Análise Semântica

Mateus Tomoo Yonemoto Peixoto

DACOM – Universidade Tecnológica Federal do Paraná (UTFPR)

Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

{mateustomoo}@gmail.com

Abstract. This paper describes the development of the semantic analysis for a compiler being designed for the T++ programming language. In the future, it will so also approach the intermediate code generation.

Resumo. Este artigo descreve o desenvolvimento da análise semântica para um compilador projetado para a linguagem de programação T++. No futuro, também abordará a geração de código intermediário.

1. Introdução

Um compilador exige alguns passos até que um código intermediário seja obtido. Esses passos são divididos em análise léxica, análise sintática, análise semântica e geração de código.

A Análise Semântica é o terceiro passo do compilador. Nessa fase é onde são verificados os erros semânticos. Com a análise léxica e sintática já concluída, temos como saída uma árvore sintática abstrata (AST). Para a realização da análise semântica, deve-se usar como entrada a AST, que será percorrida para a realização da análise sensível ao contexto e a geração da tabela de símbolos.

Para o desenvolvimento deste trabalho, foi utilizado a linguagem de programação Python (versão 3.6.2).

2. Análise Semântica

Entendido o que é a análise semântica, deve-se compreender melhor a tabela de símbolos.

A tabela de símbolos é gerada para que o analisador consiga verificar várias coisas, como por exemplo se uma variável teve atribuição ou se foi declarada. Para isso ser possível, a tabela de símbolos armazena os seguintes campos: classe (se é variável ou função), nome, utilização (se está sendo utilizado), atribuição (se tal variável possui atribuição), tipo e o escopo (a qual pertence tal variável). Quando é uma classe do tipo função, algumas informações são alteradas no armazenamento na tabela de símbolos, ficando da seguinte forma: classe, nome, lista de parâmetros, utilização e o tipo. As imagens a seguir, são exemplos de como fica a tabela de símbolos para uma classe variável e uma classe função, respectivamente.

```
['variavel', 'a', True, True, 'inteiro', 'global']
['variavel', 'b', True, True, 'flutuante', 'global']
```

"Imagem01: Tabela de símbolos de variáveis."

```
['funcao', 'fatorial', ['inteiro', 'flutuante'], True, 'inteiro']
```

"Imagem02: Tabela de símbolos de função."

O principal objetivo da análise semântica é verificar se existem erros de contexto e também gerar avisos (warnings) quando necessário. Os erros e warnings gerados nessa etapa estão descritos na imagem a seguir:

1. A quantidade de parâmetros reais de uma chamada de procedimento deve ser igual a quantidade de parâmetros formais da sua definição;
2. *Warnings* deverão ser mostrados quando uma variável for declarada mais de uma vez;
3. *Warnings* deverão ser mostrados quando uma variável for declarada mas nunca utilizada;
4. *Warnings* deverão ser mostrados quando ocorrer uma coerção implícita de tipos (inteiro<->flutuante);
5. Uma variável não declarada. Lembrando que uma variável pode ser declarada:
   - no escopo do procedimento (como expressão ou como parâmetro formal);
   - no escopo global
6. Uma variável não inicializada;
7. Verificar a existência de uma função principal que inicializa a execução do código.
8. E outros situações descritas nos arquivos de testes.

"Imagem03: Erros e warnings gerados."

2.1. Exemplo de entrada e saída

A seguir é mostrado um exemplo de código de entrada e sua respectiva saída da análise semântica.

```
inteiro func(inteiro: x, inteiro: y)
    retorna(x + y)
fim

inteiro principal()
    inteiro: a
    a := func(10)
fim
```

"Imagem04: Código em T++."

Para o código acima, deve-se gerar um warning devido ao fato de que a função principal é do tipo inteiro e não retorna nada e também um erro, pois a função func possui dois parâmetros, e em sua chamada é passado apenas um.

```
ERRO: argumentos invalidos. Espera-se 2 e foi passado 1
WARNING: a funcao principal retorna void e deveria retornar inteiro
```

“Imagem05: Erro e warning gerado do código da Imagem04.”

A tabela de símbolos gerada pelo exemplo acima é mostrada na imagem a seguir.

```
['funcao', 'func', ['inteiro', 'inteiro'], True, 'inteiro']
['variavel', 'x', True, True, 'inteiro', 'func']
['variavel', 'y', True, True, 'inteiro', 'func']
['funcao', 'principal', 'void', False, 'inteiro']
['variavel', 'a', True, True, 'inteiro', 'principal']
```

“Imagem06: Tabela de símbolos gerado do código da Imagem04.”

3. Referências

LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

http://hackingoff.com/compilers/regular-expression-to-nfa-dfa

PLY (Python Lex-Yacc). Disponível em: <http://www.dabeaz.com/ply/>.