Compiladores I - Trabalho Prático 1

Augusto Maillo Queiroga de Figueiredo, João Antonio Oliveira Pedrosa

[1]Universidade Federal de Minas Gerais

[2]Belo Horizonte - MG

augusto.maillo@gmail.com, joao.pedrosa@dcc.ufmg.br

1. Introdução

Um compilador pode ser divido em duas partes: Análise e Síntese. O objetivo deste trabalho prático é desenvolver toda a parte de análise de um compilador para a linguagem Tiger. A análise de um compilador pode ser dividida em três partes principais:

- Analise Léxica
- Analise Sintática
- Analise Semântica

A primeira consistiu em desenvolver o analisador léxico para a linguagem TIGER, o qual tinha como objetivo agrupar os caracteres de entrada dos programas em *tokens*.

Nesta segunda parte do trabalho, foi desenvolvida a **análise sintática**. A função da análise sintática é verificar se o código recebido, agora já *tokenizado* pela análise léxica, está de acordo com a gramática de determinada linguagem. Isto é, a sequência e organização dos comandos seguem as regras definidas para a linguagem alvo.

O desenvolvimento do trabalho utiliza como apoio um gerador de análise sintática, já implementado. A parte principal do trabalho consiste no desenvolvimento do arquivo de configuração desse gerador, o qual especifica as regras da gramática assim como a resoluções de eventuais conflitos.

2. Implementação

O trabalho foi desenvolvido na linguagem **C** e compilado com o **GCC 9.3.0**. Baseado na documentação da linguagem TIGER, analisamos as sequências dos *tokens* gerados pela a análise léxica para especificar as regras da gramática, nomeando os possíveis termos e explicitando sua geração.

2.1. Ferramentas Utilizadas

Para a análise léxica, a principal ferramenta de apoio utilizada foi o FLEX, um gerador de análise léxica escrito em C, em sua versão **2.4.6**.

Já para a parte de análise sintática, a principal ferramenta de apoio utilizada foi o YACC, um gerador de analisador sintático. Esta ferramenta é capaz de gerar códigos fontes para a compilação de um analisador sintático com base em um arquivo de configuração, onde as regras da gramática da linguagem são especificadas.

2.2. Alterações na Gramática

Devido a alguns conflitos gerados pela gramática, da maneira como foi descrita na especificação do trabalho, algumas alterações foram realizadas nas seguintes regras:

- *l-value*: Alteramos a regra *l-value ::= id* para *l-value ::= type-id* visando remover ambiguidade e o gerador ser capaz de identificar corretamente quando um ID se torna um l-value, visto que a precedência entre as regras garante a corretude desta forma.
  Adicionamos também a regra *l-value ::= type-id [exp]* para que o analisador seja capaz de reconhecer corretamente operações aninhadas como por exemplo *foo[0].bar[0]*.
- *type-id*: Adicionamos a regra *type-id ::= "int "string"* para conseguir englobar a *tokenização* dos termos *int* e *string* separados, como feito na etapa de análise léxica.

Além disso, incluímos regras de precedência à esquerda para os operadores +, -, *, /, & e |. Também explicitamos regras de não associatividade para os operadores relacionais.

2.3. Arquivos

As tarefas descritas tanto na primeira quanto na segunda parte do trabalho são realizadas em dois diferentes arquivos. Segue uma breve descrição do conteúdo de cada um deles:

2.3.1. yylex.lex

Especificação das expressões regulares e dos códigos em C associados à cada uma delas. No caso deste trabalho, o código C consiste em retornar o identificador do Token relacionado à expressão.

2.3.2. tiger.y

Especificação das regras de gramática da linguagem Tiger. O programa processa um código de entrada e imprime todas as regras de gramática identificadas, na ordem em que são processadas.

3. Instruções de Uso

O primeiro passo consiste em utilizar o **YACC** para obter o analisador sintático baseado nas regras descritas no arquivo **tiger.y**.

```
yacc -d tiger.y
```

Este comando gera os arquivos **y.tab.c** e **y.tab.h**. O arquivo **y.tab.h** será utilizado pela especificação do analisador léxico para importar os tokens que foram definidos (apenas os nomes, as regras de tokenização estão definidas em yylex.lex). Para obter o analisador léxico faremos

```
lex yylex.lex
```

Este comando gera o arquivo **lex.yy.c** que será utilizado junto com **y.tab.c** na compilação do executável **analyzer**, que efetivamente realiza a análise sintática.

```
gcc lex.yy.c y.tab.c -o analyzer
```

Finalmente obtemos o executável **analyzer**, o qual recebe o texto a ser analisado sintaticamente.

Todo este processo pode ser executado com o *bash script* **run.sh**

```
bash run.sh <arquivo_de_entrada>
```

Fornecemos um arquivo de entrada para teste, chamado **test.in**. O resultado do analisador léxico sobre este arquivo pode ser obtido executando

```
bash run.sh test.in
```

Fornecemos um arquivo main também, que executa dois códigos, um com sintaxe correta e um com sintaxe incorreta. O arquivo main também imprime os dois códigos fonte. Para executá-lo:

```
./main
```

A pasta **inputs** contém os arquivos utilizados para testar a implementação. Os resultados obtidos podem ser conferidos na pasta **outputs**.

4. Testes

Realizamos dois testes simples, apenas para demonstrar o comportamento do Analisador com uma sintaxe correta e uma incorreta.

4.1. Sintaxe Correta

O teste com sintaxe correta é o seguinte:

```
let
function soma(x1: int, x2: int) : int = x1 + x2
    var resultado : int := 0
    var result_string : string := ""
in
    result_string := soma(2, "ufmg", 5 )
end
```

Para o qual, a saída do analisador sintático é:

```
type_id ::= type_id
type_id ::= type_id
tyfields1 ::= , id : type_id tyfields1
```

```
tyfields ::= id : type_id tyfields1
type_id ::= type_id
type_id ::= id
l_value ::= type_id
exp ::= l_value
type_id ::= id
l_value ::= type_id
exp ::= l_value
exp ::= exp + exp
fundec ::= function id ( tyfields ) : type_id  = exp
dec ::= fundec
type_id ::= type_id
exp ::= num
vardec ::= var id : type_id := exp
dec ::= vardec
type_id ::= type_id
exp ::= string
vardec ::= var id : type_id := exp
dec ::= vardec
decs ::= dec decs
decs ::= dec decs
decs ::= dec decs
type_id ::= id
l_value ::= type_id
exp ::= num
exp ::= string
exp ::= num
args1 ::= , exp args1
args1 ::= , exp args1
args ::= exp args1
exp ::= id ( args )
exp ::= l_value := exp
expseq ::= exp expseq1
exp ::= let decs in expseq end

sintaxe correta
```

4.2. Sintaxe Incorreta

O teste com a sintaxe incorreta possui um erro na linha 3 (duas strings "of"consecutivas) o qual deveria ser identificado na análise sintática. Abaixo segue este teste:

```
let
    type tipoarranjo = array of int
    var Arranjo:tipoarranjo := tipoarranjo [10] of of 0
in
    Arranjo[2] = "nome"
```

```
end
```

Para este programa o output do analisador é o seguinte:

```
type_id ::= type_id
ty ::= array of id
tydec ::= type id = ty
dec ::= tydec
type_id ::= id
type_id ::= id
exp ::= num

erro de sintaxe
```

5. Conclusão

Durante o desenvolvimento deste trabalho foi possível compreender as dificuldades enfrentadas pelo projetista de um compilador visando evitar ambiguidade. Mais de uma vez obstáculos foram encontrados ao perceber que múltiplas regras acabam por ser conflitantes, necessitando de maior cuidado e de alterações na gramática para que os conflitos sejam resolvidos.

Com as partes de análise léxica e sintática concluídas, o próximo passo para o desenvolvimento do compilador é a análise semântica, que será realizada na próxima parte desse Trabalho Prático.

6. Referências

Todas as referências que foram consultadas pela internet possuem links clicáveis para a fonte.

- Tutorial on lex/yac - Jonathan Engelsma
- Tiger Compiler Reference Manual
- Tiger Language Manual - Prof. Stepehn A. Edwards
- Especificação das Expressões Regulares do LEX
- Slides da Disciplina