Projeto de Compilador: **Etapa 5 de Geração de Código**

Lucas Mello Schnorr

schnorr@inf.ufrgs.br

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 [1], mas com o essencial na Seção 4 deste documento. Testes poderão ser realizados através de um simulador simples escrito em Python3 e já disponível no repositório git de referência (em scripts, etapa 5, `ilocsim`).

1 Funcionalidades Necessárias

1.1 Estrutura de Dados para ILOC

Implementa uma estrutura de dados para conter o código da operação e os argumentos necessários para cada operação (consulte a Seção 4.5 para saber quais são as operações válidas em ILOC e seus argumentos). Note que os argumentos das operações são nomes de registradores, valores constantes, ou nomes de rótulos. Além disso, implementa uma estrutura de dados para manter uma lista de instruções ILOC. Dentre as várias forma de implementar, utilize aquela que mais se adequa ao uso em um compilador.

1.2 Rótulos e Registradores

Uma função deve ser adicionada ao projeto de compilador para fornecer nomes de rótulos a serem utilizados na geração de código. Os nomes são utilizados para marcar os pontos de desvio no fluxo de execução. Todas as instruções ILOC devem ser realizadas sobre valores que estão em registradores. Sendo assim, uma função deve ser adicionada ao projeto de compilador para gerenciar a criação de nomes de registradores. Os nomes de rótulos e registradores devem seguir a convenção de nomes especificada na Seção 4. Os registradores podem ser vistos como variáveis temporárias auxiliares.

1.3 Geração de Código

Existem pelo menos duas possibilidades de implementação válidas:

1. **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).
2. **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 sub-árvores, com a criação de novos símbolos intermediários e rótulos conforme necessário. Nesta etapa, deve-se traduzir as seguintes construções da linguagem:

- Cálculo de endereço na declaração de variáveis
  - Endereço de variáveis locais são um deslocamento em relação ao registrador especial `fp` (ou `rarp`)

  - Endereço de variáveis globais são um deslocamento em relação ao registrador especial `rbss`

- Expressões Aritméticas
- Expressões Booleanas com curto-circuito
- Arranjos Multidimensionais
- Comandos de Atribuição
- Comandos de Fluxo de Controle (a implementação do `switch` é opcional, mas encorajada)

Por simplicidade, os programas deverão conter apenas uma função chamada `main`.

2 Discussão Importante

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).

3 Dicas de Desenvolvimento

Abaixo listam-se algumas dicas de desenvolvimento que podem ser úteis.

3.1 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.

3.2 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, escreva 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.

3.3 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.

3.4 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 `main` é o binário do compilador):

```
./main < entrada.poa > saida.iloc
```

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

Referências

[1] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Morgan Kaufmann, 2nd edition, 2012.

4 A Linguagem ILOC

ILOC[1] é uma representação intermediária parecida com assembly para uma máquina RISC abstrata. A máquina abstrata que executa ILOC 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 quem uma operação começa a ser executada. Os operandos resultantes da operação se tornam definidos no final do ciclo no qual a operação se completa.

4.1 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`

[1] Este texto é uma tradução simplificada do apêndice A do livro do Keith [1].

O não-terminal CodigoOperacao pode ser qualquer operação ILOC, exceto `cbr`, `jump`, e `jumpI`. As tabelas na seção˜4.5 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.

4.2 Convenções de Nome

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

1. Deslocamentos de memória para variáveis são representados simbolicamente com um `@` antes do nome da variável.
2. Existe um número ilimitado de registradores. Estes são nomeados com inteiros simples, como `r1789`, ou com nomes simbólicos, como em `ri` ou `rj`.
3. O registrador `fp` ou `rarp` é reservado como um ponteiro para o registro de ativação atual. Sendo assim, a operação:

   ```
   loadAI rarp,@x => r1
   ```

   carrega o conteúdo da variável `x`, guardada no deslocamento `@x` a partir do `rarp`, em `r1`.

Comentários em ILOC começam com `//` e continuam até o final da linha.

4.3 Operações Individuais

4.3.1 Aritmética

A Linguagem ILOC tem operações de três endereços de registrador para registrador.

Opcode	Fonte		Alvo	Significado
`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`	$r3 = c2/r1$

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.

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.

4.3.2 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.

Opcode	Fonte		Alvo	Significado
`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`	$r3 = r1 >> c2$

4.3.3 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.

Opcode	Fonte		Alvo	Significado
`load`	`r1`	`=>`	`r2`	$r2 = \text{MEMORIA}(r1)$
`loadAI`	`r1, c2`	`=>`	`r3`	$r3 = \text{MEMORIA}(r1 + c2)$
`loadA0`	`r1, r2`	`=>`	`r3`	$r3 = \text{MEMORIA}(r1 + r2)$
`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.

Opcode	Fonte		Alvo	Significado
`loadI`	`c1`	`=>`	`r2`	$r2 = c1$

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

Opcode	Fonte		Alvo	Significado
`store`	`r1`	`=>`	`r2`	$\text{MEMORIA}(r2) = r1$
`storeAI`	`r1`	`=>`	`r2, c3`	$\text{MEMORIA}(r2 + c3) = r1$
`storeA0`	`r1`	`=>`	`r2, r3`	$\text{MEMORIA}(r2 + r3) = r1$
`cstore`	`r1`	`=>`	`r2`	caractere `store`
`cstoreAI`	`r1`	`=>`	`r2, c3`	caractere `storeAI`
`cstoreA0`	`r1`	`=>`	`r2, r3`	caractere `storeA0`

Não há nenhuma operação de armazenamento imediato.

4.3.4 Operações de Cópia entre Registradores

A Linguagem ILOC tem um conjunto de operações para mover valores entre registradores, sem passar pela memória.

Opcode	Fonte		Alvo	Significado
`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

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.

4.4 Operações de Fluxo de Controle

Em geral, operações de comparação na Linguagem ILOC recebem dois valores e retornam um valor booleano.

Opcode	Fonte		Alvo	Significado
`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 $r1 \neq r2$, senão $r3 = false$
`cbr`	`r1`	`->`	`l2, l3`	$PC = l2$ se $r1 = true$, senão $PC = l3$

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).

4.4.1 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. {Esta segunda forma deve ser evitada por ser ambígua.} Mais detalhes a respeito disto na referência oficial [1].

Opcode	Fonte		Alvo	Significado
`jumpI`		`->`	`l1`	$PC = l1$
`jump`		`->`	`r1`	$PC = r1$

4.5 Sumário de Operações ILOC

4.5.1 Sumários de Operações ILOC Individuais

Opcode	Fonte		Alvo	Significado
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	$r3 = r1 \wedge r2$
andI	r1, c2	=>	r3	$r3 = r1 \wedge c2$
or	r1, r2	=>	r3	$r3 = r1 \vee r2$
orI	r1, c2	=>	r3	$r3 = r1 \vee c2$
xor	r1, r2	=>	r3	$r3 = r1$ xor $r2$
xorI	r1, c2	=>	r3	$r3 = r1$ xor $c2$
loadI	c1	=>	r2	$r2 = c1$
load	r1	=>	r2	$r2 = \text{MEMORIA}(r1)$
loadAI	r1, c2	=>	r3	$r3 = \text{MEMORIA}(r1 + c2)$
loadA0	r1, r2	=>	r3	$r3 = \text{MEMORIA}(r1 + r2)$
cload	r1	=>	r2	caractere `load`
cloadAI	r1, c2	=>	r3	caractere `loadAI`
cloadA0	r1, r2	=>	r3	caractere `loadA0`
store	r1	=>	r2	$\text{MEMORIA}(r2) = r1$
storeAI	r1	=>	r2, c3	$\text{MEMORIA}(r2 + c3) = r1$
storeA0	r1	=>	r2, r3	$\text{MEMORIA}(r2 + r3) = r1$
cstore	r1	=>	r2	caractere `store`
cstoreAI	r1	=>	r2, c3	caractere `storeAI`
cstoreA0	r1	=>	r2, r3	caractere `storeA0`
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

4.5.2 Sumários de Operações ILOC de Fluxo de Controle

Opcode	Fonte		Alvo	Significado
`jumpI`		->	`l1`	$PC = l1$
`jump`		->	`r1`	$PC = r1$
`cbr`	`r1`	->	`l2, l3`	$PC = l2$ se $r1 = true$, senão $PC = l3$
`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 $r1 \neq r2$, senão $r3 = false$