



Universidade de São Paulo

**Instituto de Ciências Matemáticas
e de Computação**

**SCC0206 - Introdução à Compilação
Professor Thiago S. Pardo**

LALG

Uma implementação simplificada de Pascal

Parte III

Implementação da Análise Semântica e Geração de Código

Projeto de Curso Desenvolvido pelos Alunos

**Ubiratan F. Soares (5634292)
Ulisses F. Soares (5377365)
Rafael Pillon Almeida (5634775)**

São Carlos, 02 de julho de 2012

Contextualização

Esse relatório visa detalhar as decisões de projeto tomadas pelo grupo com base no *feedback* das partes I e II do projeto da disciplina, bem como detalhar aspectos de implementação para o trabalho final do curso de Introdução à Compilação. Além disso, esse texto apresenta instruções de como compilar o trabalho e executá-lo para validar casos de teste.

Aprendizado e porquê abondamos o javaCC

Um dos grandes aprendizados dessa disciplina foi o preço pago por se utilizar uma ferramenta do tipo *compiler compiler*. A intenção inicial do grupo de adotar a ferramenta **javaCC**, contradizendo a sugestão de utilizar o combo **lex + yacc / bison** teve origem em dois motivos fundamentais :

- Os três integrantes do grupo possuem grande familiaridade com a linguagem Java, possuindo experiência profissional com a mesma
- Do ano de 2011, na mesma disciplina porém com outro docente, já havíamos utilizado a ferramenta no projeto de curso, de forma imatura porém bem sucedida.

As dificuldades com o javaCC começaram logo na implementação do analisador léxico. Em primeiro lugar, guiados pela recente experiência profissional, procuramos maneiras de integrar a ferramenta com os ambientes de produção para Java Eclipse e Netbeans.

Os plugins existem, porém sofrem os efeitos de pouca manutenção e atualização em relação à própria linguagem Java. Era decepcionante trabalhar com a customização de mensagens de erro, uma vez que qualquer alteração na especificação (arquivo **.jj**) automaticamente sobrescreve as customizações, que são obrigatoriamente colocadas no arquivo *TokenMgrError.java*, por exemplo, para mensagens de erro léxico.

Em segundo lugar, o javaCC não oferece um bom suporte à recuperação de erros. De fato, no primeiro trabalho esse foi um ponto falho, e apenas descobrimos que um bom suporte a erros léxicos dependia de conhecimento das produções da gramática - ou seja, efetivamente do *parser* já estar implementado - para que o número de casos a serem avaliados não fosse de ordem exponencial, remontando à uma máquina de estado dedicada por *token*.

Além disso, mesmo com o suporte do analisador sintático, a própria recuperação de erros ainda se mostra confusa. Conforme excerto retirado do **FAQ** do javaCC

3.12 Can the parser force a switch to a new lexical state?

Yes, but it is very easy to create bugs by doing so. You can call the token manager's method SwitchTo from within a semantic action in the parser like this

```
{ token_source.SwitchTo(name_of_state) ; }
```

However, owing to look-ahead, the token manager may be well ahead of the parser. Take a look at Figure 1.2; at any point in the parse, there are a number of tokens on the conveyer belt, waiting to be used by the parser; technically the conveyer belt is a queue of tokens held within the parser object. Any change of state will take effect for the first token not yet in the queue. Usually there is one token in the queue, but because of syntactic look-ahead there may be many more.

If you are going to force a state change from the parser be sure that, at that point in the parsing, the token manager is a known and fixed number of tokens ahead of the parser, and that you know what that number is.

If you ever feel tempted to call SwitchTo from the parser, stop and try to think of an alternative method that is harder to get wrong.

A falta de um bom suporte ao tratamento de erros foi a principal frustração com a ferramenta. De fato, no projeto do ano anterior, a recuperação de erros, fossem léxicos, sintáticos ou semânticos não era uma especificação do projeto de curso, de forma que não tivemos como sentir essa dificuldade antes.

Por fim, tivemos o agravante de os três integrantes do grupo estarem com excesso de tarefas profissionais na época da entrega da segunda parte do projeto. De fato, o projeto foi iniciado com antecedência, mas na época da entrega - já com o trabalho atrasado em 3 dias - o único aluno com disponibilidade de tempo não foi capaz de lidar com as dificuldades na recuperação de erros, ainda que o analisador sintático estivesse funcionando para os dois casos de teste básicos, presentes no documento de especificação da LALG. Faltavam casos adicionais em quantidade para compor uma bateria consistente para o *parser*. Sendo assim, o grupo assumiu o fracasso da entrega, e se preparou para uma nova abordagem para o projeto final.

Reimplementado o projeto, utilizando Orientação a Objetos

Um das carências sentidas com a automatização com javaCC é sintetizada com a seguinte pergunta:

- “Se o código gerado é **Java**, por que um estado léxico não é um objeto manipulável?”

Recomeçamos a implementação do trabalho desde o Analisador Léxico a partir daí, de maneira que os três integrantes do grupo estivessem à vontade com a proposta “o mais OO possível” para o projeto. Decidimos utilizar a IDE **Netbeans** para auxiliar o desenvolvimento, bem com a solução Dropbox para auxiliar o trabalho remoto.

Assim como no javaCC, nosso token (**Token.java**) também é um objeto. Contudo, nossos estados léxicos agora também são objetos, ao contrário da estratégia do javaCC, onde o estado léxico não tem uma representação. A classe base do projeto dos estados léxicos é a classe abstrata **State.java** (pacote *compiler.lexer*). Ela permite que os estados de início do autômato (**StartState.java**), parado do autômato (**StopState.java**) bem como os demais estados sejam manipulados como objetos.

O projeto então foi inteiramente recodificado. No pacote *compiler.t1* está a primeira parte do projeto inteiramente repensada. Nos pacotes *compiler.t2a* e *compiler.t2b* estão as evoluções do analisador sintático, com suporte e sem suporte à recuperação de erros sintáticos respectivamente. A essa altura, já tínhamos a noção da necessidade da classe **CompilationError.java**, à qual deveria ser estendida para qualquer erro que pudesse advir do processo de compilação de um passo (léxico, sintático ou semântico).

Para auxiliar na construção do analisador sintático - e posteriormente no encaixe das rotinas semânticas - o grupo desenhou os grafos sintáticos para a LALG. Todos os diagramas de grafos sintáticos estão nos arquivos **syntatic_graph1.png** e **syntatic_graph2.png**, que estão no diretório do projeto para referência.

Para realizar o tratamento no modo pânico, foram calculados os conjuntos *FOLLOW* - e por consequência os conjuntos *FIRST* - para cada símbolo da gramática. As etapas e relatórios desse cálculo estão detalhadas nos arquivos **first.txt** e **follow.txt** e a implementação do tratamento sintático modo pânico, juntamente com a recuperação de erros, se valem desse cuidadoso trabalho. Só depois descobrimos que podíamos ter usado a ferramenta JFlap...

Nos pacotes *compiler.t3a* e *compiler.t3b* estão as implementações da análise semântica e geração de código para a LALG. Na nossa concepção, encapsulamos mesmo os conceitos de código-fonte, código-objeto e do próprio compilador em classes. O resultado final dessa estratégia, executada em modo intensivo desde o começo de junho, se reflete na classe **Main.java** (pacote *compiler.t3b*), que é absolutamente expressiva enquanto ao seu propósito :

```

package compiler.t3b;

import compiler.*;
import compiler.Compiler;
import java.util.List;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        SourceCode source = new SourceCode();
        Compiler compiler = new CompilerImpl();
        ObjectCode object = null;

        //leitura do arquivo de código-fonte
        try {
            source.load(args[0]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Parametros: <arquivo fonte> [<arquivo objeto>]");
            return;
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo \"" + args[0] + "\" nao encontrado.");
            return;
        } catch (IOException e) {
            System.out.println("Falha ao ler arquivo \"" + args[0] + "\".");
            return;
        }

        //compilação
        try {
            compiler.init(args);
            System.out.println("Compilando \"" + args[0] + "\"...");
            object = compiler.compile(source);
        } catch (LexicalException e) {
            int line = e.getToken().getLine() + 1;
            int column = e.getToken().getColumn() + 1;
            System.out.printf("Erro léxico na linha %d, coluna %d: %s%n",
                line, column, e.getMessage());
            System.out.println("Compilação abortada.");
            return;
        } catch (CompilerException e) {
            e.printStackTrace();
            return;
        }

        //listar erros sintáticos
        List<CompilationError> errors = ((CompilerImpl) compiler).getErrors();
        for (CompilationError e : errors) {
            int line = e.getToken().getLine() + 1;
            int column = e.getToken().getColumn() + 1;
            if (e instanceof SyntacticError) {
                System.out.printf(
                    "Erro sintático na linha %d, coluna %d: %s%n",
                    line, column, e.getMessage());
            } else if (e instanceof SemanticError) {
                System.out.printf(
                    "Erro semântico na linha %d, coluna %d: %s%n",
                    line, column, e.getMessage());
            }
        }

        if (errors.isEmpty()) System.out.println("Compilação bem-sucedida.");
        else System.out.println("Compilação abortada.");

        //escrita do arquivo de "código-objeto" (fachada)
        try {
            if (args.length > 1 && object != null) {
                object.println();
                object.writeFile(args[1]);
                System.out.println("Escrito arquivo \"" + args[1] + "\".");
            }
        } catch (IOException e) {
            System.out.println("Falha ao escrever arquivo \"" + args[1] + "\".");
            return;
        }

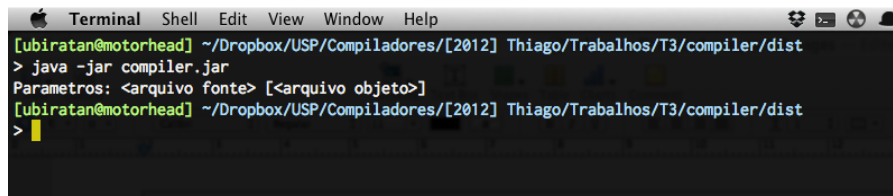
        return;
    }
}

```

Testando a Implementação

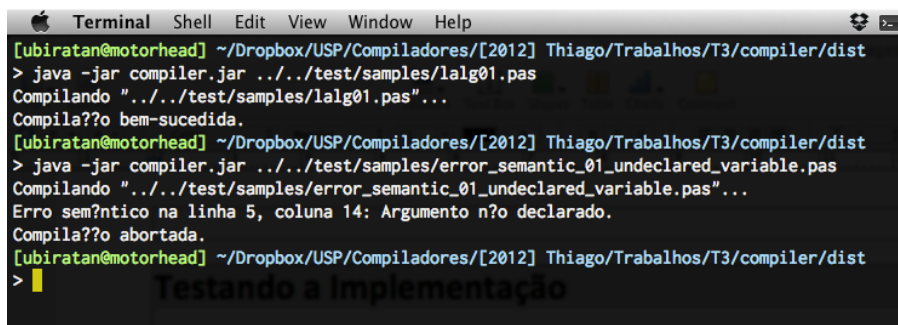
Para executar a compilação do projeto, é recomendado a utilização do Netbeans. Para isso, basta abrir o projeto e utilizar a opção **Run > Clean and Build Project** para compilar e gerar o arquivo **compilar.jar**, que por padrão, ficará no diretório *compiler/dist*.

A execução pode ser feita pelo terminal interativo do Netbeans, ou para propósitos de testes com script, pela linha de comando. O executável pede um parâmetro obrigatório com o caminho para o arquivo-fonte e um opcional, com o caminho do arquivo com o código-objeto impresso.



```
Terminal Shell Edit View Window Help
[ubiratan@motorhead] ~/Dropbox/USP/Compiladores/[2012] Thiago/Trabalhos/T3/compiler/dist
> java -jar compiler.jar
Parametros: <arquivo fonte> [<arquivo objeto>]
[ubiratan@motorhead] ~/Dropbox/USP/Compiladores/[2012] Thiago/Trabalhos/T3/compiler/dist
>
```

Para teste, é obrigatório a passagem de um arquivo-fonte, e opcionalmente, o arquivo de saída para o código-objeto resultante.



```
Terminal Shell Edit View Window Help
[ubiratan@motorhead] ~/Dropbox/USP/Compiladores/[2012] Thiago/Trabalhos/T3/compiler/dist
> java -jar compiler.jar ../../test/samples/lalg01.pas
Compilando "../../test/samples/lalg01.pas"...
Compila??o bem-sucedida.
[ubiratan@motorhead] ~/Dropbox/USP/Compiladores/[2012] Thiago/Trabalhos/T3/compiler/dist
> java -jar compiler.jar ../../test/samples/error_semantic_01_undeclared_variable.pas
Compilando "../../test/samples/error_semantic_01_undeclared_variable.pas"...
Erro sem?ntico na linha 5, coluna 14: Argumento n?o declarado.
Compila??o abortada.
[ubiratan@motorhead] ~/Dropbox/USP/Compiladores/[2012] Thiago/Trabalhos/T3/compiler/dist
>
```

Referências

1. **Thiago Salgueiro Pardo** - *Notas de Aula*. São Carlos, 2012.
2. **T. Kowaltowski** - *Implementação de Linguagens de Programação*. Guanabara Dois, 1983
3. **A. V. Aho, M. S. Lam, J. D. Ullman** - *Compilers: Principles, Techniques, and Tools (2nd edition)*. Addison Wesley, 2006
4. **Thiago S. Pardo** - *Notas de Aula*. São Carlos, 2010.
5. **A. W. Appel and J. Palsberg** - *Modern Compiler Implementation in Java (2nd edition)*. Cambridge University Press, 2002