|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ В РФ | | | | | | | | | |
|  | | | |  | | |  | | |
|  | | ФГБОУ ВО «Пермский государственный национальный исследовательский университет» | | | | | |  | |
|  | | | |  | | |  | | |
|  | | Отчёт  по дисциплине «Формальные грамматики и методы трансляции» | | | | | |  | |
|  | | |  | | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-4-18  Колесников А.С. \_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 | | | |  | Проверил  преподаватель кафедры МОВС  Пономарев Ф.А.\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 | | |  |
|  | | | | | | | | | |
| Пермь 2022 | | | | | | | | | |

Оглавление

[Оглавление 2](#_Toc100514180)

[1. Структура компилятора 3](#_Toc100514181)

[2. Модуль ввода-вывода 5](#_Toc100514182)

[2.1. Проектирование 5](#_Toc100514183)

[2.2. Тестирование 5](#_Toc100514184)

[3. Лексический анализатор 7](#_Toc100514185)

[3.1. Проектирование 7](#_Toc100514186)

[3.2. Тестирование 9](#_Toc100514187)

[3.2.1. Первая программа 9](#_Toc100514188)

[3.2.2. Вторая программа 11](#_Toc100514189)

[3.2.3. Третья программа 14](#_Toc100514190)

[4. Синтаксический анализатор 17](#_Toc100514191)

[4.1. Теория 17](#_Toc100514192)

[4.2. Проектирование 25](#_Toc100514193)

[4.3. Тестирование 26](#_Toc100514194)

[4.3.1. Первая программа 26](#_Toc100514195)

[4.3.2. Вторая программа 28](#_Toc100514196)

[4.3.3. Третья программа 30](#_Toc100514197)

[5. Семантический анализатор 34](#_Toc100514198)

[5.1. Теория 34](#_Toc100514199)

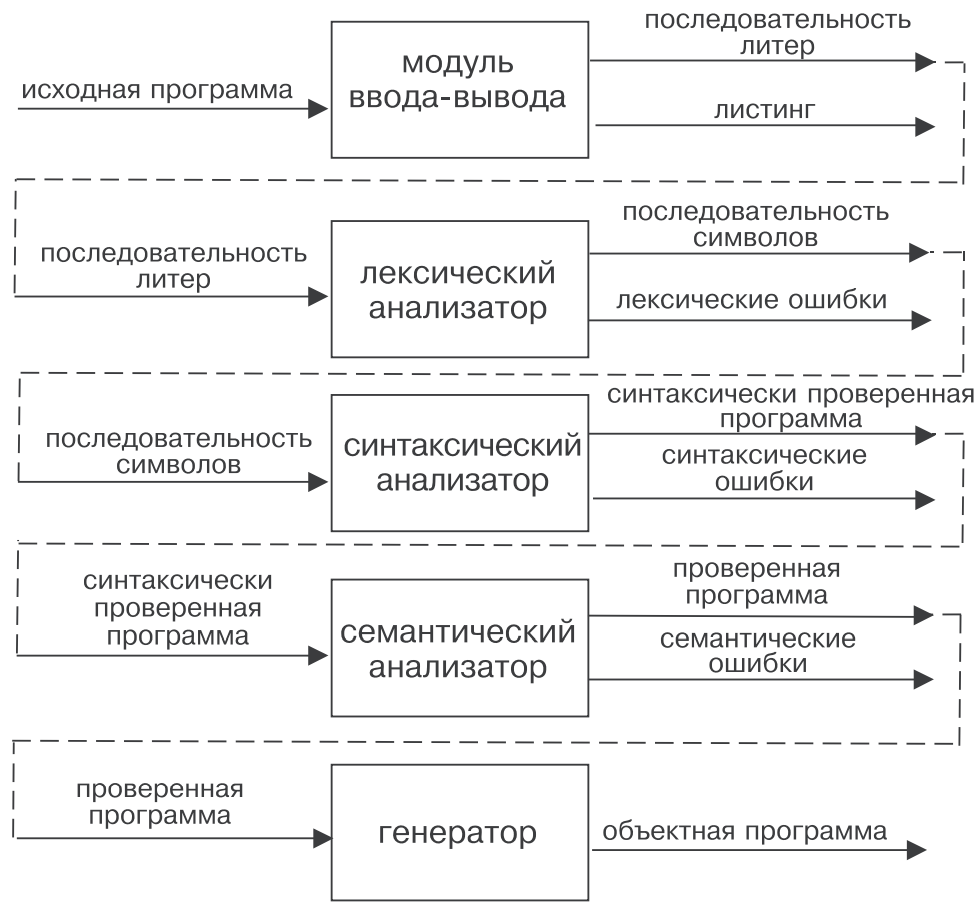
[5.2. Проектирование 38](#_Toc100514200)

[5.3. Тестирование 38](#_Toc100514201)

1. Структура компилятора

Компилятор – это программа которая переводит программу на языке высокого уровня в эквивалентную программу на другом языке. Обычно компилятор также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках.

Разработка программного обеспечения подразумевает модульность и хорошую структурированность программ. Учитывая это, представим компилятор как совокупность логически взаимосвязанных модулей (рис. 1).



Рисунок

1 – Структура компилятора

Модуль ввода вывода отвечает за все действия, связанные с чтением и записью данных. Для каждого типа устройства может существовать свой собственный модуль ввода-вывода.

Лексический анализатор осуществляет простой лексический анализ исходной программы. На вход он получает последовательность литер, а отдает последовательность лексем (токенов).

Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам.

Семантический анализатор проверяет не нарушены ли неформальные правила описания языка.

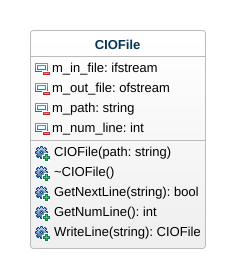
Генератор отображает независимое промежуточное представление исходной программы на реальную ЭВМ, поэтому должен переписываться для каждой новой машины.

1. Модуль ввода-вывода
   1. Проектирование

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их лексическому анализатору, который проверяет, удовлетворят ли эта последовательность литер правилам описания языка, и формирует в случае необходимости сообщения об ошибках.

Будем считать, что в результате очередного обращения к модулю ввода-вывода анализатор получает текущую строку.

Для печати сообщения об ошибке, лексический анализатор должен передать модулю ввода-вывода причину и местоположение ошибки в строке.



Рисунок

2 – Структура класса модуля ввода-вывода

* 1. Тестирование

Проверим работоспособность считывание следующего файла:

var

a,b,c:integer;

name:string;

x:real;

flag:boolean;

type

myint = integer;

str=string;

begin

//some comment

a:= 10 mod 2;

b := 6 div 4;

x:= 2.1;

name :='some string';

flag:= True;

if (a = 2) and (b < 1) then

if (x >= 0) or (name <> 'some str') then

c := 2 + 4

else

name := 'text';

while 2 >= 5 do

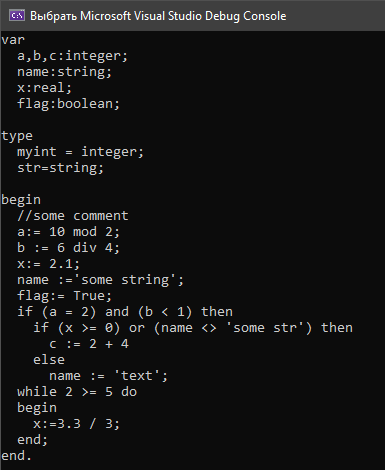
begin

x:=3.3 / 3;

end;

end.

Результат представлен на рисунке:



Рисунок

3 – Результат работы модуля ввода-вывода

1. Лексический анализатор
   1. Проектирование

Лексический анализатор формирует токены исходной программы и строит их внутреннее представление, а также распознает и исключает комментарии, которые не нужны для дальнейшей трансляции.

Лексическим анализатор запрашивает у модуля ввода-вывода следующую строку и обрабатывает ее, выделяя все лексемы. Если же файл был обработан полностью, то модуль ввода-вывода, уведомляет об этом лексический анализатор, и последний завершает работу.

Рассмотрим внутреннее представление токенов.

Лексический анализатор преобразует последовательности литер переменной длины в токены. Всего их три типа.

Константы представляют определенный тип данных. В нашем случае могут быть 4 вида константных токенов: integer, real, boolean, string.

Ключевые слова представляют последовательности символов зарезервированные для использования при написании программы. Они представлены в виде перечислений (enum).

Пользовательские идентификаторы — это последовательности символов, содержащие цифры, буквы и знаки подчеркивания, которые может использовать пользователь для обозначения переменных, функции, собственных типов и т.д.

При обнаружении ошибки на этапе анализа программы необходимо напечатать соответствующее сообщение и указать место, где обнаружена ошибка. Поэтому лексический анализатор должен формировать координаты каждого символа, а именно: номер строки исходной программы и позицию символа (позицию первой литеры символа) в этой строке.

На рисунке 3 представлена диаграмма классов, которые использует лексический анализатор для построения внутренней модели.



Рисунок

4 – Диаграмма классов лексического анализатора

Класс CToken базовый абстрактный класс для всех токенов.

Перечисление ETokenType содержит в себе все типы возвращаемых токенов классом CToken.

Класс CKeywordToken производный от CToken и отвечает за ключевые слова. Содержит тип ключевого слова, который представлен перечислением EKeyWords.

Класс CIdentToken производный от CToken и отвечает за хранения пользовательских идентификаторов.

Класс CConstToken производный от CToken и отвечает за хранения констант. Содержит указатель на CVariant.

Класс CVariant базовый абстрактный класс для типов данных. Хранит тип данных, представленный в виде перечисления EVariantType.

Класс CBooleanVariant отвечает за хранение логических значений типа boolean.

Класс CStringVariant отвечает за хранение строковых значений типа string.

Класс CRealVariant отвечает за хранение чисел с плавающей точкой типа real.

Класс CIntVariant отвечает за хранение целочисленных значений типа integer.

Класс CLexer отвечает за получение токенов по средством функции GetNextToken, которая возвращает указатель на экземпляр класса CToken, а также уведомляет об ошибках лексического уровня. Для считывания файла и выдачи листинга, данный класс использует модуль ввода-вывода CIOFile.

* 1. Тестирование

Для того чтоб протестировать работу лексического анализатора проверим его на нескольких программах.

Первая программа

Ниже представлена текст программы, которую нужно обработать:

var

a,b,c:integer;

name:string;

x:real;

flag:boolean;

begin

// some comment here

a:= 10 - 3;

b := 2 \*(3 + 4);

c:=a\*b;

x:= 5.65 /2.1;

{ and

here

}

name :='some string';

flag:= True;

flag := False;

(\* the end of

the program \*)

end.

В итоге были получены следующие результаты

1: KEYWORD: var

2: IDENT: a

3: KEYWORD: ,

4: IDENT: b

5: KEYWORD: ,

6: IDENT: c

7: KEYWORD: :

8: KEYWORD: integer

9: KEYWORD: ;

10: IDENT: name

11: KEYWORD: :

12: KEYWORD: string

13: KEYWORD: ;

14: IDENT: x

15: KEYWORD: :

16: KEYWORD: real

17: KEYWORD: ;

18: IDENT: flag

19: KEYWORD: :

20: KEYWORD: boolean

21: KEYWORD: ;

22: KEYWORD: begin

23: IDENT: a

24: KEYWORD: :=

25: CONST: integer: 10

26: KEYWORD: -

27: CONST: integer: 3

28: KEYWORD: ;

29: IDENT: b

30: KEYWORD: :=

31: CONST: integer: 2

32: KEYWORD: \*

33: KEYWORD: (

34: CONST: integer: 3

35: KEYWORD: +

36: CONST: integer: 4

37: KEYWORD: )

38: KEYWORD: ;

39: IDENT: c

40: KEYWORD: :=

41: IDENT: a

42: KEYWORD: \*

43: IDENT: b

44: KEYWORD: ;

45: IDENT: x

46: KEYWORD: :=

47: CONST: real: 5.650000

48: KEYWORD: /

49: CONST: real: 2.100000

50: KEYWORD: ;

51: IDENT: name

52: KEYWORD: :=

53: CONST: string: some string

54: KEYWORD: ;

55: IDENT: flag

56: KEYWORD: :=

57: CONST: boolean: True

58: KEYWORD: ;

59: IDENT: flag

60: KEYWORD: :=

61: CONST: boolean: False

62: KEYWORD: ;

63: KEYWORD: end

64: KEYWORD: .

Вторая программа

Ниже представлена текст программы, которую нужно обработать:

var

a,b,c:integer;

name:string;

x:real;

flag:boolean;

type

myint = integer;

str=string;

begin

//some comment

a:= 10 mod 2;

b := 6 div 4;

x:= 2.1;

name :='some string';

flag:= True;

if (a = 2) and (b < 1) then

if (x >= 0) or (name <> 'some str') then

c := 2 + 4

else

name := 'text';

while 2 >= 5 do

begin

x:=3.3 / 3;

end;

end.

В итоге были получены следующие результаты

1: KEYWORD: var

2: IDENT: a

3: KEYWORD: ,

4: IDENT: b

5: KEYWORD: ,

6: IDENT: c

7: KEYWORD: :

8: KEYWORD: integer

9: KEYWORD: ;

10: IDENT: name

11: KEYWORD: :

12: KEYWORD: string

13: KEYWORD: ;

14: IDENT: x

15: KEYWORD: :

16: KEYWORD: real

17: KEYWORD: ;

18: IDENT: flag

19: KEYWORD: :

20: KEYWORD: boolean

21: KEYWORD: ;

22: KEYWORD: type

23: IDENT: myint

24: KEYWORD: =

25: KEYWORD: integer

26: KEYWORD: ;

27: IDENT: str

28: KEYWORD: =

29: KEYWORD: string

30: KEYWORD: ;

31: KEYWORD: begin

32: IDENT: a

33: KEYWORD: :=

34: CONST: integer: 10

35: IDENT: mod

36: CONST: integer: 2

37: KEYWORD: ;

38: IDENT: b

39: KEYWORD: :=

40: CONST: integer: 6

41: IDENT: div

42: CONST: integer: 4

43: KEYWORD: ;

44: IDENT: x

45: KEYWORD: :=

46: CONST: real: 2.100000

47: KEYWORD: ;

48: IDENT: name

49: KEYWORD: :=

50: CONST: string: some string

51: KEYWORD: ;

52: IDENT: flag

53: KEYWORD: :=

54: CONST: boolean: True

55: KEYWORD: ;

56: KEYWORD: if

57: KEYWORD: (

58: IDENT: a

59: KEYWORD: =

60: CONST: integer: 2

61: KEYWORD: )

62: KEYWORD: and

63: KEYWORD: (

64: IDENT: b

65: KEYWORD: <

66: CONST: integer: 1

67: KEYWORD: )

68: KEYWORD: then

69: KEYWORD: if

70: KEYWORD: (

71: IDENT: x

72: KEYWORD: >=

73: CONST: integer: 0

74: KEYWORD: )

75: KEYWORD: or

76: KEYWORD: (

77: IDENT: name

78: KEYWORD: <>

79: CONST: string: some str

80: KEYWORD: )

81: KEYWORD: then

82: IDENT: c

83: KEYWORD: :=

84: CONST: integer: 2

85: KEYWORD: +

86: CONST: integer: 4

87: KEYWORD: else

88: IDENT: name

89: KEYWORD: :=

90: CONST: string: text

91: KEYWORD: ;

92: KEYWORD: while

93: CONST: integer: 2

94: KEYWORD: >=

95: CONST: integer: 5

96: KEYWORD: do

97: KEYWORD: begin

98: IDENT: x

99: KEYWORD: :=

100: CONST: real: 3.300000

101: KEYWORD: /

102: CONST: integer: 3

103: KEYWORD: ;

104: KEYWORD: end

105: KEYWORD: ;

106: KEYWORD: end

107: KEYWORD: .

Третья программа

Ниже представлена текст программы, которую нужно обработать:

var x,y,z:real;

a:integer;

b:integer;

function Sum(a,b: real): real;

begin

Result := a + b;

end;

begin

x:=0.5;

y := 1.4005;

z :=Sum(x,y);

for var i := 1 to 10 do

a := a + i;

repeat

b:= b + 1;

until b > 10;

end.

В итоге были получены следующие результаты

1: KEYWORD: var

2: IDENT: x

3: KEYWORD: ,

4: IDENT: y

5: KEYWORD: ,

6: IDENT: z

7: KEYWORD: :

8: KEYWORD: real

9: KEYWORD: ;

10: IDENT: a

11: KEYWORD: :

12: KEYWORD: integer

13: KEYWORD: ;

14: IDENT: b

15: KEYWORD: :

16: KEYWORD: integer

17: KEYWORD: ;

18: KEYWORD: function

19: IDENT: Sum

20: KEYWORD: (

21: IDENT: a

22: KEYWORD: ,

23: IDENT: b

24: KEYWORD: :

25: KEYWORD: real

26: KEYWORD: )

27: KEYWORD: :

28: KEYWORD: real

29: KEYWORD: ;

30: KEYWORD: begin

31: IDENT: Result

32: KEYWORD: :=

33: IDENT: a

34: KEYWORD: +

35: IDENT: b

36: KEYWORD: ;

37: KEYWORD: end

38: KEYWORD: ;

39: KEYWORD: begin

40: IDENT: x

41: KEYWORD: :=

42: CONST: real: 0.500000

43: KEYWORD: ;

44: IDENT: y

45: KEYWORD: :=

46: CONST: real: 1.400500

47: KEYWORD: ;

48: IDENT: z

49: KEYWORD: :=

50: IDENT: Sum

51: KEYWORD: (

52: IDENT: x

53: KEYWORD: ,

54: IDENT: y

55: KEYWORD: )

56: KEYWORD: ;

57: KEYWORD: for

58: KEYWORD: var

59: IDENT: i

60: KEYWORD: :=

61: CONST: integer: 1

62: KEYWORD: to

63: CONST: integer: 10

64: KEYWORD: do

65: IDENT: a

66: KEYWORD: :=

67: IDENT: a

68: KEYWORD: +

69: IDENT: i

70: KEYWORD: ;

71: KEYWORD: repeat

72: IDENT: b

73: KEYWORD: :=

74: IDENT: b

75: KEYWORD: +

76: CONST: integer: 1

77: KEYWORD: ;

78: KEYWORD: until

79: IDENT: b

80: KEYWORD: >

81: CONST: integer: 10

82: KEYWORD: ;

83: KEYWORD: end

84: KEYWORD: .

1. Синтаксический анализатор
   1. Теория

Опишем структуру синтаксического анализатора, а затем дополним его средствами семантического анализа.

В предыдущей главе мы неформально проанализировали правила, описывающие синтаксис символов, и составили соответствующие программные фрагменты сканера. Аналогичный подход может быть использован и при написании синтаксического анализатора.

Все правила, описывающие синтаксис языка, имеют вид:

<S> ::= L

где S — обозначение синтаксической конструкции, а L может состоять из символов языка, синтаксических конструкций и метасимволов «|», «[ ]» и «{ }».

Основная идея написания синтаксического анализатора заключается в следующем.

Для каждого правила необходимо описать функцию, тело которой является результатом некоторого преобразования правой части этого правила:

procs ( ){ T( L ) }

При описании функции, соответствующей синтаксической конструкции S, мы будем придерживаться следующих соглашений:

1. перед обращением к функции переменная cur\_token содержит код первого символа, который должен быть проанализирован этой функцией;
2. функция проверяет, выводима ли входная последовательность символов из S, и в случае ошибки формирует соответствующее сообщение;
3. перед тем как выйти из функции, в переменной cur\_token запоминается код символа, который встретился сразу после анализа конструкции S.

Преобразование Т(L) определяется следующим набором правил:

* **правило 1**: если L состоит из единственного символа, то T(L)--> accept(L) (где --> имеет смысл глагола «есть»). При этом функция accept имеет следующую структуру:  
  if (сканируемый символ совпадает с ожидаемым)  
   сканировать следующий символ;  
  else сформировать сообщение об ошибке;
* **правило 2**: если L состоит из единственной синтаксической конструкции (например, <A>), то T(L)--> A, где A — имя функции для правила <A>::=Q;
* **правило 3**: 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). Здесь преобразование T(Li) выполняется по правилу 1, если Li — символ языка, и по правилу 2, если Li — синтаксическая конструкция.
* **правило 4**: если L содержит несколько альтернатив, т. е. имеет вид Y1|Y2|...|Ym, то необходимо обеспечить анализ подходящей альтернативы.

Введем обозначение: start(Yi) — символы, с которых могут начинаться последовательности символов, получаемые из Yi(i=1,n). Например, конструкция <выражение> может начинаться с одного из следующих символов: «+», «–», «(», «[», «not», «», «», «nil». Следовательно, start (<выражение>)={plus,minus,leftpar,lbracket,notsy,ident,intc,floatc,charc,nilsy}. Тогда:

T (Y1|Y2| … |Ym) -->

switch (symbol)

{

case start(Y1) : T(Y1); break;

case start(Y2) : T(Y2); break;

...

case start(Ym) : T(Ym );

};

Выбор T(Yi) должен однозначно определяться на основании очередного сканируемого символа, ведь в дальнейшем нам предстоит выполнять семантический анализ параллельно с синтаксическим. И если T(Yi) выбирается неоднозначно, то может произойти возврат, и придется уничтожить результаты семантической обработки, что в большинстве случаев сделать достаточно сложно.

**Требование 1**. Если синтаксическое правило имеет вид:

<S>::=Y1|Y2|...|Ym

то множества start(Yi) (i=1,m) попарно не пересекаются, т. е.

start(Yi)\*start(Yj)=<пусто>(i<>j)

где \* — обозначение операции «пересечение множеств». Следует также обратить особое внимание на те правила, у которых одна из альтернатив — <пусто>.

**Требование 2**. Если синтаксическое правило имеет вид:

<S>::=Y1|Y2|...|Ym|<пусто>

то

start(Yi)\*follow(S)=<пусто>(i=1,m)

где follow (S) — множество символов, которые могут встретиться сразу после анализа конструкции S. Например,

follow (<раздел переменных>) = {procsy,funcsy,beginsy}

Если правило содержит в правой части только две альтернативы, то удобно воспользоваться оператором if(...)...; else...;, а если одна из двух альтернатив — , то неполным условным;

* **правило 5**: 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));

**Особенности анализа конструкций языка.**

Некоторые сложности возникают при анализе конструкций <простой тип>, <оператор>, <множитель> и др.:

1) <простой тип>::=<перечислимый тип>|<ограниченный тип>|<имя типа>

2) <перечислимый тип>::=(<имя>{,<имя>})

3) <ограниченный тип>::=<константа>..<константа>

4) <имя типа>::=<имя>

5) <константа>::=<имя константы>|...

6) <имя константы>::=<имя>

7) <оператор>::=<оператор присваивания>|<оператор процедуры>| ...

8) <множитель>::=<переменная>|<вызов функции>|<константа>| ...

В этом случае нарушено *требование* 1 (альтернативы правой части правил 1, 7 и 8 начинаются с символа <имя>). Так как синтаксический и семантический анализ выполняется параллельно, будем предполагать, что семантическая проверка поможет синтаксическому анализатору осуществить выбор подходящей альтернативы.

Теперь рассмотрим анализ условного оператора:

<условный оператор>::= if <выражение> then <оператор> | if <выражение> then <оператор> else <оператор>

Требование 1 здесь нарушено (обе альтернативы начинаются с символа if). Преобразуем это правило в:

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

<хвост>::=else <оператор>|<пусто>

Теперь выполнено требование 1, однако для вложенных условных операторов: if ... then ... if ... then ... else ... нарушается требование 2, так как <хвост> может начинаться с символа else, а за конструкцией <хвост> может следовать символ else вложенного условного оператора. Поэтому примем следующее соглашение: каждый else соответствует ближайшему к нему слева условию. Тогда описание функции синтаксического анализа для конструкции <условный оператор> будет иметь вид:

void ifstatement( )

/\* анализ конструкции \*/

{

accept (ifsy); expression ( );

accept (thensy); statement ( );

if (symbol == elsesy)

{

nextsym ( );

statement ( );

}

}

Правила 1–5 позволяют достаточно быстро написать функции синтаксического анализа, используя формальные правила описания синтаксиса языка программирования (формы Бэкуса—Наура или диаграммы Вирта). Работа анализатора начинается с вызова функции, соответствующей синтаксическому правилу <программа>. Эта функция обращается к другим функциям, которые в свою очередь снова вызывают функции, и т. д. Для анализа некоторых конструкций (например, <оператор> и <выражение>) могут потребоваться рекурсивные вызовы функций.

Таким образом, синтаксический анализатор строится по принципу детерминированного рекурсивного нисходящего анализатора.

Анализатор, созданный по правилам 1–5, правильно анализирует программы, которые удовлетворяют формальным правилам описания языка. Фактически такой анализатор работает верно только до первой ошибки в исходной программе. Чтобы продолжить анализ после обнаружения ошибки, необходимо усовершенствовать алгоритм работы анализатора.

Синтаксический анализатор, рассмотренный в главе 4, проверяет, удовлетворяет ли исходная программа формальным правилам описания языка, и определяет синтаксическую структуру этой программы. Однако этот анализатор правильно функционирует только до обнаружения первой синтаксической ошибки. Если входная последовательность символов содержит неправильную конструкцию, то анализ программы прекращается. Очевидно, что на практике такая организация работы неприемлема: вместо этого компилятор должен сформировать соответствующее сообщение и продолжить анализ, так как за один запуск программы необходимо обнаружить наибольшее количество ошибок.

Дополним анализатор действиями, позволяющими продолжить анализ исходной программы после обнаружения ошибки. Эти действия называются нейтрализацией ошибки.

Основная идея нейтрализации ошибок, которую мы будем рассматривать, состоит в следующем: после выявления ошибки надо пропустить один или несколько символов, чтобы найти символ, начиная с которого можно возобновить анализ. Опишем этот процесс более подробно.

Пусть функция с именем procs соответствует синтаксическому правилу <S>::= L. Множество символов, с которых может начинаться конструкция S, обозначим как starters. В начало функции procs добавим проверку, является ли текущий символ допустимым начальным символом анализируемой конструкции:

if (!belong (symbol, starters))

{ error ( ... ); skipto (starters); }

Здесь функция belong проверяет принадлежность символа symbol множеству starters. Функция skipto пропускает символы входного потока, пока не встретится один из символов множества, указанного в качестве параметра. Как только будет найден искомый символ, начинается анализ конструкции S.

Однако из-за ошибки в исходной программе начальный символ конструкции может вообще не встретиться. Тогда skipto пропустит символы, которые должны быть обработаны функцией, вызвавшей procs. Чтобы избежать этой ситуации, skipto должна просматривать входной текст, пока не будет найден один из символов, который по правилам описания языка может следовать за конструкцией S. Назовем эти символы внешними. Множество внешних символов передается в точку вызова функции procs через параметр.

С учетом сделанных замечаний, добавим формальный параметр в заголовок функции: procs(followers) и перепишем в несколько иной форме:

if (!belong(symbol, starters))

{ еrror(...); skipto2 (starters, followers); }

if (belong (symbol, starters))

/\*анализ конструкции S \*/

T (L);

Здесь функция skipto2 осуществляет проверку на принадлежность символа одному из множеств, указанных в качестве фактических параметров.

Появление внешнего символа в результате обращения к функции skipto2 свидетельствует об ошибке в исходной программе, но этот символ ни в коем случае нельзя пропускать!

Через параметр followers вызывающая функция сообщает, начиная с каких символов она продолжит анализ после окончания работы procs. Поэтому проверку синтаксической правильности конструкции мы завершим следующим образом:

if (!belong(symbol, followers))

{ error(...) skipto(followers); }

Тогда в итоге структура функции procs примет вид:

procs (followers)

{

if (!belong(symbol, starters))

{ error(...); skipto2 (starters, followers); }

if (belong (symbol, starters))

{ /\*анализ конструкции\*/

T(L);

if (!belong(symbol, followers))

{ error(...); skipto(followers); }

}

}

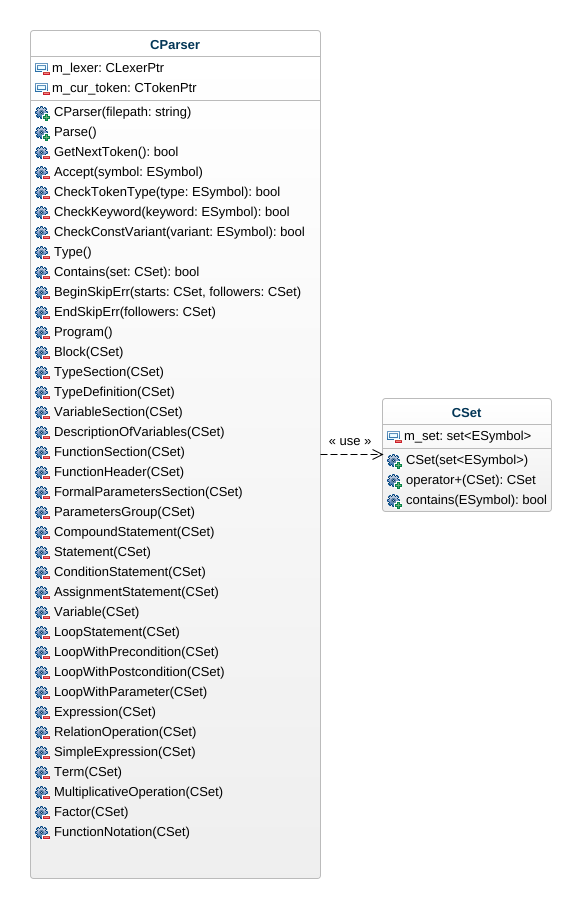
Теперь рассмотрим, как формируется значение фактического параметра при обращении к procs. Ясно, что это значение включает в себя множество символов, которые анализатор ожидает сразу после анализа S. Однако в некорректной программе эти символы могут быть пропущены. Следовательно, к значению фактического параметра необходимо добавить множество символов, которые получила вызывающая функция в качестве параметра.

Таким образом, если функция синтаксического анализа обнаруживает ошибку, то она не сообщает о случившемся вызвавшей ее функции, а продолжает просмотр входного текста до того места, откуда можно возобновить анализ. Внешние символы, передаваемые в качестве параметров, называют символами возобновления.

Компилятор, обеспечивающий выдачу сообщений об ошибках и нейтрализацию ошибок, имеет, как следствие, несколько больший объем, но это вполне компенсируется повышением производительности труда пользователя.

Для хранения множеств символов starters и followers удобно воспользоваться множествами.

* 1. Проектирование



Рисунок

5 – Структура синтаксического анализатора

Класс CParser изображенный на рисунке 5, отвечает за синтаксический анализ программы. Он имеет два приватный поля: m\_lexer, которое хранит объект лексического анализатора, и cur\_token, которое хранит текущий токен. При создании объекта данного класса необходимо указать путь до файла с кодом. Функция Parse() запускает процесс синтаксического анализа.

* 1. Тестирование

Первая программа

Сначала проверим наш синтаксический анализатор на программе, которая будет содержать только объявление переменных, операцию присваивания и арифметические операции.

program myprog;

var

a,b,c:integer;

name:string;

x:real;

flag:boolean;

begin

// some comment here

a:= 10 - 3;

b := 2 \*(3 + 4);

c:=a\*b;

x:= 5.65 /2.1;

{ and

here

}

name :='some string';

flag:= True;

flag := False;

(\* the end of

the program \*)

end.

Данная программа является синтаксически верной, следовательно, никаких ошибок синтаксический анализатор не обнаружил.

Теперь немного изменим код, добавив несколько опечаток:

program myprog;

var

a,b,:integer;

3\*name:string;

x y:tifdfy;

flag:boolean;

begin

// some comment here

a dfdd := 10 - 3;

b := \*2 \*(3 + 4);

c:= mod a\*b;

x:= 5.65 /2.1 fddfd;

{ and

here

}

name :='some string';

flag:= True;

flag = False and 423;

(\* the end of

the program \*)

end.

Проанализировав такую программу синтаксический анализатор выдал нам следующий результат:

program myprog;

var

a,b,:integer;

^ ERROR

Expected token is not type of identificator

^ ERROR

Unexpected token

3\*name:string;

^ ERROR

Unexpected token

x y:tifdfy;

flag:boolean;

begin

// some comment here

a dfdd := 10 - 3;

^ ERROR

Unexpected token

b := \*2 \*(3 + 4);

^ ERROR

Unexpected token

c:= mod a\*b;

^ ERROR

Unexpected token

x:= 5.65 /2.1 fddfd;

^ ERROR

Unexpected token

{ and

here

}

name :='some string';

flag:= True;

flag = False and 423;

^ ERROR

Unexpected token

(\* the end of

the program \*)

end.

^ ERROR

Unexpected token

Вторая программа

Теперь же задействуем более сложные конструкции. Здесь уже есть раздел типов, операции mod и div, а также условия и цикл с предусловием.

program myprog;

type

myint = integer;

str=string;

var

a,b,c:integer;

name:string;

x:real;

flag:boolean;

begin

//some comment

a:= 10 mod 2;

b := 6 div 4;

x:= 2.1;

name :='some string';

flag:= True;

if (a = 2) and (b < 1) then

if (x >= 0) or (name <> 'some str') then

c := 2 + 4

else

name := 'text';

while 2 >= 5 do

begin

x:=3.3 / 3;

end;

end.

Данная программа является синтаксически верной, следовательно, никаких ошибок синтаксический анализатор не обнаружил.

Теперь немного изменим код, добавив несколько опечаток:

program myprog;

type

myint := integer;

str=type;

1/3mytype=type;

var

a,div,c:integer;

name:string;

x:real;

flag:boolean;

begin

//some comment

a:= 10 mod 2;

b := +6 div 4;

x:= 2.1 dfdf;

name ='some string';

flag:= True;

if (a := 2) and (b < 1) then

if (x >= 0) or fdf (name <> 'some str') then dfdf

c := 2 + 4

else

name := 'text';

dfdf while 2 >= 5 dfd do

begin sdaffd

\*\*3\*\*x:=3.3 / 3;

end dfdfs;

end.

Проанализировав такую программу синтаксический анализатор выдал нам следующий результат:

program myprog;

type

myint := integer;

str=type;

^ ERROR

Unexpected token

1/3mytype=type;

^ ERROR

Unexpected token

var

a,div,c:integer;

^ ERROR

Expected token is not type of identificator

name:string;

x:real;

flag:boolean;

begin

//some comment

a:= 10 mod 2;

b := +6 div 4;

x:= 2.1 dfdf;

^ ERROR

Unexpected token

name ='some string';

^ ERROR

Unexpected token

flag:= True;

^ ERROR

Unexpected token

if (a := 2) and (b < 1) then

^ ERROR

Unexpected token

if (x >= 0) or fdf (name <> 'some str') then dfdf

^ ERROR

Unexpected token

^ ERROR

Unexpected token

c := 2 + 4

^ ERROR

Unexpected token

else

name := 'text';

dfdf while 2 >= 5 dfd do

^ ERROR

Unexpected token

begin sdaffd

\*\*3\*\*x:=3.3 / 3;

end dfdfs;

^ ERROR

Unexpected token

end.

Третья программа

Еще более усложним программу, добавив раздел функций, а также использовав цикл с постусловием и с параметром.

program myprog;

var x,y,z:real;

a:integer;

b,i:integer;

function Sum(a,b: real): real;

var

in\_x,in\_b,i:integer;

in\_z:real;

flag:boolean;

begin

for i := 1 to 10 do

a := a + i;

in\_x := a div 3;

in\_b:= in\_x + 32;

in\_z:= in\_b / 2.0;

Result := in\_z + in\_b;

end;

function Func(a,b: real): real;

var

in\_a,i:integer;

in\_z:real;

begin

for i := 1 to 10 do

in\_a := a + i;

in\_a := a div 3;

in\_z:= b / 2.0;

Result := in\_a / in\_z;

end;

begin

x:=0.5;

y := 1.4005;

//z :=Sum(x,y);

for i := 1 to 10 do

a := a + i;

repeat

b:= b + 1;

until b > 10;

end.

Данная программа является синтаксически верной, следовательно, никаких ошибок синтаксический анализатор не обнаружил.

Теперь немного изменим код, добавив несколько опечаток:

program myprog;

type

myint := integer;

str=type;

1/3mytype=type;

var

a,div,c:integer;

name:string;

x:real;

flag:boolean;

begin

//some comment

a:= 10 mod 2;

b := +6 div 4;

x:= 2.1 dfdf;

name ='some string';

flag:= True;

if (a := 2) and (b < 1) then

if (x >= 0) or fdf (name <> 'some str') then dfdf

c := 2 + 4

else

name := 'text';

dfdf while 2 >= 5 dfd do

begin sdaffd

\*\*3\*\*x:=3.3 / 3;

end dfdfs;

end.

Проанализировав такую программу синтаксический анализатор выдал нам следующий результат:

program myprog;

var x,y,z:real;

a,:integer;

^ ERROR

Expected token is not type of identificator

^ ERROR

Unexpected token

b:=integer;

function Sum(a,b: real): tuple;

^ ERROR

Unexpected token

var

in\_x,in\_b dfdfd :integer;

^ ERROR

Expected token is :

^ ERROR

Unexpected token

in\_z: fdf real;

^ ERROR

Unexpected token

flag:boolean;

begin

for i = 1 to 10 do

a := a + i;

in\_x := a div 3;

in\_b:= in\_x ++ 32;

^ ERROR

Unexpected token

in\_z:= in\_b // 2.0;

Result := -in\_z + in\_b;

^ ERROR

Unexpected token

end;

^ ERROR

Unexpected token

function +(a,b: real): real;

^ ERROR

Unexpected token

^ ERROR

Unexpected token

^ ERROR

Unexpected token

varr

^ ERROR

Unexpected token

in\_a,i:integer;

in\_z:real;

begin

for i = 1 to 10 do

in\_a := a + i;

in\_a += a div 3;

^ ERROR

Unexpected token

in\_z:= b // 2.0;

^ ERROR

Unexpected token

Result := in\_a / in\_z;

end;

^ ERROR

Unexpected token

begin

x:=fdfdf 0.5;

^ ERROR

Unexpected token

and y := 1.4005;

^ ERROR

Unexpected token

//z :=Sum(x,y);

for dfdf var i := sfd to 10 fdf do adf

^ ERROR

Unexpected token

^ ERROR

Unexpected token

a := a + i;

^ ERROR

Unexpected token

repeat dfsf

b:= b + 1 fdfd;

^ ERROR

Unexpected token

^ ERROR

Unexpected token

until b := 10;

^ ERROR

Unexpected token

^ ERROR

Unexpected token

end.

^ ERROR

Unexpected token

1. Семантический анализатор
   1. Теория

**Контекстные условия**

Формальные правила описания синтаксиса языка программирования (формы Бэкуса—Наура, а также синтаксические диаграммы) служат основой для построения синтаксического анализатора, однако они дают неполное определение языка. Язык определяется с помощью формальных и неформальных описаний. Синтаксические правила языка программирования, которые задаются с помощью естественного языка (неформально), называются контекстными условиями.

Наличие контекстных условий в языке Паскаль связано с его следующими особенностями:

1. идентификаторы используются для именования различных конструкций — типов, констант, переменных, процедур, функций. Как правило, по виду идентификатора нельзя определить способ его использования;
2. в программах один и тот же идентификатор может использоваться для обозначения различных типов. Поэтому при проверке синтаксической правильности программы необходимо знать типы значений, именуемых соответствующими идентификаторами;
3. идентификаторы, описанные в некоторой области действия, могут использоваться только в соответствии с описанием внутри этой области. Области действия идентификаторов введены в языки программирования для предоставления программисту возможности управления распределением памяти компьютера (области действия в языке Паскаль: программа, процедуры, функции).

Сформулируем 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.

Для дальнейшего изложения нам потребуются следующие определения:

* **определяющим** является вхождение идентификатора в конструкцию, описывающую этот идентификатор;
* **прикладным** называется вхождение идентификатора в конструкцию, которая не является его описанием.

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 области действия (без внутренних по отношению к ней областей)**

Для каждой области действия создается таблица идентификаторов (ТИ). Рассмотрим ее организацию. Когда начинается анализ некоторой области действия, соответствующая ТИ пуста. В процессе обработки объявлений для каждого нового идентификатора элемент добавляется в таблицу только один раз, но поиск в таблице ведется всегда, когда встречается идентификатор. Так как на этот процесс тратится много времени, важно выбрать такую организацию ТИ, которая допускала бы эффективный поиск и расширение.

**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. Эти действия предотвращают повторные сообщения о необъявленном идентификаторе. Поэтому значение функции поиска идентификатора всегда определено. 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.

**Таблица идентификаторов для программы (с вложенными областями действия).**

Теперь рассмотрим организацию ТИ для программы в целом. Один и тот же идентификатор может быть описан и использован многократно в различных областях действия, и для каждого такого описателя должна найтись вершина дерева в ТИ соответствующей области действия. Согласно алгоритму идентификации, правило нахождения определяющего вхождения идентификатора состоит в том, чтобы сначала просмотреть ТИ текущей области действия, затем — непосредственно объемлющей и т. д., пока не будет найдено описание идентификатора или не завершится просмотр таблиц идентификаторов для всех объемлющих областей. Мы можем осуществить такой поиск, сохраняя адреса всех ТИ в стеке.

**Стандартные идентификаторы.**

В языке Паскаль некоторые идентификаторы заранее предопределены, т. е. распознаются компилятором без предварительного описания в программе. Их называют стандартными. Например, false, true, maxint — стандартные константы; integer, boolean, real, char, text — стандартные типы; input, output — стандартные файловые переменные. Кроме того, существуют стандартные функции (abs, cos, eof, eoln, exp и др.) и стандартные процедуры (read, readln, write, writeln, new и др.). В отличие от ключевых слов, стандартные идентификаторы можно переопределять, так как предполагается, что они описаны в некоторой области действия, которая содержит основную программу. Назовем эту область действия фиктивной. В таблицу идентификаторов фиктивной области действия занесем информацию о стандартных идентификаторах типов, констант и переменных. В случае переопределения новое описание стандартного идентификатора всегда включается в таблицу идентификаторов текущей области действия. Следовательно, при обработке прикладного вхождения переопределенного идентификатора стандартное описание становится недоступным.

**Таблица типов.**

Информация о типе идентификатора должна храниться в таблице идентификаторов. Чтобы избежать дублирования этой информации для однотипных переменных, введем дескриптор типа, а в вершину ТИ поместим адрес этого дескриптора. Для единообразия обработки всех типов введем дескрипторы и для стандартных типов.

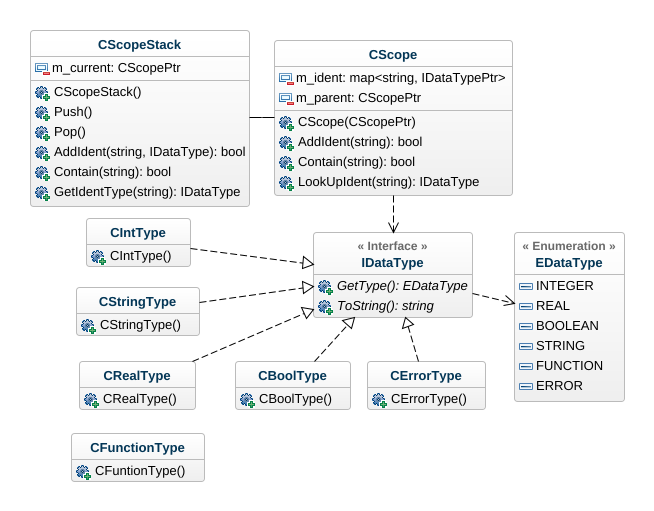
Таблица типов (ТТ) создается для каждой области действия. Когда начинается анализ области действия, соответствующая ТТ пуста. По мере обработки объявлений типов дескрипторы типов добавляются в ТТ.

При описании типа идентификатора может быть допущена ошибка. Для подавления повторных сообщений об ошибках при обработке прикладных вхождений идентификаторов будем считать, что как неопределенный, так и ошибочный тип совместим с любым другим типом. Для неопределенного и ошибочного типов значение поля в записи об идентификаторе будет равно NULL.

Дескрипторы стандартных типов будем хранить в таблице типов фиктивной области действия.

Таблица типов программы в целом имеет организацию, аналогичную ТИ, так как типы локализованы в той области действия, в которой описаны.

* 1. Проектирование



* 1. Тестирование