Manual do pacote fornecido para MC011

João Paulo Porto e Guido Araújo

1s2006

Conteúdo

1 Introdução

A idéia da disciplina MC011 é a construção de um compilador para a linguagem MiniJava (ver [1]). O compilador de MiniJava, mesmo considerando a simplicidade da lingüagem, pode ser bem complexo de ser implementado. Em especial, a parte descrita nos capítulo 6, 7 e 8 é bem complicada de ser entendida.

Visando facilitar o trabalho dos alunos, um pacote das classes descritas nos capítulos citados está sendo fornecida. Além disso, está sendo fornecida uma implementação do algorítmo do Appel para alocação de registradores por coloração de grafos.

Para utilizar o pacote fornecido, é necessário o uso do JDK 1.5 ou superior.

2 Pacote *assem*

2.1 `public abstract class` Instr

2.1.1 Descrição

Esta classe representa uma instrução em código assembly *independentemente de arquitetura*. A string que representa a instrução deve ser o mnemônico de uma instrução válida na arquitetura alvo. Para uma descrição desta string, veja [1].

2.1.2 `public abstract` util.List<temp.Temp> use()

Retorna uma lista com todos os temporários utilizados por esta instrução.

2.1.3 `public abstract` util.List<temp.Temp> def()

Retorna uma lista com todos os temporários definidos por esta instrução.

2.1.4 `public abstract` assem.Targets jump()

Retorna uma lista com todos os destinos desta instrução (válido somente para instruções de salto).

2.1.5 `public` java.lang.String format(temp.TempMap map)

Utiliza o mapeamente representado por `map` para converter o nome dos temporários utilizados pela instrução em nomes de registradores reais.

2.2 `public class` LABEL `extends` assem.Instr

2.2.1 Descrição

Representa uma instrução de *label* na arquitetura alvo.

2.2.2 `public` LABEL(java.lang.String text, temp.Label lab)

Cria uma nova instrução de label, cujo texto em assembly é representado por `text`. Esta instrução representa o nó `lab` na representação intermediária.

2.2.3 `public` temp.Label label

O nó da representação intermediária que esta instrução representa.

2.3 `public class` MOVE `extends` assem.Instr

2.3.1 Descrição

Representa a movimentação de dados entre dois temporários/registradores na arquitetura alvo. Esta intrução especial é necessária devido ao tratamento especial dado aos 'moves' pelo alocador de registradores.

2.3.2 `public` MOVE(java.lang.String text, temp.Temp dest, temp.Temp src)

Cria uma nova instrução de movimentação de dados, sendo que o destino da operação é `dest` e a origem `src`.

2.4 `public class` OPER `extends` assem.Instr

2.4.1 Descrição

Representa uma operação qualquer na arquitetura alvo. O sentido da operação é dado pela string `assem`, herdada de assem.Instr.

2.4.2 `public` OPER(java.lang.String assem, util.List<Temp> dest, util.List<Temp> src, util.List<Label> targets)

Constrói uma instrução com formato dado por `assem`, sendo que os temporários definidos e utilizados são indicados, respectivamente, por `dest` e `src`. `target` informa quais os alvos desta operação (ou seja, para onde ela pode desviar a execução do programa).

2.4.3 `public` OPER(java.lang.String assem, util.List<Temp> dest, util.List<Temp> src)

Constrói uma operação que não desvia o fluxo de execução. Ver 2.4.2 para detalhes dos parâmetros.

2.5 `public class` Targets

2.5.1 Descrição

Representa um conjunto de alvos de instruções que podem desviar o fluxo de execução do programa.

2.5.2 `public` Targets(util.List<Label> targets)

Cria um objeto representando os destinos dados por `targets`.

2.5.3 `public` util.List<Label> labels

Uma lista com os nós da representação intermediária que são representados por este objeto.

3 Pacote *cannon*

3.1 Descrição

Este pacote deve ser considerado como uma caixa preta. Basicamente, este módulo do compilador é responsável por gerar a representação intermediária canônica (para maiores informação, ver [1]). Para transformar a IR, é necessário utilizar a seguinte linha

```
cannon.TraceSchedule ts = new cannon.TraceSchedule(new
cannon.BasicBlocks(cannon.Canon.linearize(fragmento)));,
```

onde, `fragmento` é o fragmento da IR que precisa ser transformado em canônico.

4 Pacote *errors*

4.1 `public interface` ErrorEchoer

4.1.1 Descrição

Compiladores precisam, algumas vezes, imprimir mensagens de erro. Esta interface oferece métodos que são úteis para a geração de mensagens de erro informativas.

4.1.2 `public void` Print(java.lang.Object[] msg)

Imprime a mensagem `msg` sem informação sobre a posição do compilador no momento. Muito útil para imprimir mensagens de depuração.

4.1.3 `public void` Error(syntaxtree.Absyn absyn, java.lang.Object[] msg)

Exibe a mensagem de erro `msg` para o usuário. Esta mensagem ocorreu enquanto o compilador estava no nó `absyn` da AST. O fato deste método ter sido invocado indica que algo está errado com a entrada do usuário. A compilação não deve prosseguir além da fase atual.

4.1.4 `public void` Warning(syntaxtree.Absyn absyn, java.lang.Object[] msg)

Exibe a mensagem de aviso `msg` para o usuário. Esta mensagem ocorreu enquanto o compilador estava no nó `absyn` da AST. A compilação pode prosseguir, mas o resultado obtido pode não ser o esperado pelo usuário.

4.1.5 `public int` ErrorCount()

Informa quantos erros ocorreram desde a inicialização do objeto até o momento da invocação do método.

4.1.6 `public int` WarningCount()

Informa quantos avisos ocorreram desde a inicialização do objeto até o momento da invocação do método.

4.1.7 `public void` Reset()

Reinicializa os contadores de erros e avisos.

5 Pacote *flow_graph*

5.1 `public abstract class` FlowGraph `extends` graph.Graph

5.1.1 Descrição

Um grafo de fluxo é um grafo dirigido onde cada aresta indica um possível fluxo de controle no programa. Cada nó pode utilizar um grupo de temporários e definir outro (não necessariamente disjuntos). Além disso, é importante saber se o nó representa uma instrução de MOVE (essa informação é utilizada no alocador de registradores).

5.1.2 `public abstract` List<Temp> def(Node node)

Retorna o conjunto de temporários definidos por esta instrução.

5.1.3 `public abstract` List<Temp> use(Node node)

Retorna o conjunto de temporários usados por esta instrução.

5.1.4 `public abstract boolean` isMove(Node node)

Retorna se `node` é uma instrução de MOVE. Se assim o for, e $use = def$, então esta instrução pode ser deletada.

5.1.5 `public void` show(java.io.PrintStream out)

Imprime uma representação textual do grafo em `out`.

5.2 `public class` AssemFlowGraph `extends` flow_graph.FlowGraph

5.2.1 Descrição

Uma implementação do grafo de fluxo de controle para o compilador *MiniJava*.

5.2.2 `public` AssemFlowGraph(util.List<assem.Instr> list)

Constrói o grafo de fluxo de controle para a lista de instruções `list`. Os nós necessários são criados, bem como as devidas arestas. Ao término deste método, o grafo pode ser utilizado pelo alocador de registradores.

5.2.3 `private void` buildGraph(util.List<assem.Instr> ilist)

Este método deve ser implementado por vocês. Nele o grafo é efetivamente criado. Esta criação deve seguir alguns passos:

1. Criação dos nós (um para cada instrução)
2. Criação das arestas. Lembrem-se que instruções *fall-through*($instrucao.jumps() = null$) devem ser conectadas. Note, também, que os saltos contém apenas o *rótulo* do destino, e não a *instrução* destino. Sem esta instrução NÃO É possível localizar facilmente o nó destino do salto.

6 Pacote *frame*

6.1 `public abstract class` Access

6.1.1 Descrição

Representa a maneira de acessar um dado na arquitetura alvo. Esta abstração é necessária pois os valores podem estar na memória ou em registradores. Veja [1] para maiores detalhes.

6.1.2 `public abstract` tree.Exp exp(tree.Exp framePtr)

Retorna a árvore de expressão que é necessária para acessar o dado representado por este objeto. `framePtr` representa a expressão utilizada para acessar o *frame pointer* na arquitetura alvo.

6.2 `public abstract class` Frame `implements` temp.TempMap

6.2.1 Descrição

Representa um *frame* na arquitetura alvo. Como este *layout* varia de arquitetura para arquitetura, é necessário que haja uma implementação desta classe para cada alvo desejado. Com o pacote, é fornecida uma implementação desta classe para arquitetura x86.

6.2.2 `public abstract` Frame newFrame(temp.Label name, util.List¡java.lang.Boolean¿ formals)

Cria um novo frame para o procedimento de nome `name` que é aninhado com o frame que recebeu a invocação do método. Este novo procedimento recebe os parâmetros informados em `formals`. Cada item da lista de parâmetros deve valer *true* se o parâmetro escapa (é acessado em um nível diferente do declarado), e *false* caso contrário. Em *MiniJava*, nem os parâmetros nem as variáveis locais escapam.

6.2.3 `public` temp.Label name

O nome do procedimento dono deste objeto.

6.2.4 `public` util.List<frame.Access> formals

A lista de parâmetros deste frame. Cada elemento da lista contém um objeto do tipo frame.Access que gera a árvore de expressões correta para que qualquer parâmetro seja acessado.

6.2.5 `public abstract` frame.Access allocLocal(boolean escapes)

Como o nome indica, aloca uma variável local. `escapes` indica se a variável é acessada em um nível aninhado com o que declara a variável. Conforme dito anteriormente, nenhuma variável escapa em *MiniJava*. Assim, este parâmetro deve valer sempre *false*.

6.2.6 `public abstract int` wordsize()

Retorna o tamanho do inteiro da arquitetura alvo.

6.2.7 `public abstract` temp.Temp FP()

Retorna o temporário que representa o registrador utilizado como *frame pointer*.

6.2.8 `public abstract` tree.Exp externalCall(java.lang.String s, util.List<tree.Exp> args)

Esta rotina é utilizada para fazer chamadas a procedimentos externos ao programa *MiniJava*. Por exemplo, o operador `new` da linguagem precisa alocar memória, mas *MiniJava* não contém recursos para alocar memória. Assim, é necessário chamar uma função escrita em outra linguagem que consiga fazer a alocação de memória.

6.2.9 `public abstract` temp.Temp RV()

Retorna o temporário que representa o registrador utilizado para retornar valores das funções.

6.2.10 `public abstract` tree.Stm procEntryExit1(tree.Exp body)

A partir da expressão `body`, que representa o corpo de um procedimento, gera uma árvore com código necessário para salvar os registradores *callee save* e restaurá-los após o termino da função.

6.2.11 `public abstract` util.List<assem.Instr> procEntryExit2(util.List<assem.Instr> body)

Insere uma instrução de *sink* após a última instrução de `body`. Isso é feito para que os registradores de retorno, *stack pointer* e, caso exista, *base pointer* tenham um valor bem definido ao término da função.

6.2.12 `public abstract` frame.Proc procEntryExit3(util.List<assem.Instr> body)

Gera as instruções de prologo e epílogo necessárias para a função `body`.

6.2.13 `public abstract` util.List<assem.Instr> codegen(util.List<tree.Stm> body)

Faz a seleção de instruções para o procedimento representado por `body`.

6.2.14 `public abstract` util.List<temp.Temp> registers()

Retorna uma lista com todos os registradores da arquitetura alvo.

6.3 `public abstract class` Proc

6.3.1 Descrição

Representa um procedimento na arquitetura alvo, contendo todas as informações necessárias para a geração de código do procedimento.

6.3.2 `public` frame.Proc next

Ponteiro para o próximo procedimento na lista de procedimentos do programa.

6.3.3 `public abstract` java.lang.String getHeader()

Retorna uma string em *assembly* da arquitetura alvo que representa o cabeçalho deste procedimento. E.g., em x86, o cabeçalho de uma função é o nome da mesma seguida por um ':'.

6.3.4 `public abstract` util.List<assem.Instr> getBody()

Retorna a lista de instruções do procedimento, sem contar o cabeçalho, o término, o prólogo e o epílogo da função.

6.3.5 `public abstract` util.List<assem.Instr> getPrologue()

Retorna uma lista com as instruções utilizadas para *inicializar* o procedimento

6.3.6 `public abstract` java.lang.String getFooter()

Uma string que vai ser colocada como comentário no código assembly para indicar o fim do procedimento.

6.3.7 `public abstract` util.List<assem.Instr> getEpilogue()

Retorna uma lista com as instruções utilizadas para *finalizar* o procedimento

6.3.8 `public abstract void` print(java.io.PrintStream out, temp.TempMap map)

Utilizando `map` para mapear os nomes dos temporários para registradores reais, emite o procedimento em `out`.

7 Pacote *graph*

7.1 `public class` Graph

7.1.1 Descrição

Uma implementação de grafo direcionado. Muito útil em inúmeras parte do compilador.

7.1.2 `public` util.List<graph.Node> nodes()

Retorna uma lista com todos os vértices deste grafo

7.1.3 `public` graph.Node newNode()

Cria um novo nó para este grafo. Note que um nó criado em um grafo NÃO PODE ser utilizado em outro.

7.1.4 `public void` addEdge(graph.Node from, graph.Node to)

Cria a aresta <from,to> neste grafo. Se a aresta já existir, não faz nada.

7.1.5 `public void` rmEdge(graph.Node from, graph.Node to)

Remove a aresta <from,to> deste grafo. Se a aresta não existir, lança uma java.lang.Error.

7.1.6 `public void` show(java.io.PrintStream out)

Imprime, em `out`, o grafo. Útil para depuração.

7.2 `public class` Node

7.2.1 Descrição

Implementa o vértice utilizado em graph.Graph.

7.2.2 `public` Node(Graph g)

Cria um novo vértice para o grafo `g`.

7.2.3 `public` util.List<Node> succ()

Retorna a lista com os nós v sucessores deste nó u (ou seja, todos os nós para os quais existe <u,v> no grafo).

7.2.4 `public` util.List<Node> pred()

Retorna a lista com os nós u predecessores deste nó v (ou seja, todos os nós para os quais existe <u,v> no grafo).

7.2.5 `public` util.List<Node> adj()

Retorna uma lista contendo todos os nós que são sucessores e/ou predecessores deste nó.

7.2.6 `public int` inDegree()

Retorna qual o grau de entrada deste vértice.

7.2.7 `public int` outDegree()

Retorna qual o grau de saida deste vértice.

7.2.8 `public int` degree()

Retorna qual o grau deste vértice (grau de entrada mais grau de saída).

7.2.9 `public boolean` goesTo(graph.Node n)

Retorna a existência de uma aresta entre este nó u e n (ou seja, retorna *true* se existe a aresta <u,n>)

7.2.10 `public boolean` comesFrom(graph.Node n)

Retorna a existência de uma aresta entre este nó u e n (ou seja, retorna *true* se existe a aresta <n,u>)

7.2.11 `public boolean` adj(graph.Node n)

Retorna a existência de alguma aresta entre este nó u e n (ou seja, retorna *true* se existe alguma das arestas <n,u> e <u,n>)

8 Pacotes *minijava.**

8.1 Descrição

Estes pacotes, em tese, não deveriam ser incluidos no pacote que está sendo distribuído. Estes pacotes são gerados automaticamente pelo SableCC (gerador do *parser* e do *lexer*) e foram incluidos apenas para ser possível que vocês compilem o pacote, já que o conversor da AST (veja 17) precisa dessas classes.

9 Pacote *reg_alloc*

9.1 `class` Edge

9.1.1 Descrição

Implementa uma aresta de interferência entre os temporários utilizados no programa (e não nas instruções do mesmo).

Não é possível instanciar objetos desta classe diretamente. A única função de interface desta classe é `public static` reg_alloc.Edge getEdge(graph.Node u, graph.Node v) (veja 9.1.2).

9.1.2 public static Edge getEdge(Node u, Node v)

Dados os vértice u e v, esta função verifica se já existe uma aresta entre este vértices. Caso exista, a aresta existente é retornada. Caso contrário, uma nova aresta é cria e retornada.

9.2 `abstract public class` InterferenceGraph `extends` graph.Graph

9.2.1 Descrição

Classe base para implementação do grafo de interferência utilizado pelo alocador de registradores.

9.2.2 `abstract public` graph.Node tnode(temp.Temp temp)

Dado um temporário *temp*, esta função retorna o vértice correspondente ao temporário no grafo de interferência.

9.2.3 `abstract public` temp.Temp gtemp(graph.Node node)

Dado um vértice *node* do grafo, esta função retorna o temporário que *node* representa.

9.2.4 `abstract public` MoveList moves()

Retorna uma lista com todas as instruções de movimentação de dados entre temporários encontradas no programa.

9.2.5 `public int` spillCost(Node node)

Função de calcula de custo de *spill* do nó *node*. Esta implementação é bem simples e sempre retorna 1. Para melhorar a eficiência do alocador, reimplementar esta função na classe que derivar dela.

9.3 `public class` Liveness `extends` reg_alloc.InterferenceGraph

9.3.1 Descrição

Implementa a análise de fluxo de dados de *liveness*, necessária para a alocação de registradores por coloração de grafos. Esta classe implementa o algorítmo descrito no capítulo 10 de [1].

9.3.2 `public void` addEdge(graph.Node src, graph.Node dst)

Adiciona a aresta <*src*,*dst*> caso ela não exista.

9.3.3 `public void` show(java.io.PrintStream out)

Imprime este grafo de *liveness* em *out*. Muito útil para depuração.

9.3.4 `public` Liveness(flow_graph.FlowGraph cfg)

Cria a informação de *liveness* para o programa representado por *cfg*.

9.3.5 `public void` dump(java.io.PrintStream outStream)

Exibe, em *outStream*, os conjuntos *In*, *Out*, *Gen* e *Kill*. Muito útil para depuração.

9.3.6 `private void` computeDFA()

Função mais importante desta classe. É nela que vocês devem implementar o algorítmo de cálculo da informação de *liveness*. Estudem bem o algorítmo dado em sala antes de começar o trabalho neste método.

9.4 `public class` MoveList

9.4.1 Descrição

Lista ligada simples que contém todos as instruções MOVE do procedimento que está sendo analisado pela análise de *liveness*.

9.4.2 `public` graph.Node src

Origem da operação de MOVE.

9.4.3 `public` graph.Node dst

Destino da operação de MOVE.

9.4.4 `public` MoveList tail

Continuação da lista ligada de MOVE's; ou *null*, se este for o último nó da lista.

9.4.5 `public` MoveList(graph.Node s, graph.Node d, MoveList t)

Cria um nó na lista de MOVE's cuja origem é *s*, o destino é *d*. *t* contém a referência para a continuação da lista.

9.5 `public class` RegAlloc `implements` TempMap

9.5.1 Descrição

Outra classe que deve ser utilizada como caixa preta. Seria muito interessante para vocês o estudo dos métodos internos desta classe.

9.5.2 `public` RegAlloc(frame.Frame f, util.List<assem.Instr> i)

Cria o alocador de registradores para a função cujas instruções estão em *i*, e cujo frame é *f*. Além disso, este método já aloca os registradores. Assim, quando a chamada do construtor retorna, o objeto criado já pode ser utilizado para emitir as instruções.

10 Pacote *semant*

10.1 `public class` ClassInfo

10.1.1 Descrição

Esta classe contém todas as informações semânticas necessárias para implementação de um compilador funcional de *MiniJava*.

10.1.2 `public` temp.Label vtable

Label que representa o nome da *virtual table* deste objeto

10.1.3 `public` symbol.Symbol name

Símbolo que representa o nome desta classe.

10.1.4 `public` semant.ClassInfo base

Ponteiro para a informação da classe base desta classe.

10.1.5 `public` java.util.Hashtable<symbol.Symbol, semant.VarInfo> attributes

Esta tabela contém a informação sobre todos os atributos da classe que este objeto representa.

10.1.6 `public` java.util.Hashtable<symbol.Symbol, semant.MethodInfo> methods

Esta tabela contém a informação sobre todos os métodos da classe que este objeto representa.

10.1.7 `public` java.util.Vector<symbol.Symbol> attributesOrder

Este vetor é de uso interno da classe. A alteração de qualquer um dos valores dele pode resultar em bugs. Ele é utilizado para calcular o offset de um atributo dentro da classe.

10.1.8 `public` java.util.Vector<Symbol> vtableIndex

Este vetor é de uso interno da classe. A alteração de qualquer um dos valores dele pode resultar em bugs. Ele é utilizado para calcular o offset de um método dentro da *vtable* da classe.

10.1.9 `public` ClassInfo(symbol.Symbol n)

Cria um objeto para representar uma classe cujo nome é *n*, e que não deriva de nenhuma outra classe do programa que está sendo compilado.

10.1.10 `public` ClassInfo(symbol.Symbol n, semant.ClassInfo b)

Cria um objeto para representar uma classe cujo nome é *n* que deriva da classe *b* que deve ser declarada no programa que está sendo compilado.

10.1.11 `public boolean` addAttribute(semant.VarInfo var)

Adiciona o atributo representado por *var* à lista de atributos desta classe. Note que atributos duplicados (e.g., uma classe derivada que redeclara um campo declarado na classe pai) não são adicionados à lista.

10.1.12 `public int` getAttributeOffset(symbol.Symbol name)

Obtém o deslocamento do atributo cujo nome é *name* nos objetos da classe que este objeto represente.

10.1.13 `public int` getMethodOffset(symbol.Symbol name)

Obtém o deslocamento do método cujo nome é *name* na *vtable* dos objetos que são instâncias da classe que este objeto representa.

10.1.14 `public boolean` addMethod(semant.MethodInfo method)

Adiciona o método representado por *method* à lista de métodos desta classe. Note que nomes de métodos duplicados (e.g., uma classe derivada que reimplementa um método declarado na classe pai) não precisa ser adicionado à lista.

10.2 `public class` Env

10.2.1 Descrição

Esta classe armazena duas informações muito úteis durante a análise semântica do programa: o objeto que é utilizado para reportar os erros e a tabela com as classes declaradas.

10.2.2 `public` error.ErrorEchoer err

Objeto utilizado para exibir as mensagens de erro do compilador.

10.2.3 `public` symbol.Table¡semant.ClassInfo¿ classes

A tabela com todas as classes do programa que está sendo compilado.

10.2.4 `public` Env(error.ErrorEchoer e)

Cria o objeto que manterá as informações sobre o ambiente da compilação, e que utilizará *e* para exibir os erros encontrados na compilação.

10.3 `public class` MethodInfo

10.3.1 Descrição

Objetos que são instâncias desta classe contém todas as informações necessárias para a compilação de programas escritos em *MiniJava*.

10.3.2 `public` syntaxtree.Type type

Utilizado para representar o tipo de retorno da método que este objeto representa.

10.3.3 `public` symbol.Symbol name

O símbolo que representa o nome do método que este objeto representa.

10.3.4 `public` symbol.Symbol parent

O nome da classe que declarou o método que este objeto representa.

10.3.5 `public` util.List<semant.VarInfo> formals

Lista com as informações necessárias dos parâmetros do método que este objeto representa.

10.3.6 `public` util.List<semant.VarInfo> locals

Lista com as informações necessárias das variáveis locais do método que este objeto representa.

10.3.7 `public` java.util.Hashtable<symbol.Symbol, semant.VarInfo> formalsTable

Tabela utilizada para agilizar o acesso por nome às informações dos parâmetros do método que este objeto representa.

10.3.8 `public` java.util.Hashtable<symbol.Symbol, semant.VarInfo> localsTable

Tabela utilizada para agilizar o acesso por nome às informações das variáveis locais do método que este objeto representa.

10.3.9 `public` frame.Access thisPtr

Este campo é utilizado para acessar o ponteiro *this* do método que este objeto representa.

10.3.10 `public` frame.Frame frame

Este frame é a informação dependente de máquina sobre o método representado por este objeto.

10.3.11 `public` java.lang.String decorateName()

Obtém o nome 'decorado' do método representado por esta classe.

10.3.12 `public` MethodInfo(syntaxtree.Type t, symbol.Symbol n, symbol.Symbol p)

Constrói um objeto para representar um método de nome *n*, cujo tipo de retorno é representado por *t* e que foi declarado na classe *p*.

10.3.13 `public boolean` addFormal(semant.VarInfo formal)

Adiciona *formal* à lista de argumentos deste método.

10.3.14 `public boolean` addLocal(semantVarInfo local)

Adiciona *local* à lista de variáveis locais deste método.

10.4 `public class` VarInfo

10.4.1 Descrição

Objeto utilizado para representar as informções das variáveis e parâmetros necessárias para o compilador.

10.4.2 `public` syntaxtree.Type type

Objeto que representa o tipo da variável representada por este objeto.

10.4.3 `public` symbol.Symbol name

Símbolo que representa o nome da variável representada por este objeto.

10.4.4 `public` frame.Access access

Objeto dependente de arquitetura utilizado para acessar a variável representada por este objeto.

10.4.5 `public` VarInfo(syntaxtree.Type t, symbol.Symbol s)

Cria este um objeto desta classe para representar a variável de nome *s* e tipo *t*.

11 Pacote *symbol*

11.1 `public class` Symbol

11.1.1 Descrição

Em compiladores, é muito freqüente a necessidade de comparação de duas cadeias de caracteres. Para evitar a necessidade da comparação de bytes sempre que for necessário decidir pela igualdade de duas cadeias, foi criada esta classe. Dentro do compilador, se duas strings forem iguais, elas irão compartilhar o mesmo objeto *symbol.Symbol*. Desta forma, para determinação de igualdade de cadeias basta fazer uma comparação de objetos, que é extremamente rápida.

11.1.2 `public static` symbol.Symbol symbol(java.lang.String n)

Este método é a interface da classe com o mundo externo. Dada uma string *n*, este método procura pela existência de alguma instância da classe symbol que represente *n*. Se houver tal objeto, ele é retornado. Caso contrário, um novo objeto é criado e retornado.

11.2 `public class` Table<B>

11.2.1 Descrição

Esta classe é utilizada como uma tabela de símbolos sempre que uma for necessária (e.g., para manter uma tabela com as classes do programa). Esta tabela foi implementada de forma a suportar escopos aninhados.

11.2.2 `public` Table()

Cria uma tabela de símbolos vazia.

11.2.3 `public boolean` put(symbol.Symbol key, B value)

Coloca o símbolo *key* na tabela, associando a ele o objeto *value*, retornando *true* caso *key* não esteja no escopo atual da tabela. Caso contrário, retorna *false*.

11.2.4 `public` B get(symbol.Symbol key)

Obtém a informação associada à chave *key*. Retorna `null` se *key* não estiver declarada em nenhum escopo da tabela atual.

11.2.5 `public void` beginScope()

Método invocado para informar à tabela de símbolos que um novo escopo aninhado começou.

11.2.6 `public void` endScope()

Método invocado para informar à tabela de símbolos que o escopo mais externo terminou.

11.2.7 `public` java.util.Enumeration¡symbol.Symbol¿ keys()

Retorna os símbolos declarados no escopo mais externo da tabela de símbolos.

12 Pacote *syntaxtree*

12.1 Descrição

Este pacote implementa a AST (*Abstract Syntax Tree*) descrita em [1]. Neste documento estão descritas apenas as modificações feitas às estruturas para tornar possível a compilação dos programas. Essas modificações foram mínimas e não afetaram o que está descrito no livro.

12.2 Modificações

1. Adição da classe `syntaxtree.Equal`. Sem ela, não era possível comparar a igualdade de variáveis em programas escritos em *MiniJava*.

2. Adição do campo `public` syntaxtree.Type type à classe `syntaxtree.Exp`. Sem esse campo a análise semântica do compilador torna-se um pouco complicada, e torna-se impossível fazer a compilação de programas orientados a objetos (como os programas escritos em *MiniJava*!).

13 Pacote *temp*

13.1 `public class` CombineMap `implements` TempMap

13.1.1 Descrição

O intuito deste mapa é juntar dois mapas de temporários em um.

13.1.2 `public` CombineMap(temp.TempMap m1, temp.TempMap m2)

Cria um objeto que vai tentar utilizar *m1* para obter o nome dos temporários. Caso *m1* não consiga dar o nome, será utilizado o mapa *m2*.

13.2 `public class` DefaultMap `implements` TempMap

13.2.1 Descrição

Mapa de temporários simples: dado um temporário t, retorna o próprio t.

13.3 `public class` Label

13.3.1 Descrição

Gerencia a criação de *Labels* automática no compilador. Garante que, sempre que solicitado, gera uma *label* única. Esta *label* é somente um rótulo no programa!

13.3.2 `public` Label(java.lang.String l)

Cria um objeto para representar a string *l*.

13.3.3 `public` Label()

Cria uma *label* única a partir de um conjunto de rótulos.

13.3.4 `public` Label(symbol.Symbol l)

Cria um objeto para representar o símbolo *l*.

13.4 `public class` Temp

13.4.1 Descrição

Objeto utilizado para representar um temporário (dado) no programa. Este temporários são os objetos atribuidos aos registradores físicos.

13.4.2 `public` Temp()

'Aloca' um temporário do banco de temporários que é (virtualmente) infinito.

13.5 `public interface` TempMap

13.5.1 Descrição

Esta interface é implementada por classes que pretendem dar nomes à temporários.

13.5.2 `public` java.lang.String tempMap(temp.Temp t)

Converte t para alguma string (e.g., o nome do registrador atribuido a t).

14 Pacote *translate*

14.1 Descrição

Outro módulo caixa preta. Este módulo implementa a tradução da AST para IR e é descrito no capítulo 7 de [1]. Para fazer esta tradução é necessário utilizar a sequinte linha

translate.Frag f = translate.Translate.translate(frame, env, program).

O resultado desta chamada é uma lista ligada com os 'fragmentos' do programa. Estes fragmentos podem ser *vtables* ou métodos.

15 Pacote *tree*

15.1 Descrição

Este pacote implementa a representação intermediária descrita em [1] da maneira como o autor a descreve. Apenas uma modificação foi necessária: havia uma classe chamada Exp (que é pai para todos os nós que são expressões na IR) e uma chamada EXP (que deriva de Stm e serve para converter uma expressão em uma sentença). Em Java, estas classes ficam em arquivos chamados Exp.java e EXP.java, respectivamente. Infelizmente, alguns sistemas de arquivos não diferenciam estes nomes. Para evitar este conflito, a classe EXP foi renomeada para EXPSTM (tendo sido declarada em EXPSTM.java).

Para obter a descrição das classes deste pacote, veja [1].

16 Pacote *util*

16.1 `public class` List<E>

16.1.1 Descrição

Implementa uma lista ligada simples.

16.1.2 `public` E head

O elemento do nó atual da lista.

16.1.3 `public` List<E> tail

Aponta para o próximo elemento da lista. Se valer *null*, este nó é o último da lista.

16.1.4 `public` List(E h, List<E> t)

Cria um novo nó da lista, sendo que a informação armazenada neste nó é *h*. *t* contém o resto da lista (caso *t* seja *null*, o nó criado é o último da lista).

Se *h* valer *null* será lançada uma instância de `java.lang.Error`.

16.1.5 `public int` size()

Retorna o tamanho da lista do nó atual até o último nó.

16.2 `public class` SimpleError `implements` ErrorEchoer

16.2.1 Descrição

Implementa um objeto simples de emissão de mensagens de erro.

16.2.2 `public` SimpleError(java.io.PrintStream e, java.langString s)

Cria um novo objeto para emissão de mensagens de erro que escreve em *e*. O arquivo que está sendo compilado no momento é *s*.

16.2.3 `public` SimpleError()

Cria um objeto de emissão de mensagens de erro que escreve na saida de erro padrão (`java.lang.System.err`). O nome do código fonte não é definido.

16.2.4 `public` SimpleError(java.lang.String s)

Cria um objeto de emissão de mensagens de erro que escreve na saida de erro padrão (`java.lang.System.err`). O nome do código fonte é *s*.

16.2.5 `public` SimpleError(PrintStream e)

Cria um novo objeto para emissão de mensagens de erro que escreve em *e*. O nome do arquivo que está sendo compilado atualmente não é definido.

17 Pacote *util.conversor*

18 Descrição

A AST descrita em [1] é diferente da AST gerada pelo SableCC, sendo a primeira bem mais simples que a última. Tendo isso em vista, este pacote foi escrito para facilitar a conversão entre as estruturas. Quem desejar utilizá-lo deve usar a seguinte linha:

syntaxtree.Program program = util.conversor.SyntaxTreeGenerator.convert(s),

onde `s` é o resultado retornado pelo parser gerado pelo SableCC.

19 Pacote *visitor*

19.1 Descrição

Este pacote declara as interfaces necessárias para implementação dos *visitors*, que nada mais são que classe que visitam os nós de uma estrutura de dados (no caso do compilador de *MiniJava*, estas estruturas são árvores). A interface `visitor.Visitor` declara métodos que visitam os nós das estruturas e não retornam valor. Já a interface `visitor.TypeVisitor` é utilizada no momento da verificação semântica do programa. Os nós que pretendem ser visitados pelos *visitors* devem implementar `visitor.Visitable`. Para maiores informações sobre os *visitors*, veja a seção 4.3 de [1].

20 Pacote *x86*

20.1 `public class` Codegen

20.1.1 Descrição

Esta classe deve ser implementada por vocês para fazer a seleção de instruções. Há inúmeros algorítmos de seleção de instruções. Para este módulo, escolham um e o implementem.

20.1.2 util.List<assem.Instr> codegen(tree.Stm s)

Este método é responsável por gerar as instruções para a árvore *s*.

20.1.3 util.List<assem.Instr> codegen(util.List<tree.Stm> body)

Este método é responsável por gerar as instruções para a floresta *body*.

20.2 `class` InFrame `extends` frame.Access

20.2.1 Descrição

Classe utilizada para abstrair o acesso a dados que serão armazenados na memória e não em registradores.

20.3 `class` InReg `extends` frame.Access

20.3.1 Descrição

Classe utilizada para abstrair o acesso a dados que são candidatos a serem armazenados em registradores.

Referências

[1] Andrew w. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2 edition, 1998.