**Implementação de um Compilador (bytecodes)**

**Fernanda Maria de Souza1, Matias Giuliano Gutierrez Benitez1,**

Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC)

fernandamsouza833@gmail.com, [matiguti@gmail.com](mailto:matiguti@gmail.com),

**1. Introdução teórica**

**1.1 Especificações FLEX (Analisador Léxico)**

O Flex serve para gerar automaticamente programas (em C) fazendo a leitura de uma entrada de modo a varrer um texto e/ou programa a fim de obter uma sequência de unidades léxicas (tokens).

Seu modo de funcionamento é dado pela leitura de arquivos de entrada especificados, ou a entrada padrão caso nenhum arquivo for comparado, obtendo assim uma descrição do scanner a ser gerado. Este arquivo de entrada é o que chamamos arquivo de definição ou arquivo de descrição.

A descrição é realizada na forma de pares de expressões regulares e código C. Tais pares são denominados regras. As regras definem simultaneamente quais padrões devem ser procurados e quais as ações que devem ser executadas quando da identificação deste padrão, ou seja, para cada padrão desejado pode ser associado um conjunto de ações escritas sob a forma de código C.

Flex gera como saída um arquivo fonte em linguagem C, cujo nome é lex.yy.c, no qual é definida a função yylex() e as variáveis globais yytext e yyleng. A função yylex() é na verdade o analisador léxico gerado pelo arquivo de definição atraveś do flex. A variável global yytext contêm o texto do padrão reconhecido (uma string) no momento enquanto yyleng contêm o número de caracteres de tal string, podendo ambas serem usadas em trechos de código C que definem as ações associadas a cada padrão.

**1.2 Especificações Bison (Analisador Sintático)**

O objetivo da análise sintática é determinar a estrutura sintática do código-fonte que está sendo compilado. Para isso, a análise sintática utiliza o fluxo de tokens produzido pela análise léxica. A tarefa do analisador sintático é muito mais simples de realizar partindo dos tokens da entrada, ao invés dos caracteres isolados. Uma especificação bison descreve uma gramática livre do contexto que pode ser usada para gerar um parser, sendo que a mesma possui elementos membros de 4 classes:

* Tokens, que é o conjunto de símbolos terminais.
* Elementos sintáticos, que são símbolos não terminais.
* Regras de produção, que definem símbolos não terminais em termos de sequência de terminais e não terminais.
* Uma regra start, que reduz todos os elementos da gramática para uma regra.

Alguns conceitos para o Bison:

* A cada regra está associado um símbolo não terminal (lado esquerdo).
* As definições (lado direito) consistem de zero ou mais símbolos terminais (tokens ou caracteres literais) ou não terminais.
* Tokens são símbolos terminais reconhecidos pelo analisador léxico, e só aparecem no lado
* direito das regras.
* A cada regra pode ser associada uma ação em C. Estas ações ficam entre chaves ({}).
* Os nomes de símbolos podem ter qualquer tamanho, consistindo de letras, ponto, sublinhado e números (exceto na primeira posição).
* Caracteres maiúsculos e minúsculos são distintos.
* Os nomes de não terminais são usualmente especificados em minúsculos.
* E tokens, em maiúsculos.
* Definição de tokens, como por exemplo: %token T1.
* Definição de regras auxiliares para solução de ambiguidades. Exemplo: %left MAIS MENOS.
* Regras de produção: cada símbolo tem associado a ele uma pseudo variável e ações podem ser executadas sobres estas variáveis.

**2. Implementação**

**2.1 Implementação Flex (Reconhecimento léxico)**

No arquivo calc.lex realiza-se o reconhecimento léxico da gramática, ou seja, todos os tokens são reconhecidos bem como erros léxicos são apontados. Em nossa implementação fizemos uma análise para descobrir em qual linha e coluna o erro se encontra de forma a cada vez que um token é reconhecido, soma-se mais um na variável coluna, enquanto que em cada \n (nova linha) é somado mais um na variável linha. Portanto, temos a localização exata caso ocorra um erro léxico na execução do arquivo ligado ao compilador.

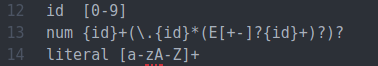


Imagem 1: definição das classes léxicas id, num e literal.

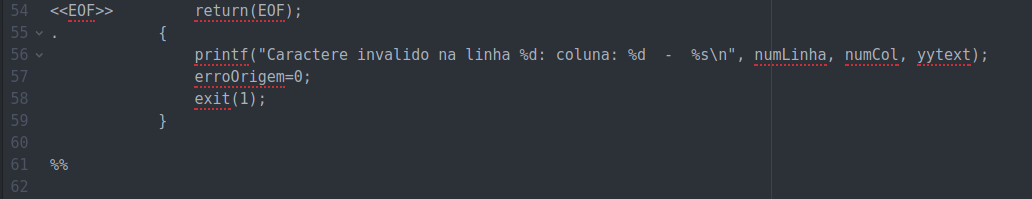


Imagem 2: Caso nenhum token seja reconhecido, ao fim do reconhecimento é escrito onde o erro ocorreu.

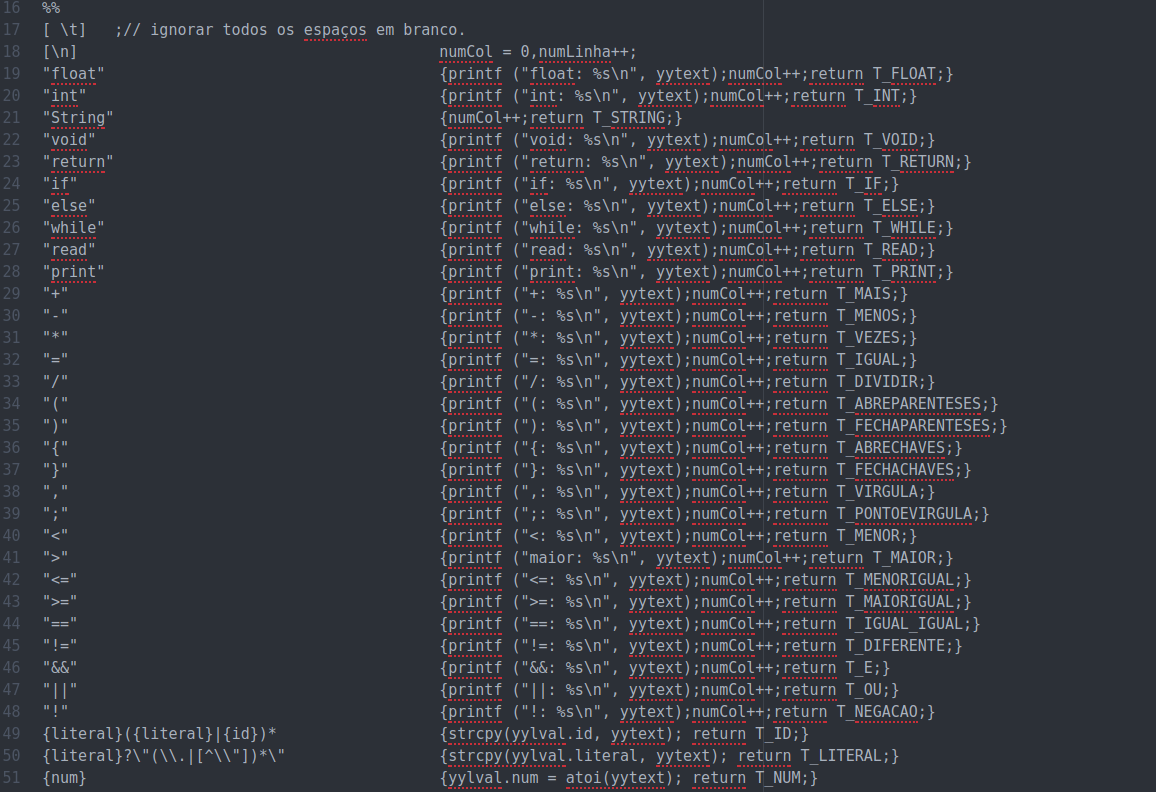


Imagem 3: Reconhecimento dos tokens.

Um destaque para a compilação se trata do armazenamento dos token T\_ID, T\_LITERAL e T\_NUM na Struct Atributo, sendo uma união em forma de struct pelo YYSTYPE.

**2.2 Implementação Bison (Reconhecimento sintático e integração com Jasmin) - Resumo**

Após passar todos os parâmetros externos para o nosso arquivo calc.y (que será responsável pelo Bison), definir nossa estrutura Atributo contendo os campos id, num e literal, além de outras especificações futuras, declaramos todos os tokens reconhecidos no flex.

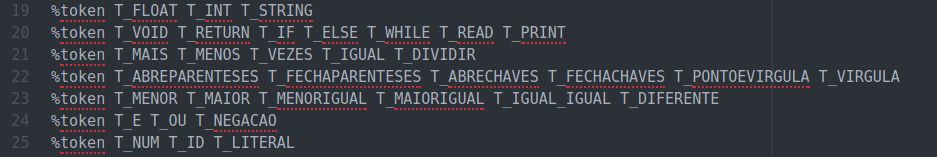


Imagem 4: Declaração de Tokens no Bison.

Após declaração dos tokens, o mapeamento da gramática inicia com o Start que é definido para gerar todas as instruções do Jasmin equivalentes ao código associado, conforme representado pelas imagens abaixo:

.

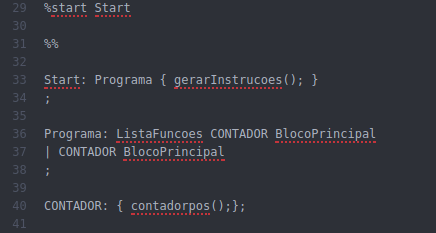


Imagem 5: Começo da gramática

Enquanto isso, em nosso implementação mapeamos o cabeçalho da linguagem Jasmin na função gerarInstruções, função que por sua vez irá escrever todas as instruções relacionadas ao compilador.

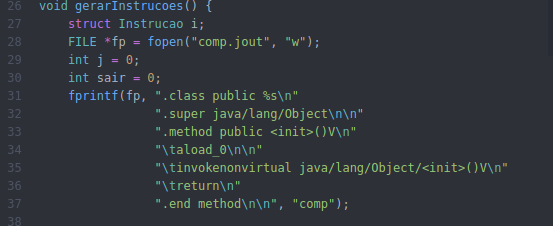


Imagem 6: Declaração do cabeçalho do Jasmin.

Voltando para a implementação do Bison realizamos a definição da gramática e dos atributos equivalentes a ela. Como a gramática é extensa decidimos não abordar parte por parte da mesma no relatório, mas sim a documentação das partes que consideramos mais importantes.

Uma maneira de ilustrar o desenvolvimento do trabalho é pelos métodos utilizados. Para a expressaoComparações chamamos um atributo em forma de função denominado CMD\_comparação que tem como parâmetro um número apenas para identificação posterior do sinal utilizado, e também implementamos a chamada do comando goto.

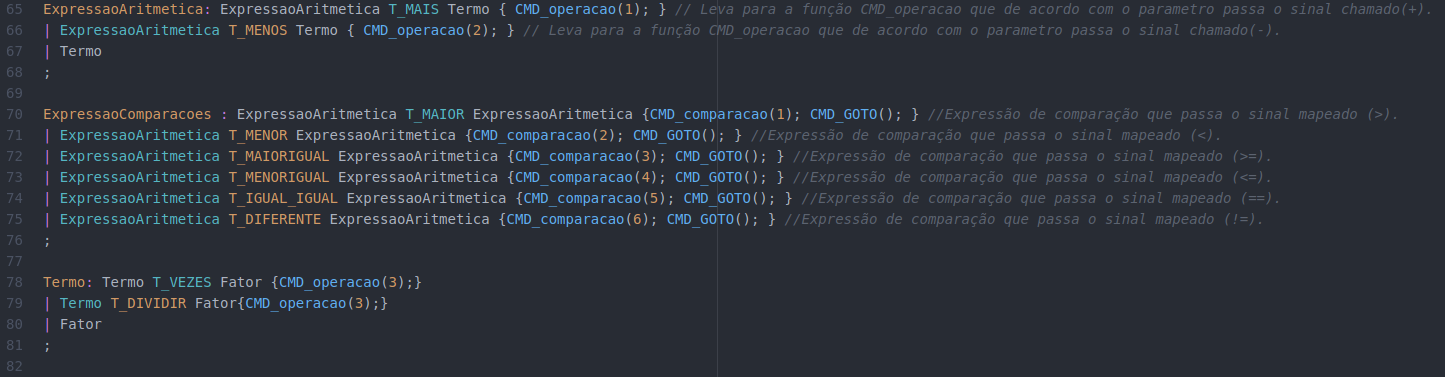


Imagem 7: Parte do Bison envolvendo a regra ExpressaoComparações.

De acordo com o parâmetro passado para dentro da função CMD\_comparação realizamos uma estrutura de condição para identificar o operador correto associado. Caso, por exemplo, o operador seja o “>” temos que ir para a função e no case específico chamar outra função, essa por sua vez se chama Instrução, que irá conter parâmetros como a label informada, a instrução, o parâmetro e uma string. A imagem abaixo informa como mapeamos a função CMD\_comparação:

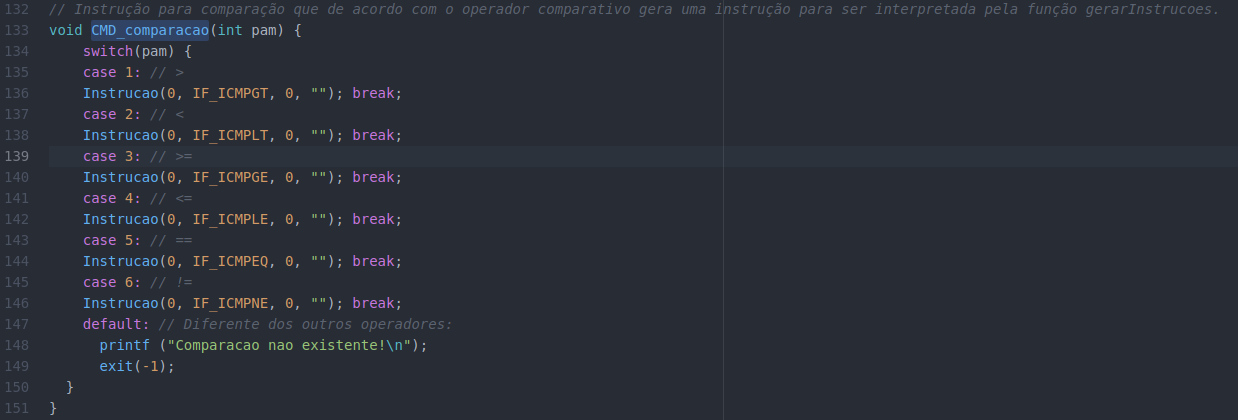


Imagem 8: Mapeamento da função CMD\_comparação presente no arquivo jasmin.c

Após, como comentado no texto acima, vamos para a função Instrução que contém as informações da instrução. Recebido os parâmetros fazemos a transferência dos mesmos para a Struct Instrução e após isso inserimos na Lista de Instruções para manipulação dos dados.

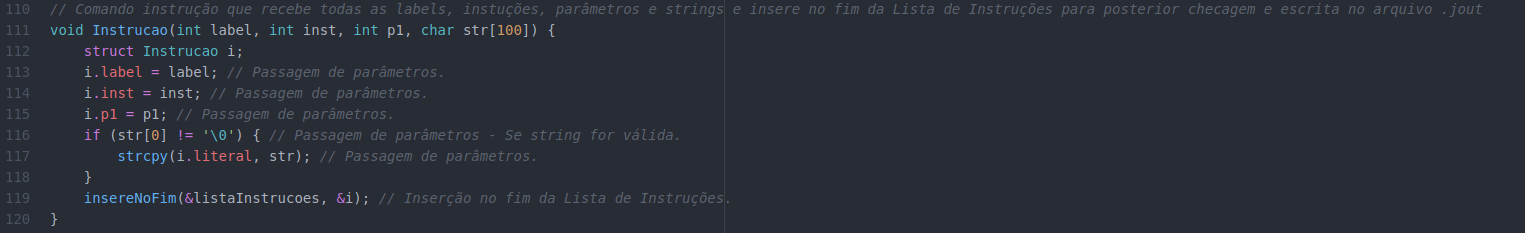


Imagem 9: Função Instrução que guarda os parâmetros recebidos.

Com o caminho feito e as informações armazenadas na Struct Instruções, vamos para a função final que se trata da escrita dessas informações no arquivo. Após percorrido todo o caminho pela gramática e sua sintaxe, voltamos para a função declarada na Imagem 5 onde chamamos a função gerarInstruções que irá escrever no arquivo “comp.jout” de acordo com a instrução o seu comando traduzido para a linguagem do Jasmin. Seguindo a linha de pensamento desenvolvida para essa lógica, para o operador “>” na função final para a escrita do mesmo ele seria chamado por uma estrutura condicional:

case IF\_ICMPGT:

fprintf(fp, "\tif\_icmpgt l%u\n", i.label); break;

Entrando na condição pelo nome da instrução, é escrito no arquivo a tradução e sua devida label, informação que foi armazenada na Struct Instrução informada anteriormente.

**2.3 Bytecodes e Jasmin**

*Bytecode* é um formato de código intermediário entre o código fonte, o texto que o programador consegue manipular, e o código de máquina, que o computador consegue executar. Os bytecodes facilitam a manipulação e geração dinâmica de modo a exigir o uso de uma JVM (Java Virtual Machine).

O Jasmim é um assembler que recebe uma descrição textual e as converte para arquivos binários no formato class file.

**3. Resultados e erros**

O compilador em questão foi desenvolvido com base em estudos passados pelo Professor Ricardo Martins e também por pesquisas profundas em maneiras de representar, manipular, armazenar e traduzir as informações que tínhamos em bytecodes Java para tradução pelo Jasmin. A ideia de manipulação de Listas e Structs foi baseada no projeto do Github enviado pelo professor, que consta nas referências deste trabalho, bem como diversos trabalhos encontrados que serviram como meio de estudo e entendimento para representar um compilador.

Com as informações estudadas, partimos para a estratégia de tentar passo por passo traduzir as informações para o Jasmin. O código apresentado está incompleto para toda a gramática, porém reconhece grande parte do exemplo enviado pelo professor. Tivemos dificuldade em gerar um executável para o Jasmin pois temos erros nas comparações e em labels. O código testado para compilação está presente na imagem a seguir:

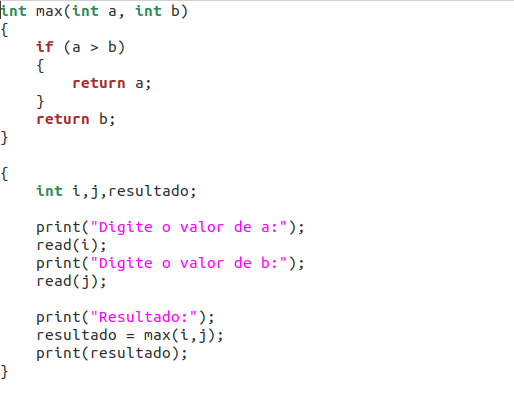


Imagem 10: Código em C para ser traduzido a bytecodes.

Conforme as imagens a seguir, temos o resultado do código .jout obtido por nós para o exemplo acima: 

Imagem 11: Resultado obtido em compilação.

E após, o código que deveríamos ter gerado:



Imagem 12: Resultado esperado.

Obs.: Cabeçalho não consta nas imagens, porém está incluído corretamente.

Portanto, conforme vemos nas imagens, obtivemos um bom resultado na geração das instruções para o Jasmin, porém não conseguimos implementar corretamente as comparações e geração de labels. Quando apresentamos o arquivo .jout para o Jasmin 2.4 o mesmo faz o diagnóstico do código para o erro apontado conforme a imagem:



Imagem 13: Erro apresentado ao passar o arquivo para a ferramenta Jasmin.

**4. Testes Gerais**

**4.1 Primeiro Teste**

Realizamos alguns testes para ilustrar a funcionalidade do programa e seu estado de execução atual:

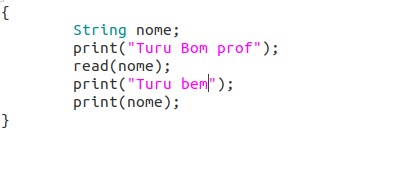


Imagem 14: Teste presente no arquivo exemplo.j

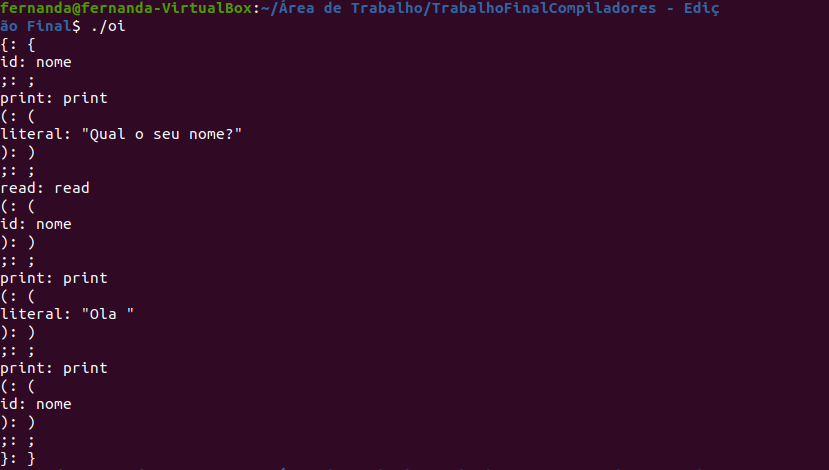


Imagem 15: Comprovação do funcionamento do Flex e Bison para esses tokens.



Imagem 16: Arquivo comp.jout gerado para o exemplo.j

Com esse formato de exemplo conseguimos gerar o arquivo .class, porém na execução do mesmo deparamos com outro problema.

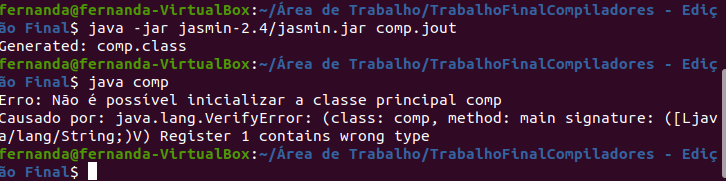


Imagem 17: Problema na execução no arquivo comp.class

**4.2 Segundo Teste**

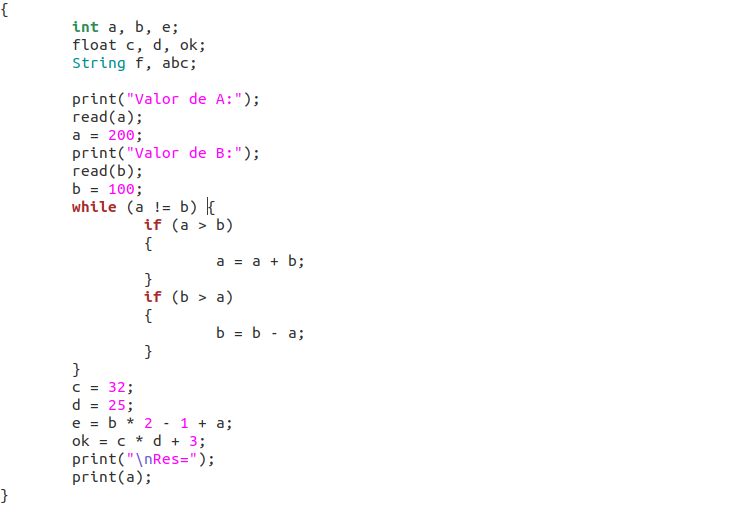


Imagem 18: Teste presente no arquivo exemplo.j

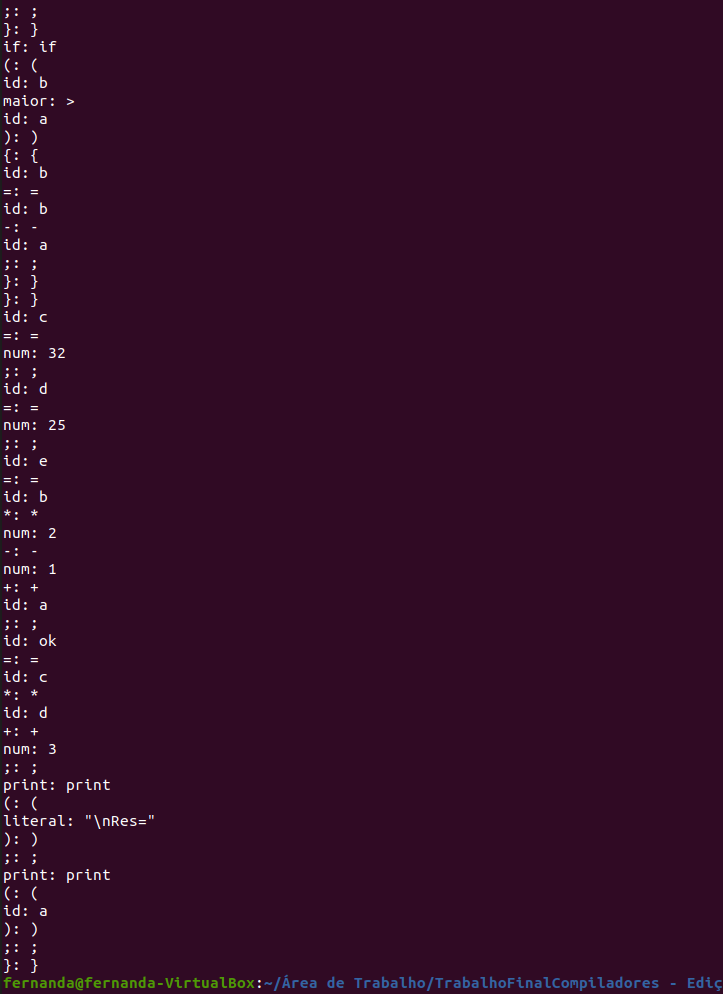
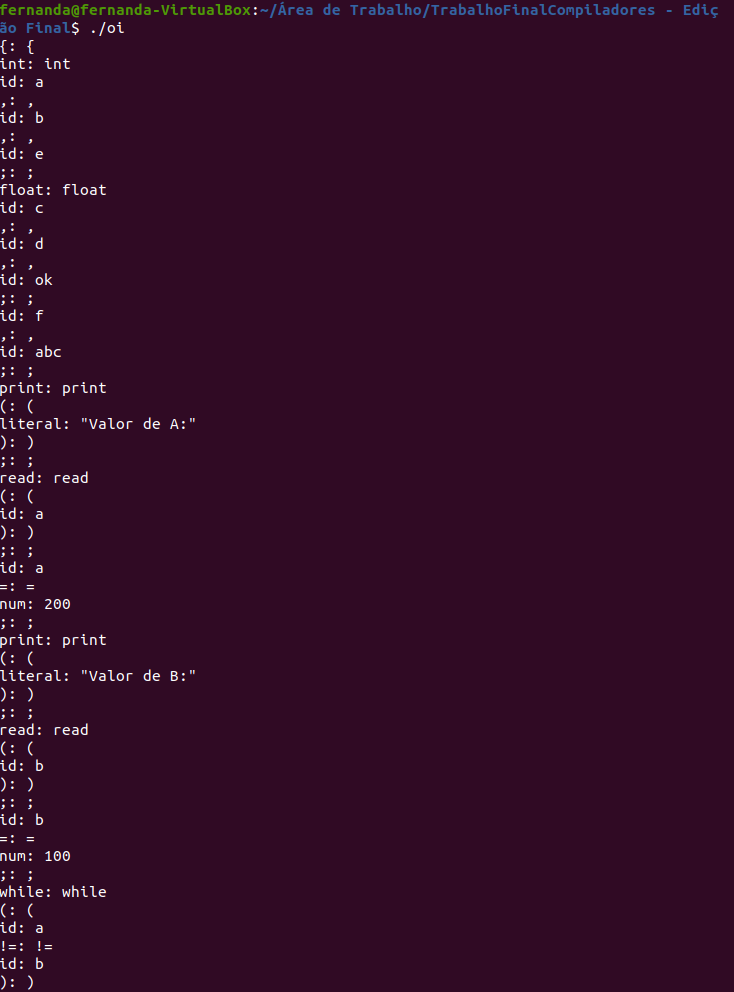


Imagem 19 e 20: Comprovação do funcionamento do Flex e Bison para o exemplo.



Imagem 21: Primeira parte do segundo teste: arquivo comp.jout

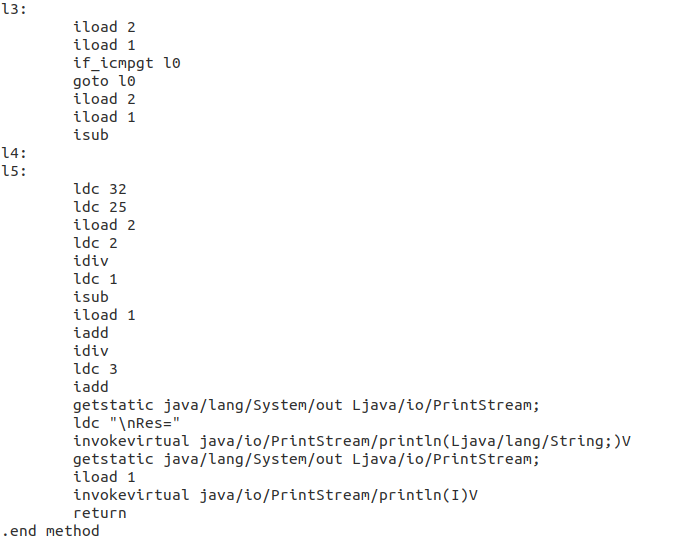


Imagem 22: Segunda parte do segundo teste: arquivo comp.jout

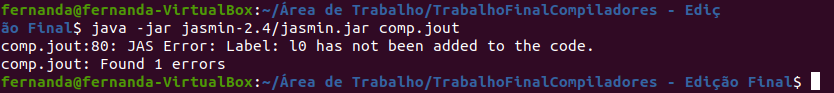


Imagem 23: Erro encontrado na chamada do arquivo comp.jout para o Jasmin.

**5. Mapeamento Estrutural do Trabalho**

Dividimos o Compilador em 7 arquivos diferentes, sendo eles:

* calc.lex - arquivo responsável por identificar o léxico do compilador (seus tokens, classes léxicas e possível posição de um erro léxico - indicando a linha e coluna do erro).
* calc.y - arquivo responsável por identificar a parte sintática e semântica do código, realizando também a tradução por intermédio do jasmin.c e headers para a linguagem que posteriormente seria lida pelo Jasmin.
* eda.c - arquivo contendo a implementação de Listas e manipulação das mesmas (o arquivo já estava pronto e foi desenvolvido na disciplina de Estrutura de Dados)
* eda.h - headers e protótipos das funções e structs vistas no arquivo eda.c
* main.c - arquivo principal que apenas lê o arquivo e passa para as fases seguintes do compilador.
* jasmin.c - arquivo contendo todas as funções para geração de instruções traduzidas para o Jasmin bem como funções auxiliares utilizadas para diagnosticar erros semânticos. Neste arquivo também é realizada a inserção na tabela de símbolos e é feita a tranmissão de parâmetros para as structs declaradas no jasmin.h, que posteriormente serão lidas novamente no mesmo arquivo para escrever no .jout as instruções traduzidas para o Jasmin.
* jasmin.h - arquivo contendo as headers do jasmin.c e definições de protótipos de funções e structs, informando todos os campos utilizados.

Além dos arquivos divididos por nós, existem os arquivos criados pelo flex e bison:

* calc.tab.c - arquivo que contém a definição das tabelas para a análise sintática e a rotina de reconhecimento, cujo nome é yyparse() - gerado pelo bison.
* calc.tab.h - header do calc.tab.c - gerado pelo bison.
* lex.yy.c - define a rotina yylex() que usa a especificações para reconhecer tokens - gerado pelo bison.

**6. Características do código desenvolvido**

**6.1 Ambiente**

Todo o trabalho foi desenvolvido em uma máquina virtual Linux com o emulador Oracle. Também, no começo utilizamos uma máquina virtual feita pelo professor Ricardo Martins mas decidimos, por questão de comodidade e conhecimento, escolher o primeiro ambiente como o oficial para o desenvolvimento do trabalho.

**6.2 Linguagem**

A linguagem utilizada para desenvolvimento do trabalho foi C.

**6.3 Comandos para compilação**

Para compilar o “Compilador” devemos executar os seguintes comandos no terminal:

bison -d calc.y

flex calc.lex

gcc main.c calc.tab.c eda.c jasmin.c -o “nome do executável” -lm

./”nome do executável”

Definimos como base de leitura arquivos nomeados como “exemplo.j” e saídas nomeadas “comp.jout”. Após:

java -jar jasmin-2.4/jasmin.jar comp.jout

java comp

Porém, é nesta etapa em que nosso compilador não executa o que deveria.

Vale destacar que adicionamos prints quando o código é executado para informar que a análise léxica está reconhecendo devidamente os tokens, bem como a análise sintática está sendo feita corretamente pois nenhum erro sintático é informado no Terminal.

**7. Conclusão**

Portanto, concluímos que a implementação de um compilador (bytecodes) para posterior análise pelo Jasmin é uma tarefa baseada em etapas sendo ordenadas basicamente em:

1- Análise Léxica (flex e geração de tokens).

2- Análise Sintática (bison e organização da gramática).

3- Análise Semântica (verificação de tipos, do fluxo de controle e da unicidade na declaração de variáveis e também tratamento de erros).

4- Tradução para o Jasmin (por meio de atribuições feitas na gramática com funções que transformam o código C para a linguagem lida pelo Jasmin).

5- Leitura do Jasmin e geração do arquivo class (parte final onde operações podem ser feitas e o programa devidamente testado).

Ainda, com os resultados e erros obtidos ao fim do trabalho, acreditamos que o compilador desenvolvido possa suprir as necessidades básicas da gramática que não envolvam alta complexidade. Porém, como não conseguimos finalizar e aplicar o Jasmin gerando o arquivo .class corretamente, não conseguimos avaliar a real funcionalidade do compilador e se o mesmo está efetuando as operações de tradução como deveria.

Logo, encerramos o trabalho com os analisadores léxico e sintático funcionando corretamente; com a análise semântica e tradução para o Jasmin implementados de forma básica; e por fim a leitura do Jasmin e geração do arquivo class que não conseguimos implementar.

**8. Referências**

Aquino, Samantha. Compiladores. Github. 2018. Disponível em: <https://github.com/SamanthaAquino/Compiladores>. Acesso em 17/11/19

Pinali, Osvaldo. Bytecode: Explicando e Revelando. DevMedia. 2009. Disponível em: <https://github.com/SamanthaAquino/Compiladores>. Acesso em 15/11/19

CompilersPhase3. 2015. Disponível em: <https://github.com/romaad/compilersPhase3>. Acesso em 12/11/19

Vicente, Rodrigo. Compilador-Morsa. Github. 2017. Disponível em: <https://github.com/Rodrigovicente/Compilador-Morsa>. Acesso em 19/11/19

Documento presente no Google Drive. Guião Compiladores. Disponível em: <https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxjb21waWxhZG9yZXNmdWNhcGl8Z3g6Nzg2ODBmZjUyNWVkMGFjNg>. Acesso em: 15/11/19

Moraes, Emanuel. Elementary. Disponível em: <https://github.com/emanuelmoraes-dev/elementary>. Acesso em 22/11/19

Maros, Alexandre. Arxs Compiler. Github. Disponível em: <https://github.com/alepmaros/Arxs_Compiler>. Acesso em 15/11/19