Выбранный язык – C++.

Общая дополнительная часть без раздела типов.

Содержание

[Структура компилятора 2](#_Toc105751672)

[Модуль ввода-вывода 3](#_Toc105751673)

[Проектирование 3](#_Toc105751674)

[Разработка 3](#_Toc105751675)

[Тестирование 4](#_Toc105751676)

[Лексический анализатор 5](#_Toc105751677)

[Проектирование 5](#_Toc105751678)

[Разработка 6](#_Toc105751679)

[Тестирование 8](#_Toc105751680)

[Синтаксический анализатор 9](#_Toc105751681)

[Проектирование 9](#_Toc105751682)

[Разработка 11](#_Toc105751683)

[Тестирование 13](#_Toc105751684)

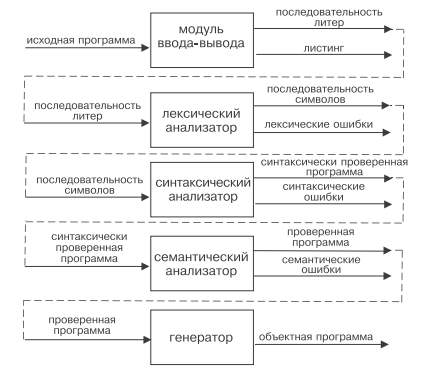
[Семантический анализатор 15](#_Toc105751685)

[Проектирование 15](#_Toc105751686)

[Реализация 16](#_Toc105751687)

[Тестирование 18](#_Toc105751688)

# Структура компилятора

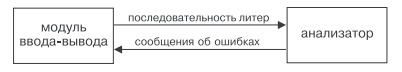


# Модуль ввода-вывода

## Проектирование

Литера – это часть алфавита языка.

Взаимодействие между модулем ввода-вывода и анализатором можно представить в виде схемы:



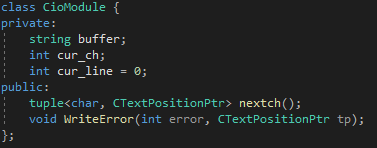
Требования к модулю ввода-вывода:

* Считывает последовательность литер исходной программы и передает ее анализатору
* Формирует листинг

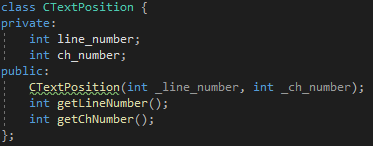
## Разработка

Разработан класс CIOModule, с методами nextch и writeError.

При запросе следующей литеры, метод nextch в случае пустоты буфера считывает из файла новую строку и возвращает следующую литеру с ее позицией в тексте программы.



Для хранения позиции символов в тексте программы разработан класс CTextPosition, хранящий номер строки и позицию символа в ней.



## Тестирование

|  |  |  |  |
| --- | --- | --- | --- |
| Название теста | Суть теста | Ожидаемый результат | Фактический результат |
| Чтение символов | Тестирование метода nextch() | Модуль считывает все литеры программы |  |
| Вывод ошибки | Тестирование метода writeError() | Выводится код ошибки, номер строки и позиция символа |  |

# Лексический анализатор

## Проектирование

Лексический анализатор – это модуль, который формирует символы исходной программы и строит их внутреннее представление.

Лексема – это синтаксическая единица, последовательность допустимых символов языка программирования, имеющая смысл для транслятора.

Токен – это каждая из значимых частей (в виде последовательности символов между разделителями), на которые разбивается текст специальной компьютерной программой.

Требования к лексическому анализатору:

* Формирует символы исходной программы
* Передает другим блокам компилятора следующую информацию: код символа, позицию символа, значение константы, адрес идентификатора в таблице имен.

Все токены можно поделить на три группы:

1. ключевое слово,
2. идентификатор (код ошибки – 2),
3. константа.

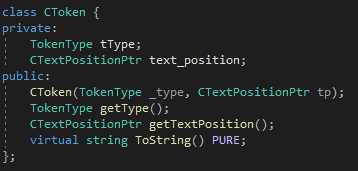
Таблица ключевых слов

|  |  |  |
| --- | --- | --- |
| Ключевое слово | Условное обозначение | Код ошибки |
| var | varSy, |  |
| begin | beginSy, | 17 |
| end | endSy, | 13 |
| and | andSy, |  |
| or | orSy, |  |
| not | notSy, |  |
| = | evenSy | 16 |
| < | lessSy, |  |
| > | moreSy, |  |
| <= | lessEvenSy, |  |
| >= | moreEvenSy, |  |
| <> | notEvenSy, |  |
| := | assignSy, | 51 |
| + | plusSy, |  |
| - | minusSy, |  |
| \* | multSy, |  |
| / | subSy, |  |
| div | divSy |  |
| mod | modSy |  |
| ( | leftBrSy, | 9 |
| ) | rightBrSy, | 4 |
| if | ifSy, | 56 |
| then | thenSy | 52 |
| else | elseSy, |  |
| while | whileSy, |  |
| do | doSy, | 54 |
| “ | quotSy, |  |
| : | colonSy, | 5 |
| ; | semiColonSy, | 14 |
| . | dotSy | 61 |
| , | commaSy | 20 |
| program | programSy | 3 |

## Разработка

Ключевые слова хранятся в структуре map, где ключ – это строковое представление ключевого слова, значение – его код. Ключевым словам, имеющим собственные коды ошибок присвоены соответствующие значения.

Для хранения токенов разработан класс CToken, содержащий тип токена и его позицию в тексте программы.



Для различных типов токенов заведены собственные классы, наследующие от класса CToken и содержащие:

* Значение идентификатора



* Код ключевого слова



* Значение константы

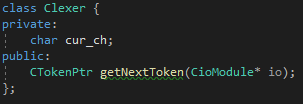








Лексический анализатор был реализован в качестве класса CLexer с методом getNextToken, формирующим и возвращающим следующий токен.



Формирование токенов методом getNextToken:

* Если модуль ввода/вывода вернул null, возвращаем null
* Пропускаем все пробелы и переносы строк
* Если встретилась буква – считать до конца последовательность букв и цифр.
  + Если получилось true или false, возвращаем токен булевой константы.
  + Если полученная строка нашлась в таблице ключевых слов, возвращаем токен ключевого слова.
  + Иначе возвращаем токен идентификатора.
* Если встретилась цифра – считать до конца целое или вещественное число, вернуть токен соответствующей константы.
* В остальных случаях в зависимости от полученных литер формируем токены не буквенных ключевых слов.

## Тестирование

|  |  |  |
| --- | --- | --- |
| Название теста | Суть теста | Ожидаемый результат |
| Формирование лексем | Тестирование метода getNextToken() | Лексический анализатор правильно формирует лексемы и определяет их типы |

Файл whileif.pas

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| program | 3 | Boolean | ident | b | ident | begin | 17 | 1 | const | ; | 14 |
| whileiftest | ident | ; | 14 | >= | 69 | c | ident | end | 13 | d | ident |
| ; | 14 | begin | 17 | c | ident | := | 51 | ; | 14 | := | 51 |
| var | 62 | r | ident | - | 72 | 2 | const | s | ident | true | const |
| s | ident | := | 51 | a | ident | ; | 14 | := | 51 | end | 13 |
| : | 5 | 0.330000 | const | then | 52 | if | 56 | A | const | . | 61 |
| char | ident | ; | 14 | while | 78 | b | ident | ; | 14 |  |  |
| ; | 14 | a | ident | a | ident | > | 67 | r | ident |  |  |
| r | ident | := | 51 | > | 67 | 1 | const | := | 51 |  |  |
| : | 5 | 1 | const | 0 | const | \* | 73 | 1 | const |  |  |
| real | ident | ; | 14 | do | 54 | 2 | const | + | 71 |  |  |
| ; | 14 | b | ident | b | ident | then | 52 | 2 | const |  |  |
| a | ident | := | 51 | := | 51 | a | ident | \* | 73 |  |  |
| , | 20 | 24 | const | b | ident | := | 51 | ( | 9 |  |  |
| b | ident | ; | 14 | + | 71 | a | ident | 3 | const |  |  |
| , | 20 | c | ident | 1 | const | - | 72 | - | 72 |  |  |
| c | ident | := | 51 | ; | 14 | 1 | const | 42 | const |  |  |
| : | 5 | a | ident | while | 78 | else | 77 | ) | 4 |  |  |
| integer | ident | + | 71 | b | ident | a | ident | / | 74 |  |  |
| ; | 14 | b | ident | <> | 70 | := | 51 | 5 | const |  |  |
| d | ident | ; | 14 | 2 | const | a | ident | + | 71 |  |  |
| : | 5 | if | 56 | do | 54 | + | 71 | 6 | const |  |  |

# Синтаксический анализатор

## Проектирование

Синтаксический анализатор – это модуль, который выполняет синтаксический анализ последовательности символов

Требования к синтаксическому анализатору:

* проверяет последовательность токенов на синтаксические ошибки
* нейтрализует синтаксические ошибки

Правила, описывающие синтаксис языка Pascal:

<программа>::=program<имя>;<блок>.

<блок>::=<раздел переменных><раздел операторов>

<раздел переменных>::=var <описание однотипных переменных> ;{<описание однотипных переменных> ; }| < пусто>

<описание однотипных переменных>::=<имя> { , <имя>} : <тип>

<раздел операторов>::=<составной оператор>

<оператор>::=<оператор присваивания>|<пустой оператор>|<составной оператор>|<условный оператор>|<цикл с предусловием>

<оператор присваивания>::=<переменная>:=<выражение>

<составной оператор>::=begin<оператор>{;<оператор>}end

<выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение>

<операция отношения>::==|<>|<|<=|>=|>

<простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}

<аддитивная операция>::=+|-|or

<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}

<мультипликативная операция>::=\*|/|div|mod|and

<множитель>::=<переменная>|<константа без знака>| (<выражение>)

<цикл с предусловием>::=while<выражение>do<оператор>

<условный оператор> ::= if <выражение> then <оператор><хвост>

Анализ конструкций T(L) проходит по правилам:

1. если L состоит из единственного символа, то T(L)--> accept(L). При этом функция accept имеет следующую структуру:

*if (сканируемый символ совпадает с ожидаемым) сканировать следующий символ;*

*else сформировать сообщение об ошибке;*

1. если L состоит из единственной синтаксической конструкции (например, <A>), то T(L)--> A, где A — имя функции для правила <A>::=Q
2. если L — последовательность вида: L1 L2 L3 ... Ln, где Li(i=1,n) — символ языка или синтаксическая конструкция, то:

T (L1 L2 ... Ln) --> T (L1)

T (L2)

...

T (Ln)

1. если L содержит несколько альтернатив, т. е. имеет вид Y1|Y2|...|Ym, то необходимо обеспечить анализ подходящей альтернативы
2. если L имеет вид {Z}, то:

T(L)--> while(symbol принадлежит start (Z)) T(Z);

Если Z встречается в правой части правила хотя бы один раз (L есть Z{Z}), то:

T(L)--> T(Z);

while(symbol принадлежит start(Z)) T(Z);

или

T(L)--> do T(Z);

while (symbol принадлежит start (Z));

**Нейтрализация синтаксических ошибок**

Основная идея – после выявления ошибки надо пропустить один или несколько символов, чтобы найти символ, начиная с которого можно возобновить анализ.

Внешние символы – символы, которые по правилам описания языка могут следовать за конструкцией S.

Алгоритм нейтрализации синтаксических ошибок при анализе конструкции:

1. Если текущий символ не принадлежит множеству символов, с которых может начинаться конструкция, то пропускаем символы до нахождения такого или внешнего символа.
2. Если нашли один из начальных символов конструкции:
   1. Анализируем конструкцию,
   2. Пропускаем символы до нахождения внешнего.

## Разработка

Разработан класс CSyntax, содержащий методы для анализа перечисленных выше конструкций и вспомогательные методы для нейтрализации синтаксических ошибок.

|  |  |
| --- | --- |
| Конструкция | Метод в классе CSyntax |
| <программа> | program |
| <блок> | block |
| <раздел переменных> | var\_part |
| <описание однотипных переменных> | var\_declaration |
| <раздел операторов> | statement\_part |
| <оператор> | statement |
| <оператор присваивания> | assign\_statement |
| <составной оператор> | composite\_statement |
| <выражение> | expr |
| <операция отношения> | is\_relation\_op |
| <простое выражение> | simple\_expr |
| <аддитивная операция> | is\_add\_op |
| <слагаемое> | term |
| <мультипликативная операция> | is\_mult\_op |
| <множитель> | factor |
| <цикл с предусловием> | while\_statement |
| <условный оператор> | if\_statement |

Структура методов соответствует набору правил, указанных в разделе проектирования синтаксического анализа.

Для нейтрализации ошибок данные методы были дополнены операциями пропуска символов до нахождения начального символа конструкции в начале метода и операциями пропуска символов до нахождения первого внешнего из списка, полученного методом в качестве аргумента, в конце.

Проверка принадлежности символа множеству реализована с помощью метода symbol, возвращающего код текущего символа и метода belong, проверяющего принадлежность символа заданному множеству.

Пропуск символов реализован методами skip\_to.

|  |
| --- |
| class CSyntax {  private:  CTokenPtr curToken;  public:  int symbol();  bool belong(int symbol, vector<int> starters);  void skip\_to(vector<int> followers);  void skip\_to(vector<int> starters, vector<int> followers);  void skip\_to\_followers(vector<int> followers);  void accept(KeyWords keyword);  void accept(TokenType tt);  void var\_declaration();  void var\_part();  bool is\_add\_op(CTokenPtr token);  bool is\_mult\_op(CTokenPtr token);  void factor();  void term();  void simple\_expr();  bool is\_relation\_op(CTokenPtr token);  void expr();  void assign\_statement();  void statement();  void composite\_statement();  void while\_statement ();  void if\_statement();  void statement\_part();  void block();  void program();  }; |

Дополнительные коды ошибок, выводящиеся при нейтрализации

|  |  |
| --- | --- |
| Код ошибки | Текст ошибки |
| 6 | запрещенный символ |
| 183 | запрещенная в данном контексте операция |
| 1337 | [ошибка в выражении] |

## Тестирование

|  |  |  |  |
| --- | --- | --- | --- |
| Название теста | Суть теста | Ожидаемый результат | Фактический результат |
| Файл assign.pas | Код без ошибок с операторами присваивания | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл if.pas | Код без ошибок с условными операторами | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл while.pas | Код без ошибок с циклами с предусловием | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл whileif.pas | Код без ошибок со всеми сложными операторами | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл syntaxerrors.pas | 1 pogram syntaxerrors; (неверно записано слово program) | *Ошибка 3*  должно быть служебное слово PROGRAM | \*\*\*\*ERROR 3\*\*\*\* line: 1 symbol position: 1 |
| 4 a,b c:integer; (пропущена запятая) | *Ошибка 5*  ожидалось двоеточие  *Ошибка 6*  запрещенный символ | \*\*\*\*ERROR 5\*\*\*\* line: 4 symbol position: 5  \*\*\*\*ERROR 6\*\*\*\* line: 4 symbol position: 6 |
| 5 d:;  (пропущен идентификатор типа) | *Ошибка 2*  должно идти имя | \*\*\*\*ERROR 2\*\*\*\* line: 5 symbol position: 3 |
| 7 r:=;  (пропущено выражение) | *Ошибка 1337*  [ошибка в выражении] | \*\*\*\*ERROR 1337\*\*\*\* line: 7 symbol position: 4 |
| 9 b:24;  (неверный оператор присваивания) | *Ошибка 51*  должен идти символ ‘:=’  *Ошибка 1337*  [ошибка в выражении] | \*\*\*\*ERROR 51\*\*\*\* line: 9 symbol position: 2  \*\*\*\*ERROR 1337\*\*\*\* line: 9 symbol position: 2 |
| 11 if b >= c-a  12 b:=b+1;  (пропущено then) | *Ошибка 52*  должен идти символ THEN | \*\*\*\*ERROR 52\*\*\*\* line: 12 symbol position: 1 |
| 13 while b <> 2  14 begin  (пропущено do) | *Ошибка 54*  должен идти символ DO | \*\*\*\*ERROR 54\*\*\*\* line: 14 symbol position: 1 |
| 21 . | *Ошибка 13*  должно идти слово END | \*\*\*\*ERROR 13\*\*\*\* line: 21 symbol position: 1 |

# Семантический анализатор

## Проектирование

Семантический анализатор – это модуль, который выполняет анализ контекстных условий программы.

Определяющим является вхождение идентификатора в конструкцию, описывающую этот идентификатор.

Прикладным называется вхождение идентификатора в конструкцию, которая не является его описанием.

Контекстные условия, которые необходимо проверять при анализе программ:

1. В любой области действия без внутренних по отношению к ней областей действия никакой идентификатор не может быть описан более одного раза.
2. Каждому прикладному вхождению нестандартного идентификатора (стандартные идентификаторы — integer, boolean, real, char, true, false и др.) должно найтись соответствующее ему определяющее вхождение. Правило поиска определяющих вхождений называется алгоритмом идентификации, который заключается в следующем:
   1. рассмотреть самую внутреннюю область действия, содержащую данное прикладное вхождение;
   2. найти определяющее вхождение в рассматриваемой области действия. Если оно найдено, то процедура идентификации закончена, и данное прикладное вхождение идентификатора удовлетворяет контекстному условию. В противном случае — перейти к шагу (3);
   3. найти область действия, непосредственно объемлющую только что рассмотренную. Если такая область найдена, то перейти на шаг (2). В противном случае процедура идентификации закончена, и так как определяющее вхождение не найдено, то данное прикладное вхождение идентификатора не удовлетворяет контекстному условию.
3. Контекстные условия предполагают также проверку соответствия типов величин, входящих в синтаксические конструкции программ; соответствия количества индексов у переменных с индексами и размерности соответствующих массивов и др.

Таблица идентификаторов будет содержать идентификатор и ссылку на соответствующую ему переменную.

Таблица типов (содержит только стандартные типы)

|  |  |
| --- | --- |
| Тип | Условное обозначение |
| integer | vtInt |
| real | vtReal |
| char | vtChar |
| Boolean | vtBoolean |
| Неопределенный тип | vtUndefined |

**Нейтрализация семантических ошибок**

Если при обработке прикладного вхождения идентификатора не нашлось соответствующего определяющего вхождения, то необходимо сформировать сообщение об ошибке и занести в ТИ неопределенный идентификатор с атрибутами, полученными из контекста.

Повторные сообщения могут возникать также из-за неправильного описания идентификатора. В этом случае в ТИ заносятся описатели всех прикладных вхождений этого идентификатора. Если использование идентификатора не соответствует описанию, то нужно просмотреть все его описания в ТИ, и если ранее встречалась такая некорректность, то формировать сообщение не следует; в противном случае в таблицу ошибок заносится информация о встретившейся ошибке, а в ТИ добавляется элемент с информацией о новом некорректном использовании идентификатора.

## Реализация

Поскольку в работе рассматривается только одна область видимости, была создана только одна таблица идентификаторов и одна таблица типов.

Таблица типов реализована с помощью структуры map, где ключ – строка с именем типа, значение – его значение в перечислении.

Для хранения типов переменных и констант класс CToken был дополнен полем vType.

Таблица идентификаторов реализована с помощью структуры map, где ключ – идентификатор, значение – ссылка на класс CIdentToken.

Для выполнения семантического анализа некоторые методы синтаксического анализатора были дополнены:

* var\_declaration: все новые идентификаторы заносятся во временный список, в конце им присваивается указанный тип, и они заносятся в ТИ. В случае повторного описания идентификатора или отсутствия типа среди стандартных выдается ошибка.
* factor, term, simple\_expr, expr: теперь возвращают тип данных и проверяют типы операндов на конфликты. Если типы операндов конфликтуют, выдается ошибка и возвращается неопределенный тип.
* factor: если множитель – необъявленный идентификатор, то выдается ошибка и он заносится в таблицу идентификаторов.
* assign\_statement: если идентификатор, которому присваивается значение не объявлен, то выдается ошибка и он заносится в таблицу идентификаторов. Если тип переменной не соответствует типу выражения, выдается ошибка.
* if\_statement и while\_statement: если выражение вернуло тип не Boolean, то выдается ошибка.

Новые вспомогательные методы класса CSyntax:

* is\_type – проверка на наличие идентификатора в таблице типов
* exists\_ident – проверка на наличие идентификатора в таблице идентификаторов
* exists\_token – проверка на наличие идентификатора определенного типа в таблице идентификаторов

Новые коды ошибок

|  |  |
| --- | --- |
| Код ошибки | Текст ошибки |
| 10 | Ошибка в типе |
| 101 | Имя описано повторно |
| 104 | Имя не описано |
| 144 | Недопустимый тип выражения |
| 145 | Конфликт типов |

## Тестирование

|  |  |  |  |
| --- | --- | --- | --- |
| Название теста | Суть теста | Ожидаемый результат | Фактический результат |
| Файл assign.pas | Код без ошибок с операторами присваивания | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл if.pas | Код без ошибок с условными операторами | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл while.pas | Код без ошибок с циклами с предусловием | Не найдено ни одной ошибки | Совпадает с ожидаемым |
| Файл whileif.pas | Код без ошибок со всеми сложными операторами | Не найдено ни одной ошибки | Совпадает с ожидаемым |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |