Trabalho Prático 3 - A linguagem Laplace

Daniel Fernandes Pinho - 2634[1], Leandro Lázaro Araújo Vieira - 3513[2], Mateus Pinto da Silva - 3489[3], Taianne Valerie Motta - 2679[4]

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-CAF) – Florestal – MG – Brasil

(daniel.f.pinho[1], leandro.lazaro[2], mateus.p.silva[3], taianne.mota[4])@ufv.br

Resumo. *Compiladores são, acima de tudo, implementações de linguagens formais. Neste trabalho, apresentamos o estado atual do nosso compilador para a linguagem funcional estrita Laplace. Utilizamos de regras léxicas simples via Flex, e regras sintáticas simples via Bison. Focamos principalmente no conserto da gramática da linguagem. Codificamos também uma simples tabela de símbolos, que será melhorada na próxima entrega. Como resultado, conseguimos implementar a nossa própria linguagem sem qualquer erro na gramática e criar vários exemplos de teste que validam nosso trabalho, tanto dos ajustes no analisador léxico quanto na criação do analisador sintático. Vamos ansiosos e animados para a análise semântica e geração de código.*

1. Mudanças no relatório

Neste relatório, foram feitas algumas modificações em relação à entrega do anterior. Foi adicionado a seção *Como executar*, visto sua importância para o leitor manipular todas as execuções possíveis do projeto. Na nova seção *regras sintáticas* é incluído a nova gramática para a linguagem e a respectiva justificativa. E, por fim, na nova seção *Tabela de símbolos*, é descrito o conteúdo da mesma, elaborada para esta etapa do projeto do Compilador Laplace e que servirá para a última entrega do mesmo. Quanto as seção das regras léxicas, ela também foi modificada na medida do necessário, contendo novos Tokens gerados, valor semântico, etc.

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:

```
sudo apt update && sudo apt upgrade
sudo apt install build-essentials
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:

```
make
```

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

```
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 0, porém poderia ser qualquer valor entre 0 e 10):

```
make run0
```

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. Comandos disponíveis e o seu funcionamento

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

4.4. 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.

4.5. Palavras-chave e palavras reservadas

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

5. Regras léxicas

5.1. 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)

5.2. Os Tokens e os valores semânticos gerados

Mantivemos todos os Tokens da entrega anterior e adicionamos mais alguns que estavam faltando. Além disso, optamos por padronizar o valor semântico de todos os Tokens que o necessitavam como String. Assim, o analisador sintático é responsável por fazer a conversão de tipos. Por exemplo: Caso o programador digite na Laplace o valor inteiro 5, ele será passado como uma String para o Bison, e apenas lá será corretamente convertido para inteiro.

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	
type_float	TYPE_FLOAT	
type_char	TYPE_CHAR	
type_bool	TYPE_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
value_float	VALUE_FLOAT	String do lexema
value_bool	VALUE_BOOL	String do lexema
value_char	VALUE_CHAR	String do lexema
value_string	VALUE_STRING	String do lexema
lowercase_identifier	LOWERCASE_IDENTIFIER	String do lexema
uppercase_identifier	UPPERCASE_IDENTIFIER	String do lexema

6. Regras sintáticas

Optamos por reescrever a gramática do zero, visto que os vários erros de shift/reduce e reduce/reduce existiam nela. Entretanto, miramos na mesma definição da nossa linguagem de programação. Acreditamos que nossa estratégia foi bem pensada, visto que o resultado final foi uma gramática sintática sem nenhum aviso de atenção (*warning*) no software *Bison*, e capaz de reconhecer sem problemas todos os exemplos apresentados.

Algumas observações precisam ser feitas, que talvez não sejam tão familiares ao leitor. O termo **fluent_api**, no bom português API fluente, significa chamadas de funções ou de constantes que estão dentro de módulos através do operador ponto final, algo que lembra um pouco orientação a objetos porém é um conceito de programação funcional. Outros é de o de **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*.

```
start:
    | stmts start
;

stmts: variable_definition
    | const_definition
    | function_declaration
```

```
    | data_declariation
    | function_import
;

vec_type: OPEN_BRACKET type_specifier SEMICOLON VALUE_INT CLOSE_BRACKET
;

type_specifier: TYPE_INT
    | TYPE_FLOAT
     | TYPE_CHAR
    | TYPE_BOOL
    | TYPE_STRUCT
    | vec_type
    | UPPERCASE_IDENTIFIER
;

operator: NOT
    | PLUS
    | MINUS
    | DIVISION
    | MULTIPLICATION
    | MOD
    | EQUAL
    | NOT_EQUAL
    | GREATER
    | LESS
    | GREATER_OR_EQUAL
    | LESS_OR_EQUAL
    | AND
    | OR
;

aux_value_vec: value
    | value COMMA aux_value_vec
;

value_vec: OPEN_BRACKET aux_value_vec CLOSE_BRACKET
;

aux_struct_value: LOWERCASE_IDENTIFIER COLON value
    | LOWERCASE_IDENTIFIER COLON value COMMA aux_struct_value
;

struct_value: OPEN_BRACE aux_struct_value CLOSE_BRACE
;

value: VALUE_INT
    | VALUE_BOOL
    | VALUE_CHAR
    | VALUE_FLOAT
    | VALUE_STRING
    | value_vec
    | struct_value
;

parameter: LOWERCASE_IDENTIFIER
```

```
    | value
    | fluent_api
    | function_call
;

expression: parameter
    | parameter operator expression
    | OPEN_PARENTHESIS expression CLOSE_PARENTHESIS
    | OPEN_PARENTHESIS expression CLOSE_PARENTHESIS operator expression
;

variable_definition: LET LOWERCASE_IDENTIFIER COLON type_specifier
        DEFINITION expression SEMICOLON
;

const_definition: CONST LOWERCASE_IDENTIFIER COLON type_specifier
        DEFINITION expression SEMICOLON
;

guards: parameter operator parameter COLON OPEN_BRACE parameter
    CLOSE_BRACE SEMICOLON
    | parameter operator parameter COLON OPEN_BRACE parameter
    CLOSE_BRACE SEMICOLON guards
    | DEFAULT COLON OPEN_BRACE parameter CLOSE_BRACE SEMICOLON
;

function_definition: OPEN_PARENTHESIS function_definition_parameters
    CLOSE_PARENTHESIS ARROW MATCH OPEN_BRACE guards CLOSE_BRACE
    | OPEN_PARENTHESIS function_definition_parameters CLOSE_PARENTHESIS
     ARROW OPEN_BRACE expression CLOSE_BRACE
;

function_types: type_specifier
    | type_specifier COMMA function_types
;

function_definition_parameters: LOWERCASE_IDENTIFIER
    | LOWERCASE_IDENTIFIER COMMA function_definition_parameters
;

function_call_parameters: parameter
    | parameter COMMA function_call_parameters
;

function_declaration: FUNCTION LOWERCASE_IDENTIFIER COLON
    OPEN_PARENTHESIS function_types CLOSE_PARENTHESIS
        type_specifier DEFINITION function_definition SEMICOLON
;

fluent_api: PERIOD LOWERCASE_IDENTIFIER
    | LOWERCASE_IDENTIFIER fluent_api
    | UPPERCASE_IDENTIFIER fluent_api
;

aux_function_call: fluent_api OPEN_PARENTHESIS function_call_parameters
    CLOSE_PARENTHESIS
```

```
    | LOWERCASE_IDENTIFIER OPEN_PARENTHESIS function_call_parameters
    CLOSE_PARENTHESIS
;

function_call: aux_function_call
;

aux_struct_definition: LOWERCASE_IDENTIFIER COLON type_specifier
    | LOWERCASE_IDENTIFIER COLON type_specifier COMMA
    aux_struct_definition
;

struct_definition: OPEN_BRACE aux_struct_definition CLOSE_BRACE
;

data_declariation: DATA UPPERCASE_IDENTIFIER COLON TYPE_STRUCT
    DEFINITION struct_definition SEMICOLON
;

function_package: LOWERCASE_IDENTIFIER
    | LOWERCASE_IDENTIFIER COMMA function_package
;

function_import: FROM UPPERCASE_IDENTIFIER IMPORT function_package
;
```

7. Tabela de símbolos

Como o monitor da disciplina nos orientou que não era necessário, para esta entrega, a questão de visibilidade de variáveis entre escopos, optamos por fazer a tabela de símbolos da forma mais simples possível. Assim, complementá-la será nosso próximo passo para a construção do compilador. Da forma que está, ela funciona como uma lista de Strings para armazenagem dos identificadores declarados ou importados pelo programador.

```
#define SYMBOL_TABLE_SIZE 1024
#define IDENTIFIER_ALLOC_SIZE 64

typedef struct
{
  char identifier[IDENTIFIER_ALLOC_SIZE];
} SymbolTableEntry;

typedef struct
{
  SymbolTableEntry table[SYMBOL_TABLE_SIZE];
  size_t next_empty_slot;
} SymbolTable;
```

Quanto a implementação em si, ela foi feita de forma muito simplificada. Optamos por fazer da forma mais estática o possível, sem utilização manual de ponteiros e, por conseguinte, sem as temíveis falhas de desreferenciação de ponteiros nulos e, por conseguinte, as falhas de segmentação.

8. Código produzido e testes executados

Mantivemos todos os seis exemplos de teste feitos para o analisador léxico, e optamos por criar mais quatro específicos testar o analisador sintático. Aqui mostramos os códigos em si, o resultado e a explicação.

8.1. Hello, World!

```
from Iostream import printLn

let main: Int = printLn("Hello, World!");
let main: Int = printLn("Hello, World!");
```

Saída:

```
laplc: well formed program

Symbol Table: <identifier>
printLn
main
```

O programa está léxico e sintaticamente correto.

8.2. Soma simples de dois termos

```
from Iostream import printLn

function soma: (Int, Int) Int = (x, y) => {x+y};
let x: Int = soma(3, 2);

let main: Int = printLn(x);
```

Saída:

```
laplc: well formed program

Symbol Table: <identifier>
printLn
y
x
soma
main
```

O programa está léxico e sintaticamente correto.

8.3. Cálculo do quinto elemento da sequencia de Fibonnaci

```
from Math import fibonnaci
from Iostream import printLn

const posicao: Int = 5;

let result: Int = fibonnaci(posicao);
let main: Int = printLn(result);
```

Saída:

```
laplc: well formed program

Symbol Table: <identifier>
fibonnaci
printLn
posicao
result
main
```

O programa está léxico e sintaticamente correto.

8.4. Mapeamento para calculo de idade

```
from Functools import map
from Iostream import printLn

let anosDeNascimento: [Int; 4] = [1960, 1985, 2002, 1999];
let idades: [Int; 4] = anosDeNascimento.map(printLn);

let main: Int = printLn(idades);
```

Saída:

```
laplc: well formed program

Symbol Table: <identifier>
map
printLn
anosDeNascimento
idades
main
```

O programa está léxico e sintaticamente correto.

8.5. Criando tipo Aluno

```
from Iostream import printLn

data Aluno: Struct = {nome: [Char; 120], matricula: Int};

let aluno: Aluno = {nome: "Mateus Pinto da Silva", matricula: 3489};

let main: Int = printLn(aluno.nome);
```

Saída:

```
laplc: well formed program

Symbol Table: <identifier>
printLn
matricula
nome
Aluno
aluno
main
```

O programa está léxico e sintaticamente correto.

8.6. Hello, World sem aspas duplas

```
from Iostream import printLn

let main: Int = printLn("Hello, World!);
```

Saída:

```
laplc: lexical error on line 5
make: *** [makefile:23: run5] Error 1
```

O programa está lexicamente incorreto, pois não há fechamento das aspas duplas, então nenhum Token pode ser gerado.

8.7. Usando printLn sem importar a função

```
let main: Int = printLn("Hello, World!");
let main: Int = printLn("Hello, World!");
```

Saída:

```
laplc error: Identifier printLn on line 3 does not exist
make: *** [makefile:26: run6] Error 1
```

O programa está sintaticamente incorreto, pois não há importação da função printLn. O erro é detectado através da ausência desse identificador na tabela de símbolos.

8.8. Somando um termo inexistente

```
from Iostream import printLn

function soma: (Int, Int) Int = (x, y) => {x+z};
let x: Int = soma(3, 2);

let main: Int = printLn(x);
```

Saída:

```
laplc error: Identifier z on line 5 does not exist
make: *** [makefile:29: run7] Error 1
```

O programa está sintaticamente incorreto, pois não há a definição da variável **z** na função soma. Assim, o erro é detectado através da ausência desse identificador na tabela de símbolos.

8.9. Uma virgula a mais

```
from Functools import map
from Iostream import printLn

let anosDeNascimento: [Int; 4] = [1960, 1985, 2002, 1999,];
let idades: [Int; 4] = anosDeNascimento.map(printLn);

let main: Int = printLn(idades);
```

Saída:

```
laplc error: syntax error on line 6
make: *** [makefile:32: run8] Error 1
```

O programa está sintaticamente incorreto, pois há uma vírgula a mais no final do vetor **anosDeNascimento**. Assim, o erro é detectado através da ausência de regra sintática correspondente.

8.10. Invocando vetor e seu método sem instancia-lo

```
from Functools import map
from Iostream import printLn

let idades: [Int; 4] = anosDeNascimento.map(printLn);

let main: Int = printLn(idades);
```

Saída:

```
laplc error: Identifier anosDeNascimento on line 6 does not exist
make: *** [makefile:35: run9] Error 1
```

O programa está sintaticamente incorreto, pois não há a definição do vetor **anosDeNascimento** na função soma. Assim, o erro é detectado através da ausência desse identificador na tabela de símbolos.

8.11. Variável inexistente

```
from Math import fibonnaci
from Iostream import printLn

let result: Int = fibonnaci(posicao);
let main: Int = printLn(result);
```

Saída:

```
laplc error: Identifier posicao on line 6 does not exist
make: *** [makefile:38: run10] Error 1
```

O programa está sintaticamente incorreto, pois não há a definição do vetor **posicao** na função soma. Assim, o erro é detectado através da ausência desse identificador na tabela de símbolos.

9. Considerações finais

Com esse trabalho foi possível elaborar e implementar com sucesso o analisador sintático da linguagem Laplace. Para isso tivemos que fazer alguns pequenos ajustes em nosso analisador léxico, como adicionar o retorno dos tokens. Ademais, criamos uma implementação simples de tabela de símbolos. Portanto, estamos um passo mais perto de conseguirmos implementar o nosso compilador e se tudo correr como esperado, já o teremos até a próxima entrega.

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.

[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.