**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 e VerificadorTipos, seguindo o padrão já estabelecido na etapa anterior como pode ser verificado no arquivo *ImpressaoArvoreAbstrata.h*.

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].

*~~A geração de código foi iniciada a partir do momento em que a foram finalizadas as implementações das classes visitantes.~~*

*~~Especificar a implementação do módulo GeradorCodigo~~*

**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 GeradorCodigo 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 GeradorCodigo 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.

Os módulos referentes à análise semântica e geração de código intermediário estão representados pela figura 1, onde também são mostradas as suas interdependências.

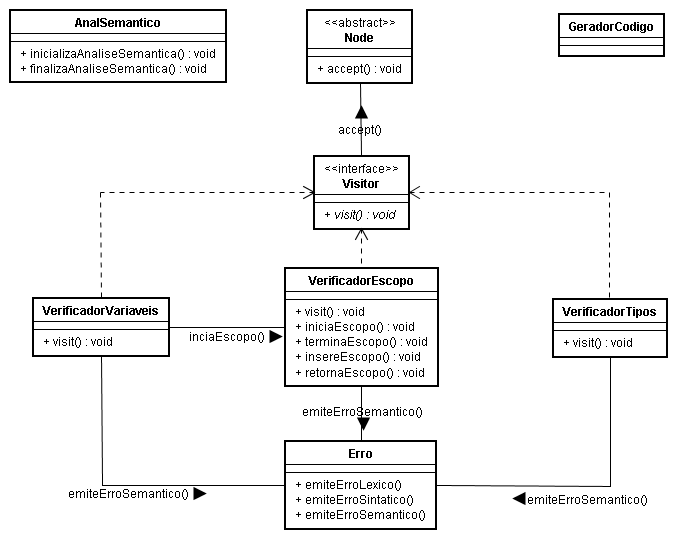


Figura 1. Interdependência entre os módulos da análise semântica.

**2. Analisador semântico**

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

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. *~~No momento em que a declaração da variável é aceita a estrutura Registro, definida na tabela de símbolos, é atualizada com o tipo da variável, o offset e o tamanho.~~*

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.

*~~Listar as regras semânticas do compilador, baseadas nas produções da gramática~~*

**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].

Sendo assim, a estrutura Registro pertencente à tabela de símbolos do compilador Marvel foi atualizada para fornecer mais informações referentes a identificadores, chamadas de função, números e vetores. A nova estrutura de Registro é listada abaixo.

typedef struct Registro {

int indiceLexema; *// Índice do lexema no array de lexemas*

int token; *// Token do registro*

int tipo; *// Tipo do Registro*

int escopo; *// Escopo do Registro*

struct Registro \*param; *// Define a qual registro este está ligado*

int offset; *// Offset do Registro*

int ativo; *// Verifica se a variável está ativa ou não*

struct Registro \*prox; *// Próximo registro*

} REGISTRO;

**4. Gerador de representação intermediária**

**5. Gerenciador de erros semântico**

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**

**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.

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.