Compilador de C−−

Felipe Bagni e Gabriel Yugo Nascimento Kishida

***Resumo*— Este documento apresenta a modelagem e implementação de um Compilador da linguagem denominada C−−, com o uso de linguagem de montagem ASM. A linguagem C−− é uma versão simplificada da original C.**

***Palavras-Chave*— Linguagem de montagem, linguagem de programação, programação de baixo nível, abstração, compilador, máquina de estados.**

***Abstract*— This document contains the modeling and implementation of a C−− Compiler, using the assembly language. The C−− programming language is a simpler version of the original C .**

***Keywords*— Assembly language, programming language, low-level programming, compiler, state machine.**

I. Introdução

A computação é uma ciência que trabalha muito com abstrações e simplificações de processos computacionais de baixo nível – o que permite com que programadores consigam desenvolver algoritmos e lógicas capazes de realizar processos extremamente complexos, como aqueles envolvidos em *Machine Learning* e *Inteligência Artificial*.

No entanto, para atingir este nível de complexidade, é necessário implementar algoritmos que realizam simplificações - para não ser necessário ter que lidar com, por exemplo, comandos de armazenamento e alocação de memória, o que pode engessar o desenvolvimento de programas mais complexos.

É neste ponto em que entram os compiladores : programas que traduzem códigos com maior nível de abstração para níveis menos abstratos e mais próximos da linguagem de máquina – possibilitando que um programa escrito em uma linguagem próxima da humana seja interpretada por uma máquina.

Neste projeto, foi desenvolvido um código para um compilador, que será discutido a seguir.

A. Compilador de C−−

Foi requerido dos alunos de PCS 3616 - Sistemas de Programação - para que desenvolvessem um compilador de uma versão simplificada da linguagem C (chamada C−−, será referenciada como *cmm* a partir deste ponto do relatório) em ASM. Neste caso, *cmm* é a linguagem de abstração de mais alto nível enquanto ASM é a linguagem de mais baixo nível.

Felipe Bagni, Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da USP, São Paulo, SP, Brasil, e-mail: febagni@usp.br; Gabriel Yugo Nascimento Kishida, Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da USP, São Paulo, SP, Brasil, e-mail: gabriel.kishida@usp.br

Para tanto, desenvolveram-se uma série de subrotinas (em linguagem de montagem) para tratar o *buffer* de dados recebidos de um arquivo *".cmm"*, separar as informações importantes por meio de diferentes algoritmos (análises léxica, sintática e semântica) e, por fim, escrever a versão compilada do código *".cmm"* em ASM como *output*.

II. Desenvolvimento

Para compreender o desenvolvimento da solução em questão, primeiro é preciso entender alguns conceitos por trás de compiladores e da própria linguagem C −−, além de algumas premissas feitas para o escopo da solução afim de fazer uma prova de conceito.

A. Compiladores em uma casca de noz

Um compilador se trata de um dos módulos do software básico de um computador, suja função é a de efetuar automaticamente a tradução de textos, redigidos em uma determinada linguagem de programação para alguma outra forma que viabilize sua execução, que em geral é a forma de uma linguagem de máquina. Assim, dada uma linguagem de entrada de alto nível (*linguagem-fonte*), o compilador tem a função de converter textos escritos nessa linguagem (*textos-fonte*) em textos correspondentes equivalentes (*textos-objeto*), apresentados em uma linguagem de saída, denominada *linguagem-objeto*.

Fig. 1: Fluxo compilador

Adicionalmente à função de tradução propriamente dita, cabe aos compiladores executar várias funções auxiliares, entre elas a mais importante é a detecção de erros. Além dela, é conveniente que o compilador mostre mensagens sobre o erro em questão. Outra atividade a ser feita pelo compilador é a de permitir ao usuário a inclusão de comentários em seu texto-fonte, facilitando assim a documentação do mesmo. Assim, é necessário que o compilador filtre do texto-fonte todos os comentários incluídos pelo programador.

A estruturação lógica de um compilador é dividida em 3 principais componentes:

1) Analisador Léxico: este componente é responsável por verificar se todas as palavras escritas no código-fonte são coerentes e não infringem as regras da linguagem-fonte e registrar identificadores e variáveis. É representado por

uma máquina de estados finita, na qual se lê a entrada de caracteres, um de cada vez, mudando de estado de acordo com os caracteres que encontra. Se a análise retornar que está correta, prossegue-se para a próxima análise, caso contrário rejeita-se a entrada e uma mensagem de erro é mostrada;
2) Analisador Sintático: este componente é responsável por verificar se as sequências de palavras no código recebido dão coerentes e não infringem as regras da linguagem-fonte. É representado por uma máquina de estados finitos.
3) Analisador Semântico: este componente é responsável por traduzir as funções da linguagem-fonte para a linguagem-objeto e escrever o arquivo de saída com o código final.

B. A linguagem C−−

A linguagem C−− é uma versão simplificada da famosa linguagem C. Para simplificar o desenvolvimento, vamos nos restringir às seguintes funcionalidades:

- Variáveis do tipo *"int"*, que representam variáveis inteiras de 2 bytes:
- Variáveis do tipo *"char"*, que representam caracteres (1 byte);
- Desvio condicional *"if"*, que somente executa determinadas instruções quando a condição é satisfeita;
- Desvio obrigatório *"goto"*, que desvia para o rótulo especificado como parâmetro:
- Leitura de variáveis pelo teclado com o comando *"scan"*, para receber valores durante o runtime;
- Escrita de variáveis na tela do monitor com o comando *"print"*, para imprimir valroes durante o runtime.

Além dessas funcionalidades, existem alguns requisitos que devem ser cumpridos para que o compilador funcione corretamente:

- Cada palavra (conjunto de letras em sequência) deve ter um *blank* (espaço, quebra de linha, ou NULL) antes e depois de sua escrita. Isso significa que uma linha do tipo > *"int a = x;"* está incorreto *"int a = x ;"*, separando "x"da pontuação.
- Cada variável e rótulo deve ter um ou dois caracteres;
- O compilador aceita no máximo 10 variáveis e 10 rótulos.

Um exemplo de código escrito em C−− é:

```
int a ;
int b ;

a = 5 ;
b = 10 ;
a = a + b ;

rt print ( a ) ;

// Comentario aleatorio

a = b + a ;
if ( a < 20) {
    goto ( rt ) ;
}
```

C. Leitura e Armazenamento do Programa

O esforço inicial feito no sentido de fazer a leitura do código-fonte em C−− e armazená-lo para um acesso posterior, uma vez que há a necessidade de revistá-lo nas diferentes análises supracitadas. Essa necessidade veio do fato da leitura do programa se dar por meio do comando GD (Get Data) no endereço /300 e só pode ser efetuado uma única vez, com o "cursor"se movendo para a próxima posição, sem possibilidade de retorno.

Assim, a solução encontrada foi usar as posições de memória finais do código em questão para guardar o código-fonte redigido em ASCII e o uso de ponteiros, ou seja, variáveis que armazenam outros endereços e por meio da movimentação de tais ponteiros, pode-se fazer a leitura e releitura do código-fonte.

Para auxiliar neste processo, criaram-se algumas subrotinas que facilitavam a análise do texto-programa que é lido:

- "LEITURA_TOTAL": realiza a primeira leitura do programa utilizando o comando GD, e escreve o que é lido no final do programa, de maneira ordenada. Além disso, essa subrotina também marca qual é o último endereço com escrita no programa;
- "READ_PROGRAM": realiza a releitura do programa da mesma forma que o comando GD realiza. (Antes de iníciar a chamada deste programa, é necessário zerar a variável de endereço "CURRENT_ADDR").

Essas subrotinas preparam os dados lidos para serem submetidos a análises futuras, como a análise léxica:

D. Analisador Léxico

A análise léxica é uma sub-rotina responsável por verificar se todas as palavras escritas no código são coerentes e não infringem as regras da linguagem-fonte. O ponto desta análise é verificar os caracteres em individual, e seu posicionamento em relação a espaçamentos e quebras de linhas. A análise de palavras como um todo ocorrerá somente na análise sintática.

Para realizar tal análise, precisamos pegar os caracteres individualmente e analisá-los no contexto do código, e verificar se fazem sentido estarem onde estão. Portanto, criaram-se subrotinas para realizar essas diferentes análises: como "LEX_PARSE", que é responsável por obter 4 bytes de ASCII e isolá-los em pares ("LEX_WORD1"e "LEX_WORD2"), permitindo a comparação por letra a letra do texto em C−−.

Também foram criadas subrotinas para verificar se as letras correspondem a determinados casos, como "IS_BLANK", que verifica se a letra isolada corresponde a um espaço vazio (espaço, EOL, NULL). Também existem rotinas para verificar se são letras (de "a"a "Z"), e para verificar se são algarismos (de 0 a 9).

No entanto, essas verificações só são úteis quando se analisa o contexto em que cada caractere se encontra. Para isso, utilizamos a seguinte máquina de estados finita:

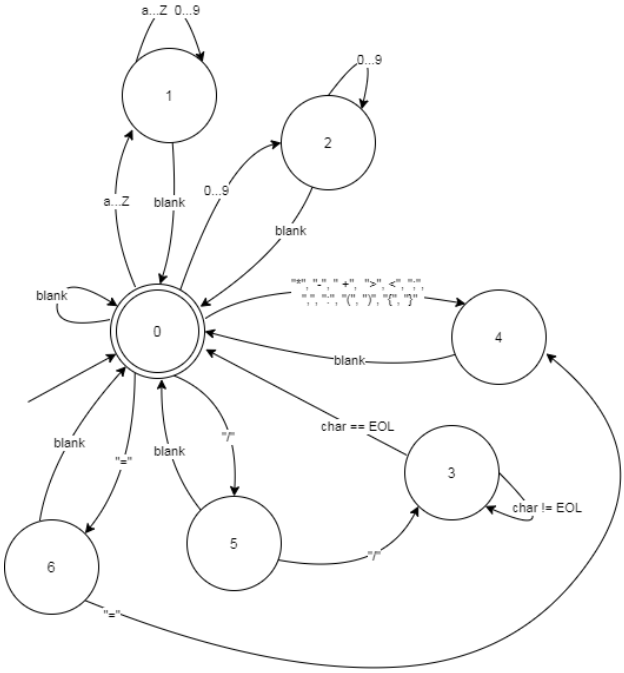

Fig. 2: Máquina de estados finita para a análise léxica

Essa máquina de estados foi implementada utilizando uma variável para armazenar o estado, e chamando uma subrotina chamada "LEX_STEP"para cada letra lida do programa. Dessa forma, damos um passo para cada letra do sistema, e avaliamos se o programa está coerente segundo os critérios pré-determinados – tudo isso utilizando as subrotinas de verificação e a função de *"parse"* para poder analisar os caracteres individualmente.

E. Analisador Sintático

O analisador sintático é responsável por reconhecer sequências de palavras como comandos, interpretando a ordem em que aparecem e que caracteres especiais a dividem.

Desta forma, o analisador sintático verifica se a ordem das palavras é coerente ou se infringe as regras do C−−. Nesta linguagem, cada comando vem separado por um caractere de ponto e vírgula (;). No entanto, existem um leque de diferentes comandos com diferentes comportamentos e sintaxes – e por isso, deve-se analisar cada caso.

Para tanto, foram criadas subrotinas em ASM que leem palavras e as analisam (como "int", "char", "goto", etc.), verificando se foram corretamente escritas ou se tem alguma incoerência. A ideia aqui é reconhecer as palavras chaves da linguagem e ir alterando os estados da máquina de estados finita de acordo ocm o input de comando feito, verificando se a estrutura do comando está correta ou não.

Com isto em mente, criou-se a subrotina "READ_WORD": especializada em isolar estas palavras, identificando os espaços em branco e escrevendo palavras individualmente. Esta subrotina necessita de outra, que ajusta as palavras (pois, como o buffer da MVN é de 4 bytes, as letras são lidas em pares - o que possibilita o armazenamento de uma palavra cuja primeira letra é um espaço em branco.), eliminando o possível espaço em branco inicial. Esta função é chamada de "ADJUST_WORD", e nela normalizamos o que é lido, para facilitar comparações.

A partir disso, desenvolve-se uma máquina de estados finita, que analisa a sequência de palavras do código e interpreta-as como comandos.

F. Analisador semântico

O analisador semântico é responsável por traduzir os comandos inseridos (utilizando o que foi interpretado pelo analisador sintático) em comandos em linguagem de montagem.

Dessa forma, precisamos transportar comandos simplificados como "a = b + a ;"para uma linguagem de montagem, no qual este comando deverá ser traduzido para um comando de LOAD, com ADD e um comando de MOVE TO MEMORY.

E escrever isso no documento de saída "codigo.asm", uma possibilidade é fazer o analisador semântico concomitantemente ao analisador sintático, ao se perceber que o comando lido está sintaticamente correto, pode ser feita a tradução para a linguagem-objeto. Outra opção seria a de manter o formato de cascata e fazer a rotina após a análise sintática.

III. Observações Finais

A. Bootstrapping

Bootstrapping é um processo no qual uma linguagem simples é usada para traduzir programas mais complicados que, por sua vez, podem ser usados para programas mais complicados. Este programa complicado pode lidar com programas ainda mais complicados e assim por diante.

Como visto acima, escrever um compilador para qualquer linguagem de alto nível é um processo complicado e demanda-se muito tempo para escrever um compilador do zero. Portanto, uma linguagem simples pode ser usada para gerar o código-alvo em alguns estágios. Para entender claramente a técnica de Bootstrapping, considere o seguinte cenário.

Suponha que queira-se escrever um compilador cruzado para a linguagem C. A linguagem de implementação desse compilador é, digamos, C−− e o código-alvo que está sendo gerado está na linguagem ASM. Ou seja, criamos C,C−−,ASM. Agora, se o compilador C−− existente for executado na máquina ASM e gerar código para ASM, ele será denominado C−−,ASM,ASM (escopo do projeto). Agora, se executarmos C,C−−,ASM usando C−−,ASM,ASM, obteremos um compilador C,ASM,ASM. Isso significa um compilador para a linguagem fonte C que gera um código

alvo na linguagem ASM e que roda na máquina ASM.

Então usando o exemplo acima, podemos acompanhar os passos:

1) Primeiro, escreve-se o compilador de C−− em Assembly:

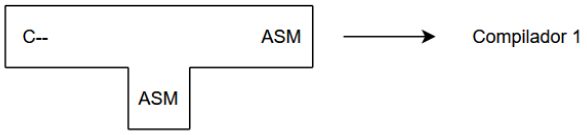

Fig. 3: Exemplo de Bootstrapping - Passo 1

2) Assim, usando a linguagem C−−, escreve-se o compilador de C:

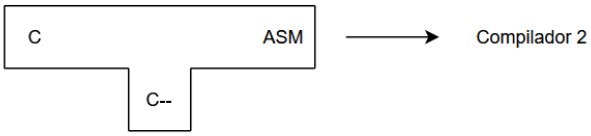

Fig. 4: Exemplo de Bootstrapping - Passo 2

3) Finalmente, compila-se o compilador 2 usando o compilador 1:

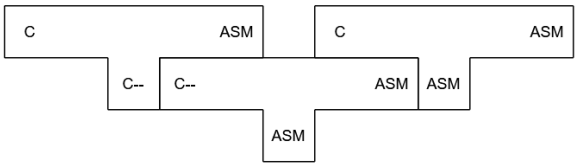

Fig. 5: Exemplo de Bootstrapping - Passo 3

4) Assim, tem-se como resultado um compilador escrito em ASM wue compila C e gera código em ASM.

Dessa forma, uma possível aplicação do projeto é no seu uso para criar um compilador para a linguagem C, utilizando-se da técnica de Bootstraping.

B. Perguntas finais

- *Os algoritmos propostos podem ser mais eficientes? Como? Se sim, por que a forma menos eficiente foi escolhida?*

  Sim, os algoritmos propostos poderiam ser mais eficientes se parte das análises fossem feitas concomitantemente ou interativamente com a primeira leitura do programa. A forma menos eficiente (de primeiro se realizar a leitura, depois a análise léxica, e depois outras análises) foi escolhida pois, como o desenvolvimento do compilador foi iterativo, era necessário primeiro ter uma análise completa para certificar de que o que foi realizado estava funcionando. Se todas as análises fossem feitas de maneira simultânea, testar o desenvolvimento seria muito mais difícil (devido ao volume de informações), e também mais difícil de projetar e modelar tudo de maneira simultânea.

  Portanto, essa perda de eficiência foi feita em detrimento de um desenvolvimento mais organizado e didático.

  Além disso, sob um ponto de vista do usuário, poderia ser feito um relatório final da compilação com os erros. Da forma que foi feita, ao encontrar o primeiro erro, o compilador pararia sua execução a apontaria o erro. Dessa forma, para depurar o código, o trabalho se torna custoso, pois cada erro é apontado e, por conseguinte, corrigido por vez. Com um relatório dos erros encontrados disponibilizado ao final da depuração, o código se torna bem mais fácil de depurar e o compilador mais amigável ao usuário.

- *Os códigos escritos podem ser mais eficientes? Como? Se sim, por que a forma menos eficiente foi escolhida?*

  Sim, os códigos escritos poderiam ser mais eficientes.

  Para tanto, deveria ser feita uma refatoração do código para analisar armazenamentos desnecessários e repetidos, com o objetivo de reutilizar espaços da memória para compactar o espaço da RAM utilizado.

  Além disso, o código está bem modularizado – dividiu-se grande parte das rotinas em subrotinas, e chamadas que se assemelham a funções de linguagens de programação de mais alto nível. Dessa forma, os programadores podem abstrair grande parte das coisas que ocorrem em subrotinas e permite-se a escrita de um código mais simples de ser lido e interpretado pelo humano. No entanto, há uma perda computacional em se modularizar as subrotinas tão extensivamente.

  Portanto, mais uma vez, essa perda computacional foi feita em detrimento de um desenvolvimento mais organizado e didático.

- *Qual foi a maior dificuldade em implementar o compilador?*

  A maior dificuldade em implementar o compilador foi a própria teoria de compiladores. Foi necessária uma pesquisa para entender o contexto de cada analisador (léxico, sintático e semântico) e a função dos mesmos dentro do compilador. A complexidade destes analisadores é bem alta, e os diagramas fornecidos no enunciado, embora úteis não são o suficiente para o desenvolvimento - existiu a necessidade de uma modelagem mais extensa e específica.

  Para suprir essas dificuldades, aulas sobre especifidades

dos compiladores e de analisadores (divisão do código em átomos e tokens, etc.) seriam de grande auxílio.

- *O que teria que ser mudado no seu código para poder suportar definição de funções? E para suportar o uso de arrays (tanto de int como de char)?*

  Para suportar definição de *arrays*, teríamos que mudar a máquina de estados finita do analisador sintático e mudar o analisador semântico. O analisador léxico não seria alterado, pois a máquina de estados não seria alterada – exceto pelo fato de que seria necessário adicionar o símbolo correspondente a ponteiros ("*"e ""no C original). Dessa forma teria que ser feita uma identificação da declaração dessas variáveis ponteiro, e alocar nelas o valor equivalente ao endereço da variável.

  Se houver um jeito de alocar um espaço na memória para variáveis adjacentes a esse endereço (com comandos semelhantes ao *malloc* de C), seria possível criar vetores (uma sequência de espaços que podem ser atingidos por meio de um ponteiro inicial).

  Dessa forma, podemos alocar variáveis sequencialmente – justamente a definição de um array. Podemos estender esse conceito para criar um vetor de vetores: alocando em cada espaço de um vetor ponteiros que apontam para uma sequência de valores – criando assim, uma matriz (um array bidimensional). Desta forma podemos estender o conceito *ad infinitum*, criando vetores n-dimensionais.

  Já para declarar funções, seria recomendado utilizar o comando SC (Subroutine Call) e RS (Return from Subroutine) para desviar para subrotinas e retornar delas. Para passar parâmetros para essas funções, seria necessário separar um espaço na memória do qual as subrotinas retirariam valores para utilizar nos processos. Assume-se que o valor retornado será armazenado no acumulador – mas podem se armazenar estes valores em variáveis disponíveis para o programa principal *main*.

IV. Conclusões

Com o desenvolver do projeto, é bem visível que a quantidade de trabalho envolvida em um compilador é muito grande. Para tanto, há grande mérito em equipes que desenvolvem compiladores para linguagens, já que são estes a fundação que permitem o desenvolvimento de códigos mais complexos. Tarefas como leitura, interpretação e comparação são difíceis de serem executadas em baixo nível, já que não são triviais. É necessário uma boa organização, modularização e uma boa modelagem para conseguir construir um compilador bem feito. Além disso, um bom tratamento de exceções deve ser feito para analisar corretamente o que está sendo interpretado e ter certeza de que o que está sendo inserido pode ser convertido em código.

Como supracitado, a extensão do trabalho de um compilador é grandíssima. Desta forma, apenas partes do compilador foram construídas – como o analisador léxico completo e testado (explicado nas seções anteriores), e as sub-rotinas do analisador sintático, que serviram como prova de conceito.

Portanto, como conclusão, uma nota de admiração para os desenvolvedores que criaram compiladores a partir de linguagens de baixo nível, já que é um trabalho extenso e fundamental para o progresso da computação. Como exemplo, citamos Dennis Ritchie, criador do primeiro compilador da linguagem C, desenvolvida em Assembly.

V. Agradecimentos

A equipe de desenvolvimento do Compilador C−− também agradece o apoio provido pelos monitores da disciplina PCS3616, que foram de imensurável auxílio no aprendizado da linguagem de montagem e linguagem de máquina, sempre tirando dúvidas e sendo atenciosos.
Também agradecemos o professor Ricardo Luis de Azevedo da Rocha.

Referências

[1] João José Neto, *Introdução à Compilação*. 1986.

Apêndice

Os códigos desenvolvidos com seus comentários podem ser encontrados no seguinte repositório do GitHub.