Caneca++

Chrystian Guth
Lucas Pereira
Renan Netto

Estrutura de dados

Foi realizada uma reestruturação na análise semântica.

Agora, o analisador semântico faz apenas uma passada na árvore sintática e a partir daí cria uma estrutura de dados própria.

As referências e as verificações de tipos passam a ser resolvidas através da estrutura de dados montada.

Isso foi feito para facilitar a etapa de geração de código, uma vez que se torna mais fácil trabalhar com uma estrutura de dados própria do que com uma estrutura de dados do **Antlr** (nesse caso, a AST).

Estrutura de dados

Por exemplo, ao caminhar na árvore sintática e encontrar uma classe, é criado um objeto da instância `Classe` que por sua vez, é adicionado na `tabelaDeClasses`.

Continuando o percorrimento da árvore sintática, ao encontrar a definição de um atributo, é criado um objeto da instância `Atributo` que é adicionado na `tabelaDeAtributos` da classe em questão.

Assim, a análise prossegue montando a estrutura da árvore sintática através de objetos especializados definidos por nós.

Estrutura de dados

Com essa alteração, foi possível montar uma estrutura de dados que representa o programa definido, onde cada objeto especializado (`Classe`, `Atributo`, `Metodo`, ..., `Expressao`, `ExpressaoSoma`..., etc) é responsável por gerar o próprio código.

Isso fez com que a geração de código se transformasse em uma tarefa relativamente fácil. Bastou escrever um método comum (`gerarCodigo`) para cada uma das classes especializadas.

Porém, antes disso foi preciso definir a linguagem de máquina alvo e a própria máquina.

Máquina de registradores *versus* Máquina de pilha

Optamos por desenvolver a própria máquina virtual ao invés de utilizar uma já existente.

O primeiro passo foi definir a arquitetura da máquina: **registradores** ou **pilha**.

Máquina de registradores

Em uma máquina de registradores existe basicamente uma área de código, uma área de dados (ambos ficam na memória) e um banco de registradores.

As operação realizadas utilizam endereços de memória e de registradores. Portanto, cada instrução carrega consigo a localização dos seus operandos.

As instruções são buscadas uma a uma na área de código e manipulam o banco de registradores e a área de dados.

Máquina de pilha

A máquina Java é um exemplo que utiliza a arquitetura baseada em pilha.

Entre os componentes básicos de uma máquina de pilha estão a área de código e a pilhade dados. Existem outros componentes adicionais como a pilha de execução e a pilha de contextos.

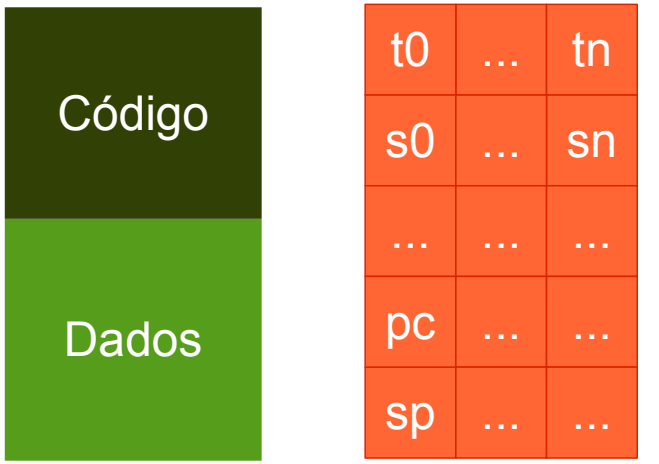
As operações realizadas buscam os operandos na pilha. Após o processamento da operação, o resultado é colocado na pilha.

Máquina de registradores *versus* Máquina de pilha

Operação: `10 + 2;`

Código gerado:

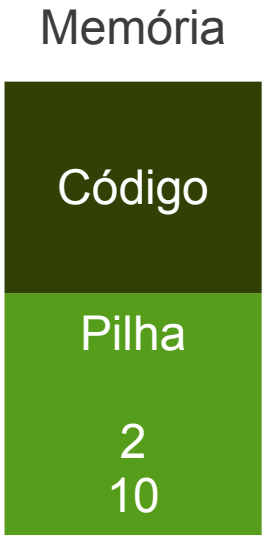
```
li $t0 10
li $t1 2
add $t0 $t0 $t1
```



Operação: `10 + 2;`

Código gerado:

```
push 10
push 2
add
```



Máquina de registradores *versus* Máquina de pilha

Optamos por construir uma máquina virtual de pilha por possuir uma implementação mais fácil.

Outro fator é que o código gerado para uma máquina de pilha é bem mais compacto e fácil de compreender.

MaquinaCaneca

A `MaquinaCaneca` possui alguns elementos básicos, sendo eles:

`pilhaDeDados`: Onde são colocados os dados das expressões. Os operandos de uma adição, por exemplo, são colocados na `pilhaDeDados` e posteriormente são consumidos pelo comando `somar`.

MaquinaCaneca

`pilhaDeContextos`: São empilhados os diversos contextos, incluindo contextos de objetos e de métodos.

Cada vez que um método ou objeto é chamado, o seu contexto é empilhado. Os contextos são responsáveis por guardar símbolos (atributos e variáveis) e até mesmo outros contextos.

Uma classe, por exemplo, guarda os pontos de entrada dos seus métodos. Os contextos também possuem uma hierarquia, da mesma forma que existe nos escopos, que foram utilizados na etapa da análise semântica.

MaquinaCaneca

`pilhaDeExecucao`: Onde são empilhados os pontos de retorno dos procedimentos executados. Cada vez que um procedimento é chamado, é empilhado um ponto de retorno.

Quando houver o retorno do procedimento, o ponto de retorno será desempilhado e a execução passará a apontar para a instrução imediatamente após a chamada.

MaquinaCaneca

`areaDeDados`: É nessa área que serão colocados os atributos, objetos e outros símbolos. A `areaDeDados` nada mais é do que um contexto global. Dentro dela existe uma hierarquia de outros contextos onde cada um é responsável pelos seus símbolos.

`areaDeCodigo`: Local onde é carregado todo o código do programa. A `areaDeCodigo` é apenas um vetor onde cada posição contém uma instrução de máquina.

MaquinaCaneca

A `MaquinaCaneca` também possui um `contadorDePrograma` que aponta para a próxima instrução de máquina a ser executada.

A execução é tipicamente sequencial, de tal forma que as instruções são executadas na mesma ordem que foram definidas na `areaDeCodigos`. Entretanto, existem instruções de máquina que manipulam o `contadorDePrograma` e dessa forma realizam o controle de fluxo de execução.

MaquinaCaneca

O *entry point* de uma programa **Caneca** é o construtor de uma classe passada como parâmetro no momento da compilação. Na **Caneca** não existe o método *main*, por isso o ponto de entrada é o construtor de uma classe.

A definição do ponto de entrada estará no inicio da `areaDeCodigos`. A execução de um programa **Caneca** irá durar enquanto o construtor do ponto de entrada estiver ativo.

MaquinaCaneca

Após terminar a execução do construtor do ponto de entrada, será desempilhado o ponto de retorno da `pilhaDeExecucao`. A execução será então direcionada para esse ponto de retorno que aponta para uma instrução de máquina responsável por encerrar a execução da `MaquinaCaneca`.

MaquinaCaneca

O coração da `MaquinaCaneca` consiste de um simples método:

```
public void executar() {
   Integer fimDoCodigo = areaDeCodigo.size();
   pilhaDeContextos.push(areaDeDados);
   while (contadorDePrograma < fimDoCodigo) {
      Codigo codigo =
areaDeCodigo.get(contadorDePrograma);
      contadorDePrograma++;
      codigo.executar(this);
   }
}
```

Todo o resto é realizado por implementações da classe `Codigo`. O código, quando for executado terá acesso à: `areaDeDados`, `areaDeCodigo`, `pilhaDeDados`, `pilhaDeExecucao`, `pilhaDeContextos` e `contadorDePrograma`.

Instruções da MaquinaCaneca

`abrirContexto`: abre o contexto para blocos de instruções. Isso permite criar uma hierarquia de blocos aninhados onde é possível ter símbolos definidos com o mesmo nome em vários níveis. O contexto pai do contexto aberto será aquele que estiver no topo da `pilhaDeContextos`.

`abrirContextoDeProcedimento`: abre um contexto para um procedimento que será executado. O pai do novo contexto será o objeto `esse`.

`atribuir`: obtém o valor e a referência na `pilhaDeDados` e atualiza a referencia no contexto que está no topo da `pilhaDeContextos`.

Instruções da MaquinaCaneca

`chamar {nomeDoProcedimento} {pontoDeRetorno}`: adiciona o ponto de retorno na `pilhaDeExecucao` e busca no contexto que está no topo da `pilhaDeContextos` o ponto de entrada do procedimento desejado.

`definirSimbolo {nome}`: define um símbolo no contexto do topo da `pilhaDeContextos`. O valor da definição será buscado na `pilhaDeDados`.

`depurar`: imprime na tela informações de depuração.

Instruções da MaquinaCaneca

`desempilhar`: desempilha um valor da `pilhaDeDados`.

`desviar {pontoDeDesvio}`: desvio incondicional que manipula o `contadorDePrograma` para que a próxima instrução seja buscada no ponto de desvio fornecido.

`desviarSeFalso {pontoDeDesvio}`: faz o mesmo que o `desviar`, porém busca na `pilhaDeDados` o próximo valor. Se este valor for falso, então o desvio será realizado.

`desviarSeFalso {pontoDeDesvio}`: faz o mesmo que o `desviarSeFalso`, porém só desvia se o próximo valor na `pilhaDeDados` for verdadeiro.

Instruções da MaquinaCaneca

`dividirI, dividirR, somarI, somarR, multiplicarI, multiplicarR, subtrairI, subtrairR`: operações aritméticas de inteiros e números reais obtidos na `pilhaDeDados`, onde o resultado é colocado na `pilhaDeDados`.

`imprimir`: imprime na tela o valor no topo da `pilhaDeDados`.

`duplicar`: duplica o valor disponível no topo da `pilhaDeDados`.

Instruções da MaquinaCaneca

`extrairContexto`: extraí o contexto da referência no tipo da `pilhaDeDados`e coloca no topo da `pilhaDeContextos`.

`fecharContexto`: fecha o contexto aberto, removedo-o da `pilhaDeContextos`.

`igual, diferente, maior, menor, maiorQue, menorQue`: operador relacionais que buscam dois operandos na `pilhaDeDados` e deixam o resultado na mesma `pilhaDeDados`.

Instruções da MaquinaCaneca

`instanciar {nomeDaClasse}`: instancia um objeto da classe desejada criando um contexto para o objeto e colocando-o na `pilhaDeContextos`.

`resolverSimbolo {nome}`: busca o símbolo no contexto do topo da `pilhaDeContextos` e adiciona o valor do símbolo na `pilhaDeDados`.

`retornar`: busca ponto de retorno na `pilhaDeExecucao` e altera o `contadorDePrograma` com o endereço obtido.

`sair`: encerra e execução da `MaquinaCaneca`.

Compilador

Nessa última etapa o compilador passa a ter duas novas opções:

lexica: Análise léxica com exibição dos erros no console.
Sintatica: Análise sintática com exibição dos erros no console.
simbolos: Apresentação dos símbolos de uma análise.
arvore: Apresentação da árvore sintática de uma análise.
semantica: Análise semântica com exibição dos erros no console.
geracaoDeCodigo: Geração de código de máquina.
execucao: Geração de código de máquina e execução do código gerado.

`fontes/java/.../Compilador.java`

```
./construir.sh
./executar.sh
<arquivo.caneca>
<opcao: {lexica, sintatica, arvore, simbolo,
semantica, geracaoDeCodigo, execucao}>
```

Exemplos

```
fontes/caneca/geracao/ExemploA.caneca
fontes/caneca/geracao/ExemploB.caneca
fontes/caneca/geracao/ExemploC.caneca
fontes/caneca/geracao/ExemploD.caneca
```