Trabalho Prático 1 - Parte 3: Análise Semântica e Geração de Código Intermediário

Geovanna Menegasse Silva
Vitor Lucio dos Santos Ferreira

1. Introdução

Este trabalho consiste no desenvolvimento de um analisador semântico, seguido da etapa de geração de código intermediário. Com a integração de todos os componentes implementados até aqui, concluímos o desenvolvimento de um compilador completo para a linguagem TIGER. A etapa abordada neste documento é responsável pela verificação de tipos, atribuição de valores semânticos para trechos de programa e geração do código intermediário.

Nas próximas seções apresentaremos a descrição da solução encontrada e as instruções de utilização do compilador implementado. Além disso, apresentaremos alguns dos resultados obtidos pela submissão dos códigos fonte feitos na linguagem TIGER, disponibilizados para teste, ao nosso compilador.

2. Descrição da solução

Para este trabalho prático, customizamos as estruturas dos arquivos do lex e do yacc, usadas nos trabalhos anteriores, bem como criamos nossa própria abstração, em C, para a AST e a tabela de símbolos. Mais detalhes dessas estruturas estão no item 2.1.

Para a criação de um executável para nosso compilador, foi feita a transformação do *"scanner.l"* no arquivo *"lex.yy.c"*, para a parte do analisador léxico e a transformação dos arquivos *"sintaxe.y"* nos arquivos *"y.tab.h"*, que contém a declaração dos tokens para serem usados no scanner, e o *"y.tab.c"*, que contém a implementação da análise sintática, contendo os estados e transições LALR(1) correspondentes a GLC usada. Ao final, criou-se o executável *"a.out"* que representa o compilador, o qual recebe um código fonte TIGER como entrada e responde com o código intermediário correspondente ou o erro associado ao código fonte submetido.

Por fim, para facilitar a compilação e execução do programa, criamos um arquivo makefile que executa os comandos correspondentes.

2.1. Descrição da saída

A saída esperada ao submeter um programa ao nosso compilador possui blocos separados para cada parte específica. O primeiro bloco consiste na impressão do código fonte que está sendo analisado. O próximo bloco consiste da listagem da sintaxe abstrata, contendo o resultado da análise sintática. O último bloco é reservado para listagem de erros semânticos, tabela de símbolos e código intermediário.

A tabela de símbolos contém as informações sobre os identificadores detectados no programa fonte. A coluna de tipo nos informa qual é o tipo atribuído àquele símbolo. Os tipos podem ser: “int”, “string”, “array”, “record”, ou algum tipo declarado anteriormente. Em caso de arrays, o tipo do array informado é armazenado na coluna de valor, por exemplo: “array of int” possui tipo “array” e valor “int”. Na coluna de classe, colocamos uma string representando a qual classe o símbolo pertence. Essas classes podem ser: “variable” para variáveis, “type” para tipos declarados, “function” para funções, “parameter” para parâmetros de funções ou “var rec” para variáveis de registro. A coluna de bloco informa a qual escopo o símbolo lido pertence, sendo “?” para símbolos sem escopo (escopo global). No exemplo abaixo, os parâmetros “p1” e “p2” pertencem ao escopo da função “f”. Na coluna de número de parâmetros, armazenamos na linha correspondente, a quantidade de parâmetros de uma função ou a quantidade de variáveis de um registro. Em caso de funções, a linha correspondente ao símbolo da função recebe a quantidade total de parâmetros da mesma e seus parâmetros recebem, na mesma coluna, o número do parâmetro que representam. No exemplo abaixo, a função “f” possui dois parâmetros (“p1” e “p2”). O parâmetro “p2” é detectado primeiro, por isso recebeu valor “1” nesta coluna. Já o parâmetro “p1” recebeu o valor “2” por representar o segundo parâmetro da função. A última coluna da tabela representa o rótulo daquele símbolo no código intermediário. Variáveis e parâmetros recebem um rótulo de um temporário e funções recebem rótulos de suas “labels”.

```
================================================================================================

TABELA DE SIMBOLOS:

|    NOME   |    TIPO   |   VALOR   |   CLASSE  |   BLOCO   | NUM DE PARAM  |  ROTULO  |
x-----------x-----------x-----------x-----------x-----------x---------------x----------x
|         v |       int |         ? |  variable |         ? |             0 |      t0  |
|         t |     array |       int |      type |         ? |             0 |          |
|         f |       int |         ? |  function |         ? |             2 |      L0  |
|        p1 |       int |         ? | parameter |         f |             2 |      t3  |
|        p2 |    string |         ? | parameter |         f |             1 |      t4  |

ANALISE SEMANTICA: OK!

================================================================================================
```

2.2. Descrição das estruturas

- Customização do arquivo Lex: Neste arquivo, nós aproveitamos as regras para geração dos tokens para também gerar as estradas iniciais na tabela de símbolos.
- Customização do arquivo yacc: Neste arquivo, usamos as produções da gramática para montar nossa AST, considerando cada produção como um nó da árvore, contendo seu próprio código intermediário e suas próprias validações de erro. Em alguns destes nós também criamos novas entradas na tabela de símbolos, quando necessário.
- Por questões de modularização separamos as estruturas da AST, tabela de símbolos e validações nestes arquivos: "*arvore.c*", "*erros.c*", "*gerais.c*" e "*tabela.c*".

2.3. Declarações

O compilador está tratando os casos de declarações de tipo, variável, função e registro, adicionando-os na tabela de símbolos.

2.4. Tratamento para tipos recursivos

Para tratar tipos recursivos, usamos a tabela de símbolos para descobrir o tipo primitivo por detrás dos tipos recursivos e, com isto, usá-los como se fossem tipos primitivos.

2.5. Tratamento para declarações de funções

Para declaração de funções, usamos as orientações, referentes à tabela de símbolo dadas em aula, onde a função contém o número de parâmetros e cada parâmetro tem seu tipo com o valor "parametro", distinguindo eles das demais declarações de variáveis. Além disto, para distinguir os parâmetros de diferentes funções, criamos um atributo na tabela de símbolo sinalizando qual função continha determinado parâmetro.

2.6. Questões não solucionadas

- Tratamento de declaração de variáveis locais dentro da declaração de uma ou mais funções.
- Demais casos relacionados a expressões e comandos.

3. Exemplos de teste

Para testarmos nosso compilador, submetemos ao programa alguns códigos para teste que foram desenvolvidos de forma a capturar erros e acertos semânticos e particularidades na geração do intermediário. Apresentamos aqui alguns desses testes dentre os utilizados, contendo erros e acertos para objetos distintos da linguagem TIGER.

3.1. Exemplos corretos

O programa abaixo aborda várias estruturas e casos semânticos que implementamos, passando tanto na análise sintática quanto na análise semântica. A declaração de tipos recursivos inicia o programa, permitindo casos de tipos e variáveis com nomes iguais. Para declarações de variáveis, vemos alguns casos diferentes como: atribuição com verificação de tipos simples e recursivos; atribuição com expressões simples que também verificam os tipos; atribuição sem comparação de tipos e com chamada de variáveis em expressões simples. Nas declarações de funções, vemos os casos de função com retorno e sem retorno. Em ambos os casos, os tipos das variáveis chamadas nas expressões do corpo da função são comparados com os tipos que elas possuem na parte de parâmetros e não com os tipos que elas possuem no escopo global, a menos que a variável não seja encontrada na lista de parâmetros daquela função, como na linha 11. Em declarações de arrays com tipos recursivos, a comparação de tipos é feita recursivamente. A expressão dentro do colchete e após o “of” podem ser variáveis. Para declarações de registro, permitimos que suas variáveis sejam recursivas.

```
============================================================
 1: let
 2:
 3:     type s = string
 4:     type c = s
 5:
 6:     var c : s := "aaa"
 7:
 8:     var a : int := 2 + 1 * 5
 9:     var b := 2 + a
10:
11:     function sum (x1: int, x2: int) : int = x1 + x2 + b
12:     function str (x1: string) = x1
13:
14:     type rec0 = array of int
15:     type rec1 = array of rec0
16:     type rec2 = rec1
17:
18:     var x : rec2 := rec1 [b] of rec0 [10] of a
19:
20:     type list = { first: int, rest: list }
21:
22: in
23:
24: end
============================================================
```

Saída: listagem do programa fonte submetido ao compilador

Para melhor analisarmos a tabela de símbolos, retiramos a declaração da linha 20 do código. A tabela abaixo mostra a atribuição de valores para cada atributo de um símbolo no nosso código fonte. Vemos as atribuições de tipos que foram feitas corretamente e a recursividade presente na verificação dos tipos recursivos e arrays. Os rótulos serão utilizados na geração de código intermediário que será mostrado a seguir.

```
=============================================================================================

TABELA DE SIMBOLOS:

|    NOME   |    TIPO   |   VALOR   |   CLASSE   |   BLOCO   | NUM DE PARAM |  ROTULO  |
x-----------x-----------x-----------x------------x-----------x--------------x----------x
|         s |    string |         ? |       type |         ? |            0 |          |
|         c |         s |         ? |       type |         ? |            0 |          |
|         c |         s |         ? |   variable |         ? |            0 |       t2 |
|         a |       int |         ? |   variable |         ? |            0 |       t3 |
|         b |       int |         ? |   variable |         ? |            0 |       t4 |
|       sum |       int |         ? |   function |         ? |            2 |       L0 |
|        x1 |       int |         ? |  parameter |       sum |            2 |       t6 |
|        x2 |       int |         ? |  parameter |       sum |            1 |       t7 |
|       str |      void |         ? |   function |         ? |            1 |       L1 |
|        x1 |    string |         ? |  parameter |       str |            1 |       t9 |
|      rec0 |     array |       int |       type |         ? |            0 |          |
|      rec1 |     array |      rec0 |       type |         ? |            0 |          |
|      rec2 |      rec1 |         ? |       type |         ? |            0 |          |
|         x |      rec2 |         ? |   variable |         ? |            0 |      t13 |

ANALISE SEMANTICA: OK!

=============================================================================================
```

Saída: listagem da tabela de símbolos

Abaixo temos o código intermediário gerado. Podemos ver que declarações de tipos não são consideradas no intermediário. Declarações de funções são

escritas ao final de todo o código gerado para o programa, sendo nomeadas por suas "labels". Na figura podemos ver que as funções e expressões utilizaram corretamente o rótulo correspondente às variáveis e parâmetros que invocaram.

```
================================================================================

Listagem do Codigo Intermediario:

null:
ESEQ(
 SEQ(
  MOVE(TEMP t2,CONST "aaa"),
  SEQ(
  MOVE(TEMP t3,BINOP(MAIS,CONST 2,BINOP(MULTIPLICACAO,CONST 1,CONST 5))),
  SEQ(
  MOVE(TEMP t4,BINOP(MAIS,CONST 2,TEMP t3)),
  SEQ(
  MOVE(TEMP t13,CALL(NAME initarray,CONST 0,TEMP t4,
  CALL(NAME initarray,CONST 0,CONST 10,TEMP t3))),
  EXP(CONST 0))))),
 EXP(CONST 0)
)

L0:
MOVE(TEMP rv,BINOP(MAIS,BINOP(MAIS,TEMP t6,TEMP t7),TEMP t4))

L1:
TEMP t9

================================================================================
```

Saída: listagem do código intermediário

Para a listagem da sintaxe abstrata, retiramos algumas partes do código para melhor visualização e análise. Na sintaxe abstrata, declarações de funções e de tipos são consideradas.

```
============================================================

Listagem da Sintaxe Abstrata:

LetExp(
 DecList(
  VarDec(c,
   StringExp("aaa")),
  DecList(
  VarDec(a,
   int,
   OpExp(MAIS,IntExp(2),OpExp(MULT,IntExp(1),IntExp(5)))),
  DecList(
  FunctionDec(sum,
   Fieldlist(x1,
    int,
   FieldList()),
    int,
    OpExp(MAIS,x1,a)),
  DecList(
  TypeDec(rec0,
   ArrayTy(int)),
  DecList(
  VarDec(x,
   rec0,
   ArrayExp(rec0,a,a)),
  DecList()))))),
 SeqExp()
)

ANALISE SINTATICA: OK!

============================================================
```

Saída: listagem da sintaxe abstrata

3.2. Exemplos com erros

1. Símbolo não declarado

```
============================================================
 1: let
 2:
 3:     function sum (x1: int, x2: int) : int = x1 + x2 + b
 4:
 5: in
 6:
 7: end
============================================================

============================================

**** Erro: Símbolo não declarado! ****

Nome do arquivo: tests/decs/00.txt
Linha: 3

ANALISE SEMANTICA: ERRO!

============================================
```

2. Expressões com tipos distintos

```
============================================================
 1: let
 2:
 3:     var c := "aaa"
 4:     function sum (x1: int, x2: int) : int = x1 + x2 + c
 5:
 6: in
 7:
 8: end
============================================================

================================================================================

**** Erro: Não é possível realizar essa operação para tipos distintos! ****

Nome do arquivo: tests/decs/00.txt
Linha: 4

ANALISE SEMANTICA: ERRO!

================================================================================
```

3. Atribuição de tipo errado

```
====================================
 1: let
 2:
 3:     type s = string
 4:     type c = s
 5:     var a : c := 2 + 1 * 5
 6:
 7: in
 8:
 9: end
====================================

============================================================

**** Erro: Tipo declarado difere do tipo esperado! ****

Nome do arquivo: tests/decs/00.txt
Linha: 5

ANALISE SEMANTICA: ERRO!

============================================================
```

4. Tipo recursivo errado

```
============================================
 1: let
 2:
 3:     type rec0 = array of int
 4:     type rec1 = array of rec0
 5:     type rec2 = rec1
 6:
 7:     var x : rec2 := rec1 [10] of 3
 8:
 9: in
10:
11: end

============================================

================================================

**** Erro: quantidade de arrays diferentes! ****

Nome do arquivo: tests/decs/00.txt
Linha: 7

ANALISE SEMANTICA: ERRO!

================================================
```

4. Instruções para utilização do analisador

1. Utilizar um sistema operacional Linux.
2. Instalar as ferramentas Flex, Bison e C:

```
sudo apt install flex
sudo apt install bison
```

3. Ir para a pasta root do projeto.
4. Executar o seguinte comando para gerar o scanner:

```
make compile
```

5. ou executar os comandos (contidos dentro do Makefile):

```
flex scanner.l
yacc -v -d sintaxe.y
cc y.tab.c
```

6. Colocar o arquivo com o código fonte no diretório do analisador.
7. Executar o comando:

```
./a.out <diretorio_do_arquivo>/<nome_do_arquivo>
```

8. Caso o analisador não acuse erro de semântica, a saída terá a mensagem:

```
“ANALISE SEMANTICA: OK!”
```

   caso contrário, a saída mostrará o erro encontrado seguido da frase:

```
"ANALISE SEMANTICA: ERRO!”
```

9. Nos casos sem erros semânticos, o código fonte, a sintaxe abstrata, a tabela de símbolos e o código intermediário serão impressos.
10. Em casos com erros semânticos, apenas o código fonte e o erro encontrado serão impressos.

5. Ferramentas de apoio

Como ferramentas de apoio para o desenvolvimento do analisador sintático, utilizamos as tecnologias Flex e Bison para gerar o código de análise léxica e dos arquivos resultantes da análise sintática e semântica.

6. Conclusão

Com este trabalho, concluímos que o desenvolvimento de um compilador não é uma tarefa trivial, sendo necessário ter um conhecimento consolidado sobre a linguagem, sua gramática e as implicações desses 2 para a análise semântica. Contudo foi uma experiência muito proveitosa.

7. Referências

1. Aho, Alfred V., Lam Monic S., Sethi, R.; Ullman and Jeffrey D., Compilers Principles, Techniques, & Tools , 2nd Edition, Pearson Addison Wesley, New York, 2007
2. https://medium.com/codex/building-a-c-compiler-using-lex-and-yacc-446262056aaa