UNEB
UNIVERSIDADE DO ESTADO DA BAHIA

SISTEMAS DE INFORMAÇÃO

FUNDAMENTOS DE COMPILADORES

COMPILADOR Cmm

Trabalho apresentado como parte das avaliações parciais da disciplina Fundamentos de Compiladores do curso de Sistemas de Informação do Campus I da UNEB.

CAIO VICTOR SAMPAIO MOTA SANTOS

2017.2

1. Introdução

A construção do Compilador Cmm (CCmm) começou com um planejamento da estrutura de código e funcionamento. Foi decidido que o mais natural seria que um suposto programador escreveria o código em um arquivo e utilizaria o compilador no mesmo, gerando um arquivo executável. Desse modo, ao executar o CCmm, é necessário informar qual é o arquivo a ser compilado juntamente com sua extensão, que pode ser .cmm. Caso o código esteja correto, o Compilador encerra a execução logo após uma mensagem afirmativa, mas caso esteja incorreto, uma mensagem correspondente ao primeiro erro encontrado é mostrada e logo em seguida a execução se encerra. O código relativo a essas funcionalidades foi embutido no Analisador Léxico por este ter sido o primeiro módulo a ser desenvolvido.

Pela natureza do projeto, o seu desenvolvimento foi separado nos respectivos módulos necessários, Léxico, Sintático, Tabela de Símbolos, Semântico e Gerador de Código. Estima-se que foram dedicadas 40 horas ao desenvolvimento do CCmm.

2. Análise Léxica

O módulo de análise léxica do CCmm seguiu as regras incluídas na Especificação da linguagem. Primeiramente foi feito um estudo dessas regras e alguns testes manuais (papel e caneta) para melhor entendê-las. Após esse estudo foram feitos alguns rascunhos de autômatos capazes de reconhecer as cadeias e sinais da linguagem. Os autômatos foram reunidos em um único autômato principal capaz de reconhecer todas elas.

Após a construção do autômato, a estrutura básica do Analisador Léxico foi construída. Essa estrutura consiste de uma série de Constantes com tamanhos de tabelas e comprimento de caracteres; uma tabela numerada de Sinais responsável por reconhecer Tokens desse tipo; uma tabela numerada de Palavras Reservadas responsável por reconhecer cadeias de caracteres que pertençam a essa categoria; uma tabela de categorias de Tokens; uma Struct Token, essencial para os demais módulos do CCmm; além de uma série de funções e variáveis de suporte que auxiliam tanto na implementação do autômato quanto para enviar mensagens de erro e lidar com o arquivo de código.

A principal função do Analisador Léxico é a *getToken()*, responsável por ler o arquivo com código em Cmm em sequência e retornar o próximo Token encontrado. A função funciona com uma variável que simula o estado do autômato, um grande comando *switch* e uma série de comandos *if*. Para os testes de funcionamento do Analisador Léxico foram criadas as funções *printToken(token)* e *analexDisplayTokens()* que trabalham juntas para imprimir em sequência a lista de Tokens encontrados.

A construção deste módulo correu sem problemas já que o método de construção utilizando um autômato e as regras da especificação tornaram todo o trabalho bastante sólido e claro.

3. Análise Sintática

Para a construção do módulo de análise sintática, primeiramente foi construído um mecanismo capaz de ler os Tokens em sequência para poderem ser verificados. Decidiu-se por usar as variáveis *currentToken* e *nextToken* que são atualizadas através de uma chamada da função *parseTokens()* para esse propósito. Após o estudo das regras gramaticais da especificação da linguagem, iniciou-se a implementação de funções capazes de reconhecer os tokens específicos e chamar recursivamente outras. Porém, foram sendo implementadas diversas funções desnecessárias de suporte que logo poluíram o código e tornaram sua leitura difícil, além de poderem comprometer muito a implementação de futuros módulos.

Após fazer uma refatoração, prezou-se por seguir o máximo possível o comportamento das regras gramaticais do Cmm, o que foi alcançado com êxito. Durante esse trabalho alguns ajustes tiveram que ser feitos na especificação, mas graças a estrutura já melhorada do código, as correções necessárias não se tornaram problemáticas. Esses ajustes na verdade foram importantes para tornar o fluxo de chamadas de funções um pouco melhor, embora ainda existam algumas funções que ficaram grandes devido a similaridade com a gramática, como por exemplo *prog()* e *cmd().*

Para tornar o código mais legível e fácil de dar manutenção, foram criadas algumas funções de suporte, como *checkToken()*, responsável por verificar se um tipo de Token específico foi lido e *syntacticError()*, que emite uma mensagem de erro específica caso o Token correto não seja lido.

Para os testes deste módulo foi desenvolvida uma função *syntacticAnalysis()*, que executa toda a análise sintática em um arquivo. Caso não encontre nenhum erro ela simplesmente retorna uma mensagem de confirmação, caso contrário uma mensagem correspondente ao erro sintático ou léxico é emitida e a análise é interrompida.

A construção deste módulo foi mais trabalhosa do que o anterior devido a refatoração do código e ajustes necessários, porém não chegou a causar grandes problemas, novamente devido ao método de construção bem especificado e objetivo.

4. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

A Tabela de Símbolos foi inicialmente embutida no código do Analisador Sintático. Para sua utilização foi construída uma estrutura capaz de armazenar o lexema dos símbolos, seu tipo, seu escopo (GLOBAL ou LOCAL), seu grupo (VARIABLE, FUNCTION, PARAMETER ou PROTOTYPE) e um estado booleano Zombie, usado para lidar com parâmetros de funções sem a necessidade de excluí-los da Tabela de Símbolos e perder sua informação. Para facilitar a inclusão de símbolos na tabela foi implementada uma função *storeSymbol()*, assim como outras variáveis de índice e de suporte. O processo de reconhecimento de símbolos e sua inclusão na tabela foi embutido nas funções do Analisador Sintático.

Como teste foi criado um mecanismo simples de impressão da tabela. Após fazer algumas correções menores no gerenciamento da tabela o mecanismo foi comentado pois esse é um processo transparente a um suposto usuário.

5. Análise Semântica

Logo no início da implementação do módulo de análise semântica, o código relativo a Tabela de Símbolos foi transferido para este novo módulo pois os tipos de erros e análises feitas por ele são de natureza semântica, portanto pareceu mais natural que a tabela ficasse neste Analisador.

Após essa rápida refatoração, foi necessário mais uma vez um estudo da especificação em busca das regras semânticas da linguagem. Nesse ponto houveram dificuldades pois o processo de construção desse analisador não parecia seguir um método sólido como os módulos anteriores. Apesar das dificuldades, decidiu-se por implementar as regras semânticas uma-a-uma, embutindo-as no código do Analisador Sintático. A cada regra, eram executados alguns testes para garantir que o funcionamento do Analisador Sintático e da Tabela de Símbolos não haviam sido comprometidos, assim como verificar o funcionamento correto da regra semântica implementada.

Para reduzir o “inchaço” no código do Analisador Sintático, foram implementadas algumas funções de suporte para, por exemplo, fazer varreduras na Tabela de Símbolos e comparar parâmetros ou verificar ordem de declarações de funções e protótipos (assinaturas).

Esse foi o módulo mais complicado de se construir e foram necessárias diversas leituras na especificação para garantir que as regras foram bem entendidas e implementadas. A falta de uma “rigidez” de método de implementação foi sentida. Apesar das dificuldades acredita-se que as verificações de consistência de tipos em expressões, declarações corretas de variáveis e funções, entre as outras regras, foram bem implementadas e estejam funcionando bem pois foram testadas com códigos de outras pessoas e os resultados foram positivos.

6. Gerador de Código da MP

O módulo de geração de código do CCmm foi implementado bem aos poucos, assim como o Analisador Semântico. Os testes também foram executados a cada novo ponto de geração de código para confirmar que estivesse correto. Devido a diversos fatores externos, a parte final desse módulo não foi bem implementada, de modo que uma boa parte do código carece de otimização e testes.

Inicialmente foram criadas funções e variáveis responsáveis por gerar um arquivo de texto que receberia o código gerado do CCmm. Também cuidou-se para que este arquivo fosse devidamente fechado e mensagens fossem enviadas no caso de erros. Após isto, variáveis e funções de suporte foram implementadas para auxiliar no próprio processo de geração de código. Dentre elas existe a função *codeGen()* que simplesmente insere um comando no arquivo de código gerado, *generateLabel()* responsável por gerar Labels numerados para a geração de código e um pequeno conjunto de procedimentos responsáveis por associar as chamadas de função do Cmm com os Labels corretos.

O processo de implementação foi feito lentamente no início devido a algumas dúvidas e dificuldades, mas assim que foram sanadas o processo correu sem grandes problemas graças à estrutura já várias vezes testadas do Analisador Sintático.

7. Conclusão

O projeto do CCmm foi definitivamente uma das melhores experiências na área de programação executados durante o curso de Sistemas de Informação. O método de construção provou-se muito eficaz e auxilia em muito no desenvolvimento modular de um projeto tão grande, uma característica considerada indispensável.

Apesar disso, ficou claro que é necessário um entendimento pleno de cada uma das fases do projeto pois qualquer dúvida, por menor que seja, pode não apenas produzir erros na fase atual como também pode repercutir de forma grave em módulos posteriores.

A conclusão é que o desenvolvimento do CCmm foi bastante positivo, mesmo que inúmeros fatores externos tenham tornado a qualidade do módulo final (Gerador de Código), inferior aos demais.