**Implementação da análise semântica: a terceira etapa na construção do compilador Marvel**

**Cláudio Lopes (peso ), Gildo Leonel (peso ), Sérgio Rossini (peso )**

**Departamento de Ciência da Computação (DCC)**

**Universidade Federal de Juiz de Fora (UFJF)**

**Campus Universitário – CEP 36036-330 – Juiz de Fora – MG**

**{cslopes, gildo.leonel, sergiorossini}@gmail.com**

**1. Visão geral**

A fase de análise semântica de um compilador conecta as definições das variáveis com sua utilização, verifica se cada expressão possui um tipo correto e traduz a análise sintática para uma representação mais simples visando à geração de código de máquina [Appel e Palsberg, 2002].

Dessa forma, pode-se dizer que o objetivo da análise semântica é trabalhar no nível de inter-relacionamento entre partes distintas do programa. As tarefas básicas desempenhada durante a análise semântica geralmente incluem a verificação de tipos, a verificação do fluxo de controle e a verificação da unicidade da declaração de variáveis.

A tradução do código de alto nível para o código do processador está associada a traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa [Ricarte, 2003].

Numa primeira etapa, a representação intermediária será gerada em forma de “fragmentos”, ou seja, blocos de comandos referentes a cada nó pertencente à árvore de sintaxe abstrata.

**1.1. Desenvolvimento do trabalho**

A construção do analisador semântico foi iniciada com a criação do módulo de análise semântica. Foram descritos os arquivos *headers* para as classes visitantes VerificadorEscopo, VerificadorTipos e VerificadorVariaveis, seguindo o padrão já estabelecido na etapa anterior como pode ser verificado no arquivo ImpressaoArvoreAbstrata.h.

A geração de código foi iniciada a partir do momento em que foram finalizadas as implementações das classes visitantes para verificação semântica. Dessa forma, foi criado o módulo CodigoIntermediario.

A implementação das classes seguiu exemplos disponíveis em [Bernardes, 2007] e anotações de aula. A ferramenta DevC++ e a linguagem C foram utilizadas para tal. Como suporte ao desenvolvimento e eventuais dificuldades com a linguagem foram utilizados [Kernighan e Ritchie], [Deshpande e Kakde, 2004] e [Mesquita, 1998].

**1.2. Módulos do analisador semântico**

O analisador semântico, como apresentado na seção 1, deve verificar as regras semânticas e iniciar a geração do código intermediário respectivo para cada nó da árvore de sintaxe abstrata. Dessa forma, os módulos AnalisadorSemantico e CodigoIntermediario foram adicionados ao projeto do compilador.

O módulo AnalisadorSemantico é composto por classes que implementam a verificação das regras semânticas de verificação de escopo, verificação de tipos e verificação de declaração de variáveis. As classes foram nomeadas respectivamente como VerificadorEscopo, VerificadorTipos e VerificadorVariaveis. A verificação do fluxo de controle não foi implementada nesse projeto pois a gramática analisada não apresenta construção desse tipo.

O módulo CodigoIntermediario é o início de um módulo maior. Nesse primeiro momento o módulo é responsável por gerar o código intermediário de cada nó pertencente à árvore de sintaxe abstrata criada durante a análise sintática, gerando uma árvore intermediária contendo o código gerado. Esse módulo possui ainda duas classes visitantes, de árvores diferentes. O visitante Tradutor, visita cada um dos elementos da árvore de sintaxe abstrata (ASA) para gerar o código intermediário respectivo. O visitante VisitorArvoreIntermediaria, percorre a árvore de código intermediário para efetuar sua impressão na tela.

As classes referentes à análise semântica estão representadas pela figura 1, onde também são mostradas as suas interdependências.

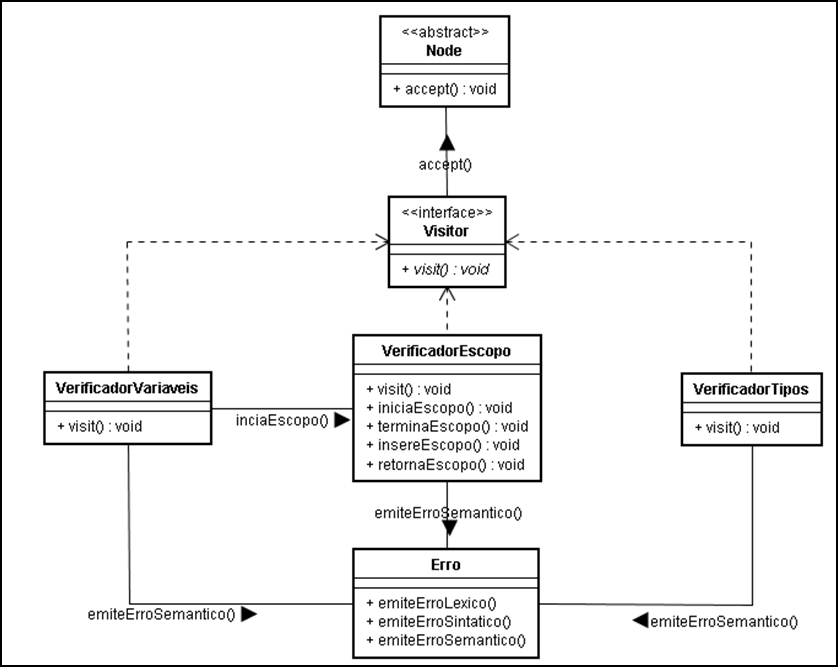


Figura 1. Interdependência entre os módulos da análise semântica.

**2. Analisador semântico**

As verificações das regras semânticas foram incluídas como forma de classes: VerificadorTipos, VerificadorVariaveis e VerificadorEscopo.

A classe VerificadorEscopo analisa a semântica do código FRA com relação ao nível de escopo, múltiplas declarações de variáveis, mesmo identificador usado em escopos mais internos. A verificação foi implementada na forma do padrão Visitor. A classe VerificadorEscopo aceita a declaração de uma variável no escopo analisado se a mesma já não tenha sido declarada no mesmo escopo ou em algum escopo inferior. Caso contrário, uma informação de erro indicando a linha em que houve o problema é encaminhada a saída de erro, utilizando o Gerenciador de Erro. Para a implementação do VerificadorEscopo a estrutura multimap do C++ foi utilizada, o que permitiu fácil inserção e busca de nós do tipo IdNode.

A responsabilidade do VerificadorTipos é de avaliar se os tipos em expressões, atribuições e passagem de parâmetros são compatíveis com respectivas operações. Para tal, foi especificada uma variável global chamada tipo, que é atualizada a todo o momento em que um método visitante é invocado. Essa variável é atualizada inicialmente por nós referentes a identificadores, números ou constantes. Dessa forma, o nó filho retorna ao nó pai o tipo de valor que ele representa: INTEGER e FLOAT nos casos de números ou CHAR no caso de um literal. A implementação de comparação entre os tipos seguiu exemplos disponíveis em [Vieira, 2007].

A seguir são listadas as regras semânticas definidas para o compilador Marvel.

**2.1. Regras semânticas**

As regras semânticas são associadas às produções gramaticais. O compilador Marvel utiliza a árvore de sintaxe abstrata que encapsula as produções gramaticais. Dessa forma, as regras semânticas são aqui demonstradas de acordo com cada um dos nós que representam a árvore. Os subitens são iniciados com a assinatura do construtor do nó, incluindo os outros elementos que o compõem e finalizados com as regras semânticas associadas.

**2.1.1.** *AddOpNode(int o, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornados por *en1* e *en2* são compatíveis entre si.
* Verifica se os tipos retornados por *en1* e *en2* são compatíveis com a operação de soma.
* Retorna o tipo de *en2*.

**2.1.2.** *ArrayNode(IdNode\* in, ExpressionNode\* en)*

* Verifica se o tipo retornado por *en* é compatível para acesso a um índice do array.
* Retorna o tipo de *in*.

**2.1.3.** *AssignNode(IdNode\* in, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornados por *en1* e *en2* são compatíveis entre si.
* Verifica se os tipos retornados por *en1* e *en2* são compatíveis com o tipo retornado por *in*.
* Retorna o tipo de *in*.

**2.1.4.** *BitwiseOpNode(int o, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornado por *en1* e *en2* são compatíveis entre si.
* Verifica se os tipos retornados por *en1* e *en2* são do tipo BOOLEAN.
* Retorna o tipo BOOLEAN.

**2.1.5.** *BoolOpNode(int o, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornado por *en1* e *en2* são compatíveis entre si.
* Verifica se os tipos retornados por *en1* e *en2* são do tipo BOOLEAN.
* Retorna o tipo BOOLEAN.

**2.1.6.** *ConstantNode(NameDeclNode\* ndn, ExpressionNode\* en)*

* Retorna o tipo de ConstantNode.
* Finaliza escopo.

**2.1.7.** *FragCallNode(IdNode\* in, ExpressionListNode\* eln)*

* Verifica se os tipos retornados por *in* e *eln* são compatíveis.
* Retorna o tipo de *in*.

**2.1.8.** *IdNode(int i, struct Registro \*reg)*

* Retorna o tipo de idNode.
* Verifica se idNode encontra-se no escopo atual.

**2.1.9.** *IfNode(ExpressionNode\* en, StatementNode\* sn1, StatementNode\* sn2)*

* Verifica se en retorna tipo BOOLEAN.

**2.1.10.** *LiteralNode(int l)*

* Retorna o tipo CHAR.

**2.1.11.** *ModifierNode(int m, int n)*

* Retorna o tipo definido por *m*

**2.1.12.** *MultOpNode(int o, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornados por *en1* e *en2* são compatíveis entre si.
* Verifica se os tipos retornados por *en1* e *en2* são do tipo INTEGER ou FLOAT.
* Retorna o tipo de *en2*.

**2.1.13.** *NegativeNode(ExpressionNode\* en)*

* Verifica se o tipo retornado por *en* é do tipo INTEGER, FLOAT ou NUM.
* Retorna o tipo de *en*.

**2.1.14.** *NotNode(ExpressionNode\* en)*

* Verifica se o tipo retornado por *en* é do tipo BOOLEAN.
* Retorna o tipo BOOLEAN.

**2.1.15.** *NumberNode(int n, struct Registro \*reg)*

* Retorna o tipo de numberNode.

**2.1.16.** *RelOpNode(int o, ExpressionNode\* en1, ExpressionNode\* en2)*

* Verifica se os tipos retornados por *en1* e *en2* são compatíveis entre si.
* Retorna o tipo BOOLEAN.

**2.1.17.** *WhileNode(ExpressionNode\* en, StatementNode\* sn)*

* Verifica se o tipo retornado por *en* é do tipo BOOLEAN.

**2.1.18.** *WriteNode(ExpressionListNode\* eln)*

* Verifica se *eln* é vazio.

**2.1.19.** *FragmentNode(StatementListNode\* sln)*

* Inicia a verificação de escopo dos elementos de *sln*.
* Finaliza o escopo.

**2.1.20.** *NameDeclNode(ModifierListNode\* mln, IdListNode\* iln)*

* Verifica se é declaração de um fragmento e inicia novo escopo.
* Verifica se é declaração de parâmetro de fragmento e insere no escopo.
* Verifica se os elementos idNode já foram declarados no escopo.

**2.1.21.** *ProgramNode(StatementListNode\* sln)*

* Finaliza escopo.

**3. Tabela de símbolos**

A tabela de símbolos de um compilador é utilizada para armazenar informações sobre os nomes declarados em um programa. A tabela de símbolos geralmente é pesquisada cada vez que um nome é encontrado no programa fonte. Alterações na tabela de símbolos são realizadas sempre que um novo nome ou nova informação sobre um nome já existente é obtida [Nicolletti, 2005].

Para disponibilizar as informações necessárias para a implementação do analisador semântico as estrutura Registro da tabela de símbolos e o nó IdNode foram atualizados e trabalhados em conjunto para essa finalidade. A estrutura Registro continuou mantendo as informações de índice do lexema no array de lexemas, o token do registro e um ponteiro para o próximo registro. A nova estrutura do nó IdNode pode ser verificada na listagem abaixo.

class IdNode : public ExpressionNode {

public:

int id; *// valor inteiro que representa o token*

ExpressionNode \*paiEscopo; *// pai do aninhamento ao qual o id pertence*

int ativo; *// representa idNode ativo(1) ou não(0)*

struct Registro \*registro; *// estrutura registro da tabela de símbolos*

int linha; *// linha no código em que o idNode está*

int tipo; *// inteiro que representa o tipo do idNode*

int escopo; *// inteiro que representa o escopo atual*

int offset; *// inteiro que representa o offset*

IdListNode \*parametros*;// lista de parâmetros no caso de ser um fragmento*

int tamanho; *// inteiro que representa o tamanho*

};

**4. Gerador de representação intermediária**

O código intermediário gerado pelo compilador é estruturado em forma de árvore. Para possibilitar a geração do código intermediário de maneira correta e eficiente, foi utilizado o padrão de projeto Visitor de forma similar à montagem da árvore de sintaxe abstrata [Lopes e outros, 2007].

A figura 2 representa de forma simplificada como a geração de código intermediário foi acrescentada ao projeto do compilador. A classe Tradutor implementa o *Visitor* de nós da ASA e tem acesso às definições dos nós que compõem a árvore intermediária através da diretiva #include "ArvoreIntermediaria.h". Ao percorrer a árvore de sintaxe abstrata o visitante gera uma lista de fragmentos de códigos intermediários através dos métodos *addFragmento(Fragmento \*frag)* e *addStm(Stm \*stm)*, utilizando as classes ListaDeFragmentos, Fragmento, StmList e Stm, definidas no arquivo CodigoIntermediario.h.

As classes Fragmento e Stm são abstratas. A classe Fragmento é utilizada como herança na definição das classes Procedimento, constanteLiteral, nomeReal e nomeInteiro também definidas no CodigoIntermediario.h e especificadas em [Bernardes, 2007]. Dessa forma, ao gerar um código intermediário durante o procedimento visit do Tradutor, o método addFragmento pode receber código intermediário referente a alguns nós da ASA, como FragmentNode, LiteralNode e ProgramNode. Stm também é uma classe abstrata e utilizada como herança na definição das classes que representam os nós da árvore de código intermediário que, por sua vez, representam nós do tipo Statement da árvore de sintaxe abstrata, ou seja, sentenças para resultados secundários e controle de fluxo. O código intermediário de nós do tipo Stm é gerado durante a invocação do método visit da classe Tradutor.

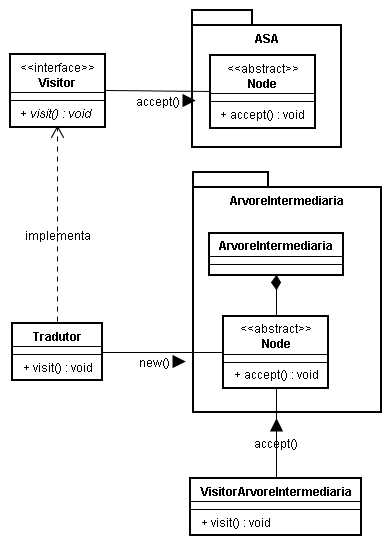


Figura 2. Geração de código intermediário.

Após a geração do código intermediário, é utilizado um novo visitante para a sua impressão. Essa responsabilidade foi delegada à classe VisitorArvoreIntermediaria.

**5. Gerenciador de erros semânticos**

O módulo de erro do compilador (Erro.h e Erro.cpp) foi atualizado, incluindo o método *emiteErroSemantico(int codigo, char \*string, int linha)*, passando a suportar situações em que são encontrados erros semânticos durante o processo de compilação.

O parâmetro codigo representa o código do erro ocorrido durante a análise. O ponteiro de char representa um texto adicional que pode ser enviado ao método de erro, sendo normalmente enviado o nome de uma operação ou nome de uma variável. O parâmetro linha representa o número da linha em que o erro encontrado ocorreu.

Foram incluídos novos códigos de erros para o gerenciador, passando a suportar os erros referentes à:

* Tipo não esperado na operação;
* Incompatibilidade entre tipos de uma expressão;
* Incompatibilidade de tipo indicado como índice de array;
* Tipos incompatíveis com o identificador durante atribuição;
* Expressão não booleana em teste condicional;
* Falta de expressão em instrução;
* Tipo incompatível a uma chamada de fragmento;
* Erro de variável não declarada no escopo analisado;
* Erro de variável já declarada no escopo analisado.

A estrutura do método de erro da análise semântica segue o padrão adotado no desenvolvimento do compilador durante as etapas anteriores. Foi utilizada a cláusula CASE, onde é realizada a decisão a partir do parâmetro int codigo do método e #define para cada uma das situações de erro.

**6. Resultados experimentais**

**6.1. Arquivo CODIGO.FRA**

**7. Conclusão**

Esta etapa do trabalho foi responsável por estruturar e implementar a análise semântica e geração do código intermediário do compilador Marvel para a linguagem FRAG. Dessa forma foi possível fazer com que códigos implementados para esta linguagem obedeçam a algumas regras semânticas propostas. São emitidas mensagens de erro nos casos em que as regras não sejam obedecidas.

A utilização da estrutura multimap, disponível no C++, foi utilizada para a implementação da verificação de escopo. Multimap é uma estrutura do tipo mapa, onde elementos são armazenados de forma associativa com um valor chave [Kohl, 2007]. Por possibilitar o armazenamento de estruturas diferentes com a mesma chave, multimap facilitou a verificação de escopo, onde mais de uma variável pôde ser declarada com mesmo identificador, em escopos diferentes.

A geração de código intermediário apresentou dificuldades para implementação devido à utilização de novos conceitos e manipulação de estrutura de dados e instruções de máquina abstrata. A utilização do padrão de projeto Visitor facilitou a construção da árvore de código intermediário e dessa forma foi de grande importância para a conclusão essa tarefa.

O compilador Marvel apresenta as funcionalidades básicas exigidas na especificação em [Bernardes, 2007] e funcionalidades extras podem ser adicionadas ao projeto em uma oportunidade seguinte.

**8. Referências bibliográficas**

Kernighan, B. e Ritchie, D. “The C Programming Language”, Prentice Hall Software Series, 2nd edition, capítulos 1, 2, 4, 5, 6 and 7.

Aho, A. e Ullman, J. (1995) Compiladores: Princípios, Técnicas e Ferramentas, Editora LTC, 1ª. Edição, capítulos 5, 6 e 7.

Menezes, P. (2000), Linguagens Formais e Autômatos, Editora Sagra Luzzato, 3ª. Edição, capítulo 3, subcapítulos 3.5 e 3.6.

Holzner, S. (2001). C++ Black Book, Editora Makron Books, 1ª. Edição, capítulos 5, 7 e 10.

Appel, A. W. e Palsberg, J (2002), Modern Compiler Implementation in Java, Cambridge University Press, 2nd. edition.

Ricarte, I. L. M. (2003), Programação de Sistemas: uma introdução, Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação, http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/progsist2003.html, acesso em nov/2007.

Deshpande, P. S. e Kakde, O.G., (2004), “C & Data Structures”, Charles River Media, 1st edition, capítulos 8, 10, 11, 12 e 15.

Nicolletti, P. S. (2005), “Compiladores – notas de aula”, Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, Departamento de Sistemas e Computação, http://www.dsc.ufcg.edu.br/~peter/cursos/cc/material/p4-semantico-2p.pdf, acesso em nov/2007.

Kohl, N. (2007) “C++ Multimaps”, C/C++ Reference, http://www.cppreference.com/cppmultimap/index.html, acesso em nov/2007.

Vieira, M. (2007) “Projeto e implementação de um compilador – 3ª. parte”, Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Vieira, M. (2007) “Teoria dos Compiladores – Notas de Aula”, Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Lopes, C., Leonel, G. e Rossini, S. (2007), Implementação da análise sintática: a segunda etapa na construção do compilador Marvel, Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas, Departamento de Ciência da Computação