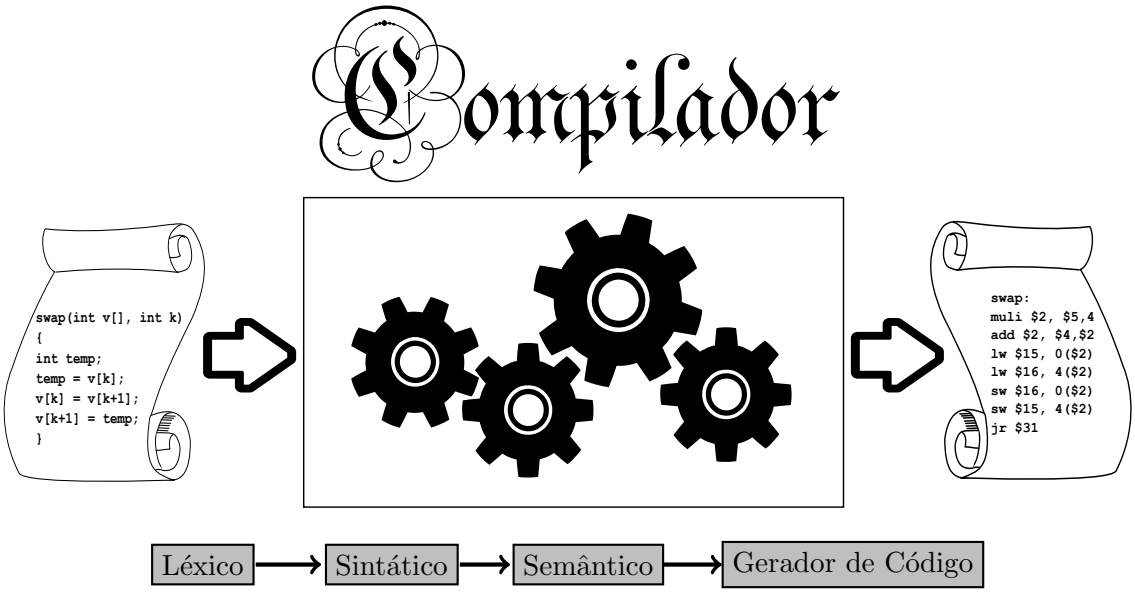
Universidade Estadual de Londrina

DC Departamento de Computação

1COP029 - Gerador de Código: *Assembly* de MIPS



Implemente um programa que recebe como entrada arquivos que contêm representações lineares de árvores sintáticas abstratas e gera o código *assembly* equivalente. O código *assembly* gerado deve ser para a arquitetura MIPS de 32 bits. O código *assembly* gerado deve ser executável no emulador QtSpim[1] ou similar. Você pode utilizar as ferramentas `flex` e `bison` se desejar, mas seu uso não é obrigatório.

De forma a exemplificar o formato dos arquivos que conterão as representações das árvores sintáticas abstratas, considere o seguinte trecho de código fonte:

Exemplo de entrada:

```
int j;

#define hkey 666

void f()
{
    int i;
    int w[10];
```

Versão deste texto: V. 2023.01

[1] http://spimsimulator.sourceforge.net

```
    if(j) { w[i++] = 0; }
    ++j;
    return;
}
```

Tal código fonte corresponde a seguinte representação de árvore sintática abstrata (AST):

```
AST -->

CONSTANT: hkey VALUE: 666
GLOBAL VARIABLE: j TYPE: int
FUNCTION: f
   RETURN_TYPE: void
   VARIABLE: i TYPE: int
   VARIABLE: w TYPE: int[10]

IF(j,=(w[(i)++],0));
++(j);
RETURN()
END_FUNCTION
```

Constantes: Na AST gerada o valor constante `hkey` é indicado através da linha `CONSTANT: hkey VALUE: 666`. As constantes sempre irão aparecer em uma linha após o texto `CONSTANT:`, seguido do nome da mesma e o seu valor após o texto `VALUE:`. As constantes sempre serão valores inteiros.

Variáveis Globais: Na AST gerada a variável global `j` é indicada através da linha `GLOBAL VARIABLE: j TYPE: int`. Após o texto `GLOBAL VARIABLE:` aparece o nome da variável e após o texto `TYPE:` aparece o tipo da variável. Todas as variáveis globais seguirão o mesmo esquema de declaração. Os tipos possíveis são: `int`, `char`, `void` e ponteiros para estes tipos. Vale observar que o tipo básico void sempre será um ponteiro, ou seja, ele só irá aparecer na forma `void*`, `void**`, `void***`, etc. Os tipos `int` e `char` podem ou não aparecer como ponteiros.

Funções: As funções sempre se iniciarão após o texto `FUNCTION:` seguido do nome da respectiva função. Na linha subsequente irá aparecer o tipo de retorno da função após o texto `RETURN_TYPE:`. A função pode retornar qualquer um dos tipos os quais uma variável pode ter, além de ter o tipo de retorno `void`, isto é, não retornar nenhum valor, como no exemplo apresentado. O escopo de uma função será finalizado através da linha `END_FUNCTION`.

Variáveis Locais: Quando uma função possuir variáveis locais, elas serão indicadas após o texto `VARIABLE:`, seguido do nome da variável e do seu tipo. As variáveis locais podem ter os mesmos tipos das variáveis globais. Quando a variável for um vetor, a sua dimensão será indicada após o tipo. O mesmo esquema se aplica a variáveis globais que sejam vetores.

Comandos: Os comandos irão aparecer dentro do escopo de uma função. Comandos não aninhados são separados por ponto-e-vírgula. Os comandos do escopo principal de uma função irão aparecer um por linha. Os comandos aninhados irão sempre aparecer na mesma linha do comando ao qual pertencem, mais ainda lembrando que comandos de mesmo escopo, mesmo que aninhados dentro de outro comando, ainda serão separados por ponto-e-vírgula. No exemplo apresentado, o escopo principal da função `f` possui 3 comandos (onde cada um

aparece em uma linha), sendo que a função como um todo possui 4 comandos (observe que um deles está aninhado dentro do `if`.

Estrutura dos Comandos

Os comandos possíveis são: `do-while`, `if`, `while`, `for`, `printf`, `scanf`, `exit`, `return`, e expressões matemáticas. A estrutura de cada um dos comandos será detalhada a seguir.

- `do-while`:

  `DO_WHILE(` <*comandos*> `,` <*condição de parada*> `)`

  Os comandos e a condição de parada são separados por uma vírgula. Como exemplo, considere o seguinte trecho de código: `do{ j++; }while(j)`. Na AST ele irá aparecer como:

  ```
  DO_WHILE((j)++,j)
  ```

- `if`:

  `IF(` <*expressão condicional* > `,` <*lista de comandos then* > `,` <*lista de comandos else* > `)`

  A expressão condicional, a lista de comandos do bloco `then` e do bloco `else` são separadas por vírgula. Como exemplo, considere o seguinte trecho de código: `if(j){j=j;}else{j=j;}`. Na AST ele irá aparecer como:

  ```
  IF(j,=(j,j),=(j,j))
  ```

- `while`:

  `WHILE(` <*condição de parada*> `,` <*lista de comandos*> `)`

  A condição de parada é separada da lista de comandos por vírgula. Como exemplo, considere o seguinte trecho de código: `while(j) {j++;}`. Na AST ele irá aparecer como:

  ```
  WHILE(j,(j)++)
  ```

- `for`:

  `FOR(` <*inicialização*> `,` <*condição de parada*> `,` <*ajuste de valores*> `,` <*lista de comandos*> `)`

  Toda a separação dos blocos do `for` é feita por vírgulas. Os três primeiros blocos são as expressões que inicializam algum contador, verificam a condição de parada e atualizam os valores do contador. Devido a natureza dessa estrutura, esses 3 primeiros blocos não são obrigatórios, podendo então qualquer um deles estar vazio ou mesmo todos eles. A lista de comandos é uma parte obrigatória da estrutura do `for` e sempre estará presente. Como exemplo, considere o seguinte trecho de código: `for(j;j;j){j++;}`. Na AST ele irá aparecer como:

  ```
  FOR(j,j,j,(j)++)
  ```

  Considere agora o seguinte trecho de código: `for(;;){j++;}`. Na AST ele irá aparecer como:

  ```
  FOR(,,,(j)++)
  ```

- `printf`:

  `PRINTF(` < *string para impressão* > `,` <*expressões*> `)`

  No `printf` a *string* para impressão é sempre obrigatória, já as expressões para impressão são opcionais e caso presentes serão separadas por vírgula. Para ilustrar o funcionamento do `printf` considere os exemplos a seguir e como eles irão aparecer na AST.

  ```
  printf("Skynet Online")
  PRINTF("Skynet Online")

  printf("j:  %d\n",j)
  PRINTF("j:  %d\n",j)

  printf("j:  %d hkey:  %d\n",j,hkey)
  PRINTF("j:  %d hkey:  %d\n",j,hkey)
  ```

- `scanf`:

  `scanf(` <*string de formato*> `,` <*endereço da variável a ser lida*> `)`

  No `scanf` a *string* de formato mostra o tipo do dado a ser lido. O campo endereço da variável a ser lida indica qual a variável que receberá o valor lido pelo usuário. Ambos os campos são separados por uma vírgula. Como exemplo, considere o seguinte trecho de código: `scanf("%d",&j)`. Na AST ele irá aparecer como:

  ```
  SCANF("%d",&(j))
  ```

- `exit`:

  `exit(` <*expressão*> `)`

  O comando `exit` retorna um valor através da expressão que ele receber como parâmetro. Como exemplo, considere o seguinte trecho de código: `exit(j+1)`. Na AST ele irá aparecer como:

  ```
  EXIT(+(j,1))
  ```

- `return`:

  `return(` <*expressão*> `)`

  O comando `return` pode retornar um valor através de uma expressão que ele receber como parâmetro. Para ilustrar o funcionamento do `return` considere os exemplos a seguir e como eles irão aparecer na AST.

  ```
  return;
  RETURN()
  ```

  ```
  return (666+0);
  RETURN(+(666,0))
  ```

Estrutura de Expressões

As expressão são formadas por operadores binários ou unários. A estrutura das expressões corresponde basicamente a um percurso em pré-ordem na árvore de expressões. Para Operadores binários, a estrutura será sempre da seguinte forma:

`BOP (` <*filho esquerdo*> `,` <*filho direito*> `)`

onde `BOP` corresponde a algum operador binário e os filhos esquerdo e direito são outras estruturas de expressões, construindo assim uma árvore binária. Como exemplo, considere o seguinte trecho de código:

```
i = j + k - l;
```

Na AST ele irá aparecer como:

```
=(i,-(+(j,k),l))
```

Para operadores unários existem duas formas distintas de construção:

`UOP (` <*expressão filha*> `)`

ou

`(` <*expressão filha*> `) UOP`

onde `UOP` corresponde a algum operador unário e a expressão filha corresponde a expressão onde o operador é aplicado. A forma `(` *<expressão filha>* `) UOP` só existe para os operadores de pós-incremento e pós-decremento. Como exemplos de operadores unários, considere os trechos de código a seguir e como ele irão aparecer na AST.

```
i++;
(i)++

++i;
++(i)
```

Operador Ternário

O operador ternário `?` é um caso a parte dos operadores binários e unários. Este tipo de operador terá sempre a seguinte estrutura:

`?` `(` *<expressão para avaliação>* `,` *<expressão se verdadeiro>* `,` *<expressão se falso>*`)`

Como exemplo, considere o seguinte trecho com duas linhas de código:

```
a = b>0 ?  c+1 :  d-1;
  a>0 ?  b+1 :  c-1;
```

Na AST ele irá aparecer como:

```
=(a,?(>(b,0),+(c,1),-(d,1)));
  ?(>(a,0),+(b,1),-(c,1));
```

Compatibilidade de tipos entre operadores

A Tabela 1 apresenta os operadores que devem ser considerados/implementados, as combinações possíveis de tipos que podem ocorrer, bem como cada combinação é tratada por cada um do operadores presentes na linguagem C simplificada. Nessa tabela, considere o seguinte:

✓: Implica que a operação é válida.

•: Implica que existem informações adicionais sobre a combinação, as quais aparecem após a tabela de compatibilidade.

Um espaço vazio implica que a combinação de tipos não é compatível para a operação apresentada e não precisa ser considerada.

- Os operadores binários `==, !=, <, >, <=, >=` não são válidos caso a comparação seja feita entre dois tipos diferentes de ponteiros como entre `int*` e `int**` e desta forma tal tipo de comparação não irá ocorrer. Para comparações entre um tipo ponteiro com um tipo `int` ou `char`, ou para ponteiros de mesmo tipo, o valor retornado é do tipo `int`, pois uma comparação retorna `0` ou `1`.

Operadores Binários	int/int	char/char	int/char	ponteiro/int	ponteiro/char	ponteiro/ponteiro
+ PLUS	✓	✓	✓	•	•	
- MINUS	✓	✓	✓	•	•	
* MULTIPLY	✓	✓	✓			
/ DIV	✓	✓	✓			
% REMAINDER	✓	✓	✓			
& BITWISE_AND	✓	✓	✓			
| BITWISE_OR	✓	✓	✓			
^ BITWISE_XOR	✓	✓	✓			
&& LOGICAL_AND	✓	✓	✓	✓	✓	✓
|| LOGICAL_OR	✓	✓	✓	✓	✓	✓
== EQUAL	✓	✓	✓	•	•	•
!= NOT_EQUAL	✓	✓	✓	•	•	•
< LESS_THAN	✓	✓	✓	•	•	•
> GREATER_THAN	✓	✓	✓	•	•	•
<= LESS_EQUAL	✓	✓	✓	•	•	•
>= GREATER_EQUAL	✓	✓	✓	•	•	•
>> R_SHIFT	✓	✓	✓	•	•	
<< L_SHIFT	✓	✓	✓	•	•	
= ASSIGN	✓	✓	✓			•
+= ADD_ASSIGN	✓	✓	✓	•	•	
-= MINUS_ASSIGN	✓	✓	✓	•	•	

Operadores Unários	int	char	ponteiro
+ PLUS	✓	✓	
- MINUS	✓	✓	
* MULTIPLY			✓
++ INC	✓	✓	✓
-- DEC	✓	✓	✓
~ BITWISE_NOT	✓	✓	
! NOT	✓	✓	✓

Tabela 1: Compatibilidade entre tipos e operadores

- Os operadores binários `<<` e `>>` quando utilizados com ponteiros, devem necessariamente ter o tipo ponteiro como o operando esquerdo e o tipo `int` ou `char` como o operando direito. A combinação inversa é inválida e não irá ocorrer. O tipo retornado nesta operação é o tipo do ponteiro que está no lado esquerdo da operação. Na sua implementação, considere que o operador de deslocamento `>>` corresponde ao operador de deslocamento a direita **aritmético**, isto é, o sinal será sempre preservado.

- Os operadores binários de atribuição `=, +=, -=` quando utilizados para atribuir um valor para um ponteiro, requerem que a expressão do lado direito gere um ponteiro do mesmo tipo que o operando do lado esquerdo. Assim um identificador do tipo `int**` só pode receber uma expressão cujo resultado seja também do tipo `int**`. Atribuições entre tipos diferentes de ponteiros não irão ocorrer.

- O operador binário de subtração `-` quando utilizado com ponteiros, terá necessariamente o tipo ponteiro como o operando esquerdo e o tipo `int` ou `char` como o operando direito. A combinação inversa é inválida e não irá ocorrer. O tipo retornado nesta operação é o tipo do ponteiro que está no lado esquerdo da operação.

- O operador binário de soma `+` quando utilizado com ponteiros, pode ter o tipo ponteiro como o operando esquerdo ou direito. Desta forma qualquer combinação entre o tipo ponteiro e os tipo `int` ou `char` é válida. O tipo retornado nesta operação é o tipo do ponteiro que está presente na operação de soma.

Uma `string` pode aparecer em uma expressão, sendo que ela irá corresponder ao tipo `char*`. Como a `string` não pode ser modificada em tempo de execução, o tipo `char*` associado a `string` funciona como uma constante, a qual não pode ser modificada, somente lida.

Parâmetros de função declarados como `arrays` podem receber ponteiros que sejam equivalentes ao tipo base do `array`. Por exemplo um parâmetro que seja declarado como `int a[10][10]`, nada mais é que um ponteiro do tipo `int*`. Desta forma um identificador que tenha o tipo `int*` é compatível com o parâmetro `int a[10][10]` de uma função qualquer. É importante ressaltar que o contrário também é verdadeiro, ou seja, uma função que tenha um parâmetro do tipo `int*` pode receber um `array` cujo tipo base seja `int`, não importando quantas dimensões tal `array` possua.

Ainda em relação à `arrays`, um identificador que tenha sido declarado como `int** a;` pode ser utilizado em uma expressão como um `array` através da sintaxe `a[5][5]`, por exemplo. Já um identificador que tenha sido declarado como `int a[5][5]`, por exemplo, também pode ser acessado através da forma `a[1]` ou somente `a`, sendo que em ambos os casos, o tipo retornado é `int*`, pois não foram utilizadas todas as dimensões declaradas.

Os operadores binários `<<` e `>>` quando empregados em expressões válidas, sempre retornam o tipo do operando esquerdo. Assim a expressão `'a'<< 2` retorna o tipo `char`.

Demais operadores binários que utilizarem uma combinação entre os tipos `int` e `char` irão retornar o tipo `int`, como por exemplo nas expressões `'c' + 2` ou `'c' * 'a'`, onde ambas irão retornar o tipo `int`.

IMPORTANTE: O operador unário `&` (para a tomada de endereços) **não** estará presente nos arquivos de teste e não precisa ser implementado.

Observações

Para este trabalho considere os seguintes tamanhos para cada um dos tipos:

- `void` 8 bits
- `char` 8 bits
- `int` 32 bits
- `ponteiro` 32 bits

Ambiente de Execução: QtSpim

O código *assembly* gerado pelo seu programa será executado no software simulador chamado QtSpim, capaz de carregar e executar programas escritos em código *assembly* para a arquitetura MIPS de 32 bits.

Em princípio, o QtSpim possui um funcionamento semelhante ao MARS[2] (MIPS Assembler and Runtime Simulator) e ambos devem ser capazes de executar o código gerado sem nenhum problema e podem ser utilizados para se testar o código gerado. Ambos os programas são gratuitos e possuem versões para diversos sistemas operacionais. Vale ressaltar que a correção será feita utilizando-se o QtSpim.

Na sua implementação, você terá que gerar código para os comandos: `printf`, `scanf` e `exit`. Na implementação desses comandos, você irá utilizar *syscalls* presentes nos simuladores de MIPS para se gerar o código equivalente.

O arquivo com código *assembly* pode empregar diversas diretivas como `.data`, `.text`, entre outras. Além disso, o programa é terminado através de uma *syscall* específica. Para obter mais informações sobre as diretivas que podem estar presentes, bem como o funcionamento de cada um das *syscalls*, consulte as informações sobre o QtSpim disponíveis no *AVA*.

Alocação de Registradores

No processo de alocação de registradores, você terá que lidar basicamente com variáveis globais, variáveis locais, valores temporários e argumentos de funções.

Para variáveis globais, você pode manter o nome delas inalterado ao longo do código. Variáveis globais não são alocadas em registradores e devem ser declaradas no seção `.data` do seu código *assembly*.

As variáveis locais devem necessariamente ser alocadas em registradores. No seu código, você deve utilizar os registradores de `$s0` até `$s7`. Como existem somente 8 desses registradores, os códigos de teste terão um máximo de 8 variáveis locais por função, bastando então associar um registrador a cada uma das variáveis locais declaradas.

Como os registradores de `$s0` até `$s7` são *calle save* (valor salvo), você deve salvar o valor desses registradores na pilha no início da execução de uma função e restaurar o valor salvo na saída da mesma.

Os valores temporários são valores que são gerados como resultado de uma computação intermediária de um código e que não estão explicitamente declarados no código fonte. Considere o seguinte trecho de código:

```
f = (g+h) - (i+j)
```

[2] http://courses.missouristate.edu/kenvollmar/mars/

onde a associação entre variáveis e registradores foi a seguinte: `f:$s0`, `g:$s1`, `h:$s2`, `i:$s3`, `j:$s4`. Esse código poderia ser traduzido para *assembly* de MIPS da seguinte forma:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Observe que o resultado da soma das variáveis `g` e `h` gera um valor temporário que foi armazenado do registrador `$t0` e que o resultado da soma das variáveis `i` e `j` foi armazenado no registrador `$t1`. O resultado dessas duas somas são valores temporários que são resultados intermediários dessas computações que são necessários para se computar a operação de subtração. Esse tipo de valor deve ser armazenado nos registradores de `$t0` até `$t9`, ou seja, existem um total de 10 registradores que podem ser utilizados para se armazenar resultados intermediários de computações, bem como na tradução das estruturas de controle.

Os registradores temporários `$t0` até `$t9` são *caller save*, isto é, **não são salvos na pilha ao início de uma função**. Esses registradores devem ser salvos na pilha somente quando o seu valor estiver *vivo* ao longo de uma chamada de função. Considere o seguinte trecho de código *assembly* de MIPS:

```
add $t0, $s0, $zero
jal foo_function
add $s1, $v0, $t0
```

Observe que no código apresentado, o registrador `$t0` recebeu um valor que só será utilizado após a chamada da função `foo_function`. Caso a função `foo_function` utilize o registrador `$t0` em seu código, o valor desse registrador será destruído, e o código exemplo não irá funcionar como se espera. Como não é possível saber se a função `foo_function` utiliza ou não o registrador temporário `$t0` e o mesmo está *vivo* ao longo da chamada de função, o correto é salvar esse registrador na pilha antes da chamada de função e restaurar seu valor logo após o retorno da função. Como o exemplo apresentado não faz isso, ele está **errado**.

O código do exemplo pode ser corrigido realizando o devido salvamento, ficando então da seguinte forma:

```
add $t0, $s0, $zero
addi $sp, $sp, -4    #alocacao de espaco na pilha
sw $t0, 0($sp)       #salvamento do registrador
jal foo_function
lw $t0, 0($sp)       #restauracao do registrador
addi $sp, $sp, 4     #desalocacao do espaco na pilha
add $s1, $v0, $t0
```

Observe que antes da chamada da função foi alocada uma posição de 32 bits na pilha, onde foi salvo o registrador `$t0`, e logo após o retorno da função o valor do registrador foi restaurado e a memória alocada na pilha foi liberada.

Dessa forma, sempre que o seu código armazenar valores nos registradores temporários e eles estiverem vivos ao longo de uma chamada de função, você deve necessariamente salvar/restaurar os mesmos ao longo dessa chamada. Somente os registradores temporários que estão vivos ao longo de uma chamada de função é que devem ser salvos/restaurados.

Por fim, o processo de alocação terá que lidar com os argumentos de funções. Para isso, o alocador deve utilizar os registradores de `$a0` até `$a3`. Como existem somente 4 registradores para argumentos, os códigos de teste irão ter funções com um máximo de 4 argumentos.

Exemplo de Código e AST

Considere o seguinte código de entrada:

```
#define c1 10
#define c2 -c1
#define c3 c1 << 2
#define c4 c2 >> c3*c3
#define NULL 0
#define CALL_ELEVEN 11
int v[c2+c3];

void** f()
{
    void** v;
    return v;
}

#define TRUE  1
#define FALSE 0
#define SERA_VERDADE (TRUE || FALSE)
char* vetor[SERA_VERDADE];

char g(int arg_1,char arg_2,int arg_3)
{
    char* c;
    c = "DDominar o mundo\n";
    printf("%s",c+1);
    c++;
    printf("E depois novamente ");
    while(*c!='\0')
    {
        printf("%c",*c);
        g(1+2*3-4*5,*c,CALL_ELEVEN+666);
    }
    return *(vetor[1 << 1]);
}

int main()
{
    int *ptr;
    int* i;
    for(ptr=v; ptr;ptr++)
    {
        while(*ptr)
        {
            if(ptr<=0)
            {
```

```
                *ptr = 666;
                *i /*Porta para o mundo invertido*/ = CALL_ELEVEN + 666;
            }
            else
            {
                do{ ptr++; }while(v);
            }
        }
    }
    return (*ptr + *i);
}
```

AST gerada:

```
AST -->

CONSTANT: SERA_VERDADE VALUE: 1
CONSTANT: NULL VALUE: 0
CONSTANT: TRUE VALUE: 1
CONSTANT: c1 VALUE: 10
CONSTANT: c2 VALUE: -10
CONSTANT: c3 VALUE: 40
CONSTANT: c4 VALUE: -10
CONSTANT: FALSE VALUE: 0
CONSTANT: CALL_ELEVEN VALUE: 11
GLOBAL VARIABLE: v TYPE: int[30]
GLOBAL VARIABLE: vetor TYPE: char*[1]

FUNCTION: f
   RETURN_TYPE: void**
   VARIABLE: v TYPE: void**

RETURN(v)

END_FUNCTION

FUNCTION: g
   RETURN_TYPE: char
   PARAMETER: arg_1 TYPE: int
   PARAMETER: arg_2 TYPE: char
   PARAMETER: arg_3 TYPE: int
   VARIABLE: c TYPE: char*

=(c,"DDominar o mundo\n");
PRINTF("%s",+(c,1));
(c)++;
PRINTF("E depois novamente ");
WHILE(!=(*(c),'\0'),PRINTF("%c",*(c));g(-(+(1,*(2,3)),*(4,5)),*(c),+(CALL_ELEVEN,666)));
```

```
RETURN(*(vetor[<<(1,1)]))

END_FUNCTION

FUNCTION: main
   RETURN_TYPE: int
   VARIABLE: ptr TYPE: int*
   VARIABLE: i TYPE: int*

FOR(=(ptr,v),ptr,(ptr)++,WHILE(*(ptr),IF(<=(ptr,0),=(*(ptr),666);=(*(i),+(CALL_ELEVEN,666)),DO_WHILE
RETURN(+(*(ptr),*(i)))

END_FUNCTION
```

• A linha que contém o comando `FOR` foi quebrada apenas para caber na página. No arquivo que contém a AST, tal texto irá aparecer em uma única linha. Algumas quebras de linha também foram removidas para que a AST coubesse em uma única página.

Especificações de Entrega

O trabalho deve ser entregue no *AVA* em um arquivo .zip com o nome `gerador.zip`. Este arquivo .zip deve conter somente os arquivos necessários à compilação, sendo que deve haver um `Makefile` para a geração do executável.

A entrega deve ser feita exclusivamente no *AVA* até a data/hora especificada. Não serão aceitas entregas atrasadas ou por outro meio que não seja o *AVA*.

Observação: o arquivo .zip não deve conter pastas, para que quando descompactado, os fontes do trabalho apareçam no mesmo diretório do .zip. O nome do executável gerado pelo `Makefile` deve ser `gerador`.

O programa gerado deve ler as suas entradas da entrada padrão do sistema e imprimir as saídas na saída padrão do sistema. Um exemplo de execução para uma entrada chamada `ast.txt` seria a seguinte:

```
$./gerador < ast.txt
```

IMPORTANTE: Arquivos ou programas entregues fora do padrão receberão nota **ZERO**. Entende-se como arquivo fora do padrão aquele que tenha um nome diferente de `gerador.zip`, que contenha subpastas ou que não seja um `.zip` por exemplo. Entende-se como programa fora do padrão aquele que não contiver um `Makefile`, que apresentar erro de compilação, que não ler da entrada padrão, não imprimir na saída padrão ou o nome do executável for diferente de `gerador`, por exemplo.