Construção de Compiladores 1 - 2015.1 - Prof. Daniel Lucrédio
Lista 01 - Conceitos

1. O que é um compilador?

   R: "um programa que recebe como entrada um programa em uma linguagem de programação - a linguagem fonte - e o traduz para um programa equivalente em outra linguagem - a linguagem objeto"

2. Quais as diferenças entre um compilador e um interpretador? Cite vantagens e desvantagens de cada um.

   R: Um compilador traduz o programa fonte em um programa objeto, e o programa objeto é quem executa as ações semânticas. Um interpretador traduz um programa fonte diretamente em ações semânticas, isto é, é ele mesmo quem executa as ações.
   - Um compilador é mais eficiente, pois o código produzido executa de forma independente.
   - Um interpretador é menos eficiente, pois ele precisa executar o código ao mesmo tempo em que o traduz.
   - Um compilador tem maior dificuldade em oferecer diagnóstico de erros, pois na compilação parte da informação é perdida. O programa objeto não tem acesso direto ao programa fonte.
   - Um interpretador tem mais facilidade em oferecer diagnóstico de erros, pois ele tem o programa fonte acessível no momento da execução.
   - Um compilador é menos flexível quanto a mudanças, pois é sempre necessário recompilar antes de re-executar um programa.
   - Um interpretador é mais flexível quanto a mudanças, pois, alterando-se o programa fonte, basta executar o interpretador uma vez, e ele já irá considerar as mudanças.

3. O que é compilação JIT?

   R: É uma forma de aumentar a eficiência em um processo de interpretação ou compilação híbrida (compilação + interpretação). Um compilador JIT (Just-In Time) é inserido dentro de um interpretador, e é responsável por compilar o programa um pouco antes da interpretação, automaticamente. Enquanto um interpretador puro analisa o programa ao longo da execução, um interpretador com JIT faz uma pré-compilação e executa o código compilado, melhorando assim seu desempenho. Exceto o desempenho, para o usuário não há diferença entre um interpretador puro e um interpretador com JIT.

4. Por que uma gramática livre de contexto não é suficiente para a maioria das linguagens de programação?

   R: Porque a maioria das linguagens de programação faz uso de nomes (de variáveis, métodos, classes, procedimentos, funções, pacotes, etc.). Um nome serve como uma referência para algo que já foi "declarado" anteriormente no programa. Para reconhecer nomes, é necessário um formalismo capaz de lembrar múltiplas sequências de caracteres aparecendo repetidamente em um programa. Trata-se de uma extensão da linguagem {ww | w pertence a um alfabeto {0,1}*}, a qual não é livre de contexto, conforme pode ser demonstrado pelo lema do bombeamento para linguagens livres de contexto.

5. Como é possível utilizar uma gramática livre de contexto em um compilador (ao invés de uma gramática com menos restrições) e usufruir da simplicidade e praticidade de um PDA?

R: Eliminando os nomes da linguagem, em um primeiro momento, para tornar a análise sintática possível por meio de um PDA (Push-Down Automata ou Autômato com pilha). Os nomes podem ser considerados posteriormente, por meio das ações realizadas durante a análise semântica / síntese.

6. Que fase do compilador utiliza um autômato de pilha?

R: Análise sintática, dentro da etapa de análise, ou front-end do compilador.

7. Qual a diferença entre sintaxe e semântica, no contexto de um compilador?

R: Sintaxe é tudo que aparece na gramática livre de contexto. O resto é semântica.

8. Qual a diferença entre análise e síntese em um compilador?

R: A análise busca entender um programa fonte, sua sintaxe e significado, enquanto a síntese busca produzir um programa objeto que reflete o mesmo significado que o programa fonte, ainda que com sintaxe diferente.

9. Qual a diferença entre o *front-end* e o *back-end* de um compilador?

R: Front-end é o mesmo que análise, e back-end é o mesmo que síntese, portanto a resposta é a mesma da pergunta anterior.

10. Descreva todas as fases de um compilador, incluindo, para cada uma delas:
a) Entrada
b) Saída
c) Descrição
d) Se faz parte do front-end ou back-end

R:
1. Analisador léxico
a) Entrada = fluxo de caracteres
b) Saída = fluxo de tokens
c) Descrição = O analisador léxico é responsável por identificar, no programa fonte, as unidades léxicas que fazem parte da linguagem. Ele busca identificar as "palavras" que fazem parte do "vocabulário" da linguagem. Para cada unidade léxica, ele produz um token, que é uma estrutura de dados que representa as informações da unidade léxica necessárias para a fase seguinte. Também detecta erros léxicos, na forma de unidades léxicas mal formadas ("palavras" que não existem no "vocabulário" da linguagem)
d) Front-end.
2. Analisador sintático
a) Entrada = fluxo de tokens
b) Saída = árvore de análise sintática
c) Descrição = O analisador sintático busca determinar a FORMA com que as unidades léxicas se compõem. Se o léxico cuida das "palavras"/"vocabulário", o sintático cuida das "frases"/"gramática" da linguagem. Ele apenas considera o TIPO das unidades léxicas. Nomes são descartados, de forma que é possível fazer uso de um formalismo baseado em um PDA. Ele produz como resultado uma árvore de análise sintática, que é uma estrutura de dados que representa as construções sintáticas do programa em uma hierarquia que obedece às regras

da linguagem. Também detecta erros sintáticos, na forma de combinações inválidas de unidades léxicas ("frases" inválidas conforme a gramática da linguagem)

d) Front-end

3. Analisador semântico

a) Entrada = árvore de análise sintática

b) Saída = árvore de análise sintática (anotada, ou enriquecida com ações semânticas)

c) Descrição = O analisador semântico busca determinar o significado das construções sintáticas identificadas pelo analisador sintático. Se o sintático cuida da FORMA com que as "frases" são construídas, o semântico tenta dar-lhes significado. Nomes, descartados na análise sintática, são resgatados aqui, para oferecer subsídio ao entendimento do significado. Ele produz como resultado modificações na árvore de análise sintática, com o objetivo de indicar a semântica associada com cada construção sintática. Também detecta erros semânticos, na forma de inconsistências conceituais ("frases" que tem a FORMA correta, mas que não fazem sentido, não tem significado)

d) Front-end

4. Gerador de código intermediário

a) Entrada = árvore de análise sintática (anotada, ou enriquecida com ações semânticas)

b) Saída = representação intermediária

c) Descrição = O gerador de código intermediário produz uma representação alternativa, a ser usada pelo back-end. A representação intermediária resume todas as informações obtidas durante toda a análise (front-end), de forma a facilitar o trabalho de síntese (back-end). A representação pode ser a própria árvore de análise sintática, ou algum tipo de código, como o de 3 endereços

d) Front-end

5. Otimizador de código independente de máquina

a) Entrada = representação intermediária

b) Saída = representação intermediária otimizada

c) Descrição = O otimizador de código tenta remover redundâncias na representação intermediária, substituindo automaticamente construções do programa por outras com o mesmo significado, mas que sejam mais eficientes em termos de desempenho, consumo de memória, etc. Nessa fase as otimizações não levam em consideração a arquitetura da máquina alvo.

d) Back-end

6. Gerador de código

a) Entrada = representação intermediária otimizada

b) Saída = código da máquina alvo

c) Descrição = O gerador de código traduz a representação intermediária em código da máquina-alvo. Ele é responsável por produzir código executável que realiza as ações conforme especificadas no programa fonte.

d) Back-end

7. Otimizador de código dependente de máquina

a) Entrada = código da máquina alvo

b) Saída = código da máquina alvo otimizado

c) Descrição = Este otimizador também tenta modificar o código para melhorar desempenho, consumo de memória, etc, substituindo construções do programa por outras com o mesmo significado.

d) Back-end

É importante ressaltar que essa divisão em fases é apenas conceitual. Na maioria dos casos, muitas fases são agrupadas e realizadas ao mesmo tempo.

11. Qual a diferença entre lexema e token?

R: Lexema é uma sequência de caracteres exatamente conforme aparece no programa fonte. Um token é uma estrutura de dados que identifica a unidade léxica. Pode incluir informações como o tipo do lexema, ou o próprio lexema.

12. O que é otimização de código?

R: Um gerador de código nem sempre produz resultados ótimos. É comum, por exemplo, que sejam geradas instruções desnecessárias, ou que poderiam ser substituídas por outras mais simples e que ocupam menos memória, consomem menos processamento, etc. Isso porque um gerador de código prima pela exatidão, e não pela eficiência. Por isso pode ser necessário otimizar o código gerado para melhorar sua eficiência.

13. Dê dois exemplos de erros sintáticos (em uma linguagem de sua escolha)

R: Java - declaração de classe:
public class Produto extends { ... } /* Está faltando o nome da superclasse */
Java - comando for:
for (int i=0,i<10,i++) { ... } /* Separador dos elementos do for é ";" e não"," */

14. Dê dois exemplos de erros semânticos (em uma linguagem de sua escolha)

R: Java - atribuição de tipos incompatíveis
int x = "Alo mundo":
Java - variável não declarada
int a = 10; int b = 20;
c = a + b + z; /* z não foi declarada */

15. Descreva diferentes aplicações para a tecnologia de compiladores (e interpretadores)

R:
- Implementação de linguagens de programação de alto nível. Compiladores permitem que humanos possam programar máquinas usando uma linguagem que lhe é mais amigável
- Otimizações para arquiteturas. Compiladores podem, automaticamente, produzir código que aproveita o máximo de uma arquitetura em particular. Por exemplo, é possível automaticamente produzir código que executa paralelamente em um processador de múltiplos núcleos, sem que o programador precise se preocupar com paralelismo.
- Projeto de novas arquiteturas. Muitas das arquiteturas de hardware existentes atualmente evoluíram para o ponto atual devido a tecnologia de compiladores.
- Traduções / transformações. Sempre que é necessário que o programador trabalhe em dois problemas diferentes, ele normalmente precisa aprender linguagens diferentes. Um compilador pode evitar esse trabalho, transformando automaticamente construções de uma linguagem para construções de outra.
- Ferramentas de produtividade de software. Os compiladores, hoje, ajudam efetivamente o programador, detectando todo tipo de erros que é possível detectar, como erros de tipos ou alguns erros semânticos. Ambientes de programação também utilizam compiladores para ajudar o programador, com tarefas como auto-complete, syntax highlighting, entre outras.

16. Qual a diferença entre aspectos estáticos e dinâmicos, em uma linguagem de programação, sob o

ponto de vista do compilador?

R: Aspectos estáticos são aqueles que podem ser verificados pelo compilador. Aspectos dinâmicos só podem ser verificados durante a execução do programa alvo.

17. Defina (e explique as diferenças entre): identificador, nome e variável

R:
Um identificador é uma cadeia de caracteres. Remete a uma unidade léxica.
Um nome se refere a alguma construção do programa. Remete a um elemento do programa, conforme percebido pelo analisador semântico.
Uma variável representa, normalmente, um espaço de memória a ser utilizado durante o programa.

Nomes e identificadores podem ser a mesma coisa, mas há nomes compostos por mais de um identificador. Por exemplo, um método soma, da classe Matematica, tem o nome composto "Matematica.soma". A diferença entre eles está no momento em que é tratado. Identificador é como esse conceito é tratado na análise léxica, isto é: o analisador léxico não "enxerga" nomes, somente identificadores. Nomes são tratados na análise semântica, isto é: o analisador semântica "enxerga" um ou mais identificadores e decide se é um nome ou não.

Variáveis são um dos tipos de elementos de um programa que possuem nomes. Uma variável representa um espaço de memória, que fica amarrado a um nome para poder ser usada. Seu nome, por sua vez, é composto por um ou mais identificadores.

18. Qual a diferença entre escopo e ambiente? Mostre com um exemplo essa diferença

R: Escopo é um trecho de programa, bem definido e delimitado. É um pedaço do programa, normalmente conforme visto pelo compilador. Um ambiente é um momento da execução de um programa. É uma configuração de memória e status de processamento, conforme visto durante a execução.

Ex: uma função recursiva:

```
int recursiva(int a) {
   int i = 0;
   ...
   return recursiva(a-1);
}
```

O trecho entre as chaves ({ e }) delimita um escopo. Neste escopo, por exemplo, há duas variáveis "válidas": "a" e "i", o que significa que não podem ser usadas em outros trechos do programa.

Durante a execução, e associado a este mesmo trecho, podem existir múltiplos ambientes, devido à recursividade. Cada vez que a função é chamada um novo ambiente é criado, para armazenar novas instâncias de "a" e "i".

Assim, pode-se constatar que escopo normalmente é um aspecto estático (apesar de existirem linguagens que tratam escopo dinamicamente), e ambiente é um aspecto dinâmico.

19. O que são sinônimos em uma linguagem de programação? Qual sua importância para o

compilador?

R: É a utilização de dois nomes diferentes para um mesmo elemento (variável, objeto, método, função, etc.) Um compilador precisa tomar cuidado com sinônimos, pois ele não pode sempre assumir que nomes diferentes significam coisas diferentes. Se assim for feito, algumas otimizações podem levar a estados inconsistentes. Por exemplo, o compilador pode achar que uma variável, após o término do escopo em que é válida, não é mais útil, e gerar código para liberar seu espaço de memória. Mas se essa variável tiver um sinônimo, isso irá causar um erro.