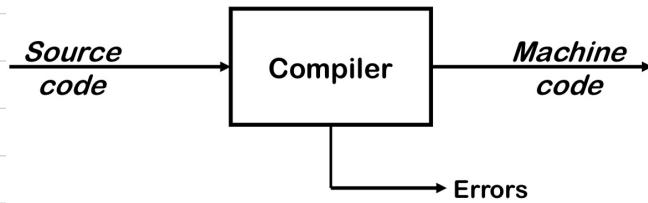




Compilador: programa que traduz um programa executável numa linguagem num programa executável noutra linguagem

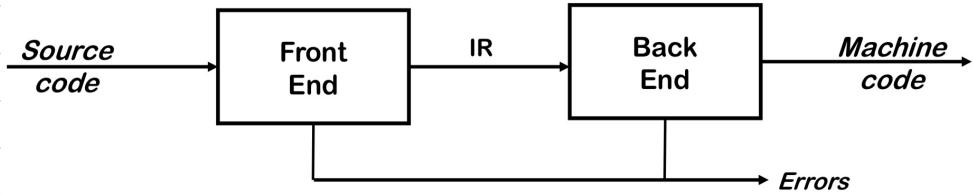
Interpretador: programa que lê um programa executável e produz os resultados de executar esse programa

A coresão é a propriedade mais importante que um compilador pode ter

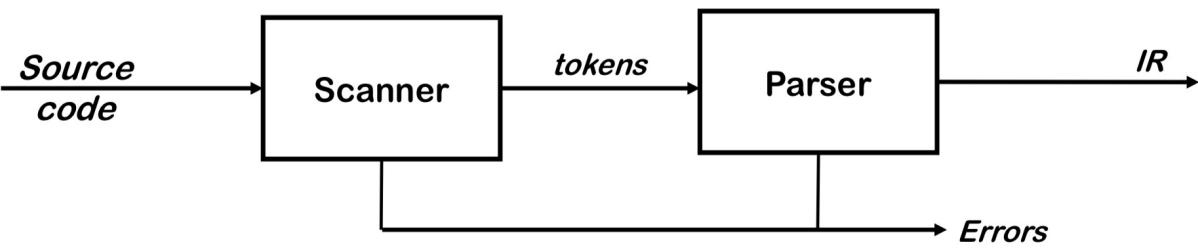


Compilador: deve...

- reconhecer programas legais (e ilegais)
- gerar código correto
- gerir o armazenamento de todas as variáveis (e código)
- concordar com o SO e o linker sobre a formatação do código objeto



- usa uma representação intermédia (IR)
- o front end mapeia o código fonte legal em IR
- o back end mapeia IR no Código máquina alvo
- admite múltiplos front ends e múltiplos backs



Front End:

- reconhece programas legais (e ilegais)
- reports errors de forma útil
- produz IR e mapa de armazenamento preliminar
- forma o código para o back end
- muitas das construções podem ser automatizadas

Scanner:

- mapeia um fluxo de caracteres em palavras - unidade básica da sintaxe
- produz tokens - uma palavra é a sua parte de fala / discurso (?)
- tokens típicos incluem número, identificador, operador, keyword, ...
- elimina espacos vazios
- a velocidade é importante

Parser:

- reconhece sintaxe livre de contexto e reporta erros
- guia análise ("semântica") sensível ao contexto
- constrói IR para o programa fonte

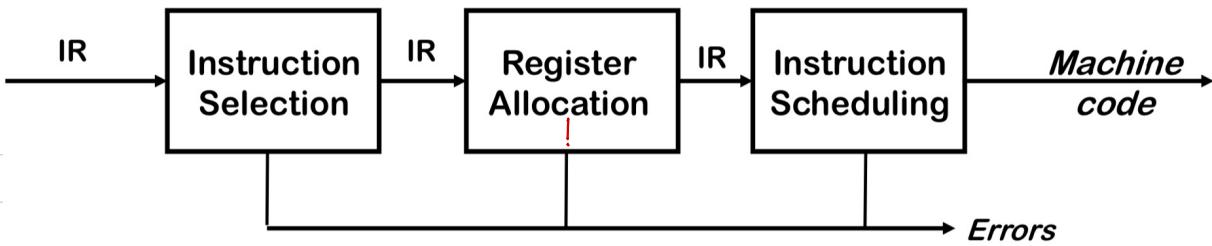
Gramática: $G = (S, N, T, P)$, $P: N \rightarrow N^* T$

S: símbolo inicial

N: conjunto de símbolos não-terminais

T: conjunto de símbolos terminais ou palavras

P: conjunto de produções ou regras de reescrita



Back End:

- traduz IR em código da máquina alvo
- escolhe instruções para implementar cada operação de IR
- decide que valores manter nos registos
- garante conformidade com as interfaces do sistema

Seleção de Instruções:

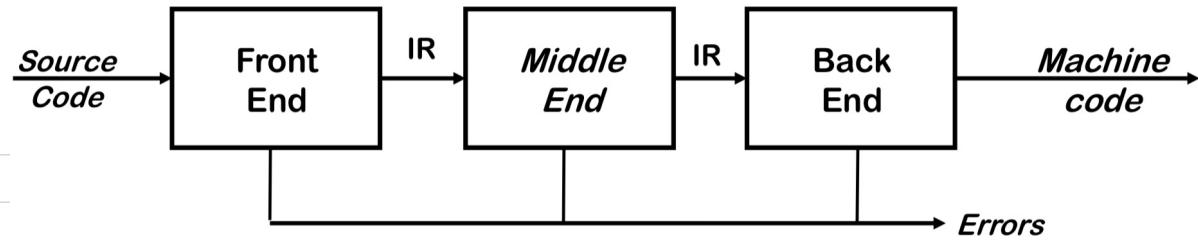
- produz código rápido e compacto
- aproveita as funcionalidades alvo (como modos de endereçamento)
- normalmente visto como um problema de "pattern matching"

Alocação de Registos:

- tem cada valor num registo quando é usado
- gera um conjunto limitado de recursos
- pode mudar escolhas de instruções e introduzir LOADs e STOREs
- a alocação ótima é NP-Completo

Escalonamento de Instruções:

- evita bloqueios e paragens do hardware
- usa todas as unidades funcionais produtivamente
- pode aumentar o tempo de vida de variáveis



Middle End: melhoria (ou otimização) do código

- analisa IR e rescreve (ou transforma IR)
- o objetivo primário é reduzir o tempo de execução do código compilado
- deve preservar o "significado" do código (valores das variáveis)

Os otimizadores modernos não são estruturados como uma série de fases

RESUMO

1. Front End: lida com a linguagem de input - E? ✓? → IR

a. Scanner: mapeia um fluxo de caracteres em palavras

b. Parser: verifica a correta gramatical e sintática das palavras

2. Middle End

3. Back End

a. Decodificação de Instruções

b. Alocação de Registros

c. Escalonamento de Instruções

Palavra: unidade básica da sintaxe

Lexema: caracteres que formam uma palavra

Tipo de Token: categoria sintática/parte do discurso

Análise Léxica

- A sintaxe da linguagem é especificada com partes do discurso, não palavras
- A verificação sintática corresponde partes do discurso com uma gramática

Analizador Léxico: texto do programa fonte → tokens

Tokens: operadores, Keywords, números, caracteres, strings

- Um analisador léxico partitiona o texto do programa de input numa sub sequência de caracteres correspondente a tokens, anexando-lhes os atributos correspondentes e eliminando espaços vazios e comentários

Expressões Regulares: descrevem linguagens regulares

Operation	Definition
Union of L and M Written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M Written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L Written L^*	$L^* = \bigcup_{0 \leq i \leq \infty} L^i$
Positive Closure of L Written L^+	$L^+ = \bigcup_{1 \leq i \leq \infty} L^i$

Precedência:
 1. fecho
 2. concatenação
 3. alternância

- Expressões Regulares podem ser usadas para especificar as palavras a serem traduzidas para partes do discurso por um analisador léxico

Alfabeto Σ : conjunto finito de símbolos

String s : sequência finita de símbolos do alfabeto

String Vazio ϵ : string especial de comprimento zero

Linguagem L : conjunto de strings sobre um alfabeto

$L(r) = \{ \text{todas as strings que satisfazem } r \}$

$$L(r \wedge s) = L(r) \cup L(s)$$

$$L(r \cdot s) = \{ xy \mid x \in L(r) \wedge y \in L(s) \}$$

$$L(r^*) = \{ r_1 \dots r_k \mid r_i \in L(r), k \geq 0 \}$$

$$L(\epsilon) = \{ \epsilon \} \neq \emptyset$$

$$r^+ = r \cdot r^*$$

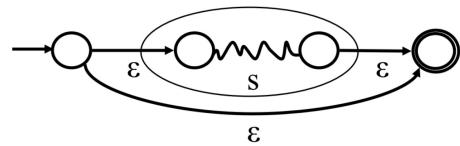
$$r? = r \mid \epsilon$$



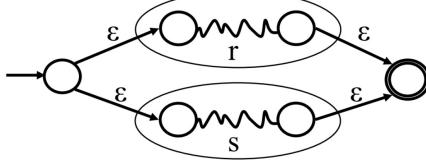
$r \cdot s$



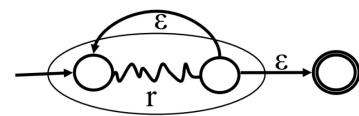
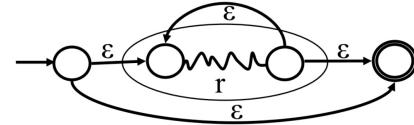
$r?$



$r \mid s$



r^+



Análise Lógica: RegEx \rightarrow NFA \rightarrow DFA

Fecho - ϵ : conjunto de estados que pode ser alcançado sem consumir input

$T \leftarrow S$

repeat

$$\begin{aligned} T' &\leftarrow T \\ T &\leftarrow T' \cup \left(\bigcup_{\lambda \in T} \text{edge}(\lambda, \epsilon) \right) \\ \text{until } T &= T' \end{aligned}$$

DFA edge (S, c): estados alcançáveis a partir do conjunto de estados S , dado o símbolo c

$$= \epsilon\text{-closure} \left(\bigcup_{\lambda \in S} \text{edge}(\lambda, c) \right)$$

Como converter NFA \rightarrow DFA?

$\text{states}[0] = s_1$

$\text{states}[1] = \epsilon\text{-closure}(\{s_1\})$

$p = 1$

$j = 0$

while ($j \leq p$) do

foreach $c \in \Sigma$ do

$e = \text{DFAedge}(\text{states}[j], c)$

if ($e = \text{states}[i]$ for some $i \leq p$) then

$\text{trans}[j, c] = i$

else

$p = p + 1$

$\text{states}[p] = e$

$\text{trans}[j, c] = p$

$j = j + 1$

end if

end foreach

end while

Construir uma tabela!

DFA State	NFA States	$\epsilon\text{-closure after transition on...}$		
		0...9	-	.
0	{1, 2}			

DFA State	NFA States	$\epsilon\text{-closure after transition on...}$		
		0...9	-	.
0	{1, 2}	{2, 3, 4, 8}	{2}	error
1	{2, 3, 4, 8}	{2, 3, 4, 8}	error	{5, 6, 8}
2	{2}	{2, 3, 4, 8}	error	error
3	{5, 6, 8}	{6, 7, 8}	error	error
4	{6, 7, 8}	{6, 7, 8}	error	error

• Um estado do DFA é de aceitação se e só se contiver um estado de aceitação do NFA

Como minimizar un DFA = $\{D, \Sigma, d, s_0, D_f\}$?

```

P ← {DF, {D - DF}}
while (P is still changing)
    T ← ∅
    for each set p ∈ P
        T ← T ∪ Split(p)
    P ← T
  
```

```

Split(S)
for each c ∈ Σ
    if c splits S into s1 and s2
        then return {s1, s2}
return S
  
```

Como convertir DFA → RE?

for i = 1 to N

for j = 1 to N

$$R^0_{ij} = \{a \mid \delta(s_i, a) = s_j\}$$

if (i = j) then

$$R^0_{ij} = R^0_{ij} \mid \{\epsilon\}$$

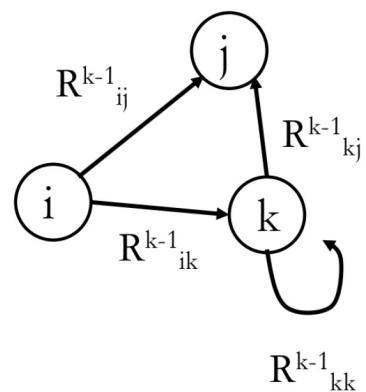
for k = 1 to N

for i = 1 to N

for j = 1 to N

$$R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \mid R^{k-1}_{ij}$$

$$L = | s_j \in S_F R^N_{1j} |$$



$$R^{k-1}_{kk}$$

Scanner: RE_s → NFA_s → NFA → DFA → DFA' → código executável

→ "Table-Driven"
→ "Direct-Coded"

LEX: .l → .c

1. Declarações: macros e declarações da linguagem - % } .. } %
2. Regras: expressões regulares e ações correspondentes a executar - { ... }
3. Código: funções de suporte

x	the character x
"x"	the character x, even if it is a special character
\x	the character x, even if it is a special character
x\$	the character x at the end of a line
^x	the character x at the beginning of a line
x?	Zero or one occurrence of x
x+	One or more occurrences of x
x*	Zero or more occurrences of x
xy	the character x followed by the character y
x y	the character x or the character y
[az]	the character a or the character z
[a-z]	from character a to character z
[^a-z]	Any character except from a to z
x{ n }	n occurrences of x
x{ m,n }	between m and n occurrences of x
x/y	x if followed by y (only x is part of the pattern)
.	Any character except \n
(x)	same as x, parentheses change operator priority
<<EOF>>	end of file

- DIGIT [0-9]
- INT { DIGIT } +
- EXP [E e] [+ -] ? { INT }
- REAL { INT } " " { INT } ({ EXP }) ?

Análise Sintática

Sintaxe: estrutura do programa

Semântica: significado do programa

→ Gramáticas Livres de Contexto

Derivação: dadas as strings α , β e γ e uma produção $A \rightarrow \beta$, um falso é $\alpha A \gamma \Rightarrow \alpha \beta \gamma$

Árvore: mostra a sequência de derivações realizadas

• Os nós internos são não terminais e as folhas são terminais

Derivação à Esquerda: aplicar uma produção ao não terminal mais à esquerda

Derivação à Direita: aplicar uma produção ao não terminal mais à direita

Parsing Top-Down: símbolo inicial \rightarrow string de tokens

Parsing Bottom-Up: string de tokens \rightarrow símbolo inicial

Gramática Ambígua: tem mais do que uma derivação mais à esquerda/mais à direita para uma única forma sentencial \rightarrow reescrever para desambiguar

Parser: se a frase de input não pertence à linguagem, o algoritmo de decisão deve provar que não pode ser derivada usando a gramática; se a frase de input pertence à linguagem, o algoritmo de decisão deve apresentar a sequência de derivações

↳ pode ter de ver à frente

Parsing

Tóp - Down: LL(1), descida recursiva - Parsing preditivo

1. Começar na raiz da árvore de parsing e crescer em direção às folhas
2. Escolher uma produção e tentar fazer corresponder o input
- Uma má escolha pode levar a backtracking

Bottom - Up: LR(1), precedência dos operadores

1. Começar nas folhas da árvore de parsing e crescer em direção à raiz
2. Conforme o input é consumido, codificar possibilidades num estado interno
- Começar num estado válido para primeiros tokens legais

Algoritmo de Parsing Tóp - Down: constrói o nó raiz da árvore de parse e repete até a margem da árvore de parse corresponder com a string de input

1. Em cada nó etiquetado A, escolher uma produção com A do lado esquerdo e, para cada símbolo do lado direito, construir o filho apropriado
2. Quando um símbolo terminal é adicionado à margem e não corresponde com a margem, fazer backtracking
3. Encontrar o próximo nó a ser expandido (etiqueta não terminal)

- A chave é escolher a produção certa no passo 1
- Se escolha errada de expansão leva a não-terminal $\exists A \in NT: A \Rightarrow^* A\alpha$
- Parceiros Tóp - Down não conseguem lidar com gramáticas recursivas à esquerda!
- Não-terminal é uma má propriedade em qualquer parte de um compilador!



Como eliminar recursões à esquerda?

$$F \rightarrow F \alpha \quad | \quad \beta \quad \rightsquigarrow \quad F \rightarrow \beta F' \quad F' \rightarrow \alpha F' \quad | \quad \epsilon$$

Parsing Preditivo: dado $A \rightarrow \alpha | \beta$, o parser deve ser capaz de escolher entre α e β

FIRST(α): conjunto de tokens que aparecem como primeiro símbolo em alguma string que deriva de α , RHS $\alpha \in G$

$$\forall \text{ RHS } \alpha \in G, \quad n \in \text{FIRST}(\alpha) \Leftrightarrow \alpha \Rightarrow^* x^n \gamma$$

FOLLOW(A): conjunto de todas as palavras na gramática que podem legalmente aparecer depois de A

$$\text{FIRST}^+(\alpha) = \begin{cases} \text{FIRST}(\alpha) \cup \text{FOLLOW}(A), & \text{se } \epsilon \in \text{FIRST}(\alpha) \\ \text{FIRST}(\alpha), & \text{se } \epsilon \notin \text{FIRST}(\alpha) \end{cases}$$

Gramática LL(1): $A \rightarrow \alpha, B \rightarrow \beta \Rightarrow \text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset$

Paser Preditivo: aproveita a propriedade LL(1)
DESCIDA RECURSIVA

Como transformar para LL(1)?

FATORIZAÇÃO À ESQUERDA

$$A \rightarrow \alpha \beta_1 \quad | \quad \alpha \beta_2 \quad | \quad \alpha \beta_3$$

$$A \rightarrow \alpha Z \quad Z \rightarrow \beta_1 \quad | \quad \beta_2 \quad | \quad \beta_3$$

RESUMO

1. Construir conjuntos FIRST e FOLLOW
2. Manter a gramática para ser LL(1)
 - a. remover recursão à esquerda
 - b. fatorar à esquerda
3. Definir um procedimento para cada não-terminal
 - a. implementar um caso para cada lado direito
 - b. chamar procedimentos conforme necessário para não-terminal
4. Adicionar código extra, conforme necessário

FIRST:

- $\text{Se } \alpha \Rightarrow^* a\beta, a \in T, \beta \in (T \cup NT)^*, \text{ então } a \in \text{FIRST}(\alpha)$
- $\text{Se } \alpha \Rightarrow^* \epsilon, \text{ então } \epsilon \in \text{FIRST}(\alpha)$
- $\text{Se } \alpha \Rightarrow \beta_1 \dots \beta_k, \text{ então } a \in \text{FIRST}(\alpha) \text{ se } \exists i : a \in \text{FIRST}(\beta_i) \wedge \epsilon \in \text{FIRST}(\beta_{i+1} \dots \beta_k)$

Nota: Se $\alpha = X\beta$, então $\text{FIRST}(\alpha) = \text{FIRST}(X)$

FOLLOW:

- $\text{Se } S \text{ é o símbolo inicial, então } \$ \in \text{FOLLOW}(S)$
- $\text{Se } A \rightarrow \alpha B \beta, \text{ então } \forall (a/\epsilon) \in \text{FIRST}(\beta), (a)\epsilon \in \text{FOLLOW}(B)$
- $\text{Se } (A \rightarrow \alpha B \vee A \rightarrow \alpha B \beta) \wedge \epsilon \in \text{FIRST}(\beta), \text{ então } \text{FOLLOW}(A) \subset \text{FOLLOW}(B)$

Nota: $\epsilon \notin \text{FOLLOW}(\alpha)$

Como construir Parson Top-Down?

- Uma linha para cada NT
- Uma coluna para cada T
- Um interpretador para a tabela

Algoritmo:

$X \leftarrow$ símbolo no topo da pilha
 $a \leftarrow$ símbolo atual de input

1. Se $X = a = \$$, o parser para e sucede
2. Se $X = a \neq \$$, o parser faz jog de X do topo da pilha e avança o input
3. Se $X \in NT$, o parser consulta a entrada $M[X, a]$ da tabela de parsing
 - se $M[X, a] = \{X \rightarrow UVW\}$, então substitui X por UVW
 - senão, erro

Como construir a tabela de parsing?

$M[X, a], X \in NT, a \in T$

1. Se $y \in FIRST(\beta)$, a entrada é a regra $X \rightarrow \beta$
2. Se $y \in FOLLOW(X) \wedge X \rightarrow E \in G$, a entrada é a regra $X \rightarrow E$
3. Senão, a entrada é erro

• Se alguma entrada é definida múltiplas vezes, G não é LL(1)

• Se $M[X, a]$ estiver vazia? modo de recuperação de erros

MODO PÂMICO

• parser símbolos no input até que um token num conjunto de sincronizações de tokens apareça no input

```
token ← next_token()  
push EOF onto Stack  
push the start symbol, S, onto Stack  
TOS ← top of Stack  
loop forever  
if TOS = EOF and token = EOF then  
    break & report success  
else if TOS is a terminal then  
    if TOS matches token then  
        pop Stack                                // recognized TOS  
        token ← next_token()  
    else report error looking for TOS  
else                                         // TOS is a non-terminal  
    if TABLE[TOS,token] is  $A \rightarrow B_1 B_2 \dots B_k$  then  
        pop Stack                                // get rid of A  
        push  $B_k, B_{k-1}, \dots, B_1$                 // in that order  
    else report error expanding TOS  
TOS ← top of Stack
```

```
graph LR; Success[exit on success] --> EOF[if TOS = EOF and token = EOF then]
```

Parser Shift-Reduce

Shift: fazer shift do elemento atual no topo da stack e mover o apontador para o input atual

Reduce: aplicar uma produção cujo lado direito coincide com o topo da stack, remover os símbolos da stack e adicionar o não terminal do lado esquerdo à stack

Quando chegar ao fim do fluxo:
aceitar se a stack só tiver o símbolo inicial e rejeitar se a stack tiver mais do que o símbolo inicial

Se...

- 1) os símbolos de topo da stack coincidem com o lado direito de uma produção → **REDUCE**: fazer pop do lado direito do topo da stack e fazer push do lado esquerdo para o topo da stack
- 2) não se encontram nenhuma produção → **SHIFT**: fazer push do input atual para a stack
- 3) o input estiver vazio → aceitar se só o símbolo inicial estiver na stack e rejeitar caso contrário

Como construir um Parser Shift-Reduce?

1. Criar um DFA: que codifique todos os estados em que o parser pode estar, com transições a ocorrer em terminais e não terminais
2. Criar uma Tabela: que guarde que ação deve ser tomada dado o estado atual e o caractere de input atual
3. Mantener duas stacks: uma de estados e uma de símbolos
↳ em cada ação, procurar na tabela e mover para o estado correspondente

de uma forma sentencial direta

Handle: substituting β da fronteira da árvore que coincide com alguma produção $A \rightarrow \beta$ que ocorre como um faro na derivação mais à direita
• Far $\langle A \rightarrow \beta, k \rangle$ em que k é a posição do símbolo mais à direita de β em γ

• Se $\langle A \rightarrow \beta, k \rangle$ é um handle, então substituir β em k com A produz a forma sentencial direta a partir da qual γ é derivada na derivação mais à direita

• Como γ é uma forma sentencial direta, a substituição à direita de um handle contém apenas símbolos terminais

Forma Sentencial: faro de uma derivação (contém pelo menos um não terminal)

• Se uma gramática não é ambígua, então cada forma sentencial direta tem um único handle

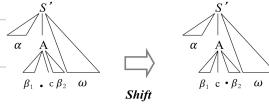
for $i \leftarrow n$ to 1 by -1

Find the handle $\langle A_i \rightarrow \beta_i, k_i \rangle$ in γ_i

Replace β_i with A_i to generate γ_{i-1}

1. Fazer shift até o topo da stack ser o lado direito de um handle
2. Encontrar o lado esquerdo de um handle e reduzir

RESUMO



$$A \rightarrow \beta_1 \bullet c \beta_2$$

Shift: shift da próxima palavra para a stack

Reduce: o lado direito do handle está no topo da stack — localizar o lado esquerdo do handle na stack, fazer pop da handle da stack e push do lado esquerdo apropriado



$$A \rightarrow \beta_1 \bullet$$

Item Válido: o item $A \rightarrow \beta_1 \cdot \beta_2$ é válido para um prefixo viável $\alpha \beta_1$ se houver uma derivação

$$S' \xrightarrow[\text{RM}]{*} \alpha A w \xrightarrow[\text{RM}]{*} \alpha \beta_1 \beta_2 w$$

Se $\beta_2 \neq \epsilon$, então o item válido $A \rightarrow \beta_1 \cdot \beta_2$ sugere que a ação é SHIFT
Se $\beta_2 = \epsilon$, então o item válido $A \rightarrow \beta_1$ sugere que a ação é REDUCE

Closure (I): qualquer item em I também está em closure (I) e se $A \rightarrow \alpha \cdot B \beta$ está em closure (I) e $B \rightarrow \cdot \gamma$ é um item, então adiciona $B \rightarrow \cdot \gamma$ ao closure (I)

$\text{goto}(I, X) = \text{closure}(\{A \rightarrow \alpha \cdot X \cdot \beta \mid A \rightarrow \alpha \cdot X \beta \text{ em } I\})$ = novo conjunto obtido ao mover o ponto sobre X

Um item capture quanto de uma produção é que se lê scan até ao momento

Como construir o DFA dos estados?

1. Começar com a produção $\langle S' \rangle \rightarrow \cdot \langle S \rangle \$$
2. Criar o primeiro estado a ser closure ($\langle S' \rangle \rightarrow \cdot \langle S \rangle \$$)
3. Escolher um estado I e, para cada $A \rightarrow \alpha \cdot X \beta$ em I, encontrar $\text{gato}(I, X)$ e torná-lo num estado (se ainda não for), adicionando uma aresta X de I até ao estado $\text{gato}(I, X)$

Como construir a tabela do parser?

1. Transição com terminal \rightarrow SHIFT
2. Transição com não-terminal \rightarrow GOTO
3. Se existir um item $A \rightarrow \alpha \cdot$ num estado, fazer uma redução com essa produção para todos os terminais \rightarrow REDUCE

Análise Sémantica

Tradução Dirigida pela Sintaxe: processo de tradução guiado por Gramáticas Livres de Contexto

↓ AUMENTADA

Gramática com Atributos: gramática livre de contexto aumentada com um conjunto de regras — cada símbolo tem um conjunto de valores/atributos e as regras especificam como computar o valor para cada atributo

- As regras semânticas estão associadas com produções que usam os valores dos símbolos (pai, filhos ou irmãos)

Definições Dirigidas pela Sintaxe: ordem implícita - abstrato

Esquema de Tradução: ordem explícita - concreto

- Gramáticas com Atributos podem especificar ações sensíveis ao contexto

RHS → LHS

Atributos Sintetizados: fluem para cima — usam valores dos próprios, dos filhos e de constantes ::

Atributos Herdados: fluem para baixo — usam valores dos próprios, do pai, dos irmãos e de constantes ::

Como determinar a ordem de avaliação das regras?

- String de input
- árvore de parsing — com atributos
- grafo acíclico de dependências — dos atributos → **ORDENAÇÃO TOPOLOGICA**
- ordem de avaliação das regras semânticas

Métodos de Avaliação:

1. Dinâmico: ordena topologicamente o grafo de dependências e segue as arestas
2. Baseado nas Regras: tenta descobrir boas ordenações ao analisar as regras
3. Oblívio: ignora a estrutura do grafo

Gramáticas Fortemente Não Circulares: geram grafos acíclicos ☺

Gramáticas "S-Attributed": todos os atributos são sintetizados — as regras podem ser avaliadas de baixo para cima num único passo BOTTOM UP ☺

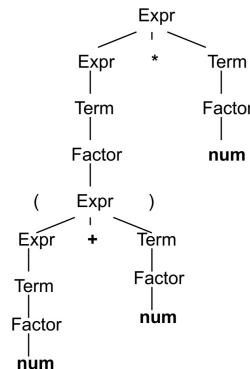
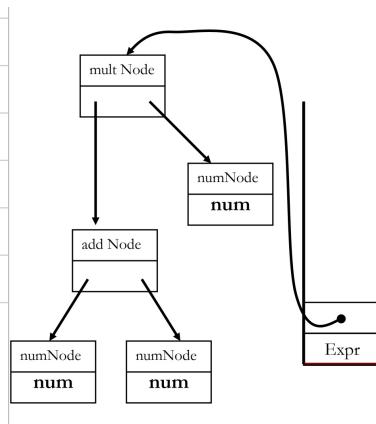
• Regras de Cópia multiplicam-se rapidamente, requerem mais espaço e aumentam o "overhead" cognitivo, pelo que uma tabela torna-se útil

Tradução Dirigida pela Sintaxe: ações semânticas são embaladas nas produções — avaliação baseada nas regras

Top-Down: associa ações com o código das funções de produções

Bottom-Up: executa ações nas reduções

Gramáticas "L-Attributed": a informação move-se da esquerda para a direita — ações semânticas não executadas em operações de redução de produções — pode ser avaliada de baixo para cima num único passo BOTTOM UP



(| num | + | num |) * num

Representação Intermédia

Representação Intermédia: codifica o conhecimento do compilador sobre o programa, representando e preservando a semântica

Propriedades:

1. Facilidade de Geração de Código
2. Facilidade de Manipulação
3. Tamanho dos Procedimentos
4. Liberdade de Expressão
5. Nível de Abstracção

Categorias {
1. Estrutural: orientada graficamente ~ alto-nível
2. Linear: pseudo-código para uma máquina abstrata ~ baixo-nível
3. Híbrida: combinação de grafos e código linear

Árvore de Sintaxe Abstrata (AST): árvore de parsing do procedimento com os nós para a maior parte dos não-terminal removidos

Grafo Dirigido Acíclico (DAG): AST com um único nó para cada valor

Código com 3 Endereços: L: $x = y \text{ of } z$ OU if exp goto L

- Os nomes simbólicos são os endereços da variável correspondente
- As variáveis temporárias (insertidas pelo compilador) têm prefixo "t"
- Os registos (em número limitado) têm prefixo "r"

Atribuições:

1. $x = y$ or y
2. $x = \emptyset$ or y
3. $x = y$
4. $x = y[i]$
5. $x[i] = y$
6. $x = \emptyset(y, z)$

Operações de Memória:

1. $x = &y$
2. $x = *y$
3. $*x = y$

Transferência de Controlo e Chamadas a Funções:

1. goto L
2. if (a relaz b) goto L
3. y = call f, n

número de argumentos

Grafo de Controlo de Fluxo: modela a transferência de controlo no procedimento - os nós no grafo são blocos básicos

Atribuição Estática Única (SSA): cada nome é definido exatamente uma vez e cada uso refere-se a exatamente um nome

Modelo Registo-a-Registo: mantém todos os valores em registos - o "back-end" do compilador deve inserir loads e stores

Modelo Memória-a-Memória: mantém todos os valores em memória - o "back-end" do compilador pode remover loads e stores

"Name Space": cada procedimento tem o seu próprio espaço de nomes com âmbito lexical para simplificar regras de nomeação e resolver conflitos
→ "Lexically Scoped Symbol Tables"

Controlo de Fluxo:

- requer código que guarda e restaura um endereço de retorno
- deve manter parâmetros reais em parâmetros formais
- deve criar arranjoamento para variáveis locais (e parâmetros)
- deve preservar o estado de \top enquanto é executado, criando uma única localização para cada activação de procedimento

Tabela de Símbolos: liga nomes (símbolos) a entidades do programa (variáveis e procedimentos) para verificação, geração de código e debug

Símbolo: (nome, tipo, classe de arranjoamento, âmbito, visibilidade, tempo de vida)

Âmbito: unidade da estrutura de um programa estático que pode ter uma ou mais variáveis declaradas lá dentro

Visibilidade: refere-se a que âmbitos um dado nome de variável se refere a uma instância particular de variável

Tempo de Vida: período de execução a partir do ponto em que uma variável se torna visível até que deixa de estar visível

Classe de Arranjoamento: global; ficheiro/módulo; automático

Modificadores: como os valores podem ser mudados e retidos

Volátil: pode ser modificado armazenamente

Estático: retém valores para além das fronteiras do tempo de vida

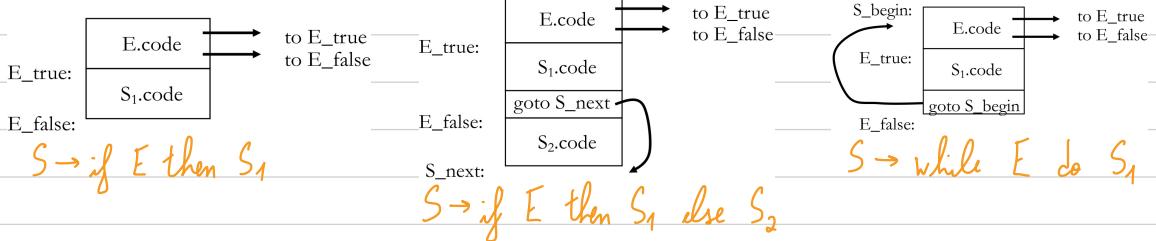
insert (name, level): cria um registo para "name" em "level"

lookup (name, level): retorna um apontador ou índice

delete (level): remove todos os "names" declarados em "level"

Avaliação Aritmética: associar 0 e 1 com o resultado de predicados e combinar com instruções lógicas

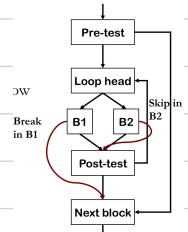
Avaliação de Controle de Fluxo: avaliação "int-to-círculo" - para a avaliação mal o resultado é conhecido



Antifazam: instrução que especifica um valor específico em ordem inversa, i.e., a instrução mais feita da chamada é o valor do último argumento

Ciclo:

1. Avaliar condição antes do ciclo
2. Avaliar condição depois do ciclo
3. Regressar ao topo



Switch-Case:

1. Avaliar a expressão de controle
2. Ir para o caso selecionado **COMO?**: if-else; tabela; endereços
3. Executar o código para esse caso
4. Ir para o código depois do caso

Como acceder a $A[i]$?

$$\text{base } A + (i - \text{low}) \times \text{sizeof}(\text{baseType}(A))$$

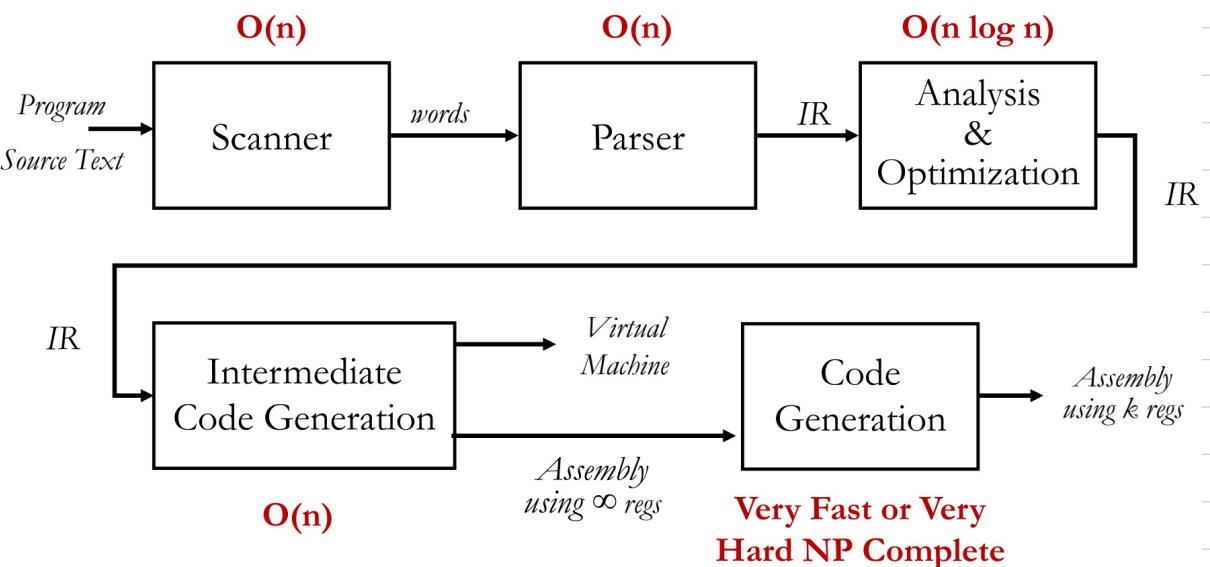
- Ordem por Linha: sequência de linhas consecutivas $A[i_1, j_1]$
 $\text{base } A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_1 - \text{low}_1) \times \text{size of}(\text{baseType}(A))$
- Ordem por Coluna: sequência de colunas consecutivas $A[j_1, i_1]$
 $\text{base } A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_2 - \text{low}_2) \times \text{size of}(\text{baseType}(A))$
- Vetores de Indicação: vetores de apontadores para apontadores para valores $^*(A[i_1])[i_2]$

• Arrays como parâmetros são passados por referência — endereço e dimensão

Structs/Records: composição lógica linear

Unions/Variants: alternância lógica exclusiva com partilha de armazenamento

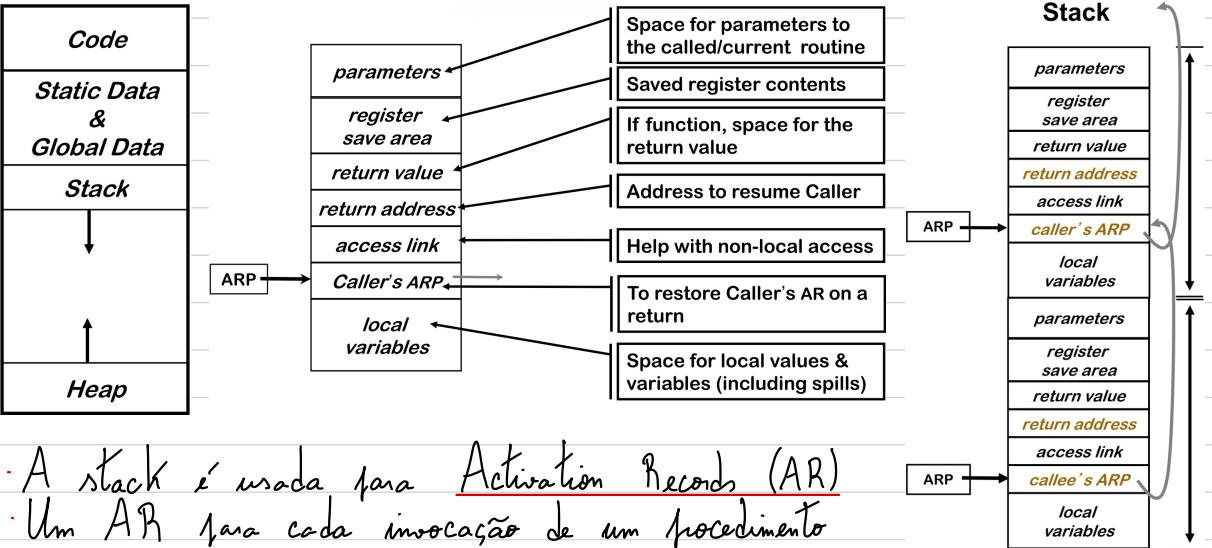
Como aceder? offset desde o início do objeto



Ambientes de Execução



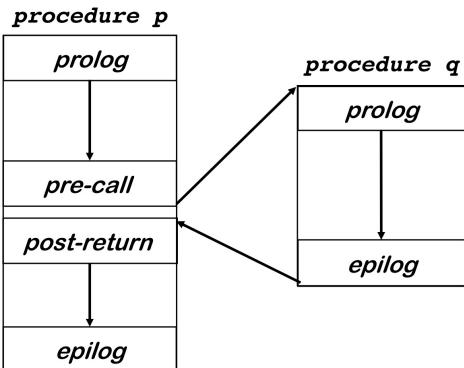
Procedimento: unidade básica de abstração e raciocínio do programa



- A stack é usada para Activation Records (AR)
- Um AR para cada invocação de um procedimento

Chamada: passagem de argumentos e transferência de controle

Retorno: recuperação de resultados e transferência de controle



Cada procedimento tem:

- prologo procedão
- epólogo procedão

Cada chamada tem:

- sequência pré-chamada
- sequência pós-retorno

Sequência Pré-Chamada: configura AR básicos da função chamada e ajuda a preservar o seu próprio ambiente

1. Aloca espaço para os AR da função chamada (exceto locais)
 2. Avalia cada parâmetro e armazena valor/endereço ^{da função chamada}
 3. Guarda o endereço de retorno e ARP da função chamadora no ARP
 4. Se não usados links de acesso, encontra os antecessores e copia para ↑
 5. Guarda os registos a guardar da função chamadora no AR dela
- ↓ SALTA PARA

Código Prólogo: acaba de configurar o ambiente da função chamada e preserva partes do ambiente da função chamadora que vão ser incomodadas

1. Preserva os registos a guardar da função chamada ^{atual}
2. Se o Display está a ser usado, guarda a entrada e armazena o ARPV
3. Aloca espaço para os dados locais
4. Encontra áreas de dados estáticos referenciadas na função chamada
5. Lida com inicializações de variáveis locais

Código Epólogo: encerra o trabalho da função chamada e começa a restaurar o ambiente da função chamadora

1. Restaura os registos a guardar da função chamada
 2. Liberta espaço para dados locais, se necessário (na heap)
 3. Carrega o endereço de retorno a partir de AR
 4. Restaura ARP da função chamadora
- ↓ SALTA PARA

Sequência Pós-Retorno: acaba de restaurar o ambiente da função chamadora e coloca os valores de volta onde pertencem

1. Copia o valor de retorno do AR da função chamada, se necessário
2. Liberta o AR da função chamada
3. Restaura os registos a guardar da função chamadora
4. Restaura os parâmetros por referência para registos, se necessário
5. Continua a execução depois da chamada

- Registos "callee-saved": voláteis; denotados na chamada
- armazenam quantidades temporárias que não têm de ser preservadas entre chamadas
 - é responsabilidade da função chamadora colocar esses registos na pilha ou copiá-los para algum lado se quiser restaurar os seus valores após a chamada
 - espera-se que a função chamada destrua valores temporários nesses registos

- Registos "callee-saved": não voláteis; preservados na chamada
- armazenam valores que devem ser preservados entre chamadas
 - é responsabilidade da função chamada colocar esses registos na pilha ou copiá-los para algum lado se quiser restaurar os seus valores após a chamada
 - espera-se que a função chamada preserve valores temporários nesses registos

Onde vivem os Activation Records?

1. Na stack, se o tempo de vida do AR corresponde ao tempo de vida da invocação ② se o código executa normalmente um retorno
2. Na heap, se o procedimento pode viver mais do que quem o chamou ou se ele pode retornar um objeto que pode referenciar o seu estado de execução
3. Alocado estaticamente, se o procedimento não fizer chamadas

Como é que o compilador encontra as variáveis? Par { nível, offset } ARP CONSTANTE

- Se um procedimento P numa profundidade léxica N_P se refere a uma variável não local A num profundidade N_Q < N_P, então A pode ser encontrada seguindo (N_P-N_Q) ligações de acesso desde o ARP de P e accedendo à variável de offset A nesse ARP

sort
a,x
quicksort(1,9)
access link k,v

sort
a,x
quicksort(1,9)
access link k,v
quicksort(1,3)
access link k,v

sort
a,x
quicksort(1,9)
access link k,v
quicksort(1,3)
access link k,v
partition(1,3)
access link i,j

sort
a,x
quicksort(1,9)
access link k,v
quicksort(1,3)
access link k,v
partition(1,3)
access link i,j
exchange(1,3)
access link

sort calls quicksort

$$n_q = n_p + 1$$

quicksort calls quicksort

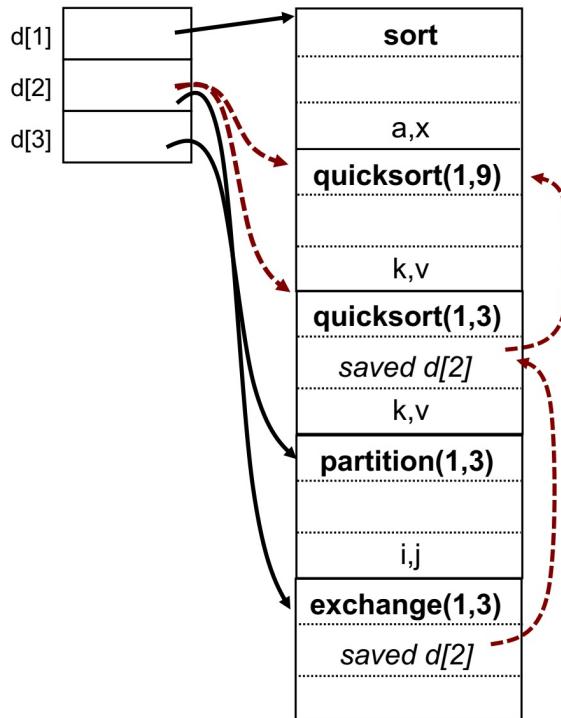
$$n_q = n_p$$

quicksort calls partition

$$n_q = n_p + 1$$

partition calls exchange

$$n_p = n_q + 1$$



Alocação de Registos

Alocação: determinar quais valores vão residir em registos

Atribuição: seleccionar um registo para cada um desses valores

Objetivo: alocação que minimiza o tempo de execução

Alocadores Locais: usam conhecimento ao nível das instruções

1. **Top-Down:** usam a frequência de utilização de variáveis para cada instrução - colocam em registos os nomes que aparecem mais frequentemente

2. **Bottom-Up:** avaliam as necessidades das instruções e reutilizam registos - registram quando os valores não são necessários mais tarde

Alocadores Globais: usam um paradigma de coloração de grafos - constroem um grafo de conflito/intervenção e encontram uma k -coloração para esse grafo (ou alteram o código para um problema próximo que pode ser k -colorido)

ALOCADOR TOP-DOWN

1. Estimar os benefícios de colocar cada variável num registo num bloco básico particular - $\text{Benefit}(V, B) =$ número de "usos" e "afis" de V em B

2. Estimar o benefício global - $\text{Tot Benefit}(V) = \text{Benefit}(V, B) \times \underset{\substack{? \\ \rightarrow \text{10 depth}}}{\text{Freq}(B)}$

3. Atribuir as variáveis com maior benefício (R-viável) aos registos

ALOCADOR BOTTOM-UP

1. Iterar pelas instruções de um bloco básico
2. Alocar o valor do operando, se não já num registo
3. Alocar registo para o resultado
4. Quando sem registo:
 - a. libertar o registo cujo valor é usado mais longe no futuro
 - b. o valor do registo não requer operação de memória para atualizar o armazenamento

Bloco Básico: seqüência máxima de instruções tal que (1) só a primeira instrução pode ser alcançada a partir de fora do bloco básico e (2) todas as instruções são executadas consecutivamente \wedge e só se a primeira instrução for executada

- saltos só na última instrução
- etiquetas só na primeira instrução

Detectar Instruções Líderes:

1. primeira instrução de um programa
2. alvos de "goto"
3. instruções imediatamente após "goto"

Determinar Instruções de Bloco: para cada instrução líder, o seu bloco básico consiste na instrução líder e todas as instruções até à próxima instrução líder, excluindo-a

ALOCADOR GLOBAL

1. Determinar intervalos em que cada variável pode beneficiar ao usar um registo - redes
2. Determinar quais destes intervalos se sobrepõem - interferência
3. Encontrar o benefício de manter cada rede num registo - custo
4. Decidir que redes obtêm um registo - alocação
5. Separar redes se necessário - separação
6. Atribuir registo físicos a redes - atribuição
7. Gerar código

Redes: na mesma rede, estão todas as definições que alcançam um uso e todos os usos de uma definição - Cadeias DEF-USE com uso comum

- Nenhum valor tem de ser transportado entre redes
- A rede é usada como a unidade de alocação de registo

Interferência: duas redes interferem se os seus intervalos de vida se sobrepõem no tempo - existe uma instrução comum a ambos os intervalos em que os valores das variáveis das redes não operando dessa instrução

- Redes que não interferem podem ser atribuídas ao mesmo registo

Grafo de Interferência: os nós são as redes e existe uma aresta entre dois nós se elas interferem - cada nó obtém um registo (cor), pelo que se dois nós têm uma aresta entre eles, então não podem ter a mesma cor

Coloração do Grafo: qual é o número mínimo de cores necessárias para colorir os nós de um grafo de modo que nós ligados por uma aresta não tenham a mesma cor?

↓
HEURÍSTICA: N cores

1. Remover os nós com grau $< N$ e colocá-los numa filha
2. Se todos os nós tem grau $\geq N$, encontrar um nó para "Mill" (sem cor) e removê-lo
3. Quando vazio, começar a colorir:
 - a. Retirar um nó da filha
 - b. Atribuir-lhe uma cor diferente dos seus nós ligados (exceto porque grau $< N$)

Quando o grafo não é N -colorável?

1. Selecionar a rede menos custosa para romper
2. Separar essa rede em múltiplas redes de modo que exista menos interferência, tornando o grafo N -colorável - guardar o valor em memória e carregá-lo de volta nos pontos em que a rede é rompida

Separação:

1. Identificar um ponto do programa em que o grafo não é R -colorável
2. Escolher uma rede que não é usada pelo maior bloco em torno desse ponto do programa
3. Separar a rede
4. Refazer o grafo de interferência
5. Tentar recolorir o grafo

Amálise do Fluxo de Controlo

Otimizações:

1. Propagação de Constantes: $x = 10; y = x + 1 \rightarrow y = 10 + 1;$
2. Simplificação Algebrica: $x = 0 \times 1 \rightarrow x = 0;$
3. Propagação de Copias: $x = x; \rightarrow$
4. Eliminação de Sub-Expresões Comuns: $x = (i+1) * (i+1); \rightarrow y = i+1; k = y * y;$
5. Eliminação de Código Morto: $\text{int } y; \rightarrow$
6. Remoção de Invariante de Ciclo: $\text{for } (...) x = 1; \rightarrow x = 1; \text{ for } (...)$
7. Redução de Força: $x * 4; \rightarrow x \ll 2;$
8. Alocação de Registros

Propagação de Constantes: encontrar uma expressão do lado direito que seja constante e substituir o uso da variável do lado esquerdo com a constante do lado direito dando que (1) todos os caminhos ao(s) uso(s) da variável do lado esquerdo passam pela atribuição do lado esquerdo com a constante e (2) não existe definição intermédia da variável do lado direito

Grafo de Fluxo de Controlo: os nós são os blocos básicos e (x, y) é aresta se e só se a primeira instrução do bloco básico y segue a última instrução do bloco básico x

Dominadores: o nó X domina o nó Y se todos os caminhos de execução possíveis a partir da entrada até ao nó Y incluem o nó X

Influência: imagine-se uma fonte de luz no nó inicial, para encontrar os nós dominados por A , colocar uma barreira opaca em A e observar que nós ficam escuros

$a \text{ dom } b \leftrightarrow \begin{cases} \text{ou } a = b \\ \text{ou } a \text{ é o único antecessor imediato de } b \\ \text{ou } a \text{ é um dominador de todos os antecessores imediatos de } b \end{cases}$

Algoritmo:

1. Construir o conjunto de dominadores do nó de entrada como ele próprio
2. Construir o conjunto de dominadores dos nós restantes como todos os nós do grafo
3. Visitar os nós por qualquer ordem
4. Tomar o conjunto de dominadores do nó atual a interseção dos conjuntos de dominadores dos nós antecessores + o nó atual
5. Repetir até não haver alterações

Aresta para Trás: (x, y) é uma aresta para trás se e só se $y \text{ dom } x$

• Num CFG, uma aresta para trás indica um ciclo natural

Como encontrar os nós de um ciclo? Dada uma aresta para trás (s, t) , atraçam para trás (contra o fluxo) desde t até chegar a s e colecionar os nós atraçados

Fronteira de Dominância: de um bloco básico N , $DF(N)$ é o conjunto de todos os blocos que são antecessores imediatos para blocos dominados por N , mas eles próprios não são dominados estritamente por N

Forma de Atribuição Estática Única (SSA): cada definição tem um nome de variável único (nome original + número da versão)

- função ϕ para resolver múltiplas definições plausíveis
- Quando um bloco N define x , coloca um nó ϕ por x em cada bloco na fronteira de dominância de N

Otimização de Ciclos

- Se uma computação produz o mesmo valor em cada iteração do ciclo, move-a para fora do ciclo
- Uma expressão pode ser movida para fora do ciclo se todos os seus operандos são invariantes no ciclo

Operandos Invariantes: valores constantes **OU** variáveis cuja definição está fora do ciclo **OU** operando com uma só definição e essa definição é invariante do ciclo

- Uma instrução pode ser movida se:
 1. Todos os usos não dominados pela instrução
 2. A saída do ciclo é dominada pela instrução
 3. Definições estão fora do ciclo

Como lidar com ciclos aninhados? processar do mais interno para o mais externo

Como detectar a computação de invariantes de ciclo?

1. Comptar as definições alcançáveis para cada variável num bloco bálico
2. Marcar uma instrução $s: a = b + c$ como invariante se todas as definições de b e c que podem alcançar essa instrução s estão fora do ciclo
3. Repetir a marcação como invariantes até não haver alterações

Algoritmo de Movimentação de Código:

1. Computar definições alcançáveis
2. Computar computações de invariantes do ciclo
3. Computar dominadores
4. Encontrar as saídas do ciclo - nós com sucessores fora do ciclo
5. Encontrar candidatos:
 - a. invariantes do ciclo em blocos que dominam todas as saídas do ciclo
 - b. atribuição a variável não atribuída em nenhuma outra parte do ciclo em blocos que dominam todos os blocos no ciclo que usam a variável atribuída
6. Realizar uma pesquisa em profundidade dos blocos
7. Mover um candidato para antes do cálculo se todas as operações invariantes de que ele depende tiverem sido movidas

Variável de Indução: variável cujo valor altera em cada iteração e é incrementada/decrementada de uma quantidade constante

Variável de Indução Básica: $x = x + k$

Variável de Indução Derivada: função linear de uma variável de indução básica

Como detectar? $(i, c, d) \Leftrightarrow j = i + c + d$

1. Encontrar as variáveis de indução básicas ao fazer scan do ciclo tal que cada variável de indução básica tem $(i, 1, b)$
2. Procurar por variáveis K tais que: $K = j * b$; $K = b * j$; $K = j/b$; $K = j - j$, sendo b uma constante e j uma variável de indução básica
3. Verificar se a atribuição domina os pontos de definição para j

A ideia é remover/transformar variáveis de indução

Análise do Fluxo de Dados

Expresão Disponível: no ponto P se e só se (1) todos os caminhos de execução que alcançam o ponto P passam pelo ponto em que a expressão foi definida e (2) nenhuma variável usada na expressão foi modificada entre o ponto da definição e o ponto atual P — a expressão ainda é atual em P

Conjunto GEN: se uma instrução define a expressão, então o número da expressão está no conjunto GEN para essa instrução

Conjunto KILL: se uma instrução (re) define uma variável na expressão, então o número da expressão está no conjunto KILL para essa instrução — a expressão é inválida depois dessa instrução

$$\text{Conjunto Out GEN} = \text{GEN} \cup (\text{In GEN} - \text{KILL})$$

$$\text{Conjunto Out KILL} = \text{KILL} \cup \text{In Kill}$$

$$\text{Conjunto OUT} = \text{GEN} \cup (\text{IN} - \text{KILL})$$

$$\text{Conjunto IN} = \cap \text{OUT}$$

Algoritmo para Expressões Disponíveis:

1. Atribuir um número a cada expressão no programa
2. Computar os conjuntos GEN e KILL para cada instrução
3. Computar os conjuntos GEN e KILL agregados para cada bloco básico — propagar os conjuntos anteriores desde o início até ao fim de cada bloco
4. Inicializar os conjuntos disponíveis em cada bloco básico — IN e OUT (no universo de expressões, vazio para o primeiro bloco em que $\text{IN} = \emptyset$)
5. Propagar iterativamente os conjuntos de expressões disponíveis sobre o CFG até alcançar uma solução
6. Propagar a solução pelo bloco básico

Formulação Iterativa

Abordagem Geral:

1. Construir o grafo de fluxo de controle - definir antecessores e sucessores
2. Calcular conjuntos GEN e KILL para cada bloco básico
3. Propagar iterativamente a informação até convergência
4. Propagar informação dentro do bloco básico

(Semi-)Lattice: conjunto L

- quantidades abstratas V sobre as quais a análise vai operar
- uma/das operações sobre os valores de V : meet (\wedge) e join (\vee)
- um valor de topo (T) e um valor de fundo (\perp)

Função de Transferência: como cada instrução e construção de fluxo de controle afeta as quantidades abstratas V — $F: V \rightarrow V$

Propriedades: $\forall a, b, c \in L$

1. $a \wedge a = a$
2. $a \wedge b = b \wedge a$
3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
4. $a \geq b \Leftrightarrow a \wedge b = b$
5. $a > b \Leftrightarrow a > b \text{ e } a \neq b$
6. $\perp \wedge a = \perp$
7. $a > \perp$

REFLEXIVO 8. $x \leq x$

ANTISSIMÉTRICO 9. $x \leq y, y \leq x \rightarrow x = y$

TRANSITIVO 10. $x \leq y, y \leq z \rightarrow x \leq z$

Maior

Límite Inferior de "Meet Semi-Lattice": $g = x \wedge y$ é o maior limite inferior de x e y se e só se $g \leq x$, $g \leq y$, $\forall z: g \leq x, g \leq y \rightarrow g \leq z$

Menor

Límite Superior de "Join Semi-Lattice": $b = x \vee y$ é o menor limite superior de x e y se e só se $x \leq b$, $y \leq b$, $\forall z: x \leq z, y \leq z \rightarrow b \leq z$

• Meet e Join formam um fecho em L porque $\forall a, b \in L$, existem únicos $c \in L$ e $d \in L$ tais que $a \wedge b = c$ e $a \vee b = d$

Problema das Definições Alcancáveis: uma definição D de uma variável V alcança uma instrução I se e só se a instrução I lê V e não existe caminhos desde D até I que (re)define V

Inicialização: $\text{Reaches}(n) = \emptyset$, $\forall n$

Equação: $\text{Reaches}(n) = \bigcup_{m \in \text{pred}(n)} (\text{DEDef}(m) \cup (\text{Reaches}(m) \cap \text{DefKill}(m)))$

DEDef_m: conjunto de definições que não são redefinidas posteriormente

DefKill_m: todos os pontos de definição que não ocorrem por uma definição da mesma variável V em m

Monotonicidade: $\forall x, y \in V, f \in F: x \leq y \Rightarrow f(x) \leq f(y) \quad \text{e} \quad f(x \wedge y) \leq f(x) \wedge f(y)$

Distributividade: $\forall x, y \in V, f \in F: f(x \wedge y) = f(x) \wedge f(y)$

Término: se a semi-lattice é monótona e de altura finita

Correção: se o algoritmo converge

A análise de fluxo de dados começa por assumir os valores mais otimistas ou conservadores (1) e, em cada fase, aplica uma função de fluxo em que $V_{\text{new}} \subseteq V_{\text{prev}}$ (move-se para baixo ou para cima na malha), até que os valores sejam estáveis (não se alterem).

Solução IDEAL: encontra-se as traiçais e aplica a função de transferência a todos os caminhos de execução possíveis desde a entrada do programa até ao inicio do bloco básico B — $\text{IDEAL}[B] = \bigwedge_{\text{caminhos possíveis } p} f_p(v_{\text{inicial}})$

Solução Meet-Over-All-Paths (MOP): $\text{MOP}[B] = \bigwedge_{\text{caminhos possíveis } p} f_p(v_{\text{final}})$

iniciável \leftarrow MOP \Leftarrow IDEAL \rightarrow indecidível

Solução Maximal-Fixed-Point (MFP): se as funções de transferência são distributivas, então $\text{MFP} = \text{MOP}$

Limitações: precisão; volatilidade; arrays; apontadores

Pós-Ordem Reversa: para cada nó, todos os seus antecessores não computados antes — os dados ainda fluem para a frente

Pré-Ordem Reversa: para cada nó, todos os seus sucessores não computados antes — os dados ainda fluem para trás

Cadeia Def-Use: liga uma definição de cada variável v_k a todos os usos possíveis dessa variável

Cadeia Use-Def: liga um uso de cada variável v_k a todas as definições possíveis dessa variável

RESUMO

Domínio: conjunto de expressões

Direção do Fluxo de Dados: para a frente - valores de saída computados com base nos valores de entrada

Valores Iniciais: conjunto vazio

$$OUT = GEN \cup (IN - KILL)$$

Expressões Disponíveis:

- $GEN = \{ \text{exp} \mid \text{exp é calculada no bloco básico} \}$
- $KILL = \{ \text{exp} \mid \exists v \in \text{exp}, v \text{ é definida no bloco básico} \}$
- $IN = \bigcap OUT$ para todos os predecessores de um bloco básico

Definições Alcancáveis:

- $GEN = \{ x \mid x \text{ é definido no bloco básico} \}$
- $KILL = \{ x \mid \text{lado esquerdo de } x \text{ é redefinido no bloco básico} \}$
- $IN = \bigcup OUT$ para todos os predecessores de um bloco básico

Otimizações Tradicionais

Simplificação Algebrica: aplicar os conhecimentos de álgebra e Teoria dos números para simplificar expressões

Propagação de Cópias: contornar múltiplas cópias - propagar um valor diretamente para o seu uso
↓ COMO?

• Em cada expressão de lado direito, para cada variável v usada nessa expressão do lado direito, se a variável v é definida por uma instrução da forma $v = u$, então substituir a variável v por u

• Uma atribuição de $v = u$ ainda é válida num dado ponto de execução se e só se:

1. Uma instrução de $v = u$ ocorre em todos os caminhos de execução que chegam ao ponto atual
2. A variável v não é redefinida em nenhum desses caminhos de execução entre a instrução de atribuição e o ponto atual
3. A variável u não é redefinida em nenhum desses caminhos de execução entre a instrução de atribuição e o ponto atual

Domínio: conjunto de tuplos (v, u) que representam uma equivalência $v = u$

Direção do Fluxo de Dados: para a frente

Valores Iniciais: conjunto vazio

- $IN = \emptyset$
- $OUT = Gen \cup (IN - Kill)$
- $Gen = \{ (v, u) \mid v = u \text{ é a instrução} \}$
- $Kill = \{ (v, u) \mid \text{variável do lado esquerdo de uma atribuição é } v \text{ ou } u \}$

Propagação de Constantes: usar a constante conhecida de uma variável v

COMO?

- Em cada expansão de lado direito, para cada variável v usada nessa expansão do lado direito, se a variável v é uma constante conhecida K , então substituir a variável v por K

- Uma variável v é a constante K num ponto de execução xc e r0

se:

1. A instrução atual é $v = K$

OU

2. Todos os caminhos até ao ponto atual têm K atribuído a v

Domínio: para cada variável, uma malha L_v

Direção do Fluxo de Dados: para a frente

- OUT = Gen \wedge (IN \vee Piso)

$$\cdot \text{Gen} = \left\{ k_v \mid k_v = \begin{cases} 1, & \text{se } v \text{ não está do lado esquerdo} \\ K, & \text{se } v \text{ é o lado esquerdo e o lado direito é } K \\ 1, & \text{caso contrário} \end{cases} \right\}$$

$$\cdot \text{Piso} = \left\{ k_v \mid k_v = \begin{cases} 1, & \text{se } v \text{ é o lado esquerdo} \\ 1, & \text{se } v \text{ não é o lado esquerdo} \end{cases} \right\}$$

Analise da Vida de Variáveis

- Para cada variável x , onde é o último ponto P do programa em que um valor específico de x é usado?
- Para cada variável x e ponto P do programa, determinar se o valor de x em P ainda pode ser usado a partir de algum caminho que comece em P - se sim, x está vivo em P ; se não, x está morto em P
- Uma variável está viva num ponto P se o seu valor é usado em pelo menos um caminho

Gen: conjunto de variáveis usadas no bloco básico B Use

Kill: conjunto de variáveis definidas no bloco básico B Def

$$OUT(B) = \bigcup_{S \in \text{succesors of } B} IN(S)$$

$$IN(B) = \text{Use}(B) \cup (OUT(B) - \text{Def}(B))$$

inicializar vazio

→ fluxo para trás

```
// Assume instruction in format "x ← y op z"  
for i ← 1 to Num Instructions in B do  
    if (instr(i) is leader of B) then  
        b ← Number(B);  
        UpExp(b) ← ∅;  
        VarKill(b) ← ∅;  
        if y ∉ VarKill(b) then  
            UpExp(b) ← UpExp(b) ∪ {y}  
        if z ∉ VarKill(b) then  
            UpExp(b) ← UpExp(b) ∪ {z}  
            VarKill(b) ← VarKill(b) ∪ {x}
```

RESUMO

Expresões Disponíveis:

Domínio: conjunto de expressões

Fluxo: para a frente - pós-ordem reversa

Valores Iniciais: conjunto vazio

Funções:

$$\rightarrow \text{OUT} = \text{gen} \cup (\text{IN} - \text{Kill})$$

$$\rightarrow \text{gen} = \left\{ \begin{array}{l} \text{exp} \\ \text{exp} \end{array} \mid \begin{array}{l} \text{exp é calculada no bloco básico} \\ \exists \text{ var } v \in \text{exp} : v \text{ é definida no bloco básico} \end{array} \right\}$$

$$\rightarrow \text{Kill} = \left\{ \begin{array}{l} \text{exp} \\ \exists \text{ var } v \in \text{exp} : v \text{ é definida no bloco básico} \end{array} \right\}$$

Operação "Meet": $\text{IN} = \cap \text{OUT}$

Definições Alcancáveis / Cadeias Def-Ute:

Domínio: conjunto de definições

Fluxo: para a frente - pós-ordem reversa

Valores Iniciais: conjunto vazio

Funções:

$$\rightarrow \text{OUT} = \text{gen} \cup (\text{IN} - \text{Kill})$$

$$\rightarrow \text{gen} = \left\{ \begin{array}{l} x \\ x \end{array} \mid \begin{array}{l} x \text{ é definido no bloco básico} \\ \exists \text{ variável do lado esquerdo de } x \text{ é redefinida no bloco básico} \end{array} \right\}$$

$$\rightarrow \text{Kill} = \left\{ \begin{array}{l} x \\ x \end{array} \mid \begin{array}{l} \exists \text{ variável do lado esquerdo de } x \text{ é redefinida no bloco básico} \end{array} \right\}$$

Operação "Meet": $\text{IN} = \cup \text{OUT}$

RESUMO

Propagação de Cópias:

Domínio: conjunto de tuplos $\langle v, u \rangle$ que representam uma equivalência $v = u$

Fluxo: para a frente - pós-ordem reversa

Valores Iniciais: conjunto vazio

Funções:

$$\rightarrow \text{OUT} = \text{gen} \cup (\text{IN} - \text{Kill})$$

$$\rightarrow \text{gen} = \{ \langle v, u \rangle \mid v = u \text{ é a instrução} \}$$

$$\rightarrow \text{Kill} = \{ \langle v, u \rangle \mid \text{a variável do lado esquerdo de uma atribuição é } v \text{ ou } u \}$$

Operação "Meet": $\text{IN} = \cap \text{OUT}$

Propagação de Constantes:

Domínio: para cada variável, uma malha ("lattice") L_v

Fluxo: para a frente - pós-ordem reversa

Valores Iniciais: conjunto vazio

Funções:

$$\rightarrow \text{OUT} = \text{gen} \wedge (\text{IN} \vee \text{prev})$$

$$\rightarrow \text{gen} = \{ x_v \mid x_v = \begin{cases} T, & \text{se } v \text{ não é lado esquerdo} \\ \text{valor, se } v \text{ é o lado esquerdo e o lado direito é constante} \\ I, & \text{caso contrário} \end{cases} \}$$

$$\rightarrow \text{prev} = \{ x_v \mid x_v = \begin{cases} T, & \text{se } v \text{ é o lado esquerdo} \\ I, & \text{se } v \text{ não é o lado esquerdo} \end{cases} \}$$

RESUMO

Análise de Vida de Variáveis:

Dominio: conjunto de variáveis

Fluxo: para trás - pré-ordem reversa

Valores Iniciais: conjunto vazio

Funções:

$$\rightarrow IN = \text{gen} \cup (\text{OUT} - \text{Kill})$$

$$\rightarrow \text{gen} = \{ x \mid x \text{ é usado no bloco básico} \}$$

$$\rightarrow \text{Kill} = \{ x \mid x \text{ é definido no bloco básico} \}$$

Operação "Meet": $\text{OUT} = \text{U} \text{IN}$