MC910 – Sugestões de Implementação

Tabela de Símbolos

As regras de visibilidade da linguagem definem os seguintes níveis de visibilidade:

- Definições globais (variáveis e funções globais)
- Definições locais a uma função
- Definições locais a um comando composto

Os comandos compostos podem ser encaixados e por isso, a tabela de símbolos deve ser organizada como uma lista de escopos, onde cada escopo tem uma referência ao escopo mais externo (na verdade a tabela de símbolos será usada como uma pilha).
Um escopo por sua vez pode ser organizado como um mapa que associa um nome (da variável ou função) à respectiva descrição.
A descrição de uma variável deve conter

- Nome da variável
- Tipo
- Local ou global
- Endereço

A descrição de uma função

- Nome
- Tipo do valor de retorno
- Lista de parâmetros

Cada parâmetro deve ter

- Tipo
- Modo de passagem (valor ou referência)

Verificação Semântica

A verificação consiste em percorrer a representação intermediária do programa verificando se

- Todas as declarações estão em acordo com a definição da linguagem. Exemplos:
  - Não existem num mesmo escopo duas declarações de 'objetos' com o mesmo nome.
  - Os tipos usados são definidos corretamente.

  A verificação das declarações ainda os seguintes 'efeitos colaterais':
  - Inserção dos 'objetos' declarados na tabela de símbolos (no respectivo escopo).
  - Preenchimento do campo 'scope' nos objetos da representação intermediária (StatOp e derivados).
- Todo uso de 'objeto' num comando é consistente com a sua definição. Exemplos:
  - Numa operação diádica (soma, subtração, and, or), os tipos dos dois operandos devem ser compatíveis com a operação, de acordo com a definição da linguagem.

- Num comando condicional, o tipo de retorno da expressão condicional deve ser booleano e os comandos associados devem ser verificados e corretos.
- Numa chamada de função
    - A função deve ter sido definida (como função).
    - Cada parâmetro 'de chamada' deve ser compatível com o respectivo parâmetro na definição da função.

O uso do pattern Visitor

A representação intermediária fornecida foi preparada para o percurso através de um Visitor, que é adequado à verificação semântica. A verificação do programa, a partir de um objeto Programa pode ter a seguinte estrutura :

```
Object Visit(Program prog){
      // verificar as declarações feitas no programa
      LinkedList<Declaration> declarations = prog.getDeclarations();
      Boolean checkDeclarations = new Boolean(true);

      for(Declaration decl:Declarations) { // para cada declaração
            Boolean b = decl.accept(this);
           if(!b) checkDeclarations = new Boolean(false);
      }
      // verificar o 'corpo' (commandos) do programa.
      Boolean checkBody = (Boolean)prog.getBody().accept(this);

      // retornar true se declarações e corpo estiverem corretos
      return new Boolean(checkDeclarations && checkBody);
}
```

No exemplo acima, supõe-se que os métodos responsáveis pela verificação de cada declaração e cada comando, emitirão as eventuais mensagens de erro na saída padrão.

A verificação de um comando de atribuição poder ter a seguinte estrutura:

```
Object Visit(AssignOp assignOp){

      // verificar o primeiro operando(variável)
      Type op1Type = assignOp.getOperand1().accept(this);

      // verificar se o primeiro operando representa uma variável
      // (este pode ser uma variável simples ou um elemento de vetor
      boolean isVar = isVariable(assignOp.getOperand1());

      // verificar o segundo operando(expressão à direita do '=')
      Type op2Type = assignOp.getOperand2().accept(this);

      // verificar se os tipos são compatíveis para atribuição
      boolean compat = assignCompatible(opType1,opType2);

      return new Boolean(isVar && compat);
}
```

No exemplo acima

- o método auxiliar assignCompatible() seria o método responsável por verificar se os dois tipos são compatíveis para atribuição. Esse mesmo método pode ser usado para verificar a compatibilidade entre os parâmetros 'de definição' e os parâmetros 'de chamada' de uma função.
- O método auxiliar isVariable(Expression exp) verifica se a expressão se refere a uma variável.
- O método accept(Expression exp) do Visitor responsável pela verificação semântica, deve retornar um objeto Type. Esse método é chamado pelo método accept(Visitor v) da classe Expression (à qual pertencem operand1 e operand2 da classe AssignOp). Neste método pode se adotar a convenção de retornar null em caso de algum erro de semântica.

Geração de Código

A geração de código deve ser feita após a verificação semântica porque necessita da tabela de símbolos para acessar os tipos e endereços das variáveis usadas nos comandos e acessar os tipos de retorno e parâmetros das funções usadas nos comandos.

O pattern visitor também pode ser usado para a geração de código. A seguir são mostrados algumas sugestões para a geração de código. Os exemplos adotam a seguinte disciplina: o objeto retornado pelos métodos Visit() do gerador de código são da classe Integer e seu valor indica o endereço da última instrução gerada.

Geração de código para o programa

O quadro abaixo mostra a sugestão para a geração de código para o programa:

```
Object Visit(Program prog){

      Integer lastAddr = prog.getBody().accept(this);
      return lastAddr;
}
```

Esse código pode ser modificado para retornar o endereço inicial da função main(). A declaração da função main() pode ser obtida da tabela de símbolos, accessível através do método prog.getScope().

Geração de código para expressões

Uma expressão, representada pelas classes derivadas da classe abstrata Expression, ao ser executada deve deixar o valor resultante no topo da pilha. O exemplo abaixo mostra como pode ser a geração de código para os operadores diádicos usados em expressões.

```
Object Visit(DiadOp diadOp){
      // gerar código para o primeiro operando
      diadOp.getOperand1.accept(this);

      // gerar código para o Segundo operando
      diadOp.getOperand2.accept(this);

      // gerar código para a operação
      int addr;
      Switch(diadOp.getKind(){
            case(ADD_OP):  addr = vm.addInstr(ADD);  break;
            case(SUB_OP):  addr = vm.addInstr(SUB);  break;
            case(MULT_OP): addr = vm.addInstr(MULT); break;
            case(DIV_OP):  addr = vm.addInstr(DIV);  break;
            case(AND_OP):  addr = vm.addInstr(AND);  break;
            case(OR_OP):   addr = vm.addInstr(OR);   break;
            case(EQ_OP):   addr = vm.addInstr(EQ);   break;
            case(NE_OP):   addr = vm.addInstr(NE);   break;
            case(LT_OP):   addr = vm.addInstr(LT);   break;
            case(LE_OP):   addr = vm.addInstr(LE);   break;
            case(GT_OP):   addr = vm.addInstr(GT);   break;
            case(GE_OP):   addr = vm.addInstr(GE);   break;
      }
      return new Integer(addr);
}
```

Neste exemplo, vm é uma variável de classe que referencia a máquina virtual para a qual se pretende gerar o código.

Geração de código para o comando condicional

O exemplo abaixo mostra como pode ser a visita para a geração de código para o comando condicional, na sua versão 'completa' (existe o comando associado à 'parte else'). É necessário considerar o caso em que a 'parte else' não existe.

```
Object Visit(IfStat ifStat){
      // gerar o código para condição
      ifStat.getCondition().accept(this);

      // desviar se falso para o 'else'
      int addr1 = vm.addInstr(JUMPF, -99); //-99: endereço inválido.

      // gerar código para o comando 'then'
      ifStat.getThenPart().accept(this);
      int addr2 = vm.addInstr(JUMP,-99);

      // acertar o endereço do JUMPF
      vm.fixAddr(addr1,(addr2+1));

      // gerar código para o comando 'else' (supondo que é != null)
      int addr3 = ifStat.getElsePart().accept(this);

      // acertar o endereço do JUMP
      vm.fixAddr(addr2,(addr2+1));

      // retorna o endereço da última instrução
      return new Integer(addr3);
}
```