COMPILADORES

Prova 1 – 01/10/2009, Prof. Marcus Ramos

1. Conceitue:

   - Linguagem-fonte e objeto;

     Linguagem-fonte é a linguagem usada para escrever programas que são entrada de processadores de linguagens. Linguagem-objeto é a linguagem usada para escrever programas que são saída de processores de linguagens.

   - Linguagem de alto-nível e de baixo-nível;

     Linguagem de alto-nível é a linguagem mais próxima da linguagem natural, e apresenta como principais características portabilidade, segurança, legibilidade e uso de abstrações. Linguagem de baixo-nível é a linguagem mais próxima do hardware, e apresenta como principais características dependência da arquitetura, baixa legibilidade, baixa segurança e pouco ou nenhum suporte para o uso de abstrações.

   - Linguagem de montagem e de máquina;

     Linguagem de montagem e linguagem de máquina são ambas linguagens de baixo-nível, e por isso compartilham praticamente as mesmas características. A única diferença é que a linguagem de montagem utiliza mnemônicos para melhorar um pouco a legibilidade dos programas, ao passo que a linguagem de máquina utiliza apenas os códigos numéricos que são interpretados pela máquina-alvo diretamente.

   - Discorra sobre as relações que existem entre todos esses tipos de linguagens.

     Linguagens-fonte e linguagens-objeto podem ser tanto de alto quanto de baixo-nível. Conforme a particular combinação, denomina-se o processador de linguagens como compilador, tradutor, filtro, montador etc. Linguagens de alto-nível e de baixo-nível podem ser tanto linguagens-fonte quanto linguagens-objeto, dependendo de serem, respectivamente, entrada ou saída de processadores de linguagens.

2. Considere dados uma máquina x86 e um compilador C/x86 que roda em x86. Mostre, utilizando a notação dos Diagramas-T, como obter um compilador Java/x86, executável em x86, considerando a seguinte seqüência de desenvolvimento:

   Considere X/Y/Z: X é a linguagem-fonte, Y é a linguagem-objeto e Z é a linguagem em que o processador está escrito.

   - Implementação de um subconjunto Java1 da linguagem-fonte (Java1 $\subset$ Java) e de programas-objeto não otimizados;

     Java1/x86n/C $\rightarrow$ *C/x86/x86* $\rightarrow$ *Java1/x86n/x86*
     Java1/x86n/Java1 $\rightarrow$ *Java1/x86n/x86* $\rightarrow$ *Java1/x86n/x86*

   - Implementação de um subconjunto Java2 da linguagem-fonte (Java1 $\subset$ Java2 $\subset$ Java) e de programas-objeto ainda não otimizados;

     Usando a implementação Java1 do item anterior:
     Java2/x86n/Java1 $\rightarrow$ *Java1/x86n/x86* $\rightarrow$ *Java2/x86n/x86*
     Java2/x86n/Java2 $\rightarrow$ *Java2/x86n/x86* $\rightarrow$ *Java2/x86n/x86*

   - Implementação da linguagem-fonte Java completa e de programas-objeto ainda não otimizados;

     Usando a implementação Java2 do item anterior:

Java/x86n/Java2 → Java2/x86n/x86 → Java/x86n/x86
Java/x86n/Java → Java/x86n/x86 → Java/x86n/x86

- Implementação da linguagem-fonte Java completa e de programas-objeto otimizados.

Usando a implementação Java do item anterior:
Java/x86/Java → Java/x86n/x86 → Java/x86/x86
Java/x86/Java → Java/x86/x86 → Java/x86/x86

A versão final do compilador deverá ser capaz de compilar a si mesma (compilador "auto-compilável").

3. Conceitue:

- Análise sintática determinística x não-determinística;

Análise determinística é aquela onde todas as decisões de movimentação do reconhecedor são tomadas "sem arrependimento". Se não houver movimentação possível, isso indica a ocorrência de um erro na cadeia de entrada. Análise não-determinística, por outro lado, opera por tentativa e erro. Erros na cadeia de entrada são apontados apenas depois de esgotadas todas as possibilidades de movimentação.

- Análise sintática descendente x ascendente;

Análise descentente, também conhecida como top-down, é aquela em que o reconhecedor inicia os seus movimentos na raiz da gramática e evolui até a cadeia de entrada (em termos de árvore, é aquela em que a árvore é montade de cima para baixo). Análise ascendente, ou bottom-up, é aquela em que os movimentos do reconhecedor vão da cadeia entrada em direção à raiz da gramática (a árvore é montada de baixo para cima).

- Gramática e linguagem LL(1) e LR(1).

Gramática LL(1) é aquela que gera uma linguagem que pode ser reconhecida da esquerda para a direita, usando derivações mais à esquerda e com look-ahead de apenas um símbolo. Linguagem LL(1) é aquela que pode ser gerada por alguma gramática LL(1). Gramática LR(1) é aquela que gera uma linguagem pode ser reconhecida da esquerda para a direita, usando reduções mais à esquerda e com look-ahead de apenas um símbolo. Linguagem LR(1) é aquela que pode ser gerada por alguma gramática LR(1).

4. Obtenha o esboço de um reconhecedor, através do método recursivo descendente, para a linguagem definida pela expressão:

$$(+|-|\varepsilon)(d^{+}(\varepsilon|.|.d^{*})|.d^{+})(e(+|-|\varepsilon)d^{+}|\varepsilon)$$

São exemplos de sentenças pertencentes à essa linguagem: `123,-45.312, +.76, 5.44e2, +0.88e-35.`

```
void parseNumero() {
if s=="+" takeIt()
  else if s=="-" takeIt();
if s=="d" {
        takeIt();
        while s=="d" takeIt();
        if s="." {
              takeIt();
              while s=="d" takeIt();
              }
        }
  else if s="." {
              takeIt();
```

```
            take("d");
            while s=="d" takeIt();
            }
        else ERRO();
if s=="e" {
        takeIt();
        if s=="+" takeIt()
         else if s=="-" takeIt();
        take("d");
        while s=="d" takeIt();
        }
```

5. Considere a gramática abaixo:

S → aXb | aYc | aaZd
X → bX | bc
Y → cY | cb
Z → dZ | ε

- Essa gramática é LL(1)? Prove a sua resposta.

  Não, pois os conjuntos starter não são disjuntos:
  starter (aXb) = {a}, starter (aYc) = {a}, starter (aaZd) = {a}
  starter (bX) = {b}, starter (bc) = {b}
  starter (cY) = {c}, starter (cb) = {c}
  starter (dZ) = {d}, follow (Z) = {d}

- Caso a gramática acima não seja LL(1), obtenha uma gramática equivalente, mas que seja LL(1). Prove que a nova gramática é LL(1).

  Eliminando recursões e fazendo substituições:
  *S → ab*bcb | ac*cbc | aad*d*
  Fatorando:
  *S → a(b*bcb | c*cbc | ad*d)*
  Logo:
  *starter (b*bcb) = {b}*
  *starter (c*cbc) = {c}*
  *starter (aad*d) = {a}*

6. Apesar de a maioria das linguagens de programação de alto-nível exibir diversos tipos de dependências de contexto, elas (as linguagens) normalmente são representadas, sintaticamente, através de gramáticas livres de contexto. Justifique:

- O motivo de se usar essa estratégia;

  Formalismos para representar dependências de contexto são geralmente mais complexos e trabalhosos de se usar do que aqueles usados para representar linguagens livres de contexto.

- As conseqüências práticas da mesma na especificação da linguagem-fonte;

  A linguagem especificada através de uma gramática livre de contexto é mais ampla do que a linguagem desejada.

- As conseqüências práticas da mesma no desenvolvimento do compilador para a linguagem.

  Torna-se necessário introduzir uma fase de análise de contexto, posterior à análise livre de contexto, para detectar eventuais erros de contexto contidos no programa-fonte.

7. Considere a gramática abaixo, sobre o alfabeto Σ={“(“,”)”,”a”,”,”}:

L → (S)
S → I , S | I
I → a | L

- Mostre os movimentos de um reconhecedor ascendente na análise da sentença (a,(a),(a,a));

  (a,(a),(a,a)) ⇒ (I,(a),(a,a)) ⇒ (I,(I),(a,a)) ⇒ (I,(S),(a,a)) ⇒ (I,L,(a,a)) ⇒ (I,I,(a,a)) ⇒ (I,I,(I,a)) ⇒ (I,I,(I,I)) ⇒ (I,I,(I,S)) ⇒ (I,I,(S)) ⇒ (I,I,L) ⇒ (I,I,I) ⇒ (I,I,S) ⇒ (I,S) ⇒ (S) ⇒ L

- Mostre os movimentos de um reconhecedor descendente na análise da sentença (a,(a),(a,a));

  L ⇒ (S) ⇒ (I,S) ⇒ (a,S) ⇒ (a,I,S) ⇒ (a,L,S) ⇒ (a,(S),S) ⇒ (a,(I),S) ⇒ (a,(a),S) ⇒ (a,(a),I) ⇒ (a,(a),L) ⇒ (a,(a),(S)) ⇒ (a,(a),(I,S)) ⇒ (a,(a),(a,S)) ⇒ (a,(a),(a,I)) ⇒ (a,(a),(a,a))

- Obtenha o esboço de um reconhecedor recursivo descendente para a linguagem por ela definida.

```
void parseL(){
     accept ("(");
     parseS();
     accept (")");
}
void parseS(){
     parseI();
     while s==","  {
           acceptIt();
           parseI();
     }
}
void parseI(){
     switch s {
           case "a": acceptIt();
                     break();
           case "(": parseS();
                     break;
           defaul: ERRO();
     }
}
```