Capítulo 1: Fases de compilação

Aarne Ranta

Slides do livro ”Implementing Programming Languages. Uma Introdução aos Compiladores e Intérpretes ”, Publicações da faculdade,

2012

Fases de Compilação

Fases no caminho do código-fonte ao código de máquina

Conceitos e terminologia para discussões posteriores

Compiladores x intérpretes

Linguagens de baixo e alto nível

Estruturas de dados e algoritmos na implementação de linguagem

Da linguagem ao binário

Máquinas manipulam bits: 0's e 1's.

Sequências de bits usado em codificação binária.

Informação = sequências de bits

Codificação binária de inteiros:

0 = 0

1 = 1

2 = 10

3 = 11

4 = 100

Codificação binária de letras, via codificação ASCII:

A = 65 = 1000001 B = 66 = 1000010 C = 67 = 1000011

Assim todos dados manipulado por computadores pode ser expresso por 0's e 1's.

Mas e quanto programas?

Codificação binária de instruções

Por exemplo, linguagem de máquina JVM (Máquina Virtual JAVA)

Programas são sequências de bytes - grupos de oito 0s ou 1s (há 256 deles)

Um byte pode codificar um valor numérico, mas também um instrução

Exemplos: adição e multiplicação (de inteiros)

+ = 96 = 0110 0000 = 60

* = 104 = 0110 1000 = 68

(A última figura é um hexadecimal, onde cada meio-byte é codificado por um dígito de base 16 que varia de 0 a F, com A = 10, B = 11 ,. . . , F = 15.)

Fórmulas aritméticas

Simplório in fi xo:

5 + 6 = 0000 0101 0110 0000 0000 0110

Usos reais da JVM pós consertar

5 + 6 = ⇒ 5 6 +

Sem necessidade de parênteses:

(5 + 6) * 7 = ⇒ 5 6 + 7 *

5 + (6 * 7) = ⇒ 5 6 7 * +

Empilhar máquinas

JVM manipula expressões com um pilha - a memória de trabalho da máquina

Os valores (sequências de bytes - 4 bytes em uma máquina de 32 bits) são empurrado na pilha

O último empurrado é o topo da pilha

Uma operação aritmética como + (geralmente chamada de "adicionar") leva (pops)
os dois elementos principais e empurra sua soma

Exemplo: calcule 5 + 6 (instruções à esquerda, pilha à direita)

```
bipush 5        ; 5
bipush 6        ; 5 6
Eu adiciono     ; 11
```

As instruções são mostradas como código de montagem, nomes legíveis por humanos para código de bytes.

Um exemplo mais complexo: o cálculo de 5 + (6 * 7)

```
bipush 5        ; 5
bipush 6        ; 5 6
bipush 7        ; 5 6 7
imul            ; 5 42
Eu adiciono     ; 47
```

No final, há sempre apenas um valor na pilha

Separando valores de instruções

Para deixar claro que um byte representa um valor numérico, ele é prefixado com a instrução bipush

5 + 6 = ⇒ bipush 5 bipush 6 iadd

Para converter tudo isso em binário, precisamos apenas do código para a instrução push,

bipush = 16 = 0001 0000

Agora podemos expressar toda a expressão aritmética como binária:

5 + 6 = 0001 0000 0000 0101 0001 0000 0000 0110 0110 0000

Por que os compiladores funcionam?

Tanto os dados quanto os programas podem ser expressos como código binário, ou seja, por 0 e 1.

Existe uma sistemática. tradução de expressões convencionais (“amigáveis”) a código binário.

É claro que precisaremos de mais instruções para representar variáveis, atribuições, loops, funções e outras construções encontradas em linguagens de programação, mas os princípios são os mesmos do exemplo simples acima.

Como funcionam os compiladores

1 Análise sintática: Analise a expressão em um operador *F* e seus operandos *X* e *Y.*

2 Tradução direcionada por sintaxe: Compile o código para *X,* seguido pelo código para *Y,* seguido pelo código para *F.*

Ambos usam recursão: são funções que chamam a si mesmas em partes da expressão.

Níveis de línguas

Um compilador pode ser mais ou menos exigente. Isso depende da distância dos idiomas entre os quais ele traduz. (Cf. Inglês para Francês é mais fácil do que Inglês para Japonês.)

Em linguagens de computador,

- Alto nível: mais perto do pensamento humano, mais difícil de compilar
- Nível baixo: mais perto da máquina, mais fácil de compilar

Isso não é um julgamento de valor, uma vez que linguagens de baixo nível são indispensáveis!

linguagem humana

ML Haskell

Lisp Prolog

C ++ Java

C

montador

linguagem de máquina

Algumas linguagens de programação do nível mais alto ao mais baixo.

Humanos e máquinas são necessários para fazer os computadores funcionarem da maneira como estamos acostumados.

Algumas pessoas podem alegar que apenas o nível mais baixo de código binário é necessário, porque os humanos podem ser treinados para escrevê-lo.

Mas os humanos nunca poderiam escrever programas muito sofisticados usando apenas código de máquina - eles simplesmente não podiam manter os milhões de bytes necessários em suas cabeças.

Portanto, normalmente é muito mais produtivo escrever código de alto nível e deixar um compilador produzir o binário.

A história das linguagens de programação mostra o progresso dos níveis inferiores aos superiores.

Os programadores podem ser mais produtivos ao escrever em linguagens de alto nível.

No entanto, aumentar o nível implica um desafio para os escritores do compilador.

Portanto, a evolução das linguagens de programação anda de mãos dadas com os desenvolvimentos na tecnologia de compiladores.

É claro que também ajudou o fato de as máquinas se tornarem mais poderosas:

- os computadores da década de 1960 não poderiam ter executado os compiladores da década de 2010
- é mais difícil escrever compiladores que produzem código e fi ciente do que aqueles que desperdiçam algum

Uma história aproximada de linguagens de programação

- 1940: conectando fios para representar 0's e 1's
- 1950: montadores, macro montadores, Fortran, COBOL, Lisp
- 1960: ALGOL, BCPL (→ B → C), SIMULA
- Década de 1970: Smalltalk, Prolog, ML
- Década de 1980: C ++, Perl, Python
- Década de 1990: Haskell, Java

Um compilador inverte a história das linguagens de programação: de uma linguagem de origem dos anos 1960:

```
5 + 6 * 7
```

para uma linguagem de montagem dos anos 1950

```
bipush 5 bipush 6 bipush 7 imul iadd
```

para uma linguagem de máquina dos anos 1940

```
0001 0000 0000 0101 0001 0000 0000 0110 0001 0000 0000
0111 0110 1000 0110 0000
```

A segunda etapa é muito fácil: procure os códigos binários para cada instrução de montagem e coloque-os juntos na mesma ordem.

O nível de montagem é frequentemente considerado como separado da compilação adequada.

Compilação vs. interpretação

Um compilador é um programa que traduz código para algum outro código. Ele executa o programa.

A intérprete não traduz, mas ~~isso~~ executa o programa.

Uma expressão da linguagem de origem,

```
5 + 6 * 7
```

é por um intérprete voltado para seu valor,

```
47
```

Combinações

- C geralmente é compilado para código de máquina pelo GCC.
- Java é geralmente compilado para o bytecode JVM por Javac, e este bytecode é geralmente interpretado, embora partes dele possam ser compiladas para código de máquina por JIT (compilação just in time).
- JavaScript é interpretado em navegadores da web.
- Os scripts de shell do Unix são interpretados pelo shell.
- Haskell os programas são compilados para código de máquina usando GHC ou para bytecode interpretado em Hugs ou GHCI.

Aviso: Java não é uma “linguagem interpretada” - mas JVM é!

Trade-o ff s

Vantagens de interpretação:

- mais rápido para ir
- mais fácil de implementar
- portátil para máquinas diferentes

Vantagens da compilação:

- se para código de máquina: o código resultante é mais rápido de executar
- se ao código-alvo independente da máquina: o código resultante é mais fácil de interpretar do que o código-fonte

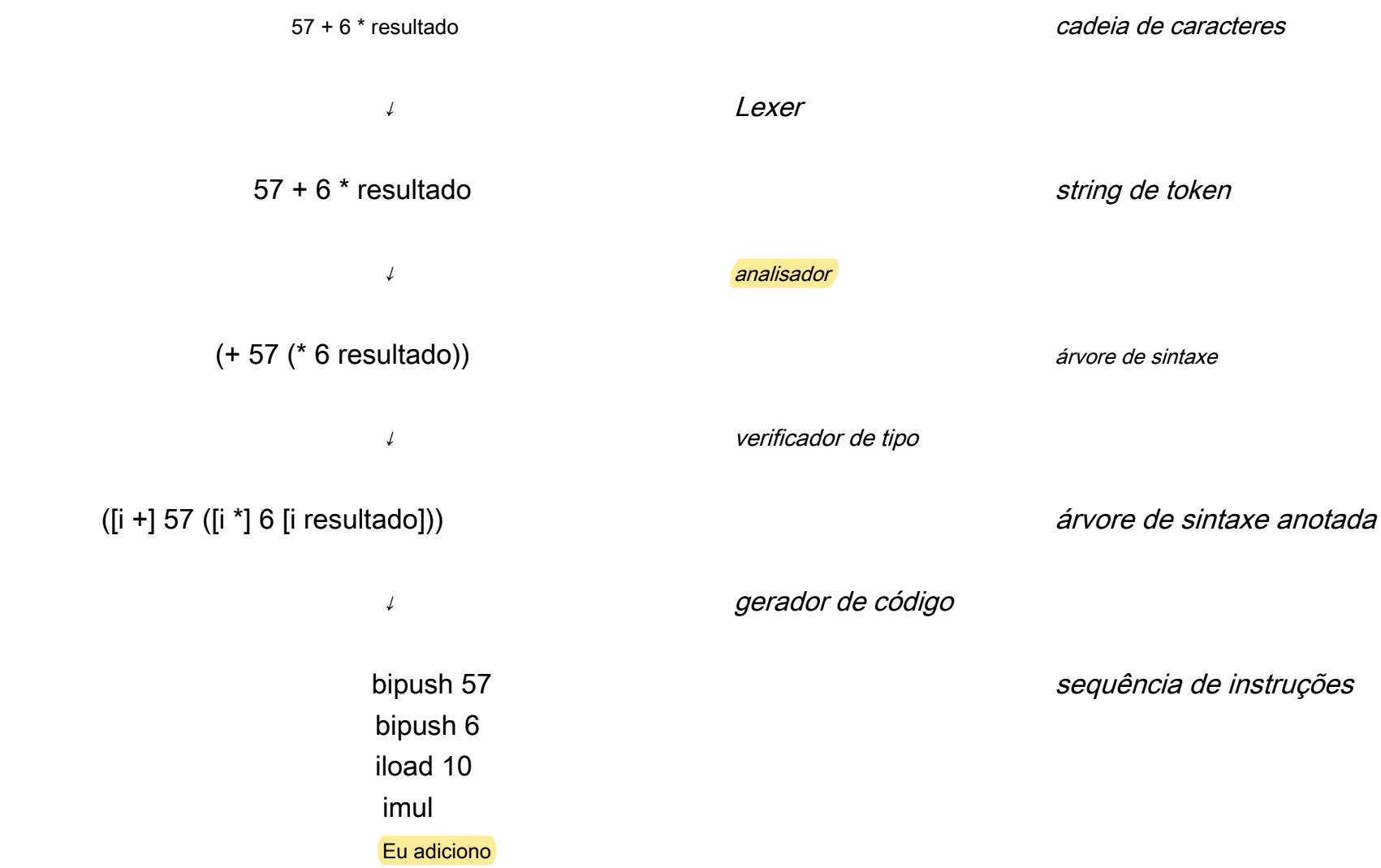
O JIT está obscurecendo a distinção, assim como as máquinas virtuais com conjuntos reais de instruções em linguagem de máquina, como VMWare e Parallels.

Fases de compilação

Um compilador é um programa complexo, que deve ser dividido em componentes menores.

Esses componentes normalmente tratam de diferentes fases de compilação -
partes de um pipeline, que transformam o código de um formato para outro.

O diagrama a seguir mostra as principais fases do compilador e como uma parte do código-fonte passa por elas.



Fases de compilação do código-fonte Java para o código de montagem JVM

- o Lexer lê uma seqüência de personagens e corta em tokens.
- o analisador lê uma sequência de tokens e os agrupa em um árvore de sintaxe.

- o verificador de tipo descobre o tipo de cada parte da árvore de sintaxe e retorna um árvore de sintaxe anotada.
- o gerador de código converte a árvore de sintaxe anotada em uma lista de instruções de código de destino.

A diferença entre compiladores e interpretadores está apenas na última fase: os interpretadores não geram novo código, mas executam o código antigo.

Erros de compilação

Cada fase do compilador pode falhar com erros característicos:

- Erros Lexer, por exemplo, cotação não fechada,

  "Olá

- Erros de análise, por exemplo, parênteses incompatíveis,

  (4 * (y + 5) - 12))

- Erros de digitação, por exemplo, a aplicação de uma função a um argumento de tipo errado,

  classificar (45)

Front end e back end

Front end: análise, isto é, inspeciona o programa: lexer, parser, type checker.

Fim de trás: síntese, ou seja, constrói algo novo: gerador de código.

Os erros em fases posteriores à verificação de tipo geralmente não são suportados; cf. Robin Milner (o criador do ML): ”Programas bem digitados não podem dar errado”.

Tempo de compilação vs. tempo de execução

Compiladores só podem encontrar tempo de compilação erros.

Detecção de erro em tempo de execução necessidades depuração.

Como a compilação é automática e a depuração é manual, esforços são feitos para encontrar mais erros em tempo de compilação.

Exemplos de erros de tempo de execução

Índice de matriz fora dos limites, se o índice for uma variável que obtém seu valor em tempo de execução.

Análise de ligação de variáveis:

```
int main () {
    int x;
    if (readInt ()) x = 1; printf ("% d", x);

}
```

Não é decidível em tempo de compilação se x tem um valor.

Mais fases de compilação

Desugaring / normalização: retirar açúcar sintático,

int i, j; = ⇒ int i; int j;

Isso pode ser feito no início (o que pode resultar em mensagens de erro piores).

Otimizações: melhorar o código em algum aspecto. Isto pode ser feito

- otimização do código-fonte, valores de pré-computação conhecidos em tempo de compilação:

  i = 5 + 6 * 7; = ⇒ i = 47;
- otimização do código alvo, substituindo as instruções por outras mais baratas:

  bipush 31 bipush 31 = ⇒ bipush 31 dup

(O ganho é que ~~o enganar~~ instrução é apenas um byte, enquanto
bipush 31 tem dois bytes.)

Os compiladores modernos podem ter dezenas de fases, muitas vezes realizadas no nível de código intermediário, para que o trabalho possa ser reutilizado para diferentes idiomas de origem e destino.

A teoria e a prática

fase	teoria
Lexer	autômatos finitos
analisador	gramáticas livres de contexto
verificador de tipo	sistemas de tipo
intérprete	semântica operacional
esquemas de compilação	de gerador de código

Uma teoria

- fornece notações declarativas que suportam implementações diferentes
- permite raciocínio, por exemplo, verificar se cada programa pode ser compilado de uma forma única

Tradução direcionada por sintaxe é um nome comum para as técnicas usadas em verificadores de tipo, interpretadores e geradores de código semelhantes.