COMPILADORES

Prova 1 – 14/11/2019 – Prof. Marcus Ramos

Questão 1 (1 ponto): A linguagem-fonte de um compilador pode ser de baixo-nível? E a linguagem-objeto, pode ser de alto-nível? Esclareça os conceitos envolvidos nestas perguntas.

Linguagem-fonte é a linguagem de entrada de um processador de linguagens (por exemplo, um compilador). Linguagem-objeto é a linguagem de saída. Tanto a linguagem-fonte quanto a linguagem-objeto podem ser tanto de alto-nível quanto de baixo-nível. A caracterização do particular processador de linguagem depende do nível da linguagem-fonte e também do nível da linguagem-objeto em questão. Um compilador, por exemplo, tem como linguagem-fonte uma linguagem de alto-nível (e não de baixo-nível) e como linguagem-objeto uma linguagem de baixo-nível (e não de alto-nível). Um processador em que a linguagem-fonte é de baixo-nível e a linguagem-objeto é de alto-nível pode ser chamado de "descompilador". Outras combinações são o tradutor (AN/AN) e o montador (BN/BN).

Questão 2 (1 ponto): Usando a notação dos diagramas-T, mostre como acontece a compilação e a execução de um programa escrito na linguagem Java numa máquina x86.

A linguagem Java é inicialmente compilada e depois interpretada. Ou seja, um programa escrito em Java deve ser primeiro compilado ("javac") e depois interpretado ("java").

Compilação:

```
  P                    P
Java   Java -> JVM   JVM
           x86
```

Interpretação:

```
 P
JVM
JVM
x86
```

Questão 3 (1 ponto): Justifique o interesse pelo uso da técnica de bootstrapping na construção de compiladores.

A técnica de bootstrapping é uma importante ferramenta quando se deseja implementar uma nova linguagem buscando independência da linguagem antiga usada no desenvolvimento da nova. Ela permite, por exemplo, escalonar o desenvolvimento da linguagem-fonte e também da linguagem-objeto, antecipando os benefícios do uso da nova ferramenta.

Questão 4 (1 ponto): Qual a principal vantagem de se organizar um compilador em vários passos ao invés de um único passo?

O principal interesse reside da possibilidade de se construir um sistema mais aderente aos princípios da Engenharia de Software: módulos com alta coesão e com baixo acoplamento

entre eles. Isso facilita a construção e a manutenção do compilador, reduzindo os custos envolvidos. Este aspecto se tornou especialmente importante nos tempos modernos, uma vez que o custos de manutenção de sistemas antigos é muito elevado.

Questão 5 (1 ponto): Quais as conseqüências práticas de se representar a sintaxe de uma linguagem de programação por meio de uma gramática livre de contexto?

A principal conseqüência é que a linguagem representada pela gramática livre de contexto é, normalmente, um super-conjunto da linguagem que efetivamente se deseja representar. Isto decorre do fato de a linguagem-fonte geralmente incluir uma série de regras de contexto (como por exemplo regras de escopo e regras de tipo) que não podem ser formalizadas por meio de uma gramática livre de contexto. Como resultado, torna-se necessário incluir no compilador uma fase de análise (chamada de análise de contexto), posterior à análise sintática (baseada numa gramática livre de contexto), especialmente para fazer valer as regras de contexto da linguagem-fonte.

Questão 6 (1 ponto): Prove que a seguinte gramática é LL(1):

$S \rightarrow abX \mid Ycba \mid bXa \mid Xba$

$X \rightarrow dX \mid e$

$Y \rightarrow fX \mid gX \mid hX \mid \varepsilon$

Y:
$first\ (fX) = \{f\};\ first\ (gX) = \{g\};\ first\ (hX) = \{h\};\ follow\ (Y) = \{c\}.$
Além disso, $\{f\} \cap \{g\} \cap \{h\} \cap \{c\} = \emptyset$
X:
$first\ (dX) = \{d\};\ first\ (e) = \{e\};$
Além disso, $\{d\} \cap \{e\} = \emptyset$
S:
$first\ (abX) = \{a\};\ first\ (Ycba) = \{f, g, h, c\};\ first\ (bXa) = \{b\};\ follow\ (Xba) = \{d, e\}.$
Além disso, $\{a\} \cap \{f, g, h, c\} \cap \{b\} \cap \{d, e\} = \emptyset$

Questão 7 (1 ponto): Descreva, em linhas gerais, como funcionam os métodos de análise sintática descendente e de análise sintática ascendente.

O análise sintática descendente, também conhecida como “top-down”, lê o arquivo-fonte da esquerda para a direita e utiliza derivações mais à esquerda. A árvore de derivação, portanto, é montada de cima para baixo (da raiz em direção às folhas). No caso da análise sintática ascendente, também conhecida como “bottom-up”, o arquivo também é lido da esquerda para a direita, mas são usadas reduções mais à esquerda ao invés de derivações mais à esquerda. Vale lembrar que uma redução é o inverso de uma derivação e também que uma seqüência de reduções mais à esquerda corresponde à ordem inversa de uma seqüência de derivações mais à direita. A árvore de derivação, portanto, é montada de baixo para cima (das folhas em direção à raiz).

Questão 8 (1 ponto): Que outras tarefas um analisador sintático típico deve realizar além de apenas verificar se o programa-fonte pertence ou não pertence à linguagem-fonte?

Ele deve (i) emitir boas mensagens de erro, incluindo informações precisas sobre o local onde o mesmo foi cometido, a natureza do erro e possíveis correções, e (ii) montar uma árvore de sintaxe abstrata que seja uma representação fiel do programa-fonte para uso nas fases posteriores do compilador.

Questão 9 (2 pontos): Obtenha um esboço de reconhecedor sintático para a linguagem cuja sintaxe é representada pela gramática abaixo. Considere a existência de um analisador léxico adequado e resolva pendências gramaticais e de especificação da linguagem-fonte.

```
<atribuição> ::= <id> := <expressão>
<comando> ::= <atribuição>
      | <condicional>
      | <iterativo>
      | <comando-composto>
<comando-composto> ::=  begin <lista-de-comandos> end
<condicional> ::= if <expressão> then <comando> ( else <comando>
| <vazio> )
<corpo> ::= < declaração-de-variável > <comando-composto>
<declaração-de-variável> ::= var <lista-de-ids> : <id>
<expressão> ::= <expressão-simples>
      | <expressão-simples> <op-rel> <expressão-simples>
<expressão-simples> ::= <expressão-simples> <op-ad> <termo>
      | <termo>
<fator> ::= <id>
      | <literal>
      | "(" <expressão> ")"
<iterativo> ::= while <expressão> do <comando>
<lista-de-comandos> ::= <comando> ;
      | <lista-de-comandos> <comando> ;
      | <vazio>
<lista-de-ids> ::= <id>
      | <lista-de-ids> , <id>
<literal> ::= <bool-lit>
      | <int-lit>
      | <float-lit>
<programa> ::= program <id> ; <corpo> .
<termo> ::= <termo> <op-mul> <fator>
      | <fator>
```

```
parsePrograma () {
accept ("programa");
accept ("id");
accept (";");
parseCorpo ();
}
parseCorpo () {
parseDeclaração-de-Variável ();
parseComando-Composto ();
}
```

```
parseDeclaração-de-Variável () {
accept (“var”);
parseLista-de-Ids ();
accept (“:”);
accept (“id”);
}
parseLista-de-Ids () {
accept (“id”);
while (currentToken = “,”) {
      acceptIt ();
      accept (“id”);
      }
}
parseComando-Composto () {
accept (“begin”);
parseLista-de-Comandos ();
accept (“end”);
}
parseLista-de-Comandos () {
while (currentToken = “if” || “while” || “id” || “begin”) {
      parseComando ();
      accept (“;”);
      }
}
parseComando () {
switch (currentToken) {
      case “id”:
            acceptIt ();
            accept (“:=”);
            parseExpressão ();
            break;
      case “if”:
            acceptIt ();
            parseExpressão ();
            accept (“then”);
            parseComando ();
            if (currentToken = “else) {
                  acceptIt ();
                  parseComando ();
                  }
            break;
      case “while”:
            acceptIt ();
            parseExpressão ();
            accept (“do”);
            parseComando ();
            break;
      case “begin”:
```

```
            acceptIt ();
            parseLista-de-Comandos ();
            accept (“end”);
            break;
      default:
            ERRO;
      }
}
parseExpressão () {
parseExpressão-Simples ();
if (currentToken = op-rel) {
      acceptIt ();
      parseExpressão-Simples ();
      }
}
parseExpressão-Simples () {
parseTermo ();
while (currentToken = op-ad) {
      acceptIt ();
      parseTermo ();
      }
}
parseTermo () {
parseFator ();
while (currentToken = op-mul) {
      acceptIt ();
      parseFator ();
      }
}
parseFator () {
switch (currentToken) {
      case “id”:
      case “bool-lit”:
      case “int-lit”:
      case “float-lit”:
            acceptIt ();
            break;
      case “(”:
            parseExpressão ();
            accept (“)”);
            break;
      default:
            ERRO
      }
}
```