RELATÓRIO DO TRABALHO DE COMPILADORES
COMPILADOR RASCAL

ALUNO: VINÍCIUS DA COSTA REGATIERI - RA 104016

Introdução

Este trabalho tem como objetivo construir um compilador para uma versão modificada da linguagem Pascal, chamada de Rascal, a fim de exercitar os conhecimentos adquiridos durante o curso de Compiladores. Este relatório visa esclarecer decisões de projeto, as etapas cumpridas e não cumpridas do trabalho, dar uma visão geral dos módulos e da organização do analisador léxico e sintático e, por fim, mostrar o passo a passo de execução do projeto.

Decisões de Projeto e de Implementação

Linguagem escolhida

A linguagem escolhida para o desenvolvimento do compilador foi o Typescript. Essa escolha se deu principalmente pela ótima tipagem oferecida pela linguagem, por ter utilitários prontos, como listas e pela grande facilidade de depuração do código. Além disso, tenho uma grande afinidade com o Typescript por usá-lo diariamente.

Decisões de Projeto

Uma decisão importante nesse projeto foi não utilizar bibliotecas e ferramentas auxiliares nas etapas léxica e sintática, optando-se por desenvolver as duas etapas desde o princípio. Essa decisão foi tomada por conhecimentos prévios em como construir tais etapas, visto que uma das etapas do TCC do autor consistia no mesmo processo.

Etapas cumpridas e não cumpridas no desenvolvimento

Todas as etapas especificadas no trabalho foram cumpridas, sendo possível gerar um código MEPA dado um código de entrada para o compilador. Entretanto, há uma limitação na etapa de análise sintática: a árvore sintática de expressões é gerada incorretamente para expressões em que a precedência dos operadores não é explícita com o uso de parênteses. Por exemplo, a expressão **1*2*3** acaba gerando um erro sintático, mas a expressão **(1*2)*3** é analisada corretamente. Com isso, foi necessário alterar alguns casos de teste fornecidos pelo professor para que todos gerassem código MEPA.

Visão geral dos módulos

Há, na pasta raíz do projeto, uma pasta "src" que contém todos os arquivos fonte do compilador. Dentro dessa pasta, subpastas com nomes significativos dos estágios do compilador foram criadas para conter os respectivos códigos, sendo elas as pastas **lexer**, **parser**, **semantic** e **code-generator**. Além destas pastas, há uma pasta **error**, usada para manter os arquivos referentes ao gerenciamento de erros e uma pasta **common**, armazenando arquivos que podem ser usados em mais de uma etapa. Há também um arquivo chamado "**main.ts**", ponto de entrada do projeto, responsável por ler o código de entrada, executar cada etapa de compilação, escrever em um arquivo o código MEPA gerado e, ocasionalmente, mostrar os erros de análise para o usuário.

Analisador léxico

Como dito anteriormente, o analisador léxico, ou lexer, foi "feito à mão", sem o uso de ferramentas externas parecidas com o flex. Para isso, o lexer analisa letra a letra o arquivo de entrada recebido pelo programa, o interpretando como uma *string*. Cada token significativo encontrado é inserido em uma lista. Um token contém o tipo abstrato do token lido, seu lexema original e sua posição (linha, coluna de início e coluna de fim) no arquivo de entrada original. Ao final da análise, o *scanner* retorna a lista de tokens criada.

Analisador sintático

No mesmo sentido do analisador léxico, o analisador semântico também foi feito sem o auxílio de ferramentas de *parsing*. Para isso, o conceito de mapear uma regra da gramática em uma função, apresentado em sala de aula, foi aplicado. Com isso, cada uma das regras

da gramática foi abstraída em uma classe que mapeia perfeitamente a regra. Além disso, os literais do programa, inteiros e booleanos, juntamente com os identificadores também foram abstraídos em classes para ser possível trabalhar com tipos de dados mais precisos.

Durante sua execução, o *parser* cria uma árvore sintática abstrata (AST) em que os nós são objetos das classes que representam a gramática. Ao final, uma referência para a regra "Programa", principal regra da gramática do rascal, é retornado caso nenhum erro seja encontrado.

É interessante ressaltar que, diferentemente das ferramentas flex e bison, a análise léxica e semântica não ocorre simultaneamente, o que pode gerar um impacto na performance do compilador para códigos grandes. Essa decisão foi tomada conscientemente, já que julgou-se que os códigos de entrada do compilador construído para a disciplina não seriam muito grandes e construir as etapas separadamente seria mais simples e mais rápido.

Analisador semântico

O analisador semântico recebe como entrada a árvore sintática abstrata criada na etapa anterior e a percorre, verificando se todos os tipos estão sendo empregados dentro do programa. Para isso, faz o uso de uma estrutura auxiliar, a tabela sintática. Essa tabela foi abstraída no programa como uma lista de dicionários, em que cada posição desta lista representa um escopo diferente, havendo assim, no máximo duas posições na lista. No dicionário, as chaves são os nomes (lexemas) dos identificadores de funções, procedimentos, variáveis ou do nome do programa, enquanto os valores podem ser estruturas que descrevem funções ou procedimentos, contendo seu tipo de retorno (apenas para funções) e lista de tipos de parâmetros, ordenada; ou interfaces que apenas explicitam o tipo daquele nome, seja ele um inteiro ou booleano. Além disso, também são salvos, para cada entrada, dados necessários para a geração de código, sendo eles o nível léxico, o *offset* e, opcionalmente, se aquela entrada é um parâmetro de função/procedimento. Para toda essa abstração, deu-se o nome de Scope.

Geração de código MEPA

A geração de código MEPA ocorre ao mesmo tempo que a análise semântica. A classe "CodeGenerator" apenas abstrai em funções o preenchimento de uma lista de código

MEPA que, ao final, terá o código pronto em ordem. Essas funções são chamadas pelo analisador semântico quando necessárias, criando o código MEPA respectivo à AST de entrada, que é uma abstração do código rascal de entrada do programa inteiro. Caso a análise semântica não gere erros, o código gerado é escrito em um arquivo de nome "a.out".

A decisão de gerar o código MEPA durante a análise semântica também foi tomada conscientemente. Julgou-se que dividir estas etapas apenas faria com que a AST tivesse que ser percorrida novamente e ambas etapas conteriam arquivos com estrutura muito semelhante. Portanto, em um primeiro momento, toda a análise semântica foi desenvolvida e mais tarde aprimorada para conter a geração de código.

Passo a passo para execução

Como descrito anteriormente, o projeto foi desenvolvido na linguagem Typescript, sendo compilado para Javascript e executado no NodeJS. Portanto, para a execução, é necessário que estejam instalados na máquina, previamente, o próprio NodeJS e um de seus gerenciadores de pacotes, **yarn** ou **npm**.

Na pasta raíz do projeto, use **npm install** ou **yarn install** para que as dependências do projeto sejam instaladas. Em seguida, utilize **npm run build** ou **yarn build** para compilar o projeto, criando os arquivos Javascript necessários. Após isso, o compilador rascal já está preparado para a execução.

Para maior facilidade, há disponível um arquivo .sh auxiliar (rascalc) para a execução do compilador. No terminal, apenas utilize **./rascalc <nome do arquivo .ras>**.

Caso nenhum erro seja reportado, o código MEPA estará disponível na mesma pasta de execução, com o nome "**a.out**" e, caso contrário, erros serão mostrados no terminal.