УДК 004.415.2:004.4’413

М.Ф. ЛЕКАРЕВ, Е.В. ПЫШКИН

СИНТАКСИЧЕСКИЙ АНАЛИЗ ВЫРАЖЕНИЙ В СКОБОЧНОЙ ФОРМЕ НА ОСНОВЕ ВИЗУАЛЬНОГО ФОРМАЛИЗМА L-СЕТИ

Исследуется задача синтаксического анализа логических выражений, обобщаемая и на случай арифметических выражений. Для распознавания выражений и реализации схемы вычислений используется визуальный формализм для разработки программного обеспечения, отличающегося существенной логической сложностью, который был предложен и реализован М.Ф. Лекаревым и апробирован ряде сложных программных проектов [2-4]. Визуальный формализм на базе L-сетей позволяет преодолеть ряд проблем управления вычислительным процессом, свойственных процедурной парадигме, и для различных классов задач может служить альтернативой использования объектно-ориентированных методов или применяться совместно в рамках реализации моделей мультипарагматического программирования.

Б. Шнейдерман указывал, что сложность программного обеспечения (ПО) может иметь логический, структурный или психологический характер [5]. Логическая сложность программы обусловлена большим числом проверяемых условий и неочевидностью их логических связей. Это обуславливает трудность, длину или невозможность доказательства корректности программы. Необходимость проверки большого количества условий возникает обычно при реализации сложных эвристических алгоритмов, в ходе синтаксической обработки входных данных, а также при большом количестве ограничений на обрабатываемые данные. Психологическая сложность связана с такими характеристиками, которые затрудняют понимание программы человеком. Структурная сложность может быть измерена в терминах абсолютной и относительной структурной сложности. Абсолютная структурная сложность измеряется количеством модулей, из которых состоит программа. Относительная структурная сложность измеряется как отношение числа взаимосвязей между модулями к общему числу модулей. Связь между модулями реализуется интерфейсом (например, списком параметров при вызове функции). В свою очередь, сложность интерфейса можно определить как количество информации, требуемое для установления или понимания связи между модулями.

Одним из систематических методов преодоления логической сложности ПО, является его визуализация [2]. При этом визуальный формализм представления программ и определяемого программами вычислительного процесса, а также соответствующие новые способы организации обработки информации и взаимодействия человека с компьютером позволяют существенно сократить трудоемкость и сроки проектирования сложного ПО.

Основу структурности визуального формализма разработки ПО на базе L-сети составляет два комплекта визуальных примитивов, один из которых ориентирован на представление программы в форме сети автоматов, которые могут вызывать другие автоматы подобно подпрограммам, а другой – на представление программы в форме иерархической сети модулей. На нижнем уровне иерархии находятся операционные модули. Они относительно просты. Для разработки таких модулей используются обычные языки программирования. Объединение модулей со всеми сложными связями по управлению и данным осуществляется с использованием средств визуального формализма, реализованного в виде модели L-сети. Представление управления последовательностью действий в L-сети основывается на использовании в L-сети модулей с двумя выходами и возможными побочными эффектами. Расширены возможности определения точки продолжения вычислений после завершения работы с модулем: возвраты и окончания с шагом, с переходом, с успехом, с неуспехом, с мультиветвлением. Для визуализации

операций управления данными используются графические примитивы, названные интерфейсами. Доказано, что связь через интерфейсы обеспечивает такие же возможности, как и связь через аргументы-параметры в распространенных языках программирования. Разработан способ соединения графических примитивов управления последовательностью действий с графическими примитивами управления данными.

Задача синтаксического разбора логических и арифметических выражений относится к числу типовых задач синтаксического анализа и компиляции, поэтому исследование решения этой задачи интересно не столько само по себе, сколько с целью оценки преимуществ, предоставляемых используемыми моделями организации данных и вычислительного процесса. При решении задач, связанных с анализом и синтезом дискретных устройств часто требуется обеспечить обработку факторизованных логических функций (ЛФ), представленных в скобочной форме записи с инфиксными операциями. Такие выражения обычно получаются после этапов минимизации и факторизации [1]. Скобочная форма записи выражений может считаться моделью, пригодной для большинства практических случаев.

Таким образом, синтаксис факторизованных ЛФ определяется правилами контекстно-свободной грамматики $\boldsymbol{E} = (VN, VT, P, S)$, где VN – множество нетерминальных символов грамматики, VT – множество терминальных символов, S – начальный символ грамматики, P – множество правил грамматики.

Пусть правила грамматики $\boldsymbol{E}$ выглядят следующим образом (элементы множества терминальных символов выделены полужирным начертанием и заключены в кавычки или апострофы):

```
S ::= факторизованная-ЛФ
факторизованная-ЛФ ::= имя-ЛФ "=" выражение ";"
имя-ЛФ ::= идентификатор
выражение ::= терм [ "+" терм ]…
терм ::= фактор [ "*" фактор ]…
фактор ::= { [ "~" ] имя-переменной | двоичная-константа | "(" выражение ")" }
имя-переменной := идентификатор
идентификатор ::= латинская-буква [ { латинская-буква | десятичная-цифра } ] ...
латинская-буква ::= {'_'|'A'|'B'|...|'Z'|'a'|'b'|...|'z'}
десятичная-цифра ::= {'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'}
двоичная-константа ::= {'0'|'1'}
```

Синтаксические диаграммы, определяющие нетерминал <выражение>, приведены на рис. 1.

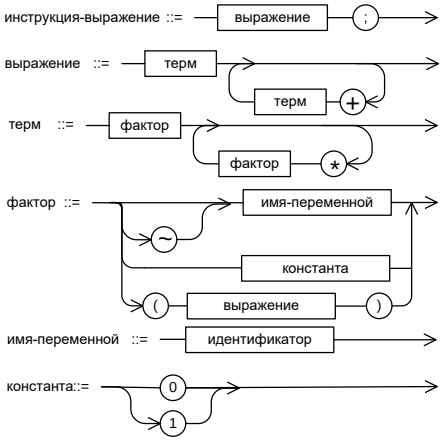

Рис. 1. Синтаксис логических выражений

В соответствии с синтаксисом логических выражений нужно формально определить классы синтермов, которые должен распознавать лексический анализатор (сканер). Такая классификация используется для уменьшения количества альтернативных условий, которые необходимо проверить. Например, для распознавания первого символа в составе идентификатора, который должен являться латинской буквой, можно использовать следующий синтерм:

<латинская- буква> ::= {**'A','B',...,'Z','a','b',...,'z'**}

Таким образом, число рассматриваемых альтернатив сокращается с 52 до 1. При необходимости несложно обеспечить возможность работы с синтермами, включающими прочие буквы (греческие, русские и т.д.). Так. представляемая далее таблица синтермов обеспечивает, в частности, определение синтерма <русская-буква>.

Для распознавания остальных литер в составе идентификатора наибольшее сокращение количества альтернатив достигается при определении синтерма <буква-или-цифра>, который пересекается с синтермом <латинская-буква> по составу литер. Заметим, что использование классов лексем с пересечениями позволяет интерпретировать, например, литеру '0' и как десятичную цифру (при распознавании идентификатора), и как двоичную константу.

Теоретически, множество синтермов S_T - это множество всех подмножеств VT. Однако на практике обычно требуется значительно меньшее количество синтермов S_P , так что $S_P \subseteq S_T$, $\#S_P << \#S_T$, где знаком # обозначена мощность множества [2]. Теперь, исходя из правил грамматики, выделим следующие две разновидности нетерминалов:

– нетерминальные символы грамматики, определяемые непосредственно через синтермы, т.е. такие $V_i \in VN$, что $V_i ::= P_t(V_j)$, $V_j \in S_T$, где $P_t(V_s)$ означает, что символ V_s содержится в правой части правила P_t, для которого справедливо: $P_t \in P$;
– одиночные литеры, которые образуют такие подмножества S_{Ti} множества синтермов S_T, что $\# S_{Ti} = 1$.

Эти две группы элементов образуют множество *классов лексем*, формируемых сканером и распознаваемых синтаксическим анализатором. Так, в грамматике ***E*** в это множество входят нетерминалы <идентификатор> и <двоичная-константа>, а также множество однолитерных лексем: ';', '+', '*', '~', '(', ')', '='.

Для использования в таблице синтермов определим следующие универсальные классы синтермов, значения которых представлены двоичными кодами, являющимися степенями двойки, то есть имеющие ровно один единичный разряд (здесь приводятся определения констант препроцессора языка «Си»):

```
/*
 * Классы синтермов
 */
#define  NOALP  0x0000      /* NOn ALPhabetic : не входящая в алфавит */

// В связи с идентификаторами:
#define  LAT    0x0001      // LATin : латинская буква
#define  LD10   0x0002      // Latin or Digit_10 : латинская буква
                            //     или десятичная цифра
#define  RUS    0x0004      // RUSsian : русская буква
#define  LRD10  0x0008      // Latin or Russian or Digit_10:
                            //     латинская или русская буква
                            //     или десятичная цифра

// В связи с константами в разных системах счисления:
#define  D10    0x0010      // Digit_10 : десятичная цифра
#define  D16    0x0020      // Digit_16 : шестнадцатеричная цифра
#define  D0     0x0040      // Digit_0  : "нуль" (также логический)
#define  D1     0x0080      // Digit_1  : "единица" (также логическая)
```

```
#define  X      0x0800      // латинская литера 'X' или 'x'
```

Для представления подмножества непересекающихся синтермов (образуемых однолитерными лексемами), определим следующие значения:

```
// В связи с классами без пересечений: из одной или нескольких допустимых
// литер алфавита, которые входят в единственный класс:
#define  ONLY       0x8000      // ONLY : признак класса без пересечений
#define  A          ONLY | '\'' // Apostrophe : апостроф
#define  BLANK      ONLY | ' '  // BLANK : "пробельная" литера (разделитель)
#define  LF         ONLY | '\n' // Line Finish : конец строки
#define  LPARENT    ONLY | '('  // Left PARENThese  : левая скобка
#define  RPARENT    ONLY | ')'  // Right PARENThese : правая скобка
#define  NEGATE     ONLY | '~'  // NEGATion opEration : отрицание

// Другие непересекающиеся синтермы:
// (логическое) умножение  ONLY | '*'
// (логическое) сложение   ONLY | '+'
// точка с запятой         ONLY | ';'
// присваивание            ONLY | '='
```

При этом константа `ONLY` устанавливает в единицу старший разряд значения синтерма, что в предложенных терминах определяет непересекающийся синтерм, значение которого однозначно соответствует значению кода литеры, образующей однолитерную лексему. Ниже приводятся фрагменты кода, позволяющие получить представление о том, как выглядит таблица синтермов в нашем случае:

```
typedef                 // Тип: синтерм литер анализируемого текста
    unsigned int
    TSynterm [256];     // Type of SYNtax TERM

// Таблица синтермов
TSynterm synterm =
{
    // Управляющие и служебные символы (не имеют графических очертаний)
    /* ..0  0x00        ^@  NUL */   LF,
    /* ..1  0x01        ^A  SOH */   BLANK,
    /* ..2  0x02        ^B  STX */   BLANK,
    /* ..3  0x03        ^C  ETX */   BLANK,
    //...
    // Символы, имеющие графические очертания
    /* .32  0x20  ' '  SP SPace */   BLANK,
    /* .33  0x21  '!'  */           NOALP,
    /* .34  0x22  '\"' */           NOALP,
    /* .35  0x23  '#'  */           NOALP,
    /* .36  0x24  '$'  */           LAT  | LD10 | LRD10,
    /* .37  0x25  '%'  */           NOALP,
    /* .38  0x26  '&'  */           NOALP,
    /* .39  0x27  '\'' */           NOALP,
    /* .40  0x28  '('  */           LPARENT,
    /* .41  0x29  ')'  */           RPARENT,
    /* .42  0x2A  '*'  */           ONLY | '*',
    /* .43  0x2B  '+'  */           ONLY | '+',
    // ...
    /* .48  0x30  '0'  */   D10  | LD10 | LRD10 | D0,
    /* .49  0x31  '1'  */   D10  | LD10 | LRD10 | D1,
```

```
    /* .50  0x32  '2'  */   D10  | LD10 | LRD10,
    // ...
    /* .57  0x39  '9'  */   D10  | LD10 | LRD10,
    /* .58  0x3A  ':'  */   NOALP,
    /* .59  0x3B  ';'  */   ONLY | ';',
    /* .60  0x3C  '<'  */   NOALP,
    /* .61  0x3D  '='  */   ONLY | '=',
    /* .62  0x3E  '>'  */   NOALP,
    /* .63  0x3F  '?'  */   NOALP,
    /* .64  0x40  '@'  */   LAT  | LD10 | LRD10,
    /* .65  0x41  'A'  */   LAT  | LD10 | LRD10,
    /* .66  0x42  'B'  */   LAT  | LD10 | LRD10,
    // ...
    /* .90  0x5A  'Z'  */   LAT  | LD10 | LRD10,
    /* .91  0x5B  '['  */   NOALP,
    /* .92  0x5C  '\\' */   NOALP,
    /* .93  0x5D  ']'  */   NOALP,
    /* .94  0x5E  '^'  */   NOALP,
    /* .95  0x5F  '_'  */   LAT  | LD10 | LRD10,
    /* .96  0x60  '`'  */   NOALP,
    /* .97  0x61  'a'  */   LAT  | LD10 | LRD10,
    /* .98  0x62  'b'  */   LAT  | LD10 | LRD10,
    // ...
    /* 122  0x7A  'z'  */   LAT  | LD10 | LRD10,
    // ...
    /* 126  0x7E  '~'  */   ONLY | '~',
    // ...
    /* 128  0x80  'А'  */   RUS  | LRD10,
    /* 129  0x81  'Б'  */   RUS  | LRD10,
    // ...
    /* 159  0x9F  'Я'  */   RUS  | LRD10,
    /* 160  0xA0  'а'  */   RUS  | LRD10,
    /* 161  0xA1  'б'  */   RUS  | LRD10,
    // ...
    /* 174  0xAE  'о'  */   RUS  | LRD10,
    /* 175  0xAF  'п'  */   RUS  | LRD10,
    /* 176  0xB0 '-'   */   NOALP,
    /* 177  0xB1   '-' */   NOALP,
    // ...
    /* 224  0xE0  'р'  */   RUS  | LRD10,
    /* 225  0xE1  'с'  */   RUS  | LRD10,
    // ...
    /* 239  0xEF  'я'  */   RUS  | LRD10,
    /* 240  0xF0  'Ё'  */   NOALP,
    /* 241  0xF1  'ё'  */   NOALP,
    // ...
    /* 255  0xFF  ' '  */   NOALP
};
// Конец инициализации таблицы синтермов
```

Отметим, что единожды определенная таблица синтермов просто перестраивается для самых разнообразных задач синтаксического анализа. При этом для определения принадлежности литер синтерму удобно работать с масками синтермов:

```
#define  MASK_LAT    ONLY | LAT
```

```
#define  MASK_LD10   ONLY | LD10
#define  MASK_RUS    ONLY | RUS
#define  MASK_LRD10  ONLY | LRD10
#define  MASK_D10    ONLY | D10
#define  MASK_D16    ONLY | D16
#define  MASK_D0     ONLY | D0
#define  MASK_D1     ONLY | D1
#define  MASK_X      ONLY | X
```

При наличии таблицы синтермов определение синтерма сканируемой литеры происходит очень просто:

```
// GET LITera from input stream
// Чтение очередной литеры из входного потока
OM( getlit ) // Операционный модуль на языке C
    litera.value = getc( inputFile );
    if( litera.value == EOF )   TRANS;
    litera.synterm = synterm[ litera.value ];
    // ...
    lnetvar = litera.synterm; // Формирование вектора для анализа
                              // в процессе сложного ветвления
    STEP;
ENDOM
```

Образование значения синтерма (на примере латинской литеры ‘A’) и последующее его распознавание иллюстрирует рис. 2.

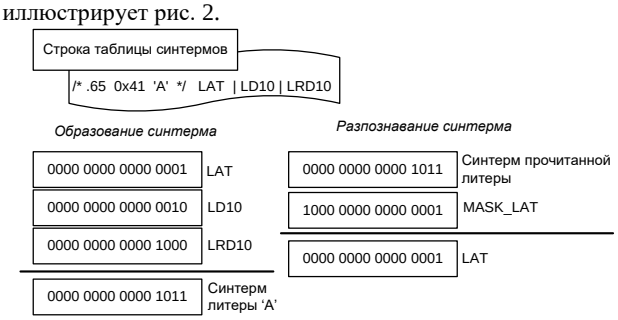

Рис. 2. Образование и распознавание синтерма

Результатом однократного вызова сканера является очередная лексема входного текста, для которой выявлен класс лексемы. В сетевой программе *ScanExpr* (рис. 3) успешному завершению работы сканера соответствует возврат с шагом по L-сети (*SReturn*), что позволяет по результатам работы сканера строить сложные ветвления при анализе соответствия текста выражения объявленному синтаксису. Возврат с переходом (*TReturn*) соответствует ситуации конца файла. Проверка условий (в данном случае – с целью распознавания синтерма) осуществляется посредством обособленных условий, являющихся одним из конструктивных элементов дуги L-сети. Содержанием обособленного условия является проверка полного или маскированного совпадения двоичного вектора (в нашем случае – класса литеры, формируемого в результате последнего считывания) с заданной константой (синтермом). В нашем случае семантика обособленного условия для распознавания, например, латинской буквы, состоит в следующем:

```
if( lnetvar & MASK_LAT == LAT )    TRANS;
else                               STEP;
```

При истинности обособленного условия управление передается функциональному модулю, находящемуся на той же дуге сети, что и сетевая программа (переход по дуге сети – TRANSition).

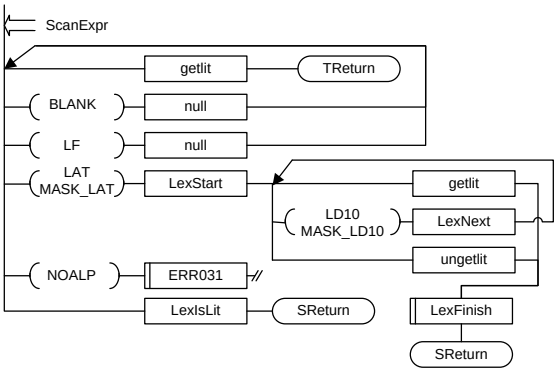

Рис. 3. Сканер логического выражения

Функциональный модуль может, в свою очередь, завершиться одним из двух способов: завершение с переходом по дуге сети (фактически, это означает, переход к следующему участку L-сети, например, к следующему сложному ветвлению), завершение с шагом по сети (STEP – шаг по сети: переход к следующей по порядку дуге сложного ветвления сети). При ложности обособленного условия также происходит переход к следующей дуге сложного ветвления (STEP). В [2] отмечается, что обособленное условие модели L-сети позволяет унифицировать не только вычисление отношения эквивалентности, но и определение принадлежности элемента множеству. Фактически, L-сеть является вариантом реализации модели конечного автомата, где операция продвижения вперед по входной цепочке (считывание) вынесена на явный уровень, а геометрическим образом состояния является вертикальная линия, объединяющая все дуги сложного ветвления.

Схема решения задачи синтаксического анализа логического выражения, представлена на рис. 4 в форме фрагмента комплекта сетевых программ L-сети. Видно, что структура анализатора графически довольно точно соответствует синтаксическим диаграммам, представленным на рис. 1.

Привычная для человека общепринятая инфиксная запись логических выражений не вполне удобна для анализа формулы на ЭВМ. Предпочтительнее оказывается постфиксная (польская) запись, которая для двуместных операций имеет вид `<операнд-1><операнд-2><операция>`. Как явствует из анализа сетевых программ, приведенных на рис. 4, в результате распознавания логического выражения обеспечивается формирование описания выражения в постфиксной форме. При этом выражение представляется в виде массива, содержащего два типа элементов: элементы-переменные, (или константы), элементы-операции. Формирование элементов массива осуществляется сетевыми программами *OutVar* (помещение в польскую запись идентификатора, соответствующего переменной в составе выражения), *OutConst* (помещение в польскую запись двоичной константы в составе выражения), *OutConj* (помещение в польскую запись операции логического умножения), *OutDisj* (помещение в польскую запись операции логического сложения), *ExprFinish* (помещение в польскую запись служебной операции завершения польской записи). На рис. 4 соответствующие фрагменты L-сети изображены с затемнением. В целом алгоритм формирования постфиксной бесскобочной формы записи с бинарными и *n*-арными операциями, в которой последовательность операций соответствует последовательности вычислений, с последующим получением древовидного представления интерпретируемого выражения, рассматривается в [4].

Использование формализма на базе L-сети применительно к решению задач синтаксического анализа позволяет заключить, что сетевые программы характеризуются четкостью и ясностью графики, при этом обеспечено согласованное и недвусмысленное использование символов, исключено их некорректное использование. Таким образом, обеспечивается *понятность* программного обеспечения.

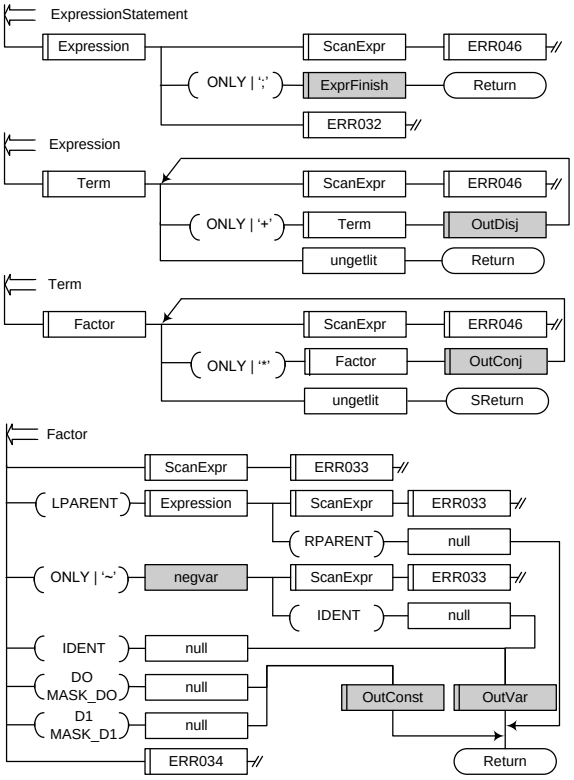

Рис. 4. Синтаксический анализ выражения

Поскольку в качестве языка операционных модулей может использоваться любой язык программирования, сетевые программы не зависят от архитектурных особенностей ЭВМ, а, следовательно, обеспечивается *мобильность* программного обеспечения. Использование единой нотации, терминологии и символики определяет *согласованность* программного обеспечения и косвенно влияет на понятность. Особенности технологии проектирования позволяют реализовать сложную структуру разветвлений, характерных для логически сложной задачи, что уменьшает вероятность пропуска возможных вариантов ошибок во входных наборах и совершения ошибок в программе. Эти факторы обеспечивают *надежность* программного обеспечения. Технология L-сети была реализована ее автором на разных вычислительных платформах и показала свою *машинонезависимость*. Одной из важных особенностей анализируемого визуального формализма проектирования программного обеспечения является учет психологических особенностей восприятия человеком сложных программ и опыта проектирования сложных программных комплексов, и учитывает *человеческий фактор* при проектировании программного обеспечения.

Визуальный формализм на базе L-сетей в своей основе развивает управляющие конструкции структурного программирования и во многих случаях позволяет преодолеть присущие процедурной парадигме сложности модификации, сопровождения и масштабирования программных проектов. При этом следует отметить, что опыт

использования формализма на базе L-сети в учебном процессе, позволяет заключить, что даже те люди, которые не знакомы с точной спецификацией исследуемого формализма, могут легко разобраться в алгоритмах решений логически сложных задач проектирования, представленных в форме фрагментов L-сетей.

СПИСОК ЛИТЕРАТУРЫ

1. Колосов В.Г., Мелехин В.Ф. Проектирование узлов и систем автоматики и вычислительной техники.- Л.: Энергоатомиздат, 1983.- 256с.

2. Лекарев М.Ф. Визуальный формализм для разработки программного обеспечения.- СПб.: Санкт-Петербургский гос. техн. ун-т., 1997.- 95с.

3. Лекарев М.Ф. L-сеть в сверхбольшом программном проекте.- СПб.: Санкт-Петербургский гос. техн. ун-т., 2000.- 96с.

4. Лекарев М.Ф., Мелехин В.Ф., Пышкин Е.В. Автоматизированный синтез комбинационных схем на задаваемом наборе логических элементов. - Вычислительные, измерительные и управляющие системы. Труды СПбГТУ №452, 1995, с.193-206

5. Шнейдерман Б. Психология программирования: человеческие факторы в вычислительных и информационных системах / Пер. с англ.- М.: Радио и связь, 1984.- 304с.

Mikhail F. Lekarev, Evgeny V. Pyshkin

Syntax Analyzing and Interpreting Factored Expressions on the Base of Software Development Visual Formalism (L-Net).

The task of syntax analyzing of logical factored expression is researched. The solution may be generalized for the case of arithmetic expressions as well. To implement the syntax control and and computing schema the software development visual formalism proposed by M.F. Lekarev is used [2]. This formalism is oriented to the software with essential logical complexity and allows overcoming some problems of procedural programming paradigm. In certain applications the approach based on L-Net may be good alternative to object-oriented methods or may be combined with them within the framework of multi-paradigm programming models. This approach of software construction was successfully used in some large projects [2-4].