UNIVERSIDADE ESTADUAL DO CEARÁ

UECE - CCT - Curso de Ciência da Computação
Disciplina: Compiladores
Semestre: 2018.1

Relatório Final de Compiladores

Alunos:	Matrícula:
Bruno Silva Barbosa	1284770
Yan Augusto Gurgel da Silva	1196463

Fortaleza – CE
2018.1

Sumário

1 Objetivos

Compiladores são programas grandes e complexos, e geralmente incluem centenas de milhares, ou mesmo milhões, de linhas de código, organizadas em múltiplos subsistemas e componentes. As várias partes de um compilador interagem de maneira complexa. Decisões de projeto tomadas para uma parte do compilador têm ramificações importantes para outras. Assim, o projeto e a implementação de um compilador é um exercício substancial em engenharia de software.

Um compilador contém um microcosmo da ciência da computação. Faz uso prático de algoritmos gulosos (alocação de registradores), técnicas de busca heurística (agendamento de lista), algoritmos de grafo (eliminação de código morto), programação dinâmica (seleção de instruções), autômatos finitos e autômatos de pilha (análise léxica e sintática) e algoritmos de ponto fixo (análise de fluxo de dados). Lida com problemas, como alocação dinâmica, sincronização, nomeação, localidade, gerenciamento da hierarquia de memória e escalonamento de pipeline. Poucos sistemas de software reúnem tantos componentes complexos e diversificados. Trabalhar dentro de um compilador fornece experiência em engenharia de software, difícil de se obter com sistemas menores.

Então o principal objetivo deste trabalho, foi aplicar vários conceitos e técnicas aprendidas na disciplina de Compiladores, de forma prática e possibilitar uma experiência que reúne tantos conceitos variados, trazendo um possível primeiro contato com um complexo projeto da engenharia de software e grande aprendizagem.

2 Introdução Geral

2.1 Compiladores

Compiladores são programas de computador que traduzem um programa escrito em uma linguagem em um programa escrito em outra linguagem. Para essa tradução, o compilador deve tanto entender a forma (sintaxe) quanto o sentido (semântica) da linguagem de entrada, e ainda, as regras que controlam a sintaxe e a semântica da linguagem de saída. Portanto, para isso, o compilador precisa de um esquema de mapeamento de conteúdo da linguagem-fonte para linguagem-alvo.

Figura 1: Estrutura compilador

Compilador tem um front-end para lidar com a linguagem de origem e um back-end para lidar com a linguagem-alvo (veja figura 1). Ligando ambos, ele tem uma estrutura formal para representar o programa num formato intermediário cujo significado é, em grande parte, independente de qualquer linguagem. Para melhorar a tradução, compiladores muitas vezes incluem um otimizador, que analisa e reescreve esse formato intermediário.

O front-end concentra-se na compreensão do programa na linguagem-fonte. Já o back-end, no mapeamento de programas para código-alvo (máquina-alvo). O front-end deve codificar seu conhecimento do programa fonte em alguma estrutura para representação intermediária (IR) torna-se a representação definitiva do compilador para o código que está sendo traduzido.

Em um compilador de duas fases, o front-end deve garantir que o programa-fonte esteja bem formatado, e mapear aquele código para a IR. Já o back-end, mapeia o programa em IR para o conjunto de instruções da linguagem-alvo. Como este último só processa a IR criada pelo front-end, pode assumir que a IR não contém erros sintáticos ou semânticos.

Na prática, a divisão conceitual de um compilador geralmente é composto por 3 fases-front-end, seção intermediária, ou otimizador, e o back-end. O front-end trata do entendimento do programa-fonte e do registro dos resultados de sua análise em forma de IR. A seção do otimizador focaliza a melhoria do formato IR. O back-end precisa mapear o programa transformando em IR em linguagem-alvo.

O termo otimização implica que o compilador descobre um solução ótima para algum problema. Às vezes, os problemas que surgem nessa fase são tão complexo que não podem, na prática, ser solucionados de forma ótima.

2.2 Front-end

Antes que o compilador possa traduzir uma expressão para código executável da máquina-alvo, precisa entender tanto sua sintaxe, quanto sua semântica. O front-end determina

se o código de entrada está bem formatado nestes termos. Se for detectado que o código é válido, então é criada uma representação deste código na representação intermediária do compilador; se não, informa ao usuário com mensagens de erro de diagnóstico para identificar os problemas com o código.

2.2.1 Sintaxe

Para verificar a sintaxe do programa de entrada, o compilador precisa comparar a estrutura do programa com uma definição para a linguagem. Isto exige uma definição formal apropriada. Matematicamente, a linguagem de origem é um conjunto, normalmente infinito, de strings definido por algum conjunto finito de regras, chamado gramática. Dois passos separados no front-end, chamados scanner e parser, determinam se o código de entrada é ou não, de fato, um membro do conjunto de programas válidos definidos pela gramática.

2.2.1.1 Scanner

O primeiro passo para entender a sintaxe de uma sentença qualquer é identificar palavras distintas no programa de entrada e classificar cada palavra com uma classe gramatical. Em um compilador, esta tarefa fica com um passo chamado scanner (ou analisador léxico). O scanner apanha um fluxo de caracteres e o converte para um fluxo de palavras classificadas. Em resumo o scanner tem a função de transformar string de caracteres em um fluxo de palavras (tokens).

2.2.1.2 Parser

Com a capacidade de gerar palavras de fluxo com o scanner, o compilador também tem que ser capaz de verificar se essas palavras representa alguma sentença na linguagem-fonte. Ele deriva uma estrutura sintática(em geral uma árvore-AST) para o programa, encaixando as palavras em um modo gramatical da linguagem-fonte. Esse encaixe é feito geralmente a partir de uma gramática livre de contexto que definem de forma recursiva componentes que podem fazer-se uma expressão e a ordem em que devem aparecer.

Uma sentença gramaticalmente correta pode não ter significado. Ela pode ter as mesmas regras gramaticais e na mesma ordem de uma sentença correta, mas não tem nenhum significado na linguagem de programação fonte.

2.2.1.3 AST

Árvores sintáticas abstratas (*Abstract syntax tree* -AST) são estruturas de dados que auxiliam na representação das estruturas sintáticas e podem conter informações extras que irão ajudar nas próximas etapas do compilador (análise semântica e no back-end - será explicado no tópico 2.2.1.5 e 2.3), de acordo com alguma gramática formal. É uma representação abstrata da estrutura sintática de um código fonte escrito em uma linguagem de programação. Cada nó da árvore denota um construtor no código fonte. A sintaxe é abstrata no sentido que ela não representa cada detalhe que aparece na sintaxe real. Uma árvore é utilizada como estrutura, pois tem facilidade para especificar estruturas de recursividade e recursão.

O compilador atravessa muitas vezes essa árvore, então é crucial que essa operação de travessia seja uma operação simples. São executadas um conjunto específico de operações pelo compilador, dependendo do tipo de cada nó. Então, frequentemente é usado o padrão Visitor (próximo tópico) para realizar essas operações de travessia.

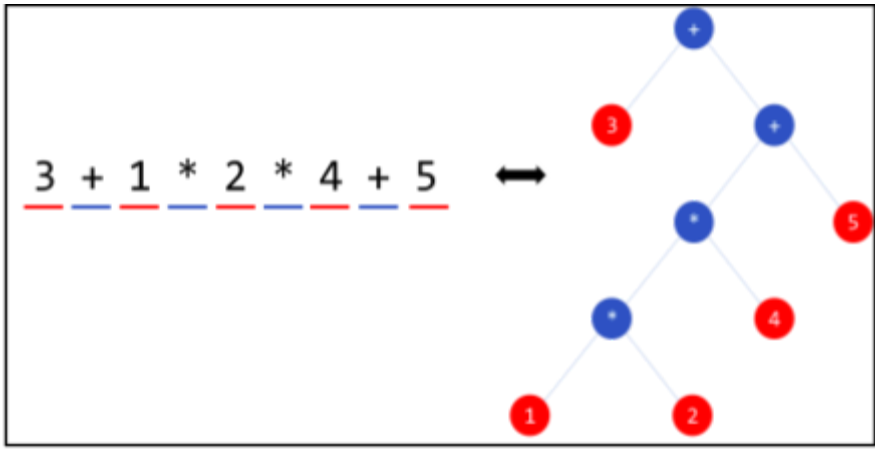


Figura 2: Exemplo de AST

2.2.1.4 Padrão Visitor

Essencialmente, o padrão Visitor permite adicionar novas funções virtuais a uma família de classes, sem modificar-las. Para isso, é criada uma classe de visitantes que implementa todas as especializações apropriadas (métodos das várias classes). Um resultado prático é a habilidade de adicionar novas funcionalidades a estruturas de um objeto pré-existente sem a necessidade de modificá-las. Esse padrão tem uma suma importância em um projeto de compilador.

2.2.1.5 Tabelas

Depois que é feita a análise léxica e sintática, existe uma terceira fase que é feita a análise semântica - elabora as implicações das expressões apenas validadas e toma a ação apropriada. Nesta fase, muitas vezes é construído a Tabela de Símbolos, pois até então (fase de análise léxica e sintática) não existem informações suficientes para poder preenchê-la.

Muitos compiladores configuram a tabela no tempo de análise lexical para as várias variáveis no programa e preenchem informações sobre o símbolo mais tarde durante a análise semântica quando é conhecida mais informações sobre a variável.

A tabela de símbolos é uma estrutura de dados importante criada e mantida pelos compiladores para armazenar informações sobre a ocorrência de várias entidades, como nomes de variáveis, nomes de funções, objetos, classes, interfaces, etc.

2.2.1.6 Representações intermediárias(IRs)

O último aspecto tratado no front-end de um compilador é a geração de um formato de IR do código. Compiladores usam diversos tipos diferentes de IR, dependendo da linguagem-fonte, da linguagem-alvo e das transformações específicas que o compilador aplica. Algumas IRs representam o programa como um grafo, outras assemelham-se a um programa em código assembly ou a própria AST. Para cada construção na linguagem-fonte o compilador precisa de uma estratégia para como implementá-la no formato IR do código. Estas estratégias podem afetar a capacidade do compilador de transformar e melhorar o código.

2.3 Back-end

O back-end do compilador atravessa o formato IR do código e emite código para o código-alvo; seleciona as operações da linguagem-alvo para implementar cada operação da IR; escolher uma ordem em que as operações serão executadas de modo mais eficiente; decide quais valores residirão nos registradores e quais na memória. Veja novamente a figura 1, a parte inferior tem a representação do back-end.

3. Introdução à construção do compilador

3.1 MiniJava

MiniJava é um subconjunto da linguagem Java, portanto pode ser executado pela JVM (*Java Virtual Machine*), e contém algumas restrições comparado com o Java. MiniJava tem um caráter mais didático para ensino de projeto e implementação de compiladores.

MiniJava restringe a linguagem Java para ter operações apenas algumas operações com inteiros, booleanos, vetores de inteiros, números de ponto flutuante, classes abstratas, strings, vetores de outros tipos etc. Não existe sobrecarga de métodos (em java explicitado com a tag '@override'), métodos estáticos (explicitado na assinatura do método com 'abstract'), exceto pelo método *main* da classe principal do programa, classes não podem ter construtores , System.out.println() só poderá imprimir inteiros etc.

```
class Factorial{
  public static void main(String[] a){
        System.out.println(new
Fac().ComputeFac(10));
  }
}

class Fac {
  public int ComputeFac(int num){
        int num_aux ;
        if (num < 1)
           num_aux = 1 ;
        else
           num_aux = num *
(this.ComputeFac(num-1)) ;
        return num_aux ;
  }
```

Figura 3: Exemplo código em MiniJava

3.2 JFlex

Ferramentas automáticas para geração de scanners são comuns, que processam uma descrição matemática da sintaxe léxica da linguagem e produzem um reconhecedor rápido. Mas também, como alternativa, muitos compiladores utilizam scanners codificados à mão.

O programa JFlex tem a capacidade de criar uma classe Java a partir de um arquivo de texto com a extensão jflex (arquivo de configuração léxica). Esse arquivo deve indicar as regras da construção desta classe. A classe que é construída faz a análise léxica de qualquer arquivo de texto.

A partir das configurações definidas, o JFlex ler esse arquivo, que contém, por exemplo algumas Expressões Regulares (R.E) que define algumas estruturas sintáticas, e constrói NFA

(Nondeterministic finite automaton). É simples a conversão de um R.E para um NFA, existem procedimentos simples e pouco custoso para essa conversão.

Em uma NFA existem situações nos quais tem que escolher qual transição deverá seguir, com isso, em um NFA, antes de escolher qual "caminho" seguir, deverá checar todas a possibilidades. Esse processo é chamado de Backtrack e esse processo será muito custoso caso seja em um automata muito grande.

Então o scanner, no caso via Jflex, converte o NFA em DFA já que neste não precisa ficar fazendo Backtrack para escolher qual a transição percorrer. Além disso, a conversão de NFA para DFA, existem algoritmos que fazem essa conversão de forma consistente (cada estado do DFA irá representar um conjunto de estados no NFA). Ver figura 3.

```
gen-scanner:
     [java] Reading "src/Scanner/minijava.jflex"
     [java] Constructing NFA : 220 states in NFA
     [java] Converting NFA to DFA :
     [java] ..................................................................................
     [java] 140 states before minimization, 129 states in minimized DFA
     [java] Old file "src/Scanner/MyScanner.java" saved as "src/Scanner/MyScanner.java~"
     [java] Writing code to "src/Scanner/MyScanner.java"
```

Figura 4: Log de saída do Jflex ao construir o scanner

Resumindo, o JFlex é um programa que tem a função de criar o scanner automaticamente a partir de especificação definida (.jflex) criando primeiramente uma NFA, depois utiliza algum algoritmo de conversão e produzindo uma DFA equivalente, depois traduz este em código, então resultando o scanner.java .

A partir das especificações, o JFlex gera um arquivo .java com uma classe que contém código para o scanner. A classe terá um construtor tomando um java.io.Reader a partir do qual a entrada é lida. A classe também terá uma função yylex() que executa o scanner e que pode ser usado para obter o próximo token da entrada (next_token()).

3.2.1 Configurações arquivo JFlex

Como já mencionado, o arquivo de configurações do JFlex terá as especificações do scanner. Por facilidade o JFlex pode ser integrado com Apache Ant que auxilia na execução no JFlex e o gerenciamento de dependências. O Ant também será utilizado com o Cup que construirá o parse onde será apresentado mais à frente.

O Apache Ant é uma biblioteca Java e é uma ferramenta de linha de comando cuja a missão é gerar processos em arquivos de compilação como alvos e pontos de extensão dependentes uns dos outros. Ant fornece uma série de tarefas internas que permitem compilar, montar, testar e executar aplicativos Java. No projeto, existe um script ANT que pode ser executado para lidar com as dependências e criar a classe scanner.java.

Para melhor entendimento, no Apêndice A contém um exemplo de arquivo de configurações que contém as definições e as expressões regulares que formam todo o léxico necessário(tudo comentado para melhor entendimento) para criar o Scanner e funcionar corretamente para os exemplos enviados por email ao professor.

3.3 CUP

CUP significa Construção de Parsers úteis e é um gerador de analisador LALR(1) - é uma opção menos poderosa do que o analisador LR(1), mas é bem mais poderoso que o analisador SLR(1), embora todos eles usem a mesma regra de produção.

O uso do CUP envolve a criação de uma especificação simples com base na gramática para o qual um analisador é necessário, juntamente com a construção de um scanner (apresentado na seção anterior) capaz de quebrar caracteres em tokens significativos.

3.3.1 Configurações arquivo JFlex

De forma semelhante ao JFlex, existe um arquivo de especificação (.cup) que contém informações auxiliam na criação do parser. Neste arquivo contém declarações como não-terminais, terminais e a associação da classe a cada um. Nesse caso, os terminais são declarados como sem tipo ou com o tipo Integer. Também existe a precedência e a associatividade entre os terminais e assim como a gramática relativa a linguagem de origem.

Para melhor entendimento o Apêndice B contém um exemplo de arquivo de configurações (tudo comentado para melhor entendimento) que contém as definições para análise sintática necessário para criar o Parser e funcionar corretamente para os exemplos enviados por email ao professor.

O CUP, neste caso, irá produzir dois arquivos fonte Java contendo partes do analisador gerado: sym.java e parse.java. A classe sym contém uma série de declarações constantes, uma para cada símbolo terminal. Isso geralmente é usado pelo scanner para se referir a símbolos. A classe parser implementa o parser em si.

Cada símbolo terminal e não-terminal é representado em tempo de execução com um objeto Symbol. No caso dos terminais, estes são devolvidos pelo scanner e colocados na pilha de análise. No caso de não-terminais, substituem uma série de objetos Symbol na pilha de análise sempre que o lado direito de alguma produção é reconhecido. Para indicar ao analisador quais tipos de objeto devem ser usados para quais símbolos, declarações terminais e não terminais são usadas.

A classe de parser gerada fornece uma série de tabelas para uso pela estrutura geral. São fornecidas três tabelas: a tabela de produção fornece o número de símbolo do lado esquerdo não terminal, juntamente com o comprimento do lado direito, para cada produção na gramática, a tabela de ação indica que ação (shift, reduzir ou erro) deve ser tomado em cada símbolo lookahead quando encontrado em cada estado, e a tabela de reduce-goto que indica qual estado mudar para depois shift (sob cada não-terminal de cada estado).

3.3.2 Implementação Padrão Visitor

AST contém nós que representam estruturas diferentes, tais como: operadores; variáveis; expressões matemáticas. Para cada estrutura (nó do AST), existe operações diferentes como: formatação; verificação de tipos; verificação se as variáveis estão definidas; geração de código. Então é perceptível que se está lidando com uma estrutura heterogênea - pode conter vários subestruturas - que exigem operações específicas para cada uma. Outro possível problema é que todas vez que for criado um nova subestrutura com operações próprias, todas as outras subestruturas deverão incluir as novas operações, assim como a nova subestrutura deverá incluir as operações antigas (ver Figura 4).

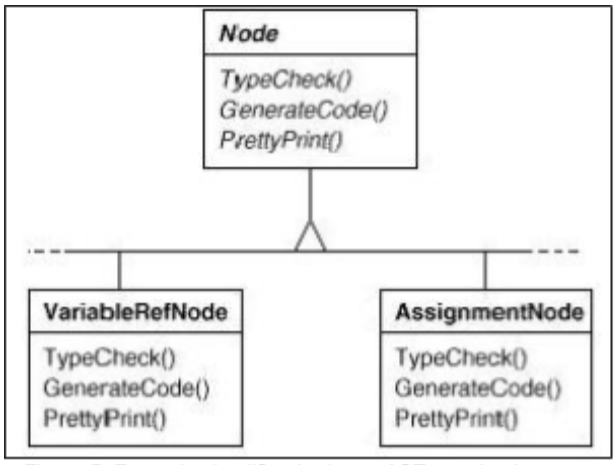


Figura 5: Exemplo simplificado de um AST que implementa todas operações de cada subestrutura.

Uma solução para esse problema é encapsular a operação em um objeto separado utilizando o padrão Visitor. Em programação orientada a objetos e engenharia de software, o *visitor pattern* é um padrão de projeto comportamental. Representa uma operação a ser realizada sobre elementos da estrutura de um objeto. O Visitor permite que se crie uma nova operação sem que se mude a classe dos elementos sobre as quais ela opera. É uma maneira de separar um algoritmo da estrutura de um objeto. Um resultado prático é a habilidade de adicionar novas funcionalidades a estruturas de um objeto pré-existente sem a necessidade de modificá-las.

A árvore sintática abstrata (AST) terá eventualmente muitos tipos diferentes de nós, todos derivados de *AbstractNode*. Se o comportamento dos nós são diferente, então isso é feito através da *override* de métodos na definição de classe dos nós. Também tem-se que percorrer um AST para diversos propósitos diferentes. Pode-se imprimir o AST, realizar-se análises semânticas ou gerar código. Cada um destes poderia ser realizado refinando a noção de percurso de árvores em extensões de alguma superclasse comum.

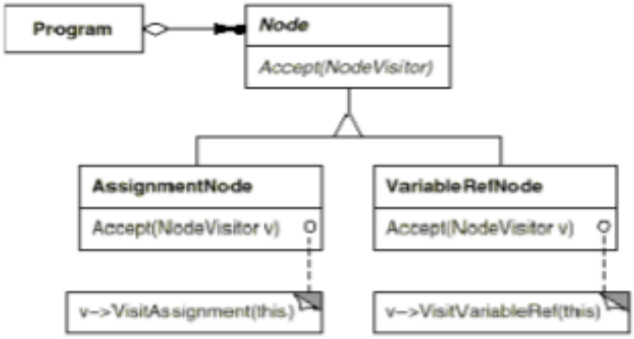
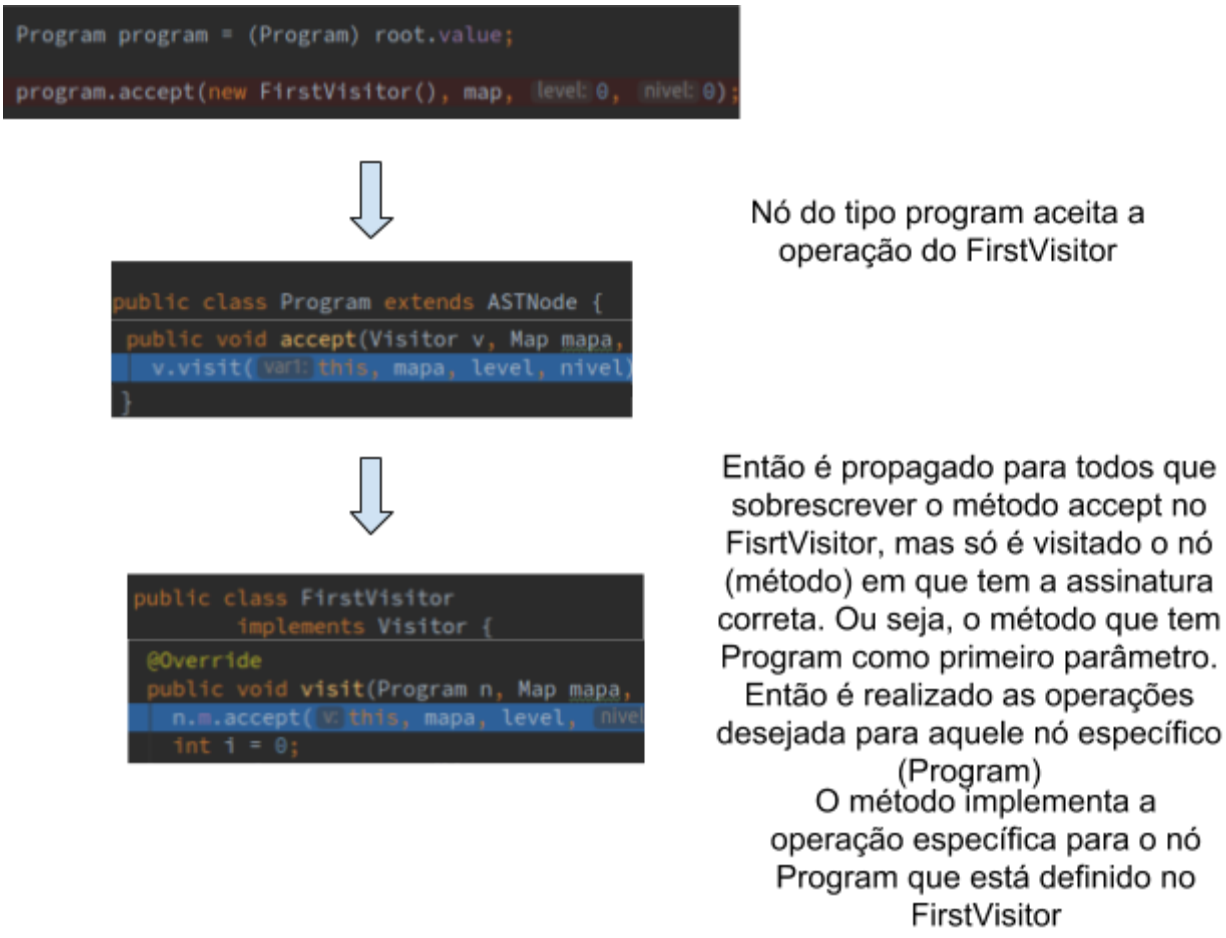


Figura 6: Apresenta algumas estruturas da AST e como estas implementam o padrão Visitor

3.3.2.1 Exemplo real Visitor

O padrão Visitor está tendo um detalhamento a mais, dado que ele tem uma grande relevância para o desenvolvimento do projeto de um complicador e inicialmente é um pouco complicado de entender.



No projeto foram implementados 3 Visitor que em seu conjunto fazem operações análise semântica e construção de código assembly (linguagem-alvo).

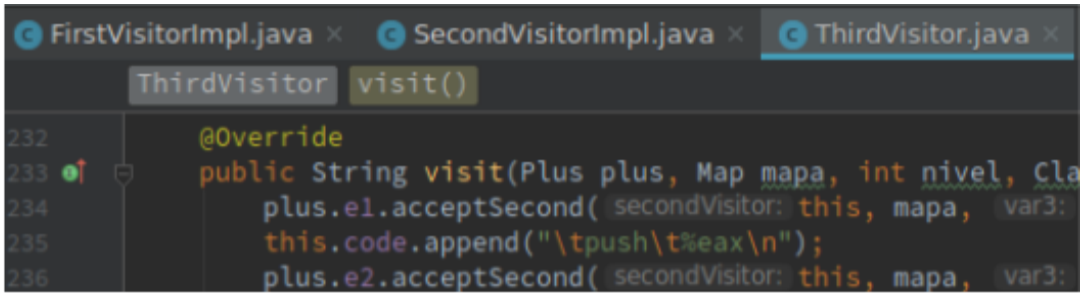


Figura 7: Três Visitor que foram implementados no projeto

4 Construção das Tabelas e Verificação de Tipos

Tabela de símbolos é construída durante a fase de análise semântica. É utilizada para armazenar todas as entidades estruturadas em um único lugar. Auxilia na verificação se foi declarada uma variável, na verificação de tipos, verificação de métodos, verificação de classes, checagem de escopo etc. A construção da tabela é com auxílio do Visitor, agora com operações de preenchimento da tabela.

Caso existe um erro semântico, o compilador tratará esse erro lançando uma exceção. Por exemplo, caso deseja executar a seguinte operação:

```
Class Teste {
        public static void main (String [] a)rgs{
        System.out.print(false);
        }
}
```

Figura 7: Exemplo de código com erro semântico

Então é apresentada a seguinte mensagem no log:

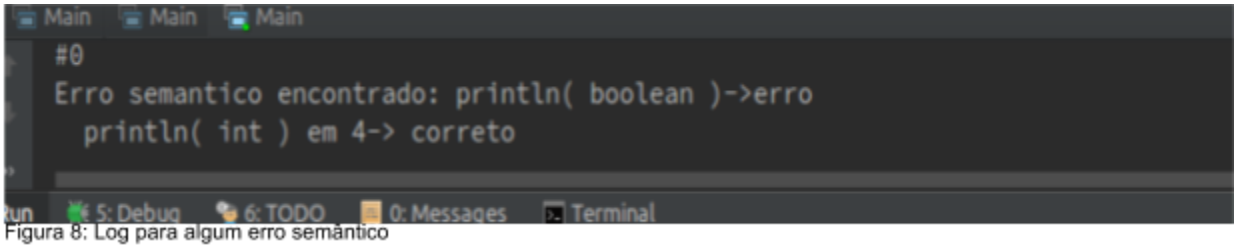


Figura 8: Log para algum erro semântico

5 Tradução em Assembly

Nesse passo, o Visitor ainda tem um papel fundamental, pois é esse que é responsável por visitar a árvore criada pelo parser e lidar com os procedimentos que gera o código assembly para cada tipo de nó. O responsável por visitar cada nó da árvore e fazer a geração do código assembly é o ThirdVisitor.java.

```
@Override
public String visit(This aThis, Map mapa, int nivel, Classe classe, Metodos metodo) {
    this.code.append("\tmov\t" + (this.methodDecl.fl.size() + 2) * 4 + "(%ebp),%eax\n");
    return classe.identificador;
}
```

Figura 11: ThirdVisitor visita o nó do tipo 'This' e gerando sua respectiva tradução em assembly.

6 Executando o Compilador

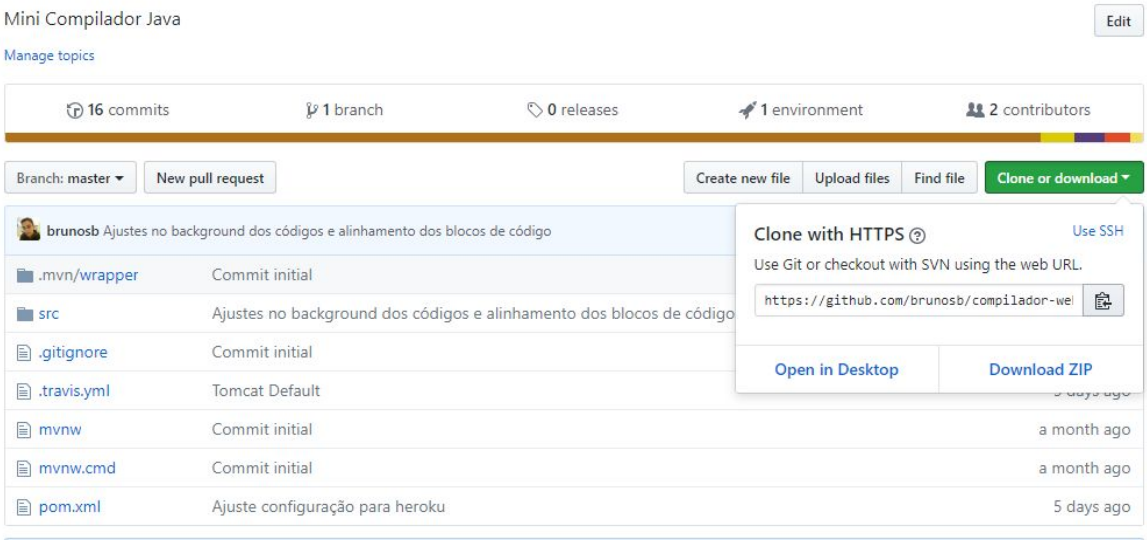
6.1 Detalhes

O projeto foi implementado usando JavaEE com o framework principal Spring Boot no backend e Thymeleaf no frontend. O gerenciamento das bibliotecas foi feito pelo *Maven*.

Colocamos o projeto em um repositório GitHub, no qual pode ser acessado em https://github.com/brunosb/compilador-web, juntamente em parceria usamos o Travis para integração contínua e finalizando com hospedagem no Heroku na url https://compilador-web.herokuapp.com. Os estágios de desenvolvimento seguiram essa linha, a cada commit realizado o Travis verificava o código e se não houvesse nenhum erro o Heroku automaticamente subia a nova versão.

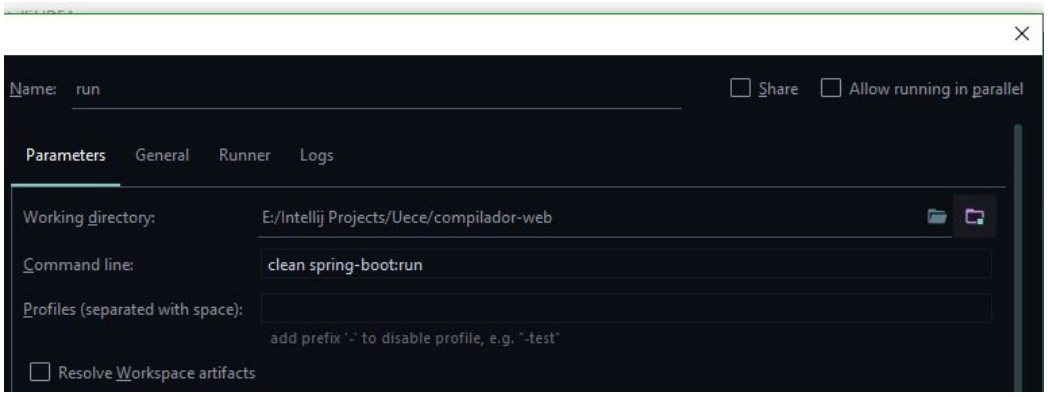
6.2 Clone e importação do projeto

Para executar o projeto localmente, é necessário ter o JDK 8 e o Maven instalados na máquina, após isso faça o clone ou o download no repositório GitHub citado anteriormente. Com uma IDE de sua preferência importe o projeto como um "projeto maven existente".



6.3 Execução Web e Local

Para executar localmente o projeto basta utilizar o maven com o goal "clean spring-boot:run" e o compilador será executado no endereço *localhost:8080*.



A versão web se encontra na url https://compilador-web.herokuapp.com, se não houver muitos acessos, a primeira vez demora um pouco para carregar a página, isso é devido a conta free do heroku que “dorme” a aplicação se não houver acessos em 30 min.

Na página carregada podemos ver um botão “Upload .java”, ao clicar nele selecione um dos arquivos MiniJava enviados por email. Quando for selecionado, automaticamente é feito a análise léxica, sintática, semântica, é construído uma AST, é construído uma tabela de símbolos, são feitos procedimentos de checagem de tipos e é gerado o código assembly.

Figura: Corpo da página e o botão “Upload .java

Figura: Exemplo QuickSort.java compilado. Gerou os Tokens, Tabela de Símbolos e o Assembly

6.4 Objetivos não-realizados

Nesse projeto não foi realizado a implementação do laço For. Para substituí-lo, tem a estrutura do while.

7 Bibliografia

1- http://courses.cs.washington.edu/courses/cse401/10au/project/index.html
2- Modern Compiler Implementation in Java, Second Edition by Andrew W. Appel and Jens Palsberg Cambridge University Press © 2002
3- Engineering a Compiler, Cooper & Torczon,