Universidade do Estado de Santa Catarina – Campus CCT

Disciplina: Compiladores I (Com001)

Docente: Dr. Ricardo Ferreira Martins

Alunos: Adilson Krischanski

Karla Alexsandra de Souza Joriatti

TRABALHO FINAL

ANÁLISE SINTÁTICA

Joinville, 29 de julho de 2022

1.0 Implementação

Com base na proposta inicial do trabalho, foi desenvolvido um compilador de uma linguagem teste com auxílio dos Softwares FLEX, versão 2.6.4, e BISON, versão 3.5.1. Para a codificação também foi necessário o uso de uma IDE (integrated development environment), o qual optamos por utilizar o Visual Studio Code na versão 1.62.3 devido a algumas facilidades por ele oferecida como extensões .lex (arquivo tipo flex) e .y (arquivo tipo bison) bem como liveshare, extensão que permite desenvolvimento conjunto em tempo real, tornando o desenvolvimento muito mais eficiente. Para o Desenvolvimento quando possível realizamos encontros presenciais na universidade, entretanto a maior parte foi desenvolvida de forma online utilizando o Discord para a comunicação.

Inicialmente foi preciso fazer as alterações no arquivo .lex entregue na primeira etapa do trabalho, para isso incluímos as regras de retorno referente a cada *token*. Continuando, avançamos para o arquivo .y do bison, onde de início apenas organizamos as informações de cabeçalho de acordo com os exemplos passados pelo professor. Na criação das regras sintáticas, feitas de acordo com a gramática passada pelo professor, foi necessário a aplicação de algumas fatorações a esquerda devido a conflitos de *shift/reduce i*dentificados por meio da opção –report=states do bison. Tratados os warnings, passamos portanto para a fase de geração de código.

Por fim, para fazer a geração de código em bytecode para JVM, foi criado um arquivo output.j, que guardaria o que foi gerado pelas regras sintáticas, e um arquivo de código teste para ser traduzido em bytecode. Aos poucos criamos funções para fazer estas traduções, começando com declarações de variável, atribuições, expressões simples e por fim comandos mais complexos dos quais fosse necessário *jumps* e criação de *labels*.

1.1 Tabela de Símbolos

A tabela de símbolos é um dos elementos mais importantes para o desenvolvimento de um compilador, utilizada para armazenar tokens e variáveis bem como seus tipos e posições. Durante a primeira parte do trabalho ela armazenava todo e qualquer token lido pelo analisador léxico durante a leitura e validação dos tokens. Já na segunda parte do trabalho implementamos uma tabela armazenando as variáveis, seus tipos e sua numeração durante a geração do código intermediário.

Durante a alteração do arquivo de análise léxica acabamos, por equívoco, deletando a tabela de símbolos pois o professor disse que para aquele momento não era importante, todavia não havia a necessidade da tabela para verificar e o arquivo estava de acordo com as regras sintáticas, mas a tabela volta a ser importante para a geração de código pois precisamos saber a quais variáveis vamos associar quais valores, então a tabela de símbolos foi reimplementada de forma simples porem dessa vez dentro do arquivo C de cabeçalho o qual chamamos de header.h ao invés de dentro do arquivo .lex como era inicialmente.

2.0 Exemplo de Código

A leitura do exemplo passado como parâmetro de teste foi feita por meio de leitura de arquivos em c. Uma primeira leitura nos gerou os seguintes erros, a partir do código que seguia a gramática:

- Letras maiúsculas
- “ - aspas
- \ - contrabarra
- / - barra
-

O código compreendia possíveis palavras reservadas apenas como id’s. Decidimos por adicionar somente a palavra reservada “print” a gramática pois outras palavras que deveriam ser tratadas como palavras reservadas já possuíam equivalentes na linguagem. Como por exemplo as palavras “bool” e “int” as quais já compreendiam os tipos “boolean” e “int-lit”.
Quanto aos tratamentos de erros: as chaves foram adicionadas com novas regras gramaticais; as letras maiúsculas também foram adicionadas na compreensão de id's; o underline foi adicionado a regra “outros”; a barra foi gerada como comentário sendo ignorado tudo que viria após a repetição de duas barras; e a contrabarra e as aspas duplas foram tratadas no reconhecimento de um print, o qual irá apenas ignorar o que está entre duas aspas duplas como token.

Funções print e read não estavam definidas então definimos

print: PRINT_WORD LEFT_BRACKET ASPAS idd ASPAS RIGHT_BRACKET

read: READ_WORD LEFT_BRACKET IDENTIFIER RIGHT_BRACKET

3.0 Dificuldades Encontradas

Como pode se ver trata-se de um trabalho longo e mesmo tendo alguns exemplos é tivemos bastante dificuldade para implementar a parte de geração do bytecode para a JVM que acabou não sendo possível concluir. Tivemos bastante dificuldade com para a visualizar o funcionamento da pilha para geração do output.j onde de inicio queriamos que acontece de uma forma e quando geravamos o arquivo ele ficava invertido. Devido a alta carga de atividades de ocasianoda pelo perildo de conclusão do semestre e com tempo escasso, abamos cometendo alguns erros “Bobos” de implementação da própria linguagem C que acabaram tomando bastante ou não foram possíveis de ser solucionado até a entrega. Como alunos recomendamos que um trabalho nessas proporções seja proposto com um prazo de mais longo e/ou em perildos menos caóticos como pelo meio do semestre, podendo ainda ser dividido em mais etapas como apenas reconhecimento das regras e geração de bytecode para a JVM, para que possa haver um resultado satisfatório.

Referências

Levine, John R. flex & bison. Simon St.Laurent 2009.

Martins, Ricardo Ferreira, Conteudo das aulas. Disponivel em : <https://ricardofm.me/index.php/pt/com0002/conteudo-das-aulas-com0002>

acessado em 28 de julho de 2019

Martins, Ricardo Ferreira, T2: Descição/Entrega . Disponivel em : <https://ricardofm.me/index.php/pt/avaliacoes-com0002/t1-com0002/t2-p1-com0002-2> acessado em 28 de julho de 2019