4º Trabalho de Compiladores - Interpretador para a linguagem Humildade++

André Pacheco e Thaylo Xavier

September 10, 2013

Abstract

Documentação referente à quarta parte do trabalho da disciplina de compiladores: Implementação de um interpretador para a linguagem Humildade++ baseado na linguagem G-Portugol.

1 Introdução

O objetivo deste trabalho é a construção de um interpretador para a linguagem Humildade++, que é baseada na linguagem G-Portugol. Para o desenvolvimento do interpretador, é necessário a utilização dos três analisadores desenvolvidos nos trabalhos anteriores, ou seja, os trabalhos anteriores são incorporados ao interpretador, criado aqui, fazendo com que o interpretador verifique erros léxicos, sintáticos e semânticos, antes de executar as instruções da linguagem. A seguir serão descritas as principais ideias, linhas de códigos e dificuldades encontradas, para o desenvolvimento do interpretador.

2 Menu

O interpertador H++ disponibiliza um menu para o usuário escolher o que deve ser feito. O menu consiste das seguintes opções:

1 Compilar programa

2 Executar programa

3 Listar programas no diretório corrente

4 Listar programas compilados

5 Imprimir árvore de execução

6 Sair

Portanto, as opções descritas no menu são basicamente o que deve ser implementado aqui no interpretador.

3 Visão geral

Como já foi dito, o interpretador utiliza os analisadores já feitos anteriormente, portanto, as estruturas de dados utilizadas no analisador sintático, são "herdadas" aqui. As principais estruturas criadas no interpretador são a MasterNode e a PilhaArv. Ambas são descritas a seguir:

```
struct MasterNode
{
   int id_unico;
   char str1[50];
   char str2[50];
   void *data;
   int dataType;
   Variavel *var;
   Funcao *func;
   struct MasterNode *prox;
   struct MasterNode *filhos[3];
};

struct PilhaArv
{
   MasterNode *pilha[max_stack_size_t];
   int quant;
};
```

A estrutura MasterNode é uma lista e árvore ao mesmo tempo. Ela pode ter no máximo três filhos, na esquerda, no meio e na direita, caracterizando uma árvore, e ela também aponta para próxima estrutura de mesma forma. Nela são contruída a árvore de cada comando do programa e encadeado todas as instruções encontradas nele. Além disso, ela possui um id único, que é utilizado como debuger, duas strings, para indicar o que é a instrução lida, uma ponteiro para void, para armazenar qualquer valor constante, um inteiro que representa o tipo do dado armazenado, e um ponteiro para a estrutura Variável e outro para estrura Funcao, ambas já descritas no analisador semântico.

Para armazenar todas as instruções existe uma dificuldade para saber se um comando é "pai" de algum outro, se ele é "filho" ou se ele está sozinho. Para isso foi criado a estrutura PilhaArv, que nada mais é que uma pilha de MasterNode. A ideia é seguinte: Quado encontramos um comando da linguagem, já sabemos de antemão que ele possui um número pré-determinado de filhos. Portanto, quando instruções são encontradas elas necessáriamente devem ser empilhadas até que se encontre o "pai" delas. E como já é conhecido o número de filhos de cada "pai", ele desempilha exatamente esse número, tendo como "filhos", o número de intruções desempilhados. Utilizando um exemplo para o que foi dito acima temos:
Utilizando como exemplo uma expressão $x := x + 1$. Inicialmente empilha-se 1 e x e quando encontra-se o operador +, sabemos que deve ser dempilhados duas posições da pilha corrente, ou seja, 1 e x, que se tornam filhos do operador +. Feito isso, o nó do operador + deve ser empilhado. Em seguida, empilha-se

o x mais a esquerda e encontra-se o operador de atribuição. Quando é encontrado este, sabemos que deve ser desempilhado duas posições da pilha, ou seja, a expressão de soma + e x, que se tornam filhos do operador de atribuição, que ao final de tudo isso, se empilha, pois pode pertencer a um bloco de comando, como um um se-então, por exemplo, ou simplismente fica na pilha e ao fim da anáise do programa, será um nó na lista de masterNode apontando para um próximo nó.

4 Criação da Pilha e MasterNode

A criação da pilha e do MasterNode andam lado a lado. Para implementar toda ideia descrita na seção anterior, foi criado os arquivos *E*xpressao.c e *C*omandos.c; Sendo assim, separou-se as expressões de comando para construção de ambas as estruturas. Nestes arquivos pode ser encontrado todos os passos para cada tipo de comando. A seguir é exibido um trecho de código para tratamento de um comando:

```
case cmd_se:
{
    MasterNode *m = criaNode("cmd", "se");
    insereNode(m,desempilhaNode(pArv),'d'); // ELSE
    insereNode(m,desempilhaNode(pArv),'m'); // THEN
    insereNode(m,desempilhaNode(pArv),'e'); // EXP
    empilhaNode(pArv,m); // se empilhando
    break;
}
```

Acima está descrito um trecho de código da parte de comandos, mais especificamente o comado se. Como é intuitivo, o arquivo possui um switch para filtra todos os comando correntes. Entrando no caso que lhe pertença, é criado um nó MasterNode e é desempilhado exatamente três posições na pilha, que se tornarão os "filhos" do comando se, e logo após este comando se empilha. Isso é feito análogamente para todos os outros comandos e expressões do programa, sendo o número de "filhos"/desempilhamentos, de acordo com a gramática construída.

Dois casos especiais são os tratamentos de variáveis e funções. Para isso é aproveitado a estrutura utilizada no trabalho 3. Quando é encontrada uma variável, o ponteiro de Variável que está dentro da estrutura MasterNode é apontado para essa variável encontrada. E no caso de funções, isso será discutido na próxima seção.

5 Árvores de execução e Árvore de execução de Função

Tudo que foi descrito nas seções anteriores, fazem parte da construção de ambas as árvores. Aqui será discutido algumas peculiaridades da contrução. Primeiramente, a árvore de execução nada mais é que a *m*ain() do programa e a árvore de execução de função são as funções contidas no programa. Ambas funcionam

da mesma maneira, porém quando iniciado a função principal do programa é que a pilha esteja vazia, para ocorrer os procedimentos já citados. Sendo assim, a partir da pilha criada para as instruções da função, é criado uma lista de MasterNode que representa a àrvore de execução de função, que é armazenada, em um ponteiro para MasterNode, que está dentro da estrutura de Função.
Após avaliadas as funções, como as declarações devem ser realizadas obrigatóriamente antes da main, todas as instruções a seguir pertencerão a árvore de execução do programa. Sendo assim, com o conteúdo que ainda está na pilha, é construída a árvore de execução.

6 Executando a Árvore de Execução

Para executar a árvore de execução, seja ela de função, ou não, foi criado o arquivo *e*xecuta.c; Nele possui a função *e*xecutaArvore, que recebe como parâmetro um MasterNode. Nesta função é avaliada todas as expressões e comandos contidas na árvore, utilizando recursão para avaliação de todas as folhas da mesma.
A função *e*xecutaComando trata todos os comandos e expressões escritos no documento. Para cada uma dela existe uma ação préviamente definido, e com isso a linguagem pode ser interpretada em cima da linguagem C.

7 Considerações finais

Certamente este trabalho é mais difícil que o três anteriores juntos, e talvez seja o trabalho mais difícil do curso. As primeira dificuldade foi a compreensão do problema, que foi resolvida após uma conversa em sala de aula com a professora. Logo após, veio a dificuldade de montar a árvore de execução. A priore a dificuldade era ter uma boa estratégia para montagem da mesma, e tendo a ideia implementá-la. A implementação da estratégia do empilhamento foi demorada e cheia de bugs que foram sendo encontrados e consertados ao decorrer do desenvolvimento. Montada a árvore, mais dificuldades: como executar? Com certeza a ideia de usar a recursão é fundamental e foi implementada. Não conseguimos executar todos os comandos da linguagem, como por exemplo, chavear, mas quase todos estão funcionando e podem ser observados nos exemplos.
Por fim, a implementalçao do interpretador como um todo, ou seja, todos os analisadores anteriores, é fundamental para a compreensão de como funciona um compilador. A implementação envolve praticamente todas as TAD's estudadas no curso e muita programação. Consideramos um trabalho pesado e talvez o mais difícil do curso. Tudo isso serve para demostrar o quão difícil é o desenvolvimento de um compilador para uma linguagem de programação.