



Lab4

Gerando executáveis



Lab4

Nesse laboratório vamos continuar trabalhando com o parser do lab 3, mas agora devemos modificar o nosso “visitor” para gerar um programa executável.

Como gerar o binário seria muito trabalhoso, nós vamos transformar o código de entrada em uma das representações intermediárias do LLVM, um compilador muito utilizado.



LLVM

O LLVM possui 3 representações intermediárias:

- Em memória: Estruturas de dados que formam um grafo que descreve o programa que vai ser gerado.
- Em “binário”: Uma versão linearizada da descrição do programa, usada para maior desempenho de armazenamento e leitura.
- Em texto: Um código muito semelhante ao “assembly” de diversas arquiteturas. Usado para facilitar a leitura e depuração do programa.



LLVM

As 3 versões são equivalentes, isso quer dizer que toda a informação que está presente em uma delas também está presente na outra.

Para facilitar o nosso trabalho, o nosso compilador ao invés de gerar um executável ele vai gerar a representação intermediária de texto do LLVM.

Essa representação é a mais simples de se entender, e por ser muito próxima a um “assembly” ainda faz o nosso compilador reter boa parte dos requisitos de um gerador de executável.



LLVM


A representação de texto, apesar de mais simples que as outras, ainda é muito complexa. Nesses slides vamos explicar apenas o necessário para fazer o nosso compilador.



LLVM

```
@a = global i32 10;
define i32 @f(i32 %x) {
    %1 = mul i32 10, 2
    %2 = load i32, i32* @a
    %3 = add i32 %2, %1
    ret i32 %3
}
define i32 @f2(i32 %x) {
    %1 = call i32 @f(i32 %x)
    %2 = sub i32 %1, 5
    ret i32 %2
}
```

```
var a = 10;
func f(x) a + 10 * 2;
func f2(x) f(x) - 5;
```



LLVM - tipos

Você deve ter percebido que no código da IR existem vários “i32”.

Diversas instruções necessitam que o tipo do dado seja especificado, no laboratório vamos usar apenas o “i32”, que é o equivalente a um “int” em C.



LLVM - variáveis globais

Variáveis globais são declaradas como essa:

- `@a = global i32 10`

Seu identificador deve ser antecedido por '@', e ela é seguida por "= global i32 10" para especificar que um espaço de 32 bits global deve ser alocado e inicializado com 10.

Uma coisa importante a se lembrar: apesar de chamar aqui de variável global, @a não é uma variável, e sim um ponteiro. Para acessar os seus dados serão necessárias operações de load e store.



LLVM - variáveis locais

No exemplo dado vemos duas formas de declarar uma variável global.

- Quando ela é um argumento fizemos “i32 %x”. Funções aceitam uma lista de argumentos separados por “,”
- Quando ela é criada dentro da função ela é declarada atribuindo um valor: “%1 = mul i32 10, 2”. Nesse exemplo multiplicamos dois valores e colocamos o seu resultado na variável %1.

Quando dizemos variáveis local aqui na verdade estamos nos referindo a valores temporários que só existem dentro de uma função, da forma mostrada não sabemos se a variável está na pilha, ou apenas em registradores. É possível forçar que ela esteja na pilha, usando uma instrução para alocar espaço, mas não será necessário para o lab.



LLVM - variáveis locais

O que é importante lembrar sobre variáveis locais:

- Começam com '%' e a partir de então podem 0-9a-zA-Z_.
- Um valor temporário não pode aparecer do lado direito e esquerdo de uma atribuição, apenas em um deles.
- Você não deve economizar neles! Toda operação nova pode ser guardada em um temporário novo.



LLVM - variáveis locais

Um cuidado especial que se deve ter quando trabalhando com temporários com identificadores puramente numéricos ($[0-9]^*$) é que o llvm cria automaticamente valores temporários numéricos, exemplos:

- É possível ao declarar uma função não dar nome aos parâmetros e sim apenas os seus tipos. Quando isso acontece os parâmetros vão ser nomeados a partir de %0.
- Identificadores numéricos também são atribuídos a blocos básicos na falta de labels. Logo se o llvm identifica que existe o início de um novo bloco básico que não recebeu uma label, ele atribui ao bloco o próximo valor numérico disponível dentro do contexto.



LLVM - funções

```
define i32 @f(i32 %x) { ... }
```

Uma função é declarada por “define” seguido do tipo de retorno, um identificador global e uma lista de parâmetros. Assim como em C, o código referente a função fica dentro do par de chaves que segue a definição.

Como já explicado, os parâmetros da função são informados dentro do parêntese por uma lista de valores temporários separadas por vírgula e acompanhadas do tipo.

O único tipo de retorno que nós podemos precisar para o lab além do i32 é o void.



LLVM - retorno

```
ret i32 %3
```

Toda função deve possuir um retorno. Até uma função void deve terminar com “ret void”.

O comando é sempre acompanhado pelo tipo do retorno e, quando o tipo não é void, de um valor que pode ser por exemplo um número constante ou um temporário.



LLVM - Operações matemáticas

```
%1 = mul i32 10, 2  
%3 = add i32 %2, %1
```

São exemplos de operações matemáticas. Um temporário deve ser especificado para receber o resultado. O tipo da operação e os valores, que podem ser tanto constantes quanto temporários.

Lembre-se que o temporário que recebe o valor NÃO pode ser usado como um dos operandos.



LLVM - load e store

```
%2 = load i32, i32* @a
```

Esse é um comando de load, ele é seguido pelo tipo da variável que vai ser carregada, o tipo do endereço e o endereço em si. Como já foi mencionado, @a é um ponteiro para um espaço global que foi declarado do tipo i32, logo @a é do tipo i32*. A operação retorna o valor de tipo i32 que está no endereço @a.

```
store i32 %value, i32* @ptr
```

No exemplo dado não temos instruções de store, mas elas podem ser muito úteis. O comando é seguido do tipo e valor a ser guardado em memória e o tipo e valor do endereço.



LLVM - ferramentas

A IR gerada deve ser guardada em um arquivo de tipo “.ll”

Usando o comando:

- `llc arquivo.ll`

Nós geramos “arquivo.s”, que é código no assembly da nossa máquina.

Esse código pode ser compilado pelo GCC da mesma forma que compilamos códigos em C para gerar um executável.

O llvm não gerá binários, e sim assembly e requer outras ferramentas para transformá-los em binário.



LLVM

Para aprender mais sobre a IR, você pode usar o manual disponível:

<https://llvm.org/docs/LangRef.html>

Outra forma de estudar pode ser analisando o resultado da IR de códigos em C. Isso pode ser feito fazendo uma compilação com o clang

```
clang -S -emit-llvm code.c
```

Esse comando compila code.c até a parte de geração de IR, e gera uma ir com o nome code.ll

Cuidado: A IR do llvm é bem complexa. Comandos e variáveis podem possuir uma quantidade muito alta de atributos. Eles não são necessários para a execução do laboratório, mas ao extrair a IR de um código C você deve notar eles em abundância no resultado.



Lab4

Nesse lab você pode assumir que todos os programas passados para o seu compilador vão estar sintaticamente corretos.

Dessa vez, todo o programa recebido vai conter uma variável chamada `sm_main`.

O objetivo do laboratório é imprimir na tela o valor dessa variável.



Lab4

Para fazer isso será necessário:

- Gerar um arquivo contendo a IR (code.ll)
- Transformar esse arquivo em assembly (code.s)
- Criar um programa em C declara como extern a variável e as possíveis funções necessárias, e executa o necessário para imprimir o valor da variável.



Lab4

Dicas:

- Esse lab pode variar de simples a difícil dependendo da qualidade da sua gramática. Se você estiver tendo muita dificuldade, considere voltar e trabalhar na gramática.
- Operações de multiplicação e divisão tem prioridade às de soma e subtração.
- Operações de mesma prioridade devem ser resolvidas da esquerda para a direita.
- Crie alguma ferramenta para criar os nomes dos valores temporários dentro da função.
- Como o valor de algumas variáveis depende de funções, é uma boa ideia criar uma função especial para inicializar elas. Essa função deve ser chamada pelo “.c” antes de imprimir a variável.



Lab4 - Entrega

- Gramatica.g4
- MyParser.java
- MyVisitor.java (o visitor agora será responsável por gerar o código).
- Printer.c (código que vai imprimir o valor de sm_main)
- script.sh (script em bash que deve gerar o compilador em java assumindo que o jar do antlr está no diretório corrente, receber o código code.sm como parâmetro, fazer os passos necessários para gerar o binário junto com o printer.c e executar esse binário para imprimir o valor).

Os arquivos podem ter nomes arbitrários, menos o script, que deve se chamar script.sh