Compiladores
Roteiro de Laboratório 06 – Interpretador para EZLang

1 Introdução

A tarefa deste laboratório é implementar um interpretador de código para programas escritos na linguagem EZLang. Um interpretador depende de uma representação intermediária (*intermediate representation* – IR) do código de entrada. Nesse laboratório vamos usar como IR a AST (*Abstract Syntax Tree*) que foi construída no Laboratório 05.

Uma AST pode ser vista como uma versão simplificada da *parse tree* aonde somente as informações necessárias para a execução do programa são mantidas. O ponto chave para a construção da AST é definir quais informações devem ficar guardadas na árvore. Uma vez que isso é feito, a implementação do executor de código deve seguir a mesma especificação. A definição da estrutura da AST já foi feita no roteiro do Laboratório 05, aonde todos os `kind`s dos nós da AST foram detalhados. Resta então implementar o executor de código segundo a semântica da linguagem e a estrutura dos nós da AST.

Como primeira versão do *back-end*, vamos implementar um interpretador de código clássico. Isso quer dizer que logo após a execução do *front-end*, a IR é imediatamente executada. O interpretador deve ser implementado como um visitador da AST, de forma muito similar ao visitador da *parse tree* para análise semântica e construção da AST. Neste roteiro vamos usar algumas estruturas auxiliares como: uma área de memória para armazenar os valores das variáveis, e uma pilha para os valores intermediários da execução. Isso torna o interpretador efetivamente uma máquina que trabalha com código de 1-endereço (*one-address IR* – veja *slides* da Aula 05).

2 Executor de código

O executor de código (interpretador) é uma função que recebe como entrada uma AST e caminha na árvore, realizando as operações indicadas nos nós da AST. Para o armazenamento de resultados temporários é utilizada uma **pilha**.

O caminhamento do executor pela AST deve ser em **profundidade**, devido às dependências de cada operação. Ao visitar um nó da AST, deve-se executar recursivamente os filhos, para gerar os resultados (dependências) deste nó. Ao se terminar a execução de uma dada operação, o resultado fica armazenado na pilha, para ser usado depois. Por exemplo, considere a expressão `(2+3)*(7-1)`. O nó raiz dessa expressão é a operação de multiplicação, portanto o caminhamento começa por ele. A seguir, devemos executar recursivamente os filhos da esquerda e da direita, para obtermos os operandos da multiplicação. Uma vez que a execução da sub-árvore da expressão `(2+3)` termina, teremos no topo da pilha o valor `5`. Da mesma forma, após a execução da sub-árvore `(7-1)`, o topo da pilha contém o valor `6` (com `5` logo abaixo). Por fim, para terminarmos a visitação do nó de multiplicação, basta desempilhar dois valores da pilha, multiplicá-los e empilhar de volta o resultado. O funcionamento do executor para todos os `kind`s da AST é descrito na próxima seção.

Baixe o arquivo `CC_Lab06_src_java.zip` disponibilizado no Classroom. O conteúdo desse arquivo é basicamente o mesmo da solução do laboratório anterior, com a inclusão dos

arquivos do interpretador no pacote `code`. A tarefa deste laboratório é desenvolver o código das funções marcadas com `TODO` no arquivo `Interpreter.java`. Além da pilha, o executor também possui uma área de memória para o armazenamento dos valores das variáveis do programa. Uma implementação bem simples dessas estruturas já é fornecida nas demais classes do pacote `code`.

2.1 Caminhando pelos nós da AST

A enumeração `NodeKind` lista todos os possíveis `kind`s dos nós da AST. O caminhamento do executor na AST deve portanto seguir essa mesma estrutura. Abaixo estão listados todos os nós e as ações que o executor deve tomar. Certifique-se que você entendeu a implementação da pilha e da área de memória para variáveis antes de continuar.

- `ASSIGN_NODE`: Executa recursivamente o filho contendo a expressão à direita da atribuição, para deixar o valor a ser atribuído à variável no topo da pilha. A seguir, desempilha esse valor e armazena o novo valor da variável, respeitando os tipos (inteiro ou real). Esse nó não deixa um valor na pilha por não ser uma expressão.
- `EQ_NODE`: Executa recursivamente os filhos da esquerda e direita, nessa ordem. A seguir desempilha dois valores da pilha e executa a operação de comparação segundo o tipo dos operandos. Note que esta operação é sobrecarregada: se os operandos são *strings*, deve-se compará-las de forma lexicográfica (ordem alfabética). Por outro lado, comparações numéricas devem ser realizadas como esperado. (*Obs.:* Note que todos os operandos são do mesmo tipo porque foram inseridos nós de conversão na AST.) A execução desse nó deixa um valor Booleano no topo da pilha: 0 se a igualdade falhar, e 1 se tiver sucesso.
- `BLOCK_NODE`: Executa recursivamente todos os filhos do nó de forma sequencial. Esse nó não deixa um valor na pilha por não ser uma expressão.
- `BOOL_VAL_NODE`: Empilha o valor armazenado no campo `data` do nó.
- `IF_NODE`: Executa recursivamente o filho contendo a expressão de teste, desempilhando o resultado. Se for verdadeiro, executa recursivamente o bloco do `then`. Caso contrário, executa o bloco do `else`, se existir. Esse nó não deixa um valor na pilha por não ser uma expressão.
- `INT_VAL_NODE`: Idem ao `BOOL_VAL_NODE`.
- `LT_NODE`: Idem ao `EQ_NODE`, trocando a expressão de comparação.
- `MINUS_NODE`: Executa recursivamente os filhos da esquerda e direita, nessa ordem. A seguir desempilha dois valores da pilha e executa a operação de subtração, empilhando o resultado. *Obs.:* Certifique-se de operar adequadamente sobre os tipos inteiro e real. Atente-se também à ordem em que os operandos ficam na pilha.
- `OVER_NODE`: Idem ao `MINUS_NODE`, trocando a expressão aritmética.
- `PLUS_NODE`: Idem ao `MINUS_NODE`, trocando a expressão aritmética. Note que o operador + é sobrecarregado: para operandos inteiros ou reais, representa soma; para operandos Booleanos, representa a operação lógica OU; para *strings*, representa concatenação.
- `PROGRAM_NODE`: Já implementado. Visita a lista de variáveis e a seguir o bloco do programa principal. Esse nó não deixa um valor na pilha por não ser uma expressão.
- `READ_NODE`: Lê um dado de `stdin` conforme o tipo da variável no nó filho. Armazena o valor lido na área de memória reservada para a variável. Esse nó não deixa um valor na pilha por não ser uma expressão.
- `REAL_VAL_NODE`: Idem ao `BOOL_VAL_NODE`, mas atente-se ao tipo real (`float`) ao manipular a pilha.
- `REPEAT_NODE`: Executa o bloco de comandos e a seguir a expressão de teste. Se o resultado da expressão for verdadeiro, sai; caso contrário, repete o *loop*. Esse nó não deixa

um valor na pilha por não ser uma expressão.

- `STR_VAL_NODE`: Já implementado. Empilha o valor armazenado no campo `data` do nó, deixando na pilha a referência para a *string* armazenada na tabela de *strings*.
- `TIMES_NODE`: Idem ao `MINUS_NODE`, trocando a expressão aritmética.
- `VAR_DECL_NODE`: Nada a ser feito nesse nó.
- `VAR_LIST_NODE`: Nada a ser feito nesse nó. Em cenários mais complexos, esse ponto pode ser utilizado para calcular os *offsets* das variáveis declaradas (veja os *slides* da Aula 06).
- `VAR_USE_NODE`: Recupera o valor atual da variável na memória e o empilha, respeitando adequadamente os tipos.
- `WRITE_NODE`: Executa recursivamente a expressão do nó filho, exibindo o resultado em `stdout`. Esse nó não deixa um valor na pilha por não ser uma expressão.

Além dos nós descritos acima, os nós de conversão de tipos também devem ser executados adequadamente. Já foi fornecida uma implementação para algumas dessas conversões, as demais devem ser simples de inferir.

3 Executando o interpretador

Como de costume, o programa de entrada é passado como um argumento na linha de comando, com o I/O ocorrendo de forma usual pela entrada e saída padrão (`stdin` e `stdout`), como ilustra o comando abaixo.

```
$ java Main io/in/c02.ezl
read (int): 5
6
```

Algumas observações importantes:

- O seu interpretador pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- As mensagens de erros léxicos, sintáticos e semânticos são as mesmas do Laboratório 05 e devem continuar sendo exibidas como antes.
- Os programas de entrada para teste são os mesmos de sempre (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out06_java.zip`.
- Uma implementação de referência para este laboratório será disponibilizada pelo professor em um futuro próximo. No entanto, você é *fortemente* encorajado a realizar a sua implementação completa antes de ver uma solução em outro lugar.