

## Trabalho Prático 5 - A linguagem Laplace e o compilador Laplc

Daniel Fernandes Pinho - 2634<sup>1</sup>, Leandro Lázaro Araújo Vieira - 3513<sup>2</sup>,  
Mateus Pinto da Silva - 3489<sup>3</sup>

<sup>1</sup>Ciência da Computação – Universidade Federal de Viçosa (UFV-CAF)  
– Florestal – MG – Brasil

(daniel.f.pinho<sup>1</sup>, leandro.lazaro<sup>2</sup>, mateus.p.silva<sup>3</sup>)@ufv.br

**Resumo.** *Compiladores são, acima de tudo, implementações de linguagens formais. Neste trabalho, apresentamos o estado final do nosso compilador para a linguagem funcional estrita focada em compilação para processadores RISC, a Laplace. Ele é capaz de analisar léxico, sintático e semanticamente um código fonte e identificar e apresentar vários erros, caso existam, ou gerar um código de três endereços RISC-Like, caso os erros não existam. Utilizamos de regras léxicas simples via Flex, regras sintáticas simples via Bison e verificações semânticas através de uma potente tabela de símbolos especificada por nós. Como resultamos, conseguimos implementar boa parte de toda a especificação da nossa linguagem de programação, e codificamos um compilador potencialmente sem erros e confiável. Ao fim, apresentamos vários testes e mostramos que suas saídas estão corretas.*

### 1. Mudanças no relatório

Para esta entrega algumas mudanças foram feitas, e é fundamental citá-las e explicá-las aqui, por se tratar de um trabalho construído ao longo de 4 etapas, estas divididas em entregas.

A primeira mudança foi uma pequena adição à especificação da linguagem Laplace, em que definimos como ela se comporta em relação a coerção, coisa que esquecemos nos relatórios anteriores por não levarem em consideração os tipos. De fato, a linguagem apresenta o comportamento comum de uma linguagem funcional qualquer, porém decidimos explicitá-lo.

A segunda mudança foi a adição de uma seção para explicação das constantes. Optamos por criar um único arquivo *Header File* para a definição de todas, a fim tornar o código mais legível e, por conseguinte, facilitar a correção do professor.

A terceira mudança foi a adição de outra seção, dessa vez para enumerar todas as mensagens de erro que o compilador pode gerar. Também optamos por colocar o tipo semântico em uma seção separada, visto que ele é base para diversas outras explicações relacionadas a análise léxica e semântica.

A terceira mudança é quanto à redução do escopo da linguagem. Foram retirados todos os tipo compostos da linguagem, sejam os definidos pelo programador (ou seja, as estruturas, na linguagem chamados de *Data*) ou os tipos compostos padrão (vetores e tuplas). Além disso, retiramos também a possibilidade de importação de pacotes juntamente com a API Fluente. Fizemos isso por questões de tempo, portanto acreditamos que,

com o devido tempo e esforço, seria possível implementar o que havíamos especificado sem maiores problemas. Todos os tokens, lexemas e regras retiradas serão destacadas em vermelho para facilitação da leitura deste relatório.

Além disso, cumprimos o que prometemos e implementamos uma potente tabela de símbolos, que consegue fazer a verificação de tipo concisa que é uma das bases de qualquer compilador de linguagem funcional. Definimos também as regras semânticas. Por fim, adicionamos ao compilador um gerador de código de três endereços RISC-Like, seguindo a nossa ideia original de criar uma linguagem funcional pura para processadores RISC.

## 2. Como executar

Todo nosso trabalho foi desenvolvido sobre o sistema operacional Linux. É possível executá-lo tanto num computador que nativamente usa esse sistema operacional quanto em um com o sistema virtualizado.

Para executar é necessário ter instalado na máquina o compilador *GCC*, o gerador de analisador léxico *Flex*, o gerador de analisador sintático *Bison* e o executor automatizados de comandos *Gnu Make*. No sistema Linux Ubuntu, é possível instalar todos esses pacotes necessários através dos comandos:

```
1 sudo apt update && sudo apt upgrade
2 sudo apt install build-essentials
3 sudo apt install flex bison make -y
```

Depois disso, basta acessar a pasta raiz do projeto via terminal e digitar o seguinte comando para a compilação:

```
1 make
```

Depois de compilado, para executar o compilador de forma interativa, basta digitar o seguinte comando:

```
1 make run
```

Para executar um dos arquivos de teste feitos por nós, é possível ainda utilizar o seguinte comando (note que o arquivo exemplo é o *functions.lapl* da pasta *expected\_working*, porém poderia ser qualquer arquivo dentro das subpastas de *input*):

```
1 ./laplc < input/expected\_working/functions.lapl
```

## 3. Motivação

Efeito colateral é uma das maiores causas de inconsistências na programação. Várias linguagens trazem abordagens para evitar ou impedir totalmente os efeitos colaterais. Das abordagens, a nossa pessoalmente favorita é a das linguagens funcionais puras, que impede o programador de declarar variáveis não-constantes e de acessar variáveis fora de seu escopo. Além disso, isso torna a implementação do compilador [Aho et al. 2013] mais simples, visto que o gerenciamento de memória pode ser feito de forma mais transparente. Um bom exemplo de linguagem com essa abordagem é Haskell [O’Sullivan et al. 2008]. Entretanto, ela traz vários problemas por ter avaliação preguiçosa. Um deles é a dificuldade na previsão do tempo de execução e no gasto de memória. Há abordagens funcionais

puras não preguiçosas, como Idris [Brady 2017], porém ela também apresenta problemas. Um deles é a avaliação preguiçosa opcional, que é particularmente difícil de mesclar com a avaliação não preguiçosa. O maior deles, entretanto, também existe em Haskell, que é sua sintaxe pouco intuitiva a programadores, não apresentando parêntesis nas chamadas de funções, o que pode tornar códigos bastante ilegíveis, reduzindo a produtividade do programador. Embora Idris não use avaliação preguiçosa por padrão, ou seja, existe a ideia de fluxo de controle simples em que uma linha deve ser executada depois da anterior, não é possível declarar funções dessa forma, sendo necessário utilizar o comando `WITH` ou mônadas de controle, o que dificulta o aprendizado da linguagem.

Além disso, com a rápida evolução das arquiteturas RISC [Patterson and Hennessy 2020], os tamanhos de variáveis (como existirem inteiros de 32 e 64 bits numa mesma linguagem) começam a perder o sentido, visto que uma arquitetura assim sempre tem os registradores de mesmo tamanho. Outro fator é que exceções tendem a ter um custo muito alto em arquiteturas com processamento de instrução fora de ordem, que são as mais comuns atualmente.

## 4. A linguagem Laplace

Apresentamos a linguagem Laplace, uma linguagem funcional pura de avaliação estrita, tipagem forte e estática e sintaxe fortemente inspirada em Typescript [Cherny 2019], permitindo funções de múltiplas linhas. É focada em legibilidade, confiabilidade e produtividade. Não apresenta o mesmo tipo em tamanhos diferentes e tem seus erros encapsulados, não disparando exceções. Além disso, permite acesso a E/S sem utilização de mônadas e sempre retorna um número inteiro ao sistema operacional para verificação de erros, semelhante a um bom uso da linguagem C. A linguagem é uma tentativa de trazer bons recursos de boas implementações do paradigma estrutural ao paradigma funcional puro.

### 4.1. A origem do nome da linguagem

A ideia do nome surgiu da carta D/D/D King Zero Laplace do jogo Yu-Gi-Oh, homenagem ao cientista matemático Laplace que teve importantes contribuições à álgebra linear. A carta é uma das favoritas do jogo de um dos integrantes do grupo e é muito poderosa no jogo por utilizar a força das cartas do oponente ao seu favor. A linguagem, da mesma forma, tenta utilizar a força de boas implementações do paradigma estrutural para fortalecer o funcional puro. Além disso, o nome é uma referência a linguagem Haskell, pois também escolhemos o nome de um cientista matemático.

Além disso, a partir do nome Laplace, são definidos os seguintes nomes para o trabalho:

Nome	Significado
.lapl	Formato de um arquivo código fonte Laplace.
laplc	Compilador Laplace.

### 4.2. Tipos primitivos

A linguagem conta com somente quatro tipos primitivos, para aumentar sua legibilidade, que são booleanos, caracteres, pontos flutuantes e inteiros. O tamanho deles varia por

arquitetura. Por exemplo, em uma arquitetura de 64 bits, o ponto flutuante e o inteiro terão tamanho igual a 64 bits.

Tipos	Valores
Bool	False ou True
Char	Caracteres da tabela ASCII
Float	Ponto flutuante de precisão de tamanho variado
Int	Inteiro de tamanho variado

### 4.3. Tipos compostos

Embora tenham sido retirados, a linguagem Laplace possui três formas de tipos compostos: tuplas, vetores e Datas (o equivalente a *struct* da linguagem C). Tuplas são mapeamentos de tipos diferentes, vetores é o equivalente para apenas um tipo, e Datas é um tipo composto criado pelo programador.

Tipos	Valores
Tuple	(tipo1, tipo2...)
Array	[tipo1, tipo1...]
Data	{nomeQualquer: tipo1, nomeQualquer: tipo2...}
String	“Char, Char...”

### 4.4. Comandos disponíveis e o seu funcionamento

Como a linguagem é funcional pura, não existem comandos na linguagem. Só existem expressões.

### 4.5. Paradigma de programação e recursos disponíveis

A linguagem Laplace oferece somente o paradigma funcional puro. Seus principais recursos são as funções anônimas, recursões e operações sobre listas (mapeamento, filtragem e reduções). Outro diferencial é que as operações sobre listas suportarão o uso do índice a ser trabalho, permitindo que expressões matemáticas como somatórios possam ser calculados desta forma, recurso presente em Typescript, porém ausente em Haskell.

### 4.6. Palavras-chave e palavras reservadas

Em Laplace, todas as palavras-chave são palavras reservadas. Elas estão especificadas juntamente com os outros tokens pensados.

### 4.7. Coerção

Laplace não permite coerção, ou seja, todos os tipos precisam ser convertidos de forma explícita pelo programador.

## 5. Constantes do compilador

Definimos todas as constantes do compilador em um arquivo *Header File* chamado *constants.h*. Nele, definimos todos os tamanhos máximos permitido pelo compilador, visto que optamos por utilizar ao máximo alocação estática a fim de evitar falhas de segmentação de desreferencialização de ponteiros nulos.

```

1 #define SYMBOL_TABLE_SIZE 1024
2 #define MAX_IDENTIFIER_SIZE 64
3 #define MAX_ARGUMENT_NUMBER 32
4
5 #define MAX_CODE_GEN_LINE_SIZE 256
6 #define MAX_CODE_GEN_LINE_NUMBER 2048
7 #define MAX_GUARD_LINE_NUMBER 64

```

## 6. As mensagens de erro

Optamos por criar um compilador que se esforça apenas em encontrar o primeiro erro no código-fonte, descrevê-lo da melhor forma possível e abortar a compilação. Fizemos isso por várias questões. Primeiro, encontrando apenas o primeiro erro, a compilação fica muito mais rápida, principalmente quando esse erro é mais inicial no código-fonte. Segundo, acreditamos que essa abordagem gera mensagens de erro com maior clareza. De fato, analisando compiladores como o do Java (o `javac`), caso o programador não defina uma variável `X` (por exemplo) e a use, isso gerará um erro por utilização mal feita, e as variáveis definidas utilizando a `X` gerarão outros erros, criando mensagens gigantescas e confusas. E terceiro, o programador precisa resolver os erros em sequência, do primeiro ao último, o que faz bastante sentido no `laplc`, que gera o primeiro erro do código-fonte de forma clara. Por último, isso torna o trabalho de fazer um compilador muito mais simples.

Todos os erros, exceto o `yyerror` (que é a redefinição do erro padrão do BISON) estão no módulo certo, e são divididos nas três grandes classes de erro que compiladores são capazes de detectar: léxico, sintático e semântico. Todos apresentam o número da linha em que o erro apareceu, a classe de erro e algumas informações adicionais.

Tipo de erro	Mensagem de erro
Léxico	<code>ErrorMessageLexicalError</code>
Sintático	<code>yyerror</code>
Sintático	<code>ErrorMessageVariableOrConstDoesNotExist</code>
Sintático	<code>ErrorMessageFunctionDoesNotExist</code>
Sintático	<code>ErrorMessageVariableOrConstDoubleDeclaration</code>
Sintático	<code>ErrorMessageFunctionDoubleDeclaration</code>
Semântico	<code>ErrorMessageExpressionTypeError</code>
Semântico	<code>ErrorMessageFunctionDeclarationIncorrectArgumentNumber</code>
Semântico	<code>ErrorMessageFunctionWrongReturnType</code>
Semântico	<code>ErrorMessageFunctionCallDoesNotExist</code>
Semântico	<code>ErrorMessageFunctionCallIncorrectArgumentNumber</code>
Semântico	<code>ErrorMessageFunctionCallIncorrectArgumentType</code>

## 7. Tipo semântico

Optamos por utilizar como tipo semântico uma estrutura que contém uma enumeração dos tipos e uma `String`, sendo essa enumeração um marcador do tipo daquele lexema e a `String` o lexema em si. Por exemplo: Caso o programador digite na Laplace o valor inteiro 5, ele será passado como uma estrutura que contém uma enumeração valendo `INT` e uma `String` “5” para o Bison. Além disso, aproveitamos essa construção de tipo semântico nas regras da análise semântica, o que será explicado mais adiante.

```

1 typedef struct
2 {
3     char String[MAX_IDENTIFIER_SIZE];
4     Type type;
5 } SemanticValue;

```

## 8. Análise léxica

Primeiro iremos a definição léxica da linguagem.

### 8.1. Definições léxicas

Aqui estão algumas definições léxicas úteis para a geração de tokens da linguagem:

Definição léxica	Gramática
digit	[0-9]
number	{digit}+
lowercase	[a-z]
uppercase	[A-Z]
ascii_char	(Qualquer caractere da tabela ASCII)

### 8.2. Os Tokens e os valores semânticos gerados

Como já explicado em tipo semântico (Veja a seção 7), os tokens são enviados já tipados. Isso vale tanto para os valores crus quanto para as definições de tipo. Além disso, os valores crus ainda vão com o lexema do valor.

Lexema	Identificador de Token	Valor semântico
let	LET	
const	CONST	
function	FUNCTION	
import	IMPORT	
as	AS	
from	FROM	
match	MATCH	
data	DATA	
default	DEFAULT	
type_int	TYPE_INT	(, INT)
type_float	TYPE_FLOAT	(, FLOAT)
type_char	TYPE_CHAR	(, CHAR)
type_bool	TYPE_BOOL	(, BOOL)
type_struct	TYPE_STRUCT	
open_parenthesis	OPEN_PARENTHESIS	
close_parenthesis	CLOSE_PARENTHESIS	
open_bracket	OPEN_BRACKET	
close_bracket	CLOSE_BRACKET	
open_brace	OPEN_BRACE	
close_brace	CLOSE_BRACE	

arrow	ARROW	
colon	COLON	
comma	COMMA	
semicolon	SEMICOLON	
period	PERIOD	
definition	DEFINITION	
not	NOT	
plus	PLUS	
minus	MINUS	
division	DIVISION	
multiplication	MULTIPLICATION	
pow	POW	
mod	MOD	
equal	EQUAL	
not_equal	NOT_EQUAL	
greater	GREATER	
less	LESS	
greater_or_equal	GREATER_OR_EQUAL	
less_or_equal	LESS_OR_EQUAL	
and	AND	
or	OR	
value_int	VALUE_INT	(String do lexema, INT)
value_float	VALUE_FLOAT	(String do lexema, FLOAT)
value_bool	VALUE_BOOL	(String do lexema, BOOL)
value_char	VALUE_CHAR	(String do lexema, CHAR)
value_string	VALUE_STRING	String do lexema
lowercase_identifier	LOWERCASE_IDENTIFIER	(String do lexema, STRING)
uppercase_identifier	UPPERCASE_IDENTIFIER	String do lexema

## 9. Análise sintática

Algumas observações precisam ser feitas, que talvez não sejam tão familiares ao leitor. O termo **guards**, guardas em português, que é uma forma mais potente de switch case de linguagens funcionais, que na linguagem Laplace é iniciada com a palavra-reservada *match*.

Optamos por especificar as regras sintáticas da forma mais simples o possível, a fim de evitar ambiguidades e os erros do Bison (como os de Shift/Reduce e Reduce/Reduce). Felizmente conseguimos criar uma gramática sem nenhum erro desse tipo. Abaixo ela está definida. Em preto estão os terminais e operadores do Bison, em rosa os Tokens e em vermelho o que foi retirado pela redução de escopo (note, por favor, que para fins de ser possível colorir, tivemos que colocar tudo que foi removido no final e não na ordem natural do que era na gramática).

Além disso, embora faça parte da análise sintática a verificação se um dado identificador existe ou não, optamos por explicá-lo juntamente na análise semântica. Fizemos isso por se adequar a nossa própria construção de compilador, que verifica se um dado

identificador ou não existe perguntando o tipo dele. Caso o retorno seja um tipo válido, o identificador existe e é do tipo retornado. Caso o retorno seja UNDEFINED, o identificador não existe.

```
1 run:
2   | statements run
3 ;
4
5 statements: variable_definition
6   | const_definition
7   | function_declaration
8 ;
9
10 type_specifier: TYPE_INT
11   | TYPE_FLOAT
12   | TYPE_CHAR
13   | TYPE_BOOL
14 ;
15
16 operator: NOT
17   | PLUS
18   | MINUS
19   | DIVISION
20   | MULTIPLICATION
21   | MOD
22   | EQUAL
23   | NOT_EQUAL
24   | GREATER
25   | LESS
26   | GREATER_OR_EQUAL
27   | LESS_OR_EQUAL
28   | AND
29   | OR
30   | XOR
31 ;
32
33 value: VALUE_INT
34   | VALUE_BOOL
35   | VALUE_CHAR
36   | VALUE_FLOAT
37 ;
38
39 element: IDENTIFIER
40   | value
41   | function_call
42 ;
43
44 expression: element
45   | element operator expression
46   | OPEN_PARENTHESIS expression CLOSE_PARENTHESIS
47   | OPEN_PARENTHESIS expression CLOSE_PARENTHESIS operator expression
48 ;
49
50 function_element: IDENTIFIER
51   | value
52 ;
```



```

53
54 function_expression: function_element
55 | function_element operator function_expression
56 | OPEN_PARENTHESIS function_expression CLOSE_PARENTHESIS
57 | OPEN_PARENTHESIS function_expression CLOSE_PARENTHESIS operator
  function_expression
58 ;
59
60 function_guard_expression: function_element
61 | function_element operator function_guard_expression
62 | OPEN_PARENTHESIS function_guard_expression CLOSE_PARENTHESIS
63 | OPEN_PARENTHESIS function_guard_expression CLOSE_PARENTHESIS
  operator function_guard_expression
64 ;
65
66 variable_definition: LET IDENTIFIER COLON type_specifier DEFINITION
  expression SEMICOLON
67 ;
68
69 const_definition: CONST IDENTIFIER COLON type_specifier
  DEFINITION expression SEMICOLON
70 ;
71 ;
72
73 guards: function_element operator function_element COLON OPEN_BRACE
  function_guard_expression CLOSE_BRACE SEMICOLON
74 | DEFAULT COLON OPEN_BRACE function_guard_expression CLOSE_BRACE
  SEMICOLON
75 | function_element operator function_element COLON OPEN_BRACE
  function_guard_expression CLOSE_BRACE SEMICOLON guards
76 ;
77
78 function_definition: OPEN_PARENTHESIS function_definition_parameters
  CLOSE_PARENTHESIS ARROW OPEN_BRACE function_expression CLOSE_BRACE
79 | OPEN_PARENTHESIS function_definition_parameters CLOSE_PARENTHESIS
  ARROW MATCH OPEN_BRACE guards CLOSE_BRACE
80 ;
81
82 function_types: type_specifier
83 | type_specifier COMMA function_types
84 ;
85
86 function_definition_parameters: IDENTIFIER
87 | IDENTIFIER COMMA function_definition_parameters
88 ;
89
90 function_call_parameters: expression
91 | expression COMMA function_call_parameters
92 ;
93
94 function_declaration: FUNCTION IDENTIFIER COLON OPEN_PARENTHESIS
  function_types CLOSE_PARENTHESIS
95 | type_specifier DEFINITION function_definition SEMICOLON
96 ;
97
98 function_call: IDENTIFIER OPEN_PARENTHESIS function_call_parameters
  CLOSE_PARENTHESIS

```

99 ;

```
1 statements: data_declaration
2   | function_import
3 ;
4
5 vec_type: OPEN_BRACKET type_specifier SEMICOLON VALUE_INT CLOSE_BRACKET
6 ;
7
8 type_specifier: TYPE_STRUCT
9   | vec_type
10  | UPPERCASE_IDENTIFIER
11 ;
12
13 aux_value_vec: value
14   | value COMMA aux_value_vec
15 ;
16
17 value_vec: OPEN_BRACKET aux_value_vec CLOSE_BRACKET
18 ;
19
20 aux_struct_value: LOWERCASE_IDENTIFIER COLON value
21   | LOWERCASE_IDENTIFIER COLON value COMMA aux_struct_value
22 ;
23
24 struct_value: OPEN_BRACE aux_struct_value CLOSE_BRACE
25 ;
26
27 value: VALUE_STRING
28   | value_vec
29   | struct_value
30 ;
31
32 parameter: fluent_api
33 ;
34
35 aux_struct_definition: LOWERCASE_IDENTIFIER COLON type_specifier
36   | LOWERCASE_IDENTIFIER COLON type_specifier COMMA
37     aux_struct_definition
38 ;
39
40 struct_definition: OPEN_BRACE aux_struct_definition CLOSE_BRACE
41 ;
42
43 data_declaration: DATA UPPERCASE_IDENTIFIER COLON TYPE_STRUCT
44   DEFINITION struct_definition SEMICOLON
45 ;
46
47 function_package: LOWERCASE_IDENTIFIER
48   | LOWERCASE_IDENTIFIER COMMA function_package
49 ;
50
51 function_import: FROM UPPERCASE_IDENTIFIER IMPORT function_package
52 ;
```

## 10. Tabela de símbolos

Optamos por implementar uma potente tabela de símbolos, capaz de fazer diversas verificações de tipo, como o tipo de retorno e o tipo dos argumentos de cada função. Note que, caso o identificador passado não exista, as funções retornam o tipo *UNDEFINED*. Fizemos isso para reduzir o código e evitar comportamentos inesperados de código, pois quanto menos código, menos *bugs* geralmente. Como já explicada na seção de Constantes do compilador (Veja a seção 5), toda a tabela de símbolos é alocada de forma estática a fim de evitar falhas de segmentação por desreferencialização de ponteiros.

```
1 typedef struct
2 {
3     SymbolTableEntry table[SYMBOL_TABLE_SIZE];
4     size_t next_empty_slot;
5 } SymbolTable;
6
7 int SymbolTableInitialize(SymbolTable *symbolTable);
8 int SymbolTablePrintf(SymbolTable *symbolTable);
9
10 int SymbolTableAddLet(SymbolTable *symbolTable, char *identifier, Type
    type);
11 int SymbolTableAddConst(SymbolTable *symbolTable, char *identifier,
    Type type);
12 Type SymbolTableGetVariableOrConstType(SymbolTable *symbolTable, char *
    identifier);
13
14 int SymbolTableAddFunctionArgumentType(SymbolTable *symbolTable, Type
    argumentType);
15 int SymbolTableFinishAddFunction(SymbolTable *symbolTable, char *
    identifier, Type returnType);
16
17 Type SymbolTableGetFunctionReturnType(SymbolTable *symbolTable, char *
    identifier);
18 int SymbolTableTestFunctionArgumentTypes(SymbolTable *symbolTable, char
    *identifier, Type *argumentTypes, int argumentNumber);
```

**Fragmento de código 1. Interface da tabela de símbolos.**

Cada elemento da tabela de símbolos contém uma String para identificador, uma enumeração de definição de tabela de símbolos (que pode ser UNDEFINED, LET, FUNCTION ou CONST, representando não-inicializado, variável, função e constante, respectivamente), o tipo de retorno (que é o tipo da variável ou da constante, se o elemento for uma variável ou constante), um vetor de tipo de argumentos (que todos são UNDEFINED se o elemento não for uma função) e um inteiro para o número de argumentos (que vale zero se o elemento não for uma função).

```
1 typedef struct
2 {
3     char identifier[MAX_IDENTIFIER_SIZE];
4
5     SymbolTableDefinition definition;
6     Type returnType;
7     Type argumentTypes[MAX_ARGUMENT_NUMBER];
8
9     int argumentNumber;
```

```

10
11 } SymbolTableEntry;

```

**Fragmento de código 2. Definição do tipo elemento da tabela de símbolos.**

## 11. Análise semântica

Usamos bastante das operações de BISON com tipos semânticos, definindo valores semânticos para as produções do BISON além das já definidas para os Tokens gerados pelo Flex (Veja a seção 8.2). Assim, começaremos essa seção explicando como funcionam as operações que utilizamos em alto nível. Por exemplo:

```

1 expression: element operator expression {
2     if ($1.type!=$3.type) ErrorMessageExpressionTypeError($1.type, $3.
3         type, yylineno);
4     $$type=$1.type;
5 }

```

**Fragmento de código 3. Fragmento da análise semântica Laplace.**

O código entre chaves define o que deverá ser feito ao encontrar a produção especificada (ou seja, tradução dirigida por sintaxe). No caso, a produção é *element operator expression*. Há algumas macros importantes. A macro \$1 representa o primeiro elemento da produção. No caso, refere-se a *element* (lembrando que a contagem é, estranhamente, iniciada em 1 e não em 0). Assim, se o tipo de *element* for diferente de *expression*, é evocado um erro semântico de erro de tipo de expressão. A macro \$\$ refere-se a cabeça, ou seja, o *expression* inicial seguido de dois pontos. Assim, caso o tipo seja o mesmo (o que é o esperado em um código correto Laplace), a produção resultante terá o mesmo tipo que qualquer um de *element* ou *expression* da produção. Essa elevação dos tipos das produções foi amplamente utilizada por nós para facilitar a escrita do código.

Além disso, optamos por utilizar o retorno da tabela de símbolos UNDEFINED (Veja a seção 10) para verificar se um dado identificador existe ou não. Caso ele seja UNDEFINED (ou seja, não exista), uma mensagem de erro é invocada. Por exemplo:

```

1 element: IDENTIFIER {
2     $$type=SymbolTableGetVariableOrConstType(&globalScope, $1.String
3 );
4     if ($$.type==UNDEFINED) ErrorMessageVariableOrConstDoesNotExist(
5         $1.String, yylineno);
6 }

```

**Fragmento de código 4. Outro fragmento da análise semântica Laplace.**

Utilizando essas duas técnicas, é possível fazer toda a verificação semântica de variáveis e expressões. Porém, é necessário um pouco mais de poder computacional do que a Gramática Livre de Contexto (através do analisador LR) do BISON para verificar a quantidade e os tipos dos argumentos, seja na definição de funções, seja na chamada de funções. Assim, utilizamos de vetores e contadores globais para isso. Por exemplo:

```

1 function_call_parameters: expression {
2     argumentTypes[parameterTypeCounter]=$1.type;
3     parameterTypeCounter++;
4 }
5

```

```

6 function_call: IDENTIFIER OPEN_PARENTHESIS function_call_parameters
  CLOSE_PARENTHESIS {
7     if (SymbolTableGetFunctionReturnType(&globalScope, $1.String)==
      UNDEFINED) ErrorMessageFunctionDoesNotExist($1.String, yylineno);
8
9     switch (SymbolTableTestFunctionArgumentTypes(&globalScope, $1.
      String, argumentTypes, parameterTypeCounter))
10    {
11        case 1:
12            ErrorMessageFunctionCallIncorrectArgumentType($1.String,
      yylineno);
13            break;
14
15        case 2:
16            ErrorMessageFunctionCallIncorrectArgumentNumber($1.String,
      yylineno);
17            break;
18
19        case 3:
20            ErrorMessageFunctionCallDoesNotExist($1.String, yylineno);
21            break;
22
23        default:
24            break;
25    }
26
27    parameterTypeCounter = 0;
28 }
29 ;

```

**Fragmento de código 5. Outro fragmento da análise semântica Laplace.**

Aqui, os argumentos são contados e tem seus tipos armazenados durante as produções de *function\_call\_parameters*. Depois, quando tal produção retorna para *function\_call*, é verificado o tipo de retorno da função, e depois é chamada uma função que retorna diferentes valores dentro do *switch*. Caso retorne 0, tudo está correto. Caso retorne 1, o tipo de algum dos argumentos está incorreto. Caso retorne 2, o número de argumentos está incorreto. E caso retorne 3, a função não existe na tabela de símbolos. Tal construção é bem parecida também na construção de funções. Assim, é possível fazer toda a análise semântica sem problemas.

### 11.1. O problema dos escopos

Além disso, é importante citar que escopos também foram considerados aqui, é claro. Como Laplace é uma linguagem funcional pura, só existem dois escopos possíveis: o de função e o global. Acesos a variáveis (ou funções) de um escopo no outro são proibidos, ou seja, os identificadores globais e de função são conjuntos totalmente disjuntos. Para isso, optamos por alocar duas tabelas de símbolos para representar isso, uma global e uma de função. Ambas são inicializadas no começo do processo de compilação. Entretanto, a de função é sempre reinicializada ao fim da compilação de cada função (visto que identificadores de uma função não podem ser usados em outra), enquanto a global nunca é reinicializada. Dessa forma (bastante simples, por sinal), conseguimos resolver o problema dos escopos na nossa linguagem.

```

1 int main(void)
2 {
3     SymbolTableInitialize(&localScope);
4     SymbolTableInitialize(&globalScope);
5
6     [...]
7 }

```

**Fragmento de código 6.** As duas tabelas de símbolo sendo inicializadas na função main. Optamos por utilizar local ao invés de function pois o nome function já havia ser usado bastante, e poderia confundir.

## 12. Geração de código intermediário

A Laplace é uma linguagem funcional estrita focada em processadores RISC. Assim, para seguir nossa própria especificação da linguagem Laplace, optamos por definir um código intermediário de três endereços RISC-Like, que lembra muito o Assembly textual de processadores RISC, e em especial o do MIPS, que é o que melhor conhecemos por o termos estudado a fundo na disciplina de Organização de Computadores 1. Evidentemente, não utilizamos conceitos como memória e registradores, mas sim de variáveis (temporárias ou não), que funcionam como registradores infinitos cujos valores são definidos e nunca modificados (pois a Laplace é funcional pura).

### 12.1. Um código de três endereços RISC-Like

Optamos por manter os nomes de variáveis, funções declaradas no código-fonte no código intermediário, a fim de criar um código intermediário instrumentável, ou seja, que é possível fazer pequenas modificações posteriores à compilação. Outro ponto que facilita essas modificações é que não fizemos remendos, ou seja, todos os saltos são feitos através de rótulos. Por fim, criamos um sistema de prefixo de nomes para facilitar a leitura e evitar conflitos de nomes (todas essas ideias foram amplamente inspirados no código intermediário da LLVM, o LLVM IR [Hsu 2021]). Descrevemos aqui os prefixos de operadores criados por nós:

Prefixo	Significado
<code>_t</code>	Variável temporária
<code>_f</code>	Variável dentro de escopo de função
<code>_tf</code>	Variável temporária dentro de escopo de função
<code>_e</code>	Rótulo de endereço

**Tabela 2.** Os diferentes prefixos do código do código de três endereços RISC-Like pensado por nós.

A maioria das operações tem três operadores, sendo o primeiro o de destino, o segundo como o primeiro operando e o terceiro como segundo operando, como indicado na tabela, embora existam exceções:

Operador	Exemplo	Eq. no livro-texto	O que faz
not	not _t0, x	_t0 = ~x	Nega x; guarda em _t0.
add	add _t0, x, y	_t0 = x + y	Soma x e y; guarda em _t0.
sub	sub _t0, x, y	_t0 = x - y	Subtrai y de x; guarda em _t0.
div	div _t0, x, y	_t0 = x / y	Divide x por y; guarda em _t0.
mul	mul _t0, x, y	_t0 = x * y	Multiplica x por y; guarda em _t0.
mod	mod _t0, x, y	_t0 = x % y	Gera o resto da divisão de x por y; guarda em _t0.
eq	eq _t0, x, y	_t0 = x == y	Compara x e y; guarda em _t0.
ne	ne _t0, x, y	_t0 = x != y	Verifica se x e y são diferentes; guarda em _t0.
gt	gt _t0, x, y	_t0 = x > y	Verifica se x é maior do que y; guarda em _t0.
lt	lt _t0, x, y	_t0 = x < y	Verifica se x é menor do que y; guarda em _t0.
ge	ge _t0, x, y	_t0 = x >= y	Verifica se x é maior ou igual do que y; guarda em _t0.
le	le _t0, x, y	_t0 = x <= y	Verifica se x é menor ou igual do que y; guarda em _t0.
and	and _t0, x, y	_t0 = x & y	Conjunção x e y; guarda em _t0.
or	or _t0, x, y	_t0 = x   y	Disjunção x e y; guarda em _t0.
xor	xor _t0, x, y	_t0 = x ^ y	Disjunção exclusiva x e y; guarda em _t0.
mov	mov _t0, x	_t0 = x	Copia o valor de x; guarda em _t0.
beq	beq _t0, 1, _efuncao	if _t0==1 goto _efuncao	Verifica se _t0==1; se for verdade, salta para _efuncao.
j	j _efuncao	goto _efuncao	Salta para _efuncao.
jal	jal _efuncao	# Não existe	Salva endereço do chamado+1 em variável especial única e implícita; salta para _efuncao.
jr	jr	# Não existe	Salta para endereço da variável especial única e implícita.

**Tabela 3. Descrição completa do código de três endereços RISC-Like pensado por nós.**

Tais exceções serão melhores explicadas aqui. Primeiro, o operador *not* é unário, ou seja, apenas nega um operando. Outro é o *mov*, que na verdade é uma pseudo-instrução *add*, porém que seu terceiro operando é a constante 0.

Sobre os saltos, vários foram usados. O primeiro é o *beq* (salta se for igual) que verifica se os dois primeiros elementos são iguais e se forem, salta para o endereço que está no terceiro operando. O operador *j* é um salto incondicional, que só recebe um operando que é o endereço do salto. De forma muito próxima, o operador *jal* faz a mesma coisa, porém salva em uma variável especial única e implícita o endereço da linha logo abaixo, que será usada para voltar do retorno de uma função, por exemplo. O seu operador parceiro, *jr* salta exatamente para o endereço dessa variável especial única e implícita.

## 12.2. Como o laplc gera código intermediário

Como o laplc é um compilador para estudos, optamos por simplesmente imprimir o código intermediário na saída padrão (terminal) quando o código está correto. Isso facilitou a escrita do código (visto que não é necessário ponteiros para arquivo, tratamento de exceções que isso gera, etc) e facilita a correção e comparação da saída. Além disso, optamos por sempre imprimir o código das funções antes do código principal, e o ponto de entrada do código é sempre o rótulo *main*.

Dadas essas informações, é muito simples gerar o código. Ao fim de cada uma das ações semânticas, basta imprimir o código intermediário, visto que caso o fluxo de execução do compilador chegue até lá, o código-fonte estará correto. Utilizamos então de dois buffers para guardar o código intermediário. Um guarda o código das funções e o outro guarda o código principal (*main*). Ao final da execução do BISON, o código intermediário das funções é impresso, e logo depois o código intermediário principal é impresso, gerando os resultados esperados.

```
1
2 [...]
3
4 int mainLineCounter;
5 int functionLineCounter;
6
7 char mainTextSection[1024][64];
8 char functionTextSection[1024][64];
9
10 [...]
11
12 int main(void)
13 {
14     [...]
15     yyparse();
16
17     for(size_t i=0; i<functionLineCounter; i++){
18         printf("%s", functionTextSection[i]);
19     }
20     printf("\n_emain:\n");
21     for(size_t i=0; i<mainLineCounter; i++){
22
23         printf("%s", mainTextSection[i]);
24     }
25     [...]
```



```
26 }
```

### Fragmento de código 7. Buffers utilizados para geração do código intermediário.

Além disso, a exata mesma abordagem é feita para imprimir as guardas antes de seu fluxo de execução, e optamos por não explicá-la novamente aqui para que a documentação não fique ainda maior do que já está.

## 13. Código produzido e testes executados

### 13.1. Códigos com erros previstos

Primeiro, fizemos vários códigos incorretos na linguagem Laplace para verificar se as mensagens de erro estão funcionando.

#### 13.1.1. Erro léxico: aspas não fechadas

```
1 let letrinha:Char='F;
```

Saída:

```
1 laplc: lexical error on line 1
```

Era necessário fechar aspas depois de F, porém isso não foi feito. Assim, não é gerado nenhum Token, o que caracteriza erro léxico que é apropriadamente exibido pelo Laplc.

#### 13.1.2. Erro sintático: dupla declaração de função

```
1 function soma: (Float, Float) Float = (x, y) => {x+y};  
2 function soma: (Float, Float) Float = (x, y) => {x+y};
```

Saída:

```
1 laplc error: syntax error, function soma on line 2 is double declared
```

Houve duas declarações com funções de mesmo nome. Assim, o Laplc não consegue adicionar a segunda função na tabela de símbolos, e isso resulta no erro que é apropriadamente exibido.

#### 13.1.3. Erro sintático: chamada de função que não existe

```
1 function if: (Bool, Int, Int) Int = (condition, a, b) => match  
2 {  
3   condition==True: {a};  
4   default: {b};  
5 };  
6  
7 function intToBool: (Int) Bool = (n) => match  
8 {  
9   n==1: {True};
```

```

10     default: {False};
11 };
12
13 function soma: (Float, Float) Float = (x, y) => {x+y};
14
15 let result:Int=sum(2,5);

```

Saída:

```

1 laplc error: syntax error, function sum does not exist, on line 15

```

Era necessário definir a função sum, que não foi definida. Assim, o Laplc não consegue recuperar o identificador na tabela de símbolos, e isso resulta no erro que é apropriadamente exibido.

#### 13.1.4. Erro sintático: dupla declaração de variável ou constante

```

1 let letrinha:Char='F';
2 let letrinha:Char='F';

```

Saída:

```

1 laplc error: syntax error, variable or constant letrinha is double
  declared on line 2

```

Houve duas declarações com variáveis de mesmo nome. Assim, o Laplc não consegue adicionar a segunda variável na tabela de símbolos, e isso resulta no erro que é apropriadamente exibido.

#### 13.1.5. Erro sintático: uso de variável ou constante não existente

```

1 let orangotango: Int=500;
2 let animais: Int=2+orangotango;

```

Saída:

```

1 laplc error: syntax error, variable or const orangotango does not exist
  , on line 2

```

Era necessário definir a variável orangotango, que não foi definida. Assim, o Laplc não consegue recuperar o identificador na tabela de símbolos, e isso resulta no erro que é apropriadamente exibido.

#### 13.1.6. Erro semântico: número de parâmetros reais e tipos incorreto

```

1 const a: Bool = True;
2 const b: Bool = True;
3
4 let and: Bool = a & b;
5 let or: Bool = a | b;
6 let xor: Bool = a ^ b;

```

```

7
8 function soma: (Float, Float) Float = (z, x, y) => {x+y};

```

Saída:

```

1 laplc error: semantic error, function soma has 2 parameter types, but 3
  parameter identifiers were given on line 8

```

Era necessário que o número de parâmetros formais e de tipos de parâmetros fosse igual, o que não aconteceu. Assim, o erro é apropriadamente exibido pelo Laplc.

### 13.1.7. Erro semântico: expressão com tipos incorretos

```

1 const x: Float = 5.1;
2 const y: Float = 4.2;
3 const z: Float = 2.1;
4
5 let result:Float=4+x;

```

Saída:

```

1 laplc error: semantic error, expression inviable. Cannot operate left
  operand of type Int with right operand of type Float on line 5

```

É impossível somar a constante 4 com a variável x. Assim, o erro é apropriadamente exibido pelo Laplc.

### 13.1.8. Erro semântico: chamada de função com número de argumentos incorreto

```

1 function parCheck: (Float) Bool = (n) => match
2 {
3     n==0.0:{True};
4     default:{False};
5 };
6
7 function soma: (Float, Float) Float = (x, y) => {x+y};
8
9 let result:Int=soma(2.2,5.6,7.1);

```

Saída:

```

1 laplc error: semantic error, function soma argument number error on
  line 9

```

A função soma precisa de exatamente 2 argumentos, e 3 foram passados. Assim, o erro é apropriadamente exibido pelo Laplc.

### 13.1.9. Erro semântico: chamada de função com tipos de argumentos incorretos

```

1 let a: Char = 'w';
2 let b: Char = 'w';
3 let c: Char = 'w';

```

```

4 let d: Char = 'w';
5
6 function soma: (Float, Float) Float = (x, y) => {x+y};
7 let result:Int=soma(2.2,5);

```

Saída:

```

1 laplc error: semantic error, function soma has argument type errors on
  line 7

```

Aqui, a função soma precisa de 2 pontos flutuantes, e é passado um ponto flutuante e um inteiro. O erro é apropriadamente exibido pelo Laplc.

### 13.1.10. Erro semântico: tipo de retorno de função incorreto

```

1 function parCheck: (Float) Bool = (n) => match
2 {
3     n==0.0:{1};
4     default:{0};
5 };

```

Saída:

```

1 laplc error: semantic error, function parCheck is invalid. The expected
  return type is Bool, but the given were Int on line 5

```

Aqui, a função parCheck é definida como retornando booleanos, mas os retornos no seu corpo são inteiros. O erro é apropriadamente exibido pelo Laplc.

## 13.2. Códigos corretos

### 13.2.1. Operadores booleanos

```

1 const a: Bool = True;
2 const b: Bool = True;
3
4 let and: Bool = a & b;
5 let or: Bool = a | b;
6 let xor: Bool = a ^ b;

```

Saída:

```

1 _emain:
2 mov a, True;
3 mov b, True;
4 and _t0, a, b;
5 mov and, _t0;
6 or _t1, a, b;
7 mov or, _t1;
8 xor _t2, a, b;
9 mov xor, _t2;
10
11 laplc: well formed program
12
13 Global Symbol Table:

```

```

14 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
15 a - Const - Bool - []
16 b - Const - Bool - []
17 and - Let - Bool - []
18 or - Let - Bool - []
19 xor - Let - Bool - []

```

Acima é possível observar a saída gerada através do código de entrada. Também é possível observar a geração dos respectivos códigos de três endereços das operações de booleanos.

### 13.2.2. Divisão por zero

```

1 function div: (Float, Float) Float = (x, y) => {x/y};
2
3 let result: Float = div(5.0, 0.0);

```

Saída:

```

1 _ediv:
2 mov _fy, _a0;
3 mov _fx, _a1;
4 div _tf0, _fx, _fy;
5 mov _r, _tf0;
6 jr;
7
8 _emain:
9 mov _a0, 0.0;
10 mov _a1, 5.0;
11 jal _ediv;
12 mov _t0, _r;
13 mov result, _t0;
14
15 laplc: well formed program
16
17 Global Symbol Table:
18 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
19 div - Function - Float - [Float, Float]
20 result - Let - Float - []

```

Apesar de divisão por zero ser matematicamente impossível sem o uso de limite, o código acima não apresenta nenhum erro durante o processo de compilação. Isso acontece pois o erro aparecerá apenas no momento de execução.

### 13.2.3. Tripla chamada de função

```

1 function soma: (Float, Float) Float = (x, y) => {x+y};
2
3 let result: Float = soma(soma(2.0, 3.0), soma(5.0, 7.0));

```

Saída:

```

1 _esoma:
2 mov _fy, _a0;
3 mov _fx, _a1;
4 add _tf0, _fx, _fy;
5 mov _r, _tf0;
6 jr;
7
8 _emain:
9 mov _a0, 3.0;
10 mov _a1, 2.0;
11 jal _esoma;
12 mov _t0, _r;
13 mov _a0, 7.0;
14 mov _a1, 5.0;
15 jal _esoma;
16 mov _t1, _r;
17 mov _a0, _t1;
18 mov _a1, _t0;
19 jal _esoma;
20 mov _t2, _r;
21 mov result, _t2;
22
23 laplc: well formed program
24
25 Global Symbol Table:
26 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
27 soma - Function - Float - [Float, Float]
28 result - Let - Float - []

```

No exemplo acima a mesma função é invocada dentro dos parâmetros da chamada de função. No código de 3 endereços é possível observar esse comportamento através da instrução `jal _esoma` sendo utilizada 3 vezes também.

### 13.2.4. Operações de ponto flutuante

```

1 const x: Float = 5.1;
2 const y: Float = 4.2;
3 const z: Float = 2.1;
4
5 let result: Float = (x+y)*(z-x)/(z+y);
6
7 let greater: Float = x>y;
8 let lower: Float = x<y;
9 let equal: Float = x==y;
10 let notEqual: Float = x!=y;
11
12 let greaterOrEqual: Float = x>=y;
13 let lowerOrEqual: Float = x<=y;

```

Saída:

```

1 _emain:
2 mov x, 5.1;
3 mov y, 4.2;

```

```

4 mov z, 2.1;
5 add _t0, x, y;
6 sub _t1, z, x;
7 add _t2, z, y;
8 div _t3, _t1, _t2;
9 mul _t4, _t0, _t3;
10 mov result, _t4;
11 gt _t5, x, y;
12 mov greater, _t5;
13 lt _t6, x, y;
14 mov lower, _t6;
15 eq _t7, x, y;
16 mov equal, _t7;
17 ne _t8, x, y;
18 mov notEqual, _t8;
19 ge _t9, x, y;
20 mov greaterOrEqual, _t9;
21 le _t10, x, y;
22 mov lowerOrEqual, _t10;
23
24 laplc: well formed program
25
26 Global Symbol Table:
27 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
28 x - Const - Float - []
29 y - Const - Float - []
30 z - Const - Float - []
31 result - Let - Float - []
32 greater - Let - Float - []
33 lower - Let - Float - []
34 equal - Let - Float - []
35 notEqual - Let - Float - []
36 greaterOrEqual - Let - Float - []
37 lowerOrEqual - Let - Float - []

```

No exemplo acima é possível visualizar algumas operações com ponto flutuante. No código de três endereços é possível observar a respectiva representação das operações feitas no arquivo lapl.

### 13.2.5. Funções

```

1 function parCheck: (Float) Bool = (n) => match
2 {
3     n==0.0:{True};
4     default:{False};
5 };
6
7 function soma: (Float, Float) Float = (x, y) => {x+y};
8
9 let x: Bool = parCheck(soma(3.2+2.2, 2.2+5.4)%2.0);

```

Saída:

```

1 _eparCheck:

```

```

2 mov _fn, _a0;
3 eq _tf0, _fn, 0.0;
4 beq _tf0, 1, _e0_1;
5 j _e0_0;
6 _e0_0:
7 mov _r, False;
8 jr;
9 _e0_1:
10 mov _r, True;
11 jr;
12
13 _esoma:
14 mov _fy, _a0;
15 mov _fx, _a1;
16 add _tf0, _fx, _fy;
17 mov _r, _tf0;
18 jr;
19
20 _emain:
21 add _t0, 3.2, 2.2;
22 add _t1, 2.2, 5.4;
23 mov _a0, _t1;
24 mov _a1, _t0;
25 jal _esoma;
26 mov _t2, _r;
27 mod _t3, _t2, 2.0;
28 mov _a0, _t3;
29 jal _eparCheck;
30 mov _t4, _r;
31 mov x, _t4;
32
33 laplc: well formed program
34
35 Global Symbol Table:
36 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
37 parCheck - Function - Bool - [Float]
38 soma - Function - Float - [Float, Float]
39 x - Let - Bool - []

```

O código acima utiliza duas funções para validar se um número é par ou impar. No código de três endereços é possível visualizar a chamada dessas duas funções por meio da instrução jal.

### 13.2.6. Exemplo de uma operação condicional

```

1 function if: (Bool, Int, Int) Int = (condition, a, b) => match
2 {
3     condition==True: {a};
4     default: {b};
5 };
6
7 function intToBool: (Int) Bool = (n) => match
8 {
9     n==1: {True};

```



```

10     default: {False};
11 };
12
13 const number: Int = 55;
14 let isNumberEven: Int = number%2==0;
15
16 let isEven: Bool = intToBool(isNumberEven);

```

### Saída:

```

1 _eif:
2 mov _fb, _a0;
3 mov _fa, _a1;
4 mov _fcondition, _a2;
5 eq _tf0, _fcondition, True;
6 beq _tf0, 1, _e0_1;
7 j _e0_0;
8 _e0_0:
9 mov _r, _fb;
10 jr;
11 _e0_1:
12 mov _r, _fa;
13 jr;
14
15 _eintToBool:
16 mov _fn, _a0;
17 eq _tf0, _fn, 1;
18 beq _tf0, 1, _e1_1;
19 j _e1_0;
20 _e1_0:
21 mov _r, False;
22 jr;
23 _e1_1:
24 mov _r, True;
25 jr;
26
27 _emain:
28 mov number, 55;
29 eq _t0, 2, 0;
30 mod _t1, number, _t0;
31 mov isNumberEven, _t1;
32 mov _a0, isNumberEven;
33 jal _eintToBool;
34 mov _t2, _r;
35 mov isEven, _t2;
36
37 laplc: well formed program
38
39 Global Symbol Table:
40 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
41 if - Function - Int - [Int, Int, Bool]
42 intToBool - Function - Bool - [Int]
43 number - Const - Int - []
44 isNumberEven - Let - Int - []
45 isEven - Let - Bool - []

```

Como a laplace é uma linguagem funcional não existem comandos condicionais explícitos. Dessa forma é preciso utilizar recursos das próprias funções para tal. Um recurso excelente para isso é o match como demonstrado no exemplo acima.

### 13.2.7. Teste de maior idade

```
1 function legalAge: (Int) Bool = (n) => match
2 {
3     n>=18:{True};
4     default:{False};
5 };
6
7 let result: Bool = legalAge(18);
```

Saída:

```
1 _elegalAge:
2 mov _fn, _a0;
3 ge _tf0, _fn, 18;
4 beq _tf0, 1, _e0_1;
5 j _e0_0;
6 _e0_0:
7 mov _r, False;
8 jr;
9 _e0_1:
10 mov _r, True;
11 jr;
12
13 _emain:
14 mov _a0, 18;
15 jal _elegalAge;
16 mov _t0, _r;
17 mov result, _t0;
18
19 laplc: well formed program
20
21 Global Symbol Table:
22 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
23 legalAge - Function - Bool - [Int]
24 result - Let - Bool - []
```

Um exemplo de uso de expressão condicional funcional para determinar se a idade é maior ou não que 18 anos.

### 13.2.8. Verificar se um número é múltiplo de outro

```
1 function floatToBool: (Float) Bool = (n) => match
2 {
3     n==0.0:{True};
4     default:{False};
5 };
6
7 function mod: (Float, Float) Float = (x, y) => {x%y};
```

```

8
9 let result: Bool = floatToBool(mod(9.0, 3.0));

```

Saída:

```

1 _efloatToBool:
2 mov _fn, _a0;
3 eq _tf0, _fn, 0.0;
4 beq _tf0, 1, _e0_1;
5 j _e0_0;
6 _e0_0:
7 mov _r, False;
8 jr;
9 _e0_1:
10 mov _r, True;
11 jr;
12
13 _emod:
14 mov _fy, _a0;
15 mov _fx, _a1;
16 mod _tf0, _fx, _fy;
17 mov _r, _tf0;
18 jr;
19
20 _emain:
21 mov _a0, 3.0;
22 mov _a1, 9.0;
23 jal _emod;
24 mov _t0, _r;
25 mov _a0, _t0;
26 jal _efloatToBool;
27 mov _t1, _r;
28 mov result, _t1;
29
30 laplc: well formed program
31
32 Global Symbol Table:
33 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
34 floatToBool - Function - Bool - [Float]
35 mod - Function - Float - [Float, Float]
36 result - Let - Bool - []

```

O código acima descobre se um número é múltiplo de outro. Para isso ele utiliza uma função para obter o resto da divisão em outra para verificar se o resto é igual a zero. Se for igual a zero, retorna True, senão retorna False.

### 13.2.9. Usando a mesma função nos parâmetros

```

1 function div: (Float, Float) Float = (x, y) => {x/y};
2
3 let result: Float = div(div(div(2.0,3.0),div(5.0,7.0)),div(div(2.0,8.0),div(9.0,7.0)));

```

Saída:

```

1 _ediv:
2 mov _fy, _a0;
3 mov _fx, _a1;
4 div _tf0, _fx, _fy;
5 mov _r, _tf0;
6 jr;
7
8 _emain:
9 mov _a0, 3.0;
10 mov _a1, 2.0;
11 jal _ediv;
12 mov _t0, _r;
13 mov _a0, 7.0;
14 mov _a1, 5.0;
15 jal _ediv;
16 mov _t1, _r;
17 mov _a0, _t1;
18 mov _a1, _t0;
19 jal _ediv;
20 mov _t2, _r;
21 mov _a0, 8.0;
22 mov _a1, 2.0;
23 jal _ediv;
24 mov _t3, _r;
25 mov _a0, 7.0;
26 mov _a1, 9.0;
27 jal _ediv;
28 mov _t4, _r;
29 mov _a0, _t4;
30 mov _a1, _t3;
31 jal _ediv;
32 mov _t5, _r;
33 mov _a0, _t5;
34 mov _a1, _t2;
35 jal _ediv;
36 mov _t6, _r;
37 mov result, _t6;
38
39 laplc: well formed program
40
41 Global Symbol Table:
42 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
43 div - Function - Float - [Float, Float]
44 result - Let - Float - []

```

O exemplo mostra várias chamadas da mesma função em seus parâmetros. Como é possível observar no código de três endereços, a chamada foi feita o mesmo número de vezes que no código.

### 13.2.10. Função que soma, subtrai, divide ou soma

```

1 function sumMinusDivMult: (Int, Float, Float) Float = (n, op1, op2) =>
  match
2 {

```

```

3     n==0:{op1+op2};
4     n==1:{op1-op2};
5     n==2:{op1/op2};
6     n==3:{op1*op2};
7     default:{op1+op2-op1/op2*op1};
8 };
9
10 let sum: Float = sumMinusDivMult(0, 5.0, 10.0);
11 let minus: Float = sumMinusDivMult(1, 10.0, 12.0);
12 let div: Float = sumMinusDivMult(2, 8.0, 5.0);
13 let mult: Float = sumMinusDivMult(3, 10.0, 2.0);
14 let result: Float = sumMinusDivMult(4, 10.0, 2.0);

```

Saída:

```

1 _esumMinusDivMult:
2 mov _fop2, _a0;
3 mov _fop1, _a1;
4 mov _fn, _a2;
5 eq _tf11, _fn, 0;
6 beq _tf11, 1, _e0_4;
7 eq _tf10, _fn, 1;
8 beq _tf10, 1, _e0_3;
9 eq _tf9, _fn, 2;
10 beq _tf9, 1, _e0_2;
11 eq _tf8, _fn, 3;
12 beq _tf8, 1, _e0_1;
13 j _e0_0;
14 _e0_0:
15 add _tf0, _fop1, _fop2;
16 sub _tf1, _fop1, _fop2;
17 div _tf2, _fop1, _fop2;
18 mul _tf3, _fop1, _fop2;
19 mul _tf4, _fop2, _fop1;
20 div _tf5, _fop1, _tf4;
21 sub _tf6, _fop2, _tf5;
22 add _tf7, _fop1, _tf6;
23 mov _r, _tf7;
24 jr;
25 _e0_1:
26 mov _r, _tf3;
27 jr;
28 _e0_2:
29 mov _r, _tf2;
30 jr;
31 _e0_3:
32 mov _r, _tf1;
33 jr;
34 _e0_4:
35 mov _r, _tf0;
36 jr;
37
38 _emain:
39 mov _a0, 10.0;
40 mov _a1, 5.0;
41 mov _a2, 0;

```

```

42 jal _esumMinusDivMult;
43 mov _t0, _r;
44 mov sum, _t0;
45 mov _a0, 12.0;
46 mov _a1, 10.0;
47 mov _a2, 1;
48 jal _esumMinusDivMult;
49 mov _t1, _r;
50 mov minus, _t1;
51 mov _a0, 5.0;
52 mov _a1, 8.0;
53 mov _a2, 2;
54 jal _esumMinusDivMult;
55 mov _t2, _r;
56 mov div, _t2;
57 mov _a0, 2.0;
58 mov _a1, 10.0;
59 mov _a2, 3;
60 jal _esumMinusDivMult;
61 mov _t3, _r;
62 mov mult, _t3;
63 mov _a0, 2.0;
64 mov _a1, 10.0;
65 mov _a2, 4;
66 jal _esumMinusDivMult;
67 mov _t4, _r;
68 mov result, _t4;
69
70 laplc: well formed program
71
72 Global Symbol Table:
73 <Identifier>, <Definition>, <ReturnType>, [<ArgumentTypes>*]
74 sumMinusDivMult - Function - Float - [Float, Float, Int]
75 sum - Let - Float - []
76 minus - Let - Float - []
77 div - Let - Float - []
78 mult - Let - Float - []
79 result - Let - Float - []

```

A função acima executa uma operação de soma, subtração, divisão ou multiplicação dependendo do valor do primeiro parâmetros.

## 14. Considerações finais: compiladores são mesmo dragões?

Aho é claro na capa de seu livro [Aho et al. 2013]. Vencedor do Turing Award de 2020, ele descreve a complexidade no design de compiladores como um dragão e a tradução dirigida por sintaxe como um bravo guerreiro que parece estar disposto a vencê-lo num árduo combate. Durante as entregas deste trabalho, que julgamos ser o mais difícil de todo o bacharelado até então, travamos essa batalha, e acreditamos tê-la vencido. Embora esperemos uma boa nota, é claro, ficamos muito satisfeitos e orgulhosos do que fizemos. No começo parecia ainda mais difícil, mas graças às ajudas do monitor (em especial), do professor e dos livros-texto, essa dificuldade foi cedendo a nossa força de vontade.

A primeira entrega foi a em que estávamos mais perdidos. À medida do tempo fomos nos encontrando, e nessa entrega já sabíamos exatamente o que fazer e nossas ex-

pectativas foram confirmadas pelo email do professor. Curiosamente, a análise semântica, que é a parte mais difícil do design de um compilador funcional (como verificado por nós no projeto do GHC, já citado compilador de Haskell) foi uma das partes mais simples, bastando a implementação da tabela de símbolos e o seu uso. Acreditamos que essa facilidade veio devido a nossa maior experiência na disciplina e às já citadas ajudas. O código de três endereços então foi bastante trivial. Com apenas algumas variáveis globais e buffers para armazenar informações que precisaríamos para construí-lo, tudo fluiu como água.

Além disso, acreditamos que nosso olhar para bons (e maus) compiladores nunca mais será o mesmo. Notando a dificuldade de criação de um compilador simplificado para estudos como o nosso, a dificuldade de um compilador excelente (como o próprio GCC ou Clang, e principalmente os funcionais como o GHC ou o GRIN) nos parece mais palpável e assustadora, e isso explica porque criá-los é uma tarefa de anos feita por milhares de programadores.

Ademais, pensamos ser bastante viável permitir o uso da linguagem C++ em um possível trabalho como esse para outras turmas. De fato, acreditamos que teria facilitado partes do trabalho que não são o foco e nos liberaria tempo para pensar mais ainda em design de compiladores. Por exemplo, caso C++ fosse permitido, poderíamos ter feito a tabela de símbolos utilizando o HashMap (especificamente o `unordered_map`) e a String para criação da tabela de símbolos, o que permitiria uma quantidade virtualmente infinita de identificadores a ser adicionados, e os nomes dos identificadores como também virtualmente infinitos. Assim, talvez sobraria tempo para fazermos, por exemplo, a API Fluente (e consequente possibilidade de incluir um arquivo no outro da linguagem Laplace).

Por outro lado de toda essa dificuldade dos compiladores, a especificação da linguagem Laplace foi especialmente fácil. Isso se deu graças a disciplina anterior que fizemos de Linguagens de Programação (que, até onde sabemos, estranhamente não é pré-requisito para esta de Compiladores). Na ocasião da disciplina, éramos o exato trio deste trabalho e escolhemos a linguagem Haskell para estudar a fundo em um trabalho prático que também durou o semestre todo. Assim, sem saber, plantamos as sementes que floresceram agora regadas a suor.

Como não há outra entrega, vamos animados para seja lá o que estiver reservado ou não para nós. Como é a última disciplina com o professor Daniel, gostaríamos de também o agradecer-lo por tudo.

## 15. Agradecimentos especiais

Gostaríamos de deixar registrado neste documento todo o nosso agradecimento e apreço pelo monitor da disciplina Daniel Freitas Martins, que nos ajudou em todas as entregas deste trabalho. Monitores como ele fazem da universidade um lugar mais agradável.

## Referências

- [Aho et al. 2013] Aho, A., Sethi, R., Lam, M., and Ullman, J. (2013). *Compilers: Pearson New International Edition PDF eBook: Principles, Techniques, and Tools*. Pearson Education.
- [Brady 2017] Brady, E. (2017). *Type-Driven Development with Idris*. Manning.

- [Cherny 2019] Cherny, B. (2019). *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media.
- [Hsu 2021] Hsu, M. (2021). *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries: Design powerful and reliable compilers using the latest libraries and tools from LLVM*. Packt Publishing.
- [O'Sullivan et al. 2008] O'Sullivan, B., Goerzen, J., and Stewart, D. (2008). *Real World Haskell: Code You Can Believe In*. O'Reilly Media.
- [Patterson and Hennessy 2020] Patterson, D. and Hennessy, J. (2020). *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science.