Linguagem Tiny - Implementação por meio de orientação por objetos
Chamada de procedimentos

Felipe Buzatti Nascimento , Leonardo Vilela Teixeira

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

buzatti@dcc.ufmg.br , vilela@dcc.ufmg.br

Resumo. *Este trabalho expande o interpretador para linguagem tiny, adicionando uma estrutura de procedimentos. Para construir o interpretador foi utilizada uma estrutura de classes seguindo o padrão de projetos interpretador definido em [Gamma et al. 1995].*

1. Estrutura de classes

As novas classes, usadas para implementar a estrutura de procedimentos são: comandoCall, comandoGlobal, comandoLocal, listaProcedimentos, listaVariaveis, Procedimento e Programa. Foi ainda criada uma classe Erro, relacionada ao tratamento de exceções.

1.1. classe Programa:

Um programa agora é visto como um comando global (facultativo) e uma lista de procedimentos. A classe programa encapsula essa visão.

```
private:
Comando * cmd_global;
ListaProcedimentos * procedimentos;
public:
Programa(Comando * g, ListaProcedimentos * p);
 Programa();
void Executa(Contexto& C);
```

Algoritmo 1: class Programa

1.2. classe Procedimento:

Procedimentos são formados por um identificador (nome do processo), parametros (facultativos), ComandoLocal (facultativo) e uma lista de comandos.

1.3. classe Contexto:

A classe contexto guarda o estado do programa, bem como sua estrutura. Ela contém os procedimentos do programa (procedimentos_do_prog), as variáveis globais (variaveis_globais) e variáveis locais, armazenadas na pilha de ativação (pilha_chamada).

2. Algoritmos

Das novas funções, apresentaremos aqui as fundamentais: Programa::Executa, Contexto::AdicionaVariavel, Contexto::AdicionaRA e Contexto::RemoveRA.

```
private:
std::string * nome_processo;
ListaVariaveis * parametros;
Comando * cmd_local;
ListaComandos * lista_cmds;
public:
Procedimento( std::string * nome_processo, ListaVariaveis * parametros, Comando *
cmd_local, ListaComandos * lista_cmds);
 Procedimento();
std::string obtemNome();
ListaVariaveis * obtemParametros();
Comando * obtemLocal();
ListaComandos * obtemComandos();
```

Algoritmo 2: class Procedimento

```
private:
Procedimentos * procedimentos_do_prog;
Variaveis variaveis_globais;
Pilha pilha_chamada;
public:
Contexto();
 Contexto();
double obtemVariavel(char nomeVar);
void defineVariavel(char nomeVar, double valor);
void adicionaVariavel(char nomeVar, double valor);
void adicionaRA();
void removeRA();
Procedimento * obtemProcedimento(std::string nome_procedimento);
void defineProcedimentos(Procedimentos * procedimentos_do_prog);
```

Algoritmo 3: class Contexto

2.1. void Programa::Executa(Contexto& C)

Primeira função da interpretação. Adiciona ao contexto a Lista de procedimentos, para futuras chamadas pela função call. Interpreta o comandoGlobal e faz um call do processo main.

2.2. void Contexto::AdicionaRA()

Função que faz um ”push”da pilha de ativação. Cada vez que um comando call é feito essa função é chamada para empilhar um novo espaço para as variáveis locais.

2.3. void Contexto::RemoveRA()

Faz um ”pop”da pilha de ativação. Executada ao final de cada chamada de procedimento, para remover o espaço de variáveis usado pelo processo anterior.

2.4. void Contexto::AdicionaVariavel(char nomevar, double valor)

Função chamada pelos comandos global e local, para criar, respectivamente, variáveis globais e locais. As variáveis globais são criadas porque a função é chamada com uma pilha de ativação vazia. Quando temos uma chamada de procedimento as variáveis criadas são colocadas na pilha referente a esse procedimento em execução, ouse ja, no topo da pilha.

3. IMPLEMENTAÇÃO

Cada classe está implementada em uma dupla de *<nome da classe>.h* e *<nome da classe>.cpp*. Há ainda um arquivo *parser.yy* usado pelo analisador sintático bison, responsável por produzir a partir desse os arquivos *parser.h* e *parser.cpp*. Há ainda os arquivos *scanner.ll* e *scanner.h* relativos ao análisador léxico Flex, que gera a partir desses o arquivo *scanner.cpp*.

Encapsulando o parser gerado pela dupla Bison/Flex, há a classe Driver. Essa classe é responsável por instanciar os analisadores léxico e sintático, conectá-los, alimentá-los com a entrada ("Código Fonte"), armazenar o resultado do processo de parsing e tratar erros de parsing encontrados no processo. A classe fornece os métodos: parse_stream, parse_string e parse_file, que iniciam o processo de parsing através de um stream, uma string ou um arquivo, respectivamente.

4. Compilação

O programa deve ser compilado pelo compilador GCC através de um makefile na pasta raiz.

5. Execução

O programa deve receber por parâmetro o nome do arquivo-fonte cujo código deve ser interpretado.

5.1. Formato da entrada

A entrada para o programa é um código fonte na sintaxe da linguagem tiny. A gramática da linguagem se encontra num arquivo junto ao código fonte do interpretado. Um exemplo de programa válido seria:

```
proc main()
  local a, b, c;
  read(a);
  read(b);

  while( a > 0 ) do
      c := c + b;
      a := a - 1;
  endw;

  writeVar(c);
  writeln;
endproc

endp
```

Algoritmo 4: Programa válido em tiny

5.2. Formato da saída

A saída do interpretador tiny será a saída gerada pelo programa fonte interpretado, ou mensagens de erro e de requisição de entrada, em caso de erro ou interpretação de um programa fornecido pela entrada padrão, respectivamente.

Referências

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company.