Relatório Compiladores

Etapa 6 - Geração de código de máquina

Contextualização

Para essa etapa, já possuíamos as TACs do nosso código gerado, e analisados semanticamente. Na parte de análise semântica, fiz uma pequena implementação de parâmetros locais, mas não variáveis locais de uma função como um todo. Dessa forma, se eu estiver dentro de uma função e utilizar uma variável global, ela altera em todas as suas possíveis chamadas (inclusive recursivas).

Escolha de ambiente

O código assembly gerado utiliza a sintaxe AT&T de Assembly, gerando código linkável pelo gcc (*gcc.exe (tdm-1) 5.1.0*) para Windows 64 bits. O código pode ser inspecionado usando gdb (que foi utilizado durante os testes, inclusive), porém não possui todas as flags de debug possíveis geradas, então o debug pode ser um pouco complicado já que não temos como se referir a linha original no código da linguagem *ere2020*.

Simplificações

Chamadas de funções recursivas foram implementadas, porém elas compartilham variáveis globais e, portanto, precisam ser implementadas com cuidado para não sobrescrever valores globais, gerando problemas na recursão, com possível perda de valor. Nos exemplos abaixos temos a implementação do fatorial e do fibonacci recursivo, funcionando corretamente, já que tomam os cuidados necessários. Como temos um escopo local para os parâmetros das funções, podemos criar funções recursivas que utilizam somente variáveis passadas por parâmetro, e que, portanto, possuem um escopo definido, para evitar quaisquer problemas.

A função de print somente sabe lidar com números inteiros e com strings, printando a representação ASCII para caracteres.

Todos os números são tratados como inteiros. Por uma simplificação que foi permitida através de questionamento ao professor, números com ponto flutuante são sempre interpretados como 0, e portanto, ficaram fora da parte de geração de código Assembly, embora sejam corretamente interpretados e analisados semanticamente.

Código com erro semântico e/ou sintático não terá seu código Assembly gerado, já que ele está incorreto e/ou inválido.

Implementação da área de dados

Todos os valores são criados na área de dados com uma única passada pela *hash table*.

Todas as variáveis tem um underscore (_) adicionado na frente delas na área de dados. Variáveis temporárias são inicializadas com valor 0, embora seja garantido que elas não serão utilizadas antes de ter um valor atribuído a elas.

Vetores sem inicialização (permitidos sintaticamente) tem seu valor inicializado para 0, garantindo valores mais confiáveis e diminuindo a chance de erro pelos usuários da linguagem.

As strings são tratadas especialmente, para criar uma *label* única para cada uma delas (que é utilizada depois na hora do *print*), substituindo variáveis não permitidos em labels para caracteres determinísticos mas que identificam unicamente a string.

As labels estão na área de dados, mas não são adicionadas na área de dados. O mesmo vale para os literais, que são adicionados utilizando indexação absoluta no código assembly, e não como uma referência para a área de dados.

Implementação da área de código

Percorremos nossas TACs, do início até o final, uma única vez, gerando algumas linhas de código assembly para cada TAC.

Existe uma única TAC auxiliar que foi adicionada nessa etapa, que não gera código, e é responsável apenas por informar para o gerador de assembly, que os parâmetros que começarão a ser passados na próxima TAC pertencem a função x. Isso é necessário já que a chamada de funções foi implementada da forma que: para cada parâmetro x da função a ser chamada, nós salvamos o valor de x na pilha (porque ele poderia estar sendo usado na função atual com o mesmo nome, já que permitimos que parâmetros tenham o mesmo nome que outros parâmetros e/ou variáveis globais), e movemos o parâmetro passado para a posição de x. Após chamarmos a função, devolvemos os valores originais removendo-os da pilha. A TAC é utilizada nesse caso, já que, como não sabemos o nome da função que vai ser chamada enquanto estamos salvando os parâmetros na pilha. Assim, temos uma TAC auxiliar antes dos parâmetros, que informa para o gerador de código, a lista com os nomes dos parâmetros da função que será chamada após os parâmetros serem salvos na posição correta.

Dessa forma, ao chamarmos funções, não utilizamos da pilha para variáveis locais, mas sim, para salvar os valores das variáveis para uso futuro, utilizando simples *popq* e *pushq.*

Nossa implementação sempre tem uma função main (o linkador não irá funcionar se não tiver), e permite recursão teoricamente infinita (obviamente limitadas pelo tamanho da pilha de chamadas).

Exemplos

Aqui temos exemplo de código na linguagem *ere2020*, sua representação em Assembly (normalmente um detalhe importante) e o resultado de sua execução. Temos primeiro exemplos mais focados em pequenos detalhes da linguagem, e posteriormente problemas inteiros. Podem ser encontrados alguns exemplos de programas na pasta tests/ nos arquivos no formato **input.test*, com o resultado no arquivo **expected.test*.

```
main() = int {
    print "Minha funçãozinha!"

    x[2] = 16;
    if (6 > 5 ^ x[2] == 16) then
        print "vou printar isso"
};
```

```
# TAC_COPY_IDX
movl    $0x2, %eax
movl    $0x16, %ebx
movl    %ebx, _x(,%eax,4

# TAC_GT
movl    $0x6, %eax
cmpl    $0x5, %eax
setg    %al
movzbl  %al, %eax
movl    %eax, _$temp_1

# TAC_VECTOR_ACCESS
movl    _x+0x8, %eax
movl    %eax, _$temp_2
```

```
$ ./test/output.exe
Minha funcaozinha!
vou printar isso
```

Prints de strings, atribuição a vetores, acesso de vetores e precedência de operadores

```
divisivel_por_3: bool = FALSE;
divisivel_por_5: bool = FALSE;

fizzbuzz() = int {
    loop (i:1,100,1) {
        divisivel_por_3 = (i / 3) * 3 == i
        divisivel_por_5 = (i / 5) * 5 == i
        if (divisivel_por_3 | divisivel_por_5) then {
            if ( divisivel_por_3 ) then
                print "Fizz"
            if ( divisivel_por_5 ) then
                print "Buzz"
        } else print i
    }
};

main() = int {
    x[0] = fizzbuzz()
};
```

```
# TAC_DIVIDE
movl    _i, %eax
movl    $0x5, %ebx
cltd
idivl   %ebx
movl    %eax, _$temp_11

# TAC_MULTIPLY
movl    _$temp_11, %ebx
movl    $0x5, %eax
imull   %ebx, %eax
movl    %eax, _$temp_12

# TAC_EQ
movl    _$temp_12, %eax
movl    _i, %ebx
cmpl    %eax, %ebx
sete    %al
movzbl  %al, %eax
movl    %eax, _$temp_13
```

```
$ ./test/output.exe
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
Fizz
Buzz
16
```

Chamada de funções, comparações aritméticas, condições, atribuições e comparações entre booleanos e loop

```
i = int : 0;
max_fib = int : 20;

fib(n = int, tmp1=int, tmp2=int) = int {
  if (n ⩽ 2) then
    return 1


  tmp1 = fib(n - 1, 0, 0)
  tmp2 = fib(n - 2, 0, 0)

  return tmp1 + tmp2
};

main () = int
{
    print "fib"

    loop (i:1,max_fib + 1,1) {
      print "computing fib →", i
      print fib(i, 0, 0)
    }
};
```

```
# TAC_FUNC_PARAM
pushl   _n
movl    _i, %eax
movl    %eax, _n

# TAC_FUNC_PARAM
pushl   _tmp1
movl    $0x0, %eax
movl    %eax, _tmp1

# TAC_FUNC_PARAM
pushl   _tmp2
movl    $0x0, %eax
movl    %eax, _tmp2

# TAC_FUNC_CALL
call    _fib
movl    %eax, _$temp_8
popl    _tmp2
popl    _tmp1
popl    _n
```

```
computing fib ->
29
514229
computing fib ->
30
832040
computing fib ->
31
1346269
computing fib ->
32
2178309
```

Funções recursivas com utilização de variáveis locais para manter estado. No detalhe como salvamos as variáveis que são parâmetros na pilha. Repare também que vamos até 0x20 = 32 em decimal.

Por comparação, esse código demora 1.5s para rodar no meu computador, e um código C leva 600s com ou sem a flag -O2