Universidad de Huelva

Procesadores de lenguaje

Ejercicios del Tema 3

ANÁLISIS DESCENDENTE

Ejercicio 3.1

Regla	Primeros	Siguientes	Predicción
$A \rightarrow B\ C\ D$	{ **a**, **c**, **e**, **g**, **i** }	{ **$**, **d** }	{ **a**, **c**, **e**, **g**, **i** }
$B \rightarrow$ **a** C **b**	{ **a** }	{ **c**, **e**, **g**, **i**, **f** }	{ **a** }
$B \rightarrow \lambda$	{ λ }		{ **c**, **e**, **g**, **i**, **f** }
$C \rightarrow$ **c** A **d**	{ **c** }	{ **i**, **b** }	{ **c** }
$C \rightarrow$ **e** B **f**	{ **e** }		{ **e** }
$C \rightarrow$ **g** D **h**	{ **g** }		{ **g** }
$C \rightarrow \lambda$	{ λ }		{ **i**, **b** }
$D \rightarrow$ **i**	{ **i** }	{ **$**, **d**, **h** }	{ **i** }

La gramática es LL(1) ya que los conjuntos de predicción de las reglas de cada símbolo son disjuntos.

Ejercicio 3.2

Regla	Primeros	Siguientes	Predicción
$A \rightarrow B\ C\ D$	{ **b**, **c**, **d**, λ}	{ **$** }	{ **b**, **c**, **d**, **$** }
$B \rightarrow$ **b**	{ **b** }	{ **c**, **d**, **$** }	{ **ab** }
$B \rightarrow \lambda$	{ λ }		{ **c**, **d**, **$** }
$C \rightarrow$ **c**	{ **c** }	{ **d**, **$** }	{ **c** }
$C \rightarrow \lambda$	{ λ }		{ **d**, **$** }
$D \rightarrow$ **d**	{ **d** }	{ **$** }	{ **d** }
$D \rightarrow \lambda$	{ λ }		{ **$**}

La gramática es LL(1) ya que los conjuntos de predicción de las reglas de cada símbolo son disjuntos.

Ejercicio 3.3

Ejercicio 3.4

a) Gramática transformada

Índice	Regla
R1	$S \rightarrow A\ B$
R2	$A \rightarrow$ **begin** S **end** B **theend**
R3	$A \rightarrow \lambda$
R4	$B \rightarrow$ **var** L **:** **tipo** BP
R5	$B \rightarrow BP$
R6	$BP \rightarrow$ **fvar** BP
R7	$BP \rightarrow \lambda$
R8	$L \rightarrow$ **id** LP
R9	$LP \rightarrow$ **,** **id** LP
R10	$LP \rightarrow \lambda$

b) Cálculo de los conjuntos de predicción

Regla	Primeros	Siguientes	Predicción
$S \rightarrow A\ B$	{ **begin**, **var**, **fvar**, λ }	{ **$**, **end** }	{ **begin**, **var**, **fvar**, **$**, **end**}
$A \rightarrow$ **begin** S **end** B **theend**	{ **begin** }	{ **var**, **fvar**, **$**, **end** }	{ **begin** }
$A \rightarrow \lambda$	{ λ }		{ **var**, **fvar**, **$**, **end** }
$B \rightarrow$ **var** L **:** **tipo** BP	{ **var** }	{ **$**, **end**, **theend** }	{ **var** }
$B \rightarrow BP$	{ **fvar**, λ }		{ **fvar**, **$**, **end**, **theend** }
$BP \rightarrow$ **fvar** BP	{ **fvar** }	{ **$**, **end**, **theend** }	{ **fvar** }
$BP \rightarrow \lambda$	{ λ }		{ **$**, **end**, **theend** }
$L \rightarrow$ **id** LP	{ **id** }	{ **:** }	{ **id** }
$LP \rightarrow$ **,** **id** LP	{ **,** }	{ **:** }	{ **,** }
$LP \rightarrow \lambda$	{ λ }		{ **:** }

d) Tabla de análisis sintáctico LL(1)

	begin	**end**	**theend**	**var**	**:**	**tipo**	**fvar**	**id**	**,**	**$**
S	R1	R1		R1			R1			R1
A	R2	R3		R3			R3			R3
B		R5	R5	R4			R5			R5
BP		R7	R7				R6			R7
L								R8		
LP					R10				R9	

e) Traza

Pila	**Entrada**	**Acción**
S **$**	**begin** ...	R1
A B **$**	**begin** ...	R2
begin *S* **end** *B* **theend** *B* **$**	**begin** ...	match(**begin**)
S **end** *B* **theend** *B* **$**	**var** ...	R1
A B **end** *B* **theend** *B* **$**	**var** ...	R3
B **end** *B* **theend** *B* **$**	**var** ...	R4
var *L* **:** **tipo** *BP* **end** *B* **theend** *B* **$**	**var** ...	match(**var**)
L **:** **tipo** *BP* **end** *B* **theend** *B* **$**	**id** ...	R8
id *LP* **:** **tipo** *BP* **end** *B* **theend** *B* **$**	id ...	match(**id**)
LP **:** **tipo** *BP* **end** *B* **theend** *B* **$**	, ...	R9
, **id** *LP* **:** **tipo** *BP* **end** *B* **theend** *B* **$**	, ...	match(,)
id *LP* **:** **tipo** *BP* **end** *B* **theend** *B* **$**	**id** ...	match(**id**)
LP **:** **tipo** *BP* **end** *B* **theend** *B* **$**	**:** ...	R10
: **tipo** *BP* **end** *B* **theend** *B* **$**	**:** ...	match(:)
tipo *BP* **end** *B* **theend** *B* **$**	**tipo** ...	match(**tipo**)
BP **end** *B* **theend** *B* **$**	**var** ...	¡ ERROR !

Ejercicio 3.5

Ejercicio 3.6

Ejercicio 3.7

Ejercicio 3.8

(a) Transformaciones necesarias para expresar la gramática en notación BNF

$\langle E' \rangle \rightarrow$ barra $\langle O \rangle \langle E' \rangle$
$\langle E' \rangle \rightarrow \lambda$
$\langle O \rangle \rightarrow \langle O \rangle \langle I \rangle$
$\langle O \rangle \rightarrow \langle I \rangle$
$\langle I \rangle \rightarrow$ sim
$\langle I \rangle \rightarrow$ parab $\langle E \rangle$ parce $\langle C \rangle$
$\langle C \rangle \rightarrow$ ast
$\langle C \rangle \rightarrow$ sum
$\langle C \rangle \rightarrow$ int

(b) Transformaciones necesarias para convertirla en LL(1)

$\langle E' \rangle \rightarrow$ barra $\langle O \rangle \langle E' \rangle$
$\langle E' \rangle \rightarrow \lambda$
$\langle O \rangle \rightarrow \langle I \rangle \langle O' \rangle$
$\langle O' \rangle \rightarrow \langle I \rangle \langle O' \rangle$
$\langle O' \rangle \rightarrow \lambda$
$\langle I \rangle \rightarrow$ sim
$\langle I \rangle \rightarrow$ parab $\langle E \rangle$ parce $\langle C \rangle$
$\langle C \rangle \rightarrow$ ast
$\langle C \rangle \rightarrow$ sum
$\langle C \rangle \rightarrow$ int

(c) Conjuntos Primeros, Siguientes y de Predicción

Símbolo	Regla	Primeros	Siguientes	Predicción
<E>	<E> → <O> <E'>	**sim**, **parab**	**parce**, $	**sim**, **parab**
<E'>	<E'> → **barra** <O> <E'>	**barra**	**parce**, $	**barra**
	<E'> → λ	λ		**parce**, $
<O>	<O> → <I> <O'>	**sim**, **parab**	**barra**, **parce**, $	**sim**, **parab**
<O'>	<O'> → <I> <O'>	**sim**, **parab**	**barra**, **parce**, $	**sim**, **parab**
	<O'> → λ	λ		**barra**, **parce**, $
<I>	<I> → **sim**	**sim**	**sim**, **parab**, **parce**, **barra**, $	**sim**
	<I> → **parab** <E> **parce** <C>	**parab**		**parab**

Símbolo	Regla	Primeros	Siguientes	Predicción
<C>	<C> → **ast**	**ast**	**sim**, **parab**, **parce**, **barra**, $	**ast**
	<C> → **sum**	**sum**		**sum**
	<C> → **int**	**int**		**int**
	<C> → λ	λ		**sim**, **parab**, **parce**, **barra**, $

(d) Analizador descendente recursivo.

```
private void simbolo_E() throws Exception {
 switch(nextToken) {
  case SIM:
  case PARAB: simbolo_O(); simbolo_Eprima(); break;
  default: String expected = "SIM o PARAB";
          throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
 }
}

private void simbolo_Eprima() throws Exception {
 switch(nextToken) {
  case BARRA: emparejar(BARRA); simbolo_O(); simboloEprima(); break;
  case PARCE:
  case EOF: break;
  default: String expected = "BARRA, PARCE o EOF";
          throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
 }
}

private void simbolo_O() throws Exception {
 switch(nextToken) {
  case SIM:
  case PARAB: simbolo_I(); simbolo_Oprima(); break;
  default: String expected = "SIM o PARAB";
          throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
 }
}

private void simbolo_Oprima() throws Exception {
 switch(nextToken) {
  case SIM:
  case PARAB: simbolo_I(); simbolo_Oprima(); break;
  case BARRA:
  case PARCE:
  case EOF: break;
  default: String expected = "SIM, PARAB, BARRA, PARCE o EOF";
          throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
 }
}
```

```
  private void simbolo_I() throws Exception {
   switch(nextToken) {
    case SIM: emparejar(SIM); break;
     case PARAB: emparejar(PARAB); simbolo_E(); emparejar(PARCE); simbolo_C();
break;
    default: String expected = "SIM o PARAB";
            throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
   }
  }

  private void simbolo_C() throws Exception {
   switch(nextToken) {
    case AST: emparejar(AST); break;
    case SUM: emparejar(SUM); break;
    case INT: emparejar(INT); break;
    case SIM:
    case PARAB:
    case PARCE:
    case BARRA:
    case EOF: break;
    default: String expected = "AST, SUM, INT, SIM, PARAB, PARCE, BARRA o EOF";
            throw new Exception("Encontrado "+nextToken+" se esperaba "+expected);
   }
  }
```

Ejercicio 3.9

Ejercicio 3.10

(a) Realize las transformaciones necesarias para expresarla en notación BNF.

Párrafo → *Frase* *ListaDeFrases* **FinDeLínea**

ListaDeFrases → *Frase* *ListaDeFrases*

ListaDeFrases → λ

Frase → *Claúsula* *ListaDeClaúsulas* **Punto**

ListaDeClaúsulas → **coma** *Claúsula* *ListaDeClaúsulas*

ListaDeClaúsulas → λ

Claúsula → *Palabra* *ListaDePalabras*

ListaDePalabras → **espacio** *Palabra* *ListaDePalabras*

ListaDePalabras → λ

Palabra → **letra** *ListaDeLetras*

ListaDeLetras → **letra** ListaDeLetras

ListaDeLetras → λ

(b) Calcule los conjuntos Primeros, Siguientes y de Predicción para las reglas y símbolos de la gramática resultante.

Símbolo	Regla	Primeros	Siguientes	Predicción
Parrafo	1	**letra**	$	**letra**
ListaDeFrases	2	**letra**	**FinDeLinea**	**letra**
	3	λ		**FinDeLinea**
Frase	4	**letra**	**letra**, **FinDeLinea**	**letra**
ListaDeClaúsulas	5	**coma**	**punto**	**coma**
	6	λ		**punto**
Claúsula	7	**letra**	**coma**, **punto**	**letra**
ListaDePalabras	8	**espacio**	**coma**, **punto**	**espacio**
	9	λ		**coma**, **punto**
Palabra	10	**letra**	**espacio**, **coma**, **punto**	**letra**
ListaDeLetras	11	**letra**	**espacio**, **coma**, **punto**	**letra**
	12	λ		**espacio**, **coma**, **punto**

Ejercicio 3.11

Ejercicio 3.12

(a) Realice las transformaciones necesarias para expresarla en notación BNF.

Type → **type id eq** *Decl* **semicolon**

BaseDecl → **id**

BaseDecl → *List*

BaseDecl → *Tuple*

Decl → *BaseDecl*

Decl → *Decl* **rel** *BaseDecl*

List → **lbra** *Decl* **rbra**

Tuple → **lpar** *Decl CommaList* **rpar**

CommaList → **comma** *Decl CommaList*

CommaList → λ

(b) Realice las transformaciones necesarias para que cumpla la propiedad LL(1).

Type → **type** **id** **eq** *Decl* **semicolon**

BaseDecl → **id**

BaseDecl → *List*

BaseDecl → *Tuple*

Decl → *BaseDecl* *DeclPrima*

DeclPrima → **rel** *BaseDecl* *DeclPrima*

DeclPrima → λ

List → **lbra** *Decl* **rbra**

Tuple → **lpar** *Decl* *CommaList* **rpar**

CommaList → **comma** *Decl* *CommaList*

CommaList → λ

(c) Calcule los conjuntos Primeros, Siguientes y de Predicción para las reglas y símbolos de la gramática resultante.

	Primeros	**Siguientes**	**Predicción**
Type	**type**	$	**type**
BaseDecl	**id**	**semicolon**, **rbra**, **rpar**, **comma**, **rel**	**id**
BaseDecl	**lbra**		**lbra**
BaseDecl	**lpar**		**lpar**
Decl	**id**, **lbra**, **lpar**	**semicolon**, **rbra**, **rpar**, **comma**	**id**, **lbra**, **lpar**
DeclPrima	**rel**	**semicolon**, **rbra**, **rpar**, **comma**	**rel**
DeclPrima	λ		**semicolon**, **rbra**, **rpar**, **comma**
List	**lbra**	**semicolon**, **rbra**, **rpar**, **comma**, **rel**	**lbra**
Tuple	**lpar**	**semicolon**, **rbra**, **rpar**, **comma**, **rel**	**lpar**
CommaList	**comma**	**rpar**	**comma**
CommaList	λ		**rpar**

(d) Construya el Analizador Sintáctico Descente Recursivo de la gramática obtenida

```
class ASDR {
  private Token nextToken;
  private Lexer lexer;

  public boolean parse(String filename) throws Exception {
    this.lexer = new Lexer(filename);
    this.nextToken = lexer.getNextToken();
    try { parseType(); match(EOF); return true; }
    catch(Exception ex) { return false; }
  }

  private void match(int kind) throws ParseException {
    if(nextToken.getKind() == kind) nextToken = lexer.getNextToken();
   else throw new ParseException(nextToken,kind);
  }

  private void parseType() throws ParseException {
    int[] expected = { TYPE };
    switch(nextToken.getKind()) {
      case TYPE:
        match(TYPE);
        match(ID);
        match(EQ);
        parseDecl();
        match(SEMICOLON);
        break;
      default:
        throw new ParseException(nextToken,expected);
    }
  }

  private void parseBaseDecl() throws ParseException {
    int[] expected = { ID, LBRA, LPAR }
    switch(nextToken.getKind()) {
      case ID:
        match(ID);
        break;
      case LBRA:
        parseList();
        break;
      case LPAR:
        parseTuple();
        break;
      default:
        throw new ParseException(nextToken,expected);
    }
  }

  private void parseDecl() throws ParseException {
    int[] expected = { ID, LBRA, LPAR }
    switch(nextToken.getKind()) {
      case ID:
      case LBRA:
      case LPAR:
        parseBaseDecl();
        parseDeclPrima();
        break;
      default:
        throw new ParseException(nextToken,expected);
    }
  }
```

```
    private void parseDeclPrima() throws ParseException {
      int[] expected = { REL, SEMICOLON, RPAR, RBRA, COMMA }
      switch(nextToken.getKind()) {
        case REL:
          match(REL);
          parseBaseDecl();
          parseDeclPrima();
          break;
        case SEMICOLON:
        case RPAR:
        case RBRA:
        case COMMA:
          break;
        default:
          throw new ParseException(nextToken,expected);
      }
    }

    private void parseList() throws ParseException {
      int[] expected = { LBRA };
      switch(nextToken.getKind()) {
        case LBRA:
          match(LBRA);
          parseDecl();
          match(RBRA);
          break;
        default:
          throw new ParseException(nextToken,expected);
      }
    }

    private void parseTuple() throws ParseException {
      int[] expected = { LPAR };
      switch(nextToken.getKind()) {
        case LPAR:
          match(LPAR);
          parseDecl();
          parseCommaList();
          match(RPAR);
          break;
        default:
          throw new ParseException(nextToken,expected);
      }
    }

    private void parseCommaList() throws ParseException {
      int[] expected = { COMMA, RPAR };
      switch(nextToken.getKind()) {
        case COMMA:
          match(COMMA);
          parseDecl();
          parseCommaList();
          break;
        case RPAR:
          break;
        default:
          throw new ParseException(nextToken,expected);
      }
    }
  }
```

Ejercicio 3.13

Ejercicio 3.14

Ejercicio 3.15

(a) Realice las transformaciones necesarias para expresar la gramática en notación BNF.

NewOperator → **new id** *Constructor*

NewOperator → **new id** *Array*

NewOperator → **new type** *Array*

Constructor → **lparen** *Aux1* **rparen**

Aux1 → λ

Aux1 → **expr** *Aux2*

Aux2 → **comma expr** *Aux2*

Aux2 → λ

Array → **lbracket expr rbracket** *Aux3* *Aux4*

Aux3 → **lbracket expr rbracket** *Aux3*

Aux3 → λ

Aux4 → **lbracket rbracket** *Aux4*

Aux4 → λ

(b) Realice las transformaciones necesarias para que cumpla la propiedad LL(1).

1: NewOperator → **new** *NO1*

2: NO1 → **id** *NO2*

3: NO1 → **type** *Array*

4*: NO2* → *Constructor*

5: NO2 → *Array*

6: Constructor → **lparen** *Const1* **rparen**

7: Const1 → λ

8: Const1 → **expr** *Const2*

9: Const2 → **comma expr** *Const2*

10: Const2 → λ

11: Array → **lbracket expr rbracket** *Arr1*

12: Arr1 → **lbracket** *Arr2*

13: Arr1 → λ

14: Arr2 → **expr rbracket** *Arr1*

15: Arr2 → **rbracket** *Arr3*

16: Arr3 → λ

17: Arr3 → **lbracket rbracket** *Arr3*

(c) Calcule los conjuntos Primeros, Siguientes y de Predicción para las reglas y símbolos de la gramática resultante.

Símbolo	Regla	Primeros	Siguientes	Predicción
NewOperator	→ **new** *NO1*	**new**	$	**new**
NO1	→ **id** *NO2*	**id**	$	**id**
	→ **type** *Array*	**type**		**type**
NO2	→ *Constructor*	**lparen**	$	**lparen**
	→ *Array*	**lbracket**		**lbracket**
Constructor	→ **lparen** *Const1* **rparen**	**lparen**	$	**lparen**
Const1	→ λ	λ	**rparen**	**rparen**
	→ **expr** *Const2*	**expr**		**expr**
Const2	→ **comma expr** *Const2*	**comma**	**rparen**	**comma**
	→ λ	λ		**rparen**

Símbolo	Regla	Primeros	Siguientes	Predicción
Array	→ **lbracket expr rbracket** *Arr1*	**lbracket**	$	**lbracket**
Arr1	→ **lbracket** *Arr2*	**lbracket**	$	**lbracket**
	→ λ	λ		$
Arr2	→ **expr rbracket** *Arr1*	**expr**	$	**expr**
	→ **rbracket** *Arr3*	**rbracket**		**rbracket**
Arr3	→ λ	λ	$	$
	→ **lbracket rbracket** *Arr3*	**lbracket**		**lbracket**

(d) Construya la tabla de análisis sintáctico LL(1) de la gramática obtenida.

	new	**id**	**type**	**lparen**	**rparen**	**lbracket**	**rbracket**	**expr**	**comma**	**$**
NewOperator	R1									
NO1		R2	R3							
NO2				R4		R5				
Constructor				R6						
Const1					R7			R8		
Const2					R10				R9	
Array						R11				
Arr1						R12				R13
Arr2							R15	R14		
Arr3						R17				R16

Ejercicio 3.16

Ejercicio 3.17

Ejercicio 3.18

Ejercicio 3.19

Ejercicio 3.20

Ejercicio 3.21

DISEÑO DE GRAMÁTICAS LL(1)

Ejercicio 3.22

Gramática LL(1) que reconoce el formato de las claúsulas *import* de Java.

ClausulaImport → **import** *ClaseOPaquete* **pyc**

ClaseOPaquete → **id** *A*

A → λ

A → **punto** *B*

B → **id** *A*

B → **asterisco**

Ejercicio 3.23

Gramática LL(1) que reconoce el formato del operador *new* de Java.

OperadorNew → **new** *ObjetoOArray*

ObjetoOArray → *SimpleType Dimensiones*

ObjetoOArray → **id** *LlamadaODimensiones*

LlamadaODimensiones → *Llamada*

LlamadaODimensiones → *Dimensiones*

Llamada → **lparen** *ListaExpresiones* **rparen**

ListaExpresiones → λ

ListaExpresiones → *Expression MasExpresiones*

MasExpresiones → λ

MasExpresiones → **comma** *Expression MasExpresiones*

Dimensiones → **lbracket** *Expression* **rbracket** *MasDimensiones*

MasDimensiones → λ

MasDimensiones → **lbracket** *SigueMasDimensiones*

SigueMasDimensiones → *Expression* **rbracket** *MasDimensiones*

SigueMasDimensiones → **rbracket** *MasDimensionesVacias*

MasDimensionesVacias → λ

MasDimensionesVacias → **lbracket rbracket** *MasDimensionesVacias*

Ejercicio 3.24

Gramática LL(1) que reconoce el formato de la declaración de variables en C.

Declaración → *TypeIdentifier ListaVariables* **pyc**
ListaVariables → *Variable MasVariables*
MasVariables → λ
MasVariables → **coma** *Variable MasVariables*
Variable → *VarId Asignación*
Asignación → λ
Asignación → **igual** *Expression*
VarId → *Asteriscos* **id** *Dimensiones*
Asteriscos → λ
Asteriscos → **asterisco** *Asteriscos*
Dimensiones → λ
Dimensiones → **lbracket** *Expression* **rbracket** *Dimensiones*

Ejercicio 3.25

La traducción directa de las reglas (a), (b), (c) y (d) conduce a la siguiente gramática:

Tipo → **id**
Tipo → **lbracket** *Tipo* **rbracket**
Tipo → **lparen** *ListaTipos* **rparen**
Tipo → *Tipo* **arrow** *Tipo*
ListaTipos → *Tipo MásTipos*
MásTipos → λ
MásTipos → **comma** *Tipo MásTipos*

La gramática anterior es correcta en cuanto a las tres primeras descripciones de Tipo, pero presenta un grabe problema en la cuarta regla. Como hemos visto en el tema 3, las construcciones de la forma " E → E ... E " generan gramáticas ambiguas y, por tanto, la gramática anterior no puede ser LL(1).

Para resolver este problema es necesario replantear la gramática para evitar la ambigüedad. Las relaciones permiten describir tipos de la forma "*Tipo* **arrow** *Tipo* **arrow** *Tipo*", es decir, listas de tipos separadas por el token **arrow**. En este caso, los tipos se refieren a cualquiera de las tres primeras formas de declaración (tipos predefinidos, listas y tuplas). En la siguiente gramática estas tres formas se describen por medio del símbolo *TipoBase*.

Tipo → *TipoBase Relaciones*

Relaciones → λ

Relaciones → **arrow** *TipoBase Relaciones*

TipoBase → **id**

TipoBase → **lbracket** *Tipo* **rbracket**

TipoBase → **lparen** *ListaTipos* **rparen**

ListaTipos → *Tipo MásTipos*

MásTipos → λ

MásTipos → **comma** *Tipo MásTipos*