UNIVERSIDADE FEDERAL DE ALAGOAS

Trabalho de compiladores: Toco

João Lucas Marques Correia

Myron David Lucena Campos Peixoto

Janeiro - 2019

Sumário

1 Introdução

O presente documento especifica a linguagem de programação Toco. Essa linguagem terá o seu analisador léxico e sintático implementados na linguagem Java, de acordo com as especificações descritas nas seções seguintes deste documento.

2 Especificação da linguagem

2.1 Estrutura base de um programa

Em Toco, as variáveis podem ser declaradas em qualquer parte do código, as variáveis declaradas fora de funções são consideradas variáveis globais, já as variáveis declaradas dentro de funções são consideradas variáveis locais pertencente ao escopo de declaração. A declaração das variáveis deve se dar da seguinte maneira:

Listing 1: Declaração de variável

```
var <tipoDaVariável> <nomeDaVariável>;
```

Antes de toda declaração de variável, deve existir a palavra reservada **var**, seguida de maneira explícita do tipo da variável, e o nome da variável.

A função principal de um programa em Toco é a função com retorno **int** e nome reservado **main**. Essa função é o ponto inicial de execução do código. Sem essa função, a compilação do código gera um erro.

Funções podem apenas ser declaradas fora de outras funções, isso equivale a dizer que funções podem apenas ser declaradas no escopo global do programa. Veja a Seção 6.1 para verificar como se dá a declaração de funções e suas características.

Os blocos da linguagem são delimitados por abre chaves ({) e fecha chaves (}). As definições locais são aquelas realizadas dentro de um bloco, e tem como escopo aquele bloco.

A linguagem apenas admite comentários de linha, esses que devem ser iniciados por (//). Todo o conteúdo de uma linha após (//) é desconsiderado na compilação do programa Toco.

A seguir temos um exemplo estrutural da linguagem Toco:

Listing 2: Exemplo estrutural de um programa Toco

```
//Declaração de variável global
var <tipoDaVariável1> <nomeDaVariável1>;
var <tipoDaVariável2> <nomeDaVariável2>;
fun <tipoDoRetorno> <nomeDaFunção> ([<tipoDoParmetro>
    <nomeDoParmetro>]*){
   //Declaração de variável local
   var <tipoDoRetorno> <nomeDaVariável2>;
```

```
    return <nomeDaVariável2>;
}

//Declaração da função principal
fun int main (){
    <conjuntoDeInstruções>
}
```

2.2 Nomes, Vinculação e Escopo

2.2.1 Nomes

Os nomes em Toco são sensíveis à capitalização e não possuem limite de caracteres. Devem ser compostos de uma *Letra*, seguida ou não de uma ou mais *Letra* ou *Dígito*, desde que a sequência nunca seja uma palavra reservada.

- *Letra* é um elemento que pertence ao conjunto $[A - Z] \cup [a - z]$.
- *Dígito* é um elemento que pertence ao conjunto $[0 - 9]$.

A expressão regular que reconhece os identificadores na linguagem Toco é a que segue:

```
[A-Za-z]([A-Za-z]|[0-9])*
```

As palavras especiais da linguagem são palavras-reservadas, consequentemente não podem ser utilizadas como identificadores. A Tabela 1 mostra o conjunto de palavras reservadas.

int	string	print	for	continue
float	fun	read	in	break
boolean	var	if	to	true
char	return	else	while	false
step				

Tabela 1: Palavras reservadas da linguagem Toco

2.2.2 Vinculação

A vinculação de tipos de dados às variáveis é estática. O tipo é definido explicitamente pelo programador durante a declaração da variável.

2.2.3 Escopo

O escopo da linguagem é estático. Variáveis declaradas fora de funções tem o escopo global. Variáveis declaradas dentro de blocos tem como escopo todo o bloco em que a declaração foi realizada, admitindo **mascaramento** de variáveis. Parâmetros de funções, tem como escopo todo o bloco da função.

3 Tipos de dados

A linguagem possui quatro tipos primitivos: inteiro (*int*), ponto flutuante (*float*), caractere (*char*), booleano (*boolean*) e dois tipos padrão: cadeia de caracteres (*string*) e arranjos unidimensionais.

3.1 Tipos de dados primitivos

3.1.1 Inteiro

As variáveis do tipo Inteiro armazenam valores contidos no conjunto dos números inteiros. Esse tipo primitivo é identificado pela palavra reservada ***int***, podendo ser declarado da seguinte maneira:

Listing 3: Declaração de variável inteira

```
var int <nomeDaVariável>;
```

As operações suportadas por esse tipo primitivo de dados, também como suas constantes literais, podem ser verificadas nas Seções 4.2 e 3.3.

3.1.2 Ponto Flutuante

As variáveis do tipo Ponto flutuante assumem valores contidos no conjunto dos reais positivos ou negativos. Esse tipo primitivo é identificado pela palavra reservada ***float*** e pode ser declarado como segue:

Listing 4: Declaração de variável ponto flutuante

```
var float <nomeDaVariável>;
```

As operações suportadas por esse tipo primitivo de dados, também como suas constantes literais, podem ser verificadas nas Seções 4.2 e 3.3.

3.1.3 Caractere

As variáveis do tipo caractere armazenam valores numéricos, correspondentes aos valores alfanuméricos da tabela **ASCII**. Esse tipo primitivo é identificado pela palavra reservada ***char*** e pode ser declarado da seguinte maneira:

Listing 5: Declaração de variável caractere

```
var char <nomeDaVariável>;
```

As operações suportadas por esse tipo primitivo de dados, também como suas constantes literais, podem ser verificadas nas Seções 4.2 e 3.3.

3.1.4 Booleano

As variáveis do tipo booleano assumem apenas dois valores: **true** ou **false**. Esse tipo é identificado pela palavra reservada ***boolean*** e pode ser declarada como segue:

Listing 6: Declaração de variável booleana

```
var boolean <nomeDaVariável>;
```

Internamente os valores **true** e **false** são representados como variáveis Inteiras de valor 1 e 0, respectivamente.

As operações suportadas por esse tipo primitivo de dados, também como suas constantes literais, podem ser verificadas nas Seções 4.2 e 3.3.

3.2 Tipos de dados padrão

3.2.1 Arranjos Unidimensionais

As variáveis do tipo *arranjo unidimensional* podem ser de qualquer um dos tipos primitivos ou padrão suportados pela linguagem. Esse tipo padrão armazena uma coleção finita de valores de um mesmo tipo.

Um arranjo unidimensional é identificado através da presença de colchetes ([]) após o nome da variável. Dentro dos colchetes deve existir uma constante literal Inteira positiva, sendo este o *tamanhoDoArranjo*.

Veja abaixo alguns exemplos de arranjos unidimensionais dos tipos primitivos suportados pela linguagem:

Listing 7: Declaração de arranjos unidimensionais

```
var int <nomeDoArranjo>[tamanhoDoArranjo];
var float <nomeDoArranjo>[tamanhoDoArranjo];
```

O limite inferior do arranjo unidimensional por padrão é zero, já o seu limite superior é definido como o *tamanhoDoArranjo - 1*. A linguagem realiza verificação de referências em relação a faixa. Ao acessar um índice negativo, ou fora da faixa do arranjo, um erro será gerado.

Para acessar um elemento do arranjo, pode-se proceder da seguinte maneira:

Listing 8: Acesso a um arranjo unidimensional

```
<nomeDoArranjo>[indice];
```

A alocação do arranjo se dá de maneira estática no momento da declaração.

Esse tipo de dado padrão não suporta operações. Seus elementos, contudo, estão sujeitos às operações de seus tipos.

3.2.2 Cadeia de caracteres

As variáveis do tipo cadeia de caracteres, são compostas de um *arranjo unidimensional* de caracteres (*char*) finalizado por um elemento *"\0"*.

O tamanho da cadeia de caracteres, assim como todo arranjo unidimensional, deve ser declarado explicitamente entre colchetes no momento da declaração da variável. A menos que após a declaração apareça uma sentença de atribuição, nesse caso a variável será inicializada com o tamanho necessário para armazenar o resultado da sentença.

Uma variável do tipo cadeia de caracteres pode ser declarada conforme descrito:

Listing 9: Declaração de cadeia de caracteres

```
var string <nomeDoArranjo>[tamanhoDaCadeia];
var string <nomeDoArranjo> = "Olá Mundo!";
```

Todas as regras de acesso a faixa dos arranjos unidimensionais são válidas para esse tipo.

As operações suportadas por esse tipo padrão de dados podem ser verificadas na Seção 4.2.

3.3 Constantes literais

As constantes literais para cada tipo de dados da linguage, são definidas conforme a Tabela 2.

3.4 Equivalência de tipos

3.4.1 Coerções e conversões de tipo admitidas

A linguagem Toco admite coerção de tipo conforme a descrito na Tabela 3.

Tipo	Intervalo	Exemplo
Inteiro	[-2 147 483 648, +2 147 483 647]	[-3,-2, -1, 0, 1, 2, 3]
Ponto Flutuante	[-3.4e+38, +3.4e+38]	[-1.5, -0.5, 0, 1.5, 2.0]
Booleano	[true, false]	true, false
Caractere	[0,127]	a, b, c, d, e, &, @
Cadeia de caracteres	-	Oi mundo!
Arranjo unidimensional	-	[0,1,2], [true, false], [1.5, 2.5, 3.5]

Tabela 2: Constantes literais por tipos de dados

Do tipo	Para o tipo
Inteiro	String, Float
Float	String, Inteiro
Boolean	String
Caractere	String

Tabela 3: Coerções admitidas entre tipos

O tipo padrão *arranjo de caracteres* não admite nenhuma coerção de tipo, entretanto os seus elementos individualmente admitem coerção de tipo.

A linguagem Toco não oferece nenhum mecanismo para a conversão explícita de tipos.

3.4.2 Verificação de tipos

A verificação de tipos é realizada de maneira estática, em tempo de compilação, garantindo que operações sejam realizadas apenas entre tipos de dados compatíveis.

Para verificar detalhes sobre a equivalência de tipos em operações aritméticas, relacionais, de concatenação ou formatação, verifique as Seções 4.2.1, 4.2.2, 4.2.4, 4.2.5 e também a Seção 3.4.1 que trata especificamente de coerções e conversões de tipo.

A verificação de tipos também age sobre as funções, verificando se essas recebem os tipos dados esperados. Na passagem de parâmetros para funções, nenhuma coerção é admitida pela linguagem. Assim, o tipo de um parâmetro qualquer passado para uma função, deve ser do mesmo tipo que foi especificado explicitamente na declaração da função.

3.4.3 Tipagem forte

A linguagem Toco é fracamente tipada.

3.5 Constantes com nome

A linguagem Toco não admite constantes com nomes.

4 Atribuição e Operadores

4.1 Atribuição

Na linguagem Toco, uma **instrução de atribuição**, assim como em grande parte das linguagens imperativas, associa um valor a uma variável. Esse valor pode provir de uma outra variável, uma constante literal, uma operação aritmética, uma operação de concatenação, uma operação de formatação ou retorno de função.

Uma instrução de atribuição é implementada através do símbolo (=), onde do lado esquerdo deste símbolo, temos a variável alvo. Abaixo estão alguns exemplos de instruções de atribuição.

Listing 10: Operação de atribuição

```
var float <nomeDaVariável1> = (3.14 * 5) * (3.14 * 5);
var string <nomeDaVariável2> = "Olá mundo";
var int <nomeDaVariavel3> = 10;
var int <nomeDaVariavel4> = 20;

<nomeDaVariavel3> = <nomeDaVariavel4>;
```

Note que a instrução de atribuição não retorna valor, assim, não pode ser utilizada como operando em operações aritméticas ou relacionais.

4.2 Operadores

Operadores definidos em uma linguagem, são mecanismos que possibilitam a realização de certos procedimentos em uma ou entre duas sentenças. Essas sentenças podemos chamar de operandos. Assim, um operador opera sobre seus operandos, produzindo algum resultado.

Os tipos de operadores, são organizados conforme a natureza do procedimento que ele realiza entre seus operandos. Assim, classificamos os operadores em: operadores aritméticos, operadores relacionais, operadores lógicos, além de operadores específicos para concatenação e formatação.

4.2.1 Operadores aritméticos

Os operadores aritméticos, implementam as operações aritméticas oriundas da matemática. Dessa maneira, esses só operam os tipos primitivos ***int*** e ***float***.

Veja na Tabela 4 os operadores aritméticos suportados pela linguagem, ordenados de acordo com seu nível de precedência.

Operador	**Símbolo**	**Quantidade de operandos**
Negativo	-	Um
Divisão e Multiplicação	/, *	Dois
Soma e Subtração	+, -	Dois

Tabela 4: Operadores aritméticos

O tipo de uma operação aritmética é determinado pelo tipo da variável alvo. Assim, os tipos ***int*** e ***float***, podem ser operados entre si. Nestes casos, a variável de tipo diferente da variável alvo é convertida implicitamente para o tipo alvo e a operação é realizada.

4.2.2 Operadores relacionais

Operadores relacionais, realizam procedimentos de comparação entre seus operandos. O tipo de uma operação relacional é sempre **boolean**, retornando **true** caso a avaliação seja verificada verdadeira, ou **false** caso contrário.

Podemos verificar na Tabela 5 a lista de operadores relacionais que esta linguagem possui.

Operador	**Símbolo**
Igualdade	==
Desigualdade	!=
Menor que	<
Maior que	>
Menor ou igual que	<=
Maior ou igual que	>=

Tabela 5: Operadores relacionais

Todos os tipos primitivos, gozam do uso desses operadores, com **excessão** do tipo primitivo ***boolean*** que só faz uso das operações de *Igualdade* e *Desigualdade*.

Diferente dos operadores relacionais, os operadores relacionais não admitem coerção para seus operandos.

4.2.3 Operadores lógicos

Operadores lógicos, estão restritos à operações somente entre variáveis do tipo ***boolean***. Veja na Tabela 6 os operadores suportados pela linguagem.

Operador	**Símbolo**	**Quantidade de operandos**
Negação	!	Um
Conjunção	&&	Dois
Disjunção	||	Dois

Tabela 6: Operadores lógicos

Esses operadores implementam as funções da lógica proposicional, assim o operador *Conjunção* retorna **true** se e somente se ambos os seus operandos forem verdadeiro. O operador *Disjunção* retorna **true** se e somente se pelo menos um dos seus operandos for verdadeiro. O operador de *Negação* apenas nega o seu operando.

A linguagem **admite** avaliação em curto circuito para operações lógicas.

4.2.4 Operador de concatenação

O operador de concatenação, é um operador especial para unir duas variáveis em uma única. Veja na Tabela 7 as características desse operador.

Operador	**Símbolo**	**Quantidade de operandos**
Concatenação	++	Dois

Tabela 7: Operador de concatenação

O operador de concatenação opera sobre os tipos *int*, *float*, *boolean*, *char* ou *string*. Resultando sempre em um tipo de dados ***string***.

Em uma operação, caso os operandos sejam de tipos de dados diferentes, os seus valores são convertidos para *string* e a operação é realizada. Caso os operandos sejam *string* a operação é realizada de tal forma que o resultado é uma variável do mesmo tipo, com o tamanho igual a soma do tamanho dos seus operandos. Veja o exemplo:

Listing 11: Exemplo de uso do operador de concatenação

```
var int year = 2019;
var string message[20] = "We are in ";
```

```
var string concat = message ++ year;
//A variável concat tem tamanho 15 e armazena: We are in 2019
```

4.2.5 Operadores de formatação

Os operadores de formatação são utilizados especialmente para explicitar o formato que um dado de uma variável deve possuir.

Operador	**Símbolo**	**Quantidade de operandos**
Limitador de campos	%	Dois
Limitador de casas decimais	%%	Dois

Tabela 8: Operadores de formatação

O operador limitador de campos é binário. Assim, opera sobre um tipo de dados **int**, **float** ou **string** como operando mais a esquerda e um tipo **int** mais a direita, não permitindo coerção. O tipo dessa operação é determinado pelo tipo da variável mais a esquerda. O resultado da operação será um valor contendo as n primeiras casas do valor da variável à esquerda do operador. Onde n será o valor indicado pelo operando à direita do operador.

Listing 12: Exemplo de uso do operador limitador de campos

```
var int date = 11121995;
var int day = date%2;

print(day);
//A  sada  será: 11
```

Do mesmo modo o operador limitador de casas decimais é binário. Logo, opera sobre um tipo de dados **float** como operando à esquerda e um tipo **int** à direita, não permitindo coerção. O tipo da operação sempre é **float**. O resultado da operação será um o valor da variável à esquerda do operador com n casas decimais. Onde n será o valor indicado pelo operando à direita do operador.

Listing 13: Exemplo de uso do operador limitador de campos

```
var float pi = 3.1415;
var float reducedPi = pi%%2;
```

```
print(reducedPi);
//A sada será: 3.14

print("O valor de pi é:" ++ 3.141592%%1);
//A sada será: 3.1
```

4.3 Ordem de precedência e associatividade

Nas seções anteriores, apresentamos os operadores da linguagem Toco e em alguns casos o seu nível de precedência por grupo. Nessa seção veremos de maneira sintetizada como se dá a precedência entre todos os operadores da linguagem e sua respectiva associatividade. Começaremos pela Tabela 9 abaixo, ela demonstra os operadores da linguagem, em ordem decrescente de precedência.

Operador	**Símbolo**
Negativo, Negação	-, !
Multiplicação e Divisão	*, /
Soma e Subtração	+, -
Limitador de campo e casas decimais	%, %%
Concatenação	++
Igualdade, Desigualdade, Menor, Maior, Menor ou igual, Maior ou igual	==, !=, <, >, <=, >=
Conjunção, Disjunção	&&, ||

Tabela 9: Ordem de precedência de operadores em ordem decrescente

Esse é o nível de precedência de operadores imposto pela linguagem, porém existe um mecanismo para alterar a precedência de operadores, estes são os parênteses (*( )*). Operadores entre parênteses, tem seu nível de precedência aumentado de tal forma que tem maior precedência do que qualquer expressão externa a ele. Isso significa na prática que, operadores dentro de parênteses sempre são avaliados primeiro do que operadores fora deles.

Quando existem operadores adjacentes com o mesmo nível de precedência a associatividade acontece da **esquerda para a direita**. Exceto para os operadores Negativo e Negação, que possuem a associatividade da **direita para a esquerda**.

5 Instruções

5.1 Entrada e Saída

Na linguagem Toco, existem duas instruções que lidam respectivamente com entrada e saída de dados: **print** e **read**

5.1.1 Print

A instrução print é responsável pela saída de dados, estes por sua vez sempre são formatados como uma cadeia de caracteres e impressos na tela. A instrução pode receber qualquer tipo primitivo, porém todos estes serão convertidos para String antes de serem impressos.

A instrução é definida com a palavra reservada **print** seguida por parênteses contendo o parâmetro da instrução.

Listing 14: Exemplo de instrução print

```
print("Hello World!");
// imprime: Hello World
```

Caso o parâmetro recebido seja do tipo String, o valor apenas é mostrado. Caso o parâmetro seja de outro tipo tal qual inteiro ou float, é feita uma concatenação interna com uma String vazia usando o operador ++, o que resulta no valor do parâmetro em formato de cadeia de caracteres, e este é mostrado na tela.

Veja abaixo exemplos de como formatar a saída de dados de tipos diferentes na tela:

Listing 15: Exemplo de instrução print com concatenador

```
print("Feliz " ++ 2019);
// imprime: Feliz 2019

var float pi = 3.14;
print("O valor de pi é: " ++ pi)
// imprime: 5.5

print(777)
// imprime: 777
```

5.1.2 Read

A instrução read é utilizada para a entrada de dados nos programas Toco. A instrução de leitura é definida pela palavra reservada **read** seguida por parênteses contendo uma ou mais variáveis alvo, separadas por vírgula. Desse modo, cada valor na entrada, separado por espaço, será associado a uma variável alvo. Veja o exemplo de código a seguir:

Listing 16: Exemplo de instrução read

```
read(<parmetro1>, <parametro2>, <parametroN>);
\\Exemplo de entrada: 1 2 3

read(<parmetro1>);
\\Exemplo de entrada: 3.14
```

5.2 Estruturas de Decisão

A linguagem Toco possui uma única estrutura de decisão: o **if/else**. A instrução começa com a palavra reservada **if**, acompanhada de uma expressão lógica entre parênteses e seguida de chaves, que são os delimitadores do bloco de instrução. Dentro do bloco podem ser declaradas outras estruturas condicionais, de repetição, chamadas de funções, operações de entrada e saída e atribuições.

Listing 17: Exemplo de estrutura de decisão

```
if(<expressãoLogica>){
   <conjuntoDeInstruções>
}
```

O bloco de instruções if pode ser seguido de uma instrução **else**, que nada mais é que a segunda via da instrução condicional. Caso a condição lógica do bloco if não seja verdadeira, as instruções contidas no bloco serão puladas e a execução será continuada no bloco else. Podemos ver a exemplificação do funcionamento logo abaixo.

Listing 18: Exemplo de estrutura de decisão

```
if(<expressãoLogica>){
    //so será executado caso <expressão lgica> = true
    <conjuntoDeInstruções>
```

```
}
else{
    <conjuntoDeInstruções>
}
```

O bloco de instrução **else** é delimitado por chaves e também pode conter outras estruturas condicionais, de repetição, chamadas de funções, operações de entrada e saída e atribuições.

5.3 Estruturas de Repetição

Existem duas estruturas de repetição na linguagem Toco: uma controlada por controle lógico e outra por contador.

5.3.1 While

A estrutura de repetição com controle lógico da linguagem Toco é o while. A instrução inicia com a palavra reservada **while** seguida por uma expressão lógica entre parênteses. Chaves delimitam o bloco de instruções, dentro do bloco podem ser declaradas outras estruturas condicionais, de repetição, chamadas de funções, operações de entrada e saída e atribuições.

Listing 19: Exemplo de instrução while

```
while (<expressãoLogica>){
    <conjuntoDeInstruções>
}
```

O conjunto de instruções será executado enquanto a expressão lógica for verdadeira.

5.3.2 For

O for é a estrutura de repetição controlada por contador da linguagem Toco. A instrução é iniciada com a palavra reservada **for** seguida de uma variável previamente declarada que será o contador. Em seguida utiliza-se a palavra reservada **in** para indicar qual intervalo o contador irá iterar. O intervalo é indicado por um valor inicial seguido da palavra reservada **to** e de um valor final. Tanto o valor inicial quanto o valor final podem ser variáveis inteiras ou constantes numéricas inteiras. O bloco da instrução é delimitado por chaves e

dentro dele podem ser declaradas outras estruturas condicionais, de repetição, chamadas de funções, operações de entrada e saída e atribuições.

Listing 20: Exemplo de instrução for

```
for <contador> in <valorInicial> to <valorFinal> {
    <conjuntoDeInstruções>
}
```

A instrução **for** também aceita controle de passo, que é feito por meio da palavra reservada **step** seguida pelo valor do passo, que é um numero inteiro. O exemplo pode ser visto abaixo:

Listing 21: Exemplo de instrução for

```
for <contador> in <valorInicial> to <valorFinal> step <valorDoPasso> {
    <conjuntoDeInstruções>
}
```

Caso o **step** for omitido, o valor padrão do passo é 1.

5.4 Desvios Incondicionais

Em Toco, existem dois tipos de desvio incondicional, o break e o continue.

5.4.1 break

O primeiro deles se dá pelo uso da palavra reservada **break**. A instrução **break** pode ser utilizada nos blocos das estruturas de repetição(for/while). Seu funcionamento é simples porém bastante útil, ao utilizar o break em uma instrução de repetição, a execução do loop é parada e a execução do programa continuará de onde estava logo após a instrução de repetição. Podemos ver seu funcionamento claramente no exemplo abaixo:

```
for i in 0 to 100 {
    if(i == 50){
        break;
    }
}
print("teste");
```

No exemplo acima, caso o contador atinja o valor 50, a repetição pára e a próxima instrução a ser executada é o print, que está logo após o for.

5.4.2 continue

O outro desvio incondicional da linguagem Toco é o **continue**. Seu funcionamento é parecido com o break, porém ao invés de pular todo o bloco, seu desvio é feito apenas pulando a iteração atual do bloco de repetição

```
for i in 0 to 100 {
    if(i == 50){
        continue;
    }
}
```

6 Subprogramas

6.1 Funções

Conforme já mencionado na Seção 2.1 funções podem apenas ser declaradas fora de outras funções, isso equivale a dizer que funções não podem ser aninhadas e apenas ser declaradas no escopo global do programa. A declaração de uma função se dá como segue:

Listing 22: Declaração de função

```
fun <tipoDoRetorno> <nomeDaFunção>([<tipoDoParmetro><nomeDoParmetro>]*){
    <conjuntoDeInstruções>
    return <valorDeRetorno>;
}
```

Antes de toda declaração de função, deve existir a palavra reservada **fun**, seguida do tipo de retorno da função, o nome da função, e um conjunto de tipos e nomes de parâmetros separados por vírgula.

6.1.1 Retorno

O retorno de uma função é indicado pela palavra reservada **return** seguida da variável ou expressão de retorno, do mesmo tipo do retorno indicado na declaração da função. Funções do tipo **void** não precisam indicar ponto de retorno da função, assim não se faz necessário o uso da palavra reservada **return**.

Funções em toco também podem retornar arrays. A sintaxe é descrita da forma abaixo:

Listing 23: Exemplo de função que retorna array

```
fun int[10] umArray(){
    int a[10];
    for(i in 0 to 10){
        a[i] = i;
    }
    return a;
}
```

6.1.2 Chamada

A chamada de uma função pode ser realizada conforme o código abaixo:

Listing 24: Chamada de função

```
<nomeDaFunção>([<nomeDoParmetro>]*);
```

Para todos os tipos de dados primitivos ou padrão, a passagem de valores para funções é por valor.

6.2 Procedimentos

Funções sem retorno, se comportam de maneira análoga a procedimentos. Desse modo, a linguagem Toco oferece suporte a procedimentos.

6.3 Características dos subprogramas

Algumas particularidades dos subprogramas devem ser destacadas. Subprogramas não podem ser passados como parâmetros, não podem ser sobrecarregados e não podem ser genéricos.

7 Programas exemplo

7.1 Alô Mundo

Listing 25: Programa Alô Mundo

```
fun int main(){
   print(" Al  Mundo ");
   return;
}
```

7.2 Fibonacci

Listing 26: Programa para listar elementos da série de fibonacci

```
fun int fib(int n){
    var int aux;
    var int a = 0;
    var int b = 1;
    var int i = 0;

    while(i < n){
        aux = a + b;
        a = b;
        b = aux;
        if(aux > n){
            print(a);
            break;
        }
        print(a ++ ",");
    }
}

fun int main(){
    var int n;
    read(n);
    fib(n);
    return 0;
}
```

7.3 Shell Sort

Listing 27: Implementação do Algoritimo Shell sort

```
fun void shellsort(int vet[], int size){
    var int i;
    var int j;
    var in value;
    var int gap = 1;

    while(gap > 1){
        gap = 3 * gap + 1;
    }

    while(gap > 1){
        gap = gap / 3;
        for(i in gap to size){
            value = vet[i];
            j = i;
            while(j >= gap && value < vet[j - gap]){
                vet[j] = vet[j - gap];
                j = j - gap;
            }
            vet[j] = value;
        }
    }

}

fun int main(){
    var int n;
    var int i;
    print("Digite o tamanho do array: ");
    read(n);
    var int a[n];

    //le elementos do array
    for(i in 0 to n){
        read(a[i]);
```

```
    }

    print("Array original: ");
    for(i in 0 to n){
        print(a[i] ++ " ");
    }

    shellsort(a, n);

    print("Array ordenado: ");
    for(i in 0 to n){
        print(a[i] ++ " ");
    }

    return 0;
}
```

8 Especificação de tokens

Nesta seção estão descritos os tokens da linguagem. Contendo sua enumeração na linguagem Java, suas categorias e suas expressões regulares.

8.1 Enumeração e descrição das categorias

Abaixo segue a enumeração das cetegorias dos tokens, escrita na linguagem Java:

Listing 28: Enumeração das categorias dos tokens

```
public enum Categoria {
MAIN(1), VOID(2), FEC_PAR(3), ABR_PAR(4), ABR_CHA(5), FEC_CHA(6),
    ABR_COL(7), FEC_COL(8), VIRGULA(9), PON_VIR(10), ATRIBUICAO(11),
    IDENTIFICADOR(12), TIP_INT(13), TIP_FLO(14), TIP_BOO(15),
    TIP_CHA(16), TIP_STR(17), OPE_ADI(18), OPE_SUB(19), OPE_MUL(20),
    OPE_DIV(21), OPE_IGU(22), OPE_DIF(23), OPE_MEN(24), OPE_MAI(25),
    OPE_MEN_IGU(26), OPE_MAI_IGU(27), OPE_OU(28), OPE_E(29),
    OPE_NEG(30), OPE_CON(31), RETURN(32), IF(33), ELSE(34), FOR(35),
    IN(36), TO(37), STEP(38), WHILE(39), VAR(40), INPUT(41),
    OUTPUT(42), FUN(43), CONTINUE(44), BREAK(45), CON_INT(46),
    CON_FLO(47), CON_BOO(48), CON_CHA(49), CON_STR(50), EOF(51),
    OPE_FOR_CAM(52), OPE_FOR_DEC(53);
}
```

A **Tabela 10** e **Tabela 11** contém as categorias simbólicas dos tokens e suas descrições.

8.2 Expressões regulares e lexemas

A **Tabela 12** contém expressões regulares auxiliares, enquanto a **Tabela 13** e **Tabela 14** contém as categorias de tokens e suas expressões regulares reconhecedoras.

Categoria simbólica	**Descrição**
MAIN	Função principal
VOID	Retorno vazio
ABR_PAR	Abertura de parênteses
FEC_PAR	Fechamento de parênteses
ABR_CHA	Abertura de chaves
FEC_CHA	Fechamento de chaves
ABR_COL	Abertura de colchetes
FEC_COL	Fechamento de colchetes
VIRGULA	Vírgula
PON_VIR	Ponto e vírgula
ATRIBUICAO	Atribuição
IDENTIFICADOR	Identificador
TIP_INT	Tipo inteiro
TIP_FLO	Tipo float
TIP_BOO	Tipo boolean
TIP_CHA	Tipo char
TIP_STR	Tipo String
OPE_ADI	Operador de soma
OPE_SUB_NEG	Operador binário subtração ou unário negativo
OPE_FOR_CAM	Operador de limitação de quantidade de campos
OPE_FOR_DEC	Operador de limitação da quantidade de decimais
OPE_MUL	Operador de multiplicação
OPE_DIV	Operador de divisão
OPE_IGU	Operador de igualdade
OPE_DIF	Operador de desigualdade
OPE_MEN	Operador menor que
OPE_MAI	Operador maior que
OPE_MEN_IGU	Operador de menor ou igual que
OPE_MAI_IGU	Operador maior ou igual que

Tabela 10: Categorias e suas descrições

Categoria simbólica	Expressão regular
OPE_OU	Operador ou
OPE_E	Operador e
OPE_NEG	Operador de negação
OPE_CON	Operador de concatenação
RETURN	Retorno de função
IF	Condicional se
ELSE	'Condicional senão
FOR	'Iterador for
IN	Definidor de range do for
TO	Limite do range do for
STEP	Passo do for
WHILE	Iterador while
VAR	Variável
INPUT	Entrada
OUTPUT	Saída
FUN	Função
CONTINUE	Desvio incondicional continue
BREAK	Desvio incondicional break
CON_INT	Constante literal inteira
CON_FLO	Constante literal float
CON_BOO	Constante literal boolean
CON_CHA	Constante literal char
CON_STR	Constante literal string
EOF	Indica o fim do arquivo

Tabela 11: Categorias e suas descrições

Nome	Expressão regular
letra	[A-Za-z]
digito	[0-9]
símbolo	'!'|'#'|'$'|'%'|'&'|'\'|"'|'('|")'|'*'|'+'|','|'-'|'.'|'/'|':'|';'|'<'|'='|'>'|'?'|'@'|'['|'\'|']'|'^'|'_'|'`'|'{'|'|'|'}'|'∼'

Tabela 12: Expressões regulares auxiliares

Categoria simbólica	**Expressão regular**
MAIN	'main'
VOID	'void'
ABR_PAR	'('
FEC_PAR	')'
ABR_CHA	'{'
FEC_CHA	'}'
ABR_COL	'['
FEC_COL	']'
VIRGULA	','
PON_VIR	';'
ATRIBUICAO	'='
IDENTIFICADOR	letra(letra|digito)*
TIP_INT	'int'
TIP_FLO	'float'
TIP_BOO	'boolean'
TIP_CHA	'char'
TIP_STR	'string'
OPE_ADI	'+'
OPE_SUB_NEG	'-'
OPE_FOR_CAM	%
OPE_FOR_DEC	%%
OPE_MUL	'*'
OPE_DIV	'/'
OPE_IGU	'=='
OPE_DIF	'!='
OPE_MEN	'<'
OPE_MAI	'>'
OPE_MEN_IGU	'<='
OPE_MAI_IGU	'>='

Tabela 13: Categorias e suas expressões regulares

Categoria simbólica	**Expressão regular**
OPE_OU	'||'
OPE_E	'&&'
OPE_NEG	'!'
OPE_CON	'++'
RETURN	'return'
IF	'if'
ELSE	'else'
FOR	'for'
IN	'in'
TO	'to'
STEP	'step'
WHILE	'while'
VAR	'var'
INPUT	'read'
OUTPUT	'print'
FUN	'fun'
CONTINUE	'continue'
BREAK	'break'
CON_INT	[digito]+
CON_FLO	[digito]+'.'[digito]+
CON_BOO	'true'|'false'
CON_CHA	"letra'|'digito'|'simbolo"
CON_STR	'"'(letra|digito|simbolo)*'"'
EOF	'EOF'

Tabela 14: Categorias e suas expressões regulares