRIG7 pro_id HEX56 exports ENDL
               { $$ = new ExportNode(*$2,*$4); delete $2; delete $4; }
           | TRIG7 error HEX56 exports ENDL
               { yyerror_("expected a proper identifier", @2); }
           | TRIG7 pro_id error exports ENDL
               { yyerror_("expected |||:::", @3); }
           | { $$ = NULL; }
           ;
```

Algoritmo 4.8 – `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.

5 ANÁLISE SEMÂNTICA

Na análise semântica, nosso objetivo é realizar as verificações necessárias referentes ao funcionamento de cada expressão ou comando da linguagem. Considerando a dificuldade de trabalhar dentro do arquivo `parse.y`, tanto por questão de legibilidade quanto por falta de ferramentas mais adequadas no editor de texto, buscamos fazer o máximo quanto possível da análise semântica em outras partes do código, em arquivos `.cpp` tradicionais.

A solução encontrada, e que se provou muito prática, foi fazer as verificações referentes a cada construção da linguagem exatamente no momento em que seus nós da árvore sintática abstrata são criados ou atualizados. Assim, quase toda a lógica está contida nos construtores dos nós da árvore. Dentre as expressões/comandos que requerem algum tipo de análise, identificamos como necessárias as verificações nos seguintes nós:

- `TypeNode` – implementar verificação de compatibilidade de tipos, e se um tipo não existente é referenciado;
- `IDNode` – verificar se a variável referida foi declarada;
- `YinNode` – verificar se o tipo de expressão é o mesmo que o da variável;
- `YangNode` – verificar se o corpo da expressão retorna o tipo esperado;
- `TypeDefNode` – verificar se há definição recursiva de tipo (não implementada);
- `IfNode`, `WhileNode`, `RepeatNode` – verificar se condição é booleana;

- `BreakNode`, `ContinueNode` – verificar se está dentro de um laço de repetição;
- `ReturnNode` – verificar se está dentro de uma função, e se retorna o tipo esperado, ou, se estiver num procedimento, se o retorno é vazio;
- `FreeNode` – verificar se o tipo de retorno da expressão é ponteiro;
- `UnaryOpNode`, `BinaryOpNode`, `CallNode` – verificar se existe declaração da função/operador, e se os tipos dos argumentos são compatíveis;
- `AssignNode` – verificar se o tipo atribuído é compatível ao da variável de destino;
- `AddressNode` – verificar se a expressão acessada é do tipo ponteiro, e verificar se o tipo da expressão do índice, se existir, é inteiro;
- `MatchNode`, `CaseNode` – verificar compatibildidade de tipos entre a expressão casada e as condições (não implementada);
- `DeconsNode` – verificar se existe o construtor de tipo referido (não implementado);
- `MallocNode` – verificar se a expressão que indica quantas posições de memória devem ser alocadas retorna inteiro.

Como é possível ver na lista, deixamos de implementar as verificações referentes aos tipos compostos de dados. Isso se deve ao fato de que não chegamos a uma conclusão sobre como lidar com os campos dos tipos com mais de um construtor, se faria sentido sobrecarga de construtores, entre outras questões. Apesar disso, o `TypeDefNode`, por exemplo, ainda assim registra valores na tabela de símbolos.

Para tornar tudo isso possível, foi necessário armazenar em cada nó da árvore qual o tipo que o nó produz, e implementar um método nas classes `TypeNode`, que descrevem os tipos, para comparar tipos e verificar se são compatíveis. Assim, todo nó possui um método `get_type` e todo `TypeNode` possui o método `check`, que retorna 0 para incompatível, 1 para compatível e 2 para idêntico – essa distinção se torna útil na escolha de funções sobrecarregadas. A noção de compatibilidade de tipo está relacionada com a coerção: `Char` pode ser transformado para `Int`, `Int` para `Real`, e tanto `Char` como `Int` podem ser `Bool`. Não chegamos a implementar o mecanismo de *type alias*, portanto as comparações de tipo são feitas nominalmente, comparando diretamente as strings dos identificadores de tipo.

Um exemplo é o `IDNode` (Alg. 5.1), que é criado quando uma variável é referenciada. Ao ser criado, seu tipo deve ser consultado na tabela de símbolos e armazenado (linha 4). Caso não haja uma entrada na tabela de símbolos, podemos emitir um erro de que a variável não foi declarada (linha 6). Note que se houver mais de uma declaração da variável naquele escopo, usa-se a referência mais recente.

```
1 IDNode::IDNode(std::string id, bool chk) {
2     this->id = id;
3     if (!chk || !env) return;
4     std::vector<SymTableEntry> *es = env->lookup(id);
5     if (!es)
6         { serr() << "`" << id << "` not in scope" << std::endl; exit(1); }
7     this->expr_type = es->back().node->get_type();
8 }
```

Algoritmo 5.1 – Análise semântica no `IDNode`.

De forma semelhante temos as chamadas a funções, no `CallNode`, com seu extenso construtor, mostrado no Algoritmo 5.2. Das linhas 4 a 6 temos a consulta à tabela de símbolos e um erro caso não haja uma entrada correspondente. Das linhas 9 a 17, coletamos os tipos de cada uma das expressões usadas como argumento, emitindo um erro (linha 13) caso alguma expressão não retorne um valor.

Em seguida, percorremos as entradas na tabela de símbolos na ordem reverse, considerando que a definição mais recente se encontra no final o vetor. Observe que neste caso, usamos o método `lookup_all`, que implementamos de forma a retornar as entradas da tabela que correspondem ao identificador em todos os seus níveis.

Na linha 27 é feita a verificação de compatibilidade de tipos. As linhas 31 e 32 fazem a escolha de priorizar os tipos que sejam idênticos, mesmo que haja funções compatíveis em algum escopo mais próximo. Se os tipos dos argumentos forem idênticos aos dos parâmetros (indicado pelo valor de enumeração `TypeChk::EQ`), o laço encerra imediatamente. No entanto, se nenhuma definição de função for compatível, um erro é emitido na linha 35.

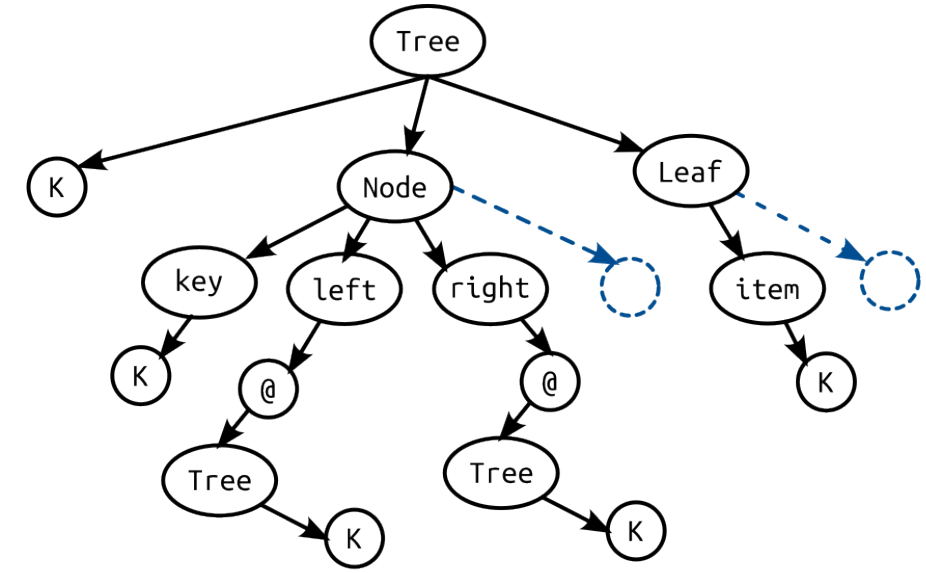
Há casos que nem toda verificação semântica pode ser feita assim que o nó é construído. Isso ocorre tanto no `IfNode` quanto do `TypeDefNode`. O `IfNode` é construído parcialmente, visto que suas produções incluem uma quantidade arbitrária de "else ifs". Como visto no Algoritmo 4.2, cada `elif` insere numa lista um novo `IfNode`. Esses nós são posteriormente encadeados uns aos outros, colocando-os como o corpo "else"do anterior e retornando o `IfNode` inicial (linhas 4 a 6).

No caso do `TypeDefNode`, vemos na Figura 4 um exemplo desse nó para alguma construção da linguagem. Assim como as funções registram a si mesmas e também seus parâmetros na tabela de símbolos, aqui deve ser registrado não apenas o identificador do tipo que estamos definindo (neste caso, `Tree`) como também cada um de seus construtores. Os construtores são como funções, com um tipo de retorno automaticamente definido para o tipo que está sendo criado. O Algoritmo 5.3 mostra esse processo em detalhes. Antes de definir os construtores, o nome do tipo já é registrado na tabela (linha 4), pois os tipos podem ser recursivos se usados como ponteiros.

```
CallNode::CallNode(std::string id, std::vector<ASTNode*> &args) {
    this->id = id;
    VCOPY(ExprNode,args);
    std::vector<SymTableEntry> *es = env->lookup_all(id);
    if (!es)
        { serr() << "function " << id << " not in scope" << std::endl;
        ↪  exit(1); }
    std::vector<TypeNode*> targs;
    int i = 0;
    for (auto &it : this->args) {
        ++i;
        TypeNode *targ = it->get_type();
        if (!targ) {
            serr() << "no value for `" << id << "` argument " << i <<
            ↪  std::endl;
            exit(1);
        }
        targs.push_back(targ);
    }
    TypeNode *ret = NULL;
    for (auto it = es->rbegin(); it != es->rend(); ++it) {
        CallableNode *cnode = dynamic_cast<CallableNode*>(it.node);
        if (!cnode) continue;
        if (cnode->params.size() != args.size()) continue;
        TypeChk chk = TypeChk::EQ;
        for (size_t i = 0; i < args.size(); ++i) {
            TypeNode *tp = cnode->params[i]->get_type();
            TypeNode *ta = targs[i];
            TypeChk c = tp->check(ta);
            if (!c) { chk = c; break; }
            if (c == TypeChk::CMP) chk = c;
        }
        if (chk == TypeChk::EQ) { ret = cnode->ret; break; }
        if (chk && !ret) ret = cnode->ret;
    }
    if (!ret) {
        serr() << "no compatible definition of `" << id << "` for argument
        ↪  types "
            << types_str(targs) << std::endl;
        exit(1);
    }
    this->expr_type = ret;
    delete es;
}
```

Algoritmo 5.2 – Análise semântica no `CallNode`.

Cada construtor, em suas produções, registra a si mesmo na tabela, porém dentro da sua produção, ele não sabe qual é o seu tipo de retorno. Por esse motivo, na linha 6, é chamado o método `add_constrs` no `TypeDefNode`, para que este último conecte os `ConstrNodes` a si, e também complete o retorno dos construtores (ilustrados pelos círculos pontilhados na Figura 4).




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

Figura 4 – Exemplo do `TypeDefNode`.

```
1  type_def: TRIG0 PRO_ID { ENVPUSH; } type_param_list '=' <node>{
2          $$ = new TypeDefNode(*$2, *$4); delete $4;
3          env->prev->install(*$2, SymTableEntry(Loc(@2), $$)); delete $2;
4      }[def] constrs[cs] {
5          ENVPOP;
6          dynamic_cast<TypeDefNode*>($def)->add_constrs(*$cs);
7          delete $cs;
8          $$ = $def;
9      };
```

Algoritmo 5.3 – Produção `type_def` e suas ações semânticas.

Outro desafio interessante foi o dos comandos `:||||:` (*break*), `:||:|:` (*continue*) e `|||||:` (*return*). Precisávamos conferir se estavam dentro de laços de repetição ou no corpo de funções. Para isso, foi usado um mecanismo de memória adicional à tabela de símbolos, que apareceu nas linhas 11 e 14 do Algoritmo 4.5. Criamos duas pilhas, `loops` e `procs`, inserindo `WhileNodes` e `RepeatNodes` no primeiro caso, ou `YangNodes` e `WujiNodes` no segundo caso. Assim, quando um `BreakNode` ou `ContinueNode` era construído, bastava consultar a pilha `loops` e registrar que o último laço encontrado é o laço ao qual esse comando se refere. Caso não haja nenhum registro, um erro pode ser emitido, informando que deve estar dentro de um laço.

```
BreakNode::BreakNode() {
    if (loops.empty())
        { serr() << ":||||: must be inside a loop" << std::endl; exit(1); }
    loop = loops.back();
}
```

Algoritmo 5.4 – Análise semântica no *BreakNode*.

O `ReturnNode` segue a mesma lógica, porém para a pilha `procs`, e com verificações adicionais para certificar que o tipo da expressão retornada é compatível no caso de um `YangNode`, ou que não haja expressão no caso de um `WujiNode`.

6 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Tendo construído toda a árvore sintática abstrata, ao final de toda a análise sintática e semântica, salvamos o nó raiz da árvore na variável `ast` (Alg. 6.1). Como mostrado no Algoritmo 4.4, todo nó da árvore possui um método `show`. Graças ao polimorfismo e herança, esse método é chamado recursivamente em cada nó, usando a implementação específica de cada um, emitindo gradativamente uma representação textual da árvore.

```
program: module_decl { ENVPUSH; } top_stmts
    { ast = new ProgramNode($1,*$3); delete $3; } ;
```

Algoritmo 6.1 – Produção de partida.

Na função `main`, somos capazes então de simplesmente executar o Algoritmo 6.2. A árvore construída e semanticamente correta deveria então ser passada ao módulo gerador de código LLVM. Separamos um arquivo dedicado a isso, em `src/codegen.cpp`, com um protótipo de implementação do padrão *Visitor*, como sugerido pelo próprio tutorial da LLVM.

```
if (ast)
    std::cout << *ast << std::endl;
```

Algoritmo 6.2 – Impressão da árvore sintática abstrata.

No entanto, devido às limitações de tempo, não fomos capazes de fazer essa etapa, limitando-nos a apenas exibir a estrutura da AST na saída padrão. Apesar disso, acreditamos que as informações contidas na árvore sintática sejam suficientes para derivar essa implementação, talvez sendo necessário apenas incrementar com algumas informações específicas da representação intermediária da LLVM.

7 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 especificar o caminho do arquivo: `$ bin/taoc entrada.tao`.

```
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 wuji main(argc: Int, argv: @@Char) {
12     fatorial(10);
13 };
```

Algoritmo 7.1 – Código do arquivo `fatorial.tao`.

No código mostrado em 7.1, temos a representação de algumas características da linguagem. Na linha 1 é definido um procedimento que calcula um fatorial de um número n, utilizando a sintaxe do `:||` (`while`). Na linha 11 há a demonstração da função principal do programa escrito em tao, utilizando o lexema wuji e o uso de ponteiros com argv além da chamada do função fatorial definida anteriormente.

No código mostrado em 7.2, é possível concluir que a sintaxe utilizada no programa 7.1 está correto, pois o código intermediário é mostrado no terminal. Note que cada função possui seu bloco, os argumentos são definidos pelo vetor após o nome da função e o seu tamanho determina a aridade do método.

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 com a seguinte mensagem: `3:5: syntax unbalance: syntax error`.

Caso o programador utilize mais de um argumento na chamada da função fatorial, como, por exemplo: `fatorial(10,2)` - não faz sentido matematicamente, mas suponha que aconteça esse erro -, o analisador semântico irá retornar um erro:

```
12:19: semantic unbalance:
          no compatible definition of `fatorial` for argument types (Int,Int)
```

```
Program(
    [
        Yang(fatorial,
                [Param(n, VarType(Int))],
                VarType(Int),
                Block(
                    [Yin(x, VarType(Int), Literal(1)),
                    Yin(i, VarType(Int), Literal(1)),
                    While(BinaryOp(<=, ID(i), ID(n)),
                    Assign(ID(i), BinaryOp(+, ID(i), Literal(1))),
                    Block(
                        [Assign(ID(x), BinaryOp(*, ID(x), ID(i)))])),
                    ID(x)])
            ),
        Wuji(main,
                [Param(argc, VarType(Int)),
                Param(argv, Ptr(@, Ptr(@, VarType(Char))))],
                Block([Call(fatorial[Literal(10)])])
            )
    ]
)
```

Algoritmo 7.2 – Código intermediário `fatorial.tao`.

Nesse caso, há um erro de semântica pois não há nenhuma definição compatível de um função com nome fatorial que possui dois argumentos `Int`. Caso, um número do tipo ponto flutuante for utilizado nessa chamada de função, `fatorial(2.9)`, o analisador também acusará um erro do tipo semântico, , pois não há uma definição de uma função com nome fatorial compatível com o tipo `Real`, conforme mostra abaixo:

```
12:17: semantic unbalance:
            no compatible definition of `fatorial` for argument types (Real)
```

Ainda é possível exemplificar o erro de não declaração de variável nessa função. Se retirarmos a declaração da variável `i`, o analisador semântico acusa o erro de que `i` não está no escopo da função fatorial: `4:11: semantic unbalance: `i` not in scope`.

É possível simular também o comportamento mostrado em 5.4. Se adicionarmos o lexema correspondente ao `break`, na linha 7, o programa estará correto, pois está dentro de um bloco de repetição. Caso o comando `break` for adicionado na linha 8, o analisador semântico mostrará um erro do tipo:

```
8:5: semantic unbalance: :||||: must be inside a loop
```

8 CONSIDERAÇÕES FINAIS

Finalizando o projeto desta linguagem, foi possível passar por todas as etapas típicas do desenvolvimento do front-end de um compilador. Desde a análise léxica, passando pela sintática, semântica e chegando na geração de código intermediário, observamos na prática vários desafios para a implementação, por vezes fazendo ajustes em etapas anteriores, o que permitiu também uma melhor compreensão de vários dos conteúdos abordados na disciplina. Por exemplo, decidir como gerenciar a tabela de símbolos e quais informações colocar nela, a escolha de construir explicitamente a árvore sintática abstrata versus emitir código durante a análise sintática, entre outros.

Por fim, podemos ressaltar que, apesar de não chegarmos a transformar o código em um binário executável, o breve estudo da arquitetura LLVM também valeu a pena, sendo possível ter contato com uma ferramenta tão poderosa. Melhorias neste projeto envolveriam então transformar para o código da LLVM de fato, e também implementar as construções da linguagem que ficaram faltando, como a importação/exportação de módulos, criação e manipulação de tipos complexos. Assim, teríamos um linguagem bastante completa.

REFERÊNCIAS

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

2 THE LLVM Compiler Infrastructure. 2022. <https://llvm.org/>. Acesso em: 31 jul. 2022.

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

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

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