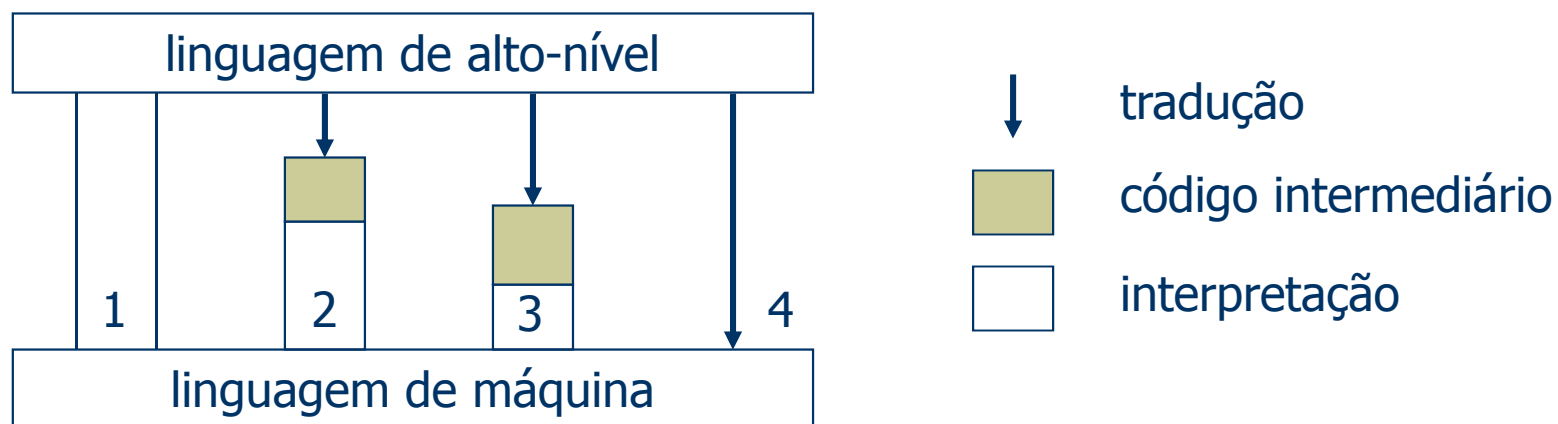


2.9 - Intérpretes

- ♦ Em geral, a implementação de uma linguagem de programação se dá de uma das seguintes maneiras:



- 1) Interpretação direta do programa fonte. O texto é inspecionado caractere a caractere e as ações correspondentes são tomadas (por exemplo, quando a sequência de caracteres "G", "O", "T", "O", " ", "L" é encontrada, o intérprete deve procurar no texto fonte por um rótulo L e ...). O uso mais comum de sistemas desse tipo (são muito ineficientes) é no processamento de macros.
- 2) Interpretação de código intermediário de alto-nível. Nesses sistemas, como o código interpretado é de alto-nível, consegue-se grande flexibilidade de execução (não dependente de máquina). Além disso, é fácil prover facilidades de diagnóstico e depuração na linguagem fonte.
- 3) Interpretação de código intermediário de baixo-nível. Perde-se flexibilidade de execução (dependente de máquina) mas ganha-se eficiência.

Intérpretes

- 4) Interpretação de linguagem de máquina. Esses sistemas podem ser vistos como do tipo 3, em que o código intermediário pode ser interpretado (ou seja, executado) diretamente pelo computador, isto é, a interpretação é feita num nível suportado pelo "hardware".
- ◆ Nesse capítulo estaremos interessados em sistemas do tipo 2, isto é, o intérprete será visto como um programa que executa um programa representado por um código intermediário, que deve ser escolhido de forma a tornar a execução tão eficiente quanto possível.
- ◆ Algumas razões:
 - a) Sistemas do tipo 2 são mais portáteis que sistemas dos tipos 3 ou 4 porque são programas mais simples, menores e não dependem de linguagem de máquina (a geração de "assembly" ou de código de máquina, evidentemente, depende da máquina alvo).
 - b) A relação (custo de software)/(custo de hardware) tende a crescer, o que implica que a importância da "portabilidade do software" tende a aumentar e a importância da "eficiência do hardware" tende a diminuir.
 - c) Em linguagens de "muito alto-nível" as operações primitivas são tão complexas a ponto de tornar irrelevante (com relação ao tempo de execução) se elas são "interpretadas" ou "compiladas". Sendo esse o caso, é preferível fazer uso das vantagens da interpretação (flexibilidade, ferramentas de depuração, ...).

Intérpretes

Esquema de interpretação

- ♦ Vamos supor que a forma interna do programa está escrita na notação polonesa e armazenada em um vetor P. Durante a interpretação, um inteiro i indica qual a instrução (em P) deve ser executada. Para executar o programa vamos admitir a existência de uma pilha S, com t apontando para o topo. Inicialmente a pilha está vazia (t = 0).
- ♦ Vamos estruturar o intérprete em termos de um comando "case" onde a ação depende de P[i]. Seja a seguinte convenção:

código	significado
1	constante (usa duas posições em P)
2	identificador (usa duas posições em P)
3	SUBS (operador de indexação)
4	:=
5	BLZ (desvio se ≤ 0)
6	BR (desvio incondicional)
7	BRL (desvio para rótulo)
8	BLOCK
9	BLOCKEND
10	ADEC
11	+
12	*
13	-

Intérpretes

Temos então:

```
next:
case P[i] of
  1: i := i+1; PUSH(P[i]);
  2: i := i+1; PUSH(P[i]);
  3: execute_indexação;
  4: execute_atribuição;
  5: t := t-2;
    if S[t+1] ≤ 0 then
    begin
      i := S[t+2];
      goto next;
    end;
  6: t := t-1; i := S[t+1]; goto next;
  7: execute_desvio_para_rótulo;
  ...
  13: execute_subtração;
end_case;
i := i+1;
goto next;
```

Intérpretes

- ♦ Em qualquer instante, podemos ter na pilha S:
 - 1) um valor (como resultado, por exemplo, da operação $A \ B \ +$)
 - 2) um ponteiro para a tabela de símbolos (por exemplo, no caso $1 \ 2 \ 3 \ A \ SUBS...$, onde A deve ser um ponteiro para que SUBS possa determinar o número de subscritos e determinar o endereço da variável)
 - 3) um endereço (por exemplo, no caso $1 \ A \ :=$, onde A deve ser um endereço de variável)
- ♦ Para distinguir cada um desses casos, vamos imaginar que os elementos da pilha S são constituídos por dois campos: KIND (1, 2 ou 3) e VALUE. Cada operador, ao ser executado, deve transformar seus operandos para o KIND correto, caso seja necessário.
- ♦ Exemplo: Operação *
 - a) se $S[t].KIND = 2$ então colocar o valor da variável descrita na tabela de símbolos pelo endereço $S[t].VALUE$ em $S[t].VALUE$;
 - b) se $S[t].KIND = 3$ então colocar o valor da variável de endereço $S[t].VALUE$ em $S[t].VALUE$;
 - c) idem para $S[t-1]$
 - d) $t := t-1$;
 $S[t].VALUE := S[t].VALUE * S[t+1].VALUE$;
 $S[t].KIND := 1$;

Intérpretes

- ♦ Esse método toma muito tempo de interpretação. Se, no tempo da tradução, for possível determinar o KIND de cada operando e o KIND do resultado, pode-se inserir operadores de conversão convenientes na forma intermediária, de maneira que quando um operador for executado, seus operandos já tem os KINDs corretos.
- ♦ Sejam os seguintes operadores de conversão:
 - CVPV converte de ponteiro para valor
 - CVPA converte de ponteiro para endereço
 - CVAV converte de endereço para valore, para um identificador I qualquer, a seguinte notação:
 - P:I empilha ponteiro para a entrada de I na tabela de símbolos
 - A:I empilha endereço da variável I
 - V:I empilha valor da variável I
- ♦ Exemplo:
C := B + A(I), que normalmente é escrito como: C B I A SUBS + :=, será escrito como:

A:C V:B V:I P:A SUBS CVAV + :=
- ♦ Esse método é, geralmente, o preferido (não deixar para o tempo de execução o que pode ser feito em tempo de compilação). Nesse caso também, não há mais necessidade do campo KIND.

Intérpretes

Conversão de tipos

- ♦ Existem também duas formas de conversão de operandos de um tipo para outro:
 - a) ter um campo em cada elemento da pilha para indicar o tipo do valor (inteiro, real, ...) e deixar cada operador verificar esse campo e efetuar as conversões necessárias;
 - b) descrever as conversões necessárias explicitamente.
- ♦ Vamos supor os tipos inteiro e real. Sejam I,J variáveis inteiras e A,B variáveis reais. Se um comando como: $I := A*B + J$ é escrito como: $I \ A \ B \ * \ J \ + \ :=$, vamos precisar de um campo TYPE para indicar o tipo de um elemento da pilha. Será preferível, no entanto, uma representação como:

$I \ A \ B \ * \ J \ CVIR \ + \ CVRI \ :=$

onde: CVIR operador de conversão de inteiro para real

 CVRI operador de conversão de real para inteiro

- ♦ Nesse caso, é interessante ter também operadores para tipos específicos, por exemplo, $*_R$ (multiplicação real), $:=_I$ (atribuição inteira), ... Assim, para o caso acima teremos:

$I \ A \ B \ *_R \ J \ CVIR \ +_R \ CVRI \ :=_I$

Intérpretes

Facilidades de depuração em tempo de execução

- ♦ A disponibilidade da tabela de símbolos no tempo de execução (interpretação) torna fácil prover as seguintes informações para a depuração:
 - a) "Dump" da tabela de símbolos
É razoavelmente simples implementar um procedimento PRINTVALUE(P) que, dado o endereço P de uma entrada da tabela de símbolos, imprime o identificador do programa fonte associado, juntamente com seu valor (inteiro, real, ...). O ponto importante aqui é que o "dump" é simbólico, feito no mesmo nível do programa fonte, em vez de um "dump" binário ou hexadecimal.

Se a linguagem não é recursiva, os valores correntes para as variáveis simples podem ser armazenados na própria tabela de símbolos, o que facilita a tarefa de PRINTVALUE. Para linguagens com estrutura de bloco e procedimentos recursivos, um "dump" simbólico de todas as variáveis e seus valores correntes também pode ser feito: em qualquer ponto da interpretação devemos ser capazes de determinar o bloco que está sendo executado. Isso pode ser feito inserindo o número do bloco junto ao operador BLOCKEND na forma intermediária. Assim, para encontrar o bloco no qual um símbolo $P[i]$ ocorre, percorremos os símbolos $P[i+1]$, $P[i+2]$, ... até que seja encontrado um operador BLOCKEND.

Intérpretes

b) Localização do erro no programa fonte

Mensagens de erro devem conter o número da linha do texto fonte na qual o erro ocorreu. Isso pode ser implementado facilmente inserindo operadores especiais (essencialmente NOOPS) na forma intermediária: cada operador é seguido pelo número da linha correspondente e podem ser usados como descrito para os operadores BLOCKEND.

c) "Trace"

Seja uma rotina TRACE(A) que, quando ativada, irá exibir o valor de A toda vez que um valor é atribuído a A. Para implementar um "trace" basta introduzir um campo em cada entrada da tabela de símbolos para "ligar" ou "desligar" o "trace" e fazer com que o operador := verifique esse campo para saber se chama ou não PRINTVALUE.