



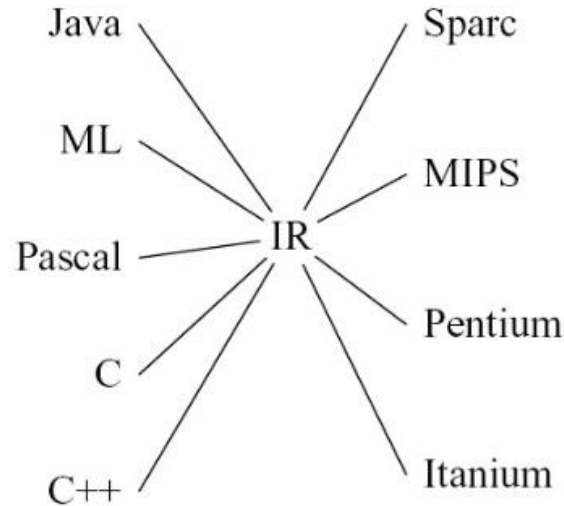
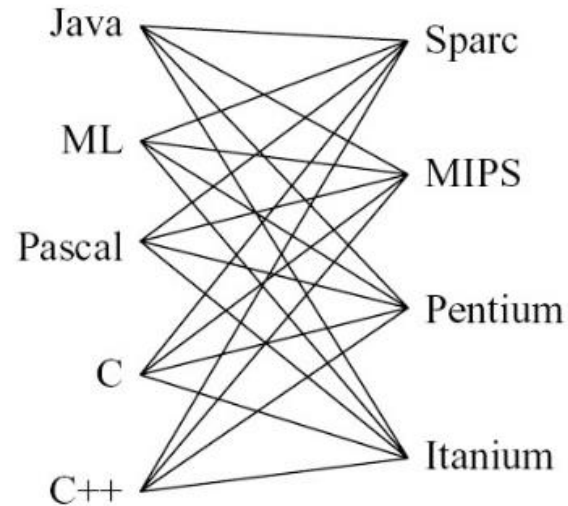
Geração de Código Intermediário

Parte I – Representações Intermediárias

Código Intermediário

Uma representação intermediária (IR) é um tipo de linguagem de máquina abstrata que pode expressar as operações da máquina-alvo sem se comprometer com muitos detalhes específicos da máquina

É também independente dos detalhes da linguagem de origem



O uso da representação intermediária

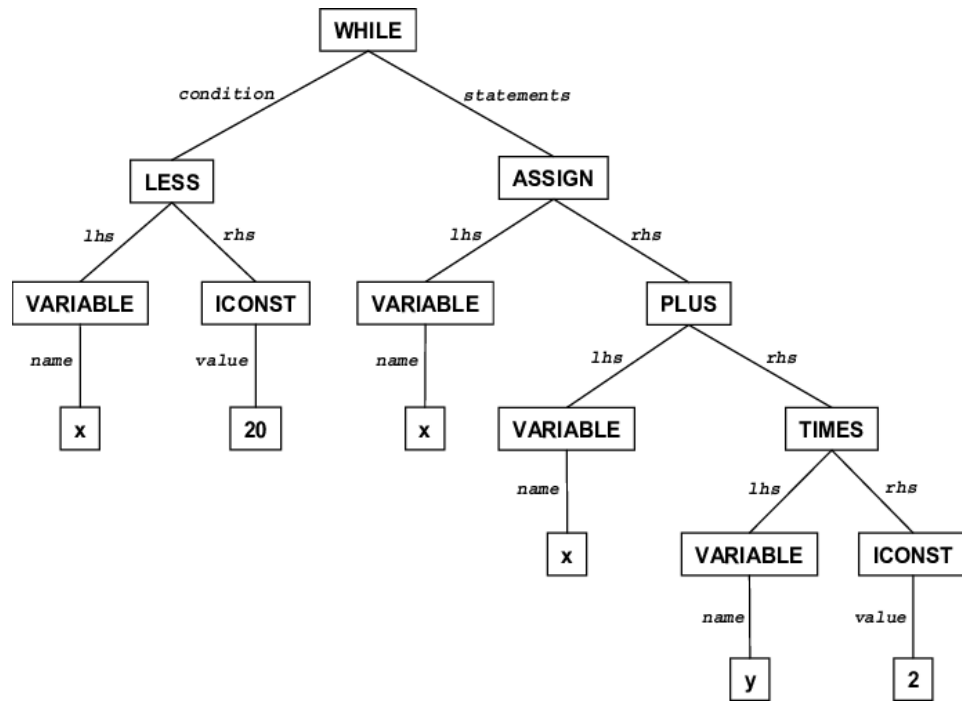
Linguagens Intermediárias

- Existem muitos tipos diferentes de IR que podem ser usados
- Alguns estão muito próximos da AST's, enquanto outros são semelhantes a uma linguagem assembly
- Alguns exemplos:
 - Representações gráficas: árvore e grafo de sintaxe
 - Máquina de Pilha
 - Códigos de 3 endereços
 - *Static Single Assignment* (SSA)

Representações Gráficas

.Árvores Sintáticas Abstratas

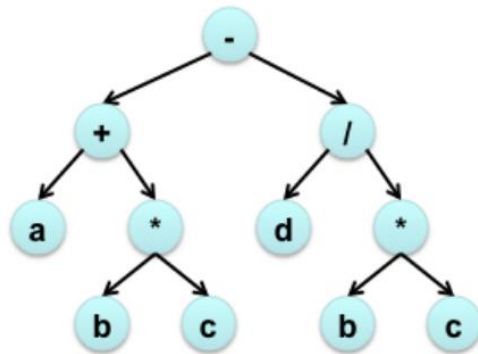
- AST é uma representação próxima ao nível da linguagem fonte
- Por causa de sua correspondência aproximada com uma árvore de análise, o analisador pode construir uma AST diretamente (*já vimos anteriormente*)
- Para gerar linguagem assembly, pode-se simplesmente executar uma travessia pós-ordem da AST e



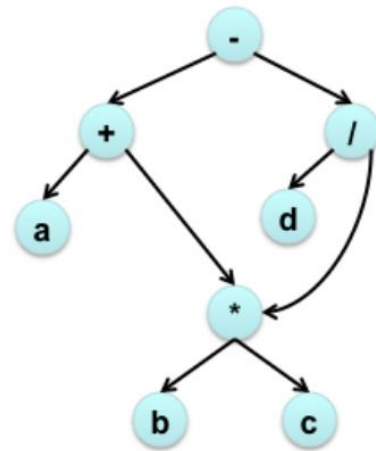
Representações Gráficas

.Grafos de sintaxe

- Um grafo é semelhante a AST, exceto que pode ter uma estrutura de grafo arbitrária
- Subexpressões comuns são identificadas




AST



Grafo

Máquina de pilha

- O código de máquina de pilha (um endereço) pressupõe a presença de uma pilha de operandos
- A maioria das operações pega seus operandos da pilha e coloca seus resultados de volta na pilha
- O código da máquina de pilha é simples de gerar e executar

$a - 2 \times b$ 
push 2
push b
multiply
push a
subtract

Código de 3 endereços

.No código de três endereços, a maioria das operações tem a forma:

$$i \leftarrow j \text{ op } k$$

com um operador (op), dois operandos (j e k) e um resultado (i)

.Alguns operadores, como atribuição imediata e salto, precisarão de menos argumentos

.Semelhantes a linguagens de montagem

Código de 3 endereços

.Atribuições:

$a = b \text{ op } c$

$a = \text{uop } b$

$a = b$

.Desvios:

goto L (salto incondicional para a label L)

if t goto L (se t é verdadeiro, salta para L)

if a relop b goto L (salta para L se a relop b for verdadeiro)

Código de 3 endereços

```
int a[10], b[10], dot_prod,i;
```

```
dot_prod = 0;
```

```
for (i=0; i<10; i++)
```

```
    dot_prod += a[i]*b[i];
```



dot_prod = 0;		T6 = T4[T5]
i = 0;		T7 = T3*T6
L1: if(i >= 10) goto L2		T8 = dot_prod+T7
T1 = addr(a)		dot_prod = T8
T2 = i*4		T9 = i+1
T3 = T1[T2]		i = T9
T4 = addr(b)		goto L1
T5 = i*4		L2:

Static Single Assignment (SSA)

- A forma de atribuição única estática (SSA) é uma representação do programa em que as variáveis são divididas em "instâncias"
- Cada nova atribuição a uma variável resulta em uma nova instância
- As instâncias da variável são numeradas de forma que cada uso de uma variável pode ser facilmente vinculado de volta a um único ponto de definição

Static Single Assignment (SSA)

Código-fonte

```
int x = 1;  
int a = x;  
int b = a + 10;  
x = 20 * b;  
x = x + 30;
```



Código em forma de SSA

```
int x_1 = 1;  
int a_1 = x_1;  
int b_1 = a_1 + 10;  
x_2 = 20 * b_1;  
x_3 = x_2 + 30;
```

Static Single Assignment (SSA)

- Uma peculiaridade surge quando uma variável recebe um valor diferente em dois ramos de um condicional
- Seguindo a condicional, a variável pode ter qualquer um dos valores, mas não sabemos qual
- Para expressar isso, introduzimos uma nova função $\phi(x, y)$ que indica que o valor x ou y pode ser selecionado em tempo de execução

```
if (y < 10) {  
    x = a;  
} else {  
    x = b;  
}
```

Se torna:

```
if (y_1 < 10) {  
    x_2 = a;  
} else {  
    x_3 = b;  
}  
x_4 = phi(x_2, x_3);
```

Static Single Assignment (SSA)

```
i ← 123
j ← i * j
repeat
  write j
  if (j > 5) then
    i ← i + 1
  else
    break
  end
until (i > 234)
```

(a)

```
i0 ← ⊥
j0 ← ⊥
i1 ← 123
j1 ← i1 * j0
repeat
  i2 ←  $\phi(i_1, i_6)$ 
  j2 ←  $\phi(j_1, j_5)$ 
  write j2
  if (j2 > 5) then
    i3 ←  $\phi(i_2)$ 
    j3 ←  $\phi(j_2)$ 
    i4 ← i3 + 1
  else
    i5 ←  $\phi(i_2)$ 
    j4 ←  $\phi(j_2)$ 
    break
  end
  i6 ←  $\phi(i_4)$ 
  j5 ←  $\phi(j_3)$ 
until (i6 > 234)
i7 ←  $\phi(i_6, i_5)$ 
j6 ←  $\phi(j_5, j_4)$ 
```

(b)

Escolhendo uma Linguagem Intermediária

.Uma linguagem intermediária deve, idealmente, ter as seguintes propriedades:

- Deve ser fácil traduzir de uma linguagem de alto nível para a linguagem intermediária
 - . Esse deve ser o caso para uma ampla gama de diferentes linguagens fonte
- Deve ser fácil traduzir da linguagem intermediária para o código de máquina
 - . Isso deve ser verdade para uma ampla gama de arquiteturas diferentes
- O formato intermediário deve ser adequado para otimizações
- **Problema: 2 primeiras propriedades são conflitantes!!!**

Exemplos de IR's usadas atualmente

- .GNU Simple Representation (GIMPLE)*
- .LLVM - Low Level Virtual Machine*
- .Java Virtual Machine (JVM)*

GNU Simple Representation (GIMPLE)

- .IR usada nos primeiros estágios do compilador GNU C
- .GIMPLE pode ser visto como uma forma simplificada de C em que todas as expressões foram divididas em operadores individuais em valores em forma de atribuição única estática (SSA)
- .Condicionais básicos são permitidos e os loops são implementados usando *goto*

GNU Simple Representation (GIMPLE)

```
void main ()  
{  
    int a, b, c;  
  
    a=10;  
    b=20;  
    c=0;  
  
    if (a<b)  
        c=a+b;  
}
```



```
main ()  
{  
    int a;  
    int b;  
    int c;  
  
    a = 10;  
    b = 20;  
    c = 0;  
    if (a < b) goto <D.2319>; else goto <D.2320>;  
    <D.2319>:  
    c = a + b;  
    <D.2320>:  
}
```

gcc -fdump-tree-gimple

LLVM - Low Level Virtual Machine

- O projeto LLVM consiste em uma linguagem e um conjunto correspondente de ferramentas para a construção de compiladores
- Uma variedade de front-ends de compilador suportam a geração de código intermediário LLVM
 - Código LLVM pode ser otimizado por uma variedade de ferramentas independentes e, em seguida, convertido novamente em código de máquina nativo ou bytecode para máquinas virtuais

LLVM - Low Level Virtual Machine

```
void main ()  
{  
    int a, b, c;  
  
    a=10;  
    b=20;  
    c=0;  
  
    if (a<b)  
        c=a+b;  
}
```



```
define dso_local void @main() #0 {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    %3 = alloca i32, align 4  
    store i32 10, i32* %1, align 4  
    store i32 20, i32* %2, align 4  
    store i32 0, i32* %3, align 4  
    %4 = load i32, i32* %1, align 4  
    %5 = load i32, i32* %2, align 4  
    %6 = icmp slt i32 %4, %5  
    br i1 %6, label %7, label %11  
  
7:                                     ; preds = %0  
    %8 = load i32, i32* %1, align 4  
    %9 = load i32, i32* %2, align 4  
    %10 = add nsw i32 %8, %9  
    store i32 %10, i32* %3, align 4  
    br label %11  
  
11:                                   ; preds = %7, %0  
    ret void  
}
```

```
clang -S -emit-llvm
```

Java Virtual Machine (JVM)

- JVM é uma definição abstrata de uma máquina baseada em pilha
- O código de alto nível escrito em Java é compilado em arquivos .class que contêm uma representação binária do bytecode JVM
 - As primeiras implementações da JVM eram interpretadores que liam e executavam o bytecode da JVM
 - Implementações posteriores executaram a compilação just-in-time (JIT) do bytecode em linguagem assembly nativa, que pode ser executada diretamente

Java Virtual Machine (JVM)

```
public static void main(String[] args) {  
    int a = 1;  
    int b = 2;  
    int c = a + b;  
}
```



```
public static void main(java.lang.String[]);  
descriptor: ([Ljava/lang/String;)V  
flags: (0x0009) ACC_PUBLIC, ACC_STATIC  
Code:  
stack=2, locals=4, args_size=1  
0: iconst_1  
1: istore_1  
2: iconst_2  
3: istore_2  
4: iload_1  
5: iload_2  
6: iadd  
7: istore_3  
8: return
```

```
javap -v Test.class
```

Geração de Código Intermediário:

Leitura Recomendada

.Livro Dragão:

- Capítulo 8 (1.a ed)**
- Capítulo 8 (2.a ed)**

.Cooper & Torcson

- Capítulo 5**

.Thain

- Capítulo 8**