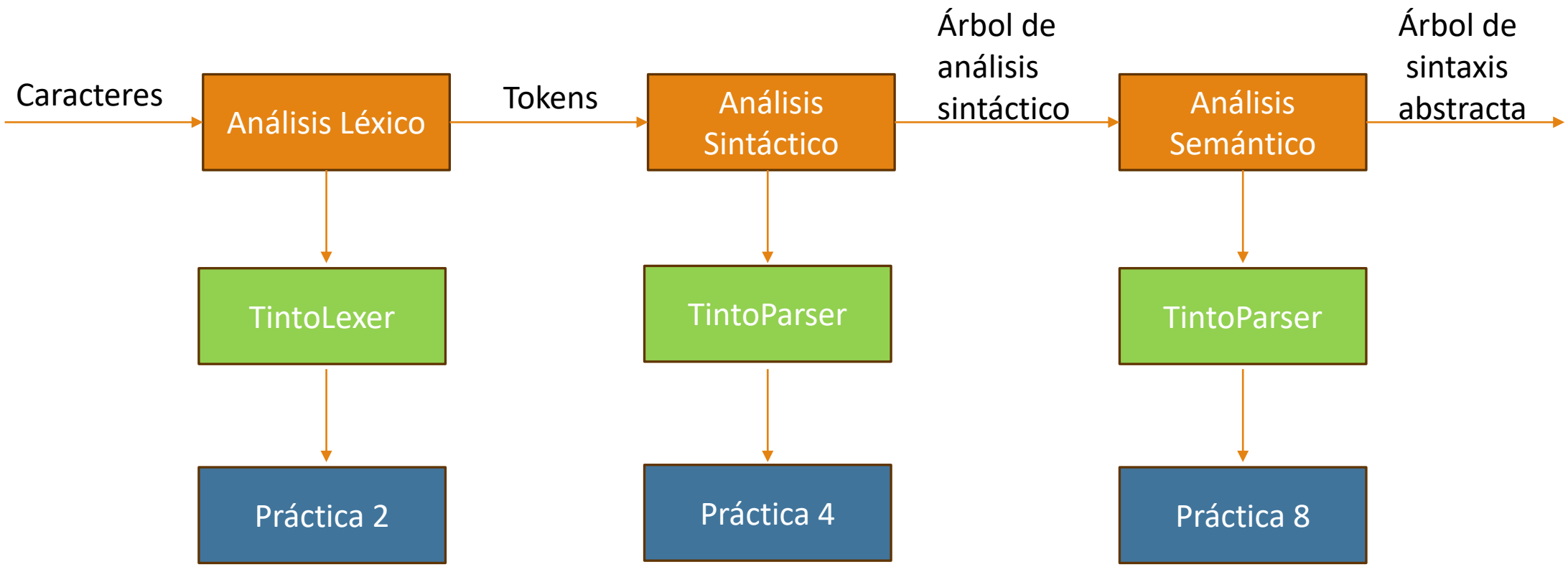
Procesadores de lenguajes. Práctica 4.

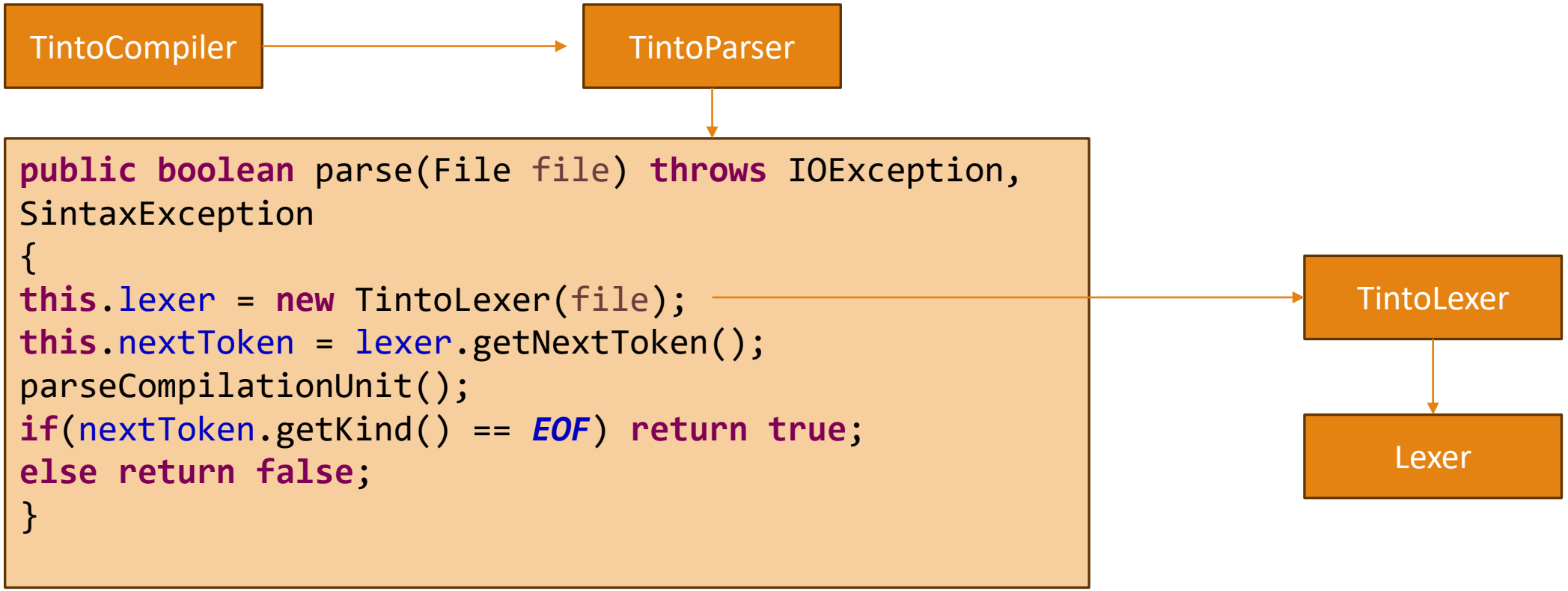
ANALIZADORES SINTÁCTICOS DESCENDENTES

Índice

1.1 Desarrollo de las prácticas



1.2 Esquema clases práctica 4



1.3 Gramática EBNF

Gramática de Tinto en notación EBNF

CompilationUnit : (ImportClause)* TintoDecl
ImportClause : **import** **identifier** **semicolon**
TintoDecl : LibraryDecl | NativeDecl
LibraryDecl : **library** **identifier** **lbrace** (Function)* **rbrace**
NativeDecl : **native** **identifier** **lbrace** (NativeFunction)* **rbrace**
Function : Access FunctionType **identifier** ArgumentDecl FunctionBody
NativeFunction : Access FunctionType **identifier** ArgumentDecl **semicolon**
Access : **public** | **private**
FunctionType : Type | **void**
Type : **int** | **char** | **boolean**
ArgumentDecl : **lparen** (Argument (**comma** Argument)*)? **rparen**
Argument : Type **identifier**
FunctionBody : **lbrace** (Statement)* **rbrace**

1.3 Gramática EBNF

Gramática de Tinto en notación EBNF

Statement : (Decl | IdStm | IfStm | WhileStm | ReturnStm | NoStm | BlockStm)
Decl : Type **identifier** Assignement (**comma** **identifier** Assignement)* **semicolon**
Assignement : (**assign** Expr)?
IfStm : **if** **lparen** Expr **rparen** Statement (**else** Statement)?
WhileStm : **while** **lparen** Expr **rparen** Statement
ReturnStm : **return** (Expr)? **semicolon**
NoStm : **semicolon**
IdStm : **identifier** (Assignement | FunctionCall | **dot** **identifier** FunctionCall) **semicolon**
BlockStm : **lbrace** (Statement)* **rbrace**

1.3 Gramática EBNF

Gramática de Tinto en notación EBNF

Expr : AndExpr (**or** AndExpr)*
AndExpr : RelExpr (**and** RelExpr)*
RelExpr : SumExpr (RelOp SumExpr)?
RelOp : (**eq** | **ne** | **gt** | **ge** | **lt** | **le**)
SumExpr : UnOp ProdExpr (SumOp ProdExpr)*
UnOp : (**not** | **minus** | **plus**)?
SumOp : (**minus** | **plus**)
ProdExpr : Factor (MultOp Factor)*
MultOp : (**prod** | **div** | **mod**)
Factor : (Literal | Reference | **lparen** Expr **rparen**)
Literal : (**integer_literal** | **char_literal** | **true** | **false**)
Reference : **identifier** (FunctionCall | **dot** **identifier** FunctionCall)?
FunctionCall : **lparen** (Expr (**comma** Expr)*)? **rparen**

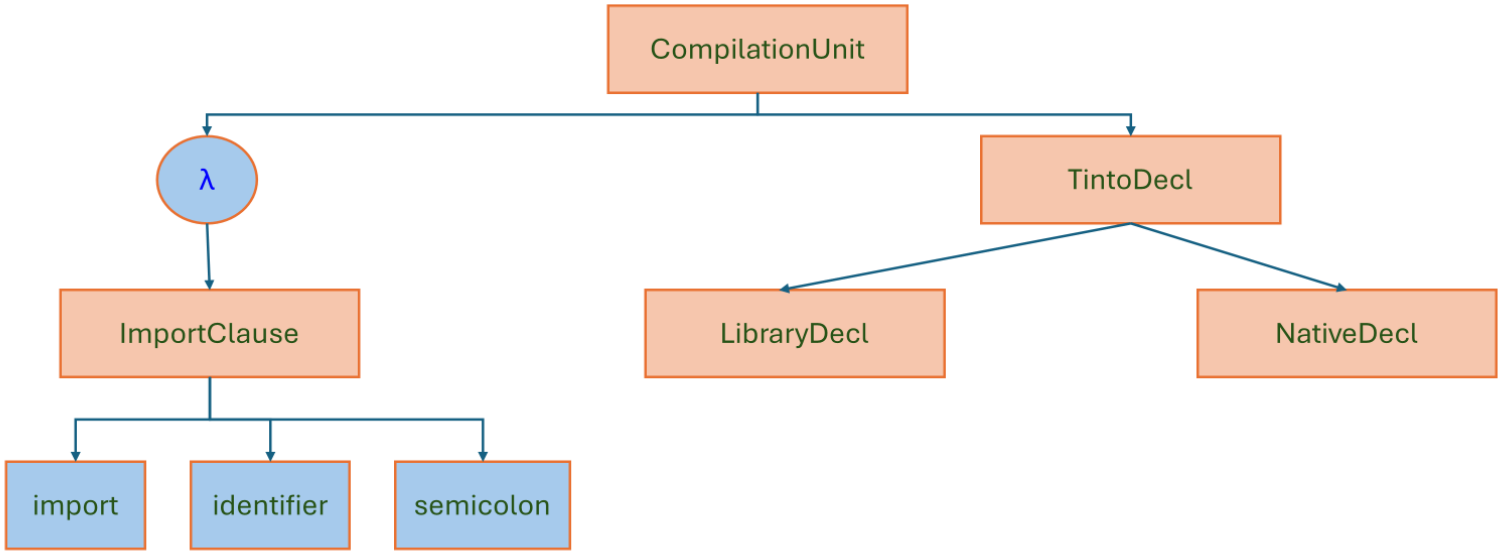
1.4 Calculo predicciones

Regla	Predicción
CompilationUnit ::= ImportClauseList TintoDecl	**import**, **library, native**
ImportClauseList ::= ImportClause ImportClauseList	**import**
ImportClauseList ::= **lambda**	**library**
ImportClause ::= **import identifier semicolon**	**import**
TintoDecl ::= LibraryDecl	**library**
TintoDecl ::= NativeDecl	**native**
LibraryDecl ::= **library identifier lbrace** FunctionList **r brace**	**library**

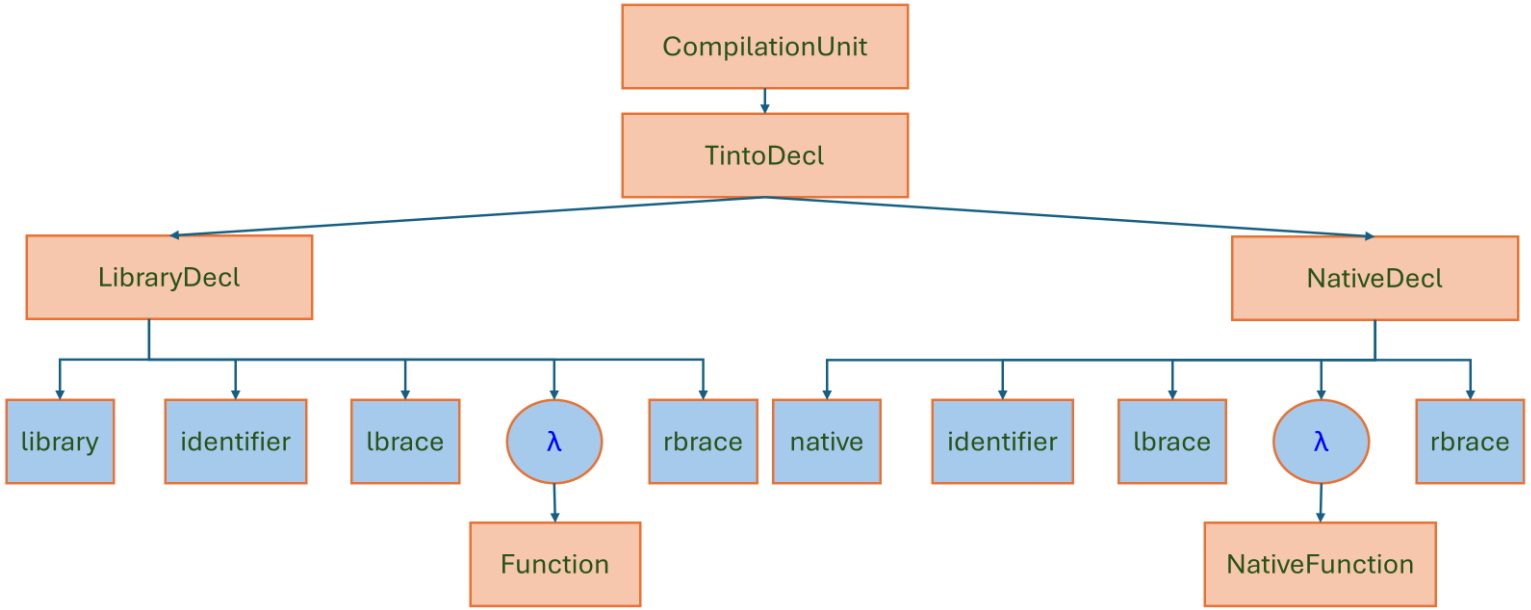
1.4 Calculo predicciones

Regla	Predicción
FunctionList ::= Function FunctionList	**public**, **private**
FunctionList ::= **lambda**	**rbrace**
Function ::= Access FunctionType **identifier** ArgumentDecl FunctionBody	**public**, **private**
NativeDecl ::= **native** **identifier** **lbrace** NativeFunctionList **rbrace**	**native**
NativeFunctionList ::= NativeFunction NativeFunctionList	**public**, **private**

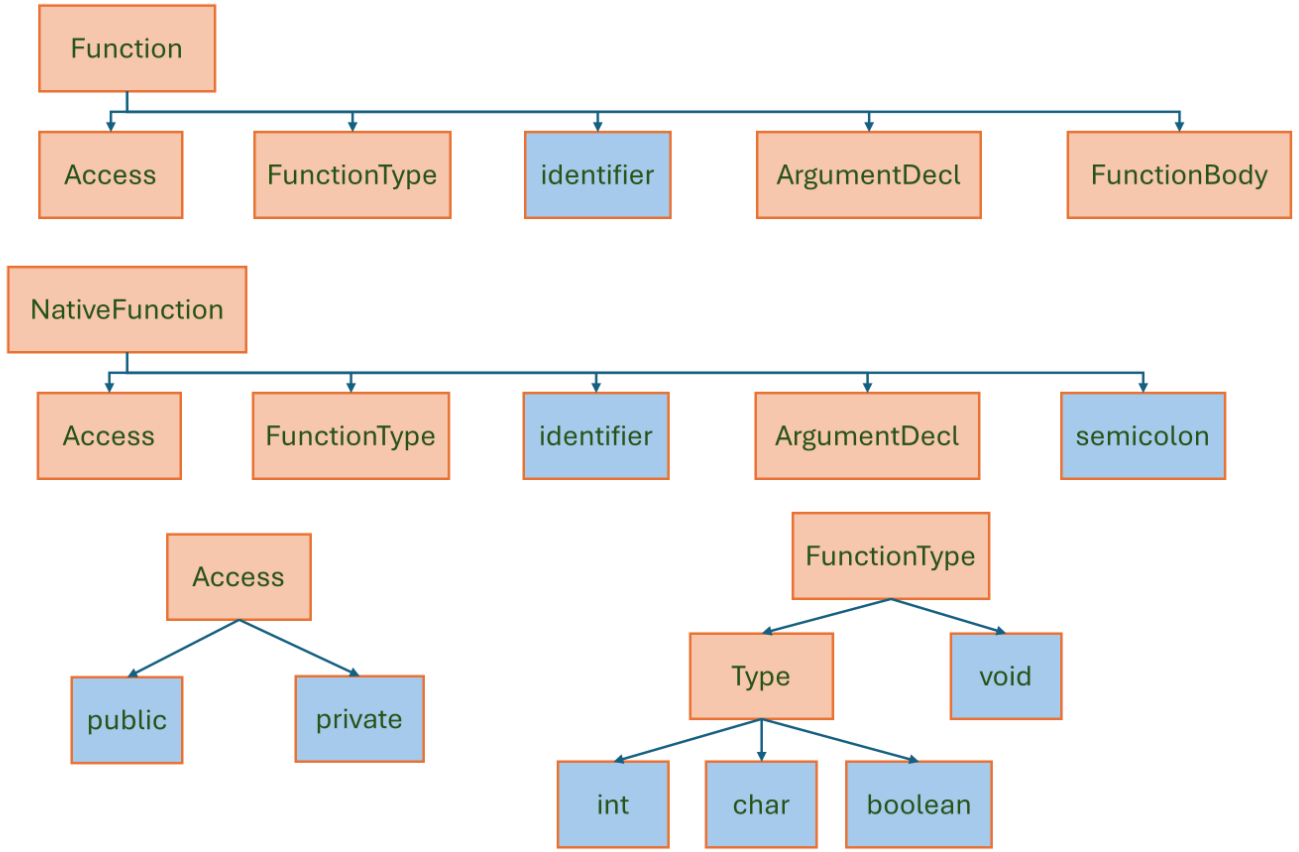
1.5 TintoParser Esquemas



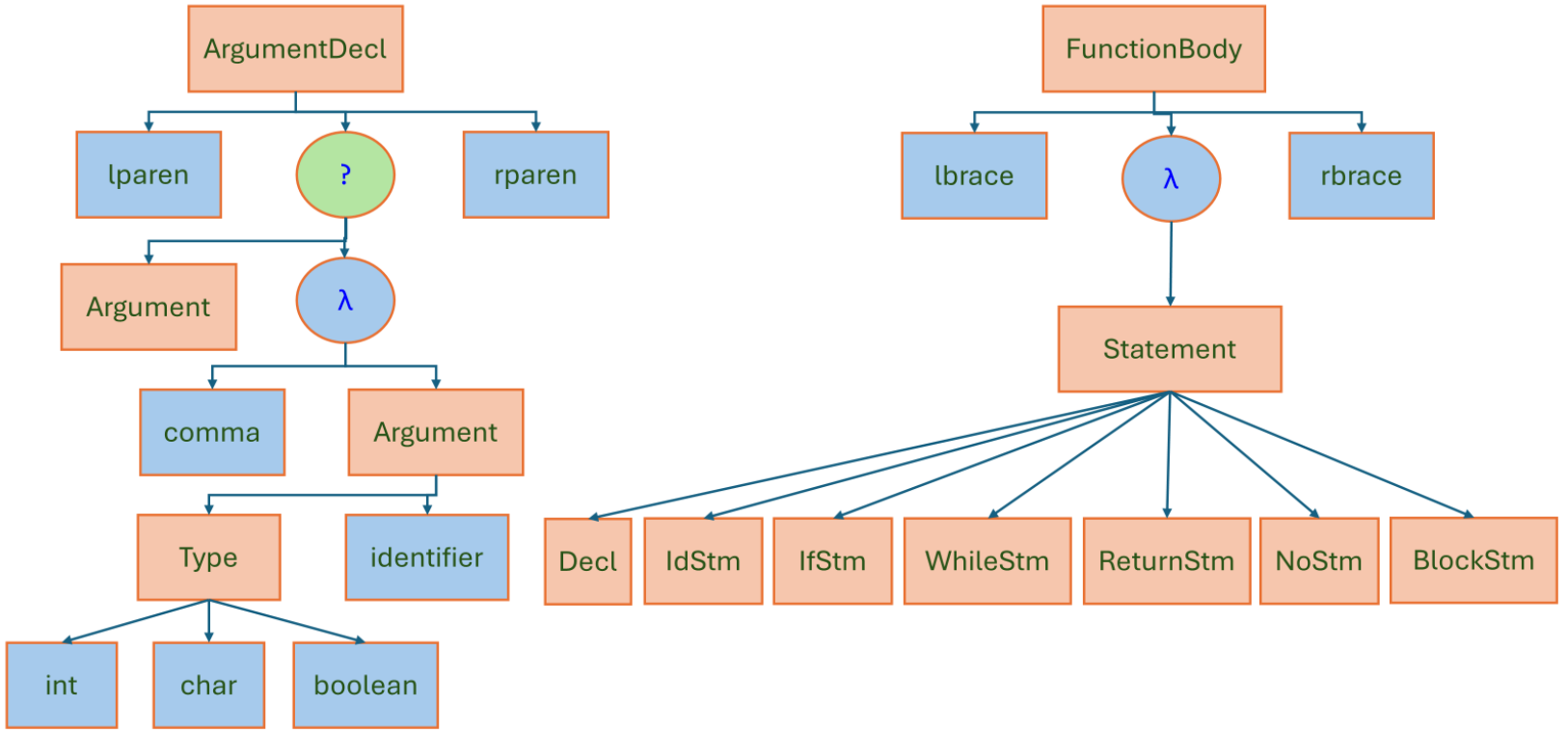
1.5 TintoParser



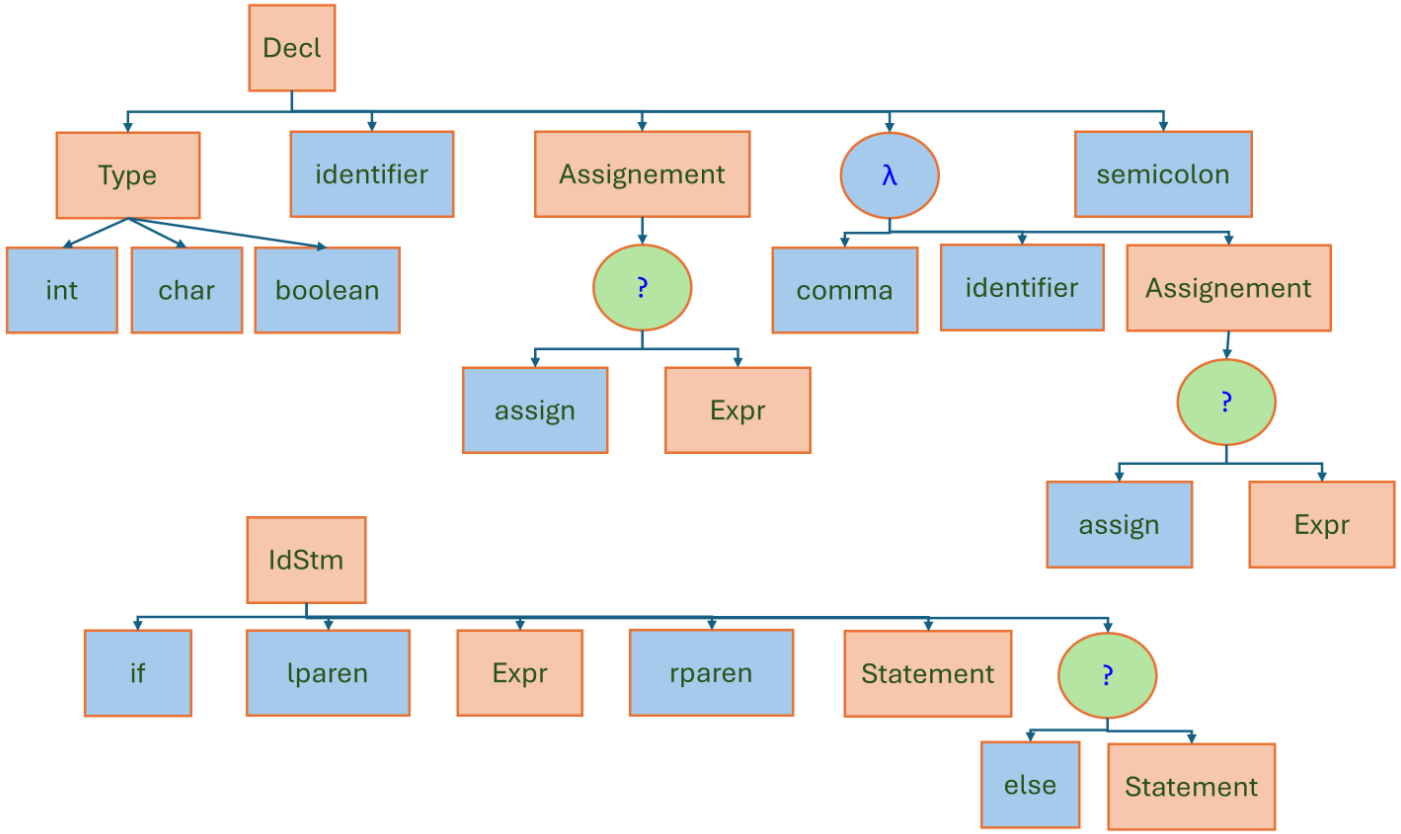
1.5 TintoParser



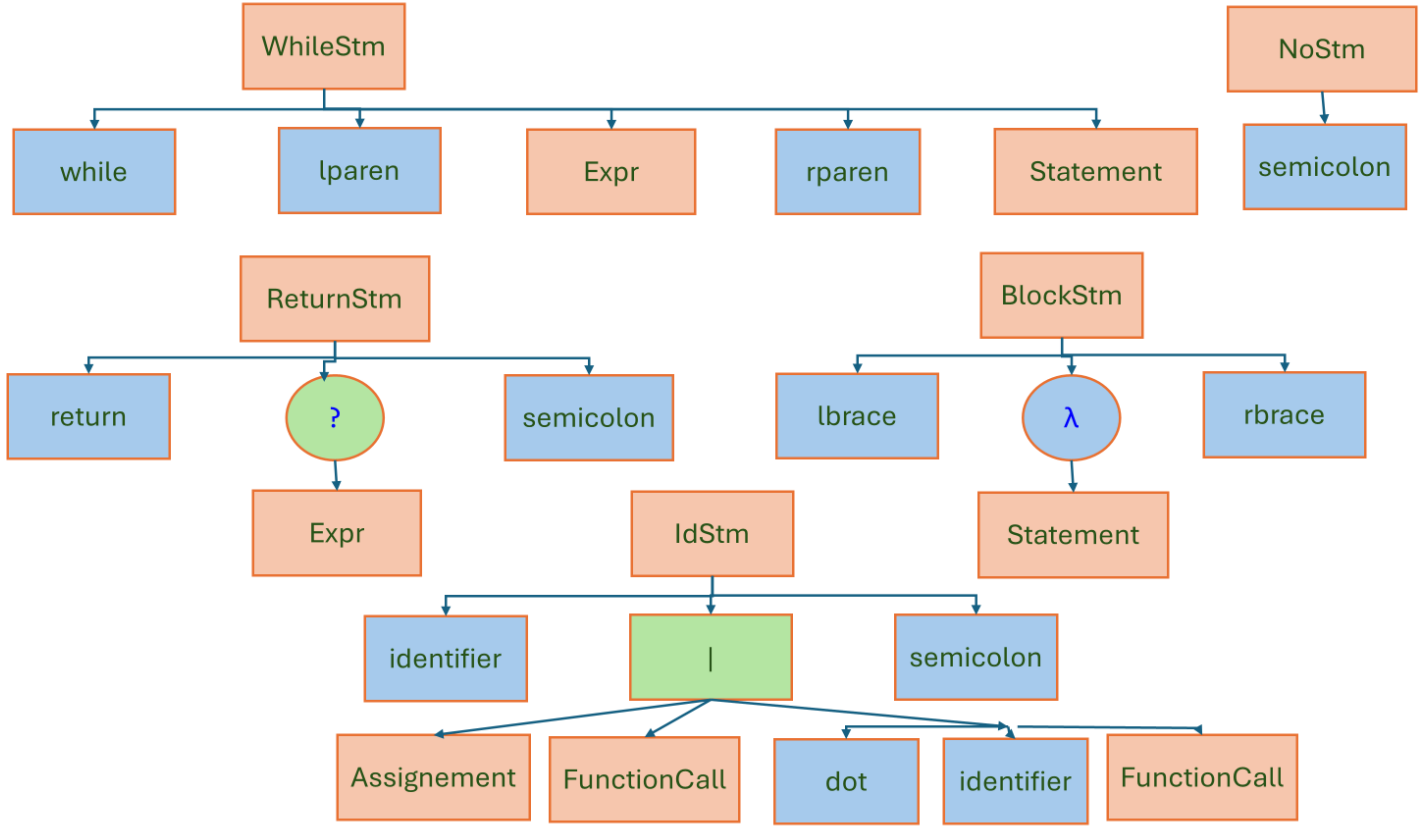
1.5 TintoParser



1.5 TintoParser

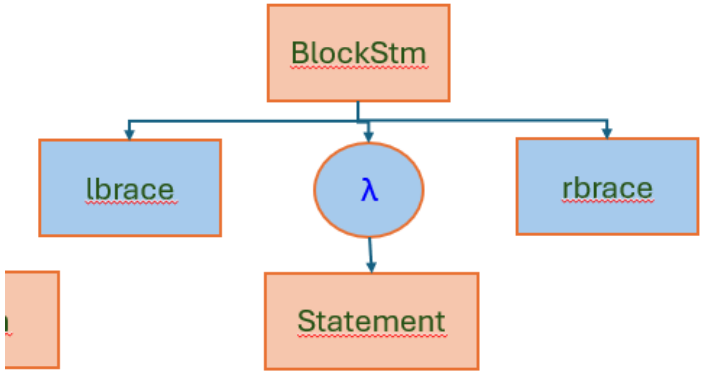


1.5 TintoParser



1.6 Ejemplo TintoParser lambda

```
794    private void parseBlockStm() throws SintaxException
795    {
796        int[] expected = { LBRACE };
797        switch(nextToken.getKind())
798        {
799            case LBRACE:
800                match(LBRACE);
801                parseStatementList();
802                match(RBRACE);
803                break;
804            default:
805                throw new SintaxException(nextToken,expe
806        }
807    }
```




```
478    private void parseStatementList() throws SintaxExcep
479    {
480        int[] expected = { INT, CHAR, BOOLEAN, IDENTIFIE
481                           RETURN, SEMICOLON, LBRACE, RB
482        switch(nextToken.getKind())
483        {
484            case INT:
485            case CHAR:
486            case BOOLEAN:
487            case IDENTIFIER:
488            case IF:
489            case WHILE:
490            case RETURN:
491            case SEMICOLON:
492            case LBRACE:
493                parseStatement();
494                parseStatementList();
495                break;
496            case RBRACE:
497                break;
498            default:
499                throw new SintaxException(nextToken,expe
500        }
```

```
public class SintaxException extends Exception implements TokenConstants {

  /**
   * Mensaje de error
   */
  private String msg;

  /**
   * Constructor con un solo tipo esperado
   * @param token
   * @param expected
   */
  public SintaxException(Token token, int expected)
  {
    this.msg = "Sintax exception at row "+token.getRow();
    msg += ", column "+token.getColumn()+".\n";
    msg += "  Found "+token.getLexeme()+"\n";
    msg += "  while expecting "+getLexemeForKind(expected)+".\n";
  }

  /**
   * Constructor con una lista de tipos esperados
   * @param token
   * @param expected
   */
  public SintaxException(Token token, int[] expected)
  {
    this.msg = "Sintax exception at row "+token.getRow();
    msg += ", column "+token.getColumn()+".\n";
    msg += "  Found "+token.getLexeme()+"\n";
    msg += "  while expecting one of\n";
    for(int i=0; i<expected.length; i++)
    {
      msg += "    "+getLexemeForKind(expected[i])+"\n";
    }
  }

  ...
}
```

1.7 SintaxException