# Árboles de Sintaxis Abstracta en ANTLR

Antlr permite construir árboles de sintaxis abstracta (ASA) mediante anotaciones en la gramática indicando qué tokens deben tratarse como raíces de subárboles, cuáles son tokens hojas y cuáles deben ignorarse.

**Notación para describir Árboles de Sintaxis Abstracta.**

La notación usada para representar un ASA es #(A B C) donde A es la raíz y B y C son los hijos. Esta notación puede ser anidada permitiendo construir árboles con una estructura más compleja. Por ejemplo: #(A B #(C D E)) es un árbol con A de ráiz y B y el subárbol #(C D E) como sus hijos.

La opción buildAST de Antlr habilita la construcción automática de ASAs. El siguiente ejemplo nos muestra una gramática con la opción de construir ASAs:

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de ASA
}

instrucciones : (expresion ";")*;

expresion : exp_mult (("+"|"-") exp_mult)*;

exp_mult : exp_base (("*"|"/") exp_base)*;

exp_base : NUMERO
        | "(" expresion ")"
        ;
```

Las anotaciones para construir ASAs son:

- Cualquier token con sufijo ^ se considera raiz del árbol. Por ejemplo, A B^C^ construye el árbol #(C #(B A)).
- Un token con sufijo ! no se incorpora al árbol.
- Una referencia a regla (símbolo no terminal) con sufijo ! indica que el árbol de dicha regla no se incorpora al árbol que se está construyendo.
- Una definición de regla con ! indica que no se construye árbol para dicha regla. Por ejemplo begin !: INT PLUS i:INT {#begin = #(PLUS INT i);} ;
- Una alternativa prefijada con ! inhabilita la construcción del árbol para dicha alternativa.

- Hay varias formas (no automáticas) de construir un nodo: #[TYPE] y #[TYPE, "text"]

Ejemplos de construcción de árboles:

(Construcción manual de árbol) r! : a:A {#r = #a;} ;

(Adición de nodos imaginarios) decl : (TYPE ID)+ {#decl = #(#[DECL, "decl"],#decl); } ;

El siguiente ejemplo muestra una construcción híbrida (manual y automática) del ASA:

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_INST;
}

instrucciones : (expresion ";"!)*
  {#instrucciones = #(#[LISTA_INST,"LISTA_INST"],#instrucciones);} //manual
    ;
expresion : exp_mult (("+"^|"-"^) exp_mult)*; //automática

exp_mult : exp_base (("*"^|"/"^) exp_base)*; //automática

exp_base : NUMERO       //automática
       | "("! expresion ")"!  //automática
       ;
```

**Interfaz AST e Implementaciones**

Antlr predefine una interfaz llamada AST (abstract sintax tree) para representar el tipo abstracto de datos árbol de sintáxis abstracta.

La interfaz AST consta de las siguientes operaciones:

```
public interface AST {

   /** añadir un hijo a la derecha del nodo 'this'*/
   public void addChild(AST c);

  /** igualdad entre dos nodos (mismo token, mismo texto)*/
   public boolean equals(AST t);
```

```
/** dos listas de nodos o subárboles iguales en estructura y contenido*/
public boolean equalsList(AST t);

/** dos listas de nodos o subárboles parcialmente iguales. 'this' contiene a 't'*/
public boolean equalsListPartial(AST t);

/** dos nodos o subárboles iguales*/
public boolean equalsTree(AST t);

/** dos nodos o subárboles parcialmente iguales. 'this' contiene a 't'*/
public boolean equalsTreePartial(AST t);

/**enumeración de todos los emparejamientos de 'tree' en 'this'*/
public ASTEnumeration findAll(AST tree);

/** enumeración de todos los emparejamientos parciales  de 'tree' en 'this'*/
public ASTEnumeration findAllPartial(AST subtree);

/** primer hijo de 'this'. Null si no existiese */
public AST getFirstChild();

/** siguiente hermano de 'this' */
public AST getNextSibling();

/** texto asociado el token del nodo 'this' */
public String getText();

/** token asociado al nodo 'this' */
public int getType();

/** Número de hijos del nodo 'this'*/
public int getNumberOfChildren();

/** inicializar el nodo 'this' con token 't' y texto 'txt' */
public void initialize(int t, String txt);

/** inicializar el nodo 'this' con el contenido del árbol 't' */
public void initialize(AST t);

/** inicializar el nodo 'this' con el contenido del token  't'*/
public void initialize(Token t);

/** establecer 'c' como primer hijo de 'this' */
public void setFirstChild(AST c);

/** establecer 'n' como siguiente hermano de 'this' */
public void setNextSibling(AST n);

/** asociar el texto 'text' al nodo 'this' */
public void setText(String text);
```

```
   /** asociar el tipo token 'ttype' al nodo 'this' */
   public void setType(int ttype);

   /** convertir 'this' a formato string imprimible*/
   public String toString();

   /** convertir 'this' en una lista en formato imprimible */
   public String toStringList();

   /** convertir 'this' en árbol en formato imprimible */
   public String toStringTree();
}
```

La clase BaseAST implementa dicha interfaz. La clase CommonAST especializa esta última y permite que cada nodo del ASA contenga el texto asociado al token reconocido, el propio token ( entero ) y una lista de hijos.

**Arboles de Sintaxis Abstracta**

Antlr permite recorrer y transformar ASAs mediante una clase predefinida denominada TreeParser. Los recorridos son interesantes para realizar chequeos semánticos y las transformaciones son interesantes para realizar optimizaciones y generar código.

Cualquier tipo de árbol que implemente la interfaz AST puede ser recorrido en Antlr. Las operaciones principales para realizar recorridos son:

```
/** primer hijo de 'this'. Null si no existiese */
   public AST getFirstChild();

   /** siguiente hermano de 'this' */
   public AST getNextSibling();
```

**Ejemplo (calculadora)**

Se propone un parser para construir una representación intermedia (árbol) de la expresión de entrada y un parser árbol para recorrer esta representación intermedia computando su valor correspondiente.

```
class Anasint extends Parser;

options{
  buildAST = true;  //construcción automáticas de ASA
}

expr :  mexpr  (PLUS^ mexpr)* SEMI!  ;
```

```
mexpr : atom (STAR^ atom)*    ;

atom : INT  ;
```

Los tokens PLUS y STAR son considerados raíces de árboles de ahí el símbolo ^. El token SEMI postfijado con el símbolo ! significa que dicho token no se incluye en el árbol.

El lexer (o scaner):

```
class Analex extends Lexer;

options{
        importVocab = Anasint;
}

WS:  (' ' | '\t' | '\n' | '\r' ) {_ttype = Token.SKIP; } ;
LPAREN : '(' ;
RPAREN : '(' ;
PLUS : '+' ;
STAR : '*' ;
SEMI : ';' ;
INT: ('0' .. '9')+ ;
```

El parser árbol :

```
class CalcTreeParser extends TreeParser;

expr returns [int r]
{
  int a,b;
  r = 0;
}
     : #(PLUS a=expr b=expr) {r=a+b ;}
     | #(STAR a=expr b=expr) {r=a*b ;}
     | i:INT {r=Integer.parserInt(i.getText( ));}
     ;
```

El código necesario para lanzar el parser y el parser árbol:

```
import java.io.* ;
import antlr.CommonAST ;

class Prog{
  public static void main(String [ ] args) {
    try{
         FileInputStream f = new FileInputStream("entrada.txt");
        Analex lexer = new Analex(f);
```

```
        Anasint parser = new Anasint(lexer);
        parser.expr( );
        antlr.CommonAST t = (CommonAST)parser.getAST( );
        System.out.println(t.toStringList());
        Anasint2 walker = new Anasint2( );
        int r = walker.expr(t);
        System.out.prinlt("El valor de la expresion es: " + r);
   }catch(Exception e)   { System.err.println("Excepcion" + e); }
  }
}
```

**Transformaciones**: Antlr da la posibilidad de transformar los árboles previamente construidos por un parser.

Ejemplo:

```
class Anasint2 extends TreeParser;
options{
  buildAST = true; //modo transformación
  importVocab = Anasint;
}

expr : ! #(PLUS left:expr right:expr)
        {
         if (#right.getType( )==INT && Integer.parseInt(#right.getText( ))==0)
           #expr = #left;
         else
           if (#left.getType( )==INT && Integer.parseInt(#left.getText( ))==0)
             #expr = #right;
           else
             #expr = #(PLUS, left, right);
       }
       | #(STAR expr expr)   //usa autotransformacion
       | i: INT
       ;
```

El código necesario para lanzar el parser y el parser árbol:

```
import java.io.* ;
import antlr.CommonAST ;

class Prog{
  public static void main(String [ ] args) {
    try{
      FileInputStream f = new FileInputStream("entrada.txt");
       Analex lexer = new Analex(f);
       Anasint parser = new Anasint(lexer);
       parser.expr( );
       antlr.CommonAST t = (CommonAST)parser.getAST( );
```

```
        System.out.println(t.toStringList());
        Anasint2 walker = new Anasint2( );
        walker.expr(t);
        antlr.CommonAST t2 = (CommonAST)walker.getAST( );
        System.out.println(t2.toStringList());
    }catch(Exception e)   { System.err.println("Excepcion" + e); }
   }
}
```

**Predicados sintácticos**

También se pueden utilizar sobre árboles para decidir sobre posibles alternativas cuando lookahead es una herramienta insuficiente.

Ejemplo:
```
expr : ( #(MINUS expr expr) ) =>  #(MINUS expr expr)
                | (#(MINUS expr) ) =>  #(MINUS expr)
               ...
                ;
```

**Predicados semánticos**

Si se sitúan al principio de una alternativa actúan como predicciones. Cuando se sitúan en cualquier otra posición de  mitad de una alternativa actúan como centinelas de condiciones elevando excepciones en caso de incumplimiento.

**Algunas observaciones sobre la notación de Árboles de Sintaxis Abstracta**

Es importante señalar que el ASA no usa comas para separar sus elementos cuando aparece en la parte derecha de la regla y usa comas para separar sus elementos cuando aparece dentro de una acción. Observando el ejemplo de la calculadora anterior:

```
class CalcTreeWalker extends TreeParser;
options{
  buildAST = true //modo transformación
}

expr : ! #(PLUS left:expr right:expr) //sin uso de comas
       { if (#right.getType( )==INT && Integer.parserInt(#right.getText( ))==0)
         {
           #expr = #left;
         }
         else if (#left.getType( )==INT&&Integer.parserInt(#left.getText( ))==0)
               {
                 #expr = #right;
               }
               else #expr = #(PLUS, left, right); //uso de comas
      }
      | #(STAR left right)   //usa autotransformacion
      | i: INT
      ;
```

# Árboles de Sintaxis Abstracta en Antlr. Ejemplos

En esta sección, mostramos en 3 ejemplos los recursos de Antlr para construir y procesar árboles de sintaxis abstracta (ASAs).

**Ejemplo 1: Construcción de un árbol de sintaxis abstracta.**

Analizador sintáctico Antlr (Anasint.g) :

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_EXPR;
}

prog : (expresion ";"!)*
   {#prog = #(#[LISTA_EXPR,"EXPRESIONES"],#prog);} //manual
      ;
expresion : exp_mult (("+"^|"-"^) exp_mult)*; //automática

exp_mult : exp_base (("*"^|"/"^) exp_base)*; //automática

exp_base : NUMERO       //automática
       | IDENT       //automática
       | "("! expresion ")"!  //automática
        ;
```

Analizador léxico Antlr (Analex.g) :

```
class Analex extends Lexer;
options{
        importVocab = Anasint;   // Importación del conjunto de tokens
}
BLANCO : (' '|'\t'|'\r\n') {$setType(Token.SKIP);};

protected DIGITO : ('0'..'9');
protected LETRA : ('a'..'z')|('A'..'Z');

OPERADOR : '+'|'-'|'*'|'/';

PARENTESIS : '('|')';

SEPARADOR : ';';

ASIG : ":=";
```

```
NUMERO : (DIGITO)+('.'(DIGITO)+)?;

IDENT : LETRA (LETRA | DIGITO)* ;
```

Programa que utiliza el analizador anterior mostrando el ASA construido (Prog.java):

```
import java.io.*;
import antlr.collections.AST;
import antlr.*;

public class Prog {
	public static void main(String args[]){
		try{
			FileInputStream fis=
				new FileInputStream("entrada.txt");
			Analex analex = new Analex(fis);
			Anasint anasint = new Anasint(analex);

			anasint.prog();
			CommonAST a = (CommonAST)anasint.getAST();
			System.out.println("Resultado ASA: "+a.toStringList());
		}catch(ANTLRException ae){
			System.err.println(ae.getMessage());
		}
		catch (FileNotFoundException fnfe){
			System.err.println("No se encontró el fichero");
		}
	}
}
```

Flujo de entrada (entrada.txt):

```
3+5;
3-5;
a +(2-9);
b+1;
1;
```

Resultado ejecución programa Prog sobre entrada.txt:

```
ASA: ( EXPRESIONES ( + 3 5 ) ( - 3 5 ) ( + a ( - 2 9 ) ) ( + b 1 ) 1 )
```

Analizador sintáctico Antlr (Anasint.g) :

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_EXPR;
}

prog : (expresion ";"!)*
  {#prog = #(#[LISTA_EXPR,"EXPRESIONES"],#prog);} //manual
    ;
expresion : exp_mult (("+"^|"-"^) exp_mult)*; //automática

exp_mult : exp_base (("*"^|"/"^) exp_base)*; //automática

exp_base : !n:NUMERO {#exp_base=#(#[NUMERO, "Const: "], #n);}  //manual
       | !i:IDENT {#exp_base=#(#[IDENT, "Var: "], i);}  //manual
       | "("! expresion ")"!  //automática
       ;
```

Resultado ejecución programa Prog sobre entrada.txt:

```
ASA: ( EXPRESIONES ( + ( Const:  3 ) ( Const:  5 ) ) ( - ( Const:  3 )
( Const:  5 ) ) ( + ( Var:  a ) ( - ( Const:  2 ) ( Const:  9 ) ) ) (
+ ( Var:  b ) ( Const:  1 ) ) ( Const:  1 ) )
```

Analizador sintáctico Antlr (Anasint.g) :

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_EXPR;
}

prog : (expresion ";"!)*
  {#prog = #(#[LISTA_EXPR,"EXPRESIONES"],#prog);} //manual
    ;
expresion : exp_mult (("+"^|"-"^) exp_mult)*; //automática

exp_mult ! : exp_base (("*"^|"/"^) exp_base)*; //automática
```

```
exp_base : !n:NUMERO {#exp_base=#(#[NUMERO, "Const: "], #n);}  //manual
        | !i:IDENT {#exp_base=#(#[IDENT, "Var: "], i);}  //manual
        | "("! expresion ")"!  //automática
        ;
```

Flujo de entrada (entrada.txt):

```
3+5;
3-5;
a *(2-9);
b+1;
1;
```

Resultado ejecución programa Prog sobre entrada.txt:

```
ASA: ( EXPRESIONES + - + )
```

**Ejemplo 2: Uso de ASAs mediante la interfaz AST**

Analizador sintáctico Antlr (Anasint.g) :

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_EXPR;
}

prog : (expresion ";"!)*
   {#prog = #(#[LISTA_EXPR,"EXPRESIONES"],#prog);} //manual
      ;
expresion : exp_mult (("+"^|"-"^) exp_mult)*; //automática

exp_mult  : exp_base (("*"^|"/"^) exp_base)*; //automática

exp_base : NUMERO //automática
       | IDENT  //automática
       | "("! expresion ")"!  //automática
       ;
```

Analizador léxico Antlr (Analex.g) :

```
class Analex extends Lexer;
options{
        importVocab = Anasint;   // Importación del conjunto de tokens
}
BLANCO : (' '|'\t'|'\r\n') {$setType(Token.SKIP);};

protected DIGITO : ('0'..'9');
protected LETRA : ('a'..'z')|('A'..'Z');

OPERADOR : '+'|'-'|'*'|'/';

PARENTESIS : '('|')';

SEPARADOR : ';';

ASIG : ":=";

NUMERO : (DIGITO)+('.'(DIGITO)+)?;

IDENT : LETRA (LETRA | DIGITO)* ;
```

Programa que utiliza el analizador anterior y realiza un recorrido del ASA (Prog.java):

```
import java.io.*;
import antlr.collections.AST;
import antlr.*;

public class Prog {
	private static void recorre(AST a){
		if (a!=null) {
			System.out.println("Lexema:	"+a.getText()+"	Token:
"+a.getType());
			recorre(a.getFirstChild());
			recorre(a.getNextSibling());
		}
	}
	public static void main(String args[]){
		try{
			FileInputStream fis=
				new FileInputStream("entrada.txt");
			Analex analex = new Analex(fis);
			Anasint anasint = new Anasint(analex);

			anasint.prog();
			CommonAST a = (CommonAST)anasint.getAST();
			recorre(a);  //primero en profundidad
			System.out.println("ASA:"+a.toStringTree());
		}catch(ANTLRException ae){
			System.err.println(ae.getMessage());
		}
		catch (FileNotFoundException fnfe){
			System.err.println("No se encontró el fichero");
		}
	}
}
```

Flujo de entrada (entrada.txt):

```
3+5;
3-5;
a +(2-9);
b+1;
1;
```

Resultado ejecución programa Prog sobre entrada.txt:

```
Lexema: EXPRESIONES Token: 4
Lexema: + Token: 6
```

```
Lexema: 3 Token: 10
Lexema: 5 Token: 10
Lexema: - Token: 7
Lexema: 3 Token: 10
Lexema: 5 Token: 10
Lexema: * Token: 8
Lexema: a Token: 11
Lexema: - Token: 7
Lexema: 2 Token: 10
Lexema: 9 Token: 10
Lexema: + Token: 6
Lexema: b Token: 11
Lexema: 1 Token: 10
Lexema: 1 Token: 10
ASA: ( EXPRESIONES ( + 3 5 ) ( - 3 5 ) ( * a ( - 2 9 ) ) (
+ b 1 ) 1 )
```

**Ejemplo 3: Construcción, Recorrido y Transformación de un ASA**

Analizador sintáctico Antlr (Anasint.g) para construir ASAs.

```
class Anasint extends Parser;

options{
        buildAST = true;  //construcción automática de AST
}
tokens
{
        LISTA_INSTR;
}

prog : (instr FINALIZADOR!)*
  {#prog = #([LISTA_INSTR,"INSTR"], prog);} //manual
  ;

instr : (IDENT ASIG) => asig
    | expr
    ;

asig : IDENT ASIG^ expr ;

expr : expr_mult ((MAS^|MENOS^) expr_mult)*; //automática

expr_mult  : expr_base (POR^ expr_base)*; //automática

expr_base : NUMERO //automática
       | IDENT //automática
       | PARA! expr PARC!  //automática
       ;
```

Analizador léxico Antlr (Analex.g) :

```
class Analex extends Lexer;
options{
        importVocab = Anasint;   // Importación del conjunto de tokens
}
BLANCO : (' '|'\t'|"\r\n") {$setType(Token.SKIP);};

protected DIGITO : ('0'..'9');
protected LETRA : ('a'..'z'|'A'..'Z');

MAS : '+';
MENOS : '-';
POR : '*';
PARA : '(';
PARC : ')';
FINALIZADOR : ';';
```

ASIG : ":=";

NUMERO : (DIGITO)+;
IDENT : LETRA (LETRA | DIGITO)*;

Analizador sintáctico Antlr (Anasint2.g) para recorrer y tranasformar ASAs construidos con Anasint (anasint.g)

```
class Anasint2 extends TreeParser;

options{
        importVocab  = Anasint;
        buildAST = true;  //construcción automática de AST
}

{
        private static java.util.Hashtable h=new java.util.Hashtable();

        private static int evalua(AST a){
        // pre: a debe ser un AST de tipo IDENT, NUMERO, '+', '-' o '*'

        int aux1=0; int aux2=0;
        if (a.getType()==AnasintTokenTypes.IDENT){
                if (h.get(a.getText())!=null)
      return ((Integer)h.get(a.getText())).intValue();
    else    return 0;
        }
        else if (a.getType()==AnasintTokenTypes.NUMERO)
                return Integer.parseInt(a.getText());
        else {
          aux1=evalua(a.getFirstChild());
          aux2=evalua((a.getFirstChild()).getNextSibling());
          if (a.getText().equals("+")) return aux1+aux2;
          else if (a.getText().equals("-")) return aux1-aux2;
            else return aux1*aux2;
          }
  }
  // post: si a es un AST de tipo IDENT se devuelve el valor almacenado de la variable
que representa
  // post: si a es un AST de tipo NUMERO se devuelve el valor del número que
representa
  // post: si a es un AST de tipo '+','-' o '*' se devuelve el valor de la expresión que
representa

  private static AST evalua(AST op, AST izq, AST der){
  // pre: op debe ser un ASA de tipo MAS, MENOS o POR
  // pre: izq y der son ASA de tipo expr

  antlr.CommonAST v = new antlr.CommonAST();
```

```
  Integer aux=null;

  if (op.getType()==MAS)
    aux = new Integer(evalua(izq)+evalua(der));
  else if (op.getType()==MENOS)
    aux = new Integer(evalua(izq)-evalua(der));
    else
      aux = new Integer(evalua(izq)*evalua(der));
  v.setText(aux.toString());
  v.setType(AnasintTokenTypes.NUMERO);
  return v;
  }
 // post: devuelve la evaluación de la expresión 'izq op der' en forma de ASA
}


prog : #(LISTA_INSTR instrs);

instrs : (instr)* ;

instr : asig
     | expr
     ;

asig : !#(ASIG id:IDENT e:expr)
  {  // evaluación de 'e' y transformación
     // de la asignación de la forma
     // #(ASIG id:IDENT e:expr) a la forma
     // #(id, v).

           antlr.CommonAST v = new antlr.CommonAST();
           Integer i=new Integer(evalua(e));
           h.put(id.getText(),i);
    v.setText(i.toString());
           v.setType(NUMERO);
    #asig = #(id, v);
  }
   ;

expr : !#(op1:MAS e1:expr f1:expr)
  {
        #expr=#evalua(op1,e1,f1);
  }
   | !#(op2:MENOS e2:expr f2:expr)
  {
        #expr=#evalua(op2,e2,f2);
  }
   |! #(op3:POR e3:expr f3:expr)
  {
        #expr=#evalua(op3,e3,f3);
```

```
}
  | num:NUMERO
  | ! id:IDENT
  {
        // valor de la variable id
        // 0 si la variable no se le asignó ningún valor.
        antlr.CommonAST v2 = new antlr.CommonAST();
        if (h.get(id.getText())!=null)
          v2.setText(((Integer)h.get(id.getText())).toString());
   else v2.setText("0");
   v2.setType(NUMERO);
   #expr=#v2;
  }
   ;
```

Programa que utiliza los analizadores anteriores (Prog.java):

```
import java.io.*;
import java.util.Hashtable;

import antlr.collections.AST;
import antlr.*;

public class Prog {
        private static void recorre(AST a){
                if (a!=null) {
                        recorre(a.getFirstChild());
                        recorre(a.getNextSibling());
                }
        }


        public static void main(String args[]){
                try{
                        FileInputStream f=
                                new FileInputStream("entrada.txt");
                        Analex analex = new Analex(f);
                        Anasint anasint = new Anasint(analex);

                        anasint.prog();
                        AST a = anasint.getAST();
                        System.out.println("ASA:"+a.toStringTree());

                        Anasint2 a2 = new Anasint2();
                        a2.prog(a);
                        AST b = a2.getAST();
                        System.out.println("ASA:"+b.toStringTree());

                        b = b.getFirstChild();
```

```
                        while (b!=null){
                                //System.out.println("ASA:"+b.toStringTree()+     "     >>>
Evaluación: "+evalua(b));
                                System.out.println("ASA:"+b.toStringTree());
                                b=b.getNextSibling();
                        }
                }catch(ANTLRException ae){
                        System.err.println(ae.getMessage());
                }
                catch (FileNotFoundException fnfe){
                        System.err.println("No se encontró el fichero");
                }
        }
}
```

Flujo de entrada (entrada.txt):

```
a:=5;
a:=a+1;
3*5;
a;
b;
1;
```

Resultado ejecución programa Prog sobre entrada.txt:

```
ASA: ( INSTR ( := a 5 ) ( := a ( + a 1 ) ) ( * 3 5 ) a b 1
)
ASA: ( INSTR ( a 5 ) ( a 6 ) 15 6 0 1 )
ASA: ( a 5 )
ASA: ( a 6 )
ASA: 15
ASA: 6
ASA: 0
ASA: 1
```