Aula 05 – Representação Intermediária

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- Uma estrutura de dados central em um compilador é a forma intermediária do programa sob compilação.
- A maioria dos módulos do compilador leem e manipulam essa estrutura.
- **Estes *slides***: apresentar uma visão geral das formas intermediárias de código usadas em compiladores.
- **Objetivos**: mostrar como as diferentes formas afetam o projeto de um compilador.

Referências

Section 8.1 – Intermediate Code and Data Structures
K. C. Louden
Chapter 5 – Intermediate Representations
K. D. Cooper
Chapters 6 & 8
D. Thain

Introdução

- Compiladores modernos são multi-passadas.
- Cada passada do compilador deriva novas informações que devem ser transmitidas para a próxima etapa.
- Essas informações são armazenadas na representação intermediária (*intermediate representation – IR*) do código de entrada.
- Um compilador pode usar uma única IR ou empregar uma série de IRs diferentes ao longo do processo de tradução.
- Segundo caso é mais comum, pois diferentes IRs são mais adequadas para diferentes etapas.
- De qualquer forma, a IR do programa de entrada é sempre tratada como a forma definitiva do programa.
- Somente o *scanner* manipula o arquivo texto do código fonte, as demais fases do compilador trabalham diretamente sobre a IR.

Tipos de Representações Intermediárias

- IRs podem ser categorizadas segundo a estrutura em:
  - ***Graph-based***: codificam o programa fonte em um grafo.
  - **Lineares**: "*assembly*" para uma máquina abstrata.
  - **Híbridas**: combinação de elementos de ambas.
- IRs baseadas em grafos podem ser de variados tipos: *parse trees*, ASTs, DAGs, grafos de dependências, etc.
- IRs lineares mais comuns: código de um, dois ou três endereços (*one-, two-, or three-address code*).
- O formato ILOC usado no livro do Cooper é uma IR linear, um exemplo de código de três endereços.
- Diferentes módulos do compilador utilizam diferentes IRs.
- Para módulos mais "próximos" do código fonte, estruturas em árvores são mais simples de manipular.
- Em geração de código, IRs lineares são mais adequadas.
- Código Intermediário (IC) é outro nome para uma IR linear.

Representações Baseadas em Grafos

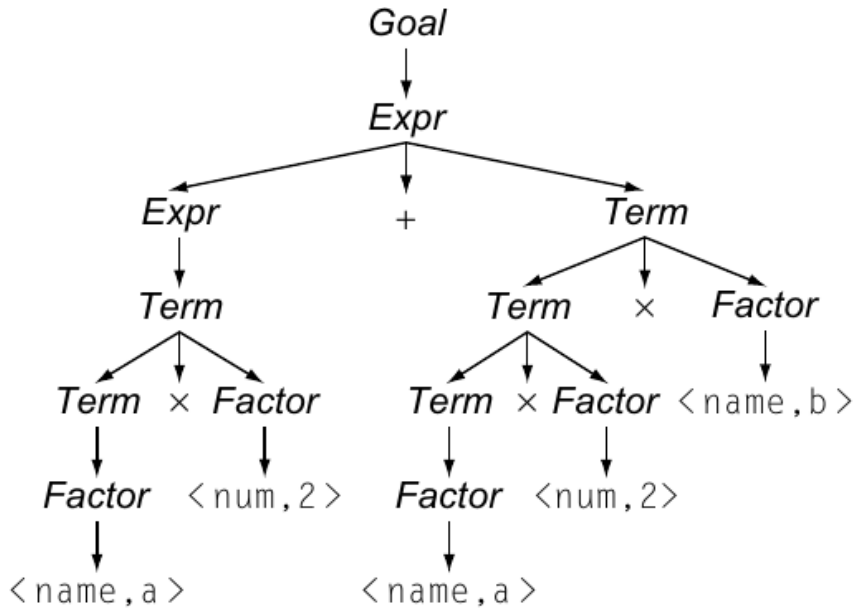
- Muitos compiladores usam IRs que representam o programa de entrada como um grafo.
- Todas essas IRs são formadas por nós e arestas, mas o significado destes elementos depende do tipo da IR.
- *Parse trees* e ASTs são exemplos de IRs baseados em grafos aonde a estrutura da árvore captura a estrutura do código de entrada.
- Outras IRs dessa categoria incluem:
  - *DAG – Directed Acyclic Graph*: variante da AST que evita repetição de estruturas.
  - *CFG – Control-Flow Graph*: estrutura que modela o fluxo de controle de execução do programa.
  - *Dependence Graph*: indica dependências entre dados do compilador.
  - *Call Graph*: sinaliza a transferência de execução entre as funções do programa.
- Vamos apresentar todas essas IRs a seguir.

Parse Trees

- Como já estudado, a *parse tree* (árvore de derivação ou árvore de sintaxe concreta) representa o processo de derivação (*parse*) da entrada.
- Exemplo de uma *parse tree* para a expressão $a \times 2 + a \times 2 \times b$ segundo a gramática de expressões clássica.

Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* - *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	num
	|	name

(a) Classic Expression Grammar



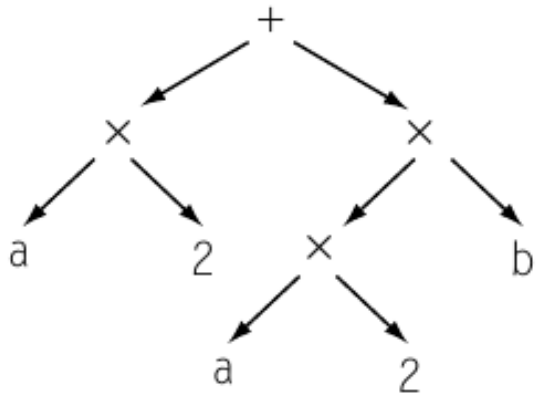
(b) Parse Tree for a×2+a×2×b

Parse Trees

- A *parse tree* é uma estrutura relativamente grande porque representa a derivação completa da entrada segundo a gramática.
- Uma vez que o compilador precisa alocar memória para todos os nós e arestas da árvore, é razoável buscar formas de compactar a *parse tree*.
- Compactação também ajuda no tempo de caminhamento pela árvore, o que diminui o tempo de compilação.
- Método usual é simplesmente abstrair os nós da *parse tree* que não têm utilidade para o restante do compilador.
- Leva a uma versão simplificada da *parse tree*, chamada *abstract syntax tree* (árvore de sintaxe abstrata).
- *Parse trees* são a forma de IR principal em discussões sobre *parsing* e gramáticas de atributos.
- ANTLR constrói explicitamente a *parse tree*, o Bison não.

Abstract Syntax Trees – ASTs

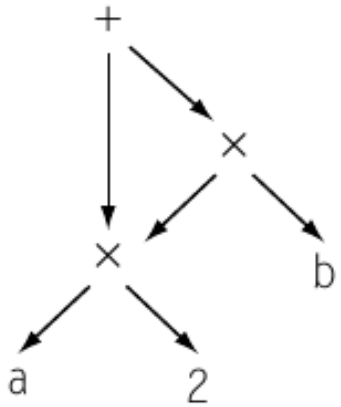
- Uma AST é um contração da *parse tree* que omite a maioria dos nós para os símbolos não terminais.
- Veja a AST para a expressão $a \times 2 + a \times 2 \times b$ abaixo.
- O significado e precedências da expressão permanecem, mas a maioria dos nós internos da *parse tree* somem.



- A AST é uma representação próxima do código fonte.
- Amplamente utilizada em compiladores e interpretadores.
- Devido à sua correspondência com a *parse tree*, o *parser* pode construir a AST durante o processo de análise sintática. (Veja a tarefa do Laboratório 05.)

Directed Acyclic Graphs – DAGs

- Um DAG é uma AST com compartilhamento. Subárvores idênticas são instanciadas uma única vez, ficando com múltiplos nós pais.
- Veja como o DAG para a expressão de exemplo evita a duplicação para a subárvore de $a \times 2$.



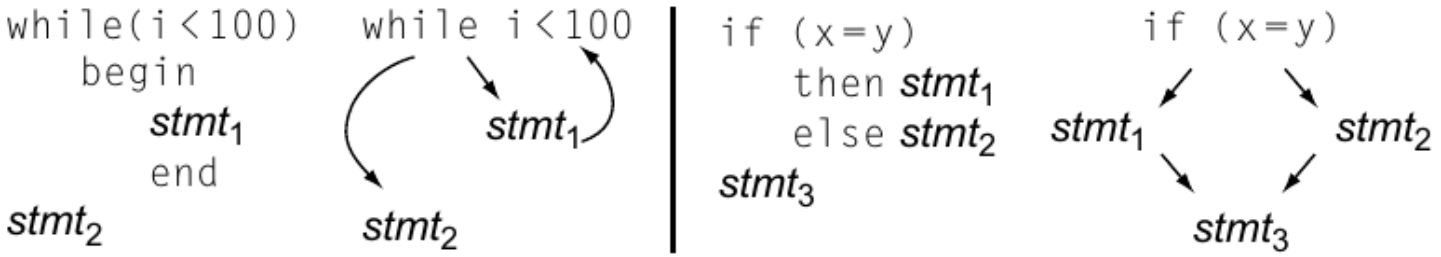
- DAGs são usados em compiladores para expor redundâncias e gerar código mais eficiente.
- O compartilhamento de $a \times 2$ indica que o compilador pode avaliar a expressão uma única vez e utilizar o resultado duas vezes.

Control-Flow Graphs – CFGs

- A unidade mais simples do fluxo de controle (*control-flow*) em um programa é um bloco básico (*basic block*), uma sequência de operações sem saltos (*branching*).
- As operações em um bloco básico sempre executam de forma sequencial.
- O controle de execução sempre entra em um bloco básico na primeira operação e sai pela última.
- Um grafo de fluxo de controle (CFG) tem um nó para cada bloco básico e uma aresta para cada possível transferência de controle entre blocos.
- O acrônimo CFG é sobrecarregado, podendo representar também *context-free grammar*. O significado deve ficar claro pelo contexto.

Control-Flow Graphs – CFGs

- Em IRs orientadas pela sintaxe, como em uma AST, as arestas indicam estrutura gramatical.
- O CFG difere dessas IRs porque as arestas representam possíveis caminhos de execução do programa de entrada.
- Veja exemplos de CFGs para os comandos `if` e `while`.

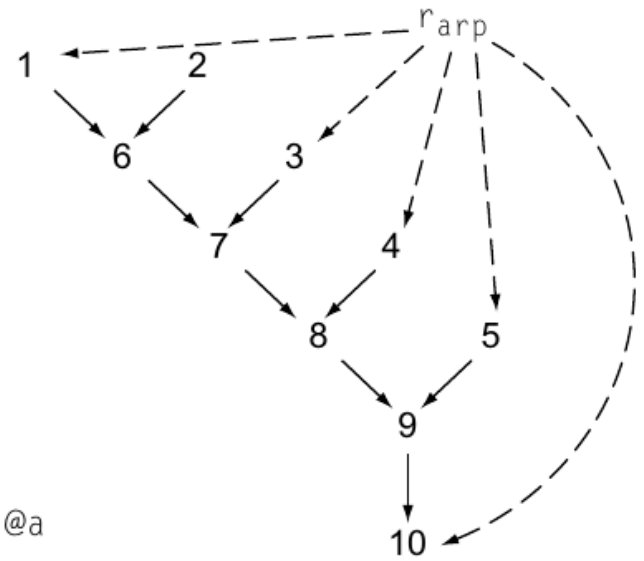


- A aresta de $stmt_1$ para o teste do `while` cria um ciclo; a AST deste fragmento seria acíclica.
- O CFG do `if` mostra que o controle sempre flui de $stmt_1$ ou $stmt_2$ para $stmt_3$. Na AST, essa conexão fica implícita.
- CFGs normalmente são usados com outras IRs.
- Operações dentro dos nós de um CFG podem usar uma IR linear. Essa combinação é uma IR híbrida.

Dependence Graphs

- Um grafo de dependência de dados modela o fluxo de valores das suas definições até o seu uso em um fragmento de código.
- Nós em um grafo de dependências representam operações. Arestas conectam os nós de definição e uso de um valor, nessa ordem.
- O exemplo abaixo mostra um bloco básico em ILOC e seu grafo de dependências.

```
1   loadAI   r_arp, @a ⇒ r_a
2   loadI    2         ⇒ r_2
3   loadAI   r_arp, @b ⇒ r_b
4   loadAI   r_arp, @c ⇒ r_c
5   loadAI   r_arp, @d ⇒ r_d
6   mult     r_a, r_2  ⇒ r_a
7   mult     r_a, r_b  ⇒ r_a
8   mult     r_a, r_c  ⇒ r_a
9   mult     r_a, r_d  ⇒ r_a
10  storeAI  r_a       ⇒ r_arp, @a
```



Dependence Graphs

- O grafo do exemplo anterior tem um nó para cada comando do bloco.
- Cada aresta indica o fluxo de um único valor.
- Por exemplo, a aresta de 3 para 7 reflete a definição de r_b no comando 3 e seu uso subsequente no comando 7.
- O valor em r_{arp} foi definido fora do bloco, por isso as arestas aparecem tracejadas.
- As arestas do grafo de dependências indicam restrições na ordem de execução das operações.
- Note que mais de uma sequência de execução pode satisfazer essas restrições.
- Essa liberdade de escolha é útil em arquiteturas com execução "*out-of-order*".
- $\Rightarrow$ Grafos de dependências têm um papel central no escalonamento de instruções. (Veja a Aula 08.)

Call Graphs

- Um grafo de chamadas (*call graph*) representa as relações de chamadas entre as funções de um programa.
- Um *call graph* tem um nó para cada função e uma aresta para cada chamada.
- Esse tipo de grafo é muito utilizado para análise e otimização interprocedural.
- Intraprocedural: qualquer análise que olha uma função de cada vez.
- Interprocedural: análise que examina as interações entre funções.
- A construção de *call graphs* é uma operação razoavelmente complexa, complicada ainda mais por compilação separada e linguagens OO.
- Veja a Seção 9.4 do livro do Cooper para mais informações.

IRs Lineares

- IRs lineares são a alternativas às IRs baseadas em grafos.
- Programas em *assembly* são uma forma de IR linear.
- As IRs lineares utilizadas em compiladores lembram código *assembly* para uma máquina abstrata.
- IRs lineares devem incluir um mecanismo para transferência de controle entre pontos do programa.
- Códigos lineares geralmente incluem comandos de saltos condicionais (*branches*) e incondicionais (*jumps*).
- Fluxo de controle demarca os blocos básicos em uma IR linear; blocos terminam em saltos ou antes de *labels*.

IRs Lineares

- Vários tipos de IRs lineares são usados em compiladores.
  - *One-address code* (código de um endereço): modela o comportamento de máquinas baseadas em pilha.
  - *Two-address code* (código de dois endereços): modela máquinas que possuem operações destrutivas.
  - *Three-address code* (TAC) (código de três endereços): modela máquinas aonde as operações recebem até dois operandos e produzem um resultado.
- *One-address code* é compacto e simples de gerar. É o formato do Java *bytecode*.
- *Two-address code* é pouco utilizado atualmente. No passado era útil por usar menos memória.
- TAC é a forma de IR linear mais comum atualmente. Ficou popular com a ascensão das arquiteturas RISC, pois lembra uma máquina RISC simples.
- TAC também é usado para arquiteturas CISC porque pode modelar operações destrutivas de forma explícita.

IRs Lineares

Considere a expressão `a-2*b`:

Código de um endereço (*one-address code*):

```
push 2
push b
mult
push a
sub
```

Código de dois endereços (*two-address code*):

```
load r1, 2
load r2, b
mult r2, r1
load r1, a
sub  r1, r2
```

Código de três endereços (*three-address code*):

```
load r1, 2
load r2, b
mult r3, r1, r2
load r4, a
sub  r5, r4, r3
```

- Código de um endereço opera sobre uma pilha. Comandos retiram operandos da pilha e empilham de volta o resultado.
- Código de dois endereços destrói o valor original do primeiro operando para guardar o resultado.
- Código de três endereços é mais geral que o de dois.

Código de Três Endereços

- Em código de três endereços, a maioria das operações tem a forma `i = j op k`, onde um operador `op` recebe dois operandos `i` e `k` e produz o resultado em `i`.
- Notação alternativa: `i` $\leftarrow$ `j op k`.
- *Exemplo*: o código do slide anterior nesta notação fica como a seguir:

```
t1 = 2
t2 = b
t3 = t1 * t2
t4 = a
t5 = t4 - t3
```

- Os nomes `t[1-5]` são ditos temporários.
- A tarefa de decidir aonde cada temporário fica armazenado (registrador ou memória) chama-se alocação de registradores. (Veja a Aula 08.)

Código de Três Endereços

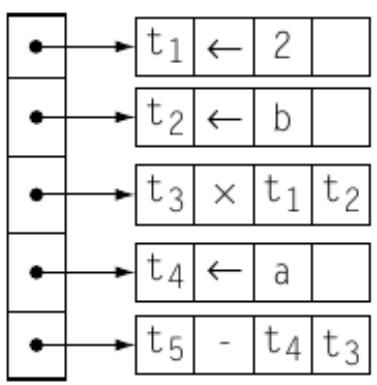
- Algumas operações em TAC como saltos incondicionais não usam todos os argumentos.
- De qualquer forma, TAC é bastante compacto.
  - Operações usam 1 ou 2 *bytes*.
  - Nomes são representados por índices ou ponteiros da tabela de símbolos.
  - Em geral, 4 *bytes* são suficientes para um comando TAC.
- Os nomes temporários em TAC facilitam as diferentes etapas de otimização de código.
- Além disso, o nível de abstração dos comandos em TAC pode variar bastante.
- Por exemplo, um compilador pode usar comandos como `max` e `min` em TAC porque isso torna essas operações mais simples de analisar e otimizar.

Estruturas para Código de Três Endereços

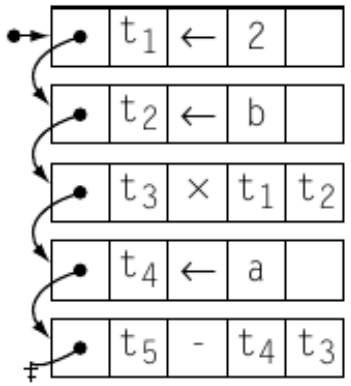
- A estrutura de dados mais comum para armazenar código de três endereços é a quádrupla: um campo para a operação e três para os endereços.
- Para formar blocos de código, o compilador precisa de um mecanismo para conectar as quádruplas.
- Existem variadas possibilidades de implementação. Veja algumas delas para o comando `a-2*b`.

Target	Op	Arg_1	Arg_2
t_1	←	2	
t_2	←	b	
t_3	×	t_1	t_2
t_4	←	a	
t_5	-	t_4	t_3

(a) Simple Array



(b) Array of Pointers



(c) Linked List

- Qual estrutura é mais adequada varia conforme o uso. Por exemplo, caso (c) facilita inserção de novos comandos, mas exige um acesso linear ao bloco.

IRs em Compiladores Reais

- Compiladores FORTRAN antigos utilizavam uma combinação de quádruplas e CFGs. A IR linear ficava armazenada em um vetor.

- Por muitos anos o GCC utilizou uma IR bem baixo-nível, chamada RTL (*register transfer language*). Exemplo:

```
(set (reg:SI 140)
     (plus:SI (reg:SI 138)
              (reg:SI 139)))
```

- O código acima é equivalente ao comando TAC `t140 <- t138 + t139`. O sufixo `SI` indica um registrador inteiro de 32 bits.
- Atualmente o GCC utiliza uma série de IRs.

IRs em Compiladores Reais

- Os *front-ends* do GCC primeiro produzem uma árvore muito próxima do código fonte.
- A seguir essa árvore é convertida para o formato GIMPLE, segunda IR utilizada na suíte.
- GIMPLE é uma IR híbrida independente da linguagem fonte. É basicamente um CFG aonde os nós contém blocos de código em TAC.
- Por fim, os *back-ends* convertem GIMPLE em RTL para as fases finais de otimização e geração de código.

IRs em Compiladores Reais

- O compilador LLVM (*low-level virtual machine*) usa uma única IR baixo-nível, como o nome já indica.
- A IR da LLVM é um TAC com extensões para tipos e acessos a vetores e estruturas.
- LLVM era originalmente utilizado em conjunto com os *frond-ends* do GCC, através de uma tradução de GIMPLE para LLVM.
- Hoje em dia já há a família `Clang` de *front-ends* desenvolvida em conjunto com a LLVM.

Conclusão

- O conhecimento das diferentes IRs existentes é essencial para a construção adequada de um compilador.
- No Laboratório 05 vamos modificar o *parser* para realizar a construção da AST do código de entrada durante a análise sintática. A AST será a IR principal do nosso compilador.
- A seguir, no Laboratório 06, vamos implementar um interpretador baseado em pilha, que caminha na AST executando o código.
- Por fim, no Laboratório 07, vamos gerar uma IR linear (TAC) a partir do caminhamento na AST. Essa IR linear leva ao passo final de geração de código.

Aula 05 – Representação Intermediária

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**