Universidade Federal de Alagoas

Especificação da Linguagem AVI

Aluno: David Silva Alexandre

Professor: Alcino Dall'lgna Júnior

2021

Sumário

Introdução

A linguagem AVI foi criada para implementação dos analisadores léxico e sintático solicitados na disciplina de Compiladores. O principal objetivo é concluir a implementação destes, adquirindo assim mais conhecimento sobre a área em questão e consequentemente aprovação na disciplina.

Objetivo

O principal objetivo do presente trabalho é fornecer uma visão geral da linguagem, bem como definir especificações da mesma, para que seja possível um melhor entendimento da linguagem que será utilizada nos trabalhos da disciplina. Já no que diz respeito a criação da linguagem, o principal objetivo a ser alcançado foi a construção de um vocabulário com facilidade de leitura e escrita para facilitar a criação de programas.

Estrutura Geral de um Programa

Faremos uma descrição de como é a estrutura geral de um programa escrito em AVI, com informações sobre seu bloco de execução, declaração de variáveis e um exemplo da estrutura de um programa nesta linguagem. Sendo assim, temos

1. Bloco de Execução

   Como na linguagem C, nosso bloco de execução é composto por chaves "{}", que delimitarão o início e o fim de um bloco de execução. AVI não exige indentação, dessa forma, ignoraremos espaços em branco.

2. Declaração de variáveis

   Variáveis podem ser declaradas em qualquer parte do programa, a restrição aqui é que devem ser usadas sempre depois de serem declaradas, portanto, a declaração sempre deve vir antes do uso da variável. Variáveis criadas dentro de um bloco de execução só podem ser acessadas dentro do bloco que foram criadas. Cada variável poderá ter até 64 caracteres e não poderá conter acentuação ou começar com um dígito [0 ... 9]. Variáveis podem conter [0 ... 9], [a ... z] e [A ... Z].

3. Exemplo de Estrutura

   A seguir um exemplo da estrutura de um programa em AVI

```
int main[]
{
    int num = 1;
    print[num];

    return 0;
}
```

Especificação de Tipos

Temos vários tipos de dados em AVI, que incluem Inteiro, Ponto Flutuante, Caractere, Booleano, Cadeia de caracteres e os famosos Arranjos Unidimensionais(Arrays). A seguir, temos a especificação de cada tipo:

1. Inteiro
    - Tipo: Primitivo
    - Palavra Reservada: **int**
    - Operações: As operações disponíveis para o tipo inteiro são as Aritméticas(adição, subtração, multiplicação, divisão, unários, incremento e decremento), Relacionais(igual, diferente, maior, menor, maior ou igual, menor ou igual) e por fim as de Concatenação(com caracteres e cadeias de caracteres)
    - Para o tipo inteiro temos constantes literais com valores inteiros de até 32 bits. Coerção para float é permitida
2. Ponto Flutuante
    - Tipo: Primitivo
    - Palavra Reservada: **float**
    - Operações: Para ponto flutuante temos operações Aritméticas(adição, subtração, multiplicação, divisão, unários, incremento e decremento), Relacionais(igual, diferente, maior, menor, maior ou igual, menor ou igual) e ainda Concatenação(com caracteres e cadeias de caracteres)
    - Para valores de ponto flutuantes, as constantes literais suportadas vão até 32 bits ou mais se estiver em notação científica. Permite coerção para inteiro. Com expressão regular ‘([:digit:]) + \. ([:digit:]) +’;
3. Caractere
    - Tipo: Primitivo(ascii)
    - Palavra Reservada: **char**
    - Operações: Para caracteres temos operações Relacionais(igual, diferente, maior, menor, maior ou igual, menor ou igual) e de Concatenação(com caracteres e cadeias de caracteres)
    - As constantes literais permitidas são as presentes na tabela ascii. Sua expressão regular equivale a ([:ascii:])
4. Booleano
    - Tipo: Primitivo
    - Palavra Reservada: **bool**
    - Operações: As operações disponíveis aqui são Lógicas(negação, conjunção, disjunção inclusiva, disjunção exclusiva), Relacionais(igual e diferente) e Concatenação(com caracteres e cadeias de caracteres)
    - As constantes literais deste tipo são as palavras reservadas true e false, permitindo coerção para inteiro(1 e 0 respectivamente) e string. Sua expressão regular é ‘true | false’
5. Cadeia de caracteres

- Tipo: Não Primitivo
- Palavra Reservada: **string**
- Operações: Incluem Relacionais(igual e diferente) e Concatenação(com caracteres e cadeias de caracteres)
- As constantes literais daqui são aceitas a partir de uma sequência de caracteres em aspas duplas. Sua expressão regular é '\"([:ascii:])*\"';

6. Arranjos Unidimensionais
    - Tipo: Não Primitivo
    - Declaramos arranjos com tamanho constante, passando diretamente o valor em parênteses ou através de uma variável com um valor já atribuído
    - A Sintaxe da declaração consiste "tipo" (quantidade de itens)
        - Exemplo: int vetor(100); declara um arranjo com 100 posições, sendo assim 100 espaços, indexados de 0 a 99 e prontos para serem preenchidos com valores inteiros
        - Exemplo: int vetorexemplo(n); cria um arranjo com n posições, indexado de 0 a n - 1. Vale lembrar que isto é possível apenas se a variável n já tiver um valor maior que 0 atribuído
    - Podemos também declarar uma arranjo, como: int a(2) = (1, 2); Onde a é o nome do arranjo, 2 a quantidade de posições, 1 e 2 os valores que serão armazenados nos índices 0 e 1.
    - Sendo assim, arranjos em AVI são indexados pelo deslocamento e sua primeira posição é 0, portanto a última posição tem índice = número de posições - 1
    - Ao tentarmos acessar uma posição fora dos limites indexados no array, uma mensagem de erro é retornada
        - Exemplo: Considere o vetor int a(4) = (1, 2, 3, 4); Se tentarmos fazer int soma = a(2) + a(10); um erro seria retornado pois o índice 10(11ª posição)não existe
    - Apenas valores armazenados nos arranjos podem ser passados e retornados em parâmetros de funções, para tal basta passar o arranjo junto ao índice em que o valor está armazenado(funcao[a(1)]) por exemplo). Vale lembrar que o tipo do parâmetro deve ser igual ao do arranjo.

Conjunto de Operadores

Aqui serão mostrados todos os operadores que fazem parte da linguagem. Temos os seguintes tipos de operadores:

1. Aritméticos
    - Adição: **+**
    - Subtração e Unário Negativo: **-**
    - Multiplicação: *****
    - Divisão: **/**
    - Incremento: **++**

- Decremento: **--**

2. Relacionais
    - Numéricos, Caracteres e Cadeia de caracteres: ==, >=, >, <, <=, !=
    - Booleanos: ==, !=
3. Lógicos
    - Booleanos: &, ^, |
    - Negação: !
    - Disjunção: ||
    - Conjunção: &&
4. Concatenação
    - Todos os tipos que suportam +=

A seguir temos uma tabela com a ordem de precedência destes operadores em ordem decrescente. Os operadores da primeira linha são executados primeiro, enquanto os da última são executados por último. Vale ressaltar que os operadores da primeira linha são unários, enquanto todos os demais são binários:

- ! -- ++
*** /**
+ -
< <= >= >
== !=
&
^
|
&&
||
= +=

Em relação a associatividade, os operadores de adição, subtração, multiplicação, divisão, concatenação, relacionais e lógicos são associativos à esquerda, enquanto os operadores de atribuição, decremento e incremento são associativos à direita.

Instruções

A seguir uma especificação das formas de controle

1. Condicional de uma via
    - Palavra reservada **if**
    - **if**[ExprLog]
      {bloco de execução}
    - O bloco só será executado se a ExprLog tiver o valor verdadeiro
2. Condicional de duas vias
    - Palavras reservadas **if** e **else** / símbolos "**?**" e "**:**"
    - **if**[ExprLog]
      {bloco de execução para ExprLog com valor verdadeiro}
      else
      {bloco de execução quando ExprLog tem valor falso}
    - .[ExprLog] **?** {bloco para verdadeiro} **:** {bloco para falso}
3. Estrutura iterativa com controle lógico
    - Palavra reservada **while**
    - **while**[ExprLog]
      {bloco de execução}
    - O bloco será executado enquanto ExprLog tiver valor verdadeiro
4. Estrutura iterativa controlada por contador
    - Palavra reservada **for**
    - **for[**varInt; expArit; expArit; expArit]
      {bloco de execução}
    - Note que para o for acima a primeira "expArit" é o ínicio e a segunda o fim. A terceira é o tamanho do passo, caso omitido assume o valor 1.
    - O bloco de execução vai ser executado até o valor de início ser igual ao valor na segunda "expArit" - 1, em outras palavras o bloco será executado até fim - 1
    - varInt deve estar declarada e receber um valor antes do bloco de execução ser iniciado, a atribuição não pode ser realizada dentro dos colchetes do bloco for[]
    - O tamanho do passo(terceira "expArit" como dito anteriormente) é o incremento que a o valor inicial do laço vai receber a cada passo, caso este não apareça dentro dos [], a linguagem o considerará como 1
5. Entrada e Saída
    - **Entrada**: palavra reservada **scan**
    - **scan**[lista de variáveis separadas por vírgula];
    - **Saída**: palavra reservada **print**
    - **print**[lista de variáveis separadas por vírgula]; aceitando também strings em aspas duplas

        - Exemplo: print[“ola mundo”];
        - O exemplo acima nos daria como saída: ola mundo
        - Vale lembrar que o mesmo se aplica para valores guardados em posições de arranjos unidimensionais, acessados como: nome_do arranjo(posição)
    - Vale ressaltar que caso venha um número após a vírgula, o valor da variável anterior será formatado com a quantidade do número
    - Exemplo: considere x como um inteiro, x = 1 e y um float y = 1, se fizermos print[x, 2, y, 3] teremos como saída 01 e 1.000
    - Nos prints são permitidas apenas variáveis, valores de arranjos(passados como <nome do arranjo>(indice) e strings
    - Caso o número seja muito grande, usaremos notação científica para expressá-lo
6. Atribuição de valores e declaração de variáveis
    - Consideramos a atribuição como um **operador**
    - Para declarar uma variável expressamos seu tipo seguido do seu nome(que não pode ser iniciado com número ou qualquer coisa diferente de [a...z] ou [A...Z]) e escolhemos se queremos atribuir valor ou não para a mesma. Por exemplo:
        - int num = 0;(variável chamada num inicializada com o valor 0)
        - int nume;(variável chamada nume declarada sem valor definido)
    - Para realizar uma atribuição fazemos “nome da variável” = “valor”;

Funções

Exibiremos agora informações sobre funções, tais como modelos semânticos de passagem de parâmetro e afins.

- Ao iniciar o programa a função main será a primeira a ser chamada
- Iniciamos a declaração da função com a palavra reservada do tipo que será retornado
- Também pode iniciar com a palavra reservada void caso não tenha retorno
- Após a definição do retorno o nome deverá ser definido da mesma forma que o nome de uma variável
- Na declaração de funções temos inicialmente o tipo, seguido do nome da mesma, em seguida o bloco de parâmetros[tipo nome, tipo nome, ...] seguida do bloco de execução{...}
- Parâmetros são semelhantes à variáveis em sua declaração
- O modelo semântico de AVI, é o modelo de entrada e saída, similar ao da linguagem C
- No final de uma função, geralmente temos a palavra reservada return seguida do nome da variável ou constante que será retornada, acompanhada de um ponto e vírgula

- Funções podem ainda retornar antes do final da função, como por exemplo em casos com desvio condicional ou simplesmente não retornar nada, como em casos de funções do tipo void
- Uma função é chamada escrevendo seu nome seguido de colchetes e seus os valores de seus respectivos parâmetros, podendo estes serem constantes ou variáveis, por exemplo, a chamada para a função soma, definida como:
    - void soma[int num1, int num2]
      {
          soma = num1 + num2;
          return soma;
      }
    - seria soma[variavel1, variavel2];
- Confira alguns exemplos abaixo
    - Função do tipo inteiro com retorno

```
int exemplo[int a, char b]
{
  //comandos
  return 1;
}
```

    - Função que retorna antes do fim caso condicional seja atendido

```
int exemploII[int a, int b]
{
  if[a < b]
  {
    return a;
  }

  return b;
}
```

    - Função sem retorno

```
void exemploIII[int a]
{
  //comandos
}
```

Programas exemplo

1. Alô mundo

```
int main[]
{
    print["Alo Mundo!"];

    return 0;
}
```

2. Programa Fibonacci

```
void fibonacci[int limite]
{
    if[limite == 0]
    {
        print["Sem elementos menores que 0"];
    }
    else if[limite == 1 || limite == 2]
    {
        if[limite == 1]
        {
            print["Sem elementos menores que 1"];
        }
        else
        {
            print["1,1"];
        }
    }
    else
    {
        int i;
        int fib1 = 1;
        int fib2 = 2;
        int soma;
        int sequencia(limite);
        sequencia(0) = fib1;
        sequencia(1) = fib2;
        i = 3;
        while[i <= limite]
        {
            soma = fib1 + fib2;
            fib1 = fib2;
            fib2 = soma;
            sequencia(i-1) = soma;
            i ++;
        }
        i = 0;
        while[sequencia(i) < limite]
        {
            print[sequencia(i)];
            if[sequencia(i + 1) < limite]
            {
                print[","];
            }
            i ++;
        }
    }
}
```

```
int main[]
{
    int limite;

    scan[limite];

    fibonacci[limite];

    return 0;
}
```

3. Programa Shellsort

```
int n;
int vetor(400);

void shellSort[]
{
    int j = 0;
    int aux;

    for[j; 0; n]
    {
       if[vetor(j) > vetor(j + 1)]
        {
            aux = vetor(j);
            vetor(j) = vetor(j + 1);
            vetor(j + 1) = aux;
        }
    }
}

    int main[]
    {
        int i = 0;
        scan[n];

        for[i; 0; n]
        {
            scan[vetor(i)];
        }

        i = 0;
        while[i < n]
        {
            print[vetor(i)];
            print[" "];
        }

        shellSort[];

        i = 0;
        while[i < n]
        {
            print[vetor(i)];
            print[" "];
        }

        return 0;
    }
```