Предмет изучения: системы программирования

- Основные определения
- Программный продукт и его жизненный цикл
- Основные компоненты систем программирования
- Трансляторы: компиляторы и интерпретаторы
- Языки программирования и средства их формального описания
- Объектно-ориентированный подход к проектированию программных продуктов
- Язык программирования Си++

Правила грамматики М-языка

P	→	***program*** D_1; *B*⊥
D_1	→	***var*** *D* {, *D*}
D	→	*I* {, *I*}: [***int*** | ***bool***]
B	→	***begin*** *S* {; *S*} ***end***
S	→	*I* := *E* | ***if*** *E* ***then*** *S* ***else*** *S*
		while *E* ***do*** *S* | *B* | ***read*** (*I*) | ***write*** (*E*)
E	→	E_1 | E_1 [= | < | > | <= | >= | !=] E_1
E_1	→	*T* {[+ | - | ***or***] *T*}
T	→	*F* {[* | / | ***and***] *F*}
F	→	*I* | *N* | *L* | ***not*** *F* | (*E*)
L	→	***true*** | ***false***
I	→	*C* | *IC* | *IR*
N	→	*R* | *NR*
C	→	*a* | *b* | … | *z* | *A* | *B* | … | *Z*
R	→	*0* | *1* | *2* | … | *9*

Правила модельного языка

- Запись вида $\{\alpha\}$ означает итерацию цепочки α (повторение её 0 или более раз): в порождаемой цепочке в этом месте может находиться либо ε, либо α, либо $\alpha\alpha$, либо $\alpha\alpha\alpha$, и так далее
- Запись вида $[\alpha|\beta]$ означает, что в порождаемой цепочке этом месте может находиться либо α, либо β
- P – цель грамматики
- Символ $\perp$ – маркер конца текста программы

Контекстные условия в М-языке

1. Любое имя, используемое в программе, должно быть описано, причём только один раз
2. В операторе присваивания типы переменной и выражения должны совпадать
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение
4. Операнды операций отношения должны быть целочисленными
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам (нельзя складывать целочисленные и логические значения); старшинство операций задано синтаксисом

Примечания в М-языке

- В любом месте программы, кроме идентификаторов, служебных слов и чисел, может находиться произвольное число пробелов и примечаний (комментариев) вида {< любые символы, кроме символов } и ⊥ >}
- Вложенных комментариев в модельном языке нет

Лексемы модельного языка

- ***Идентификаторы***: первым символом любого имени всегда является строчная или прописная латинская буква (строчные и прописные буквы различаются), следующими символами в именах могут быть любые буквы и любые десятичные цифры
- ***Целые числа по основанию 10***. В языке выбрана десятичная система счисления
- ***Односимвольные и двухсимвольные знаки операций***
- ***Служебные (ключевые) слова***

План работы анализатора языка

1. ***Ввод очередного набора символов (“лексемы”)***
   Символы текста программы “от пробела до пробела”набираются в буфер

2. ***Поиск по таблицам лексем***
   Поиск должен вестись сравнением содержимого набранного буфера с текстовым представлением стандартных правильных лексем

3. ***Успешный поиск в таблице означает обнаружение в тексте правильной лексемы***
   Правильными лексемами также считаются любые имена и десятичные константы

Действия в грамматике М-языка

GetS – ввод очередного символа исходной программы

clear – инициализация ввода символов лексемы

add – добавление символа к буферу лексем

get_token – поиск лексем в таблицах служебных слов (*TW*), ограничителей и знаков операций (*TD*)

get_object – поиск в таблице имён (*TI*) или констант (*TC*)

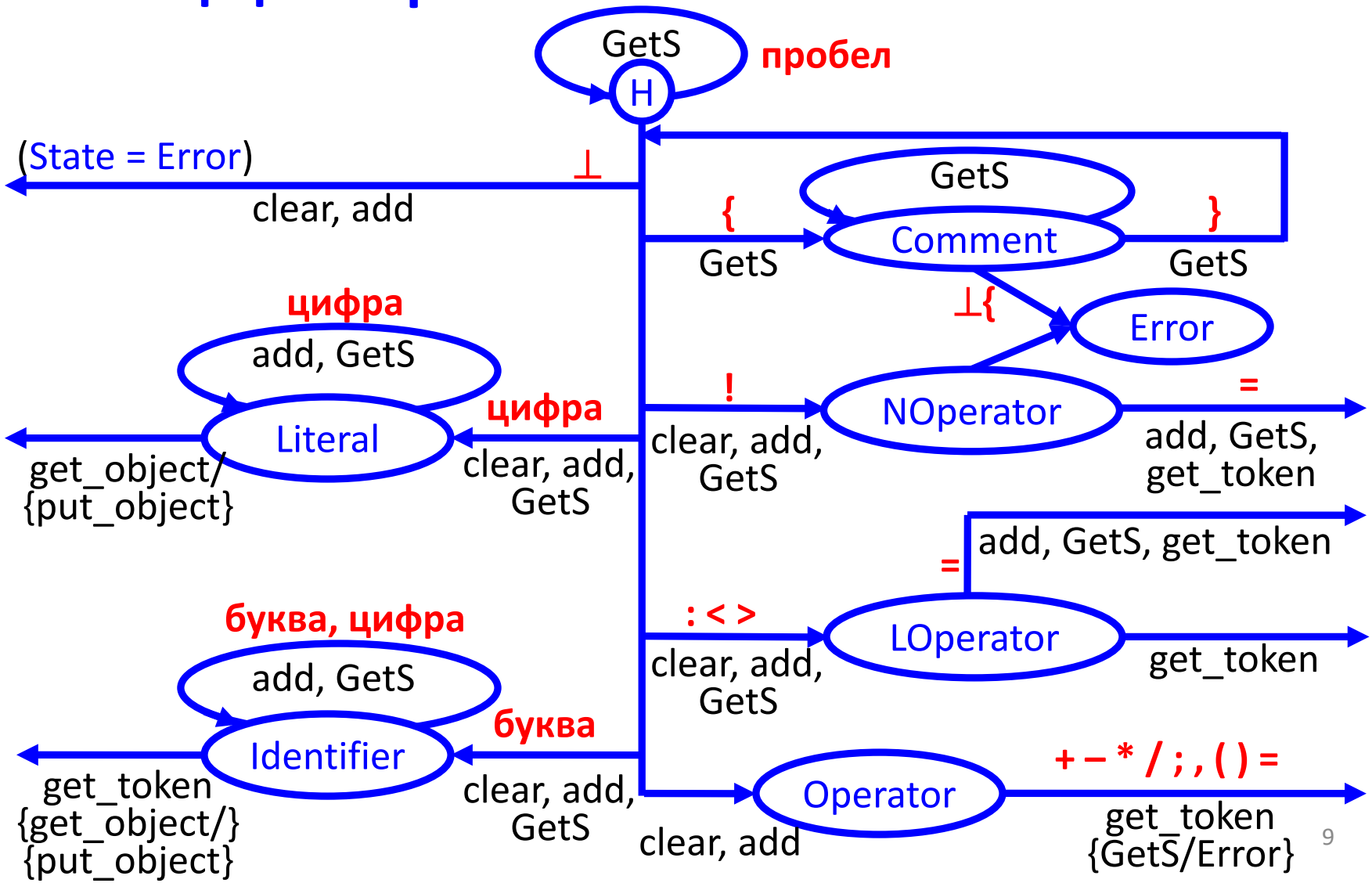
Ident – создание нового объекта “*Идентификатор*”

Number – создание нового объекта “*Константа*”

put_obj – занесение информации о вновь созданных объектах в таблицы констант (*TC*) или имён (*TI*)

Token – создание новой лексемы при вводе константы или идентификатора (имени)

Диаграмма состояний



Автомат анализатора

```
class Scanner { enum State {H,    Comment,   Noperator,   Operator,    LOperator,
                                  Identifier, Literal,    Error };
                State FA_State;   FILE * fp;  char c;     buf b;       /* … */
Token Scanner:: get_lex () { Token * res;
    for (;;)  {  switch (FA_State){                                    /* … */
                 case H:     if (isspace (c)                           GetS ();
                             else if (isalpha (c)) {b.clear (); b.add (); GetS (); FA_State = Identifier;     }
                             else if (isdigit (c)) {b.clear (); b.add (); GetS (); FA_State = Literal;        }
                             else if (c== '{' )    {                      GetS (); FA_State = Comment;      }
                             else if (c== ':' || c== '<' || c== '>')
                                                   {b.clear (); b.add (); GetS (); FA_State = LOperator;     }
                             else if (c == '⊥')    {                              FA_State = Error;
                                                    b.clear (); b.add ();         return TD.get_token (b);  }
                             else if (c == '!')    { b.clear (); b.add (); GetS (); FA_State = NOperator;    }
                             else                  {b.clear (); b.add ();         FA_State = Operator;      }
                             break;
      /* { */    case Comment:
                          if (c == '}')    {                          GetS (); FA_State = H;                }
                             else if (c == '⊥')    {                           FA_State = Error; throw c;}
                             else if (c == '{')    {                                             throw c;}
                             else                                     GetS ();
                             break;
/* … */ };
```

Автомат анализатора

```
class Scanner { enum State {H,    Comment,    Noperator,    Operator,    LOperator,
                                  Identifier, Literal,      Error };
                State FA_State;   FILE * fp;  char c;       buf b;       /* … */
Token Scanner:: get_lex () { Token * res;
    for (;;)  {  switch (FA_State){                                      /* … */
      /* : < > */  case LOperator:                                      FA_State = H;
                        if (c == '=')   {        b.add (); GetS ();                  }
                                                                        return TD.get_token (b);
                    break;
      /* ! */      case NOperator:                                      FA_State = H;
                        if (c == '=')   {        b.add (); GetS ();  return TD.get_token (b);  }
                    else                                                          throw '!';
                    break;
                 case Operator:                                         FA_State = H;
                        if ((res = TD.get_token (b)) != 0) {  GetS ();  return res;          }
                    else                                                          throw c;
                    break;
                 case Error:                                                      throw c;
                    break;
```

/* … */ };

Автомат анализатора

```
class Scanner { enum State {H,     Comment,    Noperator,    Operator,     LOperator,
                                   Identifier, Literal,      Error };
                State FA_State;    FILE * fp;  char c;       buf b;        /* … */
Token Scanner:: get_lex () { Token * res;
    for (;;)  {  switch (FA_State){                                        /* … */
                 case Identifier: if (isalnum(c)){      b.add (); GetS ();          }
                       else                     {      FA_State = H;
                                                  if ((res = TW.get_token (b)) !=0 ) return res;
                                                  return new Token (LEX_ID,  CreateIdentObject (b));
                                                }
                       break;
                 case Literal:    if (isdigit (c)) {    b.add (); GetS ();          }
                       else                     {      FA_State = H;
                                                  return new Token(LEX_NUM,CreateNumberObject(b));
                                                }
                       break;
              } // end switch (FA_State)
    } // end for
};
```

Обработка ошибок при анализе

```
    Parser::  Parser   (char * program): scan (program) { }
    Parser::~Parser   ()                                 { }
void Parser:: Analyze  ()  { GetL (); P (); }
int   main (int argc, char ** argv) { /* ... */
        try { Parser *  M = new Parser (“program.txt”);
                        M -> Analyze ();
            }
        catch (char c)
            { cout << “Неверный символ при лексическом”
                   “ анализе: ” << c << endl;
              /* ... */
            } /* ... */
    }
```

Класс описания лексем

```
class Token {  type_of_lex type;          // тип лексемы
               ProgramObject * value;     // значение лексемы:
                                          // указатель на объект
      // Конструктор лексем:
public: Token (const type_of_lex t, ProgramObject * v = 0);
      // Выдать тип лексемы по запросу из-вне
        type_of_lex   get_type  () const;
      // Выдать указатель на объект лексемы
        ProgramObject * get_value () const;
      // Установить новый тип лексемы
        void   set_type  (const type_of_lex t);
      // Связать лексему с объектом
        void   set_value (ProgramObject * v);
};
```

Типы лексем

```
enum type_of_lex { LEX_NULL,
    LEX_PROGRAM, LEX_VAR,    LEX_BOOL,   LEX_INT,
    LEX_FALSE,   LEX_TRUE,   LEX_BEGIN,  LEX_END,
    LEX_ASSIGN,  LEX_IF,     LEX_THEN,   LEX_ELSE,
    LEX_WHILE,   LEX_DO,     LEX_READ,   LEX_WRITE,
    LEX_AND,     LEX_NOT,    LEX_OR,     LEX_LT,
    LEX_LE,      LEX_EQ,     LEX_NE,     LEX_GE,
    LEX_GT,      LEX_DIV,    LEX_PLUS,   LEX_MINUS,
    LEX_MULT,    LEX_LPAREN, LEX_RPAREN,
    LEX_COMMA,   LEX_COLON,  LEX_SEMICOLON,
    LEX_ID,      LEX_NUM,    LEX_FIN
};  // Сведения о лексемах берутся из правил языка
```

Лексемы, определяемые языком

- Таблицы служебных слов языка и знаков операций и ограничителей:

```
static char * KeyWords [] = {
"",      "program", "begin", "end",   "var",  "int",
"bool",  "true",    "false", "if",    "then", "else",
"do",    "while",   "read",  "write", "not",  "or",
                    "and",   0};

static char * Delimiters [] = {
"",      "⊥",       ",",     ":",     ":=",   ";",
"<",     "<=",      "=",     "!=",    ">=",   ">",
"+",     "-",       "*",     "/",     "(",    ")",
                              0};
```

Лексемы, определяемые языком

- Таблицы типов лексем ***LKeyWords*** и ***LDelimiter***

```
static type_of_lex   LKeyWords  [] = {
    LEX_NULL,    LEX_PROGRAM,   LEX_BEGIN,    LEX_END,
    LEX_VAR,     LEX_INT,       LEX_BOOL,     LEX_TRUE,
    LEX_FALSE,   LEX_IF,        LEX_THEN,     LEX_ELSE,
    LEX_DO,      LEX_WHILE,     LEX_READ,     LEX_WRITE,
    LEX_NOT,     LEX_OR,        LEX_AND,      LEX_NULL};

static type_of_lex   Ldelimiters  [] = {
    LEX_NULL,    LEX_FIN,       LEX_COMMA,    LEX_COLON,
    LEX_ASSIGN,  LEX_SEMICOLON, LEX_LT,       LEX_LE,
    LEX_EQ,      LEX_NE,        LEX_GE,       LEX_GT,
    LEX_PLUS,    LEX_MINUS,     LEX_MULT,     LEX_DIV,
    LEX_LPAREN,  LEX_RPAREN,    LEX_NULL};
```

Класс таблиц лексем

```
class     TokenTable { Token ** p;  char ** c; int size;
public:  TokenTable (int max_size,  char  * data [],
                                    type_of_lex t []);
        ~TokenTable ();             // деструктор таблицы
     int  get_size     () const;    // доступ к размеру таблицы
    void  put_obj      (ProgramObject * t, int i);
 Token *get_token      (const buf & b) const;
 /* ... */
};
```

```
TokenTable TW (sizeof( KeyWords)/sizeof ( KeyWords[0]),
                      KeyWords, LKeyWords);

TokenTable TD  (sizeof( Delimiters)/sizeof(Delimiters[0]),
                      Delimiters, LDelimiters);
```

Буфер для сборки лексем

```
class buf { char * b;  // указатель на буфер ввода лексем
      int  size;       // размер буфера ввода лексем
      int  top;        // текущая позиция для ввода в буфер
public:  buf           (int max_size = 260)   // конструктор буфера
                          { b = new char [size = max_size]; clear (); }
        ~buf           ()  { delete b; }            // деструктор буфера
    void  clear        ()  { memset (b, '\0', size); top = 0; }
    void  add          (const char c)   { b [top ++] = c; }
 char   *get_string    () const;   // выдать представление лексемы
};

Token * TokenTable::get_token (const buf & b) const
{ Token ** q = p; char ** s = c; Token * t;
  while (*q) { t = *q ++; if (! strcmp (b.get_string (), * s ++)) return t; }
  return 0;
}
```

Класс программных объектов

- Лексический анализатор работает с объектами трёх видов:
  - Имена, введённые программистом (***Ident***)
  - Константы, введённые программистом (***Number***)
  - Операции, определённые в языке (***Operation***)
- В программе создаётся базовый класс объектов и система производных классов:

```
class ProgramObject { protected: type_of_lex type; int value;
    public:  type_of_lex  get_type      () const;
             void         set_type      (type_of_lex t);
             int          get_value     () const;
             void         set_value     (int v);
    virtual  bool         is_object     (const buf & b) const = 0;
};           class Ident:        public ProgramObject { /* ... */ };
             class Number:       public ProgramObject { /* ... */ };
             class Operation:    public ProgramObject { /* ... */ };
```

Операционные объекты

```
ObjectTable<Operation> TO (20);          // Таблица для операционных объектов
int size = TW.get_size  () - 1;   for (int i = 0; i < size; i ++)
{  Token * T = TW (i); char * Ep = TW [i]; type_of_lex tp = T -> get_type ();
   switch (tp)
     {  case LEX_AND:      TW.put_obj (TO.put_obj (new  AndObject     (Ep, tp)), i ); break;
        case LEX_WRITE:    TW.put_obj (TO.put_obj (new  WriteObject   (Ep, tp)), i ); break;
         /*   LEX_TRUE     LEX_FALSE       LEX_NOT      LEX_OR        LEX_READ */
     }
}
int size = TD.get_size () - 1;     for (i = 0; i < size; i ++)
{  Token * T = TD (i); char * Ep = TD [i]; type_of_lex tp = T -> get_type ();
   switch (tp)
     {  case LEX_ASSIGN:  TD.put_obj  (TO.put_obj  (new AssignObject  (Ep, tp)), i); break;
        case LEX_LT:      TD.put_obj  (TO.put_obj  (new LtObject      (Ep, tp)), i); break;
         /*   LEX_EQ       LEX_GT          LEX_LE       LEX_GE        LEX_NE       */
         /*   LEX_PLUS     LEX_MINUS       LEX_MULT     LEX_DIV                    */
     }
}          class TokenTable {   Token  ** p; char ** c; int size; // дополнительные методы
           /* … */                Token  * TokenTable:: operator () (int k) { return p [k]; }
                                  char   * TokenTable:: operator [] (int k) { return c [k]; }  };
```

Классы операционных объектов

```
class     TrueObject :public Operation { public:     TrueObject (char *s, type_of_lex t); };
class    FalseObject :public Operation { public:    FalseObject (char *s, type_of_lex t); };
class      NotObject :public Operation { public:      NotObject (char *s, type_of_lex t); };
class       OrObject :public Operation { public:       OrObject (char *s, type_of_lex t); };
class      AndObject :public Operation { public:      AndObject (char *s, type_of_lex t); };
class       EqObject :public Operation { public:       EqObject (char *s, type_of_lex t); };
class       LtObject :public Operation { public:       LtObject (char *s, type_of_lex t); };
class       GtObject :public Operation { public:       GtObject (char *s, type_of_lex t); };
class       LeObject :public Operation { public:       LeObject (char *s, type_of_lex t); };
class       GeObject :public Operation { public:       GeObject (char *s, type_of_lex t); };
class       NeObject :public Operation { public:       NeObject (char *s, type_of_lex t); };
class     PlusObject :public Operation { public:     PlusObject (char *s, type_of_lex t); };
class    MinusObject :public Operation { public:    MinusObject (char *s, type_of_lex t); };
class     MultObject :public Operation { public:     MultObject (char *s, type_of_lex t); };
class      DivObject :public Operation { public:      DivObject (char *s, type_of_lex t); };
class   AssignObject :public Operation { public:   AssignObject (char *s, type_of_lex t); };
class    WriteObject :public Operation { public:    WriteObject (char *s, type_of_lex t); };
class     ReadObject :public Operation { public:     ReadObject (char *s, type_of_lex t); };
```

Таблицы программных объектов

- Таблица любых программных объектов (идентификаторов, констант и операций) может иметь такие элементы данных:
  - внешнее представление текущего объекта-идентификатора или операции (буквы, цифры, знаки операций), который надо найти в таблице или записать туда (внешнее представление констант в производном классе констант отсутствует)
  - указатель на свободное место в таблице
  - общий размер таблицы

Шаблон таблиц объектов

```
template <class Object> class ObjectTable int size; public: Object ** p; int free;
    public:        ObjectTable    (int max_size);
                 ~ ObjectTable    ();
                   Object * operator [] (int k);
                   Object * put_obj   (Object * t = 0)
                            {   p [free ++] = t; return t;  }
                   Object * get_object (const buf & b) const
                            {   Object ** q = p; Object * t;
                                for (int i = 0; i < free; i ++)
                                  {  t = * q ++;
                                     if (t -> is_object (b)) return t; // Виртуальная функция
                                  }
                                return 0;
                            }
};
ObjectTable<Ident>          TI        (100);
ObjectTable<Number>         TC         (40);
ObjectTable<Operation>      TO         (20);   // Операционные объекты
```

Производные классы объектов

```
class Ident:      public ProgramObject { char * name; public: Ident (const buf & b);
                  char *  get_name    ()                  const;
                  bool    is_object   (const buf & b)  const;    };

class Number:     public ProgramObject { public: Number (const buf & b);
                  bool    is_object   (const buf & b)  const;    };

Ident *     Scanner::CreateIdentObject  (const buf & b)
{                 Ident  * I = TI.get_object(b);
                  return I == 0 ? TI.put_obj  (new Ident  (b)) : I;
}
Number * Scanner::CreateNumberObject (const buf & b)
{                 Number * N = TC.get_object(b);
                  return N == 0 ? TC.put_obj (new Number (b)) : N;
}

class Operation: public ProgramObject { char * sign;
                  protected: Operation (  char * str, type_of_lex t);
        public:   bool    is_object   (const buf & b)  const;    };
```

Списки в модельном языке

- Правила со списками грамматики модельного языка:

  $B \rightarrow$ ***begin*** S {;S} ***end***

  $D_1 \rightarrow$ ***var*** D {,D} $\qquad D \rightarrow I$ {,I}: [***int*** | ***bool***]

  $E_1 \rightarrow T$ {[+ | - | ***or***] T} $\qquad T \rightarrow F$ {[* | / | ***and***] F}

- Вычисление множеств *first* и *follow*:

  D_1: *first (List)* = {,} $\qquad$ *follow (List)* = {;} $\oplus$

  D: *first (List)* = {,} $\qquad$ *follow (List)* = {:} $\oplus$

  B: *first (List)* = {;} $\qquad$ *follow (List)* = {***end***} $\oplus$

  E_1: *first (List)* = {+ – ***or*** } $\qquad$ *follow (List)* = {< > <= >= = !=}
  $\cup$ *follow (E)* = {) ***then do*** } $\cup$ *follow (S)* = { ; ***end else*** } $\oplus$

  T: *first (List)* = {* / ***and***} $\qquad$ *follow (List)* = { + – ***or*** }
  $\cup$ *follow* (E_1) = { < > <= >= = !=) ; ***do else end then*** } $\oplus$

Списки в модельном языке

- Терминализация правил для символов *S* и *E*:

  $S \rightarrow B$

  $E \rightarrow E_1 \mid E_1 [= \mid < \mid > \mid <= \mid >= \mid !=] E_1$

- Удаление общих начал в правилах для символа *E*:

  $E \rightarrow E_1 E_2$

  $E_2 \rightarrow [= \mid < \mid > \mid <= \mid >= \mid !=] E_1 \mid \varepsilon$

- Вычисление множеств *first* и *follow* для символа E_2:

  E_2: *first* $(E_2) = \{< > <= >= = !=\}$

  follow (E_2) = *follow* $(E) = \{) ;$ ***do else end then*** $\}$ ⊕

- ***Метод рекурсивного спуска к модельному языку применим***

Семантический анализ описаний

- Правила грамматики модельного языка, на основе которых могут порождаться операторы описания данных:

  $P \rightarrow$ program D_1;B⊥

  $D_1 \rightarrow$ var D {,D}

  $D \rightarrow$ I {,I}: [int | bool]

- При компиляции :
  - имена локальных объектов блоков дополняются именами блоков (функций, процедур), в которых они описаны
  - имена внутренних переменных и функций модулей программы дополняются именами самих этих модулей
  - имена процедур и функций дополняются именами классов или объемлющих процедур
  - имена методов классов и перегруженных функций дополняются именами, строящимися в зависимости от числа и типов их формальных параметров

Оператор присваивания

- ***Оператор присваивания*** является двухместным: *I := E*
- Оператор присваивания в ПОЛИЗ: *&I E := ;*
  или *I E := ;*
- Операнды двухместной операции присваивания ‘*:=*’
  - адрес переменной *I* (обозначается как *&I* или *I*) и
  - ПОЛИЗ выражения *E* (обозначается как *E*)
- Операция ‘*;*’ удаляет ненужный результат

Операторы ввода/вывода

- ***Операторы ввода/вывода*** являются одноместными: *read (I)* *write (E)*
- Представление операторов ввода/вывода в ПОЛИЗ: *&I Read* и *E Write*
- Запись *&I* означает, что операндом операции является адрес переменной *I*, а не её значение
- Двойное подчёркивание означает использование ПОЛИЗ подчёркнутого элемента
- Образ ***составного оператора*** есть последовательность образов составляющих операторов

Классы объектной модели

- class buf
- class TokenTable — TD, TW
- class Token
- template<class Object> class ObjectTable — TA, TC, TI, TL, TO, PLZ
- class ProgramObject
  - class Address: public ProgramObject
  - class Ident: public ProgramObject
  - class Label: public ProgramObject
  - class Number: public ProgramObject
  - class Operation: public ProgramObject
    - class TrueObject: public Operation …
    - class ReadObject: public Operation
- class Scanner
- class Parser
- class Simulator
- template <class T, int max_size> class Stack — Names, Types, Values

Обработка ошибок при анализе

- Возбуждение исключительных ситуаций осуществляется внутри методов *Analyze ()* и *P ()*:

```
Parser::  Parser  (char * program): scan (program)  {}
Parser::~Parser  ()                                 {}
void  Parser:: Analyze () { GetL (); P (); } // запуск анализатора
int main (int argc, char ** argv)
{ /* ... */
  try { Parser * M = new Parser ("program.txt");
        M -> Analyze (); delete M;
      }
  catch (char c)      { cout << "Неверный символ при "
                        "лексическом анализе: "      << c << endl; }
  catch (Token * t)  { cout << "Неверная лексема при "
                        "синтаксическом анализе: " << t << endl; }
}
```

Отладочная информация

```
virtual ostream & ProgramObject::print (ostream & s) const = 0;

ostream&     Ident ::print  (ostream& s) const {s << “Имя   ”   << name << endl; return s; }
ostream& Number::print  (ostream& s) const {s << “Число=” << value  << endl; return s; }

ostream & operator <<(ostream & s, const Token * t)
{  ProgramObject * p; int i;
   s << “(Тип = ”;
   s.width (2); s << t -> type << “)”;
   if  ((p = t -> get_value ()) != 0) { s << “ ”; p -> print (s); }
   else { for (i = 0; i < sizeof (KeyWords) / sizeof (KeyWords [0]); i ++)
                if (LKeyWords [i] == t -> type)
                    { s << “ Слово ” <<   KeyWords [i] << endl; return s; }
           for (i = 0; i < sizeof (Delimiters) / sizeof (Delimiters [0]); i ++)
                if (LDelimiters [i] == t -> type)
                    { s << “ Знак  ” << Delimiters [i] << endl; return s; }
           s << endl;
         }
   return s;
}
```

Анализ описаний

- Класс идентификаторов:

```
class Ident: public class ProgramObject { /* ... */
  char * name;  /* Ссылка на внешнее представление */
  bool declare;  /* Признак описания идентификатора */
  bool assign;   /* Признак значения идентификатора */ };
```

- Таблицы имён TI и констант TC:

```
ObjectTable<Ident>  TI (100);
ObjectTable<Number> TC  (40);
```

- Методы класса программных объектов и производных классов:

```
type_of_lex ProgramObject::get_type () const  { return type;     }
void        ProgramObject::set_type (type_of_lex t) { type = t; }
bool        Ident::get_declare       () const  { return declare; }
void        Ident::set_declare       ()        { declare = true; }
bool        Ident::get_assign        () const  { return  assign; }
void        Ident::set_assign                  { assign  = true; }
```

Анализ описаний

- Раздел описаний в модельном языке:

  $D \rightarrow I\ \{,I\}: [\ \mathbf{int} \mid \mathbf{bool}\ ]$

- Пример конкретного описания: ***var*** *m, n, p:* ***int;***

- Стек *Names* используется для хранения указателей на лексемы, содержащие в поле *value* ссылки на строки таблицы *TI*: *Stack <Token *, 100> Names;*

- Шаблон стеков:

```
template <class T, int max_size> class Stack { T s [max_size]; int top;
    public:    Stack       ()          {  reset ();                                  }
      void     reset       ()          {  top = 0;                                   }
      void     push        (T i)       {  if (! is_full  ())        s [top ++] = i;
                                           else throw "Стек переполнен";            }
         T     pop         ()          {  if (! is_empty ())        return s [--top];
                                           else throw "Стек исчерпан";              }
      bool     is_empty    () const    {  return top <= 0;                           }
      bool     is_full     () const    {  return top >= max_size;                    }  };
```

Анализ описаний

- Занесение в таблицу идентификаторов информации с помощью метода *decl ()* класса *Parser*

```
void Parser::decl (type_of_lex type) const
  { while (! Names.is_empty ())
      {  Token * Ident_lex = Names.pop ();
         Ident * t = dynamic_cast<Ident*> (Ident_lex -> get_value ());
         delete Ident_lex;
         if (t -> get_declare ()) throw “Повторное описание”;
         else { t -> set_declare ();  t -> set_type (type); }
      }
  }
```

Анализ описаний

D → <Reset ()> I <Push (name)> {, I <Push (name)>} :
*[**int** <Decl ("**int**")> | **bool** <Decl ("**bool**")>]*

- Процедуры анализа описаний:

```
void Parser::D () {  Names.reset ();                    // D → I {,I}: [int|bool]
    if (c_type != LEX_ID)          throw curr_lex;
                                   Names.push (curr_lex);          GetL ();
    while (c_type == LEX_COMMA)  {                                 GetL ();
        if (c_type != LEX_ID)      throw curr_lex;
                                   Names.push (curr_lex);          GetL ();  }
    if (c_type != LEX_COLON)       throw curr_lex;
                                                                   GetL ();
    if (c_type == LEX_INT || c_type == LEX_BOOL) { decl (c_type);  GetL ();  }
    else                           throw curr_lex;
}

void Parser::D1 () {                                    // D1 → var D {,D}
    if (c_type != LEX_VAR)         throw curr_lex;
    do { GetL (); D  (); } while (c_type == LEX_COMMA);
}
```

Обработка ошибок при анализе

```
void  Parser:: Analyze () { GetL (); P (); } // запуск анализатора

int main (int argc, char ** argv) { bool res = false; /* ... */
  try { Parser * M = new Parser (“program.txt”);
          M -> Analyze (); delete M;
        }
  catch (char c)                    { cout << “Неверный символ при ”
                                “лексическом анализе: ”      << c << endl; }
  catch (Token * t)                 { cout << “Неверная лексема при ”
                                “синтаксическом анализе: ”   << t << endl; }
  catch (ProgramObject * l)     { cout << “Неверный объект при ”
                                “синтаксическом анализе: ”   << l << endl; }
  catch (const char * source)   { cout << source << endl; }
  cout << “res = ” << (res ? “true ” : “false”) << endl;
  return 0;
}
```

Анализ выражений

- Стек *Types* хранит типы используемых в выражениях операндов и/или промежуточных значений выражений:

  Stack <type_of_lex, 100> Types;

- Семантические процедуры класса Parser:

  check_op – проверка совпадения типов двух операндов бинарной операции

  check_not – проверка типа операнда унарной операции отрицания

  check_id – контроль наличия описания идентификатора

  eq_type – сравнение типов двух операндов из стека

  check_id_in_read – контроль наличия описания идентификатора в операторе чтения

Анализ выражений

- Процедуры анализа выражений:

```
void Parser::check_op () const {
    type_of_lex t  = LEX_INT, r = LEX_BOOL;
    type_of_lex t2 = Types.pop ();
    type_of_lex op = Types.pop ();
    type_of_lex t1 = Types.pop ();
    if (op == LEX_PLUS  || op == LEX_MINUS ||
        op == LEX_MULT  || op == LEX_DIV)      r = LEX_INT;
    if (op == LEX_OR    || op == LEX_AND)      t = LEX_BOOL;
    if (t1 == t2  &&  t1 == t)                 Types.push (r);
    else throw "Неверные типы в двухместной операции";
}
void Parser::check_not () const {
    if  (Types.pop () != LEX_BOOL) throw "Неверный тип в операции отрицания";
    else Types.push (LEX_BOOL);
}
void Parser::check_id () const {
    Ident * t = dynamic_cast<Ident*> (curr_lex -> get_value ());
    if (t -> get_declare ()) Types.push (t -> get_type ()); else throw "Нет описания";
}
```

Действия для выражений

- Правила грамматики модельного языка для выражений:

  $E \rightarrow E_1\ E_2$

  $E_1 \rightarrow T\ \{[\ +\ |\ -\ |\ \textbf{or}\ ]\ T\}$

  $E_2 \rightarrow [\ =\ |\ <\ |\ >\ |\ <=\ |\ >=\ |\ !=\ ]\ E_1\ |\ \varepsilon$

  $T \rightarrow F\ \{[\ *\ |\ /\ |\ \textbf{and}\ ]\ F\}$

  $F \rightarrow I\ |\ N\ |\ L\ |\ \textbf{not}\ F\ |\ (E)$

- Правила грамматики с учётом семантических процедур:

  $E \rightarrow E_1\ E_2$

  $E_1 \rightarrow T\ \{[\ +\ |\ -\ |\ \textbf{or}]$ <Push (type)> T <check_op ()>$\}$

  $E_2 \rightarrow [\ =\ |\ <\ |\ >\ |\ <=\ |\ >=\ |\ !=]$ <Push (type)> E_1 <check_op ()> $|\ \varepsilon$

  $T \rightarrow F\ \{[\ *\ |\ /\ |\ \textbf{and}]$ <Push (type)> F <check_op ()>$\}$

  $F \rightarrow I$ <check_id ()> $|$

  N <Push (type=**int**)> $|$

  L <Push (type=**bool**)> $|$

  not F <check_not ()> $|\ (E)$

Анализ выражений

```
void Parser::E ()                          // E → E1 E2   E2 → [=|<|<=|>=|>|!=] E1 | ε
{ E1 (); if (c_type == LEX_EQ || c_type == LEX_LT || c_type == LEX_GT ||
             c_type == LEX_LE || c_type == LEX_GE || c_type == LEX_NE )
               { Types.push (c_type);        GetL ();        E1 ();      check_op (); }    }
void Parser::E1 ()                         // E1 → T {[+|-|or] T}
{ T (); while (c_type == LEX_PLUS || c_type == LEX_MINUS || c_type == LEX_OR)
               { Types.push (c_type);        GetL ();        T  ();      check_op (); }    }
void Parser::T  ()                         // T → F {[*|/|and] F}
{ F (); while (c_type == LEX_MULT || c_type == LEX_DIV || c_type == LEX_AND)
               { Types.push (c_type);        GetL ();        F  ();      check_op (); }    }
void Parser::F  () {                       // F → I | N | L | not F | (E)
       if (c_type == LEX_ID)       { check_id  ();          delete curr_lex;GetL (); }
  else if (c_type == LEX_NUM)      { Types.push (LEX_INT); delete curr_lex;GetL (); }
  else if (c_type == LEX_TRUE)     { Types.push (LEX_BOOL);                GetL (); }
  else if (c_type == LEX_FALSE)    { Types.push (LEX_BOOL);                GetL (); }
  else if (c_type == LEX_NOT)      {       GetL ();        F ();       check_not ();    }
  else if (c_type == LEX_LPAREN){          GetL ();        E ();
       if (c_type == LEX_RPAREN)           GetL (); else throw curr_lex;              }
  else throw curr_lex;
}
```

Контроль типов в операторах

1. *В операторах присваивания*: типом результата должен быть тип идентификатора, которому осуществляется присваивание; одновременно должна проводиться проверка описания у идентификатора, которому присваивается новое значение
2. *В условных операторах* у выражений, результат вычисления которых влияет на выбор альтернативы вычислений, должен быть только логический тип
3. *В операторах цикла* у выражений, результат вычисления которых влияет на продолжение выполнения операторов тела цикла, должен быть только логический тип
4. *В операторах чтения* должна проводиться проверка наличия описания идентификаторов, которые получают значения вводом из внешнего файла

- Грамматика: $S \rightarrow I := E \mid \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{read } (I) \mid \text{write } (E) \mid B$

Контроль типов в операторах

1. Оператор присваивания

```
void Parser::eq_type () const
{ if (Types.pop () != Types.Pop ())
        throw  "Неверные типы в присваивании"; }
```

S → I <check_id ()> := E <eq_type ()>

```
void Parser::S () {    /* … */
  if  (c_type == LEX_ID)                                  // S → I := E
       { check_id (); delete curr_lex;     GetL ();
         if  (c_type == LEX_ASSIGN)    {  GetL (); E (); eq_type (); }
         else throw  curr_lex;
       } //end assign
  /* … */ }
```

Контроль типов в операторах

2. Условный оператор

```
void Parser::eq_type (type_of_lex token) const
{ if  (Types.pop () != token) throw  “Неверный тип”; }
```

S → if E <eq_type (“bool”)> then S else S

```
void Parser::S () {    /* … */
  else if (c_type == LEX_IF)                    // S → if E then S else S
        { GetL (); E (); eq_type (LEX_BOOL);
          if  (c_type != LEX_THEN) throw curr_lex;
          GetL (); S ();
          if (c_type != LEX_ELSE) throw  curr_lex;
          GetL (); S ();
        } //end if
  /* … */ }
```

Контроль типов в операторах

3. Оператор цикла

```
void Parser::eq_type (type_of_lex token) const
{ if  (Types.pop () != token) throw  "Неверный тип"; }
```

S → while E <eq_type ("bool")> do S

```
void Parser::S () {    /* … */
  else if (c_type == LEX_WHILE)              // S → while E do S
        { GetL (); E (); eq_type (LEX_BOOL);
          if (c_type != LEX_DO) throw curr_lex;
          GetL (); S ();
        } //end while
  /* … */ }
```

Контроль типов в операторах

4. Оператор чтения данных

```
void Parser::check_id_in_read () const
{ Ident * t = dynamic_cast<Ident*> (curr_lex -> get_value ());
  if (! t -> get_declare ()) throw "Не описанное имя"; }
```

S → read (I <check_id_in_read ()>)

```
void Parser::S () {    /*...*/
  else if (c_type == LEX_READ)                  // S → read (I)
        { GetL ();       if (c_type != LEX_LPAREN)   throw curr_lex;
          GetL ();       if (c_type != LEX_ID)       throw curr_lex;
          check_id_in_read ();      delete curr_lex;
          GetL ();       if (c_type != LEX_RPAREN)   throw curr_lex;
          GetL ();
        } //end read
  /* ... */ }
```

Контроль типов в операторах

5. Оператор вывода данных

S → write (E <Pop ()>)

```
void Parser::S () {    /* … */
  else if (c_type == LEX_WRITE)              // S → write (E)
        { GetL ();
          if (c_type == LEX_LPAREN)
              {  GetL (); E ();
                 Types.pop ();  /* убрать тип выражения из стека */
                 if (c_type == LEX_RPAREN) GetL (); else throw curr_lex;
              }
          else throw curr_lex;
        } //end write
  else B ();                                 // S → B
}
```

Объекты для генерации ПОЛИЗ

- Производные классы *Label* и *Address* для класса *ProgramObject*

```
class Label:     public ProgramObject { public: Label     (int n); ...};
class Address: public ProgramObject { public: Address (int n); ...};
```

- Операции “безусловный переход” и “условный переход по лжи” с внешним представлением в виде строк “*!*” и “*!F*” в таблице *TW* или в виде строк “*goto*” и “*go_if_not*” в таблице *TD*:

```
enum type_of_lex { /* ... */
  PLZ_GO,   /*40: лексема !   - “безусловный переход”  */
  PLZ_FGO   /*41: лексема !F  - “условный переход”      */  };
```

- Таблица меток *TL* и таблица адресов *TA*:

```
ObjectTable<Label>      TL (100);
ObjectTable<Address>   TA (sizeof (TI) / sizeof (TI [0]));
```

- Массив указателей на программные объекты *PLZ*:

```
ObjectTable<ProgramObject> PLZ (1000);
```

Методы для генерации ПОЛИЗ

- Функция *put_obj ()* записывает в массив обратной польской *PLZ* записи указатели на объекты, а также резервирует место для отложенной записи указателей на объекты:

```
template<class Object>Object * ObjectTable<Object>::
        put_obj (Object * t, int i)    { p [i]      = t;      return t; }
template<class Object>Object * ObjectTable<Object>::
        put_obj (Object * t = 0)       { p [free ++] = t; return t; }

template<class Object>Object * ObjectTable<Object>::
        operator []   (int k)          { return p [k]; }
template<class Object>int       ObjectTable<Object>::
        get_place () const             { return free; }

void Parser::check_op ()
        { /* ... */   PLZ.put_obj (TO [TO.get_index (op)]); }
```

Выражения и присваивание

- Правила грамматики с учётом процедур генерации:

$E \rightarrow E_1\ E_2$

$E_1 \rightarrow T$ {[+ | - | **or**] <Push (type)> T <check_op ()>}

$E_2 \rightarrow$ [= | < | > | <= | >= | !=] <Push (type)>
E_1 <check_op ()> | ε

$T \rightarrow F$ {[* | / | **and**] <Push (type)> F <check_op ()>}

$F \rightarrow$ I <check_id (); **Put (I)**> |
N <Push (type=int); **Put (N)**> |
L <Push (type=bool); **Put (L)**> |
not F <check_not ()> | (E)

Выражения и присваивание

- Действия для оператора присваивания:

*S → I <check_id (); **Put (&I)**> := E <eq_type (); **Put (“:=”)**>*

```
void Parser::S () { ProgramObject * Oper; /* ... */
  if (c_type == LEX_ID) {
        check_id ();  PLZ.put_obj (CrAddrObject (curr_lex));
        delete  curr_lex;                    GetL ();
        Oper = curr_lex -> get_value ();
        if (c_type == LEX_ASSIGN) { GetL ();
              E (); eq_type ();  PLZ.put_obj (Oper); }
        else throw curr_lex;
  } // assign-end
/* ... */}
```

Условный оператор

- Семантика условного оператора

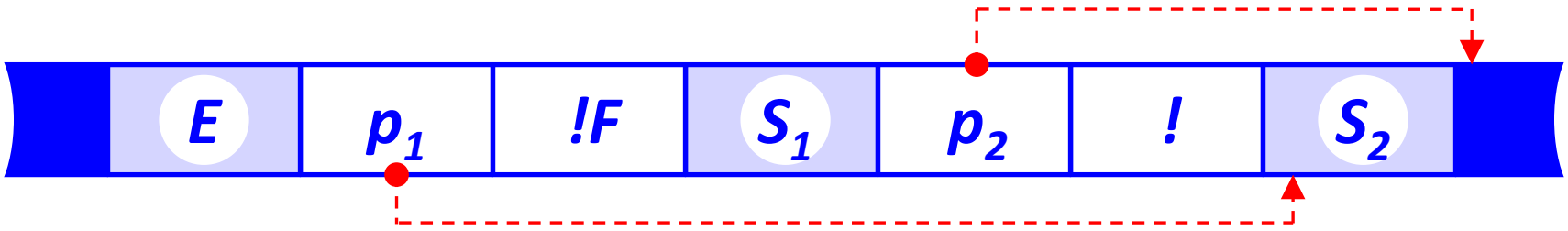
  `if` E **`then`** S_1 **`else`** S_2

  `if` $(!\,(E))$ **`goto`** Lab_1; S_1; **`goto`** Lab_2; Lab_1: S_2; Lab_2: …

- ПОЛИЗ условного оператора:

  $\underline{E}\ p_1\ !F\ \underline{S_1}\ p_2\ !\ \underline{S_2}$ … где

  p_i – номер элемента, с которого начинается ПОЛИЗ оператора с меткой Lab_i, $i = 1, 2$



Перевод условного оператора

- Действия для условного оператора:

*S → if E <eq_type (bool); **Put (“&”, lab1); Put (“!F”)***>
*then S <**Put (“&”, lab2); Put (“!”); Put (“p”, lab1)***>
*else S <**Put (“p”, lab2)***>

```
void S () { ProgramObject * Oper;   int lab1, lab2;
  if (c_type == LEX_IF)
   {  GetL (); E (); eq_type (LEX_BOOL);
      lab1=PLZ.get_place (); PLZ.put_obj (); PLZ.put_obj (TO [ind_FGO]);
      if (c_type != LEX_THEN) throw curr_lex;
      GetL (); S ();
      lab2=PLZ.get_place (); PLZ.put_obj (); PLZ.put_obj (TO [ind_GO]);
      PLZ.put_obj (CrLabelObject (PLZ.get_place ()), lab1);
      if (c_type != LEX_ELSE) throw curr_lex;
      GetL (); S ();
      PLZ.put_obj (CrLabelObject (PLZ.get_place ()), lab2);
   } // end if
/* ... */}
```

Оператор цикла с предусловием

- Семантика оператора цикла с предусловием

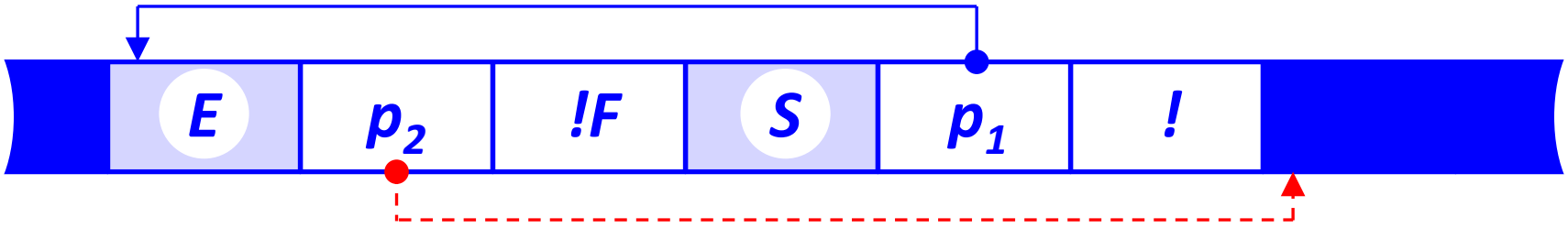
  `while` *(E)* **`do`** *S*

  L_1: **`if`** *(! (E))* **`goto`** L_2; *S*; **`goto`** L_1; L_2: …

- ПОЛИЗ оператора цикла с предусловием:

  $\underline{E}\ p_2\ !F\ \underline{S}\ p_1\ !\ \ldots$ где

  p_i – номер элемента, с которого начинается ПОЛИЗ оператора с меткой L_i, $i = 1, 2$



Перевод оператора цикла

- Действия для оператора цикла:

S → *while <**Place (lab1)**>*
*E <eq_type (bool); **Put (“&”, lab2)**; **Put (“!F”)**>*
*do S <**Put (lab1)**; **Put (“!”)**; **Put (“p”, lab2)**>*

```
void S () { /* … */
  else if (c_type == LEX_WHILE)
      {  GetL (); lab1 = PLZ.get_place (); E (); eq_type (LEX_BOOL);
                  lab2 = PLZ.get_place (); PLZ.put_obj ();
         PLZ.put_obj (TO [ind_FGO]);
         if (c_type != LEX_DO) throw curr_lex;
         GetL (); S();
         PLZ.put_obj (CrLabelObject (lab1));
         PLZ.put_obj (TO [ind_GO]);
         PLZ.put_obj (CrLabelObject (PLZ.get_place ()), lab2);
      } // end while
/* … */}
```

Создание новых объектов

- Процедуры создания новых меток и адресов:

```
Address * Parser::CrAddrObject (Token * t)
{ int Ident_index  = TI.get_index (t);
  int Adr_index    = TA.get_index (Ident_index);
  return Adr_index >= 0 ? TA [Adr_index] :
                  TA.put_obj (new Address(Ident_index));
}

Label * Parser::CrLabelObject (int label)
{ int Label_index  = TL.get_index (label);
  return Label_index >= 0 ? TL [Label_index] :
                  TL.put_obj (new Label (label));
}
```

Интерпретация ПОЛИЗ

- Чистая виртуальная функция в классе *ProgramObject*:

```
virtual void ProgramObject::exec (int & i) const = 0;

ObjectTable<ProgramObject> PLZ (1000);
Stack<int, 100> Values;

void Simulator::Simulate ()
    { int size = PLZ.get_place (); Values.reset ();
      for (int index = 0; index < size; ++index)
                        PLZ [index] -> exec (index);
    }

int main () { /* ... */
      try { Simulator * S = new Simulator (); S -> Simulate (); }
      catch (const char * source) { cout << source << endl; }
    }
```

Функции интерпретации ПОЛИЗ

```
void          Ident::exec (int&) const { if (get_assign ()) Values.push (get_value ());
                                         else throw “PLZ: неопределённое значение имени”;}
void         Number::exec (int&) const { Values.push (get_value ());                    }
void        Address::exec (int&) const { Values.push (get_value ());                    }
void          Label::exec (int&) const { Values.push (get_value ());                    }
void     TrueObject::exec (int&) const { Values.push (1);                               }
void    FalseObject::exec (int&) const { Values.push (0);                               }
void      NotObject::exec (int&) const { Values.push (1 - Values.pop ());               }
void       OrObject::exec (int&) const { Values.push (Values.pop () || Values.pop ());  }
void       EqObject::exec (int&) const { Values.push (Values.pop () == Values.pop ());  }
void       LeObject::exec (int&) const { Values.push (Values.pop ()   > Values.pop ());  }
void     PlusObject::exec (int&) const { Values.push (Values.pop ()   + Values.pop ());  }
void   MinusObject::exec (int&) const { int k = Values.pop (); Values.push (Values.pop () - k);  }
void      DivObject::exec (int&) const { int k = Values.pop ();
                    if (k) Values.push (Values.pop () / k); else throw “PLZ: деление на нуль”;  }
void  AssignObject::exec (int&) const { int k = Values.pop (); Ident * t = TI [Values.pop ()];
                                        t -> set_value (k); t -> set_assign ();          }
Void   WriteObject::exec (int&) const { cout << Values.pop () << endl;                   }
void    GoToObject::exec (int& i) const { i = Values.pop () - 1;                         }
voidGoIfNotObject::exec (int& i) const { int k = Values.pop (); if (! Values.pop ()) i = k - 1;  }
```