Лабораторная работа № 2.4 «Рекурсивный спуск»

18 июня 2025 г.

Александр Старовойтов, ИУ9-61Б

Цель работы

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

Индивидуальный вариант

Подмножество Рефала-5

```
/* Декартово произведение */
CartProd {
  (t.X e.X) (e.Y) = <CartProd-Bind t.X e.Y> <CartProd (e.X) (e.Y)>;
  (/* пусто */) (e.Y) = /* пусто */;
}

CartProd-Bind {
  t.X t.Y e.Y = (t.X t.Y) <CartProd-Bind t.X e.Y>;
  t.X /* пусто */ = /* пусто */;
}

/*
  Прибавление 1 к строковой записи числа:
  123 → 124, 99 → 100, 007 → 008
*/
Inc {
 e.Prefix s.Last, '0123456789' : e.1 s.Last s.Next e.2 = e.Prefix s.Next;
  e.Prefix '9' = <Inc e.Prefix> '0';
  /* пусто */ = '1';
}

/* Функция возвращает уникальные идентификаторы с заданным префиксом */
NextId {
```

```
  e.prefix
    , <Dg counter>
    : {
        s.n = e.prefix <Symb s.n> <Br counter '=' <Add 1 s.n>>;
        /* пусто */ = <Br counter '=' 0> <NextId e.prefix>;
      };
}
```

Реализация

Лексическая структура

```
VAR ::= (s|t|e)\.(IDENT|NUMBER)
STR ::= C string: 'abc', "abc\"\n"
IDENT ::= [a-zA-Z][a-zA-Z0-9_-]*
NUMBER ::= [0-9]+
COMMENT ::= c comment /* */
{, }, <, >, (, ), =, ;, :, \,
```

Грамматика языка

```
Program -> Function*
Function -> Name Body
Name -> IDENT
Body -> { Sentence+ }
Sentence -> SentenceBody ;
SentenceBody -> Pattern SentenceBodyTail
SentenceBodyTail -> = Result | , Result : BlockTail
BlockTail -> Pattern SentenceBodyTail | Body
Pattern -> PatternElement*
PatternElement -> Variable | STR | NUMBER | IDENT | \( Pattern \)
Result -> (PatternElement | Call)*
Variable -> VAR
Call -> < Name Result >
```

Программная реализация

Реализован чистый функциноальный парсер с поддержкой восстановления из ошибок.

```
*$FROM LibraryEx
$EXTERN Map;

/* t.Token ::= (s.Domain t.Position t.Position e.Attr) */
/* s.Domain ::= s.WORD */
/* t.Position ::= (s.Line s.Column) */
```

```
/* s.Line ::= s.NUMBER */
/* s.Column ::= s.NUMBER */
/* e.Attr = e.ANY */

/* t.Error ::= (s.CHAR+) */

$ENTRY Println {
    (e.1) = <Prout e.1>;
}

$ENTRY ReadAll {
    /* empty */ = <ReadAll <Card>>;
    e.Text 0 = e.Text;
    e.Text = <ReadAll e.Text '\n' <Card>>;
}

/* <Tokenize e.Text> == (t.Token*) t.Error* */
$ENTRY Tokenize {
    e.Text = <DoTokenize e.Text () () (1 1)>;
}

DoTokenize {
    '' (e.Tokens) (e.Errors) t.Position = (e.Tokens) e.Errors;
    e.Text (e.Tokens) (e.Errors) t.Position
    , e.Text
    : {
        s.First e.Other
            , '\n\t '
            : e.1 s.First e.2
        = <DoTokenize e.Other (e.Tokens) (e.Errors) <Move s.First t.Position>>;
        s.P e.Value s.P e.Other
            , '\'\"'
            : e.1 s.P e.2
            , <EndsWith '\\' e.Value>
            : False
            = <DoTokenize e.Other
           (e.Tokens <MakeToken "STR" t.Position s.P e.Value s.P>)
            (e.Errors) <Move s.P e.Value s.P t.Position>>;
        s.First e.Other
            , '{}<>()=;:,'
            : e.1 s.First e.2
            = <DoTokenize e.Other
        (e.Tokens <MakeToken <GetSymbolDomain s.First> t.Position s.First>)
            (e.Errors) <Move s.First t.Position>>;
        e.Comment e.Other
            , e.Comment
```

```
            : '/''*' e.Inner '*''/'
            = <DoTokenize e.Other (e.Tokens) (e.Errors)
            <Move e.Comment t.Position>>;
        e.Var e.Other
            , <IsVar e.Var>
            : True
            , <StartsWithOrEmpty '\t\n {}<>()=\'\";:,' (e.Other)>
            : True
            = <DoTokenize e.Other
            (e.Tokens <MakeToken "VAR" t.Position e.Var>)
            (e.Errors) <Move e.Var t.Position>>;
        e.Ident e.Other
            , <IsIdent e.Ident>
            : True
            , <StartsWithOrEmpty '\t\n {}<>()=\'\";:,' (e.Other)>
            : True
            = <DoTokenize e.Other
            (e.Tokens <MakeToken "IDENT" t.Position e.Ident>)
            (e.Errors) <Move e.Ident t.Position>>;
        e.Number e.Other
            , <IsNumber e.Number>
            : True
            , <StartsWithOrEmpty '\t\n {}<>()=\'\";:,' (e.Other)>
            : True
            = <DoTokenize e.Other
            (e.Tokens <MakeToken "NUMBER" t.Position e.Number>)
            (e.Errors) <Move e.Number t.Position>>;
    };
}

EndsWith {
    s.EndsWith e.Text
        , e.Text
        : e.1 s.EndsWith
        = True;
    e.Else = False;
}

StartsWithOrEmpty {
    e.StartsWithOrEmpty (s.First e.Rest)
        , e.StartsWithOrEmpty
        : e.1 s.First e.2
        = True;
    e.StartsWithOrEmpty () = True;
    e.Else = False;
}
```

```
IsType {
    s.Symbol
        , 'set'
        : e.1 s.Symbol e.2
        = True;
    e.Other = False;
}

$ENTRY IsVar {
    s.Type '.' e.Rest
        , <IsType s.Type>
        : True
        , <IsIdent e.Rest> <IsDigit e.Rest>
        : e.1 True e.2
        = True;
    e.Other = False;
}

$ENTRY IsIdent {
    s.First e.Rest
        , <IsAlpha s.First>
        : True
        , <All IsIdentSymb e.Rest>
        : True
        = True;
    e.Else = False;
}

$ENTRY IsIdentSymb {
    s.Symb, <IsDigit s.Symb> : True = True;
    s.Symb, <IsAlpha s.Symb> : True = True;
    s.Symb, '_-' : e.1 s.Symb e.2 = True;
    e.Else = False;
}

All {
    s.Pred e.Elements
        , <Map s.Pred e.Elements> : e.1 False e.2 = False;
    e.Else = True;
}

$ENTRY IsNumber {
    e.Number, <All IsDigit e.Number> : True = True;
    e.Else = False;
}
```

```
$ENTRY IsDigit {
    s.Symb
    , <Ord s.Symb>
    : {
        s.N
            , <Compare s.N <- <Ord '0'> 1>>
            : '+'
            , <Compare s.N <+ <Ord '9'> 1>>
            : '-'
            = True;
        s.Else = False;
    };
    e.Else = False;
}

$ENTRY IsAlpha {
    s.Symb
    , <Ord s.Symb>
    : {
        s.N
            , <Compare s.N <- <Ord 'a'> 1>>
            : '+'
            , <Compare s.N <+ <Ord 'z'> 1>>
            : '-'
            = True;
        s.N
            , <Compare s.N <- <Ord 'A'> 1>>
            : '+'
            , <Compare s.N <+ <Ord 'Z'> 1>>
            : '-'
            = True;
        s.Else = False;
    };
    e.Else = False;
}

GetSymbolDomain {
    '{' = "LCB";
    '}' = "RCB";
    '<' = "LT";
    '>' = "GT";
    '(' = "LB";
    ')' = "RB";
    '=' = "EQ";
    ';' = "SEMICOLON";
```

```
    ':' = "COLON";
    ',' = "COMMA";
}

/* <MakeToken s.Domain t.Position e.Image> == t.Token */
MakeToken {
   s.Domain t.Position e.Image = (s.Domain t.Position <Move e.Image t.Position> e.Image);
}

$ENTRY TokenToString {
    (s.Domain t.Start t.End e.Attr)
   = (s.Domain '(' <PositionToString t.Start> '-' <PositionToString t.End> ')' ':' ' ' '"' e.Attr
}

PositionToString {
    (s.Line s.Column) = <Symb s.Line> ':' <Symb s.Column>;
}

Move {
    (s.Line s.Column) = (s.Line s.Column);
    '\n' e.Rest (s.Line s.Column) = <Move e.Rest (<+ 1 s.Line> 1)>;
   s.Symbol e.Rest (s.Line s.Column) = <Move e.Rest (s.Line <+ 1 s.Column>)>;
}
```

```
*$FROM LibraryEx
$EXTERN Map;

*$FROM lexer
$EXTERN Tokenize, ReadAll, TokenToString;

/* <N (e.Symbols) t.NTerm? e.Tokens t.ErrorList> */
/* == t.NTerm e.Tokens t.ErrorList */
/* e.Symbol ::= { t.Token | t.NTerm }* */
/* e.Tokens = t.Token* */
/* t.Token ::= (s.Tag t.Pos e.Info) */
/* t.NTerm ::= (s.Tag e.Info) */

/* Program -> Function* */
/* <Program (e.Children) e.Tokens t.ErrorList> == t.Child e.Tokens t.ErrorList */

/* t.Node ::= (s.NT s.Child*) */
/* s.Child ::= t.Node | s.Token */


/* Program -> Function* */
Program {
```

```
    (e.Children) t.ErrorList = (Program e.Children) t.ErrorList;
    (e.Children) e.Tokens t.ErrorList
    , <Function () e.Tokens t.ErrorList>
    : {
      t.Child e.NewTokens t.NewErrorList = <Program (e.Children t.Child) e.NewTokens t.NewErrorL
    }
}

/* Function -> Name Body */
Function {
    () (IDENT e.Rest) e.Tokens t.ErrorList
    = <Function () <Name () (IDENT e.Rest) e.Tokens t.ErrorList>>;
    () (Name e.Rest) e.Tokens t.ErrorList
    = <Function ((Name e.Rest)) e.Tokens t.ErrorList>;
    () t.Other e.Tokens (e.Errors)
   = <Function () e.Tokens (e.Errors (Error "expected IDENT, got" t.Other))>;
   () (e.Errors) = (Function) (e.Errors (Error "expected IDENT, got EOF"));
   ((Name e.1)) (Body e.2) e.Tokens t.ErrorList = (Function (Name e.1) (Body e.2)) e.Tokens t.Err
   ((Name e.Rest)) e.Tokens t.ErrorList = <Function ((Name e.Rest)) <Body () e.Tokens t.ErrorLis
   t.Else e.Tokens t.ErrorList = (Function t.Else) e.Tokens t.ErrorList;
}

/* Name -> IDENT */
Name {
   () (IDENT e.Rest) e.Tokens t.ErrorList = (Name (IDENT e.Rest)) e.Tokens t.ErrorList;
   t.Else e.Tokens t.ErrorList = (Name t.Else) e.Tokens t.ErrorList;
}

/* Body -> { Sentence+ } */
Body {
   () (LCB e.Rest) e.Tokens t.ErrorList = <Body ((LCB e.Rest)) e.Tokens t.ErrorList>;
   () t.Other e.Tokens (e.Errors) = <Body () e.Tokens (e.Errors (Error "expected LCB, got" t.Othe
   () (e.Errors) = (Body ()) (e.Errors (Error "expected LCB, got EOF"));
   (e.1) (Sentence e.Rest) e.Tokens t.ErrorList = <Body (e.1 (Sentence e.Rest)) e.Tokens t.Error
   ((LCB e.Rest)) e.Tokens t.ErrorList = <Body ((LCB e.Rest)) <Sentence () e.Tokens t.ErrorList>
    ((LCB e.1) e.2 (Sentence e.3)) (RCB e.Rest) e.Tokens t.ErrorList
    = (Body e.2 (Sentence e.3)) e.Tokens t.ErrorList;
    (e.1 (Sentence e.2)) e.Tokens t.ErrorList
   = <Body (e.1 (Sentence e.2)) <Sentence () e.Tokens t.ErrorList>>;
   t.Else e.Tokens t.ErrorList = (Body t.Else) e.Tokens t.ErrorList;
}

/* Sentence -> SentenceBody ; */
Sentence {
    () (SentenceBody e.Rest) e.Tokens t.ErrorList
    = <Sentence ((SentenceBody e.Rest)) e.Tokens t.ErrorList>;
```

```
   () e.Tokens t.ErrorList = <Sentence () <SentenceBody () e.Tokens t.ErrorList>>;
    ((SentenceBody e.1)) (SEMICOLON e.2) e.Tokens t.ErrorList
    = (Sentence e.1) e.Tokens t.ErrorList;
    ((SentenceBody e.1)) t.Other e.Tokens (e.Errors)
   = <Sentence ((SentenceBody e.1)) e.Tokens (e.Errors (Error "expected SEMICOLON, got " t.Other
    ((SentenceBody e.1)) (e.Errors)
   = (Sentence ((SentenceBody e.1))) (e.Errors (Error "expected SEMICOLON, got EOF"));
   t.Else e.Tokens t.ErrorList = (Sentence t.Else) e.Tokens t.ErrorList;
}

/* SentenceBody -> Pattern SentenceBodyTail */
SentenceBody {
   () (Pattern e.1) e.Tokens t.ErrorList = <SentenceBody ((Pattern e.1)) e.Tokens t.ErrorList>;
   () e.Tokens t.ErrorList = <SentenceBody () <Pattern () e.Tokens t.ErrorList>>;
   ((Pattern e.1)) (SentenceBodyTail e.2) e.Tokens t.ErrorList = (SentenceBody e.2) e.Tokens t.E
    ((Pattern e.1)) e.Tokens t.ErrorList
   = <SentenceBody ((Pattern e.1)) <SentenceBodyTail () e.Tokens t.ErrorList>>;
   t.Else e.Tokens t.ErrorList = (SentenceBody t.Else) e.Tokens t.ErrorList;
}

/* SentenceBodyTail -> = Result | , Result : BlockTail */
SentenceBodyTail {
    () (EQ e.Rest) e.Tokens t.ErrorList
    = <SentenceBodyTail ((EQ e.Rest)) e.Tokens t.ErrorList>;
    ((EQ e.1)) (Result e.2) e.Tokens t.ErrorList
   = (SentenceBodyTail (EQ e.1) (Result e.2)) e.Tokens t.ErrorList;
    ((EQ e.Rest)) e.Tokens t.ErrorList
   = <SentenceBodyTail ((EQ e.Rest)) <Result () e.Tokens t.ErrorList>>;
    () (COMMA e.Rest) e.Tokens t.ErrorList
   = <SentenceBodyTail ((COMMA e.Rest)) <Result () e.Tokens t.ErrorList>>;
    () t.Other e.Tokens (e.Errors) =
   <SentenceBodyTail () e.Tokens (e.Errors (Error "expected COMMA or EQ, got " t.Other))>;
   () (e.Errors) = (SentenceBodyTail ()) (e.Errors (Error "expected COMMA or EQ, got EOF"));
    ((COMMA e.1)) (Result e.2) e.Tokens t.ErrorList
   = <SentenceBodyTail ((COMMA e.1) (Result e.2)) e.Tokens t.ErrorList>;
    ((COMMA e.Rest)) e.Tokens t.ErrorList
   = <SentenceBodyTail ((COMMA e.Rest)) <Result () e.Tokens t.ErrorList>>;
    ((COMMA e.1) (Result e.2)) (COLON e.3) e.Tokens t.ErrorList
   = <SentenceBodyTail ((COMMA e.1) (Result e.2) (COLON e.3)) e.Tokens t.ErrorList>;
    ((COMMA e.1) (Result e.2)) t.Other e.Tokens (e.Errors)
    = <SentenceBodyTail ((COMMA e.1) (Result e.2))
    e.Tokens (e.Errors (Error "expected COLON, got" t.Other))>;
    ((COMMA e.1) (Result e.2)) (e.Errors)
   = (SentenceBodyTail ((COMMA e.1) (Result e.2))) (e.Errors (Error "expected COLON, got EOF"));
   ((COMMA e.1) (Result e.2) (COLON e.3)) (BlockTail e.4) e.Tokens t.ErrorList
   = (SentenceBodyTail (COMMA e.1) (Result e.2) (COLON e.3) (BlockTail e.4)) e.Tokens t.ErrorLis
```

```
    ((COMMA e.1) (Result e.2) (COLON e.3)) e.Tokens t.ErrorList
   = <SentenceBodyTail ((COMMA e.1) (Result e.2) (COLON e.3)) <BlockTail () e.Tokens t.ErrorList
   t.Else e.Tokens t.ErrorList = (SentenceBodyTail t.Else) e.Tokens t.ErrorList;
}

/* Call -> < Name Result > */
Call {
    ((LT e.1)) (IDENT e.2) e.Tokens t.ErrorList
    = <Call ((LT e.1) (Name (IDENT e.2))) e.Tokens t.ErrorList>;
    ((LT e.1)) t.Other e.Tokens (e.Errors)
   = <Call ((LT e.1)) e.Tokens (e.Errors (Error "expected IDENT, got " t.Other))>;
   ((LT e.1)) (e.Errors) = (Call ((LT e.1))) (e.Errors (Error "expected IDENT, got EOF"));
    ((LT e.1) (Name e.2) (Result e.3)) (GT e.4) e.Tokens t.ErrorList
    = (Call (Name e.2) (Result e.3)) e.Tokens t.ErrorList;
    ((LT e.1) (Name e.2)) (Result e.3) e.Tokens t.ErrorList
   = <Call ((LT e.1) (Name e.2) (Result e.3)) e.Tokens t.ErrorList>;
    ((LT e.1) (Name e.2)) e.Tokens t.ErrorList
   = <Call ((LT e.1) (Name e.2)) <Result () e.Tokens t.ErrorList>>;
   t.Else e.Tokens t.ErrorList = (Call t.Else) e.Tokens t.ErrorList;
}

/* BlockTail -> Pattern SentenceBodyTail | Body */
BlockTail {
   () (Pattern e.Rest) e.Tokens t.ErrorList = <BlockTail ((Pattern e.Rest)) e.Tokens t.ErrorList
   () (s.Domain e.Rest) e.Tokens t.ErrorList, <IsPatternElement s.Domain> : True
   = <BlockTail () <Pattern () (s.Domain e.Rest) e.Tokens t.ErrorList>>;
   () (LCB e.Rest) e.Tokens t.ErrorList = <BlockTail () <Body ((LCB e.Rest)) e.Tokens t.ErrorList
   () (Body e.Rest) e.Tokens t.ErrorList = <BlockTail ((Body e.Rest)) e.Tokens t.ErrorList>;
    () t.Other e.Tokens (e.Errors)
   = <BlockTail () e.Tokens (e.Errors (Error "expected block or pattern, got " t.Other))>;
   () (e.Errors) = (BlockTail ()) (e.Errors (Error "expected block or pattern, got EOF"));
   ((Body e.Rest)) e.Tokens t.ErrorList = (BlockTail (Body e.Rest)) e.Tokens t.ErrorList;
    ((Pattern e.Rest)) (SentenceBodyTail e.2) e.Tokens t.ErrorList
   = <BlockTail ((Pattern e.Rest) (SentenceBodyTail e.2)) e.Tokens t.ErrorList>;
    ((Pattern e.Rest)) e.Tokens t.ErrorList
   = <BlockTail ((Pattern e.Rest)) <SentenceBodyTail () e.Tokens t.ErrorList>>;
    ((Pattern e.Rest) (SentenceBodyTail e.2)) e.Tokens t.ErrorList
   = (BlockTail (Pattern e.Rest) (SentenceBodyTail e.2)) e.Tokens t.ErrorList;
   t.Else e.Tokens t.ErrorList = (BlockTail t.Else) e.Tokens t.ErrorList;
}

/* Pattern -> PatternElement* */
Pattern {
    (e.Contents) (PatternElement e.Rest) e.Tokens t.ErrorList
   = <Pattern (e.Contents (PatternElement e.Rest)) e.Tokens t.ErrorList>;
    (e.Contents) (LB e.Rest) e.Tokens t.ErrorList
```

```
   = <Pattern (e.Contents) <PatternElement () (LB e.Rest) e.Tokens t.ErrorList>>;
    (e.Contents) (s.Domain e.Rest) e.Tokens t.ErrorList,
    <IsPatternElement s.Domain> : True = <Pattern (e.Contents)
    <PatternElement () (s.Domain e.Rest) e.Tokens t.ErrorList>>;
   (e.Contents) e.Tokens t.ErrorList = (Pattern e.Contents) e.Tokens t.ErrorList;
   t.Else e.Tokens t.ErrorList = (Pattern t.Else) e.Tokens t.ErrorList;
}

/* PatternElement -> Variable | STR | NUMBER | IDENT | \( Pattern \) */
PatternElement {
    () (LB e.Rest) e.Tokens t.ErrorList
    = <PatternElement ((LB e.Rest)) e.Tokens t.ErrorList>;
    ((LB e.1)) (Pattern e.2) e.Tokens t.ErrorList
   = <PatternElement ((LB e.1) (Pattern e.2)) e.Tokens t.ErrorList>;
    ((LB e.Rest)) e.Tokens t.ErrorList
   = <PatternElement ((LB e.Rest)) <Pattern () e.Tokens t.ErrorList>>;
    ((LB e.1) (Pattern e.2)) (RB e.3) e.Tokens t.ErrorList
   = (PatternElement (LB e.1) (Pattern e.2) (RB e.3)) e.Tokens t.ErrorList;
    () (s.Domain e.Rest) e.Tokens t.ErrorList,
    <IsPatternElement s.Domain> : True
    = (PatternElement (s.Domain e.Rest)) e.Tokens t.ErrorList;
   t.Else e.Tokens t.ErrorList = (PatternElement t.Else) e.Tokens t.ErrorList;
}

IsPatternElement {
    IDENT = True;
    STR = True;
    NUMBER = True;
    LB = True;
    VAR = True;
    e.Else = False;
}

/* Result -> (PatternElement | Call)* */
Result {
    (e.Contents) (PatternElement e.Rest) e.Tokens t.ErrorList
   = <Result (e.Contents (PatternElement e.Rest)) e.Tokens t.ErrorList>;
    (e.Contents) (s.Domain e.Rest) e.Tokens t.ErrorList,
    <IsPatternElement s.Domain> : True =
   <Result (e.Contents) <PatternElement () (s.Domain e.Rest) e.Tokens t.ErrorList>>;
    (e.Contents) (Call e.Rest) e.Tokens t.ErrorList
    = <Result (e.Contents (Call e.Rest)) e.Tokens t.ErrorList>;
    (e.Contents) (LT e.Rest) e.Tokens t.ErrorList
   = <Result (e.Contents) <Call ((LT e.Rest)) e.Tokens t.ErrorList>>;
   (e.Contents) e.Tokens t.ErrorList = (Result e.Contents) e.Tokens t.ErrorList;
   t.Else e.Tokens t.ErrorList = (Result t.Else) e.Tokens t.ErrorList;
```

```
}

Parse {
    e.Tokens
    , <Program () e.Tokens ()>
    : {
      t.Program t.ErrorList = <Prout t.Program> <Prout> <Prout t.ErrorList>;
    }
}

$ENTRY Go {
    /* empty */
    , <Tokenize <ReadAll>>
    : {
        (e.Tokens) = <Parse e.Tokens>;
        (e.Tokens) e.Errors = <Prout error>;
    };
}
```

Тестирование

Входные данные

```
/* Декартово произведение */
CartProd {
  (t.X e.X) (e.Y) = <CartProd-Bind t.X e.Y> <CartProd (e.X) (e.Y)>;
  (/* пусто */) (e.Y) = /* пусто */;
}

CartProd-Bind {
  t.X t.Y e.Y = (t.X t.Y) <CartProd-Bind t.X e.Y>;
  t.X /* пусто */ = /* пусто */;
}

/*
  Прибавление 1 к строковой записи числа:
  123 → 124, 99 → 100, 007 → 008
*/
Inc {
 e.Prefix s.Last, '0123456789' : e.1 s.Last s.Next e.2 = e.Prefix s.Next;
  e.Prefix '9' = <Inc e.Prefix> '0';
  /* пусто */ = '1';
}

/* Функция возвращает уникальные идентификаторы с заданным префиксом */
```

```
NextId {
  e.prefix
    , <Dg counter>
    : {
        s.n = e.prefix <Symb s.n> <Br counter '=' <Add 1 s.n>>;
        /* пусто */ = <Br counter '=' 0> <NextId e.prefix>;
      };
}
```

Вывод на stdout

```
(Program (Function (Name (IDENT (2 1 )(2 9 )CartProd))(Body (Sentence (EQ (3 19 )(3 20 )=)(Result
(Name (IDENT (3 22 )(3 35 )CartProd-Bind))(Result (PatternElement (VAR (3 36 )(3 39 )t.X))(Patter
(VAR (3 40 )(3 43 )e.Y))))(Call (Name (IDENT (3 46 )(3 54 )CartProd))(Result (PatternElement (LB
(3 56 )()(Pattern (PatternElement (VAR (3 56 )(3 59 )e.X)))(RB (3 59 )(3 60 ))))(PatternElement (
(3 62 )()(Pattern (PatternElement (VAR (3 62 )(3 65 )e.Y)))(RB (3 65 )(3 66 ))))))))(Sentence (EQ
(4 29 )=)(Result ))))(Function (Name (IDENT (7 1 )(7 14 )CartProd-
Bind))(Body (Sentence (EQ (8 15 )(8 16 )=)
(Result (PatternElement (LB (8 17 )(8 18 )()(Pattern (PatternElement (VAR (8 18 )(8 21 )t.X))
(PatternElement (VAR (8 22 )(8 25 )t.Y)))(RB (8 25 )(8 26 ))))(Call (Name (IDENT (8 28 )(8 41 )
CartProd-Bind))(Result (PatternElement (VAR (8 42 )(8 45 )t.X))(PatternElement (VAR (8 46 )(8 49
(Sentence (EQ (9 24 )(9 25 )=)(Result ))))(Function (Name (IDENT (16 1 )(16 4 )Inc))(Body (Senten
(17 18 )(17 19 ),)(Result (PatternElement (STR (17 20 )(17 32 )'0123456789')))(COLON (17 33 )(17
(BlockTail (Pattern (PatternElement (VAR (17 35 )(17 38 )e.1))(PatternElement (VAR (17 39 )(17 45
(PatternElement (VAR (17 46 )(17 52 )s.Next))(PatternElement (VAR (17 53 )(17 56 )e.2)))(Sentenc
(EQ (17 57 )(17 58 )=)(Result (PatternElement (VAR (17 59 )(17 67 )e.Prefix))(PatternElement (VAR
(17 74 )s.Next))))))(Sentence (EQ (18 16 )(18 17 )=)(Result (Call (Name (IDENT (18 19 )(18 22 )Inc
(PatternElement (VAR (18 23 )(18 31 )e.Prefix))))(PatternElement (STR (18 33 )(18 36 )'0'))))(Se
(19 20 )(19 21 )=)(Result (PatternElement (STR (19 22 )(19 25 )'1'))))))(Function (Name (IDENT (2
NextId))(Body (Sentence (COMMA (25 5 )(25 6 ),)(Result (Call (Name (IDENT (25 8 )(25 10 )Dg))(Res
(PatternElement (IDENT (25 11 )(25 18 )counter)))))(COLON (26 5 )(26 6 ):)(BlockTail (Body (Sente
(27 13 )(27 14 )=)(Result (PatternElement (VAR (27 15 )(27 23 )e.prefix))(Call (Name (IDENT (27 2
Symb))(Result (PatternElement (VAR (27 30 )(27 33 )s.n))))(Call (Name (IDENT (27 36 )(27 38 )Br))
(PatternElement (IDENT (27 39 )(27 46 )counter))(PatternElement (STR (27 47 )(27 50 )'='))(Call (
(27 52 )(27 55 )Add))(Result (PatternElement (NUMBER (27 56 )(27 57 )1))(PatternElement (VAR (27
s.n))))))))(Sentence (EQ (28 26 )(28 27 )=)(Result (Call (Name (IDENT (28 29 )(28 31 )Br))(Result
(PatternElement (IDENT (28 32 )(28 39 )counter))(PatternElement (STR (28 40 )(28 43 )'='))(Patte
(NUMBER (28 44 )(28 45 )0))))(Call (Name (IDENT (28 48 )(28 54 )NextId))(Result (PatternElement
(VAR (28 55 )(28 63 )e.prefix)))))))))))

()
```

Вывод

В рамках данной работы я изучил алгоритмы построения парсеров методом рекурсивного спуска, реализовал чистый функциональный парсер с поддержкой

восстановления из ошибок.