

UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE FLORESTAL
CIÊNCIA DA COMPUTAÇÃO

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)
THIAGO OLIVEIRA (3037)
VITOR HUGO (3049)



COMPILADORES
TRABALHO PRÁTICO 2

FLORESTAL, MG
7 de dezembro de 2020

ATHENA SARANTÔPOULOS (2652)

JOSE GRIGORIO NETO (3046)

THIAGO OLIVEIRA (3037)

VITOR HUGO (3049)



COMPILADORES
TRABALHO PRÁTICO 2

Documentação do 2º trabalho prático parte
3 de Compiladores que tem como objetivo de
criar uma linguagem de programação

Orientador: Prof. Dr. Daniel Mendez Bar-
bosa

FLORESTAL, MG

7 de dezembro de 2020

Resumo

Este trabalho tem por finalidade o desenvolvimento de uma linguagem de programação que possa ser reconhecida por uma gramática livre de contexto e computada por um autômato de pilha. Para esta linguagem foi implementado também um compilador, que é capaz de gerar código de máquina à partir de um código fonte escrito na mesma. A objetivo principal é fomentar a aplicação de maneira prática, os conceitos e o conteúdo teórico visto em sala de aula. Sendo assim, a criação da linguagem, que envolve a escolha dos tipos primitivos, escolhas de palavras chave, palavras reservadas, paradigma de programação, gramática a ser utilizada, dentre outras decisões de projeto, serão melhor explicadas ao longo deste documento. Ao fim deste trabalho espera-se que o leitor obtenha algum entendimento acerca das principais decisões tomadas, bem como o que levou a elas e como se dá o funcionamento da linguagem, tanto sintática quanto semanticamente, sendo assim capaz de desenvolver códigos nela.

Palavras-chaves: lex, yacc, gramática, compiladores, linguagem.

Abstract

This work aims to develop a programming language that can be recognized by a context-free grammar and computed by a stack automaton. A compiler has also been implemented for this language, which is capable of generating machine code from a source code written in it. The main objective is to promote practical application, concepts and theoretical content seen in the classroom. Therefore, the creation of the language, which involves the choice of primitive types, choice of keywords, reserved words, programming paradigm, grammar to be used, among other design decisions, will be better explained throughout this document. At the end of this work, the reader is expected to gain some understanding about the main decisions made, as well as what led to them and how the language works, both syntactically and semantically, thus being able to develop codes in it.

Key-words: lex, yacc, grammar, compilers.

Sumário

1	Introdução	5
2	A Linguagem	6
2.1	Dados primitivos	6
2.2	Comandos	7
2.3	Paradigmas	8
2.4	Palavras-chave e palavras reservadas	8
2.5	Gramática	9
2.6	Modificações	15
2.7	Implementação	16
	2.7.0.1 Tabela de Símbolos	18
	2.7.1 Representação Intermediária	20
2.8	Testes	23
	2.8.1 Como executar	23
	2.8.1.1 run.sh	24
	2.8.1.2 run_tests.sh	25
3	Conclusão	30
4	Referências	31

1 Introdução

Este trabalho consiste na criação de uma linguagem. Na primeira parte, foi definida o nome da linguagem e origem do nome, tipos de dados primitivos, comandos disponíveis, paradigma de programação, palavras-chave, palavras reservadas, gramática da linguagem, com suas variáveis, terminais (tokens) e padrões de lexema dos tokens e a implementação do analisador léxico. Essa documentação equivale a terceira parte do trabalho, onde iremos utilizar o Yacc que é um analisador sintático, junto com o *lex* criado anteriormente para realizar a análise semântica. Com isso, o trabalho completo irá fazer a análise sintática, semântica e uma representação intermediária dos códigos de entrada da linguagem KUALA.



Figura 1 – Logo da linguagem

2 A Linguagem

A linguagem desenvolvida recebeu o nome de KUALA. A origem desse nome se deu pelo fato da linguagem ser inteiramente inspirada em animais, sendo todas as suas palavras-chave e palavras reservadas definidas com nomes de animais e além do fato de que o animal Coala dorme por dia 14 horas, sendo parecido com os membros do grupo. Apesar da inspiração incomum, esta linguagem incentiva os programadores (especialmente iniciantes) no aprendizado do idioma inglês, levando em consideração que todos os nomes de palavras reservadas e palavras chaves da linguagem são nomes de animais em inglês.

A linguagem criada, a KUALA, foi inspirada na linguagem de programação C. E por inspirada, quer dizer que ela possui o mesmo paradigma de programação, que é o paradigma estruturado e case sensitive, e os mesmos formatos de estruturas e comandos. Assim como na linguagem C os delimitadores de início e fim de escopo são marcados pelo uso de chaves. Os comandos de repetição (que serão melhor abordados na seção Comandos) possuem a sintaxe idêntica a da linguagem C. Além disso a linguagem permite a criação de funções, podendo receber alguns valores como parâmetro e retornando um valor que corresponde ao tipo definido para a função. Da mesma forma, é possível a definição de variáveis, também de algum tipo primitivo. Possui o comando condicional *if* e outros dois tipos de comandos de repetição. Apesar da linguagem ser inspirada na linguagem de programação C, esta não possui todo o poder computacional e nem mesmo oferece todos os recursos que a linguagem C oferece a seus programadores. Porém para a finalidade deste trabalho ela pode ser considerada suficiente, já que apresenta as principais características de uma linguagem de alto nível, além de proporcionar o poder computacional de tais linguagens.

2.1 Dados primitivos

Como a linguagem foi construído sobre o paradigma estruturado, e é baseada na linguagem C, ela possui tipos de dados muito bem definidos, sendo assim uma linguagem fortemente tipado. Por não ser do paradigma orientado a objeto, a linguagem possui apenas os tipos e estruturas definidos na própria linguagem, não possibilitando ao programador a criação de uma nova estrutura de dados. Diferentemente de C que possui um tipo de dados chamado struct, que possibilita a criação de novas estruturas de dados, para esta linguagem este recurso não foi implementado. A justificativa para isso é o fato de que levaria a um maior esforço de implementação, aumentando assim o escopo do projeto da linguagem.

Para esta linguagem foi definido os tipos de dados: inteiro, decimal, texto e boole-

ano. Cada um desses tipos de dados foi representado com um nome de animal que possui uma inicial com a mesma letra do tipo de dados que ele representa. Cada um desses quatro tipos de dados citados estão representados abaixo.

Tipo de dados em C	Tipo de dados em KUALA
<i>int</i>	<i>ibis</i>
<i>float</i>	<i>frog</i>
<i>string</i>	<i>snake</i>
<i>bool</i>	<i>bug</i>

2.2 Comandos

Esta linguagem possui, como já mencionado, um comando condicional e dois comandos de repetição. O comando condicional é de fundamental importância em linguagens de programação, pois eles dão ao programador a possibilidade de mudar o fluxo de execução do programa. Isto pois, em muitos momentos é necessário avaliar uma condição para decidir qual o próximo trecho de código a ser executado. Levando isso em consideração, foi decidido no escopo do projeto que somente um comando condicional seria implementado, sem possibilidade de uso de comandos incondicionais, pois eles podem ser mal utilizados pelo programador.

O comando *if* recebe como parâmetro uma expressão lógica relacional, que deve retornar um valor booleano, sendo *verdadeiro* ou *falso* (este valor decide qual será o próximo trecho de código a ser executado). Após a expressão do comando é possível escrever um ou mais comandos e expressões, contanto que estejam entre os delimitadores de início e fim de escopo. Além disso pode haver também o comando *else* que determina um trecho de código a ser executado caso a expressão retorne um valor falso. Ao fim desta seção será apresentada uma tabela com as relações entre os nomes dos comandos da linguagem C e KUALA.

Além deste, foram implementados dois comandos de repetição. Estes, assim como os comandos condicionais, também são fundamentais para uma linguagem de programação. Isto pois, as linguagens tem por finalidade possibilitar a automação de determinadas tarefas, que poderiam (ou deveriam) ser feitas por pessoas. Para este propósito, os comandos de repetição são necessários. Os comandos de repetição implementados foram: o comando *for* e o comando *while*.

Para o comando de repetição *for* foram implementadas duas formas de uso. Uma recebe como parâmetro três expressões, uma que será executada uma única vez ao início do comando, uma expressão lógica que verifica a condição de parada do algoritmo e uma que é executada a cada iteração do algoritmo. Após ela é executado a cada iteração do comando uma sequência de uma ou mais expressões e/ou comandos, contanto que estejam

entre os delimitadores de início e fim de escopo. No outro modo de uso deste comando, ao invés dele receber três comando, ele recebe uma variável e um vetor do mesmo tipo da variável. E a cada iteração do comando, esta variável recebe um elemento do vetor.

Por fim o comando *while*, diferentemente do comando *for* recebe somente uma expressão, que é avaliada a cada fim de iteração da mesma. Caso a condição seja *verdadeira* a execução continua por mais uma iteração. E assim como os outros comandos, ele também recebe um ou mais comandos e/ou expressões, dentro dos delimitadores de início e fim de escopo.

Abaixo é apresentada a relação de nomes dos comandos em linguagem C e KUALA.

Comandos em C	Comandos em KUALA
<i>if</i>	<i>iguana</i>
<i>for</i>	<i>fox</i>
<i>while</i>	<i>whale</i>

Por fim vale ressaltar que, para cada um dos comandos citados, não necessariamente deve haver um ou mais comandos e/ou expressões, já que eles devem funcionar mesmo que dentro dos delimitadores de escopo esteja um conjunto de zero expressões e comandos (conjunto vazio).

2.3 Paradigmas

Assim como já foi mencionado, esta linguagem possui como paradigma de programação, o paradigma estruturado. Este é um paradigma de programação simples que consiste em três elementos básicos, que são: sequência, decisão e iteração. Por ser um paradigma de programação relativamente simples, os comandos implementados para esta linguagem são suficientes para que esta linguagem funcione conforme o manda o paradigma.

2.4 Palavras-chave e palavras reservadas

As palavras-chave são identificadores predefinidos e que são reservados com significados especiais para o compilador. Nem sempre as palavras-chave são reservadas, mas na linguagem KUALA, as palavras-chave são palavras reservadas. Segue abaixo as palavras reservadas:

- *ibis* (int);
- *frog* (float);

- snake (string);
- bug (boolean);
- iguana (if);
- eel (else);
- fox (for);
- whale (while);
- kuala (main);
- rabbit (return);
- viper (void);
- Desenho do coala*;

2.5 Gramática

Inicialmente a primeira produção da linguagem, que consiste na variável de partida da mesma, representa as derivações possível para a variável *program*. Esta variável é chamada com o comando *%start program* no *translate.y* para indicar que ela é a variável de partida e consequentemente a primeira variável a ser chamada ao iniciar a análise sintática. Esta pode ser visualizada abaixo.

```
1  program : funct_list kuala funct_list
2          | funct_list kuala
3          | kuala funct_list
4          | kuala
5          ;
```

Esta produção afirma que podem haver uma lista de funções dentro de um arquivo escrito em linguagem *KUALA*, porém deve necessariamente haver exatamente uma função do tipo *KUALA*, e esta pode ser do tipo *vazio*, não ter tipo ou ser escrita utilizando-se o emoji do *kuala*.

Conforme mostra a produção, ela pode derivar na função *main* ou em outro tipo de variável da gramática e esta variável pode ser derivada concordante as produções abaixo.

```
1  funct_list : funct funct_list
2             | funct
3             ;
```

```

1      funct   : funct\_types id funct\_body
2              ;
3      \begin{lstlisting}

```

```

1      funct_body : '(' ')' simple_block
2                  | '(' ')' ';'
3                  | '(' param\_list ')' ';'
4                  | '(' param\_list ')' simple_block
5                  ;

```

Estas produções, de uma maneira bem resumida, se referem basicamente à definição de uma função e/ou desenvolvimento dela, ou seja, à estruturação lógica dela com o que ela faz e o que ela pode retornar. Lembrando que deve haver exatamente uma função do tipo *kuala* dentre essa várias outras funções que podem haver dentro do arquivo. Uma observação é que até o momento não é possível dividir os programas em arquivos diferentes, já que, como mencionado, não há espaço para inclusão de novos arquivos no arquivo principal, apenas definição e desenvolvimento de funções.

Seguindo com as produções da linguagem, é possível notar que as produções se dividem em dois tipos, as que apenas derivam em uma definição e as que desenvolvem essa função. As que desenvolvem esta função possuem a variável *funct_body*, e esta variável é a que deriva no corpo da função efetivamente. Nota-se que, ela também está apresentada na variável *kuala*, já que também possui um corpo. É possível visualizar melhor seu funcionamento através das produções abaixo.

```

1      kuala   : KUALA funct\_body
2              | VOID KUALA funct\_body
3              | INT KUALA funct\_body
4              ;

```

Conforme mostram as produções acima, é possível notar que a variável *funct_body* determina a estrutura do corpo de uma função, melhor dizendo, é derivando à partir dela que se dá a estrutura do código. Esta estrutura consiste inicialmente em um tipo de retorno para a função (caso não seja a função principal), um id para nomear a função e um par de parênteses composto de abertura e fechamento do mesmo, que pode ou não haver parâmetros dentro dele. Após isto, caso não seja apenas uma definição, ainda há a variável *block*, que é responsável por definir uma estrutura de bloco. Sua produção encontra-se abaixo, juntamente com a de *param_list* e outras que serão explicadas abaixo.

```

1  param_list : param_dclr ',' param_list
2              | param_dclr
3              | error // continue parsing
4              ;

```

```

1  param_dclr : type_consts id
2              | type_consts id '[' ']'
3              ;

```

```

1  block : simple_block
2         | simple_block stmt
3         ;

```

```

1  simple_block : '\{' '\}'
2               | '\{' stmt '\}'
3               ;

```

```

1  stmt : IF '(' logic ')' block
2         | IF '(' logic ')' stmt
3         | ELSE IF '(' logic ')' block
4         | ELSE block
5         | WHILE '(' rel ')' block
6         | FOR '(' expr_statement rel ';' attr ')' block
7         | FOR '(' expr_statement rel ';' attr ')' stmt
8         | block
9         | expr_statement stmt
10        | expr_statement
11        | error // continue parsing
12        ;

```

Estas produções informam a estrutura básica de um bloco e o que pode estar contido nela, que são basicamente, os comandos da linguagem com o *iguana* ou o *fox*, suas variantes, outros blocos e as expressões e suas variantes. Para visualizar melhor abaixo encontra-se as produções que derivam a variável *expr_statement*.

```

1  expr_statement : dclr ';'
2                 | attr ';'
3                 | call_func ';'
4                 | RETURN oper ';'
5                 | RETURN ';'
6                 | error
7                 ;

```

As expressões podem ser várias, cada uma com um propósito específico, indo desde uma chamada de função, uma atribuição, uma definição de variável e entre outros. Cada uma destas expressões possuem sua sintaxe própria, sendo a atribuição a mais simples, seguindo o mesmo padrão da linguagem C de programação, conforme abaixo.

```

1  attr      : id '=' attr_recive
2              | array_entry '=' attr_recive
3              | id INCRMT
4              | id DECRMT
5              | array_entry INCRMT
6              | array_entry DECRMT
7              ;

```

```

1  attr_recive : oper
2              | array_list
3              ;

```

```

1  array_entry : array_entry '[' oper ']'
2              | id '[' oper ']'
3              ;

```

```

1  oper      : oper '+' oper
2              | oper '-' oper
3              | oper '*' oper
4              | oper '/' oper //division by 0
5              | oper '%' oper
6              | '-' oper
7              | '+' oper
8              | '(' oper ')'
9              | consts
10             | id
11             | string
12             | array_entry
13             | call_funct
14             ;

```

Porém outras expressões possuem sintaxe própria, um exemplo disso é a definição de variáveis. Na definição de variáveis, os arrays só podem ser definidos um por linha, e só podem haver eles na linha, ou seja, cada definição de array deve conter naquela linha apenas a definição daquele array em questão e um ';' para marcar o fim da expressão.

```

1      attr      : id '=' attr_recive
2                  | array_entry '=' attr_recive
3                  | id INCRMT
4                  | id DECRMT
5                  | array_entry INCRMT
6                  | array_entry DECRMT
7                  ;

```

```

1      array      : '[' array_list ']'
2                  | '[' consts_list ']'
3                  ;

```

```

1      array_list : array ',' array_list
2                  | array
3                  ;

```

```

1      dclr       : list_dclr
2                  | type_consts id array_size
3                  | type_consts id array_size '=' array_list
4                  ;

```

```

1      list_dclr  : list_dclr ',' list_dclr
2                  | id '=' oper
3                  | id
4                  ;

```

```

1      array_size : array_size array_size
2                  | '[' consts ']'
3                  | '[' id ']'
4                  | '[' ']'
5                  ;

```

Já as chamadas de função, podem ter várias outras expressões como expressões lógicas, strings, atribuição, operações relacionais e etc. Conforme mostram as produções abaixo.

```

1      call_func  : id '(' expr_list ')'
2                  ;

```

```
1  expr_list  : expr ',' expr_list
2              | expr
3              ;
```

```
1  expr       : logic
2              | rel
3              | oper
4              ;
```

```
1  rel        : rel_oper RELOP rel_oper
2              | rel_oper
3              ;
```

```
1  rel_oper   : oper
2              | array_entry
3              ;
```

Por fim, os comandos da linguagem, que possuem um campo para a passagem de parâmetro, como o *iguana* e o *whale*. Estes comandos recebem uma expressão lógica que será verificada para determinar o fluxo de execução do programa. Esta expressão lógica segue a produção da gramática apresentada abaixo.

```
1  logic : logic logic_op logic
2         | NOT logic
3         | '(' logic ')'
4         | rel
5         ;
```

Uma gramática possui várias produções que a definem, e nem todas podem estar contidas neste relatório, porém estas servem de base para explicar de maneira prática, como funciona esta linguagem. Como já foi mencionado, KUALA uma linguagem estruturada e toda a sua sintaxe foi contruída para que se comportasse como tal. Dada esta explicação sobre a construção da gramática e como funciona a linguagem, abaixo serão apresentados, a forma de execução da linguagem, ou seja, como realizar a análise sintática de um arquivo escrito na mesma, e o resultado dos testes realizados pelos projetistas da mesma, afim de comprovar sua eficiência. Para mais informações segue em anexo o arquivo `translate.y` contendo toda a gramática da linguagem já no formato estabelecido pelo yacc.

2.6 Modificações

Nesta fase do trabalho nem tantas modificações foram realizadas em comparação às outras fases em que o trabalho ainda estava em fase de desenvolvimento. Assim como nas fases anteriores algumas modificações se fizeram necessárias devido tanto à mudanças de perspectivas dos desenvolvedores quanto aos desafios encontrados, e até mesmo devido a alguns bugs encontrados no funcionamento após o término da última fase do projeto.

De maneira mais objetiva, as mudanças se deram na gramática, na tabela de símbolos e em alguns tokens da análise léxica, que não alteram o funcionamento da linguagem, eles apenas sofreram mudança em seus nomes. Estas modificações serão melhor abordadas abaixo, com um adendo, como a tabela de símbolos sofreu alterações severas ela não será abordada aqui, ao invés disso ela será abordada na sessão de implementação. Na sessão de implementação a tabela será abordada com todo o seu desenvolvimento e funcionamento para este trabalho especificamente.

O foco maior dessa fase do compilador se tornou a análise semântica, a geração de código intermediário e a recriação da tabela de símbolos. E com essas novas implementações surgiu a necessidade de realizar algumas mudanças, que recaíram sobre a gramática da linguagem. No entanto, diferentemente da implementação anterior, nenhuma das mudanças na gramática afetaram seu funcionamento, deixando toda a gramática e o código funcionando da mesma forma. Então a pergunta que fica é: o que exatamente foi mudado? De maneira resumida, assim como na análise léxica, as mudanças não foram significativas da perspectiva de funcionamento da linguagem, porém algumas produções se tornaram menos redundantes e mais otimizadas. Isto significa que o código de um modo geral se tornou mais objetivo, indo direto ao ponto onde ele deve chegar, sem a possibilidade de realizar a mesma computação passando por outras produções, ou com produções que possuem funcionamentos semelhantes. Agrupando as produções que possuem funcionamento em comum de maneira modular e utilizando essas produções de maneira mais eficiente.

Estas mudanças representam uma pequena parte do código, entretanto se somada com a inclusão das ações semânticas que foram inseridas no código para realizar determinadas tarefas, isto pode ser considerado uma grande parte do código. Mas como a gramática em si não sofreu muita mudança, estas foram corrigidas na própria documentação na sessão da gramática. Esta atualização direta sem citar especificamente quais foram as mudanças se deu pelo fato de que, como elas foram pequenas e não alteraram o funcionamento (conforme já foi mencionado), a decisão de apenas explicar seu objetivo se tornou mais eficiente que explicar pequenos detalhes dessas pequenas modificações.

2.7 Implementação

Esta segunda parte do trabalho conta com algumas pequenas modificações no arquivo do gerador de analisador léxico *lex*, realizadas com a finalidade de deixá-lo compatível com o gerador de analisador sintático *yacc*. E como já mencionado, toda a análise sintática foi feita utilizando-se de um gerador de analisador sintático o *yacc*. Esta é mais uma das ferramentas chamadas de compilador de compiladores, já que ela é capaz de gerar partes do código do compilador de uma linguagem. O que o *Yacc* faz, fundamentalmente é, dadas as produções de uma determinada gramática (produções estas que a definem), ele é capaz de gerar um arquivo em *C* que ao ser executado, lê um fluxo de tokens gerados por um analisador léxico e determina se este fluxo de tokens condiz com a sintaxe desta gramática que foi passada inicialmente para o *translate.y*. Ou seja, se estes fluxo de caracteres está de acordo com as produções desta gramática. Sua função é basicamente dizer se um código está ou não sintaticamente correto, porém ele pode fazer mais do que isso. E foi pensando nas possibilidades que um analisador sintático pode oferecer, a fim de facilitar o desenvolvimento do código desta linguagem por parte dos programadores, sendo assim os desenvolvedores da linguagem decidiram informar um log de erros sintáticos ao programador informando a linha onde se encontra o erro. Estes detalhes de implementação serão melhor explicados abaixo.

Nesta fase do projeto, houveram modificações na implementação do *lex*, modificações estas que se deram exatamente para que ele pudesse funcionar em parceria com o *yacc*. Na fase anterior, o *lex* apenas deveria reconhecer os lexemas e dizer quais deles casaram com as expressões regulares passadas para o gerador de analisador léxico. Contudo, nessa parte ele deverá retornar tokens para que o *yacc* seja capaz de lê-los e à partir deles delibere a validade de um terminado fluxo de tokens de entrada, ou seja, se está sintaticamente correto. Além de retornar os tokens, o *lex* não gera uma tabela de tokens, pois quando utilizado junto ao *yacc*, o próprio gerador de análise sintática realiza a tarefa de gerar uma tabela contendo os tipos de tokens gerados pelo *lex*. Uma outra alteração realizada do arquivo *lex* é a contagem de linhas e caracteres. Anteriormente, essa contagem estava como optativa, todavia com esta nova implementação foi necessário numerar as linhas do código e também identificar erros por linhas. Dessa forma, o *lex* é capaz de realizar essa identificação das linhas no código e também realizar a contagem de caracteres do código. O arquivo *lex* encontra-se em anexo, porém vale ressaltar o seu funcionamento. Ele é capaz de reconhecer:

- **Palavras reservadas:** todas as palavras reservadas da linguagem, que coincidentemente são as palavras chave;
- **String:** reconhece como string, qualquer sequência alfanumérica e de símbolos que encontra-se entre aspas;

- **Espaços e quebras de linha:** todos os espaços e quebras de linha ele reconhece, mas ignora, ou seja, não gera tokens para eles, porém, realiza uma contagem como sendo parte de uma linha e como sendo um caractere (ou seja, linha e coluna do código);
- **Comentários:** reconhece comentários em linha e em blocos. Em linha utilizando duas barras(//) e em blocos utilizando barra e asterisco para abrir e asterico e barra para fechar (/* */). Também ignora, porém reconhece como parte de uma linha e como caractere;
- **Identificadores:** é capaz de reconhecer como identificadores quaisquer sequência alfanumérica, contanto que não comece com um número, podendo conter *underline* como parte do identificador. Pode começar com *underline*, porém obrigatoriamente após, deve haver outro símbolo igual (outro *underline*) ou um caractere. Nunca um número;
- **Inteiros:** é capaz de reconhecer inteiros, apesar disso não os trata como iguais, visto que é diferenciado números inteiros positivos de números inteiros negativos;
- **Decimais:** números decimais também são reconhecidos, todavia eles também são diferenciados entre decimais positivos e negativos;
- **Caracteres especiais:** como a linguagem possui como base uma representação de um coala(com caracteres especiais), é esperado que esses caracteres também sejam reconhecidos, portanto assim, como todos os outros reconhecimentos do *lex*, eles também são avaliados como parte de uma linha e reconhecidos como caractere.

Além das modificações feitas no arquivo *lex*, foi gerado o arquivo *translate.y* contendo as produções da gramática, que anteriormente apenas haviam sido definidos, contudo não estavam estruturado de acordo com a sintaxe e funcionamento do gerador de analisador sintático. O arquivo do *yacc* possui uma estruturação própria que é dividida em três partes. A primeira parte para definições, segunda parte para as produções e a terceira para funções. Entretanto, ao longo do desenvolvimento foram incluídas itens a mais. Com isso, gerou-se um arquivo que foi separado em seis partes, sendo elas:

- **includes:** onde são colocados os includes e definidas as funções utilizadas na análise sintática, bem como variáveis globais;
- **define:** onde é definido o *verbose*, que é capaz de conceder um poder a mais para o analisador sintático, ele será melhor abordado mais abaixo;
- **tokens:** onde são definidos os tokens gerados pelo *lex*, que são definidos no mesmo. Ainda assim, são inseridos no *yacc* como uma pré-definição deles;

- **precedência:** onde é definida a ordem de precedência dos operadores da linguagem, lembrando que a prioridade de execução dos operadores é definida como sendo mais à esquerda;
- **produções:** onde se encontram as produções da linguagem. Será melhor explicado na seção de *Gramática*;
- **funções:** onde são desenvolvidas funções auxiliares que foram definidas no primeiro item desta lista.

O resultado deste programa são dois arquivos, um *.c* e um *.h*, onde estão as versões do analisador sintático gerado pelo *yacc* em linguagem C. Ao ser executado juntamente com o analisador léxico gerado pelo *lex*, ele é capaz de reconhecer os tokens gerados por ele e informar caso haja algum erro de sintaxe no arquivo desenvolvido em linguagem KUALA (.kl). É exatamente nessa parte que entra o *verbose*. O analisador sintático ao reconhecer um erro, ele deve, em teoria, apenas informar ao usuário que houve um erro ao compilar o código. Entretanto, com a ajuda do *verbose*, os desenvolvedores da linguagem foram capazes de ao reconhecer um erro, o analisador sintático é capaz de informar a linha onde ocorre o erro, e com o auxílio desse *parse*, informar exatamente qual o tipo de erro que ocorreu nessa linha. Após isto, ao invés de apenas parar a execução, ele continua executando de forma a mostra ao usuário outros pontos do código onde foram detectados outros tipos de erro. Isto, não seria possível sem o *verbose*. Caso ele não seja chamado, o código apenas para de executar ao encontrar um erro, e também não informa qual o tipo do erro, apenas onde ele se encontra.

A gramática da linguagem ao ser passada para o *yacc* sofreu uma série de mudanças, e por ter passado por tantas modificações, estas serão melhores abordadas abaixo, na seção Gramática, onde as mudanças serão melhor explicadas conforme se encontram no *yacc*.

2.7.0.1 Tabela de Símbolos

Ao longo deste trabalho foi possível observar que a tabela de símbolos encontrava-se em desacordo com o código em JAVA apresentado no livro texto, o qual serviu de referência para o desenvolvimento deste projeto. Tendo isto em visto, foi decidido pela reimplementação desta, partindo do que foi desenvolvido para o compilador da literatura em linguagem JAVA. Levando em consideração que o compilador do KUALA foi desenvolvido em linguagem C, e que a mesma possui muitas diferenças em termos de recurso que a linguagem JAVA, algumas modificações um tanto quanto significativas foram realizadas no código. Na literatura, o autor utiliza de algumas estruturas como hashtable e o sistema de classes para criar um *ambiente* onde aquelas definições de variáveis serão válidas, ou

seja, um escopo. Neste trabalho, foi desenvolvido uma estrutura bastante semelhante, porém construída a partir de *structs* fornecidas pela linguagem C. Dessa forma o ambiente, ou escopo, é composto de uma hashtable (implementada pelo grupo) juntamente com um ponteiro para o escopo mais externo. Isto pois, uma variável é válida dentro de seu escopo ou sub-escopos que o compõem. Dessa forma é possível controlar onde cada variável terá validade e também qual das variáveis será utilizada dentro daquele escopo. Mas como assim? Suponha que se esteja interessado em uma variável. Para esta, é realizada uma busca dentro de seu *env*, e caso não seja encontrada procura-se no *env* anterior a este (ou seja, no escopo mais externo). No entanto suponha que esta variável foi definida com o mesmo nome dentro e fora deste escopo, em outras palavras, no escopo atual e em algum escopo mais externo. Neste caso o esperado é que se utilize a do escopo mais interno, quer dizer, a primeira definição encontrada partindo do escopo atual e seguindo esta lista de escopos mais externos. Para melhor entendimento segue a estrutura chamada *env*, que corresponde aos escopos válidos das tabelas de símbolos.

```
1 typedef struct env {  
2     hashtable_t *hash;  
3     struct env *next;  
4 } Env;
```

Olhando para este código é possível notar o que foi explicado acima, mostrando claramente as relações entre os ambientes e suas tabelas de símbolos, contudo a *hashtable* mostrada neste código ainda não foi explicada, portanto esta virá a seguir.

Conforme mencionado, cada um do ambiente possuem uma tabela de símbolos que armazena os identificadores definidos dentro daquele escopo. A estrutura escolhida para fazer este armazenamento de dados foi uma tabela hash. Este é uma estrutura em formato de tabela, que possui índices que podem ser acessados diretamente à partir de uma chave. Diferentemente de uma linked list ou uma árvore, as tabelas hash possuem uma chave ligada a cada índice de sua tabela, de forma que para acessar uma determinada linha da tabela basta informar a chave que o algoritmo retorna o conteúdo associado a esta chave. Na teoria o seu uso é bastante simples, mas na prática sua implementação se torna um tanto quanto custosa, já que longe da visão do usuário, no background da aplicação, o que o algoritmo faz é pegar as chaves passadas pelo usuário como parâmetro e executar sobre essa chave uma codificação padrão afim de encontrar o índice exato onde se encontra este elemento na tabela.

A implementação desta tabela consiste em uma estrutura chamada *hashtable* que possui dentro de sua construção, uma lista de endereços que apontam para listas encadeadas. No entanto, qual é a razão para esta tabela possuir "uma lista de listas encadeadas"? O motivo disto é a forma como esta chave foi codificada. De maneira resumida o que as funções da tabela hash fazem ao adicionar uma nova linha a tabela, é codificar de

maneira binária a chave passada como parâmetro. Dito isso, é necessário dizer também que apesar de remota, há a possibilidade de que dois elementos da tabela possuam a mesma chave. Portanto, na remota possibilidade de dois elementos da tabela possuírem a mesma chave, estas são adicionadas no mesmo índice da tabela, porém daí em diante elas são armazenadas utilizando-se uma lista encadeada. Levando em consideração que esta possibilidade é um tanto quanto remota, isto não compromete o bom funcionamento da busca do algoritmo. Segue abaixo as estruturas da tabela e de seus índices.

```
1 struct hashtable_s {  
2     int size;  
3     struct entry_s **table;  
4 };  
5  
6 struct entry_s {  
7     char *key;  
8     Symbol *value;  
9     struct entry_s *next;  
10 };
```

Como pode ser visualizado acima, a estrutura da tabela é bastante simples. Apesar de não ter sido mencionado anteriormente, a tabela é alocada do tamanho passado como parâmetro para a função de criar a tabela.

Uma curiosidade é que a *table* dentro de *hashtable* é um ponteiro de ponteiro. Porque como a tabela insere uma linha em um índice da tabela, não necessariamente em ordem (exemplo: 0, 1, 2, 3, ...), ela deve estar previamente criada em seu tamanho total. Logo, para evitar que a tabela fique com um tamanho desnecessariamente grande sem necessidade (com várias linhas vazias alocadas), os desenvolvedores optaram por armazenar uma lista de ponteiros para listas. Assim, quando um novo item é adicionado à lista ele apenas aloca este índice desse "vetor".

2.7.1 Representação Intermediária

A representação intermediária é criada pelo front-end ao analisar um programa fonte. Existe várias representações intermediárias, incluindo árvores de sintaxe e código de três endereços. Para o compilador criado, foi escolhido a representação árvore sintática abstrata. A árvore sintática é uma representação de alto nível, ou seja, essa representação está mais próxima da linguagem fonte. Uma característica importante da árvore sintática são os nós, onde os nós representam as construções do programa fonte e os filhos dos nós representam componentes significativos de uma construção.

Partindo dessa ideia, a representação intermediária implementada segue este mesmo segmento. Conforme foi mencionado, ele armazena construções do programa, porém de

forma abstraída, o que significa que determinados elementos da sintaxe da linguagem não entram nessa árvore sintática abstrata. Um exemplo disso seriam os parênteses ou as chaves. O objetivo principal é apenas guardar os elementos realmente significativos da linguagem.

A árvore é composta de nós internos e nós folhas, sendo os nós folha do tipo symbol, podendo guardar desde um literal até uma variável. Já os nós internos, eles guardam os "operadores" envolvidos na operação entre os nós filhos. Nota-se que, dessa vez não mais foi usada a palavra "folha" para se referir ao nó. Isto pois, em alguns casos os nós envolvidos nas operações serão nós internos, e o motivo disso pode ser visualizado abaixo.

```
1  typedef struct abstract_syntax_tree {
2      enum code op;
3      int val;
4      float val_float;
5      struct symbol *sym;
6      struct abstract_syntax_tree *left, *right;
7      char *str;
8  } AST;
```

```
1  enum code {
2      LIST,
3      NUM,
4      FLT,
5      STR,
6      SYM,
7      AND,
8      OR,
9      EX_EQ,
10     PLUS_OP,
11     MINUS_OP,
12     MUL_OP,
13     DIV_OP,
14     REST_OP,
15     GET_ARRAY_OP,
16     SET_ARRAY_OP,
17     CALL_OP,
18     DCLR_FUNC,
19     FUNC,
20     KUALA_AST,
21     // PRINTLN_OP,
22     IF_STATEMENT,
```

```
23     ELSE_IF_STATEMENT ,
24     ELSE_STATEMENT ,
25     BLOCK_STATEMENT ,
26     RETURN_STATEMENT ,
27     WHILE_STATEMENT ,
28     FOR_STATEMENT ,
29     LE_OP_AST ,
30     EQ_OP_AST ,
31     LT_OP_AST ,
32     GE_OP_AST ,
33     GT_OP_AST ,
34     DIF_OP_AST ,
35 };
```

Os trechos de código acima representam respectivamente um nó interno da árvore e o tipo do nó interno. Conforme pode ser observado nestes trechos, alguns desses nós internos representam *statement* ou *blocks* da gramática. E como a gramática já foi explicada, é fácil perceber que alguns desses nós internos realizam operações entre outros nós internos, até chegar aos nós folhas, que serão aqueles que possuirão símbolos. E somente para deixar claro, abaixo está o trecho de código que representa um nó da árvore sintática abstrata.

```
1     typedef struct symbol {
2         char *name;
3         int type;
4     } Symbol;
```

O código acima mostra a construção de um nó folha da árvore. Uma coisa que não foi mencionada anteriormente é sobre a construção destes nós. Para o nó folha do tipo *symbol* os seus valores representam o nome do symbol, seja um literal ou um id e o seu tipo. Já a construção do nó interno é um pouco mais interessante. Se observar bem, ele possui ponteiros para nós à sua esquerda e à sua direita. Via de regra, um nó interno sempre possui ao menos um filho à esquerda, no entanto, em alguns casos ele pode não ter um filho à direita. Este escolha se dá, pois, nem todas as operações são binárias, porém todo nó interno possui ao menos um filho. Os argumentos *val* e *val_float* não são preenchidos juntos. Sendo assim, cada nó interno possui somente um deles em sua definição. Por fim, vale notar que os nós internos também possuem um ponteiro do tipo *symbol*. Este será sempre nulo, caso seja um nó interno. Porém, caso seja um nó folha este ponteiro será alocado e os ponteiros para *left* e *right* serão nulos.

Dada as devidas explicações acerca da estrutura da árvore, é hora de dar uma breve explicação sobre sua construção. Assim como na análise ascendente, esta árvore é construída de baixo para cima, ou seja, das folhas para a "raiz" da árvore. À medida que o *yacc* vai lendo o arquivo e realizando a análise sintática do algoritmo, este vai também executando as funções de geração de código intermediário. E assim por diante o algoritmo vai construindo, partindo das produções que representam os nós folhas, e subindo na árvore. A explicação para essa construção de baixo para cima é bem simples. O *yacc* é um gerador de analisadores léxicos que coincidentemente realiza a geração de analisadores léxicos ascendentes, ou seja, que fazem a análise ascendente. Dada esta explicação fica bem claro o motivo dessa construção das folhas para a raiz, pois ela é feita juntamente com a análise sintática.

Segue abaixo uma representação intermediária, usando a árvore sintática abstrata implementada:

Código Intermediário - Abstract Syntax Tree

```
(LIST (KUALA_AST () (DECLR_FUNC () (BLOCK_STATEMENT (BLOCK_STATEMENT (EX_EQ 'int' 19) (BLOCK_STATEMENT (
1 2) (LIST (LIST 0 2) (LIST (LIST 3 4) (LIST 1 2)))))) (LIST (LIST 0 2) (LIST (LIST 3 4) (LIST (LIST 1
Q 'string' '"x-7"') (FOR_STATEMENT (LIST (EX_EQ 'i' 0) (LE_OP 'i' 10) (EX_EQ 'i' (EX_EQ 'i' (PLUS_OP
'call' (PLUS_OP (PLUS_OP 'i' 'i') 2)) (EX_EQ 'int' 19))) ())) (ELSE_IF_STATEMENT (LE_OP 'i' 5) (BLOCK_S
STATEMENT (BLOCK_STATEMENT (CALL_OP 'print' (DIV_OP 'i' 'i')) ())) (IF_STATEMENT (AND (EQ_OP 2 'i') (EQ
_OP 'print' '"Logic !!hhaey"') ())) (WHILE_STATEMENT (EQ_OP 'i' 2) (BLOCK_STATEMENT (BLOCK_STATEMENT (CA
OP (PLUS_OP '"THIS IS A FUNCTION"' 'str') '"opa"')) ())))))
```

Figura 2 – Árvore Sintática Abstrata

2.8 Testes

Nesta seção, será mostrado exemplos de execução, onde se foi criado códigos sintaticamente e semanticamente corretos e sintaticamente e semanticamente incorretos. Mas, antes de se mostrar o resultado dos testes, será mostrado como executar os arquivos .kl, isto será mostrado na seção como executar, logo abaixo.

2.8.1 Como executar

Para executar os testes, foi criado dois shell script para automatizar a execução dos arquivos de testes (.kl). Um para executar somente um arquivo de teste e o outro executa todos os testes criados. Logo abaixo será explicado cada parte criada do shell script.

2.8.1.1 run.sh

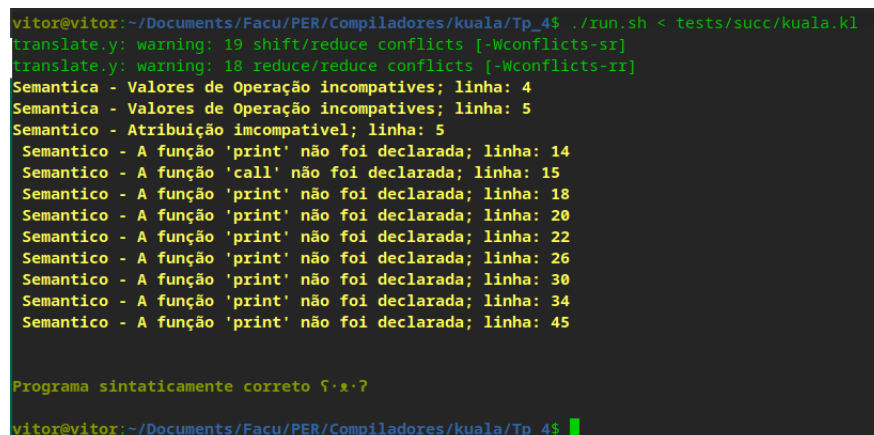
Este shell foi criado para executar somente um arquivo de teste, sendo ele sintaticamente e semanticamente correto(pasta succ) ou sintaticamente e semanticamente incorreto(pasta fail). Segue abaixo como é feito as execuções possíveis:

```
1 ./run.sh < tests/succ/<nome_do_teste>
2 ./run.sh < tests/fail/<nome_do_teste>
3 ./run.sh --verbose < tests/succ/<nome_do_teste>
4 ./run.sh --verbose < tests/fail/<nome_do_teste>
```

A explicação de cada parte desse shell será explicado a seguir, junto com o seu código.

```
1 cd src
2 cd yacc && bison -dy --debug translate.y && cd ..
3 cd lex && lex lex.l && cd ..
4 clang lex/lex.yy.c yacc/y.tab.c symbolTable/symbolTable.c
  symbolTable/hash.c AST/AST.c -ly && ./a.out $1
```

O primeiro bloco tem como função limpar o terminal e o segundo bloco tem como função entrar na pasta src, onde contém todos os arquivos que são necessários executar. Já o terceiro, quarto e quinto bloco tem como função entrar na pasta que contém o *yacc* e o outro que contém o *lex*. No *yacc* o arquivo selecionado se chama *translate.y* e do *lex* se chama *lex* e depois é necessário sair dessas pastas para realizar a execução. O último bloco executa o *lex* e o *yacc* que gera uma saída a.out que também é executada, o \$1 recebe como parâmetro *--verbose*, o que o *verbose* representa foi dita na subseção Gramática dentro de Implementação. Caso seja passado como parâmetro o *verbose*, será mostrado no terminal todos os casamentos do *lex*, já se não for escolhido o modo *verbose*, será somente mostrado se o arquivo é sintaticamente correto. Também é executado a tabela de símbolos, a tabela hash e a árvore sintática. Segue abaixo imagens de exemplo.



```
vitor@vitor:~/Documents/Facu/PER/Compiladores/kuala/Tp_4$ ./run.sh < tests/succ/kuala.kl
translate.y: warning: 19 shift/reduce conflicts [-Wconflicts-sr]
translate.y: warning: 18 reduce/reduce conflicts [-Wconflicts-rr]
Semantica - Valores de Operação incompatives; linha: 4
Semantica - Valores de Operação incompatives; linha: 5
Semantica - Atribuição imcompativel; linha: 5
Semantico - A função 'print' não foi declarada; linha: 14
Semantico - A função 'call' não foi declarada; linha: 15
Semantico - A função 'print' não foi declarada; linha: 18
Semantico - A função 'print' não foi declarada; linha: 20
Semantico - A função 'print' não foi declarada; linha: 22
Semantico - A função 'print' não foi declarada; linha: 26
Semantico - A função 'print' não foi declarada; linha: 30
Semantico - A função 'print' não foi declarada; linha: 34
Semantico - A função 'print' não foi declarada; linha: 45

Programa sintaticamente correto ¶.¶.¶
vitor@vitor:~/Documents/Facu/PER/Compiladores/kuala/Tp_4$
```

Figura 3 – Teste certo sem o modo verbose

2.8.1.2 run_tests.sh

Este shell foi criado para executar todos os arquivos de testes, sendo ele sintaticamente e semanticamente correto(pasta succ) ou sintaticamente e semanticamente incorreto(pasta fail). Segue abaixo como é feito as execuções possíveis:

```
1 ./run_tests.sh
2 ./run_tests.sh --log
3 ./run_tests.sh --log --verbose
```

A explicação de cada parte desse shell será explicado a seguir, junto com o seu código.

```
1 cd src
2 cd yacc && bison -dy --debug translate.y && cd ..
3 cd lex && lex lex.l && cd ..
4 gcc lex/lex.yy.c yacc/y.tab.c symbolTable/symbolTable.c
   symbolTable/hash.c AST/AST.c -ly
5
6 log=false;
7 verbose=0;
8
9
10 if [[ $1 == *"--log"* || $2 == *"--log"* ]]; then
11     log=true;
12 fi
13
14 if [[ $1 == *"--verbose"* || $2 == *"--verbose"* ]]; then
15     verbose="--verbose";
16 fi
17
18
19
20 for d in ../tests/succ/* ; do
21     result=$(./a.out $verbose < $d)
22     if [[ $result == *"Programa sintaticamente correto"* ]]; then
23         echo -e "Teste: $d - \e[1;32m Parsing Successful
24                 ! \e[0m\n"
25     else
26         echo -e "Teste: $d - \e[1;31;5m Parsing Fail! \e[0m\n"
27         if $log ; then
28             echo -e "LOG ERROR:"
29             echo -e "-----"
30             echo -e "$result"
```

```

30         echo -e "-----\n"
31     fi
32 fi
33 done
34
35
36 for d in ../tests/fail/* ; do
37     result=$(./a.out $verbose < $d)
38     if [[ $result == *"Programa sintaticamente correto"* ]]; then
39         echo -e "Teste: $d - \e[1;32m Parsing Successful
40                 ! \e[0m\n"
41     else
42         echo -e "Teste: $d - \e[1;31;5m Parsing Fail! \e[0m\n"
43         if $log ; then
44             echo -e "\e[1mLOG ERROR: \e[0m"
45             echo -e "-----"
46             echo -e "$result"
47             echo -e "\e[0m-----\n"
48         fi
49     fi
50 fi
51 done

```

O primeiro bloco é exatamente igual ao shell script `run.sh`, que já foi explicado logo acima. Depois desse bloco é atribuído `false` para a variável `log` e zero para a variável `verbose`, isto será utilizado para verificar se foi digitado no terminal `-log` ou `-verbose` ou os dois juntos. Logo após, é mostrado esta atribuição às variáveis, dentro do `if`. Os próximos dois blocos são praticamente iguais, onde se lê todos os arquivos da pasta `succ` e da pasta `fail` que contém os arquivos de teste. Se ao executar os arquivos for recebido a string "Programa sintaticamente correto", então é mostrado no terminal a mensagem :

```
"Teste: \ $d - \e[1;32m Parsing Successful (desenho do coala)! \e[0m\n"
```

, onde `$d` mostra o arquivo que foi lido, o `\e` é para inserir cor no texto, neste caso verde. Caso for recebido outra string, é exibido no terminal a mensagem:

```
"Teste: \ $d - \e[1;31;5m Parsing Fail! \e[0m\n"
```

, onde o `$d` mostra o arquivo que foi lido e como já dito o `\e` é para inserir cor no texto, neste caso vermelho.

Para passagem de parâmetros para o `$1` neste caso, pode ser passado `-log` ou `-verbose` e `-log` juntos. Ao ser passado somente o `log` será mostrado os erros de sintaxe.

Ao passar os dois será mostrado tanto os erros léxicos como os erros sintáticos. E ao não passar nenhum parâmetro será mostrado somente se o arquivo é sintaticamente correto. Segue abaixo imagens de exemplo:

```
vitor@vitor:~/Documents/Facu/PER/Compiladores/kuala/Tp_4$ ./run_tests.sh --log
translate.y: warning: 19 shift/reduce conflicts [-Wconflicts-sr]
translate.y: warning: 18 reduce/reduce conflicts [-Wconflicts-rr]
Teste: ../tests/succ/correto_teste_1.kl - Parsing Successful ☺
Teste: ../tests/succ/correto_teste_2.kl - Parsing Successful ☺
Teste: ../tests/succ/correto_teste_3.kl - Parsing Successful ☺
Teste: ../tests/succ/correto_teste_4.kl - Parsing Successful ☺
Teste: ../tests/succ/debug.kl - Parsing Successful ☺
Teste: ../tests/succ/expr.kl - Parsing Successful ☺
Teste: ../tests/succ/kuala.kl - Parsing Successful ☺
Teste: ../tests/succ/simple.kl - Parsing Successful ☺
Teste: ../tests/succ/sort_test.zoo - Parsing Successful ☺
Teste: ../tests/succ/testecerto.kl - Parsing Successful ☺
Teste: ../tests/succ/thiago.kl - Parsing Successful ☺
Teste: ../tests/fail/erro_semantico2.kl - Parsing Successful ☺
Teste: ../tests/fail/erro_semantico.kl - Parsing Successful ☺
Teste: ../tests/fail/incorreto_teste_1.kl - Parsing Fail!

LOG ERROR:
-----
Syntatic Error:
Erro próximo a linha(yylineno): 11
Erro próximo a linha(yyline): 12 coluna: 234
Mensagem:
  syntax error, unexpected id, expecting ';'

Lexical Error, Couldn't find a compatible lexeme: "
DEGUG -> Line: 18 -> Lineo: 17 Char: 389

Semantica - Valores de Operação Incompatives; linha: 18
Semantico - A variavel 'salario' não foi declarada; linha: 18
Semantico - Atribuição incompativel; linha: 18
Semantico - A variavel 'empregado' não foi declarada; linha: 19
Semantico - Atribuição incompativel; linha: 19
Syntatic Error:
Erro próximo a linha(yylineno): 21
Erro próximo a linha(yyline): 22 coluna: 6
Mensagem:
  syntax error, unexpected '}', expecting $end

Programa sintaticamente incorreto ☹

Teste: ../tests/fail/incorreto_teste_2.kl - Parsing Fail!

LOG ERROR:
-----
Lexical Error, Couldn't find a compatible lexeme: "
DEGUG -> Line: 10 -> Lineo: 9 Char: 16
```

Figura 4 – Todos os testes com o modo log

```
vitor@vitor:~/Documents/Facu/PER/Compiladores/kuala/tp_4$ ./run_tests.sh --log --verbose
translate.y: warning: 19 shift/reduce conflicts [-Wconflicts-sr]
translate.y: warning: 18 reduce/reduce conflicts [-Wconflicts-rr]
Teste: ../tests/succ/correto_teste_1.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/correto_teste_2.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/correto_teste_3.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/correto_teste_4.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/debug.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/expr.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/kuala.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/simple.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/sort_test.zoo - Parsing Successful 🟢?!\n
Teste: ../tests/succ/testecerto.kl - Parsing Successful 🟢?!\n
Teste: ../tests/succ/thiago.kl - Parsing Successful 🟢?!\n
Teste: ../tests/fail/erro_semantico2.kl - Parsing Successful 🟢?!\n
Teste: ../tests/fail/erro_semantico.kl - Parsing Successful 🟢?!\n
Teste: ../tests/fail/incorrecto_teste_1.kl - Parsing Fail!\n
LOG ERROR:
-----\n
Verbose Mode\n
-->Analise Lexica & Sintatica & Semantica\n
void LEXEMA: viper\n
Kuala LEXEMA: kuala\n
char especial LEXEMA: (\n
char especial LEXEMA: )\n
char especial LEXEMA: {\n
comentario simples LEXEMA: //Deveria haver um fechamento para as aspas duplas ali\n
STRING LEXEMA: snake\n
ID LEXEMA: pessoa\n
char especial LEXEMA: =\n
STRING LEXEMA: "Adivinha;\n
    ibis idade = 45, x;\n
    bug empregado = true;\n
    frog salario = 4523.00;\n\n
    //whale deveria ter o fechamento do parenteses\n
    whale(salario > 1000{\n
        iguana(empregado){\n
            pessoa = "\n
ID LEXEMA: Novo\n
Syntatic Error:\n
Erro próximo a linha(yylineno): 11\n
Erro próximo a linha(yyline): 12  coluna: 234\n
Mensagem:\n
    syntax error, unexpected id, expecting ';\n
ID LEXEMA: Nome
```

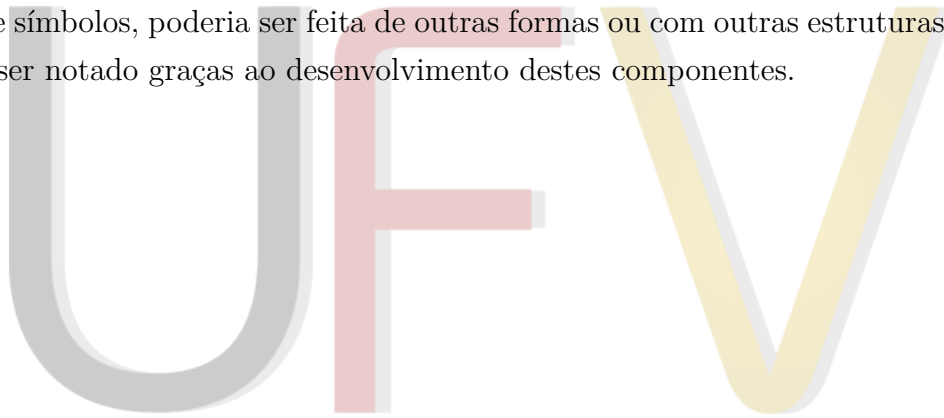
Figura 5 – Todos os testes com o modo verbose e log

```
vitor@vitor:~/Documents/Facu/PER/Compiladores/kuala/Tp_4$ ./run_tests.sh
translate.y: warning: 19 shift/reduce conflicts [-Wconflicts-sr]
translate.y: warning: 18 reduce/reduce conflicts [-Wconflicts-rr]
Teste: ../tests/succ/correto_teste_1.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/correto_teste_2.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/correto_teste_3.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/correto_teste_4.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/debug.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/expr.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/kuala.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/simple.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/sort_test.zoo - Parsing Successful ☺x-?!
Teste: ../tests/succ/testecerto.kl - Parsing Successful ☺x-?!
Teste: ../tests/succ/thiago.kl - Parsing Successful ☺x-?!
Teste: ../tests/fail/erro_semantico2.kl - Parsing Successful ☺x-?!
Teste: ../tests/fail/erro_semantico.kl - Parsing Successful ☺x-?!
Teste: ../tests/fail/incorrecto_teste_1.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_2.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_3.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_4.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_5.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_6.kl - Parsing Fail!
Teste: ../tests/fail/incorrecto_teste_7.kl - Parsing Fail!
./run_tests.sh: line 37: $d: ambiguous redirect
Teste: ../tests/fail/kuala_copy.kl - Parsing Fail!
Teste: ../tests/fail/sort_test.kl - Parsing Fail!
Teste: ../tests/fail/test_error.kl - Parsing Fail!
Teste: ../tests/fail/thiago_error.kl - Parsing Fail!
```

Figura 6 – Todos os testes sem nenhum modo

3 Conclusão

Ao longo do desenvolvimento deste projeto foi possível enfrentar alguns dos desafios da construção de um bom compilador. É difícil dizer quais foram os maiores desafios enfrentados durante esta etapa do desenvolvimento, porém é fácil dizer que todos eles foram igualmente trabalhosos e igualmente ricos em termos de conhecimento adquirido. De um modo geral, todo o trabalho gerado para a construção das partes deste compilador ajudou a melhorar outras partes do compilador que já haviam sido previamente desenvolvidas, levando a um melhor aprimoramento do compilador. Por fim, foi possível ter um contato maior com essas partes que podemos chamar de "caixas pretas" do compilador, que são as partes que apenas estudando sua teoria não é possível ter noção do quão amplas elas são. Boa parte da implementação, tanto da representação intermediária quanto da tabela de símbolos, poderia ser feita de outras formas ou com outras estruturas. Tudo isso só pode ser notado graças ao desenvolvimento destes componentes.



4 Referências

Lex - A Lexical Analyzer Generator . Disponível em: <<http://dinosaur.compilertools.net/lex/>> Acesso em: 23 de out.

Yacc: Yet Another Compiler-Compiler . Disponível em: <<http://dinosaur.compilertools.net/yacc/>> Acesso em: 23 de out.

Freepik. catalyststuff . Disponível em: <<https://www.freepik.com/free-vector/cute-koala-sleeping>> . Acesso em: 20 de out. de 2020. Tonius. Hash. Disponível em: <<https://gist.github.com/tonious/1377667>> . Acesso em: 02 de dez. de 2020.

lu1s.dragon-book-source-code. Disponível em: <<https://github.com/lu1s/dragon-book-source-code>> . Acesso em: 02 de dez. de 2020.

