Лабораторная работа № 2.4 «Рекурсивный спуск»

6 мая 2024 г.

Сергей Виленский, ИУ9-62Б

Цель работы

Целью данной работы является изучение алгоритмов построения парсеров методом рекурсивного спуска.

Индивидуальный вариант

Объявления типов и констант в Паскале:

В `record`’е точка с запятой разделяет поля и после `case` дополнительный end не ставится. См. https://bernd-oppolzer.de/PascalReport.pdf, третья с конца страница.

```
Type
  Coords = Record x, y: INTEGER end;
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

(* графический и текстовый дисплеи *)
const
  Heigh = 480;
  Width = 640;
  Lines = 24
  Columns = 80;
type
  BaseColor = (red, green, blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
```

```
      SymColor : Color;
      BackColor : Color
    end;

{ определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL
      )
  end;

  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end;
```

Ключевые слова и идентификаторы не чувствительны к регистру.

Реализация

Лексическая структура

Лексическая структура в порядке убывания приоритетов доменов

```
TypeIdentifier      -> INTEGER
                     | BOOLEAN
                     | REAL
                     | CHAR
                     | TEXT
ConstantIdentifier  -> Identifier
```

```
Identifier          -> Letter (Letter|Digit)*
Letter              -> [a-zA-Z]
UnsignedInteger     -> Digit+
UnsignedNumber           ->  UnsignedInteger  (.  Digit+)?  (E  [+-
]? UnsignedInteger)?
Digit               -> [0-9]
```

Грамматика языка

```
# program
Program -> Block

# block
Block -> KW_CONST BlockConstSequence Block
       | KW_TYPE BlockTypeSequence Block
       | ε
BlockConstSequence -> BlockConst
                    | BlockConst BlockConstSequence
BlockConst -> IDENTIFIER '=' Constant ';'
BlockTypeSequence -> BlockType
                   | BlockType BlockTypeSequence
BlockType -> IDENTIFIER '=' Type ';'

# field list
FieldList -> IdentifierWithTypeList
           | IdentifierWithTypeList ';' CaseBlock
IdentifierWithTypeList -> IdentifierWithType
                    | IdentifierWithType ';' IdentifierWithTypeList
IdentifierWithType -> IdentifierList ':' Type
CaseBlock -> KW_CASE IDENTIFIER ':' TypeIdentifier KW_OF CaseVariantSequence
CaseVariantSequence -> CaseVariant
                     | CaseVariant ';' CaseVariantSequence
CaseVariant -> ConstantList ':' '(' FieldList ')'
ConstantList -> Constant
              | Constant ',' ConstantList

# type
Type -> SimpleType
      | '^' TypeIdentifier
      | KW_PACKED TypeAfterPacked
      | TypeAfterPacked

TypeAfterPacked -> KW_ARRAY SimpleTypeList KW_OF Type
                 | KW_FILE KW_OF Type
                 | KW_SET KW_OF SimpleType
                 | KW_RECORD FieldList KW_END
```

```
SimpleTypeList -> SimpleType
                | SimpleType ',' SimpleTypeList

# simple type
SimpleType -> TypeIdentifier
            | '(' IdentifierList ')'
            | Constant '.' '.' Constant
TypeIdentifier -> CommonTypeIdentifier
                | IDENTIFIER
CommonTypeIdentifier -> KW_INTEGER
                      | KW_BOOLEAN
                      | KW_REAL
                      | KW_CHAR
                      | KW_TEXT
IdentifierList -> IDENTIFIER
                | IDENTIFIER ',' IdentifierList

# constant
Constant -> UnarSign ConstantIdentifier
          | ConstantIdentifier
          | UnarSign UNSIGNED_NUMBER
          | UNSIGNED_NUMBER
          | CHAR_SEQUENCE
UnarSign -> '+'
          | '-'
ConstantIdentifier -> IDENTIFIER
```

Программная реализация

Parser/Block/Block.ref

```
*$FROM Parser/Block/BlockConstSequence
*$FROM Parser/Block/BlockTypeSequence
$EXTERN BlockConstSequence, BlockTypeSequence;

$ENTRY Block {
    /* Block -> KW_CONST BlockConstSequence Block */
    () (KW_CONST e.C) e.Tokens t.Errs
        = <Block ((KW_CONST e.C)) e.Tokens t.Errs>;

    ((KW_CONST e.C)) (BlockConstSequence e.B) e.Tokens t.Errs
     = <Block ((KW_CONST e.C) (BlockConstSequence e.B)) e.Tokens t.Errs>;
    ((KW_CONST e.C)) e.Tokens t.Errs
     = <Block ((KW_CONST e.C)) <BlockConstSequence () e.Tokens t.Errs>>;

    ((KW_CONST e.C) t.BS) (Block e.B) e.Tokens t.Errs
```

```
       = <Block ((KW_CONST e.C) t.BS (Block e.B)) e.Tokens t.Errs>;
    ((KW_CONST e.C) t.BS) e.Tokens t.Errs
       = <Block ((KW_CONST e.C) t.BS) <Block () e.Tokens t.Errs>>;

    ((KW_CONST e.C) t.BS t.B) e.Tokens t.Errs
     = (Block (KW_CONST BlockConstSequence Block) (KW_CONST e.C) t.BS t.B) e.Tokens t.Errs;

    /* Block -> KW_TYPE BlockTypeSequence Block */
    () (KW_TYPE e.T) e.Tokens t.Errs
        = <Block ((KW_TYPE e.T)) e.Tokens t.Errs>;

    ((KW_TYPE e.T)) (BlockTypeSequence e.B) e.Tokens t.Errs
     = <Block ((KW_TYPE e.T) (BlockTypeSequence e.B)) e.Tokens t.Errs>;
    ((KW_TYPE e.T)) e.Tokens t.Errs
     = <Block ((KW_TYPE e.T)) <BlockTypeSequence () e.Tokens t.Errs>>;

    ((KW_TYPE e.T) t.BS) (Block e.B) e.Tokens t.Errs
       = <Block ((KW_TYPE e.T) t.BS (Block e.B)) e.Tokens t.Errs>;
    ((KW_TYPE e.T) t.BS) e.Tokens t.Errs
        = <Block ((KW_TYPE e.T) t.BS) <Block () e.Tokens t.Errs>>;

    ((KW_TYPE e.T) t.BS t.B) e.Tokens t.Errs
     = (Block (KW_TYPE BlockTypeSequence Block) (KW_TYPE e.T) t.BS t.B) e.Tokens t.Errs;

    /* Block -> ε */
    () e.Tokens t.Errs
        = (Block ()) e.Tokens t.Errs;
}
```

Parser/Block/BlockConst.ref

```
*$FROM Parser/Constant/Constant
$EXTERN Constant;

$ENTRY BlockConst {
    /* BlockConst -> IDENTIFIER '=' Constant ';' */
    () (IDENTIFIER e.I) e.Tokens t.Errs
        = <BlockConst ((IDENTIFIER e.I)) e.Tokens t.Errs>;
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <BlockConst ()
            e.Tokens (e.Errs (t.Start BlockConst '.'
                Unexpected t.Type '.'
                Expected '[' IDENTIFIER '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (BlockConst ())
```

```
        '$'
        (e.Errs ((EOF) BlockConst '.'
            Unexpected End Of File '.'
            Expected '[' IDENTIFIER '].'
            Terminated '.'));

(t.I) ('=' e.E) e.Tokens t.Errs
    = <BlockConst (t.I ('=' e.E)) e.Tokens t.Errs>;
(t.I) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <BlockConst (t.I)
       ('=' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start BlockConst '.'
            Unexpected t.Type '.'
            Expected '[' '=' '].'
            Inserted '.'))>;
(t.I) '$' (e.Errs)
    = <BlockConst (t.I)
        ('=' ((0 0) (0 0))) '$'
        (e.Errs ((EOF) BlockConst '.'
            Unexpected End Of File '.'
            Expected '[' '=' '].'
            Inserted '.'))>;

(t.I t.E) (Constant e.C) e.Tokens t.Errs
    = <BlockConst (t.I t.E (Constant e.C)) e.Tokens t.Errs>;
(t.I t.E) e.Tokens t.Errs
    = <BlockConst (t.I t.E) <Constant () e.Tokens t.Errs>>;

(t.I t.E t.C) (';' e.S) e.Tokens t.Errs
    = <BlockConst (t.I t.E t.C (';' e.S)) e.Tokens t.Errs>;
(t.I t.E t.C) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <BlockConst (t.I t.E t.C)
       (';' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start BlockConst '.'
            Unexpected t.Type '.'
            Expected '[' ';' '].'
            Inserted '.'))>;
(t.I t.E t.C) '$' (e.Errs)
    = <BlockConst (t.I t.E t.C)
        (';' ((0 0) (0 0))) '$'
        (e.Errs ((EOF) BlockConst '.'
            Unexpected End Of File '.'
            Expected '[' ';' '].'
            Inserted '.'))>;

(t.I t.E t.C t.S) e.Tokens t.Errs
```

```
     = (BlockConst (IDENTIFIER '=' Constant ';') t.I t.E t.C t.S) e.Tokens t.Errs;
}
```

Parser/Block/BlockConstSequence.ref

```
*$FROM Parser/Block/BlockConst
$EXTERN BlockConst;

$ENTRY BlockConstSequence {
    /* BlockConstSequence -> BlockConst BlockConstSequence */
    () (BlockConst e.B) e.Tokens t.Errs
        = <BlockConstSequence ((BlockConst e.B)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <BlockConstSequence () <BlockConst () e.Tokens t.Errs>>;

    (t.B) (BlockConstSequence e.BS) e.Tokens t.Errs
     = <BlockConstSequence (t.B (BlockConstSequence e.BS)) e.Tokens t.Errs>;
    (t.B) (IDENTIFIER e.I) e.Tokens t.Errs
     = <BlockConstSequence (t.B) <BlockConstSequence () (IDENTIFIER e.I) e.Tokens t.Errs>>;

    (t.B t.BS) e.Tokens t.Errs
     = (BlockConstSequence (BlockConst BlockConstSequence) t.B t.BS) e.Tokens t.Errs;

    /* BlockConstSequence -> BlockConst */
    (t.B) e.Tokens t.Errs
        = (BlockConstSequence (BlockConst) t.B) e.Tokens t.Errs;
}
```

Parser/Block/BlockType.ref

```
*$FROM Parser/Type/Type
$EXTERN Type_;

$ENTRY BlockType {
    /* BlockType -> IDENTIFIER '=' Type ';' */
    () (IDENTIFIER e.I) e.Tokens t.Errs
        = <BlockType ((IDENTIFIER e.I)) e.Tokens t.Errs>;
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <BlockType ()
            e.Tokens (e.Errs (t.Start BlockType '.'
                Unexpected t.Type '.'
                Expected '[' IDENTIFIER '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (BlockType (EOF))
            '$'
```

```
        (e.Errs ((EOF) BlockType '.'
            Unexpected End Of File '.'
            Expected '[' IDENTIFIER '].'
            Terminated '.'));

(t.I) ('=' e.E) e.Tokens t.Errs
    = <BlockType (t.I ('=' e.E)) e.Tokens t.Errs>;
(t.I) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <BlockType (t.I)
       ('=' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start BlockType '.'
            Unexpected t.Type '.'
            Expected '[' '=' '].'
            Inserted '.'))>;
(t.I) '$' (e.Errs)
    = <BlockType (t.I)
        ('=' ((0 0) (0 0))) '$'
        (e.Errs ((EOF) BlockType '.'
            Unexpected End Of File '.'
            Expected '[' '=' '].'
            Inserted '.'))>;

(t.I t.E) (Type e.C) e.Tokens t.Errs
    = <BlockType (t.I t.E (Type e.C)) e.Tokens t.Errs>;
(t.I t.E) e.Tokens t.Errs
    = <BlockType (t.I t.E) <Type_ () e.Tokens t.Errs>>;

(t.I t.E t.C) (';' e.S) e.Tokens t.Errs
    = <BlockType (t.I t.E t.C (';' e.S)) e.Tokens t.Errs>;
(t.I t.E t.C) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <BlockType (t.I t.E t.C)
       (';' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start BlockType '.'
            Unexpected t.Type '.'
            Expected '[' ';' '].'
            Inserted '.'))>;
(t.I t.E t.C) '$' (e.Errs)
    = <BlockType (t.I t.E t.C)
        (';' ((0 0) (0 0))) '$'
        (e.Errs ((EOF) BlockType '.'
            Unexpected End Of File '.'
            Expected '[' ';' '].'
            Inserted '.'))>;

(t.I t.E t.C t.S) e.Tokens t.Errs
 = (BlockType (IDENTIFIER '=' Type ';') t.I t.E t.C t.S) e.Tokens t.Errs;
```

```
}
```

Parser/Block/BlockTypeSequence.ref

```
*$FROM Parser/Block/BlockType
$EXTERN BlockType;

$ENTRY BlockTypeSequence {
    /* BlockTypeSequence -> BlockType BlockTypeSequence */
    () (BlockType e.B) e.Tokens t.Errs
        = <BlockTypeSequence ((BlockType e.B)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <BlockTypeSequence () <BlockType () e.Tokens t.Errs>>;

    (t.B) (BlockTypeSequence e.BS) e.Tokens t.Errs
     = <BlockTypeSequence (t.B (BlockTypeSequence e.BS)) e.Tokens t.Errs>;
    (t.B) (IDENTIFIER e.I) e.Tokens t.Errs
     = <BlockTypeSequence (t.B) <BlockTypeSequence () (IDENTIFIER e.I) e.Tokens t.Errs>>;

    (t.B t.BS) e.Tokens t.Errs
     = (BlockTypeSequence (BlockType BlockTypeSequence) t.B t.BS) e.Tokens t.Errs;

    /* BlockTypeSequence -> BlockType */
    (t.B) e.Tokens t.Errs
        = (BlockTypeSequence (BlockType) t.B) e.Tokens t.Errs;
}
```

Parser/Constant/Constant.ref

```
*$FROM Parser/Constant/UnarSign
*$FROM Parser/Constant/ConstantIdentifier
$EXTERN UnarSign, ConstantIdentifier;

$ENTRY Constant {
    /* Constant -> UnarSign ConstantIdentifier */
    () (UnarSign e.U) e.Tokens t.Errs
        = <Constant ((UnarSign e.U)) e.Tokens t.Errs>;
    () ('+' e.P) e.Tokens t.Errs
        = <Constant () <UnarSign () ('+' e.P) e.Tokens t.Errs>>;
    () ('-' e.M) e.Tokens t.Errs
        = <Constant () <UnarSign () ('-' e.M) e.Tokens t.Errs>>;

    ((UnarSign e.U)) (ConstantIdentifier e.C) e.Tokens t.Errs
     = <Constant ((UnarSign e.U) (ConstantIdentifier e.C)) e.Tokens t.Errs>;
    ((UnarSign e.U)) (IDENTIFIER e.I) e.Tokens t.Errs
     = <Constant ((UnarSign e.U)) <ConstantIdentifier () (IDENTIFIER e.I) e.Tokens t.Errs>>;
```

```
 ((UnarSign e.U) (ConstantIdentifier e.C)) e.Tokens t.Errs
  = (Constant (UnarSign ConstantIdentifier) (UnarSign e.U) (ConstantIdentifier e.C)) e.Toker

 /* Constant -> ConstantIdentifier */
 () (ConstantIdentifier e.C) e.Tokens t.Errs
     = <Constant ((ConstantIdentifier e.C)) e.Tokens t.Errs>;
 () (IDENTIFIER e.I) e.Tokens t.Errs
  = <Constant () <ConstantIdentifier () (IDENTIFIER e.I) e.Tokens t.Errs>>;

 ((ConstantIdentifier e.C)) e.Tokens t.Errs
  = (Constant (ConstantIdentifier) (ConstantIdentifier e.C)) e.Tokens t.Errs;

 /* Constant -> UnarSign UNSIGNED_NUMBER */
 ((UnarSign e.U)) (UNSIGNED_NUMBER e.UN) e.Tokens t.Errs
  = <Constant ((UnarSign e.U) (UNSIGNED_NUMBER e.UN)) e.Tokens t.Errs>;
((UnarSign e.U)) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <Constant ((UnarSign e.U))
         e.Tokens (e.Errs (t.Start Constant '.'
             Unexpected t.Type '.'
         Expected '[' ConstantIdentifier or UNSIGNED_NUMBER '].'
             Skipped '.'))>;
 ((UnarSign e.U)) '$' (e.Errs)
     = (Constant (EOF))
         '$'
         (e.Errs ((EOF) Constant '.'
             Unexpected End Of File '.'
         Expected '[' ConstantIdentifier or UNSIGNED_NUMBER '].'
             Terminated '.'));

 ((UnarSign e.U) (UNSIGNED_NUMBER e.UN)) e.Tokens t.Errs
  = (Constant (UnarSign UNSIGNED_NUMBER) (UnarSign e.U) (UNSIGNED_NUMBER e.UN)) e.Tokens t.E

 /* Constant -> UNSIGNED_NUMBER */
 () (UNSIGNED_NUMBER e.UN) e.Tokens t.Errs
     = <Constant ((UNSIGNED_NUMBER e.UN)) e.Tokens t.Errs>;

 ((UNSIGNED_NUMBER e.UN)) e.Tokens t.Errs
  = (Constant (UNSIGNED_NUMBER) (UNSIGNED_NUMBER e.UN)) e.Tokens t.Errs;

 /* Constant -> CHAR_SEQUENCE */
 () (CHAR_SEQUENCE e.CS) e.Tokens t.Errs
     = <Constant ((CHAR_SEQUENCE e.CS)) e.Tokens t.Errs>;

 ((CHAR_SEQUENCE e.CS)) e.Tokens t.Errs
  = (Constant (CHAR_SEQUENCE) (CHAR_SEQUENCE e.CS)) e.Tokens t.Errs;
```

```
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <Constant ()
            e.Tokens (e.Errs (t.Start Constant '.'
                Unexpected t.Type '.'
           Expected '[' UnarSign or ConstantIdentifier or UNSIGNED_NUMBER or '\'' '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (Constant (EOF))
            '$'
            (e.Errs ((EOF) Constant '.'
                Unexpected End Of File '.'
           Expected '[' UnarSign or ConstantIdentifier or UNSIGNED_NUMBER or '\'' '].'
                Terminated '.'));
}
```

Parser/Constant/ConstantIdentifier.ref

```
$ENTRY ConstantIdentifier {
    /* ConstantIdentifier -> IDENTIFIER */
    () (IDENTIFIER e.I) e.Tokens t.Errs
        = <ConstantIdentifier ((IDENTIFIER e.I)) e.Tokens t.Errs>;
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <ConstantIdentifier ()
            e.Tokens (e.Errs (t.Start ConstantIdentifier '.'
                Unexpected t.Type '.'
                Expected '[' IDENTIFIER '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (ConstantIdentifier (EOF))
            '$'
            (e.Errs ((EOF) ConstantIdentifier '.'
                Unexpected End Of File '.'
                Expected '[' IDENTIFIER '].'
                Terminated '.'));

    (t.I) e.Tokens t.Errs
        = (ConstantIdentifier (IDENTIFIER) t.I) e.Tokens t.Errs;
}
```

Parser/Constant/UnarSign.ref

```
$ENTRY UnarSign {
    /* UnarSign -> '+' */
    () ('+' e.P) e.Tokens t.Errs
```

```
        = <UnarSign (('+' e.P)) e.Tokens t.Errs>;

    (('+' e.P)) e.Tokens t.Errs
        = (UnarSign ('+') ('+' e.P)) e.Tokens t.Errs;

    /* UnarSign -> '-' */
    () ('-' e.M) e.Tokens t.Errs
        = <UnarSign (('-' e.M)) e.Tokens t.Errs>;

    (('-' e.M)) e.Tokens t.Errs
        = (UnarSign ('-') ('-' e.M)) e.Tokens t.Errs;

    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <UnarSign ()
            e.Tokens (e.Errs (t.Start UnarSign '.'
                Unexpected t.Type '.'
                Expected '[' '+' or '-' '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (UnarSign (EOF))
            '$'
            (e.Errs ((EOF) UnarSign '.'
                Unexpected End Of File '.'
                Expected '[' '+' or '-' '].'
                Terminated '.'));
}
```

Parser/FieldList/CaseBlock.ref

```
*$FROM Parser/SimpleType/TypeIdentifier
*$FROM Parser/FieldList/CaseVariantSequence
$EXTERN TypeIdentifier, CaseVariantSequence;

$ENTRY CaseBlock {
   /* CaseBlock -> KW_CASE IDENTIFIER ':' TypeIdentifier KW_OF CaseVariantSequence */
    () (KW_CASE e.C) e.Tokens t.Errs
        = <CaseBlock ((KW_CASE e.C)) e.Tokens t.Errs>;
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <CaseBlock ()
        (KW_CASE (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start CaseBlock '.'
                Unexpected t.Type '.'
                Expected '[' KW_CASE '].'
                Inserted '.'))>;
    () '$' (e.Errs)
        = <CaseBlock ()
```

```
        (KW_CASE ((0 0) (0 0))) '$'
        (e.Errs ((EOF) CaseBlock '.'
            Unexpected End Of File '.'
            Expected '[' KW_CASE '].'
            Inserted '.'))>;

(t.C) (IDENTIFIER e.I) e.Tokens t.Errs
    = <CaseBlock (t.C (IDENTIFIER e.I)) e.Tokens t.Errs>;
(t.C) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <CaseBlock (t.C)
        e.Tokens (e.Errs (t.Start CaseBlock '.'
            Unexpected t.Type '.'
            Expected '[' IDENTIFIER '].'
            Skipped '.'))>;
(t.C) '$' (e.Errs)
    = (CaseBlock (EOF))
        '$'
        (e.Errs ((EOF) CaseBlock '.'
            Unexpected End Of File '.'
            Expected '[' IDENTIFIER '].'
            Terminated '.'));

(t.C t.I) (':' e.D) e.Tokens t.Errs
    = <CaseBlock (t.C t.I (':' e.D)) e.Tokens t.Errs>;
(t.C t.I) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <CaseBlock (t.C t.I)
       (':' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start CaseBlock '.'
            Unexpected t.Type '.'
            Expected '[' ':' '].'
            Inserted '.'))>;
(t.C t.I) '$' (e.Errs)
    = <CaseBlock (t.C t.I)
        (':' ((0 0) (0 0))) '$'
        (e.Errs ((EOF) CaseBlock '.'
            Unexpected End Of File '.'
            Expected '[' ':' '].'
            Inserted '.'))>;

(t.C t.I t.D) (TypeIdentifier e.T) e.Tokens t.Errs
 = <CaseBlock (t.C t.I t.D (TypeIdentifier e.T)) e.Tokens t.Errs>;
(t.C t.I t.D) e.Tokens t.Errs
 = <CaseBlock (t.C t.I t.D) <TypeIdentifier () e.Tokens t.Errs>>;

(t.C t.I t.D t.T) (KW_OF e.O) e.Tokens t.Errs
   = <CaseBlock (t.C t.I t.D t.T (KW_OF e.O)) e.Tokens t.Errs>;
```

```
   (t.C t.I t.D t.T) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <CaseBlock (t.C t.I t.D t.T)
          (KW_OF (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start CaseBlock '.'
                Unexpected t.Type '.'
                Expected '[' KW_OF '].'
                Inserted '.'))>;
    (t.C t.I t.D t.T) '$' (e.Errs)
        = <CaseBlock (t.C t.I t.D t.T)
            (KW_OF ((0 0) (0 0))) '$'
            (e.Errs ((EOF) CaseBlock '.'
                Unexpected End Of FIle '.'
                Expected '[' KW_OF '].'
                Inserted '.'))>;

   (t.C t.I t.D t.T t.O) (CaseVariantSequence e.CV) e.Tokens t.Errs
     = <CaseBlock (t.C t.I t.D t.T t.O (CaseVariantSequence e.CV)) e.Tokens t.Errs>;
    (t.C t.I t.D t.T t.O) e.Tokens t.Errs
     = <CaseBlock (t.C t.I t.D t.T t.O) <CaseVariantSequence () e.Tokens t.Errs>>;

    (t.C t.I t.D t.T t.O t.CV) e.Tokens t.Errs
     = (CaseBlock (KW_CASE IDENTIFIER ':' TypeIdentifier KW_OF CaseVariantSequence)
           t.C t.I t.D t.T t.O t.CV) e.Tokens t.Errs;
}
```

Parser/FieldList/CaseVariant.ref

```
*$FROM Parser/FieldList/ConstantList
*$FROM Parser/FieldList/FieldList
$EXTERN ConstantList, FieldList;

$ENTRY CaseVariant {
    /* CaseVariant -> ConstantList ':' '(' FieldList ')' */
    () (ConstantList e.C) e.Tokens t.Errs
        = <CaseVariant ((ConstantList e.C)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <CaseVariant () <ConstantList () e.Tokens t.Errs>>;

    (t.C) (':' e.D) e.Tokens t.Errs
        = <CaseVariant (t.C (':' e.D)) e.Tokens t.Errs>;
    (t.C) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <CaseVariant (t.C)
           (':' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start CaseVariant '.'
                Unexpected t.Type '.'
                Expected '[' ':' '].'
```

```
            Inserted '.'))>;
 (t.C) '$' (e.Errs)
     = <CaseVariant (t.C)
         (':' ((0 0) (0 0))) '$'
         (e.Errs ((EOF) CaseVariant '.'
             Unexpected End Of File '.'
             Expected '[' ':' '].'
             Inserted '.'))>;

 (t.C t.D) ('(' e.B) e.Tokens t.Errs
     = <CaseVariant (t.C t.D ('(' e.B)) e.Tokens t.Errs>;
 (t.C t.D) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <CaseVariant (t.C t.D)
        ('(' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
         e.Tokens (e.Errs (t.Start CaseVariant '.'
             Unexpected t.Type '.'
             Expected '[' '\(' '].'
             Inserted '.'))>;
 (t.C t.D) '$' (e.Errs)
     = <CaseVariant (t.C t.D)
         ('(' ((0 0) (0 0))) '$'
         (e.Errs ((EOF) CaseVariant '.'
             Unexpected End Of File '.'
             Expected '[' '\(' '].'
             Inserted '.'))>;

 (t.C t.D t.B) (FieldList e.F) e.Tokens t.Errs
   = <CaseVariant (t.C t.D t.B (FieldList e.F)) e.Tokens t.Errs>;
 (t.C t.D t.B) e.Tokens t.Errs
    = <CaseVariant (t.C t.D t.B) <FieldList () e.Tokens t.Errs>>;

 (t.C t.D t.B t.F) (')' e.BC) e.Tokens t.Errs
    = <CaseVariant (t.C t.D t.B t.F (')' e.BC)) e.Tokens t.Errs>;
(t.C t.D t.B t.F) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <CaseVariant (t.C t.D t.B t.F)
        (')' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
         e.Tokens (e.Errs (t.Start CaseVariant '.'
             Unexpected t.Type '.'
             Expected '[' '\)' '].'
             Inserted '.'))>;
 (t.C t.D t.B t.F) '$' (e.Errs)
     = <CaseVariant (t.C t.D t.B t.F)
         (')' ((0 0) (0 0))) '$'
         (e.Errs ((EOF) CaseVariant '.'
             Unexpected End Of File '.'
             Expected '[' '\)' '].'
```

```
                Inserted '.'))>;

    (t.C t.D t.B t.F t.BC) e.Tokens t.Errs
     = (CaseVariant (ConstantList ':' '(' FieldList ')') t.C t.D t.B t.F t.BC) e.Tokens t.Errs;
}
```

Parser/FieldList/CaseVariantSequence.ref

```
*$FROM Parser/FieldList/CaseVariant
$EXTERN CaseVariant;

$ENTRY CaseVariantSequence {
   /* CaseVariantSequence -> CaseVariant ';' CaseVariantSequence */
    () (CaseVariant e.C) e.Tokens t.Errs
       = <CaseVariantSequence ((CaseVariant e.C)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
       = <CaseVariantSequence () <CaseVariant () e.Tokens t.Errs>>;

    (t.C) (';' e.D) e.Tokens t.Errs
        = <CaseVariantSequence (t.C (';' e.D)) e.Tokens t.Errs>;

    (t.C t.D) (CaseVariantSequence e.CS) e.Tokens t.Errs
     = <CaseVariantSequence (t.C t.D (CaseVariantSequence e.CS)) e.Tokens t.Errs>;
    (t.C t.D) e.Tokens t.Errs
     = <CaseVariantSequence (t.C t.D) <CaseVariantSequence () e.Tokens t.Errs>>;

    (t.C t.D t.CS) e.Tokens t.Errs
     = (CaseVariantSequence (CaseVariant ';' CaseVariantSequence) t.C t.D t.CS) e.Tokens t.Errs

    /* CaseVariantSequence -> CaseVariant */
    (t.C) e.Tokens t.Errs
        = (CaseVariantSequence (CaseVariant) t.C) e.Tokens t.Errs;
}
```

Parser/FieldList/ConstantList.ref

```
*$FROM Parser/Constant/Constant
$EXTERN Constant;

$ENTRY ConstantList {
    /* ConstantList -> Constant ',' ConstantList */
    () (Constant e.C) e.Tokens t.Errs
        = <ConstantList ((Constant e.C)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <ConstantList () <Constant () e.Tokens t.Errs>>;
```

```
    (t.C) (',' e.CM) e.Tokens t.Errs
        = <ConstantList (t.C (',' e.CM)) e.Tokens t.Errs>;

    (t.C t.CM) (ConstantList e.CL) e.Tokens t.Errs
     = <ConstantList (t.C t.CM (ConstantList e.CL)) e.Tokens t.Errs>;
    (t.C t.CM) e.Tokens t.Errs
      = <ConstantList (t.C t.CM) <ConstantList () e.Tokens t.Errs>>;

    (t.C t.CM t.CL) e.Tokens t.Errs
     = (ConstantList (Constant ',' ConstantList) t.C t.CM t.CL) e.Tokens t.Errs;

    /* ConstantList -> Constant */
    (t.C) e.Tokens t.Errs
        = (ConstantList (Constant) t.C) e.Tokens t.Errs;
}
```

Parser/FieldList/FieldList.ref

```
*$FROM Parser/FieldList/IdentifierWithTypeList
*$FROM Parser/FieldList/CaseBlock
$EXTERN IdentifierWithTypeList, CaseBlock;

$ENTRY FieldList {
    /* FieldList -> IdentifierWithTypeList ';' CaseBlock */
    () (IdentifierWithTypeList e.I) e.Tokens t.Errs
      = <FieldList ((IdentifierWithTypeList e.I)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
      = <FieldList () <IdentifierWithTypeList () e.Tokens t.Errs>>;

    (t.I) (';' e.C) e.Tokens t.Errs
        = <FieldList (t.I (';' e.C)) e.Tokens t.Errs>;

    (t.I t.C) (CaseBlock e.CB) e.Tokens t.Errs
        = <FieldList (t.I t.C (CaseBlock e.CB)) e.Tokens t.Errs>;
    (t.I t.C) (KW_CASE e.CB) e.Tokens t.Errs
     = <FieldList (t.I t.C) <CaseBlock () (KW_CASE e.CB) e.Tokens t.Errs>>;

    (t.I t.C t.CB) e.Tokens t.Errs
     = (FieldList (IdentifierWithTypeList ';' CaseBlock) t.I t.C t.CB)  e.Tokens t.Errs;

    /* FieldList -> IdentifierWithTypeList */
    (t.I) e.Tokens t.Errs
       = (FieldList (IdentifierWithTypeList) t.I) e.Tokens t.Errs;
}
```

Parser/FieldList/IdentifierWithType.ref

```
*$FROM Parser/SimpleType/IdentifierList
*$FROM Parser/Type/Type
$EXTERN IdentifierList, Type_;

$ENTRY IdentifierWithType {
    /* IdentifierWithType -> IdentifierList ':' Type */
    () (IdentifierList e.I) e.Tokens t.Errs
      = <IdentifierWithType ((IdentifierList e.I)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
      = <IdentifierWithType () <IdentifierList () e.Tokens t.Errs>>;

    (t.I) (':' e.D) e.Tokens t.Errs
        = <IdentifierWithType (t.I (':' e.D)) e.Tokens t.Errs>;
    (t.I) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <IdentifierWithType (t.I)
           (':' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start IdentifierWithType '.'
                Unexpected t.Type '.'
                Expected '[' ':' '].'
                Inserted '.'))>;
    (t.I) '$' (e.Errs)
        = <IdentifierWithType (t.I)
            (':' ((0 0) (0 0))) '$'
            (e.Errs ((EOF) IdentifierWithType '.'
                Unexpected End Of File '.'
                Expected '[' ':' '].'
                Inserted '.'))>;

    (t.I t.D) (Type e.T) e.Tokens t.Errs
       = <IdentifierWithType (t.I t.D (Type e.T)) e.Tokens t.Errs>;
    (t.I t.D) e.Tokens t.Errs
       = <IdentifierWithType (t.I t.D) <Type_ () e.Tokens t.Errs>>;

    (t.I t.D t.T) e.Tokens t.Errs
     = (IdentifierWithType (IdentifierList ':' Type) t.I t.D t.T) e.Tokens t.Errs;
}
```

Parser/FieldList/IdentifierWithTypeList.ref

```
*$FROM Parser/FieldList/IdentifierWithType
$EXTERN IdentifierWithType;

$ENTRY IdentifierWithTypeList {
   /* IdentifierWithTypeList -> IdentifierWithType ';' IdentifierWithTypeList */
    () (IdentifierWithType e.I) e.Tokens t.Errs
     = <IdentifierWithTypeList ((IdentifierWithType e.I)) e.Tokens t.Errs>;
```

```
    () e.Tokens t.Errs
     = <IdentifierWithTypeList () <IdentifierWithType () e.Tokens t.Errs>>;

    (t.I) (';' e.C) e.Tokens t.Errs
       = <IdentifierWithTypeList (t.I (';' e.C)) e.Tokens t.Errs>;

    (t.I t.C) (IdentifierWithTypeList e.IL) e.Tokens t.Errs
     = <IdentifierWithTypeList (t.I t.C (IdentifierWithTypeList e.IL)) e.Tokens t.Errs>;
    (t.I t.C) (IDENTIFIER e.ID) e.Tokens t.Errs
       = <IdentifierWithTypeList (t.I t.C)
       <IdentifierWithTypeList () (IDENTIFIER e.ID) e.Tokens t.Errs>>;
    (t.I t.C) e.Tokens t.Errs
     = (IdentifierWithTypeList (IdentifierWithType) t.I) t.C e.Tokens t.Errs;

    (t.I t.C t.IL) e.Tokens t.Errs
     = (IdentifierWithTypeList (IdentifierWithType ';' IdentifierWithTypeList) t.I t.C t.IL)
           e.Tokens t.Errs;

    /* IdentifierWithTypeList -> IdentifierWithType */
    (t.I) e.Tokens t.Errs
     = (IdentifierWithTypeList (IdentifierWithType) t.I) e.Tokens t.Errs;
}
```

Parser/Program/Program.ref

```
*$FROM Parser/Block/Block
$EXTERN Block;

$ENTRY Program {
    /* Program -> Block */
    () (Block e.B) e.Tokens t.Errs
       = <Program ((Block e.B)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
       = <Program () <Block () e.Tokens t.Errs>>;

    (t.B) e.Tokens t.Errs
       = (Program (Block) t.B) e.Tokens t.Errs;
}
```

Parser/SimpleType/CommonTypeIdentifier.ref

```
$ENTRY CommonTypeIdentifier {
    /* CommonTypeIdentifier -> KW_INTEGER */
    () (KW_INTEGER e.I) e.Tokens t.Errs
      = <CommonTypeIdentifier ((KW_INTEGER e.I)) e.Tokens t.Errs>;
```

```
    ((KW_INTEGER e.I)) e.Tokens t.Errs
     = (CommonTypeIdentifier (KW_INTEGER) (KW_INTEGER e.I)) e.Tokens t.Errs;

    /* CommonTypeIdentifier -> KW_BOOLEAN */
    () (KW_BOOLEAN e.B) e.Tokens t.Errs
       = <CommonTypeIdentifier ((KW_BOOLEAN e.B)) e.Tokens t.Errs>;

    ((KW_BOOLEAN e.B)) e.Tokens t.Errs
     = (CommonTypeIdentifier (KW_BOOLEAN) (KW_BOOLEAN e.B)) e.Tokens t.Errs;

    /* CommonTypeIdentifier -> KW_REAL */
    () (KW_REAL e.R) e.Tokens t.Errs
        = <CommonTypeIdentifier ((KW_REAL e.R)) e.Tokens t.Errs>;

    ((KW_REAL e.R)) e.Tokens t.Errs
     = (CommonTypeIdentifier (KW_REAL) (KW_REAL e.R)) e.Tokens t.Errs;

    /* CommonTypeIdentifier -> KW_CHAR */
    () (KW_CHAR e.C) e.Tokens t.Errs
        = <CommonTypeIdentifier ((KW_CHAR e.C)) e.Tokens t.Errs>;

    ((KW_CHAR e.C)) e.Tokens t.Errs
     = (CommonTypeIdentifier (KW_CHAR) (KW_CHAR e.C)) e.Tokens t.Errs;

    /* CommonTypeIdentifier -> KW_TEXT */
    () (KW_TEXT e.T) e.Tokens t.Errs
        = <CommonTypeIdentifier ((KW_TEXT e.T)) e.Tokens t.Errs>;

    ((KW_TEXT e.T)) e.Tokens t.Errs
     = (CommonTypeIdentifier (KW_TEXT) (KW_TEXT e.T)) e.Tokens t.Errs;

    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <CommonTypeIdentifier ()
            e.Tokens (e.Errs (t.Start CommonTypeIdentifier '.'
                Unexpected t.Type '.'
           Expected '[' KW_INTEGER or KW_BOOLEAN or KW_REAL or KW_CHAR or KW_TEXT '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (CommonTypeIdentifier (EOF))
            '$'
            (e.Errs ((EOF) CommonTypeIdentifier '.'
                Unexpected End Of File '.'
           Expected '[' KW_INTEGER or KW_BOOLEAN or KW_REAL or KW_CHAR or KW_TEXT '].'
                Terminated '.'));
}
```

Parser/SimpleType/IdentifierList.ref

```
$ENTRY IdentifierList {
    /* IdentifierList -> IDENTIFIER ',' IdentifierList */
    () (IDENTIFIER e.I) e.Tokens t.Errs
        = <IdentifierList ((IDENTIFIER e.I)) e.Tokens t.Errs>;
    () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <IdentifierList ()
            e.Tokens (e.Errs (t.Start IdentifierList '.'
                Unexpected t.Type '.'
                Expected '[' IDENTIFIER '].'
                Skipped '.'))>;
    () '$' (e.Errs)
        = (IdentifierList (EOF))
            '$'
            (e.Errs ((EOF) IdentifierList '.'
                Unexpected End Of File '.'
                Expected '[' IDENTIFIER '].'
                Terminated '.'));

    (t.I) (',' e.C) e.Tokens t.Errs
        = <IdentifierList (t.I (',' e.C)) e.Tokens t.Errs>;

    (t.I t.C) (IdentifierList e.IL) e.Tokens t.Errs
     = <IdentifierList (t.I t.C (IdentifierList e.IL)) e.Tokens t.Errs>;
    (t.I t.C) e.Tokens t.Errs
     = <IdentifierList (t.I t.C) <IdentifierList () e.Tokens t.Errs>>;

    (t.I t.C t.IL) e.Tokens t.Errs
     = (IdentifierList (IDENTIFIER ',' IdentifierList) t.I t.C t.IL) e.Tokens t.Errs;

    /* IdentifierList -> IDENTIFIER */
    (t.I) e.Tokens t.Errs
        = (IdentifierList (IDENTIFIER) t.I) e.Tokens t.Errs;
}
```

Parser/SimpleType/SimpleType.ref

```
*$FROM Parser/SimpleType/TypeIdentifier
*$FROM Parser/SimpleType/IdentifierList
*$FROM Parser/Constant/Constant
$EXTERN TypeIdentifier, IdentifierList, Constant;

$ENTRY SimpleType {
    /* SimpleType -> TypeIdentifier */
    () (TypeIdentifier e.T) e.Tokens t.Errs
```

```
    = <SimpleType ((TypeIdentifier e.T)) e.Tokens t.Errs>;
 () (KW_INTEGER e.I) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (KW_INTEGER e.I) e.Tokens t.Errs>>;
 () (KW_BOOLEAN e.B) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (KW_BOOLEAN e.B) e.Tokens t.Errs>>;
 () (KW_REAL e.R) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (KW_REAL e.R) e.Tokens t.Errs>>;
 () (KW_CHAR e.C) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (KW_CHAR e.C) e.Tokens t.Errs>>;
 () (KW_TEXT e.T) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (KW_TEXT e.T) e.Tokens t.Errs>>;
 () (IDENTIFIER e.I) e.Tokens t.Errs
  = <SimpleType () <TypeIdentifier () (IDENTIFIER e.I) e.Tokens t.Errs>>;

 ((TypeIdentifier e.T)) e.Tokens t.Errs
  = (SimpleType (TypeIdentifier) (TypeIdentifier e.T)) e.Tokens t.Errs;

 /* SimpleType -> '(' IdentifierList ')' */
 () ('(' e.B) e.Tokens t.Errs
    = <SimpleType (('(' e.B)) e.Tokens t.Errs>;

 (('(' e.B)) (IdentifierList e.I) e.Tokens t.Errs
  = <SimpleType (('(' e.B) (IdentifierList e.I)) e.Tokens t.Errs>;
 (('(' e.B)) e.Tokens t.Errs
  = <SimpleType (('(' e.B)) <IdentifierList () e.Tokens t.Errs>>;

 (('(' e.B) t.I) (')' e.BC) e.Tokens t.Errs
    = <SimpleType (('(' e.B) t.I (')' e.BC)) e.Tokens t.Errs>;

 (('(' e.B) t.I t.BC) e.Tokens t.Errs
  = (SimpleType ('(' IdentifierList ')') ('(' e.B) t.I t.BC) e.Tokens t.Errs;

 /* SimpleType -> Constant '.' '.' Constant */
 () (Constant e.C1) e.Tokens t.Errs
    = <SimpleType ((Constant e.C1)) e.Tokens t.Errs>;
 () e.Tokens t.Errs
    = <SimpleType () <Constant () e.Tokens t.Errs>>;

 ((Constant e.C1)) ('.' e.D1) e.Tokens t.Errs
   = <SimpleType ((Constant e.C1) ('.' e.D1)) e.Tokens t.Errs>;
((Constant e.C1)) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
    = <SimpleType ((Constant e.C1))
       ('.' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
        e.Tokens (e.Errs (t.Start SimpleType '.'
            Unexpected t.Type '.'
            Expected '[' '.' '].'
```

```
                Inserted '.'))>;
    ((Constant e.C1)) '$' (e.Errs)
        = <SimpleType ((Constant e.C1))
            ('.' ((0 0) (0 0))) '$'
            (e.Errs ((EOF) SimpleType '.'
                Unexpected End Of File '.'
                Expected '[' '.' '].'
                Inserted '.'))>;

    ((Constant e.C1) t.D1) ('.' e.D2) e.Tokens t.Errs
     = <SimpleType ((Constant e.C1) t.D1 ('.' e.D2)) e.Tokens t.Errs>;
   ((Constant e.C1) t.D1) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <SimpleType ((Constant e.C1) t.D1)
           ('.' (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start SimpleType '.'
                Unexpected t.Type '.'
                Expected '[' '.' '].'
                Inserted '.'))>;
    ((Constant e.C1) t.D1) '$' (e.Errs)
        = <SimpleType ((Constant e.C1) t.D1)
            ('.' ((0 0) (0 0))) '$'
            (e.Errs ((EOF) SimpleType '.'
                Unexpected End Of File '.'
                Expected '[' '.' '].'
                Inserted '.'))>;

    ((Constant e.C1) t.D1 t.D2) (Constant e.C2) e.Tokens t.Errs
     = <SimpleType ((Constant e.C1) t.D1 t.D2 (Constant e.C2)) e.Tokens t.Errs>;
    ((Constant e.C1) t.D1 t.D2) e.Tokens t.Errs
     = <SimpleType ((Constant e.C1) t.D1 t.D2) <Constant () e.Tokens t.Errs>>;

    ((Constant e.C1) t.D1 t.D2 t.C2) e.Tokens t.Errs
     = (SimpleType (Constant '.' '.' Constant) t.D1 t.D2 t.C2) e.Tokens t.Errs;
}
```

Parser/SimpleType/TypeIdentifier.ref

```
*$FROM Parser/SimpleType/CommonTypeIdentifier
$EXTERN CommonTypeIdentifier;

$ENTRY TypeIdentifier {
    /* TypeIdentifier -> IDENTIFIER */
    () (IDENTIFIER e.I) e.Tokens t.Errs
        = <TypeIdentifier ((IDENTIFIER e.I)) e.Tokens t.Errs>;

    ((IDENTIFIER e.I)) e.Tokens t.Errs
```

```
     = (TypeIdentifier (IDENTIFIER) (IDENTIFIER e.I)) e.Tokens t.Errs;

    /* TypeIdentifier -> CommonTypeIdentifier */
    () (CommonTypeIdentifier e.C) e.Tokens t.Errs
     = <TypeIdentifier ((CommonTypeIdentifier e.C)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
     = <TypeIdentifier () <CommonTypeIdentifier () e.Tokens t.Errs>>;

    ((CommonTypeIdentifier e.C)) e.Tokens t.Errs
     = (TypeIdentifier (CommonTypeIdentifier) (CommonTypeIdentifier e.C)) e.Tokens t.Errs;
}
```

Parser/Type/SimpleTypeList.ref

```
*$FROM Parser/SimpleType/SimpleType
$EXTERN SimpleType;

$ENTRY SimpleTypeList {
    /* SimpleTypeList -> SimpleType ',' SimpleTypeList */
    () (SimpleType e.S) e.Tokens t.Errs
        = <SimpleTypeList ((SimpleType e.S)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <SimpleTypeList () <SimpleType () e.Tokens t.Errs>>;

    (t.S) (',' e.C) e.Tokens t.Errs
        = <SimpleTypeList (t.S (',' e.C)) e.Tokens t.Errs>;

    (t.S t.C) (SimpleTypeList e.ST) e.Tokens t.Errs
     = <SimpleTypeList (t.S t.C (SimpleTypeList e.ST)) e.Tokens t.Errs>;
    (t.S t.C) e.Tokens t.Errs
     = <SimpleTypeList (t.S t.C) <SimpleTypeList () e.Tokens t.Errs>>;

    (t.S t.C t.ST) e.Tokens t.Errs
     = (SimpleTypeList (SimpleType ',' SimpleTypeList) t.S t.C t.ST) e.Tokens t.Errs;

    /* SimpleTypeList -> SimpleType */
    (t.S) e.Tokens t.Errs
        = (SimpleTypeList (SimpleType) t.S) e.Tokens t.Errs;
}
```

Parser/Type/Type.ref

```
*$FROM Parser/SimpleType/TypeIdentifier
*$FROM Parser/Type/TypeAfterPacked
*$FROM Parser/SimpleType/SimpleType
$EXTERN TypeIdentifier, TypeAfterPacked, SimpleType;
```

```
$ENTRY Type_ {
    /* Type -> '^' TypeIdentifier */
    () ('^' e.U) e.Tokens t.Errs
        = <Type_ (('^' e.U)) e.Tokens t.Errs>;

    (('^' e.U)) (TypeIdentifier e.T) e.Tokens t.Errs
       = <Type_ (('^' e.U) (TypeIdentifier e.T)) e.Tokens t.Errs>;
    (('^' e.U)) e.Tokens t.Errs
        = <Type_ (('^' e.U)) <TypeIdentifier () e.Tokens t.Errs>>;

    (('^' e.U) t.T) e.Tokens t.Errs
      = (Type ('^' TypeIdentifier) ('^' e.U) t.T) e.Tokens t.Errs;

    /* Type -> KW_PACKED TypeAfterPacked */
    () (KW_PACKED e.P) e.Tokens t.Errs
        = <Type_ ((KW_PACKED e.P)) e.Tokens t.Errs>;

    ((KW_PACKED e.P)) (TypeAfterPacked e.T) e.Tokens t.Errs
     = <Type_ ((KW_PACKED e.P) (TypeAfterPacked e.T)) e.Tokens t.Errs>;
    ((KW_PACKED e.P)) e.Tokens t.Errs
     = <Type_ (KW_PACKED e.P) <TypeAfterPacked () e.Tokens t.Errs>>;

    ((KW_PACKED) t.T) e.Tokens t.Errs
     = (Type (KW_PACKED TypeAfterPacked) (KW_PACKED) t.T) e.Tokens t.Errs;

    /* Type -> TypeAfterPacked */
    () (TypeAfterPacked e.T) e.Tokens t.Errs
        = <Type_ ((TypeAfterPacked e.T)) e.Tokens t.Errs>;
    () (s.TypeKW e.K) e.Tokens t.Errs
        , KW_ARRAY KW_FILE KW_SET KW_RECORD : e.1 s.TypeKW e.2
     = <Type_ () <TypeAfterPacked () (s.TypeKW e.K) e.Tokens t.Errs>>;

    ((TypeAfterPacked e.T)) e.Tokens t.Errs
     = (Type (TypeAfterPacked) (TypeAfterPacked e.T)) e.Tokens t.Errs;

    /* Type -> SimpleType */
    () (SimpleType e.S) e.Tokens t.Errs
        = <Type_ ((SimpleType e.S)) e.Tokens t.Errs>;
    () e.Tokens t.Errs
        = <Type_ () <SimpleType () e.Tokens t.Errs>>;

    ((SimpleType e.S)) e.Tokens t.Errs
        = (Type (SimpleType) (SimpleType e.S)) e.Tokens t.Errs;
}
```

Parser/Type/TypeAfterPacked.ref

```
*$FROM Parser/Type/SimpleTypeList
*$FROM Parser/Type/Type
*$FROM Parser/SimpleType/SimpleType
*$FROM Parser/FieldList/FieldList
$EXTERN SimpleTypeList, Type_, SimpleType, FieldList;

$ENTRY TypeAfterPacked {
    /* TypeAfterPacked -> KW_ARRAY SimpleTypeList KW_OF Type */
    () (KW_ARRAY e.A) e.Tokens t.Errs
        = <TypeAfterPacked ((KW_ARRAY e.A)) e.Tokens t.Errs>;

    ((KW_ARRAY e.A)) (SimpleTypeList e.S) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_ARRAY e.A) (SimpleTypeList e.S)) e.Tokens t.Errs>;
    ((KW_ARRAY e.A)) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_ARRAY e.A)) <SimpleTypeList () e.Tokens t.Errs>>;

    ((KW_ARRAY e.A) t.S) (KW_OF e.O) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_ARRAY e.A) t.S (KW_OF e.O)) e.Tokens t.Errs>;
   ((KW_ARRAY e.A) t.S) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
        = <TypeAfterPacked ((KW_ARRAY e.A) t.S)
          (KW_OF (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
            e.Tokens (e.Errs (t.Start TypeAfterPacked '.'
                Unexpected t.Type '.'
                Expected '[' KW_OF '].'
                Inserted '.'))>;
    ((KW_ARRAY e.A) t.S) '$' (e.Errs)
        = <TypeAfterPacked ((KW_ARRAY e.A) t.S)
            (KW_OF ((0 0) (0 0))) '$'
            (e.Errs ((EOF) TypeAfterPacked '.'
                Unexpected End Of File '.'
                Expected '[' KW_OF '].'
                Inserted '.'))>;

    ((KW_ARRAY e.A) t.S t.O) (Type e.T) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_ARRAY e.A) t.S t.O (Type e.T)) e.Tokens t.Errs>;
    ((KW_ARRAY e.A) t.S t.O) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_ARRAY e.A) t.S t.O) <Type_ () e.Tokens t.Errs>>;

    ((KW_ARRAY e.A) t.S t.O t.T) e.Tokens t.Errs
     = (TypeAfterPacked (KW_ARRAY SimpleTypeList KW_OF Type) (KW_ARRAY e.A) t.S t.O t.T) e.Toker

    /* TypeAfterPacked -> KW_FILE KW_OF Type */
    () (KW_FILE e.F) e.Tokens t.Errs
        = <TypeAfterPacked ((KW_FILE e.F) e.Tokens t.Errs)>;
```

```
 ((KW_FILE e.F)) (KW_OF e.O) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_FILE e.F) (KW_OF e.O)) e.Tokens t.Errs>;
((KW_FILE e.F)) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <TypeAfterPacked ((KW_FILE e.F))
       (KW_OF (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
         e.Tokens (e.Errs (t.Start TypeAfterPacked '.'
             Unexpected t.Type '.'
             Expected '[' KW_OF '].'
             Inserted '.'))>;
 ((KW_FILE e.F)) '$' (e.Errs)
     = <TypeAfterPacked ((KW_FILE e.F))
         (KW_OF ((0 0) (0 0))) '$'
         (e.Errs ((EOF) TypeAfterPacked '.'
             Unexpected End Of File '.'
             Expected '[' KW_OF '].'
             Inserted '.'))>;

 ((KW_FILE e.F) t.O) (Type e.T) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_FILE e.F) t.O (Type e.T)) e.Tokens t.Errs>;
 ((KW_FILE e.F) t.O) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_FILE e.F) t.O) <Type_ () e.Tokens t.Errs>>;

 ((KW_FILE e.F) t.O t.T) e.Tokens t.Errs
  = (TypeAfterPacked (KW_FILE KW_OF Type) (KW_FILE e.F) t.O t.T) e.Tokens t.Errs;

 /* TypeAfterPacked -> KW_SET KW_OF SimpleType */
 () (KW_SET e.S) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_SET e.S)) e.Tokens t.Errs>;

 ((KW_SET e.S)) (KW_OF e.O) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_SET e.S) (KW_OF e.O)) e.Tokens t.Errs>;
((KW_SET e.S)) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <TypeAfterPacked ((KW_SET e.S))
       (KW_OF (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
         e.Tokens (e.Errs (t.Start TypeAfterPacked '.'
             Unexpected t.Type '.'
             Expected '[' KW_OF '].'
             Inserted '.'))>;
 ((KW_SET e.S)) '$' (e.Errs)
     = <TypeAfterPacked ((KW_SET e.S))
         (KW_OF ((0 0) (0 0))) '$'
         (e.Errs ((EOF) TypeAfterPacked '.'
             Unexpected End Of File '.'
             Expected '[' KW_OF '].'
             Inserted '.'))>;
```

```
 ((KW_SET e.S) t.O) (SimpleType e.T) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_SET e.S) t.O (SimpleType e.T)) e.Tokens t.Errs>;
 ((KW_SET e.S) t.O) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_SET e.S) t.O) <SimpleType () e.Tokens t.Errs>>;

 ((KW_SET e.S) t.O t.T) e.Tokens t.Errs
  = (TypeAfterPacked (KW_SET KW_OF SimpleType) (KW_SET e.S) t.O t.T) e.Tokens t.Errs;

 /* TypeAfterPacked -> KW_RECORD FieldList KW_END */
 () (KW_RECORD e.R) e.Tokens t.Errs
     = <TypeAfterPacked ((KW_RECORD e.R)) e.Tokens t.Errs>;
 () (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <TypeAfterPacked ()
         e.Tokens (e.Errs (t.Start TypeAfterPacked '.'
             Unexpected t.Type '.'
        Expected '[' KW_ARRAY or KW_FILE or KW_SET or KW_RECORD '].'
             Skipped '.'))>;
 () '$' (e.Errs)
     = (TypeAfterPacked (EOF))
         '$'
         (e.Errs ((EOF) TypeAfterPacked '.'
             Unexpected End Of File '.'
        Expected '[' KW_ARRAY or KW_FILE or KW_SET or KW_RECORD '].'
             Terminated '.'));

 ((KW_RECORD e.R)) (FieldList e.F) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_RECORD e.R) (FieldList e.F)) e.Tokens t.Errs>;
 ((KW_RECORD e.R)) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_RECORD e.R)) <FieldList () e.Tokens t.Errs>>;

 ((KW_RECORD e.R) t.F) (KW_END e.E) e.Tokens t.Errs
  = <TypeAfterPacked ((KW_RECORD e.R) t.F (KW_END e.E)) e.Tokens t.Errs>;
((KW_RECORD e.R) t.F) (t.Type (t.Start t.End) e.Attrs) e.Tokens (e.Errs)
     = <TypeAfterPacked ((KW_RECORD e.R) t.F)
      (KW_END (t.Start t.Start)) (t.Type (t.Start t.End) e.Attrs)
         e.Tokens (e.Errs (t.Start TypeAfterPacked '.'
             Unexpected t.Type '.'
             Expected '[' KW_END '].'
             Inserted '.'))>;
 ((KW_RECORD e.R) t.F) '$' (e.Errs)
     = <TypeAfterPacked ((KW_RECORD e.R) t.F)
         (KW_END ((0 0) (0 0))) '$'
         (e.Errs ((EOF) TypeAfterPacked '.'
             Unexpected End Of File '.'
             Expected '[' KW_END '].'
```

```
                Inserted '.'))>;

    ((KW_RECORD e.R) t.F t.E) e.Tokens t.Errs
     = (TypeAfterPacked (KW_RECORD FieldList KW_END) (KW_RECORD e.R) t.F t.E) e.Tokens t.Errs;
}
```

Parser/Parser.ref

```
*$FROM Parser/Program/Program
$EXTERN Program;

/*
    <Parse t.Token*> == t.Errors t.TreeNode
    t.Errors ::= (t.Error*)
    t.Error ::= s.Char*
    t.TreeNode ::= (s.Nterm (s.Token*) t.TreeNode*)
*/
$ENTRY Parse {
    e.Tokens
        , <Program () e.Tokens '$' ()> : {
            t.ParseTree '$' t.Errors = t.ParseTree t.Errors;
           t.ParseTree (t.Type (t.Start t.End) e.Attr) e._ (e.Errs)
                = t.ParseTree (e.Errs (t.Start Parser '.'
                    Token t.Type and all next didnt recognized '.'
                    Expected End Of File '.'));
        };
}
```

Lexer.ref

```
*$FROM LibraryEx
$EXTERN Inc, Dec, Map;

/*
    e.Lines ::= t.Line*
    t.Line ::= (s.Sym*)
    t.Token ::=
    t.Fragment ::= (t.StartPos t.EndPos)
    t.StartPos ::= t.Pos
    t.EndPos ::= t.Pos
    t.Pos ::= (s.Line s.Col)
    t.Lexem ::= (s.Char+)

    s.Bool ::= TRUE | FALSE
*/
```

```
/* <Lexem-Pos e.Lines> == t.Token* */
$ENTRY Lexer {
    e.Lines = <Lexer-Pos (1 1) e.Lines>;
}

/* <Count e.Element*> ::= s.Count */
Count {
    /* пусто */ = 0;
    t._ e.Elements = <Inc <Count e.Elements>>;
}

/* <Join (e.Sep) e.Lines> ::= e.Sym* */
Join {
    (e.Sep) = /* пусто */;
    (e.Sep) t.Line = t.Line;
    (e.Sep) (e.Line) e.Lines = e.Line e.Sep <Join (e.Sep) e.Lines>
}

/* <Or s.Bool*> ::= s.Bool */
Or {
    e._ TRUE e._ = TRUE;
    e._ = FALSE;
}

/* <And s.Bool*> ::= s.Bool */
And {
    e._ FALSE e._ = FALSE;
    e._ = TRUE;
}

/* <CharEQ s.Sym s.Sym> ::= s.Bool */
CharEQ {
    s.Left s.Right
        , <Compare <Ord s.Left> <Ord s.Right>> : {
            '+' = FALSE;
            '0' = TRUE;
            '-' = FALSE;
        };
}

/* <CharLE s.Sym s.Sym> ::= s.Bool */
CharLE {
    s.Left s.Right
        , <Compare <Ord s.Left> <Ord s.Right>> : {
            '+' = FALSE;
            '0' = TRUE;
```

```
            '-' = TRUE;
        };
}

/* <CharGE s.Sym s.Sym> ::= s.Bool */
CharGE {
    s.Left s.Right
        , <Compare <Ord s.Left> <Ord s.Right>> : {
            '+' = TRUE;
            '0' = TRUE;
            '-' = FALSE;
        };
}

/* <IsDigit s.Sym> ::= s.Bool */
IsDigit {
    s.Sym = <And <CharGE s.Sym '0'> <CharLE s.Sym '9'>>;
}

/* <IsAlpha s.Sym> ::= TRUE | FALSE */
IsAlpha {
    s.LwSym
        , <Upper s.LwSym> : s.Sym
        = <And <CharGE s.Sym 'A'> <CharLE s.Sym 'Z'>>;
}

/* <Lexem-Pos (s.Line s.Col) e.Lines> == t.Token* */
Lexer-Pos {
    (s.Line s.Col) = /* пусто */;

    /* Space ::= \s */
   (s.Line s.Col) (' ' e.Syms) e.Lines = <Lexer-Pos (s.Line <Inc s.Col>) (e.Syms) e.Lines>;
   (s.Line s.Col) ('\r' e.Syms) e.Lines = <Lexer-Pos (s.Line <Inc s.Col>) (e.Syms) e.Lines>;
   (s.Line s.Col) ('\t' e.Syms) e.Lines = <Lexer-Pos (s.Line <Inc s.Col>) (e.Syms) e.Lines>;
   (s.Line s.Col) () e.Lines = <Lexer-Pos (<Inc s.Line> 1) e.Lines>;

    /* Comment ::= {.*} */
    (s.Line s.Col) ('{' e.Inf '}' e.Postf) e.Lines
      = <Lexer-Pos (s.Line <Add <Count e.Inf> 3>) (e.Postf) e.Lines>;
   (s.Line s.Col) ('{' e.Postf) e.Inf (e.Pref '}' e.Postf2) e.Lines
     = <Lexer-Pos (<Add s.Line <Add <Count e.Inf> 1>> <Add <Count e.Pref> 2>)
            (e.Postf2) e.Lines>;

    /* Comment ::= (*.**) */
    (s.Line s.Col) ('(*' e.Inf '*)' e.Postf) e.Lines
      = <Lexer-Pos (s.Line <Add <Count e.Inf> 5>) (e.Postf) e.Lines>;
```

```
(s.Line s.Col) ('(*' e.Postf) e.Inf (e.Pref '*)' e.Postf2) e.Lines
  = <Lexer-Pos (<Add s.Line <Add <Count e.Inf> 1>> <Add <Count e.Pref> 3>)
        (e.Postf2) e.Lines>;

 /* UNAR_SIGN ::= [+-] */
 (s.Line s.Col) ('+' e.Syms) e.Lines =
     (UNAR_SIGN ((s.Line s.Col) (s.Line <Inc s.Col>)) '+')
     <Lexer-Pos (s.Line <Inc s.Col>) (e.Syms) e.Lines>;
 (s.Line s.Col) ('-' e.Syms) e.Lines =
     ('-' ((s.Line s.Col) (s.Line <Inc s.Col>)) '-')
     <Lexer-Pos (s.Line <Inc s.Col>) (e.Syms) e.Lines>;

 /* CHAR_SEQUENCE ::= (?<=\')[^\']+(?=\') */
(s.Line s.Col) ('\'' e.Postf) e.Inf (e.Pref '\'' e.Postf2) e.Lines
  , <Add s.Line <Add <Count e.Inf> 1>> <Add <Count e.Pref> 1> : s.Line_ s.Col_
     =   (CHAR_SEQUENCE
             ((s.Line <Inc s.Col>) (s.Line_ s.Col_))
             (<Join ('\n') (e.Postf) e.Inf (e.Pref)>))
         <Lexer-Pos (s.Line_ <Inc s.Col_>) (e.Postf2) e.Lines>;
 (s.Line s.Col) ('\'' e.Inf '\'' e.Postf) e.Lines
     , <Add s.Col <Add <Count e.Inf> 1>> : s.Col_
     =   (CHAR_SEQUENCE
             ((s.Line <Inc s.Col>) (s.Line s.Col_))
             (e.Inf))
         <Lexer-Pos (s.Line <Inc s.Col_>) (e.Postf) e.Lines>;

 /*
     UNSIGNED_NUMBER ::= [0-9]+(\.[0-9]+)?(E[+-]?[0-9]+)?
     IDENTIFIER ::= [a-zA-Z][a-zA-Z0-9]*
     CHAR ::= .
 */
 (s.Line s.Col) (s.LwSym e.LwSyms) e.Lines
     , <Upper s.LwSym> : s.Sym
     , <Map Upper e.LwSyms> : e.Syms
     , <IsDigit s.Sym> : {
     TRUE, <ReadNum (s.Col) s.Sym e.Syms> : e.Lexem_ (s.Col_) e.Syms_
             = UNSIGNED_NUMBER ((e.Lexem_)) (s.Col_) e.Syms_;
         FALSE, <IsAlpha s.Sym> : {
        TRUE, <ReadIdent (s.Col) s.Sym e.Syms> : e.Lexem_ (s.Col_) e.Syms_
                 , e.Lexem_ : {
                     'INTEGER' = KW_INTEGER;
                     'BOOLEAN' = KW_BOOLEAN;
                     'REAL' = KW_REAL;
                     'CHAR' = KW_CHAR;
                     'TEXT' = KW_TEXT;
                     'PACKED' = KW_PACKED;
```

```
                        'ARRAY' = KW_ARRAY;
                        'OF' = KW_OF;
                        'FILE' = KW_FILE;
                        'SET' = KW_SET;
                        'RECORD' = KW_RECORD;
                        'END' = KW_END;
                        'CASE' = KW_CASE;
                        'CONST' = KW_CONST;
                        'TYPE' = KW_TYPE;
                        e._ = IDENTIFIER (e.Lexem_);
                    } : t.LexemType_ e.Lexem__
                    = t.LexemType_ (e.Lexem__) (s.Col_) e.Syms_;
                FALSE = s.Sym () (<Inc s.Col>) e.Syms;
            };
        } : s.LexemType (e.Lexem) (s.Col__) e.Syms__
        = (s.LexemType ((s.Line s.Col) (s.Line s.Col__)) e.Lexem)
            <Lexer-Pos (s.Line s.Col__) (e.Syms__) e.Lines>;
}

/* <ReadNum (s.Col) e.Syms> == e.Lexem (s.Col) e.Syms */
ReadNum {
    (s.Col) e.Syms
        , <ReadDigits (s.Col) e.Syms> : e.Int (s.Col_) e.Syms_
        , <ReadNumDot (s.Col_) e.Syms_> : e.Dot (s.Col__) e.Syms__
        = e.Int e.Dot <ReadNumExp (s.Col__) e.Syms__>;
}
ReadDigits {
    (s.Col) = (s.Col);
    (s.Col) s.Sym e.Syms
        , <IsDigit s.Sym> : {
            TRUE = s.Sym <ReadDigits (<Inc s.Col>) e.Syms>;
            FALSE = (s.Col) s.Sym e.Syms;
        }
}
ReadNumDot {
    (s.Col) '.' s.Sym e.Syms
        , <IsDigit s.Sym> : {
            TRUE = '.' <ReadDigits (<Inc s.Col>) s.Sym e.Syms>;
            FALSE = (s.Col) '.' s.Sym e.Syms;
        };
    (s.Col) e.Syms = (s.Col) e.Syms;
}
ReadNumSign {
    (s.Col) '+' e.Syms = '+' (<Inc s.Col>) e.Syms;
    (s.Col) '-' e.Syms = '-' (<Inc s.Col>) e.Syms;
    (s.Col) e.Syms = (s.Col) e.Syms;
```

```
}
ReadNumExp {
    (s.Col) 'E' s.Sym e.Syms
        , <ReadNumSign (<Inc s.Col>) s.Sym e.Syms> : {
            e.Sign (s.Col_) s.Sym_ e.Syms_
                , <IsDigit s.Sym_> : {
              TRUE = 'E' e.Sign <ReadDigits (s.Col_) s.Sym_ e.Syms_>;
                    FALSE = (s.Col) 'E' e.Sign s.Sym_ e.Syms_;
                };
            e.Sign (s.Col_) = (s.Col) 'E' e.Sign;
        };
    (s.Col) e.Syms = (s.Col) e.Syms;
}

/* <ReadIdent (s.Col) e.Syms> == e.Lexem (s.Col) e.Syms */
ReadIdent {
    (s.Col) = (s.Col);
    (s.Col) s.Sym e.Syms
        , <Or <IsAlpha s.Sym> <IsDigit s.Sym>> : {
            TRUE = s.Sym <ReadIdent (<Inc s.Col>) e.Syms>;
            FALSE = (s.Col) s.Sym e.Syms;
        }
}
```

Main.ref

```
*$FROM LibraryEx
*$FROM Lexer
*$FROM Parser/Parser
$EXTERN ArgList, LoadFile, Map, Lexer, Parse;

$ENTRY Go {
    /* пусто */ = <Main <ArgList>>
}

Main {
    (e.ProgName) (e.InputFile)
        /** /, <Lexer <LoadFile e.InputFile>> : e.Tokens
        = <Map Prout e.Tokens>/*/
        , <Lexer <LoadFile e.InputFile>> : e.Tokens
        , <Parse e.Tokens> : t.ParseTree (e.Errors)
        = <Print-ParseTree () t.ParseTree>
          <Prout "\nERRORS:">
          <Map {(e.Err) = <Prout 'Error: ' e.Err>;} e.Errors>;/**/
}
```

```
Print-ParseTree {
    (e.Indent) () = /* пусто */;
    (e.Indent) (s.Nterm (t.Start t.End))
        = <Prout e.Indent '\'' s.Nterm '\' -> ' t.Start '-' t.End>;
    (e.Indent) (s.Nterm (t.Start t.End) (e.Value))
            = <Prout e.Indent '\'' s.Nterm '\' -> ' t.Start '-
' t.End ' "' e.Value '"'>;
    (e.Indent) (s.Nterm ())
        = <Prout e.Indent s.Nterm ' -> ε'>;
    (e.Indent) (s.Nterm (e.Tokens) e.TreeNodes)
        = <Prout e.Indent s.Nterm ' -> ' e.Tokens>
          <Map {
          t.TreeNode = <Print-ParseTree (e.Indent '  ') t.TreeNode>;
          } e.TreeNodes>;
}
```

Тестирование

Входные данные

```
Type
  Coords = Record x, y: INTEGER end;
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

const
  Heigh = 480;
  Width = 640;
  Lines = 24;
  Columns = 80;
type
  BaseColor = (red, green, blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
      SymColor : Color;
      BackColor : Color
    end;

(* определения токенов }
{ определения токенов *)
```

```
(* определения токенов *)
{ определения токенов }
{ определения токенов *)
(* определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL

  end;

  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end
```

Вывод на `stdout`

```
*Compiling Main.ref:
+Linking C:\...\refal-5-lambda\lib\references\Library.rasl
+Linking C:\...\refal-5-lambda\lib\slim\exe\LibraryEx.rasl
*Compiling Lexer.ref:
*Compiling Parser/Parser.ref:
*Compiling Parser/Program/Program.ref:
*Compiling Parser/Block/Block.ref:
*Compiling Parser/Block/BlockConstSequence.ref:
*Compiling Parser/Block/BlockConst.ref:
*Compiling Parser/Constant/Constant.ref:
*Compiling Parser/Constant/UnarSign.ref:
*Compiling Parser/Constant/ConstantIdentifier.ref:
*Compiling Parser/Block/BlockTypeSequence.ref:
*Compiling Parser/Block/BlockType.ref:
```

```
*Compiling Parser/Type/Type.ref:
*Compiling Parser/SimpleType/TypeIdentifier.ref:
*Compiling Parser/SimpleType/CommonTypeIdentifier.ref:
*Compiling Parser/Type/TypeAfterPacked.ref:
*Compiling Parser/Type/SimpleTypeList.ref:
*Compiling Parser/SimpleType/SimpleType.ref:
*Compiling Parser/SimpleType/IdentifierList.ref:
*Compiling Parser/FieldList/FieldList.ref:
*Compiling Parser/FieldList/IdentifierWithTypeList.ref:
*Compiling Parser/FieldList/IdentifierWithType.ref:
*Compiling Parser/FieldList/CaseBlock.ref:
*Compiling Parser/FieldList/CaseVariantSequence.ref:
*Compiling Parser/FieldList/CaseVariant.ref:
*Compiling Parser/FieldList/ConstantList.ref:
** Compilation succeeded **
Program  -> Block
  Block  -> KW_TYPE BlockTypeSequence Block
    'KW_TYPE ' -> (1 1 )-(1 5 )
    BlockTypeSequence  -> BlockType
      BlockType  -> IDENTIFIER =Type ;
        'IDENTIFIER ' -> (2 3 )-(2 9 ) "COORDS"
        '=' -> (2 10 )-(2 11 )
        Type  -> TypeAfterPacked
          TypeAfterPacked  -> KW_RECORD FieldList KW_END
            'KW_RECORD ' -> (2 12 )-(2 18 )
            FieldList  -> IdentifierWithTypeList
              IdentifierWithTypeList  -> IdentifierWithType
                IdentifierWithType  -> IdentifierList :Type
                  IdentifierList  -> IDENTIFIER ,IdentifierList
                    'IDENTIFIER ' -> (2 19 )-(2 20 ) "X"
                    ',' -> (2 20 )-(2 21 )
                    IdentifierList  -> IDENTIFIER
                      'IDENTIFIER ' -> (2 22 )-(2 23 ) "Y"
                  ':' -> (2 23 )-(2 24 )
                  Type  -> SimpleType
                    SimpleType  -> TypeIdentifier
                      TypeIdentifier  -> CommonTypeIdentifier
                        CommonTypeIdentifier  -> KW_INTEGER
                          'KW_INTEGER ' -> (2 25 )-(2 32 )
            'KW_END ' -> (2 33 )-(2 36 )
        ';' -> (2 36 )-(2 37 )
    Block  -> KW_CONST BlockConstSequence Block
      'KW_CONST ' -> (3 1 )-(3 6 )
      BlockConstSequence  -> BlockConst
        BlockConst  -> IDENTIFIER =Constant ;
          'IDENTIFIER ' -> (4 3 )-(4 12 ) "MAXPOINTS"
```

```
    '=' -> (4 13 )-(4 14 )
    Constant  -> UNSIGNED_NUMBER
      'UNSIGNED_NUMBER ' -> (4 15 )-(4 18 ) "100"
    ';' -> (4 18 )-(4 19 )
Block  -> KW_TYPE BlockTypeSequence Block
  'KW_TYPE ' -> (5 1 )-(5 5 )
  BlockTypeSequence  -> BlockType
    BlockType  -> IDENTIFIER =Type ;
      'IDENTIFIER ' -> (6 3 )-(6 15 ) "COORDSVECTOR"
      '=' -> (6 16 )-(6 17 )
      Type  -> TypeAfterPacked
       TypeAfterPacked  -> KW_ARRAY SimpleTypeList KW_OF Type
          'KW_ARRAY ' -> (6 18 )-(6 23 )
          SimpleTypeList  -> SimpleType
            SimpleType  -> Constant ..Constant
              '.' -> (6 25 )-(6 26 )
              '.' -> (6 26 )-(6 27 )
              Constant  -> ConstantIdentifier
                ConstantIdentifier  -> IDENTIFIER
                 'IDENTIFIER ' -> (6 27 )-(6 36 ) "MAXPOINTS"
          'KW_OF ' -> (6 37 )-(6 39 )
          Type  -> SimpleType
            SimpleType  -> TypeIdentifier
              TypeIdentifier  -> IDENTIFIER
                'IDENTIFIER ' -> (6 40 )-(6 46 ) "COORDS"
      ';' -> (6 46 )-(6 47 )
  Block  -> KW_CONST BlockConstSequence Block
    'KW_CONST ' -> (8 1 )-(8 6 )
    BlockConstSequence  -> BlockConst BlockConstSequence
      BlockConst  -> IDENTIFIER =Constant ;
        'IDENTIFIER ' -> (9 3 )-(9 8 ) "HEIGH"
        '=' -> (9 9 )-(9 10 )
        Constant  -> UNSIGNED_NUMBER
          'UNSIGNED_NUMBER ' -> (9 11 )-(9 14 ) "480"
        ';' -> (9 14 )-(9 15 )
      BlockConstSequence  -> BlockConst BlockConstSequence
        BlockConst  -> IDENTIFIER =Constant ;
          'IDENTIFIER ' -> (10 3 )-(10 8 ) "WIDTH"
          '=' -> (10 9 )-(10 10 )
          Constant  -> UNSIGNED_NUMBER
            'UNSIGNED_NUMBER ' -> (10 11 )-(10 14 ) "640"
          ';' -> (10 14 )-(10 15 )
        BlockConstSequence  -> BlockConst BlockConstSequence
          BlockConst  -> IDENTIFIER =Constant ;
            'IDENTIFIER ' -> (11 3 )-(11 8 ) "LINES"
            '=' -> (11 9 )-(11 10 )
```

```
        Constant  -> UNSIGNED_NUMBER
          'UNSIGNED_NUMBER ' -> (11 11 )-(11 13 ) "24"
        ';' -> (11 13 )-(11 14 )
      BlockConstSequence  -> BlockConst
        BlockConst  -> IDENTIFIER =Constant ;
          'IDENTIFIER ' -> (12 3 )-(12 10 ) "COLUMNS"
          '=' -> (12 11 )-(12 12 )
          Constant  -> UNSIGNED_NUMBER
            'UNSIGNED_NUMBER ' -> (12 13 )-(12 15 ) "80"
          ';' -> (12 15 )-(12 16 )
Block  -> KW_TYPE BlockTypeSequence Block
  'KW_TYPE ' -> (13 1 )-(13 5 )
  BlockTypeSequence  -> BlockType BlockTypeSequence
    BlockType  -> IDENTIFIER =Type ;
      'IDENTIFIER ' -> (14 3 )-(14 12 ) "BASECOLOR"
      '=' -> (14 13 )-(14 14 )
      Type  -> SimpleType
        SimpleType  -> (IdentifierList )
          '(' -> (14 15 )-(14 16 )
          IdentifierList  -> IDENTIFIER ,IdentifierList
            'IDENTIFIER ' -> (14 16 )-(14 19 ) "RED"
            ',' -> (14 19 )-(14 20 )
           IdentifierList  -> IDENTIFIER ,IdentifierList
             'IDENTIFIER ' -> (14 21 )-(14 26 ) "GREEN"
             ',' -> (14 26 )-(14 27 )
            IdentifierList  -> IDENTIFIER ,IdentifierList
                'IDENTIFIER ' -> (14 28 )-(14 32 ) "BLUE"
                 ',' -> (14 32 )-(14 33 )
                 IdentifierList  -> IDENTIFIER
           'IDENTIFIER ' -> (14 34 )-(14 43 ) "HIGHLITED"
        ')' -> (14 43 )-(14 44 )
      ';' -> (14 44 )-(14 45 )
    BlockTypeSequence  -> BlockType BlockTypeSequence
      BlockType  -> IDENTIFIER =Type ;
        'IDENTIFIER ' -> (15 3 )-(15 8 ) "COLOR"
        '=' -> (15 9 )-(15 10 )
        Type  -> TypeAfterPacked
          TypeAfterPacked  -> KW_SET KW_OF SimpleType
            'KW_SET ' -> (15 11 )-(15 14 )
            'KW_OF ' -> (15 15 )-(15 17 )
            SimpleType  -> TypeIdentifier
              TypeIdentifier  -> IDENTIFIER
            'IDENTIFIER ' -> (15 18 )-(15 27 ) "BASECOLOR"
        ';' -> (15 27 )-(15 28 )
      BlockTypeSequence  -> BlockType BlockTypeSequence
        BlockType  -> IDENTIFIER =Type ;
```

```
                    'IDENTIFIER ' -> (16 3 )-(16 16 ) "GRAPHICSCREEN"
                      '=' -> (16 17 )-(16 18 )
                      Type  -> TypeAfterPacked
                  TypeAfterPacked  -> KW_ARRAY SimpleTypeList KW_OF Type
                           'KW_ARRAY ' -> (16 19 )-(16 24 )
                           SimpleTypeList  -> SimpleType
                             SimpleType  -> Constant ..Constant
                               '.' -> (16 26 )-(16 27 )
                               '.' -> (16 27 )-(16 28 )
                               Constant  -> ConstantIdentifier
                                 ConstantIdentifier  -> IDENTIFIER
                             'IDENTIFIER ' -> (16 28 )-(16 33 ) "HEIGH"
                           'KW_OF ' -> (16 34 )-(16 36 )
                           Type  -> TypeAfterPacked
                                                  TypeAfterPacked    -
> KW_ARRAY SimpleTypeList KW_OF Type
                               'KW_ARRAY ' -> (16 37 )-(16 42 )
                               SimpleTypeList  -> SimpleType
                                 SimpleType  -> Constant ..Constant
                                   '.' -> (16 44 )-(16 45 )
                                   '.' -> (16 45 )-(16 46 )
                                   Constant  -> ConstantIdentifier
                                     ConstantIdentifier  -> IDENTIFIER
                              'IDENTIFIER ' -> (16 46 )-(16 51 ) "WIDTH"
                               'KW_OF ' -> (16 52 )-(16 54 )
                               Type  -> SimpleType
                                 SimpleType  -> TypeIdentifier
                                   TypeIdentifier  -> IDENTIFIER
                             'IDENTIFIER ' -> (16 55 )-(16 60 ) "COLOR"
                      ';' -> (16 60 )-(16 61 )
                    BlockTypeSequence  -> BlockType
                      BlockType  -> IDENTIFIER =Type ;
                       'IDENTIFIER ' -> (17 3 )-(17 13 ) "TEXTSCREEN"
                        '=' -> (17 14 )-(17 15 )
                        Type  -> TypeAfterPacked
                   TypeAfterPacked  -> KW_ARRAY SimpleTypeList KW_OF Type
                             'KW_ARRAY ' -> (17 16 )-(17 21 )
                             SimpleTypeList  -> SimpleType
                               SimpleType  -> Constant ..Constant
                                 '.' -> (17 23 )-(17 24 )
                                 '.' -> (17 24 )-(17 25 )
                                 Constant  -> ConstantIdentifier
                                   ConstantIdentifier  -> IDENTIFIER
                             'IDENTIFIER ' -> (17 25 )-(17 30 ) "LINES"
                             'KW_OF ' -> (17 31 )-(17 33 )
                             Type  -> TypeAfterPacked
```

```
                                                TypeAfterPacked   -
> KW_ARRAY SimpleTypeList KW_OF Type
                              'KW_ARRAY ' -> (17 34 )-(17 39 )
                              SimpleTypeList  -> SimpleType
                                SimpleType  -> Constant ..Constant
                                  '.' -> (17 41 )-(17 42 )
                                  '.' -> (17 42 )-(17 43 )
                                  Constant  -> ConstantIdentifier
                                 ConstantIdentifier  -> IDENTIFIER
                                        'IDENTIFIER ' -> (17 43 )-
(17 50 ) "COLUMNS"
                              'KW_OF ' -> (17 51 )-(17 53 )
                              Type  -> TypeAfterPacked
                      TypeAfterPacked  -> KW_RECORD FieldList KW_END
                                  'KW_RECORD ' -> (18 5 )-(18 11 )
                               FieldList  -> IdentifierWithTypeList
                                           IdentifierWithTypeList  -
> IdentifierWithType ;IdentifierWithTypeList
                                              IdentifierWithType  -
> IdentifierList :Type
                                     IdentifierList  -> IDENTIFIER
                                         'IDENTIFIER ' -> (19 7 )-
(19 13 ) "SYMBOL"
                                        ':' -> (19 14 )-(19 15 )
                                        Type  -> SimpleType
                                     SimpleType  -> TypeIdentifier
                                                 TypeIdentifier  -
> CommonTypeIdentifier
                                   CommonTypeIdentifier  -> KW_CHAR
                                    'KW_CHAR ' -> (19 16 )-(19 20 )
                                     ';' -> (19 20 )-(19 21 )
                                        IdentifierWithTypeList  -
> IdentifierWithType ;IdentifierWithTypeList
                                              IdentifierWithType  -
> IdentifierList :Type
                                     IdentifierList  -> IDENTIFIER
                                         'IDENTIFIER ' -> (20 7 )-
(20 15 ) "SYMCOLOR"
                                          ':' -> (20 16 )-(20 17 )
                                          Type  -> SimpleType
                                     SimpleType  -> TypeIdentifier
                                     TypeIdentifier  -> IDENTIFIER
                                                   'IDENTIFIER ' -
> (20 18 )-(20 23 ) "COLOR"
                                        ';' -> (20 23 )-(20 24 )
                                        IdentifierWithTypeList  -
```

```
> IdentifierWithType
                                                IdentifierWithType  -
> IdentifierList :Type
                                          IdentifierList  -> IDENTIFIER
                                                      'IDENTIFIER ' -
> (21 7 )-(21 16 ) "BACKCOLOR"
                                               ':' -> (21 17 )-(21 18 )
                                                 Type  -> SimpleType
                                          SimpleType  -> TypeIdentifier
                                          TypeIdentifier  -> IDENTIFIER
                                                      'IDENTIFIER ' -
> (21 19 )-(21 24 ) "COLOR"
                                     'KW_END ' -> (22 5 )-(22 8 )
                        ';' -> (22 8 )-(22 9 )
              Block  -> KW_TYPE BlockTypeSequence Block
                'KW_TYPE ' -> (30 1 )-(30 5 )
                BlockTypeSequence  -> BlockType BlockTypeSequence
                  BlockType  -> IDENTIFIER =Type ;
                    'IDENTIFIER ' -> (31 3 )-(31 9 ) "DOMAIN"
                    '=' -> (31 10 )-(31 11 )
                    Type  -> SimpleType
                      SimpleType  -> (IdentifierList )
                        '(' -> (31 12 )-(31 13 )
                       IdentifierList  -> IDENTIFIER ,IdentifierList
                          'IDENTIFIER ' -> (31 13 )-(31 18 ) "IDENT"
                          ',' -> (31 18 )-(31 19 )
                        IdentifierList  -> IDENTIFIER ,IdentifierList
                        'IDENTIFIER ' -> (31 20 )-(31 29 ) "INTNUMBER"
                            ',' -> (31 29 )-(31 30 )
                            IdentifierList  -> IDENTIFIER
                       'IDENTIFIER ' -> (31 31 )-(31 41 ) "REALNUMBER"
                        ')' -> (31 41 )-(31 42 )
                    ';' -> (31 42 )-(31 43 )
                    BlockTypeSequence  -> BlockType BlockTypeSequence
                      BlockType  -> IDENTIFIER =Type ;
                        'IDENTIFIER ' -> (32 3 )-(32 8 ) "TOKEN"
                        '=' -> (32 9 )-(32 10 )
                        Type  -> TypeAfterPacked
                         TypeAfterPacked  -> KW_RECORD FieldList KW_END
                            'KW_RECORD ' -> (32 11 )-(32 17 )
                        FieldList  -> IdentifierWithTypeList ;CaseBlock
                          IdentifierWithTypeList  -> IdentifierWithType
                            IdentifierWithType  -> IdentifierList :Type
                                  IdentifierList  -> IDENTIFIER
                             'IDENTIFIER ' -> (33 5 )-(33 13 ) "FRAGMENT"
                                  ':' -> (33 14 )-(33 15 )
```

```
                                Type  -> TypeAfterPacked
                         TypeAfterPacked  -> KW_RECORD FieldList KW_END
                                    'KW_RECORD ' -> (33 16 )-(33 22 )
                                  FieldList  -> IdentifierWithTypeList
                                             IdentifierWithTypeList  -
> IdentifierWithType
                                                IdentifierWithType  -
> IdentifierList :Type
                                                    IdentifierList  -
> IDENTIFIER ,IdentifierList
                                                 'IDENTIFIER ' -> (34 7 )-
(34 12 ) "START"
                                                  ',' -> (34 12 )-(34 13 )
                                               IdentifierList  -> IDENTIFIER
                                                 'IDENTIFIER ' -> (34 14 )-
(34 23 ) "FOLLOWING"
                                                ':' -> (34 24 )-(34 25 )
                                               Type  -> TypeAfterPacked
                                                       TypeAfterPacked  -
> KW_RECORD FieldList KW_END
                                           'KW_RECORD ' -> (34 26 )-(34 32 )
                                         FieldList  -> IdentifierWithTypeList
                                                     IdentifierWithTypeList  -
> IdentifierWithType
                                                         IdentifierWithType  -
> IdentifierList :Type
                                                             IdentifierList  -
> IDENTIFIER ,IdentifierList
                                                               'IDENTIFIER ' -
> (35 9 )-(35 12 ) "ROW"
                                                    ',' -> (35 12 )-(35 13 )
                                                IdentifierList  -> IDENTIFIER
                                                               'IDENTIFIER ' -
> (35 14 )-(35 17 ) "COL"
                                                    ':' -> (35 18 )-(35 19 )
                                                        Type  -> SimpleType
                                                SimpleType  -> TypeIdentifier
                                                             TypeIdentifier  -
> CommonTypeIdentifier
                                                   CommonTypeIdentifier  -
> KW_INTEGER
                                                             'KW_INTEGER ' -
> (35 20 )-(35 27 )
                                            'KW_END ' -> (36 7 )-(36 10 )
                                       'KW_END ' -> (37 5 )-(37 8 )
                                ';' -> (37 8 )-(37 9 )
```

```
                  CaseBlock  -> KW_CASE IDENTIFIER :TypeIdentifier KW_OF CaseVariantSequence
                             'KW_CASE ' -> (38 5 )-(38 9 )
                           'IDENTIFIER ' -> (38 10 )-(38 17 ) "TOKTYPE"
                             ':' -> (38 18 )-(38 19 )
                             TypeIdentifier  -> IDENTIFIER
                           'IDENTIFIER ' -> (38 20 )-(38 26 ) "DOMAIN"
                             'KW_OF ' -> (38 27 )-(38 29 )
                                                 CaseVariantSequence   -
> CaseVariant ;CaseVariantSequence
                          CaseVariant  -> ConstantList :(FieldList )
                                 ConstantList  -> Constant
                                   Constant  -> ConstantIdentifier
                                   ConstantIdentifier  -> IDENTIFIER
                              'IDENTIFIER ' -> (39 7 )-(39 12 ) "IDENT"
                                 ':' -> (39 13 )-(39 14 )
                                 '(' -> (39 15 )-(39 16 )
                               FieldList  -> IdentifierWithTypeList
                                           IdentifierWithTypeList  -
> IdentifierWithType
                                              IdentifierWithType  -
> IdentifierList :Type
                                        IdentifierList  -> IDENTIFIER
                             'IDENTIFIER ' -> (40 9 )-(40 13 ) "NAME"
                                          ':' -> (40 14 )-(40 15 )
                                          Type  -> TypeAfterPacked
                                                    TypeAfterPacked  -
> KW_ARRAY SimpleTypeList KW_OF Type
                                       'KW_ARRAY ' -> (40 16 )-(40 21 )
                                         SimpleTypeList  -> SimpleType
                                      SimpleType  -> Constant ..Constant
                                                '.' -> (40 23 )-(40 24 )
                                                '.' -> (40 24 )-(40 25 )
                                           Constant  -> UNSIGNED_NUMBER
                                                   'UNSIGNED_NUMBER ' -
> (40 25 )-(40 27 ) "32"
                                          'KW_OF ' -> (40 28 )-(40 30 )
                                              Type  -> SimpleType
                                          SimpleType  -> TypeIdentifier
                                                      TypeIdentifier  -
> CommonTypeIdentifier
                                       CommonTypeIdentifier  -> KW_CHAR
                                         'KW_CHAR ' -> (40 31 )-(40 35 )
                                    ')' -> (41 7 )-(41 8 )
                                  ';' -> (41 8 )-(41 9 )
                                                 CaseVariantSequence   -
> CaseVariant ;CaseVariantSequence
```

```
                        CaseVariant  -> ConstantList :(FieldList )
                                ConstantList  -> Constant
                                  Constant  -> ConstantIdentifier
                                ConstantIdentifier  -> IDENTIFIER
                                        'IDENTIFIER ' -> (42 7 )-
(42 16 ) "INTNUMBER"
                                    ':' -> (42 17 )-(42 18 )
                                    '(' -> (42 19 )-(42 20 )
                             FieldList  -> IdentifierWithTypeList
                                        IdentifierWithTypeList  -
> IdentifierWithType
                                            IdentifierWithType  -
> IdentifierList :Type
                                     IdentifierList  -> IDENTIFIER
                                         'IDENTIFIER ' -> (43 9 )-
(43 15 ) "INTVAL"
                                       ':' -> (43 16 )-(43 17 )
                                       Type  -> SimpleType
                                     SimpleType  -> TypeIdentifier
                                                TypeIdentifier  -
> CommonTypeIdentifier
                                  CommonTypeIdentifier  -> KW_INTEGER
                                    'KW_INTEGER ' -> (43 18 )-(43 25 )
                                    ')' -> (44 7 )-(44 8 )
                                 ';' -> (44 8 )-(44 9 )
                              CaseVariantSequence  -> CaseVariant
                        CaseVariant  -> ConstantList :(FieldList )
                                   ConstantList  -> Constant
                                  Constant  -> ConstantIdentifier
                                ConstantIdentifier  -> IDENTIFIER
                                        'IDENTIFIER ' -> (45 7 )-
(45 17 ) "REALNUMBER"
                                    ':' -> (45 18 )-(45 19 )
                                    '(' -> (45 20 )-(45 21 )
                             FieldList  -> IdentifierWithTypeList
                                        IdentifierWithTypeList  -
> IdentifierWithType
                                            IdentifierWithType  -
> IdentifierList :Type
                                     IdentifierList  -> IDENTIFIER
                                         'IDENTIFIER ' -> (46 9 )-
(46 16 ) "REALVAL"
                                          ':' -> (46 17 )-(46 18 )
                                          Type  -> SimpleType
                                     SimpleType  -> TypeIdentifier
                                                TypeIdentifier  -
```

```
> CommonTypeIdentifier
                                    CommonTypeIdentifier  -> KW_REAL
                                      'KW_REAL ' -> (46 19 )-(46 23 )
                                     ')' -> (48 3 )-(48 3 )
                          'KW_END ' -> (48 3 )-(48 6 )
                        ';' -> (48 6 )-(48 7 )
                    BlockTypeSequence  -> BlockType BlockTypeSequence
                       BlockType  -> IDENTIFIER =Type ;
                         'IDENTIFIER ' -> (50 3 )-(50 7 ) "YEAR"
                         '=' -> (50 8 )-(50 9 )
                         Type  -> SimpleType
                           SimpleType  -> Constant ..Constant
                             '.' -> (50 14 )-(50 15 )
                             '.' -> (50 15 )-(50 16 )
                            Constant  -> UNSIGNED_NUMBER
                         'UNSIGNED_NUMBER ' -> (50 16 )-(50 20 ) "2050"
                         ';' -> (50 20 )-(50 21 )
                       BlockTypeSequence  -> BlockType
                         BlockType  -> IDENTIFIER =Type ;
                           'IDENTIFIER ' -> (52 3 )-(52 7 ) "LIST"
                           '=' -> (52 8 )-(52 9 )
                           Type  -> TypeAfterPacked
                        TypeAfterPacked  -> KW_RECORD FieldList KW_END
                              'KW_RECORD ' -> (52 10 )-(52 16 )
                              FieldList  -> IdentifierWithTypeList
                                            IdentifierWithTypeList   -
> IdentifierWithType ;IdentifierWithTypeList
                           IdentifierWithType  -> IdentifierList :Type
                                     IdentifierList  -> IDENTIFIER
                                'IDENTIFIER ' -> (53 5 )-(53 10 ) "VALUE"
                                      ':' -> (53 11 )-(53 12 )
                                      Type  -> SimpleType
                                        SimpleType  -> TypeIdentifier
                                          TypeIdentifier  -> IDENTIFIER
                                             'IDENTIFIER ' -> (53 13 )-
(53 18 ) "TOKEN"
                                   ';' -> (53 18 )-(53 19 )
                         IdentifierWithTypeList  -> IdentifierWithType
                           IdentifierWithType  -> IdentifierList :Type
                                       IdentifierList  -> IDENTIFIER
                                    'IDENTIFIER ' -> (54 5 )-(54 9 ) "NEXT"
                                      ':' -> (54 10 )-(54 11 )
                                      Type  -> ^TypeIdentifier
                                        '^' -> (54 12 )-(54 13 )
                                        TypeIdentifier  -> IDENTIFIER
                                           'IDENTIFIER ' -> (54 13 )-
```

```
(54 17 ) "LIST"
                            'KW_END ' -> (55 3 )-(55 6 )
                        ';' -> (57 1 )-(57 1 )
              Block  -> KW_CONST BlockConstSequence Block
                'KW_CONST ' -> (57 1 )-(57 6 )
                BlockConstSequence  -> BlockConst
                  BlockConst  -> IDENTIFIER =Constant ;
                    'IDENTIFIER ' -> (58 3 )-(58 10 ) "CHARSEQ"
                    '=' -> (58 11 )-(58 12 )
                    Constant  -> CHAR_SEQUENCE
                   'CHAR_SEQUENCE ' -> (58 14 )-(59 10 ) "ASD123DWQ
(  weqe213)"
                    ';' -> (59 11 )-(59 12 )
                Block  -> ╬

ERRORS:
Error: (48 3 )CaseVariant .Unexpected KW_END .Expected [)].Inserted .
Error: (57 1 )BlockType .Unexpected KW_CONST .Expected [;].Inserted .
```

Вывод

В результате выполнения данной работы были изучениы алгоритмов построения парсеров методом рекурсивного спуска.