COMPILADORES

Prova 1- 06/02/2018 – Prof. Marcus Ramos

Questão 1 (1,5 ponto): Em que consiste a condição LL(1) e por que ela é tão importante na construção de analisadores sintáticos?

Consiste numa verificação que é feita na gramática que gera a linguagem para a qual se deseja construir o analisador sintático. Quando ela é LL(1), isto significa que as sentenças da linguagem podem ser analisadas de forma descendente (ou seja, por meio da ordem direta das derivações mais à esquerda) com o look-ahead de apenas um símbolo. Isto significa, então, que é possível construir um analisador determinístico e muito simples para a linguagem. Caso a gramática não seja LL(1), pode-se verificar se existe alguma gramática equivalente que seja LL(1), mas nem sempre será o caso.

Questão 2 (2,0 ponto): Determine se a gramática abaixo é LL(1). Justifique a sua resposta.

$S \rightarrow aX \mid Y \mid Ze$

$X \rightarrow aX \mid bY \mid cZY$

$Y \rightarrow bY \mid c$

$Z \rightarrow dZ \mid fYg \mid \varepsilon$

Sim, ela é LL(1). De fato, para o não-terminal S temos:

$first_1(aX) = \{a\}$

$first_1(Y) = \{b, c\}$

$first_1(Ze) = \{d, f, e\}$

Para o não-terminal X temos:

$first_1(aX) = \{a\}$

$first_1(bY) = \{b\}$

$first_1(cZY) = \{c\}$

Para o não-terminal Y temos:

$first_1(bY) = \{b\}$

$first_1(c) = \{c\}$

E para o não-terminal Z temos:

$first_1(dZ) = \{d\}$

$first_1(fYg) = \{f\}$

$follow_1\ (Z) = \{b, c, e\}$

Como todos os conjuntos são disjuntos para cada não-terminal, segue que a gramática é LL(1).

Questão 3 (2,0 ponto): Construa o esboço de um reconhecedor recursivo descendente para a linguagem descrita pela gramática abaixo.

$Programa \rightarrow Declarações\ Comandos$

$Declarações \rightarrow Declaração; Declarações \mid Declaração$;

$Comandos \rightarrow Comando; Comandos \mid Comando$;

$Declaração \rightarrow var\ Nome : integer \mid const\ \ Nome = Literal$

$Comando \rightarrow Nome := Expressão$

$Expressão \rightarrow Expressão + Fator \mid Expressão * Fator \mid Fator$

$Fator \rightarrow Nome \mid Literal \mid (Expressão)$

Considere que $Nome$ e $Literal$ representam, respectivamente, identifiadores e literais inteiros, e são reconhecidos pelo analisador léxico.

```
void parsePrograma () {
parseDeclarações ();
parseComandos ();
}

void parseDeclarações () {
parseDeclaração ();
accept (";");
while (currentToken = var) || (currentToken = const)
parseDeclarações ();
}

void parseComandos () {
parseComando ();
accept (";");
while (currentToken = Nome) parseComandos ();
}

void parseDeclaração () {
switch (currentToken)
case var:
      accept (Nome);
      accept (":");
      accept ("integer");
      break;
case const:
      accept (Nome);
      accept (":");
      accept (Literal);
      break;
default: erro ()
```

```
}

void parseComando () {
accept (Nome);
accept (“:=”);
parseExpressão ();
}

parseExpressão () {
parseFator ();
while (currentToken = “+”) || (currentToken == “*”) {
  acceptIt ();
  parseExpressão ();
  }
}

parseFator () {
switch (currentToken)
case Nome:
     acceptIt ();
     break;
case Literal:
     acceptIt ();
     break;
case “(“:
     acceptIt ();
     parseExpressão ();
     accept (“)”);
     break;
}
```

Questão 4 (1,5 ponto): Discorra sobre as vantagens da utilização do padrão de projeto Visitor no projeto e implementação de compiladores.

O Visitor permite desacoplar a representação física de uma estrutura de dados das operações que devem ser aplicadas sobre a mesma. Desta forma pode-se, por exemplo, separar a descrição da árvore de sintaxe abstrata das operações de visualização, análise de contexto e geração de código que devem ser aplicadas sobre ela. Isto permite a construção de um compilador onde as fases de visualização de árvore, análise de contexto e geração de código são implementadas em classes distintas e totalmente desvinculadas umas das outras. Desta forma o Visitor contribui para uma maior coesão e menor acoplamento dos módulos que compõem o compilador.

Questão 5 (1,5 ponto): Descreva (i) o que acontece durante a subfase de identificação da análise de contexto, (ii) como é feita a implementação da mesma e (iii) o resultado produzido por ela.

A subfase de identificação tem como objetivo resolver as referências aos identificadores dentro do programa. Para isso, ela busca vincular cada uso de cada nome à respectiva declaração, gerando dessa forma uma árvore de sintaxe abstrata decorada. A implementação é feita por meio de uma classe que implementa a interface Visitor. Cada método desta classe implementa um algoritmo de visitação em um dos nós que compõem a árvore de sintaxe

abstrata. Eles trabalham de forma cooperativa para fazer a identificação do programa completo. Uma segunda classe, auxiliar, é usada na forma de uma Tabela de Símbolos para facilitar a pesquisa dos nomes referenciados. Como resultado, é possível gerar mensagens de erro para nomes declarados em duplicidade e nomes não declarados. Também, é possível gerar alertas para nomes declarados e não utilizados. Fora isso, ela produz uma AST decorada com as vinculações entre usos e respectivas declarações.

Questão 6 (1,5 ponto): Descreva (i) o que acontece durante a subfase de verificação de tipos da análise de contexto, (ii) como é feita a implementação da mesma e o (iii) resultado produzido por ela.

Na subfase de verificação de tipos, que ocorre depois da identificação, o objetivo é verificar se todas as operações estão sendo usadas de forma compatível com a maneira como foram declaradas. Isto implica determinar se a quantidade e os tipos dos operandos (variáveis, literais etc) são compatíveis coma s declarações das operações, e também se os tipos dos resultados são compatíveis com a utilização que deles é feita. A implementação é feita por uma classe que implementa a interface Visitor, e que consiste de uma coleção de métodos, um para cada tipo de nós que compõe a árvore de sintaxe abstrata. Os métodos trabalham de forma cooperativa para fazer a inferência de todos os tipos das expressões utilizadas no programa, e também para determinar se os tipos resultantes são compatíveis com o uso que é feito dos mesmos. O resultado são mensagens de erro para tipos incompatíveis ou tipos diferentes do esperado por exemplo num comando. Além disso, esta subfase complementa a decoração da AST com informações sobre o tipo dos nós internos.