INF1018 - Software Básico (2015.1) Segundo Trabalho

Gerador de Código Dinâmico

O objetivo deste trabalho é desenvolver, em C, uma função **`geracod`** que implementa um pequeno gerador de código dinâmico para uma linguagem de programação bastante simples, chamada *SB*.

A função **`geracod`** deverá ler um arquivo texto contendo o código fonte de uma função escrita em *SB* e retornar um ponteiro para o início da região de memória que contém o código de máquina que corresponde à tradução da função *SB*.

Além disso, deve ser implementada uma função **`liberacod`**, que libera a memória alocada para armazenar o código criado por **`geracod`**.

Leia com atenção o enunciado do trabalho e as instruções para a entrega. **Em caso de dúvidas, não invente. Pergunte!**

A Linguagem *SB*

A linguagem *SB* contém apenas três tipos de instruções: atribuição, repetição condicional e retorno.

- Uma **atribuição** tem a seguinte forma:

  `var = varc1 op varc2`

  Essa instrução faz com que o valor da expressão `varc1 op varc2` seja calculado e atribuído a **`var`**. **`var`** pode ser uma variável local ou um parâmetro da função e **`varc1`** e **`varc2`** podem ser variáveis locais, parâmetros ou constantes inteiras. **`op`** é um dos operadores: + - * (soma, subtração e multiplicação).

  As variáveis locais são da forma **`vi`**, sendo o índice ***i*** utilizado para identificar a variável (ex. `v0, v1`, etc...). Da mesma forma, os parâmetros são da forma **`pi`**, sendo p0 o primeiro parâmetro, p1 o segundo, e assim sucessivamente. A linguagem permite o uso de no máximo 10 variáveis locais e 5 parâmetros.

  Na linguagem *SB* as constantes são escritas na forma **`$i`**. Por exemplo, **`$15`** representa o valor **15** e **`$-15`** representa o valor **-15**.

  Alguns exemplos de atribuição:

  - `v0 = p0 + p1`
  - `v1 = v0 * $100`

    - `p0 = p0 - $1`

- Uma **repetição condicional** tem a seguinte forma:

  ```
  'while' varc
  <instruções>
  'end'
  ```

  Ela repete a execução do bloco de instruções entre o teste de condição (`while varc`) e o terminador (`end`) enquanto o valor de **`varc`** (uma variável local, um parâmetro ou uma constante) for **diferente** de 0 (a condição é testada **antes** da execução do bloco de instruções).

  Por exemplo, o trecho abaixo incrementa a variável local v0 e decrementa o parâmetro p0 enquanto o valor de p0 for diferente de 0:

  ```
  while p0
  v0 = v0 + $1
  p0 = p0 - $1
  end
  ```

- A instrução de **retorno** é apenas

  ```
  'ret'
  ```

  Essa instrução faz com que a função *SB* retorne. O valor de retorno da função é o valor da variável local **v0**.

A sintaxe da linguagem *SB* pode ser definida formalmente como abaixo:

func	::	cmd '\n' | cmd '\n' func
cmd	::	att | while | ret
att	::	var '=' varc op varc
while	::	'while' varc '\n' func 'end'
ret	::	'ret'
var	::	'v' digito | 'p' digito
varc	::	var | '$' snum
op	::	'+' | '-' | '*'
num	::	digito | num digito
snum	::	[-] num
digito	::	0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'| '8' | '9'

Exemplos

Veja a seguir alguns exemplos de funções *SB*. Atenção: os comentários **não** fazem parte da linguagem!

- Um exemplo bastante simples é uma função *f(x) = x + 1*:

```
v0 = p0 + $1
ret
```

- O exemplo a seguir mostra a implementação da função *f(a,b,c) = a * (b + c)* na linguagem *SB*:

```
v0 = p1 + p2    // i = b + c
v0 = p0 * v0    // i = a * i
ret
```

- O próximo exemplo implementa a função *fatorial(n)* em *SB*:

```
v0 = $1 + $0    // f = 1
while p0        // while (n != 0)
v0 = v0 * p0    // f = f * n
p0 = p0 - $1    // n = n - 1
end
ret
```

- Para o último exemplo, considere a seguinte função C:

```
int f(int x, int y) {
  int i,v=0;
  for (; x ; x--)
    for (i = y; i ; i--)
      v += x * i;
  return v;
}
```

  Uma função equivalente em *SB* poderia ser escrita assim:

```
v0 = $0 + $0    // v = 0
while p0
v1 = p1 + $0    // i = y
while v1
v2 = p0 * v1    // x * i
v0 = v0 + v2    // v += x * i
v1 = v1 - $1    // i--
end
p0 = p0 - $1    // x--
end
ret
```

Implementação e Execução

O que fazer

Você deve desenvolver, em C, uma função chamada **`geracod`** que leia um arquivo de entrada contendo o código fonte de **uma** função na linguagem *SB*, gere o código de máquina IA32 correspondente, e retorne um valor do tipo "ponteiro para função"; este valor será o endereço da área de memória que contém o código gerado.

O protótipo de **`geracod`** é o seguinte:

```
typedef int (*funcp) ();
funcp geracod (FILE *f);
```

O único parâmetro de **`geracod`** é o descritor de um arquivo texto **já aberto para leitura**, de onde deve ser lido o código fonte *SB*. Esse código fonte terá no máximo 50 linhas.

Você deverá desenvolver também uma função que libere a área de memória alocada por **`geracod`**, com o protótipo

```
void liberacod (void *p);
```

Implementação

Para cada instrução *SB* imagine qual uma tradução possível para *assembly*. Além disso, lembre-se que a tradução de uma função *SB* deve começar com o prólogo usual (preparação do registro de ativação, incluindo o espaço para variáveis locais) e terminar com a finalização padrão (liberação do registro de ativação antes do retorno da função). O código gerado deverá seguir as convenções de C/Linux quanto à passagem de parâmetros, valor de retorno e salvamento de registradores.

O código gerado por `geracod` deverá ser um ***código de máquina IA-32***, e não um código fonte assembly. Ou seja, você deverá descobrir o código de máquina que corresponde às instruções de assembly que implementam a tradução das instruções da linguagem *SB*. Para isso, você pode usar o programa `objdump` e possivelmente a documentação das instruções da Intel, disponível na página do curso.

Por exemplo, para descobrir o código gerado por `movl %eax, %ecx`, você pode criar um arquivo meuteste.s contendo apenas essa instrução, traduzi-lo com o gcc (usando a opção -c) para gerar um arquivo objeto `meuteste.o`, e usar o comando

```
objdump –d meuteste.o
```

para ver o código de máquina gerado.

A função **`geracod`** deve alocar um bloco de memória para escrever o código gerado. O valor de retorno de `geracod` será um ponteiro para essa área alocada. Lembre-se que as instruções de máquina ocupam um número variável de bytes na memória.

Não é necessário fazer o tratamento de erros do arquivo de entrada, você pode supor que o código *SB* estará sempre correto. Vale a pena colocar alguns testes só para facilitar a própria depuração do seu código, mas as entradas usadas como testes na correção do trabalho **sempre estarão corretas**.

Para ler e interpretar cada linha da linguagem *SB*, teste se a linha contém cada um dos formatos possíveis. Veja um esboço de código C para fazer a interpretação de código aqui. Lembre-se que você terá que fazer adaptações pois, dentre outros detalhes, essa interpretação **não será feita na *main***!

Estratégia de Implementação

Este trabalho não é trivial. Implemente sua solução passo a passo, **testando separadamente cada passo implementado!**

Por exemplo:

1. Compile um arquivo *assembly* contendo uma função simples usando:

   ```
   minhamaquina> gcc -m32 -c code.s
   ```

   (para apenas compilar e não gerar o executável) e depois veja o código de máquina gerado usando:

   ```
   minhamaquina> objdump -d code.o
   ```

   Construa uma versão inicial da função **`geracod`**, que aloque uma área de memória, coloque lá esse código "colado" do compilador, bem conhecido, e retorne o endereço da área alocada.

   Crie uma função `main` e teste essa versão inicial da função (leia o próximo item para ver como fazê-lo). Teste também a sua função de liberação de memória (chamada pela `main`!)

2. Ainda sem começar a traduzir uma função *SB* lida de um arquivo, você pode implementar a "montagem" dinâmica de um código que contenha o prólogo e a finalização da função. Novamente, teste essa implementação.
3. Comece agora a implementação de atribuições e operações aritméticas e da instrução de retorno. Pense em que informações você precisa extrair para poder traduzir as instruções (quais são os operandos, qual é a operação, onde armazenar o resultado da operação). Implemente a leitura e "interpretação" do código fonte *SB* (você pode se basear no código fornecido aqui).

   Implemente **e teste** uma operação por vez. Experimente usar constantes, parâmetros, variáveis locais, e combinações desses tipos como operandos.

   Lembre-se que é necessário alocar espaço (na pilha) para as variáveis locais!

4. Deixe para implementar a instrução `while` apenas quando **todo o resto** estiver funcionando!

   Pense em que informações você precisa guardar para traduzir completamente essa instrução (note que há **desvios** envolvidos nessa tradução.

Testando o gerador de código

Você deve criar um arquivo contendo a função `geracod` e **outro arquivo** com uma função `main` para testá-la.

Sua função *main* deverá abrir um arquivo texto que contém um "programa fonte" na linguagem *SB* (i.e, uma função *SB*) e chamar *geracod*, passando o arquivo aberto como argumento. Em seguida, sua *main* deverá chamar a função retornada por *geracod*, passando os parâmetros apropriados, e imprimir o valor de retorno dessa função. Esse retorno é um valor inteiro, que pode ser exibido com código de formação ("%d\n"). A função criada por *geracod* é a tradução da função *SB* lida do arquivo de entrada.

Para testar a chamada de uma função *SB* com diferentes parâmetros, sua função *main* pode receber argumentos passados na linha de comando. Para ter acesso a esses argumentos (representados por *strings*), a sua função `main` deve ser declarada como

```
int main(int argc, char *argv[])
```

sendo **`argc`** o número de argumentos fornecidos na linha de comando e **`argv`** um array de ponteiros para *strings* (os argumentos).

Note que o primeiro argumento para `main` (argv[0]) é sempre **o nome do seu executável!**. Os parâmetros que deverão ser passados para a função criada por `geracod` serão o argumento 1 em diante.

Note também que os argumentos recebidos pela `main` são **strings**! Para convertê-los para inteiros, você pode usar a função `atoi`.

Entrega

Deverão ser entregues **via Moodle** dois arquivos:

1. Um arquivo fonte chamado **`geracod.c`** , contendo as funções **`geracod`** e **`liberacod`** (e funções auxiliares, se for o caso).
   - Esse arquivo **não** deve conter a função `main`.
   - Coloque no início do arquivo, como comentário, os nomes dos integrantes do grupo da seguinte forma:

     ```
     /* Nome_do_Aluno1 Matricula Turma */
     /* Nome_do_Aluno2 Matricula Turma */
     ```

2. Um arquivo texto, chamado **`relatorio.txt`**, contendo um pequeno relatório.
   - O relatório deverá explicar o que está funcionando e o que não está funcionando. Não é necessário documentar da sua função no relatório. Seu código deverá ser claro o suficiente para que isso não seja necessário.
   - O relatório deverá conter também **alguns** exemplos de **funções** da linguagem *SB* que você usou para testar o seu trabalho. Mostre tanto as funções *SB* traduzidas e executadas com sucesso como as que resultaram em erros (se for o caso).
   - Coloque também no relatório o nome dos integrantes do grupo

Indique na área de texto da tarefa do Moodle o nome dos integrantes do grupo. Apenas uma entrega é necessária (usando o *login* de um dos integrantes do grupo) se os dois integrantes pertencerem à mesma turma.

Prazo

- O trabalho deve ser entregue **até meia-noite do dia 19 de junho**.
- Trabalhos entregues com atraso perderão **um ponto por dia de atraso**.

Observações

- Os trabalhos devem preferencialmente ser feitos **em grupos de dois alunos** .
- Alguns grupos poderão ser chamados para apresentações orais / demonstrações dos trabalhos entregues.