Notas de aula - 11/02/2014

Introdução

1. O que são compiladores
   (a) Um programa que transforma um programa-fonte escrito em uma linguagem de programação em outra linguagem (tipicamente código de máquina).
   (b) Tradução de código de alto nível para código de mais baixo nível
2. Diferença entre compilador e interpretador
   (a) Interpretador é um programa que executa diretamente (isto é, realiza as ações) descritas em um programa-fonte sem antes compilá-lo.
   (b) Compilador é um passo "offline", interpretação é "online"
3. Mas há níveis de cinza aí
   (a) Um interpretador pode ir "lendo e executando" o fonte diretamente (exemplo: bash, interpretadores clássicos de Lisp e BASIC)
   (b) Um interpretador pode antes "compilar" o fonte para um formato intermediário e depois interpretá-lo (bytecodes e "virtual machines": Perl, Python, Ruby, Lua...)
   (c) Uma virtual machine pode compilar código para linguagem de máquina como um compilador e executar estes trechos junto à interpretação dos bytecodes (compilação JIT (just-in-time): Java, LuaJIT...)
4. A linguagem X é compilada ou interpretada?
   (a) Melhor pergunta: A *implementação* da linguagem X que usamos é interpretada ou compilada?
      i. Java: javac/java, gcj...
      ii. Lua: Lua, LuaJIT, LLVM-Lua, luaj, lua.js...
5. Boa parte do que veremos nessa disciplina serve tanto para o desenvolvimento de compiladores como de interpretadores

Histórico

1. Inicialmente os computadores eram programados diretamente em linguagem de máquina
2. Início dos anos 1950: primeiro assembler: SOAP (Symbolic Optimizing Assembly Program) para o IBM 650
3. Grace Hopper: termo "compilador"

4. 1953: Uma das primeiras linguagens de alto nível: Speedcoding (John Backus): código 10-20x mais lento que assembler escrito à mão
5. 1957: IBM lança a linguagem FORTRAN, com o primeiro compilador moderno: desempenho similar (mas ainda menor) a assembler escrito à mão
6. 1958: metade dos programas existentes para os mainframes IBM já eram escritos em FORTRAN
   (a) A estrutura geral de um compilador ainda se parece com a do primeiro compilador FORTRAN
   (b) As partes, porém, mudaram e evoluíram muito
   (c) Muitas das técnicas que veremos têm 30-40 anos, pois são fundamentadas em teoria sólida, mas a área segue evoluindo

Estrutura de um compilador

1. Cinco grandes fases, divididas em duas partes:
   (a) Front-end
      i. Análise léxica
      ii. Análise sintática
      iii. Análise semântica
   (b) Back-end
      i. Otimização
      ii. Geração de código
2. Front-end faz análise: extrai a estrutura do programa
   (a) Verifica a corretude do programa de entrada (mas o que é um programa correto?)
   (b) Produz uma representação interna do programa como uma árvore sintática abstrata
   (c) Erros detectáveis em tempo de compilação (normalmente não são todos!)
3. Back-end faz síntese: construção do programa de saída
   (a) Geração de código intermediário
      i. e otimização
   (b) Geração de código "final" (agora com registradores)
      i. e mais otimização
4. Análise léxica
   (a) Primeira fase em transformar o texto de entrada em algo estruturado
   (b) Quebrar o programa em "tokens"
      i. ignorar comentários
      ii. diferenciar identificadores de palavras reservadas
      iii. tokens com alguma estrutura (valores literais: strings, números)
      iv. tokens com vários símbolos (<= , etc.)
   (c) Expressões regulares

5. Análise sintática
   (a) Análise léxica é como análise morfológica
      i. ovo é substantivo, para é preposição
      ii. if é keyword, foo é identificador
   (b) Análise sintática é o entendimento dos tokens em termos sintáticos
      i. Sujeito, predicado, objeto direto, oração...
   (c) if (x == y) then z = true;
      i. x == y é expressão relacional
      ii. z = true é uma atribuição
      iii. o todo é um comando if-then
   (d) Gramáticas livres de contexto
      i. Formalismo criado por Chomsky para linguagens naturais!
6. Análise semântica
   (a) Semântica → análise do sentido
   (b) O que é fazer sentido?
      i. Tipagem
      ii. Declarações
      iii. Visibilidade
   (c) Limites (exemplo: retorno em Java)
7. Otimização
   (a) Transformação automática de programas de modo que eles
      i. rodem mais rápido
      ii. usem menos memória
      iii. sejam menores (gcc -Os)
      iv. bateria? rede? Usem menos recursos e tenham melhor desempenho
   (b) É uma arte: mistura de análise e síntese, uso de heurísticas...
   (c) Atenção à corretude! Não pode mudar o sentido do programa
      i. x = y * 0; versus x = 0;
8. Geração de código
   (a) Como mapear o programa na linguagem destino
   (b) Dificuldade varia bastante, dependendo das características das linguagens
   (c) Exemplo: if x == y then z = 1 else z =2

```
        cmp eax, ebx
        jne l1
        mov ecx, 1
        jmp l2
l1:     mov ecx, 2
l2:
```