Disciplina Compiladores - UFS
Turmas T01 e T02 2023.1
Profa. Beatriz Trinchão Andrade

Especificação da Linguagem Bem-te-vi

1. Introdução

A linguagem **Bem-te-vi** é uma linguagem orientada a objetos. Em alguns aspectos, lembra a forma como a IDE BlueJ lida com o Java, então temos aqui a inspiração para o seu nome de passarinho. Além disso, Bem-te-vi traz aspectos do paradigma funcional, o que a torna levemente diferente das linguagens OO mais populares, e é uma linguagem em português (porque podemos).

Bem-te-vi, obviamente, é uma linguagem experimental, então esta especificação é passível de adaptações. Em caso de modificações na especificação, versões atualizadas serão postadas via SIGAA (no tópico de aula "Definições do Projeto") e notificações serão enviadas aos alunos.

As Seções 2, 3, 4 e 5 deste documento apresentam a especificação da linguagem. A Seção 6 contém informações sobre a avaliação e entregas.

2. Características principais e léxico

Moldes (equivalem às classes do paradigma OO):

Suportam herança e polimorfismo
Contêm atributos e métodos
Atributos das classes são constantes de tipos primitivos ou instâncias de outras classes
O tempo de vida de um atributo é o tempo de vida da instância da classe à qual ele pertence
O escopo dos objetos e constantes de um método da classe é o bloco onde foram declarados
Métodos sempre são funções

Tipos primitivos:

- A linguagem aceita os tipos **Bool, Numero e Texto** (escritos em letras maiúsculas)
- Booleanos podem assumir dois valores: **verdade** e **falso**
- Números podem ser escritos em decimal ou notação científica (com E ou e). Frações devem ser separadas por vírgula. Exemplo: 5e-3 == 5E-3 == 0,005
- Textos são escritos entre aspas duplas e podem ser de qualquer tamanho

Blocos de expressões

- Delimitados por colchetes

Comentários:

- A linguagem aceita comentários de linha, indicados pelo símbolo -- no início da linha
- A linguagem aceita comentários de bloco (possivelmente de múltiplas linhas) delimitados por { e }.
- O funcionamento dos comentários de bloco em Bem-te-vi são similares aos da linguagem C. Exemplo: o que acontece se você compilar /* */ */ em C? Estudem como um compilador C reconhece o fim de um comentário de bloco.

Estruturas de controle:

- repetição: recursão
- condição: se/senão ternário

Métodos:

- Funções (**funcao**): definidas pelos seus parâmetros (opcionais), nome, tipo de retorno e corpo
- Os parâmetros podem ser de tipos primitivos, classes, ou funções
- O retorno pode ser apenas de tipos primitivos e de classes (funções não retornam funções aqui)
- Há suporte a funções recursivas

Regras para identificadores:

- Pode-se utilizar: letras maiúsculas, letras minúsculas e *underline* ('_')
- Identificadores de constantes, objetos e funções devem começar com letra minúscula (na próxima seção, serão referenciados pelo token **`ID`**)
- Identificadores de moldes devem começar com letra maiúscula (na próxima seção, serão referenciados pelo token **`ID_MOLDE`**)
- Não são permitidos espaços em branco e outros caracteres (ex.: @, $, +, -, ^, ` , ~ , % etc.).
- **Identificadores não podem ser iguais às palavras reservadas ou operadores da linguagem.**

3. Sintático

A gramática, a seguir, foi escrita em uma versão de E-BNF seguindo as seguintes convenções:
1 - variáveis da gramática (não terminais) são escritas em letras minúsculas sem aspas.
2 – lexemas que correspondem diretamente a tokens são escritos entre aspas simples
3 - símbolos escritos em letras maiúsculas representam o lexema de um token do tipo especificado.
4 - o símbolo | indica produções diferentes de uma mesma variável.
5 - o operador ? indica uma estrutura sintática opcional.
6 - o operador * indica uma estrutura sintática que é repetida zero ou mais vezes.

```
programa ->  familia def_molde (def_molde)*

familia -> relação (';' relação)* '.'
          |  Ɛ

relação -> 'molde' ID_MOLDE 'se' 'passa' 'por' ID_MOLDE

def_molde -> 'molde' ID_MOLDE 'contem' atributos metodos 'fim'

atributos  -> (dec_cons | dec_obj)*
metodos -> dec_funcao*

dec_cons -> 'constante' ':' tipo_primitivo ID ';'
dec_obj -> 'objeto' ':' tipo_molde ID ';'

tipo_primitivo  -> 'Bool' | 'Numero' | 'Texto'
tipo_molde -> ID_MOLDE
tipo -> tipo_molde | tipo_primitivo

dec_funcao  -> ('>>')? 'funcao' ':' tipo ID '(' parametros ')' bloco
             | ('>>')? 'funcao' ':' tipo ID '(' parametros ')' ';'

parametros -> parametro ('|' parametro)*
          |  Ɛ
parametro -> tipo ID  | assinatura

assinatura -> tipo ID '(' parametros_assinatura ')'

//os parâmetros na assinatura de uma função passada como parametro não têm
//identificador
parametros_assinatura -> parametro_assinatura ('|' parametro_assinatura)*
                       | Ɛ
parametro_assinatura -> tipo | assinatura

bloco -> '[' ( dec_cons | dec_obj | definicao )* exp ']'

definicao  -> ( ID | atributo) '=' exp ';'

atributo -> ID '.' ID ('.' ID)*
```

```
exp -> VALOR_NUMERO | VALOR_TEXTO | 'verdade' | 'falso'
      | ID
      | instancia
      | atributo
      | chamada
      | 'se' '(' exp ')' 'entao' exp 'senao' exp
      | '(' exp ')'
      | '-' exp
      | exp '+' exp
      | exp '-' exp
      | exp '*' exp
      | exp '/' exp
      | exp '%' exp
      | exp '==' exp
      | exp '<' exp
      | exp '<=' exp
      | '!' exp
      | exp 'e' exp
      | exp 'ou' exp
      | bloco


//retorna um objeto da classe ID_MOLDE
instancia -> ID_MOLDE '[' lista_inicializacao ']'

lista_inicializacao  -> inicializacao_objeto ('|' inicializacao_objeto )*
                      | ε
inicializacao_objeto = 'objeto' '.' ID ('.' ID)* '=' exp


chamada  -> (ID '.')* ID '(' lista_exp ')'
lista_exp -> exp ('|' exp)*
           | ε
```

4. Semântico:

- O ponto de entrada do programa é a função precedida pelos caracteres '>>'. No máximo, uma função deve ter essa marcação em um programa. A execução fictícia de um programa em **Bem-te-vi** é iniciada com a criação automática de uma instância do molde (classe) onde a função marcada está declarada. A seguir, ela é chamada como ponto de entrada do processo de execução. Um programa sem esta marcação gera uma biblioteca após a compilação. Se a função de entrada pertence a um molde abstrato, deve haver um erro de compilação.

- Não há modificadores de acesso nos moldes. Por padrão, todos os métodos e atributos são públicos. Isso afeta o encapsulamento, um dos pilares da OO, mas não quis deixar a linguagem mais complicada.

- Na herança, um molde filho herda todos os atributos e métodos de seu molde pai. Em **Bem-te-vi**, um molde só pode ter um molde como pai.

- Atributos podem ser inicializados com o operador '.', ou durante a criação do objeto (usando a palavra reservada '**objeto**' para indicar atributos do mesmo). A inicialização só pode ocorrer uma vez por atributo (atributos têm comportamento de constantes). Não é preciso que todos os atributos sejam inicializados na mesma forma ou de uma só vez.

Exemplo 1.
```
objeto: Cachorro bicho; --assumindo que a Cachorro tem atributos peso e altura
bicho.altura = 0,3;
bicho.peso = 10;
```

Exemplo 2.
objeto: Cachorro bicho = Cachorro [objeto.altura = 0,3 | objeto.peso = 15 peso];

- Moldes que não são declarados como filhos de nenhum outro são implicitamente filhos do Molde `_Todos`. O molde `_Todos` não pode ser instanciado e não tem atributos ou métodos próprios.

- Caso o atributo ou método pertença a um objeto ou seja de suas classes ancestrais, ele pode ser chamado diretamente. Chamadas a atributos e métodos de outros objetos devem ser feitas usando o identificador deste outro objeto.
Exemplo: `identificador.chamada()`

- Os atributos de um objeto só podem ser definidos uma vez.

- Um molde pode conter métodos abstratos. Neste caso, ele deve conter apenas a assinatura do método, sem seu corpo. Eles adquirem um comportamento similar ao que acontece com classes abstratas, ou seja, só podem ser usados como ancestrais de classes concretas. Um molde filho só é considerado concreto se implementa todos os métodos abstratos de seus ancestrais.

- As funções podem ser sobrepostas, mas não há sobrecarga de operadores.

- Não há variáveis compostas homogêneas (arrays).

- A prioridade dos operadores é igual à de C.

- Constantes só podem ser usadas se forem inicializadas.
- Constantes e atributos só podem ser usados se forem inicializados.

- Parâmetros sempre são passados por cópia (valor) em funções e procedimentos.

- Existe uma classe `Comunicacao`, com alguns métodos pré-definidos:
    - `escreva()`: recebe como argumento uma ou mais expressões de tipo primitivo e as imprime em tela (retorna também o texto impresso).
    - `leiaTexto()`: função que retorna um valor do tipo Texto, inserido pelo usuário no teclado.
    - `leiaNumero()`: função que retorna um valor do tipo Numero, inserido pelo usuário no teclado.
    - `leiaBool()`: função que retorna um valor do tipo Bool, inserido pelo usuário no teclado.

Obs: Assumam que os dois métodos são compatíveis com os tipos primitivos

O que deve ser verificado na análise semântica:

- Se as entidades criadas pelo usuário são inseridas na tabela de símbolos - com os atributos necessários - quando são declaradas;
- Se uma entidade foi declarada e está em um escopo válido no momento em que ela é utilizada;
- Se entidades foram definidas (inicializadas) quando isso se fizer necessário;
- Checar a compatibilidade dos tipos de dados envolvidos nos comandos, expressões e atribuições;
- Polimorfismo e herança nas classes criadas (a ser refinado).

5. Geração de Código

- O código alvo do compilador é Java.

6. Desenvolvimento do Trabalho

Trabalhos devem ser desenvolvidos em grupos de 4 alunos (preferencialmente), trios, duplas ou individualmente. Os grupos devem se cadastrar na planilha:

Turma **T01**
https://docs.google.com/spreadsheets/d/1YdI5CGqArjIjQ7Pz4Mc0Sxje-OVH0yxLblFDUYC0XhM/edit?usp=sharing

Turma **T02**
https://docs.google.com/spreadsheets/d/14V4qO5gHsy-XkiwKqgN4LNKM_jr0zSH6qzkJO8CQVIg/edit?usp=sharing

Prazo de preenchimento da planilha: 03/07/2023.

A submissão das tarefas do projeto deve ser feita via SIGAA, nas datas previstas na Seção 6.3.

Foi aberto um fórum no SIGAA para a discussão sobre as etapas. Em caso de dúvida, verifique inicialmente no fórum se ela já foi resolvida. Se ela persiste, poste-a, ou consulte a professora.

6.1. Ferramentas

- Implementação com SableCC, linguagem Java.
- IDE Java (recomendação: Eclipse).
- Submissão via SIGAA.

6.2. Avaliação

- A avaliação será feita com base nas etapas entregues e em arguições feitas com os grupos.
- O valor de cada etapa está definido no plano de curso da disciplina.
- O cumprimento das requisições de formato também será avaliado na nota de cada etapa.

6.3. Entregas

Etapa 1. Análise Léxica - Parte 1 Códigos em Bem-te-vi

- **Prazo: 05/07/2023**
- **Atividade:** escrever três códigos em **Bem-te-vi** que, unidos, usem todas as alternativas gramaticais (ou seja, todos os recursos) da linguagem.
- **Formato de entrega:** arquivo comprimido contendo três códigos, onde cada código deve estar escrito em um arquivo de texto simples, com extensão "**.btv**".

Etapa 2. Análise Léxica - Parte 2

- **Prazo: 24/07/2023**
- **Atividade:** implementar analisador léxico em SableCC, fazendo a impressão dos lexemas e tokens reconhecidos ou imprimindo erro quando o token não for reconhecido.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos). O nome do pacote a ser gerado pelo sablecc deve se chamar **`bemtevi`** (em letras minúsculas e sem hífen).

Etapa 3. Análise Sintática

- **Prazo: 30/08/2023**
- **Atividade:** implementar analisador sintático em SableCC, fazendo impressão da árvore sintática em caso de sucesso ou impressão dos erros caso contrário.
- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos). O nome do pacote a ser gerado pelo sablecc deve se chamar **`bemtevi`**.

Etapa 4. Sintaxe Abstrata

- **Prazo: 18/09/2023**
- **Atividade:** implementar analisador sintático abstrato em SableCC, com impressão da árvore sintática resultante.

- **Formato de entrega:** apenas o arquivo .sable deve ser enviado. O nome do arquivo deve ser `grupo_X.sable`, onde `X` é o número do grupo (vide planilha de cadastro de grupos).
  O nome do pacote a ser gerado pelo sablecc deve se chamar **`bemtevi`**.

Etapa 5. Análise Semântica

- **Prazo: 04/10/2023**
- **Atividade:** implementação da tabela de símbolos e da árvore de relacionamentos entre classes, validação de escopo, declaração e definição de identificadores. Implementar verificação de tipos com e sem polimorfismo (a ser refinado).
- **Formato de entrega:** projeto completo, incluindo obrigatoriamente: o arquivo .sable; todas as classes java escritas pelo grupo ou geradas automaticamente; e arquivos .btv que demonstrem o que foi feito nesta tarefa.
  Também é obrigatória a entrega de um pdf contendo uma breve explicação sobre o que foi implementado nesta etapa e como.

Tarefa 7. Geração de código (extra: 2.0 pontos):

- **Prazo: o mesmo da Tarefa 6**
- Compilação de código **Bem-te-vi** com geração de código em linguagem alvo (Java)

Observações importantes:

- Entregas após o prazo sofrem penalidade de metade da nota da etapa por dia de atraso.
- Trabalhos entregues com atraso devem ser submetidos na Tarefa 'Entrega após prazo', no SIGAA, que ficará aberta durante todo o período.
- Arquivos enviados por e-mail não serão considerados.

Bom trabalho!