Projeto de Compilador
E5 de **Geração de Código**

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

1 Introdução

A quinta etapa do trabalho de implementação de um compilador para a Linguagem consiste na geração de código intermediário a partir da árvore sintática abstrata (AST). Utilizaremos como representação intermediária a Linguagem ILOC, descrita em detalhes no apêndice A de *Engineering a Compiler*, mas com o essencial no anexo deste documento. Testes poderão ser realizados através de um simulador simples escrito em Python3 e já disponível em anexo.

2 Funcionalidades Necessárias

2.1 Estrutura de Dados para ILOC

Implementa uma estrutura de dados para conter o código da operação e os argumentos necessários para cada operação. Note que os argumentos das operações são nomes de registradores, valores constantes, ou nomes de rótulos. Além disso, deve-se implementar uma estrutura de dados para manter uma lista de instruções ILOC. Utilize a implementação mais adequada ao uso em um compilador.

2.2 Rótulos e Registradores

Implemente uma função para fornecer nomes de rótulos a serem utilizados na geração de código. Os rótulos são utilizados para marcar os pontos de desvio no fluxo de execução. Implemente uma função para gerar nomes de registradores. Os nomes de rótulos e registradores devem seguir a convenção de nomes especificada na descrição da Linguagem ILOC. Os registradores podem ser vistos como variáveis temporárias auxiliares.

2.3 Geração de Código

Os programas a serem compilados devem conter pelo menos a função `main`. Ela deverá ser a primeira a ser executada. Existem pelo menos duas possibilidades de implementação válidas. **Em uma passagem**: realizar eventuais alterações gramaticais para realizar a geração de código em uma única passagem (juntamente com a análise léxica, sintática e semântica). **Em duas passagens**: Outra é optar por fazer o processo de geração de código em uma segunda passagem sobre a AST, através da implementação de uma função que faça o percorrimento. Informe o professor da sua escolha. Em ambos os casos, ao final da execução, o compilador deve ter na raiz da árvore AST um ponteiro para todo o código intermediário ILOC do programa de entrada. A geração de código consiste na criação de uma ou mais instruções ILOC juntamente com a união de trechos das subárvores, com a criação de novos símbolos intermediários e rótulos conforme necessário. Nesta etapa, deve-se traduzir as seguintes construções da linguagem, considerando que existe apenas o tipo `int`:

- Cálculo de endereço na declaração de variáveis
  - Endereço de variáveis locais são um deslocamento em relação ao registrador especial `rfp`
  - Endereço de variáveis globais são um deslocamento em relação ao registrador especial `rbss`
- Expressões aritméticas, com operações unárias, binárias e ternária
  - soma, subtração, divisão e multiplicação
- Expressões booleanas por controle de fluxo com curto-circuito
  - todos os operadores relacionais
  - todos os operadores lógicos (adote `&&` para o e lógico, `||` para o ou lógico)
- O comando de atribuição
- Os seguintes comandos de fluxo de controle
  - if com else opcional
  - while
  - for
- Chamada de função (com `jumpI` para um label)

Simplificações para a Geração de Código: Não é obrigatório implementar tradução de código para qualquer outro elemento da linguagem que tenha sido definido anteriormente (principalmente na E2), mas que não esteja listado acima. Nada precisa ser feito referente aos tipos diferentes de `int`.

2.4 Endereçamento de variáveis

Na tradução para ILOC, deve-se considerar que o conteúdo de cada variável da Linguagem está em um endereço de memória. Este endereço deve ser calculado no momento da declaração da variável considerando o escopo atual e seu endereço base. Antes de realizar qualquer operação sobre uma variável, deve-se antes de tudo carregar o seu conteúdo (a partir de um endereço de memória) para um registrador (utilizando a operação `load`, por exemplo), para só então realizar a operação sobre a variável. Ao final desta operação, o valor resultante estará obrigatoriamente em um registrador. Este valor final deve ser transferido para o endereço da variável na memória (utilizando a operação `store`, por exemplo) caso necessário.

A Instruções

As instruções em código intermediário servem para isolar as tarefas de geração da sequência básica de instruções dos detalhes e formato específicos de uma arquitetura alvo. Além disso, a geração usada nesse trabalho emprega técnicas genéricas de forma funcional,

didática, mas pode ser otimizada de várias formas antes da geração de código assembly. Dois exemplos de otimização são a reutilização de símbolos temporários em expressões e o uso de registradores. Entretanto, essas otimizações não fazem parte desta etapa do trabalho e são portanto opcionais.

B Geração de Código

A geração de código será feita de baixo para cima e da esquerda para a direita, na árvore. O modo mais simples de encadear novas instruções é representar os trechos de código como listas encadeadas invertidas, isto é, com um ponteiro para a última instrução de um trecho, e cada instrução apontando para a anterior. Ao final da geração, tu podes escrever uma função que percorre o código completo e inverte a lista de forma que se possa escrever o código na ordem em que deve ser executado. Isso pode não ser necessário dependendo da forma como se gera o código.

C Funções Auxiliares

Para a geração de código, além das rotinas utilitárias de TACs e da rotina recursiva principal que percorre a AST, utilize outras funções auxiliares. Isto tem dois motivos: primeiro, a semelhança na geração de código em vários nós da árvore, especialmente nas expressões aritméticas e relacionais. Segundo, evitar o tamanho da função de geração. Ela deve realizar um `switch(node->type)` e chamar a função auxiliar de geração de código apropriada para o tipo deste nó da AST.

D Entrada e Saída Padrão

Organize a sua solução para que o compilador leia o programa da entrada padrão e gere o programa em ILOC na saída padrão. Dessa forma, pode-se realizar o seguinte comando (considerando que `etapa5` é o binário do compilador):

```
./etapa5 < entrada > saida
```

Onde `entrada` contém um programa na linguagem sendo compilada, e `saida` contém em ILOC traduzido.

E Avaliação automática

Os testes automáticos utilizarão sistematicamente o simulador `ilocsim` (veja em anexo) para verificar se o código sintetizado pela solução do grupo realiza o que o programa na entrada diz que tem que ser feito. Vários mecanismos são empregados para isso, mas o principal é o estado da memória (valores que estão na pilha e segmento de dados) no final do programa.

F Arquivo `main.c`

Utilize uma função `main.c` semelhante aquela da E3. O grupo está autorizado a modificá-la para que a geração de código reflita as decisões de implementação do grupo. Não esqueça de liberar a memória corretamente.

G Linguagem ILOC

G.1 Introdução

ILOC é uma representação intermediária parecida com assembly para uma máquina RISC abstrata. A máquina abstrata que executa ILOC (veja o simulador abaixo escrito em Python) tem um número ilimitado de registradores. ILOC é um código de três endereços com operações de registrador a registrador, operações de carga (`load`) e armazenamento (`store`), comparações e desvios. Suporta apenas modos de endereçamento simples, tais como – direto, endereço + offset, endereço + imediato, e imediato. Os operandos são lidos no início do ciclo que uma operação começa a ser executada. Os operandos resultantes da operação se tornam definidos no final do ciclo no qual a operação se completa.

Nota: Este texto é uma tradução simplificada do apêndice A do livro do Keith com adaptação para a disciplina de compiladores na UFRGS.

G.2 Gramática da Linguagem ILOC

Um programa ILOC consiste em uma lista sequencial de instruções. Cada instrução pode ser precedida por um rótulo. Um rótulo é apenas uma cadeia de caracteres sendo separada da instrução por dois pontos. Por convenção, limita-se o formato dos rótulos com a expressão regular `[a-z]([a-z]|[0-9]|-)*`. Se alguma instrução precisa de mais de um rótulo, deve ser inserido uma instrução que contém apenas um operação `nop` antes dela, colocando o rótulo adicional na instrução `nop`. Um programa ILOC é definido mais formalmente:

ProgramaILOC	→	ListaInstrucoes
ListaInstrucoes	→	Instrucao | `label:` Instrucao | Instrucao ListaInstrucoes

Cada instrução pode conter uma ou mais operações. Uma instrução com uma única operação é escrita em uma linha própria, enquanto que uma instrução com múltiplas operações pode ser escrita em várias linhas. Para agrupar operações em uma instrução única, nós envolvemos a lista de operações entre colchetes e separamos cada operações com ponto e vírgulas. Mais formalmente:

Instrucao	→	Operacao | `[` ListaOperacoes `]`
ListaOperacoes	→	Operacao | Operacao `;` ListaOperacoes

Uma operação ILOC corresponde a uma instrução em nível de máquina que pode ser executada por uma única unidade funcional em um único ciclo. Ela tem um código de operação (`opcode`), uma sequência de operandos fontes separados por vírgulas, e uma sequência de operandos alvo separados também por vírgulas. Os operandos fonte são separados dos operandos alvo pelo símbolo "=>", que significa "em". Formalmente:

Operacao	→	OperacaoNormal | OperacaoFluxoControle
OperacaoNormal	→	CodigoOperacao ListaOperandos => ListaOperandos
ListaOperandos	→	Operando | Operando `,` ListaOperandos
Operando	→	`registrador` | `numero` | `rotulo`

O não-terminal *CodigoOperacao* pode ser qualquer operação ILOC, exceto `cbr`, `jump`, e `jumpI`. As tabelas do sumário abaixo mostram o número de operandos e seus tipos para cada operação da Linguagem ILOC.

Um *Operando* pode ser um de três tipos: `registrador`, `numero` e `rotulo`. O tipo de cada operando é determinado pelo código da operação e a posição que o operando aparece na operação. Por convenção, os registradores começam pela letra `r` (minúscula) e são seguidos por um número inteiro ou uma cadeia de caracteres qualquer. Ainda por convenção, rótulos sempre começam pela letra `L` (maiúscula).

A maioria das operações tem um único operando alvo; algumas operações de armazenamento (*store*) tem operandos alvos múltiplos, assim como saltos. Por exemplo, `storeAI` tem um único operando fonte e dois operandos alvo. A fonte deve ser um registrador, e os alvos devem ser um registrador e uma constante imediata. Então, a operação da linguagem ILOC:

```
storeAI ri => rj,4
```

calcula o endereço adicionando `4` ao conteúdo de `rj` e armazena o valor encontrado no registrador `ri` na localização da memória especificada pelo endereço calculado. Em outras palavras:

Memória (`rj + 4`) ← *Conteúdo* (`ri`)

Operações de fluxo de controle tem sintática diferente. Uma vez que estas operações não definem seus alvos, elas são escritas com uma flecha simples `->` ao invés da flecha dupla `=>`. Formalmente:

```
OperacaoFluxoControle  →  cbr register -> label, label
                       |  jumpI -> label
                       |  jumpI -> register
```

A primeira operação, `cbr`, implementa um desvio condicional. As outras duas operações são desvios incondicionais.

G.3 Convenções de Nome

O código ILOC usa um conjunto simples de convenções de nome.

1. Existe um número ilimitado de registradores. Eles são nomeados com um `r` seguido de um número inteiro positivo.
2. Existe um número ilimitado de rótulos. Eles são nomeados com um `L` seguido de um número inteiro positivo.
3. O registrador `rfp` é reservado como um ponteiro para a base do registro de ativação atual (o registro do topo da pilha).
4. O registrador `rsp` é reservado como um ponteiro para o topo da pilha.
5. O registrador `rbss` é reservado para apontar para a base do segmento de dados do programa.
6. O registrador `rpc` é reservado para manter o contador do programa (*program counter*).
7. Comentários em ILOC começam com `//` e continuam até o final da linha.

G.4 Operações Individuais

G.4.1 Aritmética

A Linguagem ILOC tem operações de três endereços de registrador para registrador. Todas estas operações realizam a leitura dos operandos origem de registradores ou constantes e escrevem o resultado de volta para um registrador. Qualquer registrador pode servir como um operando origem ou destino.

```
add     r1, r2  =>  r3    // r3 = r1 + r2
sub     r1, r2  =>  r3    // r3 = r1 - r2
mult    r1, r2  =>  r3    // r3 = r1 * r2
div     r1, r2  =>  r3    // r3 = r1 / r2
addI    r1, c2  =>  r3    // r3 = r1 + c2
subI    r1, c2  =>  r3    // r3 = r1 - c2
rsubI   r1, c2  =>  r3    // r3 = c2 - r1
multI   r1, c2  =>  r3    // r3 = r1 * c2
divI    r1, c2  =>  r3    // r3 = r1 / c2
rdivI   r1, c2  =>  r3    // r3 = c2 / r1
```

As primeiras quatro operações da tabela são operações registrador para registrador clássicas. As próximas seis especificam um operando imediato. As operações não comutativas, `sub` e `div`, tem duas formas imediatas alternativas para permitir o operando imediato em qualquer lado do operador. As formas imediatas são úteis para expressar resultados de certas otimizações, para escrever exemplos de forma mais concisa, e para registrar jeitos óbvios de reduzir a demanda por registradores.

G.4.2 Shifts

ILOC suporta um conjunto de operações aritméticas de *shift*, para a esquerda e para a direita, em ambas as formas, com registradores e imediata.

```
lshift   r1, r2  =>  r3    // r3 = r1 << r2
lshiftI  r1, c2  =>  r3    // r3 = r1 << c2
rshift   r1, r2  =>  r3    // r3 = r1 >> r2
rshiftI  r1, c2  =>  r3    // r3 = r1 >> c2
```

G.4.3 Operações sobre a Memória

ILOC suporta um conjunto de operadores de carga e armazenamento para mover valores entre a memória e registradores. As operações `load` e `cload` movem dados da memória para os registradores.

```
load    r1       =>   r2    // r2 = Memoria(r1)
loadAI  r1, c2   =>   r3    // r3 = Memoria(r1 + c2)
loadA0  r1, r2   =>   r3    // r3 = Memoria(r1 + r2)
cload   r1       =>   r2    // caractere load
cloadAI r1, c2   =>   r3    // caractere loadAI
cloadA0 r1, r2   =>   r3    // caractere loadA0
```

As operações diferem nos modos de endereçamento que elas suportam. As operações `load` e `cload` assumem um endereço direto na forma de um único operando registrador. As operações `loadAI` e `cloadAI` adicionam um valor imediato ao conteúdo do registrador para formar um endereço imediatamente antes de realizar a carga. Nós chamamos estas de operações de *endereçamento imediato*. As operações `loadA0` e `cloadA0` adicionam o conteúdo de dois registradores para calcular o endereço efetivo imediatamente antes de realizar a carga. Estas operações são chamadas de *endereçamento por deslocamento*.

Uma outra forma de carga que a Linguagem ILOC suporta é uma operação `loadI` de carga imediata. Ela recebe um inteiro como argumento e coloca este inteiro dentro do registrador alvo.

```
loadI c1 => r2 // r2 = c1
```

As operações de armazenamento são semelhantes, conforme a tabela abaixo.

```
store    r1 => r2        // Memoria(r2) = r1
storeAI  r1 => r2, c3    // Memoria(r2 + c3) = r1
storeAO  r1 => r2, r3    // Memoria(r2 + r3) = r1
cstore   r1 => r2        // caractere store
cstoreAI r1 => r2, c3    // caractere storeAI
cstoreAO r1 => r2, r3    // caractere storeAO
```

Não há nenhuma operação de armazenamento imediato.

G.4.4 Operações de Cópia entre Registradores

A Linguagem ILOC tem um conjunto de operações para mover valores entre registradores, sem passar pela memória. Na listagem abaixo, as primeiras duas operações, `i2i` e `c2c`, copiam um valor de um registrador para outro, sem conversão. As duas últimas operações realizam conversões considerando a codificação de caracteres ASCII.

```
i2i r1 => r2 // r2 = r1 para inteiros
c2c r1 => r2 // r2 = r1 para caracteres
c2i r1 => r2 // converte um caractere para um inteiro
i2c r1 => r2 // converte um inteiro para caractere
```

G.5 Operações de Fluxo de Controle

G.5.1 Visão Geral

Em geral, operações de comparação na Linguagem ILOC recebem dois valores e retornam um valor booleano. A operação condicional `cbr` recebe um booleano como argumento e transfere o controle para um de dois rótulos alvo. Os dois rótulos alvo não precisam estar definidos previamente (pode-se saltar para um código mais a frente do programa).

```
cmp_LT r1, r2 -> r3        // r3 = true se r1 < r2, senão r3 = false
cmp_LE r1, r2 -> r3        // r3 = true se r1 <= r2, senão r3 = false
cmp_EQ r1, r2 -> r3        // r3 = true se r1 = r2, senão r3 = false
cmp_GE r1, r2 -> r3        // r3 = true se r1 >= r2, senão r3 = false
cmp_GT r1, r2 -> r3        // r3 = true se r1 > r2, senão r3 = false
cmp_NE r1, r2 -> r3        // r3 = true se r1 != r2, senão r3 = false
cbr     r1    -> l2, l3    // PC = endereço(l2) se r1 = true, senão PC = endereço(l3)
```

G.5.2 Saltos

A Linguagem ILOC tem duas formas de operações de salto. A primeira é um salto incondicional e imediato que transfere o controle para um a primeira instrução após um rótulo. A segunda recebe um registrador como argumento. O conteúdo do registrador é interpretado como um endereço de código, transferindo o controle incondicionalmente e imediatamente para este endereço.

```
jumpI -> l1 // PC = endereço(l1)
jump  -> r1 // PC = r1
```

G.6 Sumário de Operações ILOC

G.6.1 Sumários de Operações ILOC Individuais

```
nop                          // não faz nada
add      r1, r2   =>  r3     // r3 = r1 + r2
sub      r1, r2   =>  r3     // r3 = r1 - r2
mult     r1, r2   =>  r3     // r3 = r1 * r2
div      r1, r2   =>  r3     // r3 = r1 / r2
addI     r1, c2   =>  r3     // r3 = r1 + c2
subI     r1, c2   =>  r3     // r3 = r1 - c2
rsubI    r1, c2   =>  r3     // r3 = c2 - r1
multI    r1, c2   =>  r3     // r3 = r1 * c2
divI     r1, c2   =>  r3     // r3 = r1 / c2
rdivI    r1, c2   =>  r3     // r3 = c2 / r1
lshift   r1, r2   =>  r3     // r3 = r1 << r2
lshiftI  r1, c2   =>  r3     // r3 = r1 << c2
rshift   r1, r2   =>  r3     // r3 = r1 >> r2
rshiftI  r1, c2   =>  r3     // r3 = r1 >> c2
and      r1, r2   =>  r3     // r3 = r1 && r2
andI     r1, c2   =>  r3     // r3 = r1 && c2
or       r1, r2   =>  r3     // r3 = r1 || r2
orI      r1, c2   =>  r3     // r3 = r1 || c2
xor      r1, r2   =>  r3     // r3 = r1 xor r2
xorI     r1, c2   =>  r3     // r3 = r1 xor c2
loadI    c1       =>  r2     // r2 = c1
load     r1       =>  r2     // r2 = Memoria(r1)
loadAI   r1, c2   =>  r3     // r3 = Memoria(r1 + c2)
loadA0   r1, r2   =>  r3     // r3 = Memoria(r1 + r2)
cload    r1       =>  r2     // caractere load
cloadAI  r1, c2   =>  r3     // caractere loadAI
cloadA0  r1, r2   =>  r3     // caractere loadA0
store    r1       =>  r2     // Memoria(r2) = r1
storeAI  r1       =>  r2, c3 // Memoria(r2 + c3) = r1
storeAO  r1       =>  r2, r3 // Memoria(r2 + r3) = r1
cstore   r1       =>  r2     // caractere store
cstoreAI r1       =>  r2, c3 // caractere storeAI
cstoreAO r1       =>  r2, r3 // caractere storeAO
i2i      r1       =>  r2     // r2 = r1 para inteiros
c2c      r1       =>  r2     // r2 = r1 para caracteres
c2i      r1       =>  r2     // converte um caractere para um inteiro
i2c      r1       =>  r2     // converte um inteiro para caractere
```

G.6.2 Sumários de Operações ILOC de Fluxo de Controle

```
jumpI           ->  l1        // PC = endereço(l1)
jump            ->  r1        // PC = r1
cbr    r1       ->  l2, l3    // PC = endereço(l2) se r1 = true, senão PC = endereço(l3)
cmp_LT r1, r2   ->  r3        // r3 = true se r1 < r2, senão r3 = false
```

```
cmp_LE r1, r2  ->  r3        // r3 = true se r1 \leq r2, senão r3 = false
cmp_EQ r1, r2  ->  r3        // r3 = true se r1 = r2, senão r3 = false
cmp_GE r1, r2  ->  r3        // r3 = true se r1 \geq r2, senão r3 = false
cmp_GT r1, r2  ->  r3        // r3 = true se r1 > r2, senão r3 = false
cmp_NE r1, r2  ->  r3        // r3 = true se r1 \ne r2, senão r3 = false
```