# Compiladores – 2017/1

Trabalho Prático T4

29 de junho de 2017

## 1 Objetivo

O objetivo deste trabalho é a criação de um interpretador de código como *back-end* de um compilador para a linguagem C-Minus.

## 2 Descrição do Problema

Um interpretador depende de uma representação intermediária (*intermediate representation* – IR) do código de entrada. Nesse trabalho vamos usar como IR uma **Árvore de Sintaxe Abstrata** – ***Abstract Syntax Tree (AST)***. Os nós de uma AST correspondem aos comandos da linguagem e os filhos de um nó são as dependências do comando.

Uma AST pode ser vista como uma versão simplificada da *parse tree* aonde somente as informações necessárias para a execução do programa são mantidas. O ponto chave para a construção da AST é definir quais informações devem ficar guardadas na árvore. Uma vez que isso é feito, a implementação do executor de código deve seguir a mesma especificação.

O analisador semântico desenvolvido no Trabalho 3 realiza a construção da AST. Caso o professor tenha indicado algum erro na sua implementação da AST na correção de T3, você deve corrigi-la antes de começar este trabalho.

## 3 Executor de código

O executor de código (interpretador) é um programa que recebe como entrada uma AST e caminha na árvore, realizando as operações indicadas nos nós da AST. Para o armazenamento de resultados temporários é utilizada uma **pilha**. Desta forma, o executor trata a AST como uma IR de um endereço (*one-address IR*).

O caminhamento do executor pela AST deve ser em **pós-ordem**, devido às dependências de cada operação. Ao terminar a execução de uma dada operação, o resultado fica armazenado na pilha, para ser usado depois. Por exemplo, uma operação de soma de inteiros desempilha os dois primeiros valores da pilha, realiza a operação e empilha de volta o resultado.

O caminhamento pela AST pode ser implementado tanto de forma recursiva quanto de forma iterativa, como você preferir. Além da pilha, o executor também possui uma área de memória para o armazenamento dos valores das variáveis do programa. As ações que devem ser executadas pelo interpretador dependem do tipo do nó da AST que está sendo visitado, e são detalhadas a seguir.

- **FUNC LIST**: registra na tabela de funções um ponteiro para a implementação da função (nó da AST), de forma que as chamadas das funções podem recuperar essa informação da tabela. Chama a função `main` para começar a execução do programa.
- **FUNC DECL**: executa o cabeçalho da função e a seguir o corpo.
- **FUNC HEADER**: executa a lista de parâmetros.

- **PARAM LIST**: desempilha os argumentos que estão no topo da pilha de execução, armazenando os valores para os parâmetros em uma área de memória temporária.
- **FUNC BODY**: executa a declaração de variáveis e a seguir os comandos da função.
- **VAR LIST**: inicializa uma área de armazenamento para as variáveis declaradas na função.
- **BLOCK**: executa sequencialmente os comandos do bloco.
- **INPUT**: lê um inteiro do teclado e empilha o valor lido.
- **OUTPUT**: desempilha o valor no topo da pilha e exibe na tela.
- **WRITE**: exibe na tela a *string* indicada no nó filho.
- **ASSIGN**: executa a expressão da direita e atribui o valor deixado na pilha à variável da esquerda. Note que aqui você deve tratar tanto variáveis simples quanto compostas (vetores).
- **NUM**: empilha o valor do número.
- **SVAR**: empilha o valor atual da variável simples.
- **CVAR**: empilha o valor atual da variável composta. O índice no vetor está indicado no nó filho.
- **ARITH OP**: executa as expressões, desempilha os dois primeiros valores e empilha o resultado da operação.
- **COMP OP**: executa as expressões, desempilha os dois primeiros valores e empilha o resultado da operação: 0 se a comparação falhar, 1 caso contrário.
- **IF**: executa o teste e desempilha o topo, se for 1, executa o bloco do 'então', se for 0 e existir um bloco de 'senão', executa este bloco.
- **WHILE**: executa o teste e repete o bloco de comandos enquanto o teste passar.
- **FCALL**: executa os argumentos e a seguir pula com a execução para o nó da árvore aonde a função está declarada.
- **ARG LIST**: avalia os argumentos e empilha os resultados. Deve ser feito no sentido inverso ao PARAM LIST, isto é, se o executor do nó PARAM LIST espera que o primeiro parâmentro esteja no topo da pilha, o executor do nó ARG LIST deve avaliar os argumentos da direita para a esquerda.
- **RETURN**: avalia a expressão de retorno, caso exista, cujo resultado fica no topo da pilha.

# 4 Implementando e Testando o Trabalho

As convenções léxicas, sintáticas e semânticas da linguagem C-Minus já foram apresentadas nas especificações dos Trabalhos 1, 2 e 3, respectivamente. Você deve adaptar o *scanner*, o *parser* e o analisador semântico desenvolvidos anteriormente para compor o interpretador deste trabalho.

O seu programa de entrada é lido da entrada padrão (*stdin*), como abaixo.

```
$ ./trab4 < program.cm
```

A leitura dos valores de entrada para programas com o comando `read` deve ser feita do teclado. No entanto, como o código fonte do programa de entrada é lido por redirecionamento do `stdin`, é necessário restaurar a conexão do `stdin` com o terminal. Faça isso com o comando abaixo, após terminar a leitura do programa de entrada. (Veja um exemplo no gabarito do Laboratório 11.)

```
stdin = fopen(ctermid(NULL), "r");
```

Observações importantes:

- O seu interpretador pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- Os arquivos de entrada de exemplo são os mesmos do Trabalho 1.
- As mensagens de erros léxicos, sintáticos e semânticos são as mesmas do Trabalho 3 e devem continuar sendo exibidas como antes.
- Valide o seu programa usando os gabaritos disponibilizados na sala da disciplina no AVA.
- Ao submeter o seu trabalho para correção, além dos códigos-fonte envie um arquivo `Makefile` que gera como executável para o seu analisador semântico um arquivo de nome `trab4`.

# 5 Regras para Desenvolvimento e Entrega do Trabalho

- **Data da Entrega**: O trabalho deve ser entregue até às 23:55 h do dia 27/07/2017 (Quinta-feira). Não serão aceitos trabalhos após essa data.
- **Grupo**: O trabalho é **individual**.
- **Linguagem de Programação e Ferramentas**: Para implementar o seu analisador sintático você deve obrigatoriamente usar o `bison`.
- **Como entregar**: Pela atividade criada no AVA. Envie um arquivo compactado com todo o seu trabalho.
- **Recomendações**: Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

# 6 Avaliação

- O trabalho vale 4.0 pontos na média parcial do semestre.
- Trabalhos com erros de compilação receberão nota zero. Isso inclui trabalhos com conflitos de *shift-reduce* ou *reduce-reduce*.
- Caso seja detectado plágio (entre alunos ou da internet), todos os envolvidos receberão nota zero.
- Serão levadas em conta, além da correção da saída do seu programa, a clareza e simplicidade de seu código.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre plágio, acima.)