Trabalho Prático 1.2 - Analisador Sintático

André Luiz Moreira Dutra - 2019006345
Pedro Dias Pires - 2019007040

Compiladores I

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

cienciandre@ufmg.br
pdp2019@ufmg.br

1. Introdução

O trabalho prático 1 da disciplina Compiladores I consiste em implementar incrementalmente um compilador para a linguagem TIGER. A interpretação de uma linguagem passa por diversas etapas, cada uma correspondendo a uma parte do compilador: o analisador léxico, o analisador sintático, o analisador semântico, o gerador de código e um otimizador, se houver. A segunda parte do trabalho, à qual se refere esta documentação, consiste em implementar um analisador sintático para a linguagem usando a ferramenta Cup, de modo que, dados os tokens gerados pela análise léxica anteriormente, ele apresente as produções da gramática utilizadas e retorne se o programa fonte está ou não sintaticamente correto de acordo com a sintaxe da linguagem TIGER fornecida. As demais partes do compilador citadas não serão solucionadas nesse trabalho, e serão implementadas nas etapas seguintes.

2. Descrição do Trabalho

Esta etapa do trabalho corresponde à implementação das fases de análise léxica e sintática do compilador TIGER que está sendo desenvolvido de forma modular. A fase de análise léxica foi implementada na etapa anterior e, nesta etapa, o foco foi direcionado à implementação da análise sintática. As fases restantes do *front-end* do compilador, que correspondem à análise semântica e geração de código intermediário, serão desenvolvidas em etapas posteriores deste trabalho.

Para a execução dessa parte do trabalho, foi necessário especificar a definição livre do contexto da linguagem TIGER na sintaxe do Cup para gerar a tabela do analisador sintático LALR que será utilizado pelo compilador. Para tanto, foi necessário realizar pequenas mudanças na gramática para resolver conflitos na análise. Além disso, foi necessário integrar o analisador léxico, implementado na fase anterior deste trabalho, para o obter os *tokens* a

serem analisados a partir do código fonte. Por fim, foi necessário escrever um programa para processar arquivos fonte de teste na linguagem TIGER, a fim de verificar se programas sintaticamente corretos são aceitos e programas com erros de sintaxe são rejeitados e têm seus erros apontados pelo analisador.

A especificação da definição livre do contexto no formato utilizado pela ferramenta Cup (arquivo no formato ".cup") para gerar a tabela de análise sintática em Java, foi criada a partir da gramática livre do contexto disponível na especificação da linguagem TIGER. Além de descrever as regas da gramática no arquivo "Grm.cup", disponível no diretório "Grammar", também foi necessário especificar corretamente as relações de precedência e associatividade dos operadores, para evitar possíveis ambiguidades na gramática. Algumas modificações também foram realizadas nas regras da gramática para eliminar um conflito do tipo *reduce-reduce* na análise. Também foram encontrados dois conflitos do tipo *shift-reduce*, os quais foram resolvidos de forma automática pelo Cup em favor do *shift*.

Também foi necessário fazer algumas modificações na especificação dos *tokens*, utilizada pelo JLex para gerar o analisador léxico, para corrigir alguns erros cometidos na etapa anterior (a qual focou na implementação do analisador léxico) que identificavam alguns *tokens* de forma incorreta ou imprópria para a análise sintática. A nova especificação léxica, já integrada à representação gerada pelo Cup, pode ser encontrada no arquivo "Tiger.lex", no diretório "Lexer".

Além das classes geradas automaticamente pelas ferramentas JLex (gerador de analisador léxico) e Cup (gerador de analisador sintático), também foram implementadas duas classes: *Analisador* e *Main*. A classe *Analisador* implementa um método estático "*analisar",* para realizar a análise sintática de um arquivo fonte TIGER cujo caminho é dado como parâmetro e um método *main* que processa quaisquer arquivos cujos caminhos sejam dados como parâmetro pela linha de comando, utilizando o método *analisar*. A classe *Main*, por sua vez, executa o método *analisar,* da classe *Analisador*, para dois arquivos fonte de teste, um sintaticamente correto e outro com erro de sintaxe, que estão disponíveis junto ao código fonte desta etapa do trabalho.

Por fim, foram escolhidos dois códigos fonte de teste dentre os fontes TIGER disponibilizados junto à especificação deste trabalho prático. Como exemplo correto, foi utilizado o arquivo "FATORIAL.TIG", que implementa um fatorial na linguagem TIGER e, como exemplo de código sintaticamente incorreto, foi utilizada uma modificação do fonte "TESTE6.TIG". O programa resultante foi, então, executado com esses dois arquivos de entrada para verificar seu funcionamento.

3. Conflitos e Alterações na gramática

Como a gramática dada era ambígua em alguns pontos, naturalmente o compilador gerou conflitos entre algumas produções. Para resolvê-los, foi necessário definir relações de precedência entre operadores e fazer ligeiras alterações na gramática. Os conflitos gerados, bem como as soluções para cada conflito, serão apresentadas a seguir.

3.1. Reduce-reduce

Na gramática original, o terminal id poderia assumir duas formas, segundo duas produções:

(1) type-id ::= id

(2) l-value ::= id

Em quase todas as produções o analisador soube qual delas usar, mas, em duas em específico, houve um conflito reduce-reduce:

(3) exp ::= type-id [exp] of exp

(4) l-value ::= l-value [exp]

Perceba que em ambos os casos, após dar shift no id, ele teria como próximo símbolo um abre colchete seguido do que se tornaria uma expressão. Então o analisador não saberia decidir entre reduzir o id para type-id, considerando que seria usada a produção 3, ou reduzir o id para l-value, considerando que seria usada a produção 4, gerando o único conflito *reduce-reduce* da gramática.

O conflito foi resolvido transformando-o em um conflito *shift-reduce*, mais fácil de lidar, e depois resolvendo-o. É notável que, se as produções 3 e 4 se iniciassem diretamente com o terminal id, o problema seria resolvido. O analisador executaria um *shift* até o fecha colchete e, se o próximo símbolo fosse um *of,* faria um *shift* e seguiria pela produção 3 e, caso contrário, faria um *reduce* segundo a produção 4.

Observe ainda que o caso em que ambas se iniciam com somente um id é o único caso de conflito, pois o type-id só pode assumir a forma de um único id. Se o l-value assumisse uma outra forma, como id.id, ele seria reconhecido antes do abre colchete e o conflito não se iniciaria. Então para resolver o problema bastaria criar cópias das produções conflitantes levando em consideração o caso em que os símbolos iniciais são um único id, e adicioná-las junto às originais na gramática:

(3.1) exp ::= id [exp] of exp

(4.1) l-value ::= id [exp]

Como type-id só pode ter a forma de id, a produção 3 foi substituída na gramática pela 3.1, enquanto as produções 4 e 4.1 coexistem. Nesse caso, após dar um shift no id, o compilador teria que decidir entre fazer um *reduce* para l-value, para seguir com a produção 4, ou fazer um shift no abre colchete, para seguir com as produções 3.1 ou 4.1, transformando o problema em um shift-reduce. Como por padrão o compilador sempre escolhe o shift, ele seguiria com as produções 3.1 e 4.1 e distinguiria elas pela presença ou não do *of* mais à frente, levando a uma escolha correta de produções. Observe ainda que, no caso em que a produção não se inicia com um único id, por exemplo, id.id.id[2], o analisador primeiro reduziria id.id.id para l-value, e depois a única opção seria seguir com a produção 4, que é a correta, ou seja, a alteração não interfere com o resto da gramática.

Apesar de as produções 3.1 e 4.1 terem tirado a caracterização de type-id/l-value do id, é fácil descobrir a forma de id pela produção escolhida. Se for escolhida a produção 3.1, o id só pode ser um type-id, enquanto se for escolhida a produção 4, o id só pode ser um l-value, pois as produções 3.1 e 4.1 vieram das produções 3 e 4 com essas definições. Assim, o conflito foi resolvido sem perda de informação sobre o símbolo inicial.

3.2. Shift-reduce

Além do conflito shift-reduce indicado acima, a gramática tinha outros conflitos gerados pela ambiguidade em operadores entre expressões e no if then else. No caso dos operadores, todas as produções foram definidas da forma:

exp ::= exp + exp

exp ::= exp*exp

E assim por diante. O problema dessa definição é que, no caso de um exp+exp*exp, dando shift no exp+exp, o compilador não sabe se dá shift no * ou se reduz o exp+exp que já tem, gerando o shift-reduce de operadores. Esse conflito foi resolvido definindo, de forma separada, a precedência entre os operadores, uma funcionalidade que o cup oferece. O menos unário sendo o de maior precedência, seguido do * e /, seguidos do + e -, seguidos dos operadores relacionais (=, <>, …), seguidos dos operadores lógicos (& e |). A associatividade de cada operador também

foi definida (todos à esquerda, exceto o menos unário, que se associa à direita, e os relacionais, que não se associam), eliminando toda ambiguidade entre operadores, e seus respectivos conflitos.

Por fim, o conflito shift-reduce restante foi entre as produções relativas aos condicionais:

exp ::= if exp then exp

exp ::= if exp then exp else exp

O compilador, após dar *shift* no if exp then exp, não saberia decidir entre dar *shift* no else ou reduzir até o then, deixando o else para um if exterior. No entanto, como a definição da linguagem afirma que o else deve ser casado com o if mais interior ainda não casado, e esse if é aquele no topo da pilha de símbolos, a opção correta é sempre fazer o *shift* do else, que é a escolha feita por padrão pelo Cup para resolver o conflito.

4. Ferramentas de Apoio

Conforme descrito anteriormente, para a implementação do analisador sintático, foi utilizada a ferramenta Cup, que gera um analisador sintático LALR a partir da especificação de uma gramática livre do contexto e de regras de precedência e associatividade especificadas em um arquivo de texto em um formato específico, cuja extensão ".cup". A ferramenta Cup cria, de forma automática, classes Java que representam os *tokens* e implementam o analisador sintático.

O analisador léxico utilizado, que foi implementado na etapa anterior deste trabalho prático, utiliza a ferramenta JLex para gerar uma classe Java que implementa a interface do analisador léxico usado pelo analisador sintático gerado pelo Cup a partir das definições dos *tokens* e especificação das expressões regulares utilizadas para reconhecê-los no código fonte. A representação dos *tokens* gerada pelo Cup é utilizada para representar cada *token* reconhecido.

5. Instruções de uso e teste

Para utilizar o analisador léxico é necessário, primeiro, gerar o código Java referente aos analisadores léxico e sintático e, em seguida, compilar esse código juntamente ao código das classes *Analisador e Main*. Para isso, é necessário garantir que o interpretador Java e as ferramentas de desenvolvimento do java (JDK) estejam instalados na máquina de destino.

Além disso, é necessário que os arquivos ".jar" e ".class" das versões mais recentes das ferramentas JLex e Cup estejam no *classpath* do compilador e interpretador Java para compilar e executar o analisador sintático. Esses arquivos estão disponíveis juntamente ao código fonte do trabalho.

A fim de facilitar o processo de compilação, foi utilizada a ferramenta GNU *make*, que permite realizar todas essas tarefas com um simples comando, que as executa na ordem necessária, respeitando as dependências existentes. Além disso, as regras de compilação do *make* garantem que as versões corretas do JLex e Cup estejam no *classpath*. Sendo assim, para compilar o código do analisador sintático basta executar o comando *make* no terminal estando na raiz do diretório onde foi extraído o fonte deste trabalho.

Por fim, para executar o analisador sintático com os arquivos de teste utilizados no desenvolvimento deste trabalho basta executar o script "main.sh", disponibilizado juntamente ao código do trabalho. Para executar com um arquivo diferente dos utilizados para o teste, basta usar o script "run.sh" e passar o caminho de um arquivo fonte TIGER como parâmetro para o programa. Os scripts "main.sh" e "run.sh" facilitam a execução do analisador léxico, garantindo que as versões corretas do Cup e JLex estão no *classpath* no momento da execução, mas a execução também pode ser feita de forma manual, garantido que as dependências necessárias estejam no *classpath* e executando as classes "*Main*", para utilizar os arquivos de teste, ou "*Analisador*", para executar com um arquivo qualquer.

6. Resultados dos testes

Foram fornecidos dois programas teste, um sintaticamente correto e um com erros de sintaxe, para testar o analisador. O código fonte desses programas de teste está disponível no diretório "Testes", com os arquivos de "teste_certo.tig" e "teste_errado.tig". Os resultados dos testes podem ser obtidos na íntegra executando a "main" do projeto, conforme especificado no item anterior. A seguir, iremos apresentar os códigos dos programas em mais detalhe e apresentar os resultados do analisador.

Começando pelo programa correto, temos uma simples função recursiva que realiza o fatorial de um número. Ela foi extraída do arquivo FATORIAL.TIG, fornecido entre os testes do compilador Tiger fornecido. O código fonte é o seguinte:

```
/* define a recursive function */
let
```

```
/* calculate n! */
function nfactor(n: int): int =
               if  n = 0
                         then 1
                         else n * nfactor(n-1)

in
     nfactor(10)
end
```

O resultado, conforme esperado, indica que a sintaxe está correta, conforme as produções usadas no reconhecimento:

```
Produções da Análise sintática:

type-id ::= id
tyfields1 ::= empty
tyfields ::= id : type-id tyfields1
type-id ::= id
l-value ::= id
exp ::= l-value
exp ::= num
exp ::= exp = exp
exp ::= num
l-value ::= id
exp ::= l-value
l-value ::= id
exp ::= l-value
exp ::= num
exp ::= exp - exp
args1 ::= empty
args ::= exp args1
exp ::= id ( args )
exp ::= exp * exp
exp ::= if exp then exp else exp
fundec ::= function id ( tyfields ) : type_id = exp
dec ::= fundec
decs ::= empty
decs ::= dec decs
exp ::= num
args1 ::= empty
args ::= exp args1
exp ::= id ( args )
expseq1 ::= empty
expseq ::= exp expseq1
exp ::= let decs in expseq end
```

```
Sintaxe Correta.
```

Já para o programa errado, temos uma declaração de um array na parte de declarações e a atribuição de valores ao array na parte de expressões. O código foi extraído do arquivo TESTE6.TIG, fornecido entre os testes do compilador a ser integrado, e modificado, incluindo o erro e outros detalhes. O código fonte é o seguinte:

```
let
  type tipoarranjo = array of int
  var Arranjo:tipoarranjo := tipoarranjo [10] of 0
in
  Arranjo[2] = "nome";
  Arranjo[3] = "data"
  Arranjo[4] = "telefone"
end
```

O erro está em um ponto e vírgula faltando após a segunda expressão, erro muito comumente feito nas mais diversas linguagens. O resultado, conforme esperado, indica um erro de sintaxe, e ponto e vírgula (PVIRG) entre os símbolos esperados após o ponto de erro:

```
Produções da Análise sintática:

ty ::= array of id
tydec ::= type id = ty
dec ::= tydec
type-id ::= id
exp ::= num
exp ::= num
exp ::= id [ exp ] of exp
vardec ::= var id : type-id := exp
dec ::= vardec
decs ::= empty
decs ::= dec decs
decs ::= dec decs
exp ::= num
l-value ::= id [ exp ]
exp ::= l-value
exp ::= str
exp ::= exp = exp
exp ::= num
l-value ::= id [ exp ]
exp ::= l-value
Syntax error
instead expected token classes are [MINUS, TIMES, DIV, AND,
OR, PVIRG, END]
Couldn't repair and continue parse
```

```
Erro de Sintaxe.
```

Repare que o erro não ocorre na declaração do array e nem no acesso ao array, onde o conflito reduce-reduce ocorreria, indicando que ele foi resolvido corretamente nos dois casos. O erro foi especificamente no ponto onde faltava o ponto e vírgula, conforme esperado.

7. Conclusão

Com a execução deste trabalho, foi possível aprender a produzir analisadores sintáticos funcionais, que reconheçam uma linguagem de programação desde o código fonte, o que é útil não só na produção de compiladores, mas também em qualquer outra área em que gramáticas livres de contexto sejam usadas. Foi possível, ainda, conhecer ferramentas que nos ajudem a construir um analisador sintático poderoso de forma fácil, como o Cup e o YACC. Além disso, com o desenvolvimento deste trabalho, foi possível observar a importância de definir uma linguagem de programação sem ambiguidades, ou de uma forma que permita um tratamento para remoção das ambiguidades. Por fim, com a ocorrência dos conflitos, fomos estimulados a compreender melhor a lógica por trás dos algoritmos de reconhecimento sintático LR, interpretando as ações *shift* e *reduce* para descobrir formas de resolver os pontos conflitantes.

Bibliografia

BIGONHA, Mariza. *Slides da disciplina Compiladores I. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2021.*

HUDSON, Scott. *CUP User's Manual.* Cs.princeton.edu. *Disponível em:* <https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>. *Acesso em:* 28 nov. 2021.

PETTER, Michael HUDSON, Scott. *CUP.* Www2.cs.tum.edu. *Disponível em:* <http://www2.cs.tum.edu/projects/cup/>. *Acesso em:* 28 nov. 2021.

BERK, Elliot. *JLex: A lexical analyzer generator for Java™. Latest Version.* Cs.princeton.edu. *Disponível em:* <https://www.cs.princeton.edu/~appel/modern/java/JLex/>. *Acesso em:* 2 nov. 2021.

BERK, Elliot. *JLex: A lexical analyzer generator for Java™. Manual.* Cs.princeton.edu. *Disponível em:*<https://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html *Acesso em:* 2 nov. 2021.