UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

IMPLEMENTAÇÃO DE COMPILADOR

Marcos Silva Laydner
Nathan Sargon Werlich
Higor Nocetti
Sadi Júnior Domingos Jacinto

Professor orientador: Rafael de Santiago

Florianópolis
2020

Marcos Silva Laydner
Nathan Sargon Werlich
Higor Nocetti
Sadi Júnior Domingos Jacinto

IMPLEMENTAÇÃO DE COMPILADOR

Trabalho prático da disciplina INE5622 – Introdução a Compiladores, consistindo na implementação de um compilador (analisador léxico e sintático), com o uso da ferramenta ANTLR4, necessário para obtenção de nota.

Professor orientador: Rafael de Santiago

Florianópolis
2020

Sumário

1 CONTRIBUIÇÃO DOS MEMBROS

Versão	Participante	Contribuição
1.0	Marcos Silva Laydner	Tipos obrigatórios Operações obrigatórias
	Nathan Sargon Werlich	Laços *for* e *while* Estruturas de controle *if-then-else* e *switch-case*
	Higor Nocetti	Características adicionais da linguagem Códigos de teste
	Sadi Júnior Domingos Jacinto	Elaboração do Relatório Definição de funções
2.0	Marcos Silva Laydner	Operações relacionadas aos cálculos básicos (adição, subtração, divisão e multiplicação) e operação de *print*
	Nathan Sargon Werlich	Não participou
	Higor Nocetti	Não participou
	Sadi Júnior Domingos Jacinto	Elaboração do Relatório, implementação das estruturas *while*, *for*, *switch-case*, *if-then-else*, operações lógicas, definição e chamada de funções, reporte de erros semânticos, arquivos de teste e declaração e alocação de variáveis

Importante frisar que, na primeira entrega, cada participante, além de suas respectivas contribuições, também ajudaram a testar a linguagem, além de também auxiliarem os demais colegas no desenvolvimento das outras partes do trabalho.

Já na segunda entrega, apenas dois dos participantes ajudaram a testar a linguagem.

2 DESCRIÇÃO DA LINGUAGEM

A linguagem criada chama-se **sadbeep**. A mesma consiste em uma linguagem que não se utiliza de tipagem, além de apresentar a possibilidade de definições de variáveis, e um código inteiro propriamente dito, fora de funções. Além disso, a criação de variáveis não é obrigatória, visto que um código como o do exemplo abaixo é válido:

```
1 1 + 1;
2 "Teste de string não atribuída a nenhuma variável.";
3 true;
```

2.1 REQUISITOS OBRIGATÓRIOS DA LINGUAGEM

2.1.1 TIPOS

- **INT:**
  Para o tipo inteiro, foi considerado qualquer quantidade, maior ou igual à 1, de dígitos de 0 à 9, inicialmente sem limite de quantidade de dígitos.

- **FLOAT:**
  Para o tipo de ponto flutuante, foram considerados dois grupos de dígitos, de qualquer quantidade, maior ou igual à 1 estando no intervalo de 0 à 9, concatenados por um ponto. Assim como o tipo inteiro, a princípio, não foi definido nenhuma limitação da quantidade máxima de dígitos.

  Visando facilitar a leitura da gramática, esses dois tipos (INT e FLOAT) foram representados por um mesmo *token*, chamado **NUMBER**.

```
1 NUMBER: INT | FLOAT;
2 INT: [0-9]+;
3 FLOAT: [0-9]+.[0-9]+;
```

- **BOOL:**
  Um dos tipos adicionais da linguagem, definido devido a utilidade do tipo *booleano* em operações lógicas e condicionais utilizadas por laços como *for* e *while*, e por estruturas condicionais como o *if-then-else*.

```
1 TRUE: 'true';
2 FALSE: 'false';
3 BOOL: TRUE | FALSE;
```

2.1.2 OPERADORES

Os operadores implementados foram baseados no código de exemplo disponibilizado no *moodle*, sendo divididos em três categorias:

1. **exp:** Operações lógicas.

```
1 exp: left=summ (op=('>' | '<' | '>=' | '<=' | '==' |
    '!=') right=exp)*;
```

2. **mult:** Operações de multiplicação, divisão e módulo.

```
1 mult: left=atom (op=('*' | '/' | '\%') right=mult)*;
```

3. **summ:** Operações de adição e subtração.

```
1 summ: left=mult (op=('+' | '-') right=summ)*;
```

Importante observar que a ordem de precedência dos operadores é definida pelo atributo "*left*", indicando quais operadores tem precedência sobre quais outros. Assim, tem-se *mult* com a maior precedência, seguida de *summ*[1], e tendo *exp* como o de menor precedência[2].

2.1.3 FUNÇÕES

A definição de uma função se dá com o prefixo *func* seguido do nome da função, uma lista de argumentos dentro de dois parênteses, sendo possível tal lista estar vazia, e um conjunto de colchetes, dentro dos quais se encontra, obrigatóriamente, o corpo da função. A sintaxe simplicada é:

func nome(args) { body; }

E sua gramática é:

```
1 function_def : 'func' name=ID '(' args? ')' block;
2 args : ID (',' ID)*;
3 block: '{' expr* '}';
```

Sendo que a chamada de um função pode ocorrer dentro ou fora de uma função, além de permitir o uso de recursão. Para chamar uma função, basta invocar o nome da mesma, passando para ela uma lista de parâmetros, que podem ser variáveis, valores ou outras funções, A sintaxe é:

call_func(args);

E a gramática:

```
1 call : name=ID '(' exprs? ')' ';';
```

2.1.4 *IF-THEN-ELSE* e *SWITCH CASE*

– ***IF-THEN-ELSE*:**
Foi seguido a estrutura clássica do *if-then-else*, baseando-se na sintaxe do *Java*. Inicia-se com o *if*, seguido de uma condicional, que pode estar ou não contida entre parênteses, depois se encontra a abertura do par de chaves, dentro do qual se encontra o bloco que será executado caso a condicional seja verdadeira.

Em seguida existe a possibilidade de se usar o else seguido por mais um bloco entre chaves, conforme o exemplo na seção 3.

[1] graças ao uso de *left=mult*
[2] *left=summ*

```
'if' cond=expr ('&&' cond=expr)* | ('||' cond=expr)*
   then-block ('else' otherwise=block)?;

block: '{' expr* '}';
```

– ***SWITCH*:**
Foi baseado na sintaxe do *Java*, porém, sem a inclusão do bloco *default* e sem a necessidade da palavra reservada *break*.

```
'switch' expr? '{' (cases)+ '}'

cases: 'case' expr ':' expr 'break;'?;
```

2.1.5 *FOR* e *WHILE*

– ***FOR*:**
Foi seguida a sintaxe padrão do Java, com capacidades de atribuição, seguida da condição de parada e posterior incremento.

```
'for' forexpr block;

forexpr: '(' variable '=' expr ';' cond=expr ';'
   variable '=' expr ')';
```

– ***WHILE*:**
Usada a sintaxe padrão do Java, onde existe apenas a condição de parada do laço:

```
'while' cond=expr ('&&' cond=expr)* | ('||' cond=expr)
   * block;
```

2.2 CARACTERÍSTICAS ADICIONAIS DA LINGUAGEM

Fora o operador adicional (módulo), comentado na seção de operadores, a presente linguagem apresenta, como características adicionais:

– Suporte à Comentários:
Sejam eles de linha única ou de múltiplas linhas, seguindo a sintaxe de comentários da linguagem *Java*.

```
//Comentário de linha única
/*Comentário longo
com mais de uma linha
*/
```

```
COMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;]
```

– && e || Lógicos:
Foi adicionado também a capacidade de adicionar Es (&&) e OUs (||) nas condicionais das estruturas *if-then-else* e *while*. Sendo que o uso desses operadores pode ocorrer com o uso de duas sintaxes:

```
1  if a > 3 && b == 0 {
2  ...
3  }
4
5  if (a > 3) && (b == 0) {
6  ...
7  }
8
9  while c < 1 || d == 1 {
10 ...
11 }
12
13 while (c < 1) || (d == 1) {
14 ...
15 }
```

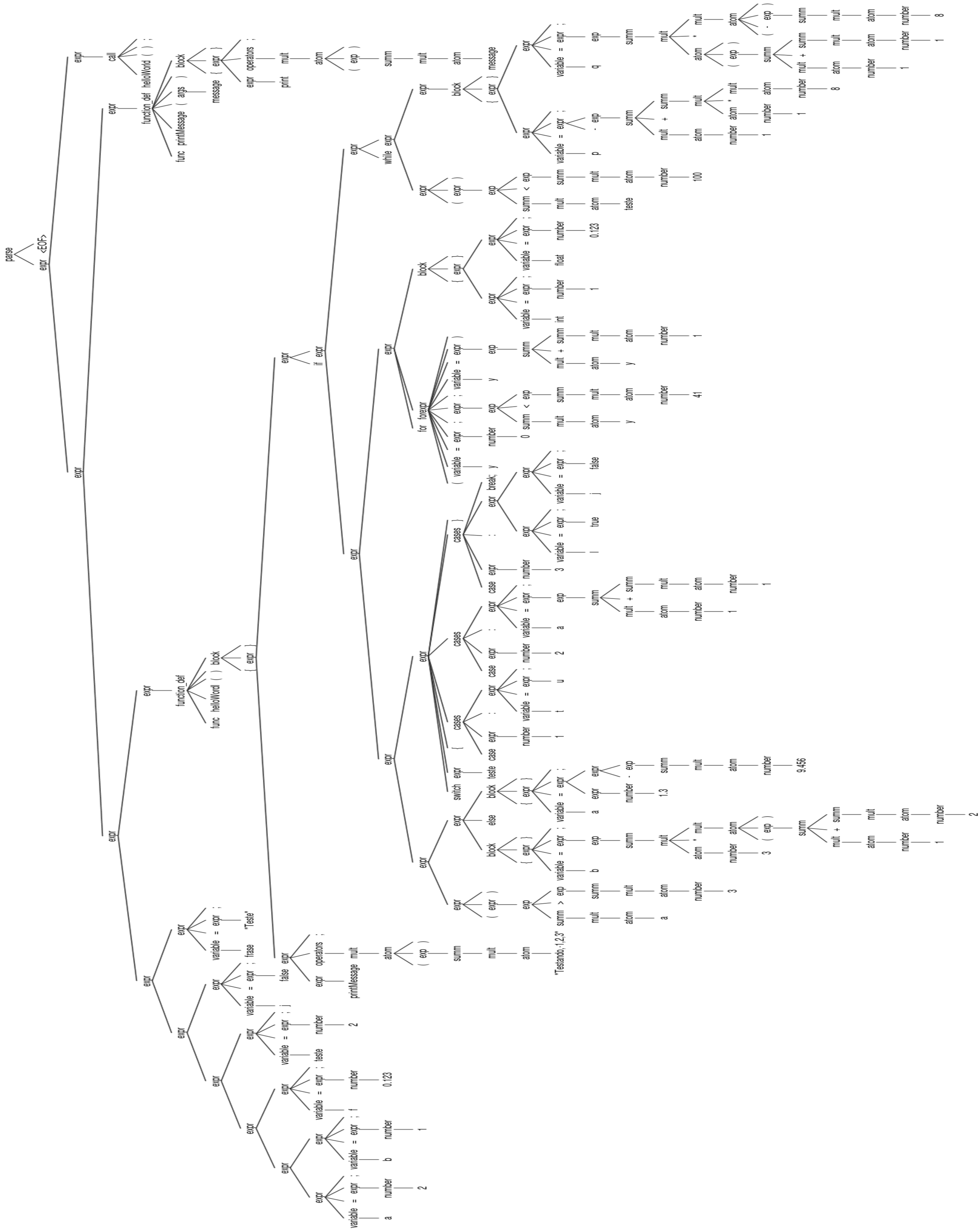
– **STRING:**
A definição de *string* usada nesse trabalho comporta tanto caractéres isolados quanto textos longos. Além disso, existem dois caractéres usados para se delimitar e definir uma *string*, a saber: ' e ".

```
1  STRING: '\''~[\r\n']* '\'' | '"'~[\r\n']*'"';
```

3 EXEMPLOS DE CÓDIGOS

```
1  //Tipos numéricos:
2  a = 2;
3  b = 1;
4  f = 0.123;
5  teste = 2;
6
7  //Boolean
8  j = false;
9
10 //String
11 frase = "Teste";
12
13 //Função
14 func helloWordl() {
15
16   //Chamar um função
17   printMessage("Testando, 1,2,3");
18
19   //If-then-else
20   if (a > 3) {
21     b = 3*(1+2);
22   } else {
23     a = 1.3 - 9.456;
24   }
25
26   //Switch
27   switch teste {
28     case 1:
29       t = u;
30     case 2:
31       a = 1 + 1;
32     case 3:
33       l = true;
34       j = false;
35       break;
36   }
37
38   //For
39   for(y = 0; y < 41; y = y + 1) {
40     int = 1;
41     float = 0.123;
42   }
43
44   //While
45   while (teste < 100) {
46     p = -1 + 1 * 8;
```

```
47      q = (1+1)*(-8);
48    }
49  }
50
51  func printMessage(message) {
52    print(message);
53  }
54
55  helloWorld();
```



4 MUDANÇAS NA LINGUAGEM ORIGINAL

As seguintes mudanças ocorreram na linguagem original:

- Regra *expr*:
  - Foram adicionados *labels* em todas as cláusulas;
  - Foi adicionada uma nova cláusula (*'print'expr ';'*);
  - Foi removida a cláusula ID, dessa forma, a declaração de uma variável sem valor definido deixa de ser possível.
  - Foi removida a cláusula *expr operators ';'*, uma vez que uma análise mais detalhada mostrou que tal cláusula era redundante.
  - O símbolo '-'foi renomeado para LESS.
  - As estruturas *if* e *while* foram refatoradas, uma vez que o símbolo | se encontrava fora dos parênteses, o que, para o ANTLR4, criava uma nova cláusula. [3]
- Regra *for*:
  - Basicamente a regra *for* foi dividida em duas novas regras, a saber: *init* e *finish*.
  - Para exemplificar, a antiga regra era assim:

    * 'for' forexpr block
      forexpr : '(' variable '=' expr ';' cond=expr ';' variable '=' expr ')';

    E agora a mesma é assim:

    * 'for' forexpr
      forexpr : '(' init ';' cond=expr ';' finish ')' block;

      init: variable '=' expr;
      finish: variable '=' expr;
- Regras *args* e *function_def*:
  Ambas as regras foram alteradas para poderem suportar tipagem sobre o retorno da função e os parâmetros recebidos.

  Antes era assim:

  - function_def : 'func' name=ID '(' args? ')' block;

    args : ID (',' ID)*;

  E agora a mesma é assim:

[3] Exemplificando:
antes o *if* era assim:
'if' cond=expr ('&&' cond=expr)* | ('||' cond=expr)* then=block ('else' otherwise=block)?

e agora ficou assim:
'if' cond=expr ('&&' cond=expr | '||' cond=expr)* then=block ('else' otherwise=block)?

– function_def : 'func' name=ID '(' args? ')'':' precision block;

  args : ID ':' precision (',' ID ':' precision)*;

  precision : 'int' | 'float';

5 IMPLEMENTAÇÃO DO COMPILADOR

A implementação do compilador se baseou principalmente na linguagem de exemplo da disciplina, sendo o resto da implementação baseada em pura força-bruta[4].

5.1 CARACTERÍSTICAS FUNCIONAIS DO COMPILADOR

- Tipos *int* e *float* implementados corretamente;
- Possibilidade de definir e chamar funções;
- Estruturas *if-then-else*, *while*, *for* e *switch-case* funcionais;
- Operações matemáticas e lógicas funcionais, para ambos os tipos (*int* e *float*);
- Toda a interface de comando (CLI) exigida.

5.2 CARACTERÍSTICAS NÃO-FUNCIONAIS

- O reporte de erros semânticos foi implementado, e testado, apenas parcialmente, podendo haver erros ainda não detectados e/ou tratados.
- Fora os tipos *int* e *float*, nenhum outro tipo foi implementado. Além disso, somente as operações matemáticas básicas (adição, subtração, divisão e multiplicação) foram implementadas, mesmo que a definição da linguagem suporte a operação de módulo.

5.3 LIMITAÇÕES E ERROS DO COMPILADOR

- Alguns erros sintáticos podem ser falsos positivos, porém, esses erros não interrompem o processo de compilação, apenas emitem uma mensagem de erro.
- Não é possível executar operações ou declarar variáveis fora de funções;
- Não é possível chamar funções dentro de funções[5], nem passar para funções parâmetros que consistem em operações matemáticas ou lógicas;
- Obrigatoriamente deve haver uma função nomeada *main*;
- Não é possível ter duas variáveis com o mesmo nome em funções diferentes, desde que uma dessas funções seja a *main*;
- Todas as funções precisam ter *return*;
- Para uma função poder chamar a outra, a função a ser chamada precisa ser definida antes;
- Não é possível declarar uma variável de um tipo e depois a alterar para outro tipo;
- Operações de *loop* usando variáveis do tipo *float*, especialmente o *while*, podem apresentar comportamentos estranhos, embora o resultado final apresentado não esteja totalmente errado.

[4] A implementação de todas as características não contempladas na linguagem de exemplo da disciplina foi por tentativa e erro.

[5] Exemplo: print(teste(teste2));

5.4 NOTAS DO REDATOR

Todo o progresso de implementação do trabalho, assim como a contribuição e participação, em termos de código escrito e *commits*, pode ser visualizado no seguinte repositório: `https://github.com/SadiJr/INE5622-Compiler`.

Importante frisar que, a partir da postagem desse trabalho, tal repositório, antes privado, irá ter sua visibilidade alterada para público, o que pode, infelizmente, implicar em cópia por colegas desesperados.