UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Análise Semântica e Geração de Código

Relatório 3: Análise Semântica e Geração de Código
Construção de Compiladores 2
Profª. Drª. Helena de Medeiros Caseli

César Roberto de Souza 298603
Leonardo Sameshima Taba 298816

São Carlos, 1º de dezembro de 2009

1. Introdução

A análise semântica é o terceiro passo na construção de um compilador, e é a última etapa de análise. Subseqüente a essa fase vêm as etapas de geração e otimização de código intermediário e alvo, que no presente trabalho foram condensadas em uma única etapa de geração de código. Portanto, aqui se conclui a elaboração do compilador **getToken();** para a linguagem hipotética *ALG*, sendo desenvolvidos o analisador semântico e o gerador de código.

2. Desenvolvimento

2.1. Visão Geral

O analisador semântico, assim como o analisador sintático e léxico, também foi desenvolvido utilizando-se o paradigma Orientado a Objeto. Assim, foi desenvolvida a classe **AnalisadorSemantico** que realiza as operações de análise semântica sobre o código de entrada fornecido. Entretanto, a classe **AnalisadorSintatico** é a principal classe responsável pela condução da análise semântico; Isso ocorre pois a semântica tem um forte acoplamento com a sintaxe – de fato, a semântica é dirigida pela sintaxe. Portanto, a classe AnalisadorSemantico é implementada como uma propriedade da classe AnalisadorSintatico.

O analisador semântico foi implementado como uma máquina de estados, cuja representação pode ser vista a seguir:

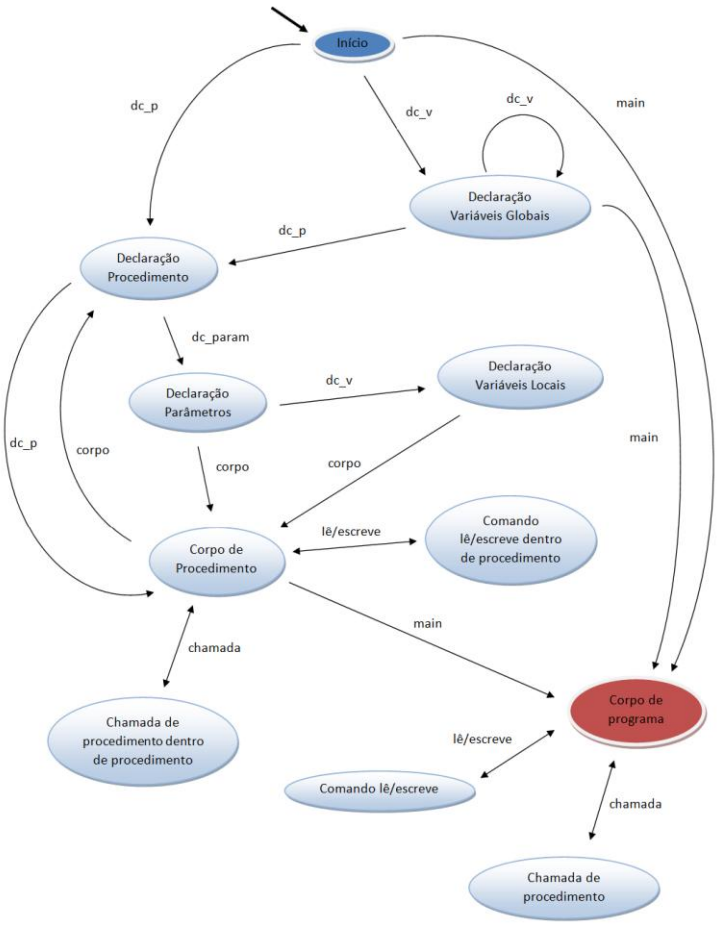

Figura 1. Máquina de estados que representa o funcionamento do analisador semântico

Essa abordagem foi escolhida por facilitar o processamento semântico, que é sensível ao contexto. Deste modo, não temos diversas variáveis booleanas indicando qual é o estado atual do analisador, mas sim apenas uma única variável "**estado"** que coordena o funcionamento do analisador.

Internamente, a classe AnalisadorSintatico utiliza a classe **TabelaSimbolos**, que implementa uma tabela de símbolos onde são guardadas as informações sobre as variáveis declaradas no programa de entrada. Mais detalhes sobre a tabela são fornecidos na seção Decisões de projeto.

As operações implementadas na tabela de símbolos são inserção, verificação e remoção de variáveis, mapeadas respectivamente nos métodos **inserir**, **verificar** e **remover**.

Todos os erros semânticos descritos na especificação são relatados por nosso compilador.

A outra etapa desenvolvida foi o gerador de código alvo, implementado na classe **Gerador**. Assim como a análise semântica, a geração de código também é dirigida pela sintaxe; Logo, a classe Gerador é uma propriedade da classe AnalisadorSintatico.

Todavia, o gerador de código é mais simples que o analisador semântico e, por isso, não foi implantado como uma máquina de estados. Sua estrutura é composta por diversos métodos sucintos que, conforme são chamados, constroem instrução por instrução o programa no código alvo, que em nosso caso é ANSI C. Os únicos métodos que têm funcionamento um pouco complexo são os relativos a listas de variáveis, que precisam guardar nomes e tipos, já que as declarações de variáveis na linguagem *ALG* têm formato invertido das declarações em linguagem C ("variáveis : tipo" na primeira e "tipo variáveis" na segunda).

2.2 Decisões de Projeto

A tabela de símbolos foi implementada como uma tabela hash, que é a estrutura intrínseca de objetos em JavaScript. A chave dos registros da tabela é o identificador da variável, ou seja, variáveis com nomes diferentes são guardadas em locais diferentes.

Os casos de colisão de nomes, como por exemplo, quando há variáveis locais com mesmo identificador em procedimentos diferentes, são resolvidos através do uso de um vetor dinâmico, que funciona como uma lista ligada: quando uma variável cujo nome já havia sido usado por outra variável é inserida, o vetor aumenta uma posição e a nova variável é guardada nesse novo espaço.

Essas estruturas (tabela hash e vetor dinâmico) foram escolhidas pois utilizam características intrínsecas da linguagem JavaScript, em especial a sua dinamicidade, facilitando a construção e utilização da tabela de símbolos, sem a necessidade de utilizar estruturas de dados mais complexas como listas ligadas.

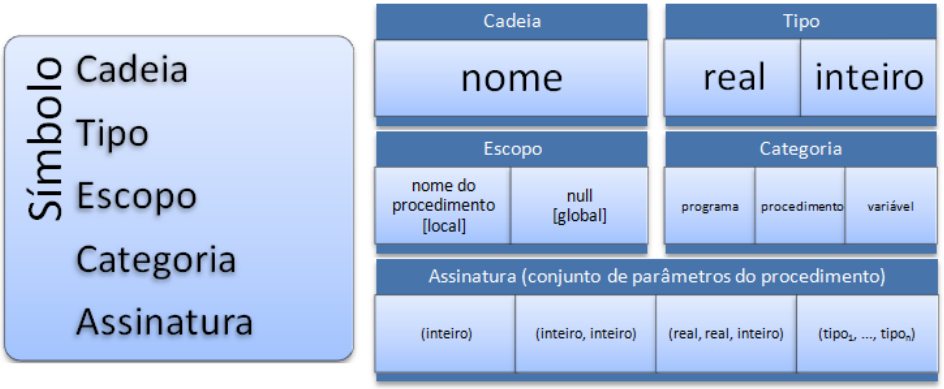

Figura 2. Representação gráfica de um símbolo e seus atributos

Em nossa implementação utilizamos uma única tabela de símbolos que guarda as seguintes informações sobre cada variável: **cadeia** (identificador), **tipo** (inteiro ou real), **escopo** (local ou global), **categoria** (variável, programa ou procedimento) e **assinatura** (apenas para procedimentos – indica quais os tipos e a ordem dos parâmetros). Esses cinco atributos foram escolhidos pois representam eficazmente os símbolos que podem ser encontrados na linguagem ALG.

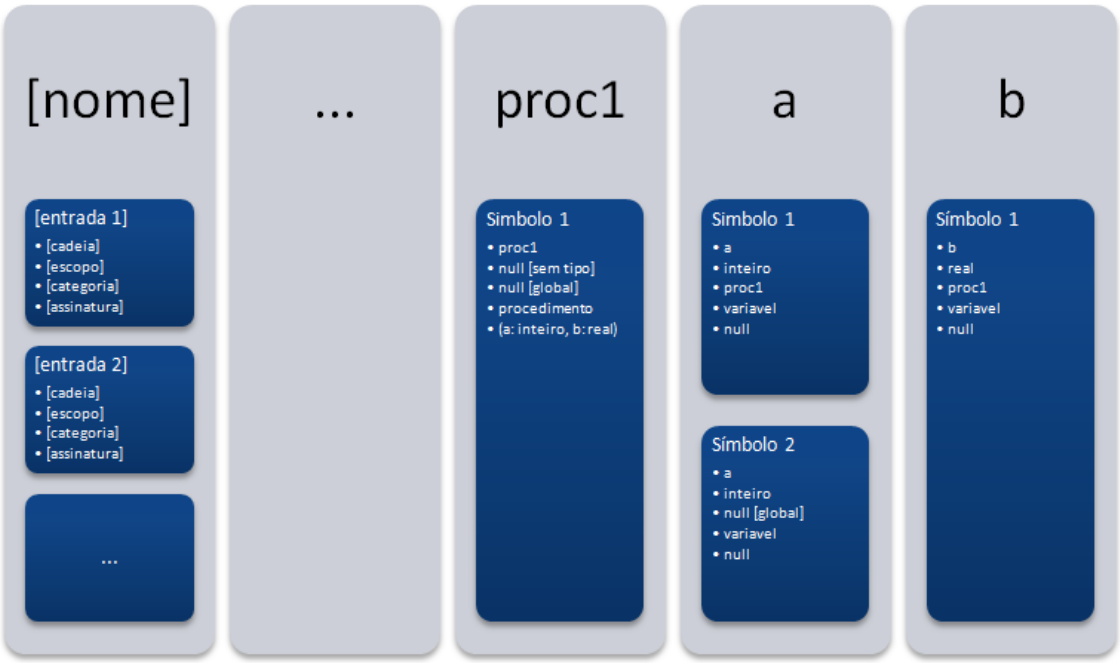

Figura 3. Representação gráfica da tabela de símbolos

Adicionalmente, quando ocorre um erro sintático na hora em que haveria uma inserção de símbolo na tabela de símbolos, o símbolo não é incluído. Assim, caso seja declarada uma variável sem que seja especificado seu tipo, o que consiste num erro sintático, esta não será incluída na tabela.

Não julgamos necessário haver uma tabela para cada tipo de símbolo ou para cada nível de escopo, já que cada variável pode ser apenas global ou local a procedimentos, e existem apenas dois tipos de dados. Assim sendo, decidimos manter o uso de uma única tabela para simplificar a implementação.

3. Compilação e execução

3.1. Compilando

Por JavaScript ser uma linguagem interpretada, não há necessidade de compilação. No entanto, alguns browsers modernos, como por exemplo o Google Chrome, realizam uma compilação Just-in-Time (JIT) do código para tornar sua execução mais ágil. Isso ocorre de maneira transparente para o usuário.

3.2. Executando

A execução do compilador é feita através de um navegador. Portanto, basta clicar duas vezes no arquivo *index.html* contido na pasta raiz do compilador para iniciá-lo.

Sua interface consiste em apenas duas caixas de texto, uma delas destinada ao código fonte de entrada e a outra dedicada a mostrar a saída desejada. Através de uma caixa

drop-down localizada no canto superior esquerdo, é possível escolher o tipo da análise desejada, seja léxica, sintática, semântica ou geração de código. Neste caso, deve ser escolhida a opção **análise semântica** ou **geração de código** (opção já selecionada por padrão).

O código fonte a ser processado deve ser colocado na caixa de texto superior. Em seguida, deve-se clicar no botão "Processar" para iniciar a análise ou a geração de código. Seus resultados serão então escritos na caixa inferior.

Note que também é possível optar pela execução de alguns exempos pré-determinados, alguns bem formados, segundo a linguagem ALG, e outros contendo erros sintáticos, léxicos e/ou semânticos.

3.2.1. Exemplos de execução

Entrada	Saída

programa fibonacci;
{ imprime os n primeiros numeros de fibonacci }
 	
var n1, n2, n3, qtde, i : inteiro;
inicio
  n1 := 0;
  n2 := 1;
  i := 0;
  le(qtde);
  enquanto i < qtde faca
  inicio
  escreve(n2);
  n3 := n1 + n2;
  n1 := n2;
  n2 := n3;
  i := i + 1;
  fim;
fim.
	/* programa fibonacci */
	
	#include <stdio.h>
	
	int n1, n2, n3, qtde, i;
	
	int main (int argc, char **argv) {
	
	  n1 = 0;
	  n2 = 1;
	  i = 0;
	  scanf("%d", &qtde);
	  while (i < qtde)
	  {
	  printf("%d\n", n2);
	  n3 = n1 + n2;
	  n1 = n2;
	  n2 = n3;
	  i = i + 1;
	  }
	
	  return 0;
	
	}

Entrada	Saída

programa exemplo2;
{ testa erros no uso de condicionais }
 	
var a: real;
var b: inteiro;
procedimento nomep(x: real);
  var a, c: inteiro;
  inicio
  le(c, a);
  se a<x+c entao
  inicio
  a:= c+x;
  escreve(a);
  fim
  senao c:= a+x;
  fim;
 	
inicio {programa principal}
  le(b);
  nomep(b);
fim.
	Analise terminada com erros
	Erro na linha 12: Atribuicao de valor real a variavel inteira 'a'.
	Erro na linha 15: Atribuicao de valor real a variavel inteira 'c'.
	Erro na linha 20: Parametro 'b' incorreto. Esperado valor real mas encontrado valor inteiro.