Universidade Federal de Alagoas
Instituto de Computação

COMPILADORES - 2018.1

Especificação mínima da linguagem: Albireo 18

Aluno: Victor Rafael Almeida Cavalcante
Curso: Ciência da Computação
Professor: Alcino Dall' Igna Junior

Maceió-AL
Jul, 2018

Sumário

1 Introdução

Albireo é uma linguagem de programação idealizada em 2018 pelo estudante de graduação em Ciência da Computação (UFAL), Victor Cavalcante, com objetivo de aprender os detalhes de implementação de uma linguagem e seu relacionamento com a fase de compilação.

2 Visão geral

2.1 Domínio de programação

A linguagem **Albireo** não deve ser utilizada em programas críticos, programas de médio ou grande porte, sendo seu uso restrito apenas a programas simples.

2.2 Nomes

Quanto as convenções de Nome, **Albireo** apenas aceita letras e números. O nome deve sempre começar por uma letra minúscula, sendo assim case-sensitive, o que auxilia o programador a manter um determinado padrão a fim de manter sua legibilidade.

Exemplo:

```
charstring carmenSandiego = "not found";
```

A linguagem faz uso de **palavras-reservadas**, ou seja, palavras que não podem ser usadas como um nome. Exemplos de palavras reservadas:

foo return if do else whilecontrol while empty

2.3 Vinculações

A vinculação ocorre de forma estática, o programador especifica o tipo da variável antes de seu nome, esse tipo é então mantido desde a fase de compilação até a fase de execução do programa.

Exemplo:

```
char tea;
tea = 't';
```

2.4 Escopo

O escopo da linguagem é **estático**, ou seja, é determinado antes de sua execução. Assim um nome se refere ao seu ambiente léxico local, por exemplo:

```
int coffeeBeans = 93;

while (coffeeBeans > 0){
    float price = coffeeBeans * 0.2;
    coffeeBeans = coffeeBeans - 1;
}

float lastBeanPrice = price; // Erro
```

No código acima ocorreria um erro, pois nesse caso a variável **price** é acessível apenas no bloco **while** não podendo ser acessada fora do bloco. Esse comportamento ocorre também nos blocos **if** e **whilecontrol**.

2.5 Inicialização

Para inicializar a aplicação usa-se a função nativa initializeApp() que carrega e executa todo o programa.

```
foo empty initializeApp() {
    ...
}
```

2.6 Comentários

Os trechos de código que devem ser ignorados podem ser introduzidos utilizando-se barras duplas na forma: "// <código>".

Exemplo:

```
// int codeError = 400;
```

3 Tipos de Dados

A linguagem possui suporte aos seguintes tipos:

3.1 Inteiro

O tipo *Inteiro* é representado pela constante literal **int**. Ele representa um número do conjunto dos números inteiros e possui um range de valores de -2,147,483,648 a 2,147,483,647 ocupando 4 bytes de espaço.

Exemplo de uso:

```
int numOfCakes = 1000;
```

Operações legais: **Soma, subtração, multiplicação, divisão e resto.**

3.2 Ponto Flutuante

O tipo *Ponto Flutuante* é representado pela constante literal **float**. Ele representa um número do conjunto dos números reais e possui um range de valores de 1.2E-38 to 3.4E+38 ocupando 4 bytes de espaço, e com precisão de 6 casas decimais.

Exemplo de uso:

```
float cakePrice = 1.99;
```

Operações legais: **Soma, subtração, multiplicação e divisão.**

3.3 Caractere

O tipo *Caractere* é representado pelo literal **char**. Ele representa um caractere único pertencente ao conjunto dos caracteres do padrão ASCII, ocupando 1 byte cada. Nota: a expressão do lado direito da atribuição deve utilizar aspas simples.

Exemplo de uso:

```
char graphVertex = 'B';
```

Operação legal: **Soma**

3.4 Booleano

O tipo *Booleano* é representado pelo literal **boolean**. Ele representa os valores lógicos Verdadeiro (yes) e Falso (no).

Exemplo de uso:

```
boolean answer = yes;
```

Operações legais: **Negação, Conjunção e Disjunção**

3.5 Cadeia de Caracteres

O tipo *Cadeia de Caracteres* é representado pelo literal **charstring**. Ele representa uma cadeia/grupo de caracteres pertencentes ao conjunto dos caracteres do padrão ASCII. Nota: a expressão do lado direito da atribuição deve utilizar aspas duplas.

Exemplo de uso:

```
charstring quote = "May your choices reflect your hopes, not
your fears";
```

Operação legal: **Concatenação**

3.6 Arranjos Unidimensionais

O tipo *Arranjos Unidimensionais* é representado pelo literal **group**. Ele representa uma coleção/grupo de tipos primitivos.

Todos os tipos primitivos de um grupo devem ser do mesmo tipo. Um elemento de um grupo pode ser acessado através de seu índice. O índice do primeiro elemento de todo grupo é sempre 0. O tamanho máximo de cada grupo é especificado na declaração do nome.

Exemplos de uso:

```
group int randomNumbers[5] = {7, 123, 34, 52, 0};
randomNumbers[0] = 5; // Muda valor (7) na posicao 0, para 5
```

4 Operações

Não há suporte para *Sobrecarga de operadores*.

4.1 Precedência

As regras de precedência de operadores são baseadas naquelas da matemática. Logo, os separadores (), [] e {} tem a mais alta precedência, seguida pela multiplicação e divisão no mesmo nível, depois pela adição e subtração binária no mesmo nível.

4.2 Associatividade

O tipo de associatividade das operações na linguagem é sempre da esquerda para a direita, exceto nos casos onde há operadores unários ('-' e '!') e operadores relacionais. Logo, na seguinte expressão o operador esquerdo é avaliado primeiro:

```
a - b + c ;
```

Porém a expressão a seguir, constituiria em um erro de sintaxe:

```
a > b > c ; //Incorreto
```

Podendo ser substituído por:

```
(a > b) && (b > c)
```

4.3 Efeitos colaterais

Um efeito colateral de uma função, chamado de um **efeito colateral funcional**, ocorre quando a função modifica um de seus parâmetros ou uma variável global.

Dado que a linguagem garante a ordem de precedência, não há problemas referente a efeitos colaterais.

4.4 Conjunto de Operadores

4.4.1 Aritméticos

A linguagem possui suporte as seguintes operações matemáticas usuais para a construção de expressões aritméticas:

Tabela 1: Operadores Aritméticos

Função	**Operador**	**Exemplo**
Negativo (unário)	-	- a
Soma	+	a + b
Subtração	-	a - b
Multiplicação	*	a * b
Divisão	/	a / b
Resto	%	a % b

Tais operações obedecem as seguintes ordens de precedência, onde 1 é a maior precedência:

Tabela 2: Precedência entre operadores aritméticos

Operadores	**Precedência**
- (unário)	1
* / %	2
+ -	3

4.4.2 Relacionais

A linguagem possui suporte aos seguintes operadores relacionais:

Tabela 3: Operadores Relacionais

Operador	**Função**	**Exemplo**
Maior que	>	a > b
Menor que	<	a < b
Maior ou igual	>=	a >= b
Menor ou igual	<=	a <= b
Igual	==	a == b
Diferente	!=	a != b

Tais operações obedecem as seguintes ordens de precedência (sendo 1 a maior precedência e resumindo a associatividade à esquerda nos casos onde a precedência é a mesma):

Tabela 4: Precedência entre operadores relacionais

Operadores	**Precedência**
>, <, >=, <=	1
==, ! =	2

4.4.3 Lógicos

Para os operadores lógicos temos os seguintes valores:

Tabela 5: Operadores Lógicos

Função	**Operador**
Negação	!
Conjunção	&&
Disjunção	||

4.4.4 Concatenação

Para gerar cadeias de caracteres podemos utilizar o operador binário (+) da seguinte maneira:

```
charstring water = 'H' + 2 + 'O';
```

4.4.5 Precedência e associatividade entre operações

Além dos operadores citados acima, temos operadores que auxiliam na alteração de precedência de operadores, eles são: "()", "[]" e "{}" e naturalmente possuem a maior precedência.

A tabela de precedência entre todas as operações fica assim:

Tabela 6: Precedência entre operações

Operador	Associatividade	Precedência
( ), [ ], { }	Esquerda	1
!, - (unário)	Direita	2
*, /, %	Esquerda	3
+, -	Esquerda	4
>, <, >=, <=	Esquerda	5
==, ! =	Esquerda	6
&&	Esquerda	7
||	Esquerda	8

5 Instruções

5.1 Estrutura condicionais

As estruturas condicionais permitem executar uma série de instruções caso uma condição se realize. Elas definem um bloco de instruções e recebem uma expressão booleana como parâmetro. De acordo com o valor de dada expressão, a estrutura condicionalmente executa ou evita as instruções contidas no bloco em si.

5.1.1 Condicional de uma via

Podemos definir a estrutura condicional de uma via da seguinte forma:

```
if (<ExpBool>) do {
    // <instruction_1>
    // <instruction_2>
}
```

Dessa forma as instruções dentro do bloco só serão executadas se a <ExpBool> for verdadeira

5.1.2 Condicional de duas vias

De forma análoga podemos definir a estrutura condicional de duas vias da seguinte forma:

```
if (<ExpBool_1>) do {
    // <instruction_1>
```

```
} else do {
    // <instruction_2>
}
```

Nesse caso se o valor de <ExpBool_1> for verdadeiro, as instruções dentro do primeiro bloco (if-do) serão executadas, porém se for falso, as instruções do próximo bloco condicional (else-do) serão executadas.

5.2 Estruturas iterativas

As estruturas iterativas permitem que laços sejam implementados, ou seja, podemos iterar sobre um determinado bloco de instruções quantas vezes for necessária. A linguagem Albireo possui duas estruturas de iteração.

5.2.1 Iteração com controle lógico

Na iteração com controle lógico, o bloco de instruções definido será executado do início ao fim enquanto a expressão booleana passada no controle seja verdadeira, sendo checada sempre ao fim da execução do bloco. A estrutura pode ser definida da seguinte forma:

```
while (<ExpBool>) {
    // <instruction_1>
    // <instruction_2>
}
```

5.2.2 Iteração controlada por contador

A iteração controlada por contador executa o mesmo bloco de instruções num intervalo especificado no seguinte formato:

```
whilecontrol (<start> : <end> : <step>){
    // <instructions>
}
```

- <start> : Valor inicial do contador que delimita o inicio do intervalo a ser usado na iteração.
- <end> : Inteiro cujo valor que delimita o final do intervalo na iteração.
- <step> : Inteiro que define o passo do contador a cada iteração. Essa instrução é opcional, se omitida seu valor passa a ser igual a 1.

Onde as expressões <end> e <step> são avaliadas a cada passo.

Exemplo:

```
whilecontrol (int i = 0 : 10 : 2) {
    printout(i + ' ');
}
// Prints: 0 2 4 6 8 10

whilecontrol (int i = 0 : 5) {
    printout(i + ' ');
}
// Prints: 0 1 2 3 4 5
```

5.3 Estruturas de entrada

Para a leitura de dados de entrada utilizamos o **método readin**. Ele pode receber mais de um parâmetro, dados parâmetros são passados por referência e seus valores são atribuídos ao final da leitura. Exemplo:

```
int a;
int b;

readin(a, b); // Popula inteiros a e b com valores recebidos
```

5.4 Estruturas de saída

Para a impressão de dados na tela utilizamos o **método printout**. Ele recebe qualquer tipo de variável como parâmetro e converte implicitamente seu tipo para o tipo charstring antes de ser imprimido na tela. O método é usado da seguinte forma:

```
charstring city = "Clock Town";
int building = 312;
boolean hasMask = no;

printout(city);       // Imprime: "Clock Town"
printout(building);   // Imprime: "312"
printout(hasMask);    // Imprime: "no"
```

Se o parâmetro for do tipo **float** ele pode ser precedido da expressão .fn onde n é o número de casas decimais a serem imprimidas na tela, exemplo:

```
float price = 2.5;
printout(.f3 price);  // Imprime: "2.500"
```

6 Atribuições

A atribuição pode ser feita com o uso do operador '=', ele auxilia na atribuição do valor a direita da operação (R_VALUE) no endereço da variável que se encontra a esquerda da operação (L_VALUE).

Exemplo:

```
boolean weatherIsNice;
weatherIsNice = yes;
```

7 Funções

7.1 Definição

As funções em Albireo são definidas no formato:

```
foo <retorno> <nome> (<parametro, ...>) {
    // <instrucao_1>
}
```

- <retorno> : Valor que será retornado pela função, pode ser qualquer um dos tipos da linguagem além do tipo **empty**.
  - Ex: foo empty imprimirTexto() { ... }
- <nome> : Nome da função usado para invocar a função, deve ser único no escopo.
- <parametro> : Lista de parâmetros separados por vírgula. Cada parâmetro deve ser declarado com seu tipo explícito. Não há limites na quantidade de parâmetros.

Para retornar um valor na função usa-se a palavra reservada **return** seguida do valor a ser retornado. Por exemplo:

```
foo float buscarValorTotal (float salario, float fatura) {
    return salario - fatura;
}
```

8 Exemplos

8.1 Alô mundo

```
foo empty initializeApp() {
    printout("Alo Mundo");
}
```

8.2 Fibonacci

```
foo empty fibonacci(int numOfTerms) {
    int next, first = 0, second = 1;

    whilecontrol (int i = 0 : numOfTerms - 1) {
      if ( i <= 1 ) do {
         next = i;
      } else do {
         next = first + second;
         first = second;
         second = next;
      }
      printout(next);
      if (i != numOfTerms - 1) do {
        printout(", ");
      }
   }
}

foo empty initializeApp() {
    int numOfTerms;
    readin(int numOfTerms); // Ler numero de termos
    printout("Primeiros " + numOfTerms + " termos:");

    fibonacci(numOfTerms);
}
```

8.3 Shell Sort

```
foo empty shellSort(group int vet, int size) {
    int i, j, value, gap = 1;

    while (gap < size) {
        gap = 3 * gap + 1;
    }
    while ( gap > 1) {
        gap = gap / 3;
```

```
            whilecontrol (int i = gap : size - 1) {
                value = vet[i];
                j = i - gap;
                while (j >= 0 && value < vet[j]) {
                    vet [j + gap] = vet[j];
                    j -= gap;
                }
                vet [j + gap] = value;
            }
        }
    }

    foo empty initializeApp() {
      group int num[5];
      whilecontrol (int i = 0 : 4) {
          readin(num[i]);
      }

      shellsort(num, 5);
    }
```