Compiladores - Trabalho Prático

Paulo Viana Bicalho[1]

[1]Departamento de Ciência da Computação – Universidade Federal de minas gerais (UFMG)

{p.bicalho}@dcc.ufmg.br

Resumo. *Este relatório descreve a implementação de um compilador de uma determinada linguagem para bytecode Java. O processo de desenvolvimento foi dividido em três partes principais: Desenvolvimento do front-end, Desenvolvimento do back-end, e Integração de ambos. O principal objetivo deste trabalho é o de praticar conceitos e técnicas estudados na disciplina de Compiladores. Para avaliar o compilador desenvolvido, foram realizados uma série de testes e os resultados obtidos ficaram dentre os esperados.*

1. Introdução

Computadores e máquinas computacionais no geral, são capazes de executar uma série de comandos para executar determinadas tarefas. Para tanto, eles possuem um conjunto de instruções, onde cada instrução desempenha uma tarefa especifica. O grande problema, é que de maneira geral, estas instruções são pouco intuitivas, o que torna difícil a arte de programação. Para solucionar tal problema, surgiram as linguagens de programação. Uma linguagem de programação aumenta o nível de abstração das instruções da máquina, disponibilizando comandos e procedimentos que possuem mair valor semântico.

Como advento destas linguagens, foi então, necessário desenvolver um tipo de programa que seja capaz de converter ou traduzir um programa fonte desta linguagem mais abstrata, para um programa objeto da linguagem alvo. Tais programas são chamados de compiladores e são vitais nos dias atuais para o sucesso da computação no geral.

Este trabalho possui como objetivo, implementar um compilador de uma determinada linguagem para bytecode da JVM (*Java Virtual Machine*). O desenvolvimento foi dividido em três partes principais:

- *Front-end*: Responsável por realizar a análise do programa fonte e geração de uma forma intermediária do programa;
- *Back-end*: Responsável pela síntese, recebe como entrada a saída do *front-end* e constrói o programa objeto;
- Integração: Integra as duas partes anteriores;

2. A Gramática

A gramática da linguagem de origem é dada na figura 1.

3. *Front-End*

A parte da análise geralmente é subdividida em três fases: Análise Léxica, Análise Sintática e Análise Semântica. Cada uma destas análises possui um propósito específico que se complementam. A figura 2 exibe uma decomposição típica do *front-end* em fases

$$
\begin{array}{rcl}
program & \rightarrow & block \\
block & \rightarrow & \texttt{\{}\ decls\ stmts\ \texttt{\}} \\
decls & \rightarrow & decls\ decl \mid \epsilon \\
decl & \rightarrow & type\ \mathbf{id}\ \texttt{;} \\
type & \rightarrow & type\ \texttt{[}\ \mathbf{num}\ \texttt{]} \mid \mathbf{basic} \\
stmts & \rightarrow & stmts\ stmt \mid \epsilon \\
 & & \\
stmt & \rightarrow & loc\ \texttt{=}\ bool\ \texttt{;} \\
 & \mid & \mathbf{if}\ \texttt{(}\ bool\ \texttt{)}\ stmt \\
 & \mid & \mathbf{if}\ \texttt{(}\ bool\ \texttt{)}\ stmt\ \mathbf{else}\ stmt \\
 & \mid & \mathbf{while}\ \texttt{(}\ bool\ \texttt{)}\ stmt \\
 & \mid & \mathbf{do}\ stmt\ \mathbf{while}\ \texttt{(}\ bool\ \texttt{)}\ \texttt{;} \\
 & \mid & \mathbf{break}\ \texttt{;} \\
 & \mid & block \\
loc & \rightarrow & loc\ \texttt{[}\ bool\ \texttt{]} \mid \mathbf{id} \\
bool & \rightarrow & bool\ \texttt{||}\ join \mid join \\
join & \rightarrow & join\ \texttt{\&\&}\ equality \mid equality \\
equality & \rightarrow & equality\ \texttt{==}\ rel \mid equality\ \texttt{!=}\ rel \mid rel \\
rel & \rightarrow & expr\ \texttt{<}\ expr \mid expr\ \texttt{<=}\ expr \mid expr\ \texttt{>=}\ expr \mid \\
 & & \quad expr\ \texttt{>}\ expr \mid expr \\
expr & \rightarrow & expr\ \texttt{+}\ term \mid expr\ \texttt{-}\ term \mid term \\
term & \rightarrow & term\ \texttt{*}\ unary \mid term\ \texttt{/}\ unary \mid unary \\
unary & \rightarrow & \texttt{!}\ unary \mid \texttt{-}\ unary \mid factor \\
factor & \rightarrow & \texttt{(}\ bool\ \texttt{)} \mid loc \mid \mathbf{num} \mid \mathbf{real} \mid \mathbf{true} \mid \mathbf{false}
\end{array}
$$

Figure 1. Gramática

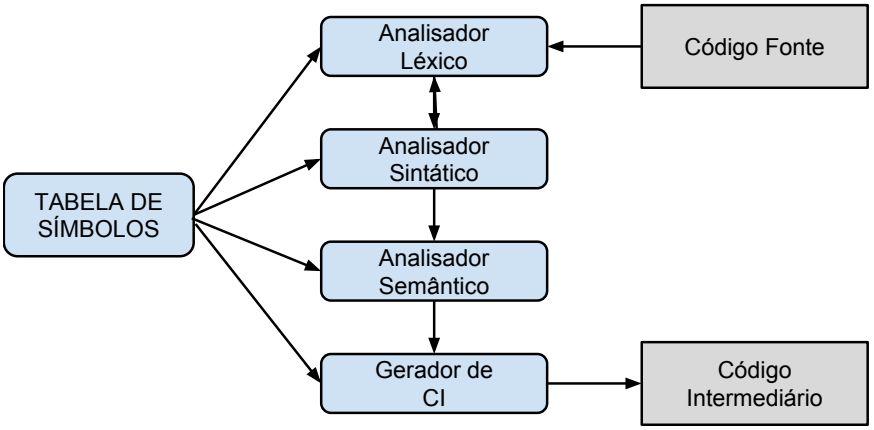

Figure 2. Fases do *Front-End* de um compilador

3.1. Análise Léxica

O Analisador Léxico é a única parte do compilador que tem acesso direto ao código fonte original. Seu principal papel é o de realizar a leitura de tal arquivo, agrupando sequências de caracteres que são de alguma forma significativas. Estas sequências, chamadas de lexemas, são palavras que possuem algum valor semântico associados à elas.

O Analisador Léxico deve garantir que todos os lexemas lidos pertencem à gramática da linguagem fonte e deve ser capaz de associar a cada lexema o tipo deste. Assim, para cada lexema o analisador produz como saída um token na forma:

$$< tipo - token, lexema >$$

O Analisador desenvolvido neste trabalho realiza casamento de padrões para extrair os lexemas e associar tipos a estes. A implementação foi baseada na secção 2.6 do livro texto da disciplina [Aho et al. 2006].

3.2. Análise Sintática

O principal papel deste Analisador é o de utilizar os *tokens* produzidos pelo Analisador Léxico e criar uma árvore de derivação sintática. A construção desta árvore é realizada seguindo as regras de produções da gramática e geralmente cada nó desta árvore esta associada à uma operação e cada filho deste nó os operandos.

Esta análise pode ser realizada utilizando diversas técnicas diferentes. Estas técnicas podem ser agrupadas em dois grandes grupos:

- *Top-Down*: As técnicas que utilizam esta abordagem iniciam a construção da derivação sintática a partir das regras iniciais da gramática fazendo o refinamento a cada passo.
- *Bottom-Up*: A construção é realizada inicialmente pelos nós inferiores da árvore aumentando o nível a cada etapa.

Neste trabalho foi implementado um Analisador Sintático preditivo (*Top-Down*) conforme descrito na seção 2.4 do livro texto [Aho et al. 2006].

3.3. Análise Semântica

Durante as etapas da Análise Léxica e Sintática os lexemas, seus tipos e algumas outras informações pertinentes como tamanho de *arrays* são armazenados na tabela de símbolos. O Analisador Semântico utiliza destes valores para validar as operações executados no código fonte original.

Nesta verificação ocorre principalmente a checagem de tipos, que verifica se os operandos das operações são válidos para a mesma.

O Analisador Semântico desenvolvido neste trabalho foi baseado na descrição da secção 2.6 do livro texto da disciplina [Aho et al. 2006].

3.4. Geração de Código Intermediário

Realizadas as análises, ou juntamente com elas, o *Front-End* é capaz de gerar um código intermediário (CI) que possui a mesma semântica que o código fonte original.

A grande utilidade do CI é o de permitir a separação e a independência entre o *Front-End* e o *Back-End*. Além disso, como o geralmente este tipo de código é muito simples, facilita o processo de otimização.

O CI desenvolvido é um CI de três endereços baseado na seção 2.8 do livro texto [Aho et al. 2006].

Para facilitar o processo de criação do *back-end*, foram utilizadas algumas convenções para a construção do código intermediário

1. Todas as variáves não temporárias são representadas na forma: var_$type(x)$ onde $type$ é o tipo da variável e x o identificador da mesma;
2. Todas as constantes são identificadas da seguinte forma: var_ $type$(const(x)) onde $type$ representa o tipo da variável, x é o identificador e $const$ informa que é alguma constante;
3. As operações aritméticas também são diferenciadas de acordo com o tipo da variável de destino.;
4. As variáveis temporáriras são nomeadas da seguinte forma: $t1, t2, t3, ..., tn$;

3.5. Código

O Código do *Front-End* foi desenvolvido em **C++ 11** em um ambiente Unix e se encontra da pasta *front* do anexo enviado com esta documentação. A estruturação foi baseada nos exemplos do livro texto e o código foi organizados em sub-pastas sendo:

- inter: Geração de Código Intermediário / Analise Semântica;
- lexer: Analisador Léxico;
- parser: Analise Sintática;
- symbols: Tabela de simbolos;

Um Makefile acompanha o código para facilitar a compilação. A chamada do executável do *front-end* espera como parâmetro o arquivo fonte e deve ser realizada da seguinte forma:

```
./frontend arquivo_fonte
```

A saida do programa é a saida padrão e caso seja necessário, pode ser redirecionada através do comando $>$ do bash;

```
./frontend arquivo_fonte.p > arquivo_saida.i
```

4. *Back-End*

O *Back-End* desenvolvido recebe como entrada, a saida do *Front-End* e realiza o mapeamento deste código em bytecode da JVM.

A JVM é uma máquina de pilha e possui um grande número de instruções. Muitas destas instruções não foram utilizadas pois a gramática da linguagem fonte é muito simples.

Como referencia de estudo das instruções, foi utilizado a wikipédia principalmente a página `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`.

4.1. Código

O código do *Back-End* também foi desenvolvido em **C++ 11** em um ambiente Unix e se encontra na pasta *back*. Como a quantidade arquivos necessários é muito menor que a do *Front-End*, não foi necessário a estruturação em sub-pastas.

Um Makefile acompanha o código e a chamada do executável deve ser executada da seguinte forma:

```
./backend arquivo_intermediario.i
```

O parâmetro esperado é um arquivo com um código na representação intermediária (saída do *Front-End*). A saída é a saída padrão do sistema mas pode ser redirecionada.

```
./backend arquivo_intermediario.i > arquivo_saida.jvm
```

5. Integração

A integração das duas etapas foi realizada na pasta principal do anexo enviado e possui somente um arquivo que realiza a chamada do *Front-End* seguido pela chamada do *Back-End*.

O executável espera três parâmetros: Arquivo fonte, Arquivo de saída do *Front-End*, Arquivo de saída do *Back-End*.

```
./pauloCompiler arquivo_font.p arquivo_ci.i arquivo_final.jvm
```

Um Makefile acompanha o código para facilitar a compilação.

6. Testes

Para avaliar o compilador proposto foram realizados vários testes. Como dito anteriormente, tanto o *Front-End*, o *Back-End* quanto o compilador encontrado possuem um Makefile que acompanha o código. Tal Makefile contém uma seção de testes que pode ser invocada da seguinte maneira

```
make test
```

Ao realizar tal chamada, os arquivos dentro da pasta $test_in$ são passados como parâmetro de entrada para o executável e as saídas são armazenadas na pasta $test_out$.

7. Conclusão

Este relatório discutiu a implementação de um compilador para uma determinada gramática. Tal compilador foi dividido em duas grandes partes: *Front-End* e *Back-End*. O *Front-End* é o responsável de realizar as análises no código fonte e gerar uma representação intermediária do mesmo enquanto o *Back-End* deve converter este código intermediário em bytecode da JVM.

A compilador foi testado com uma gama de arquivos de teste e os resultados obtidos ficaram dentre os esperados. Desta forma, este trabalho cumpriu seus objetivos pois proporcionou ao aluno vivenciar a experiência e as dificuldades de se projetar e implementar um compilador.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.