Projeto de Compilador
E6 de **Geração de Código Assembly**

Prof. Lucas Mello Schnorr
schnorr@inf.ufrgs.br

1 Introdução

A sexta etapa do trabalho de implementação de um compilador consiste na geração de código assembly a partir do código intermediário gerado na etapa anterior. O código assembly gerado como resultado deve poder ser traduzido para um executável por um montador. Utilizaremos como referência o assembly `x86_64` e o compilador `gcc` como montador.

2 Funcionalidades Necessárias

2.1 Traduzir para assembly o código ILOC

Implementar uma função `generateAsm()` que traduz o programa em linguagem intermediária obtido na etapa anterior para a linguagem assembly na saída padrão. Esta função deve gerar na saída o segmento de dados (com o valor de inicialização das variáveis globais) e o segmento de código. O segmento de dados é criado a partir da tabela de símbolos do escopo global, além de constantes literais se por ventura o grupo optou por tal estratégia em etapas anteriores. O segmento de código deve conter a tradução das instruções ILOC a partir da lista de instruções (que contém todo o programa) na raiz da AST. Normalmente, uma instrução ILOC é mapeada para uma instrução assembly, mas esse número pode variar um função das escolhas do grupo. Por exemplo, uma instrução ILOC pode se tornar algumas instruções assembly. Esse mapeamento entre instruções do código intermediário e do código assembly deve ser decidido, projetado e implementado pelo grupo.
Devem ser traduzidos os elementos que foram previamente implementados na etapa anterior, onde o código ILOC foi gerado como código intermediário.

A Dicas Básicas

A.1 Entrada e Saída Padrão

Organize a sua solução para que o compilador leia da entrada padrão o programa em nossa linguagem e gere o programa em assembly na saída padrão. Dessa forma, pode-se realizar o seguinte comando (`etapa6` é o binário do compilador):

```
./etapa6 < entrada > saida.s
```

O código assembly traduzido deverá ser capaz de ser reconhecido e montado para um programa executável através do seguinte comando (onde `programa` é um programa executável): `gcc saida.s -o programa`

A.2 Arquivo `main.c`

Utilize a função principal no arquivo `main.c` semelhante aquela já implementada na etapa anterior. O grupo deve modificá-la, caso necessário, para implementar as funcionalidades necessárias da etapa corrente. Não esqueça de liberar a memória corretamente, como uma boa prática de programação.

B Testes automáticos

Os testes automáticos utilizarão o compilador `gcc` (versão 12 ou mais recente) para verificar se o código assembly gerado pode ser transformado em um binário executável que efetue as operações do programa fornecido na entrada. O teste utilizará o valor de retorno da função principal `main` na avaliação. Por exemplo, para o programa `ex3.z` com o código:

```
int main() {
   int a;
   int b;
   a = 1;
   b = 6;
   while (a < b) {
      a = a + 1;
   };
   return a;
}
```

O valor de retorno do programa acima deve ser 6:

```
./etapa6 < ex3.z > ex3.s
gcc ex3.s -o ex3
./ex3
echo $?
```

C Assembly

Existe extensa documentação na internet para a linguagem assembly em sua versão `x86_64` e o compilador `gcc` como montador. Um bom tutorial para ter uma visão geral da linguagem assembly está disponível. Recomenda-se no entanto que o grupo estude a linguagem a partir de exemplos práticos, gerados a partir de programas minimalistas escritos na linguagem C e traduzidos para assembly com o compilador `gcc` através do comando `gcc -S programa.c`.

C.1 Simples

Por exemplo, assumindo que o código do seguinte programa minimalista esteja no arquivo `ex1.c`:

```
int a = 3;
int main() {
  int b;
  b = a;
  return 0;
}
```

Podemos obter o código assembly com:

```
gcc -S ex1.c
cat ex1.s
```

Na parte inicial da saída no arquivo `ex1.s`, teremos o segmento de dados, com informações da variável global `a` e da função `main`. Percebam o valor 3 da inicialização da variável `a` e, tamanho `.size` e o tipo `.type` de cada objeto.

```
        .file   "ex1.c"
        .text
        .globl  a
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   3
        .text
        .globl  main
        .type   main, @function
```

Na parte seguinte, temos o segmento de código, iniciado por um rótulo para a função principal, com uso do registrador que aponta para o topo da pilha (`%rsp`), do registrador que aponta para a base do registro de ativação (`%rbp`), do registrador de acumulação (`%eax`), e do registrador `%rip` para fazer referência às variáveis globais. Literais inteiros devem aparecer antecedidos de `$` diretamente na saída. A chamada de função pode ser simplesmente uma instrução `call` e o retorno para a função anterior com a instrução `ret`.

```
main:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    a(%rip), %eax
        movl    %eax, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
```

O binário `ex1` pode ser obtido a partir do comando:

```
gcc ex1.s -o ex1
```

C.2 Completo

O exemplo `ex2.c` abaixo demonstra outro exemplo mais completo que envolve uma chamada de função com dois parâmetros.

```
int mult (int z, int w)
{
  int x;
  if (z > 0) {
    x = z * w;
  }else{
    x = w;
  };
  return x;
}

int main()
{
  int x;
  int y;
  x = 2;
  y = mult (x, x);
  return 0;
}
```

C.3 Mais exemplos

Recomenda-se fazer outros exemplos na linguagem C e traduzí-los para a linguagem assembly usando o método acima. Estudar a saída obtida e implementar a tradução para assembly no compilador de maneira semelhante senão idêntica. Repetir esse procedimento até que todas os elementos que foram traduzidos para ILOC na etapa anterior tenham sido traduzidos para assembly.