Лексический анализатор. Итерация №1.

Определим следующие типы распознаваемых токенов:

1. Ключевые слова. Список слов: if, else, then, while, begin, end, do, const, function, var, to, +, -, \*, /, =, <, <=, >, >=, <>, :=, and, or, not, (, ) и т.д.
2. Константы. Выделим четыре типа констант: строковые, целочисленные, вещественные, логические.
3. Идентификаторы. Представляют собой строку однозначно определяющий идентификатор. Например: “x”, “y” и так далее.

Лексический анализатор – программа, принимающая на вход текст (последовательность символов из алфавита) и разбивающая его на подстроки (лексемы) в соответствии с некоторым набором правил.

Далее определим основные классы, необходимые для реализации лексического анализатора.

***CLexer.*** Представляет собой класс лексического анализатора. Хранит строковую константу разделителей токенов (separators), текст программы, строку, хранящую текущий токен, а также позицию текущего читаемого символа в тексте паскаль-программы.

Класс имеет следующие методы: GetTokenType() - определяет тип текущего токена и возвращает умный указатель на созданный объект класса (CIdentToken/ CKeywordToken/CConstToken).

GetNextToken() - выделяет следующий токен, начиная с позиции currentPosition. После выделения токена возвращает умный указатель на него.

IsReal (string s) - проверка строки s на соответствие типу данных real. Для того, чтобы строка соответствовала типу данных real, необходимо, чтобы в числе присутствовала одна точка. Возвращает bool.

IsInteger (string s) - проверка строки s на соответствие типу данных Integer. Чтобы строка соответствовала типу данных integer, необходимо, чтобы в строке присутствовали исключительно цифры.

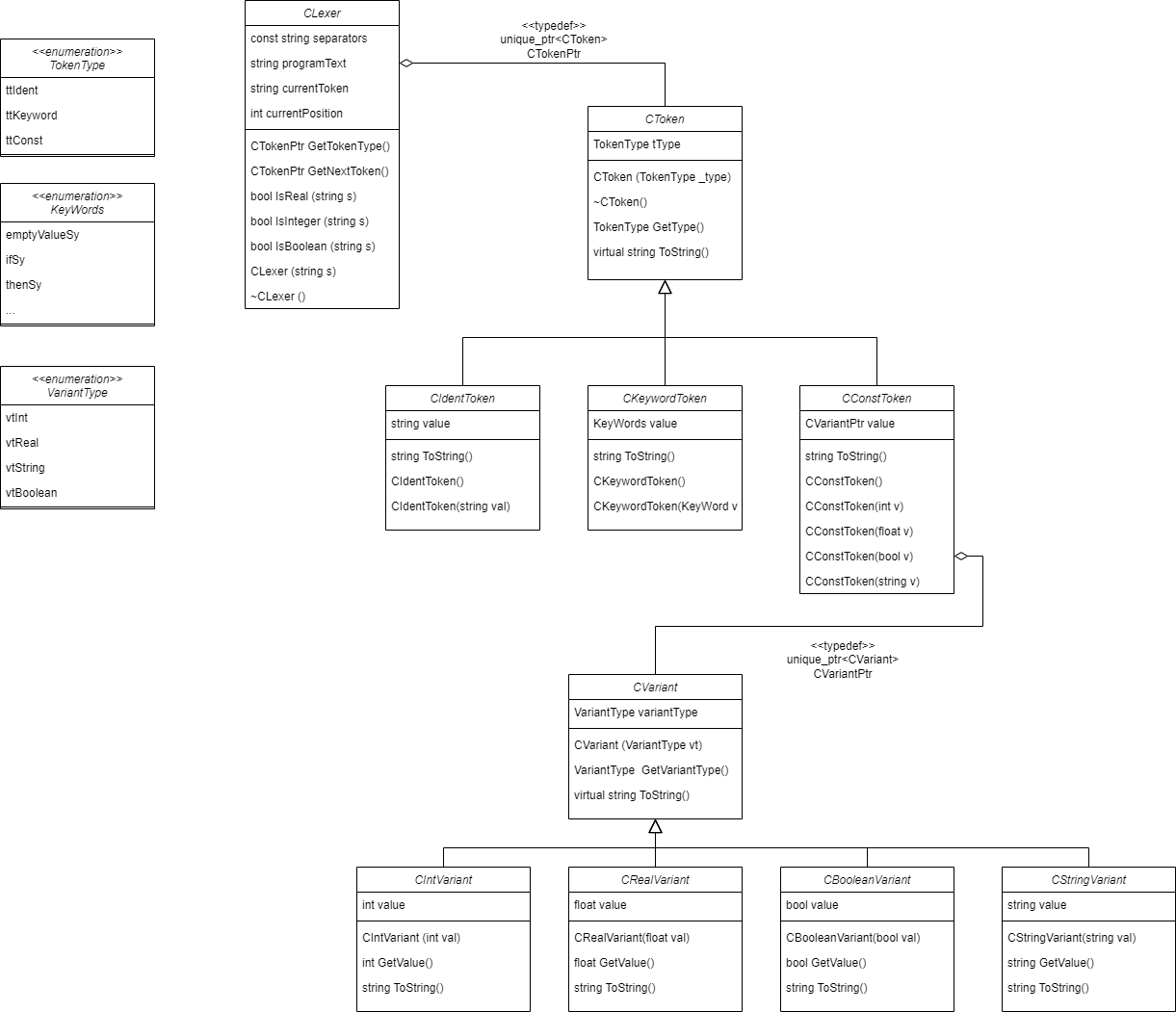
IsBoolean (string s) - проверка строки s на соответствие типу данных boolean. Строка s должна принимать одно из значений: “true”/”false”.

***CToken.*** Абстрактный класс. Наследуемые классы имеют поле типа TokenType, характеризующее один из трех видов токенов. Метод GetType() возвращает значение приватной переменной типа TokenType. Метод ToString() возвращает строковое представление токена.

***CIdentToken / CKeywordToken / CConstToken.*** Наследуются от CToken. Представляются как три разных вида токенов - идентификаторы, ключевые слова, константы. Все три класса переопределяют метод ToString(), а также имеют различные поля. CIdentToken имеет поле string, т.к. идентификатор представляет собой строку. CKeywordToken имеет поле KeyWords, где хранит enum значение одного из ключевых слов. CConstToken имеет поле CVariantPtr, предоставляющее умный указатель на объект класса CVariant, хранящий значение константы.

***CVariant.*** Абстрактный класс. Наследуемые классы имеют поля типа VariantType - enum значение, характеризующее тип константы (string/real/integer/boolean), а также методы GetVaraintType(), возвращающее тип константы и ToString(), преобразующий константу в строку.

***CIntVariant/CRealVariant/CBooleanVariant/CStringVariant.*** Наследуются от CVariant. Представляются как 4 разных вида констант - целочисленные, вещественные, строковые, логические. В каждом классе есть метод GetValue(), возвращающий значение константы, а также переопределенный метод ToString(), возвращающий значение константы в виде строки.



Умные указатели. В качестве указателей на объекты СToken и CVariant использовались умные указатели. Благодаря умным указателям возможно избежать утечек памяти в программе и обеспечить безопасное ее использование.

Далее опишем алгоритм работы лексического анализатора на данном этапе. При создании объекта лексического анализатора, в качестве параметра конструктора передается путь к файлу, из которого необходимо прочитать программу. Лексический анализатор (далее Лексер или CLexer) открывает файл на чтение и прочитывает первую строку файла. Переменные currentPosition и currentLine, ответственные за положение указателя в строке и текущую читаемую строку соответственно обнуляются.

Далее осуществляется чтение всех токенов в текущей строке. При этом символы пробела и табуляции пропускаются. Лекскер посимвольно проходит по строке, аккумулируя символы в переменной currentToken. Разделение токенов происходит на основе следующих правил.

1. Если currentToken= «:», а текущий символ= «=», значит токен = «:=»
2. Если currentToken= «<», а текущий символ= «=», значит токен = «<=»
3. Если currentToken= «>», а текущий символ= «=», значит токен = «>=»
4. Если currentToken= «<», а текущий символ= «>», значит токен = «<>»
5. Если currentToken= «:», а текущий символ != «=», значит токен = «:»
6. Если currentToken= «<», а текущий символ!= «=» и текущий символ != «>», значит токен = «<»
7. Если currentToken= «>», а текущий символ != «=», значит токен = «>»
8. Если встречен разделитель из :=<>;,()\*+/- ", , значит токен = currentToken
9. Если текущий символ != «(пробел)» и текущий символ != «/t», то currentToken+=текущий символ.

При выделении токена осуществляется его идентификация. Токен может принадлежать одному из трех типов: ключевое слово, константа, идентификатор.

Таким образом, изначально сопоставляется выделенный токен с возможными ключевыми словами. Если токен является ключевым словом, то возвращается уникальный указатель на объект класса CKeywordToken со значением ключевого слова.

В противном случае, если токен не является ни одним ключевым словом, то проверяется, не является ли токен какой-либо константой. На данной итерации возможна проверка на три вида констант: real, integer, Boolean.

Boolean – token= «true» или token = «false»

Integer – строка, состоящая исключительно из символов 0-9, где первый символ не ноль.

Real - строка, состоящая из символов 0-9, а также «.». Причем «.» может встречаться только один раз и не может стоять в начале строки, вначале строки не могут идти несколько нулей подряд.

Если выявлено, что текущий токен является константой, то создается уникальный указатель на объект класса CConstToken с значением, указанным в токене. В противном случае осуществляется проверка на идентификатор.

Идентификатор не может начинаться с цифры, а также в нем не может быть недопустимых символов.

Если текущий токен является идентификатором, то, следовательно, возвращается умный указатель на объект класса CIdentToken, в качестве значения которого выступает имя идентификатора – содержимое токена.

Для вывода значений идентификаторов, констант и кодов ключевых слов используется метод GetValue().

Лексический анализатор. Итерация №2

В ходе реализации лексера в первой итерации, было установлены следующие недоработки:

1. Нельзя использовать строковые константы
2. Нельзя использовать комментарии
3. Паскаль – язык программирования не чувствительный к регистру
4. Необходимо учитывать переполнение типов при проверке
5. Необходимо осуществить проверку программы на наличие лексических ошибок и выводить позиции ошибки.

Проблема чувствительности к регистру решается путем приведения всех токенов к нижнему регистру при помощи функции tolower(), которая переводит текущий символ к нижнему регистру.

Проблема переполнения на этапе лексического анализатора не может быть решена, т.к. необходимо понимание, к какому типу относится переменная.

Проверка программы на наличие лексических ошибок. Необходимо реализовать нахождение следующих ошибок:

1. Идентификатор начинается с цифры
2. В идентификаторе присутствуют недопустимые символы

При нахождении несоответствия, функция GetNextToken возвращает токен типа Keyword со значением enum «errorValueSy». Помимо этого, в листинг программы записывается сообщение об ошибке и положение начала некорректного токена (строка + позиция в строке).

Обработка строковых констант подразумевает отслеживание открывающих и закрывающих кавычек. Если программист забыл добавить закрывающую кавычку, то соответствующее сообщение будет выведено в конце листинга.

Тестирование лексического анализатора.

Далее проведем тестирование работы лексического анализатора и постараемся покрыть весь его функционал. Стоит отметить, что на данном этапе лексический анализатор не способен находить ошибки, выделять комментарии и строковые константы. Этот функционал будет добавлен в последующей итерации.

Тест 1. Стандартная программа паскаль (Корректная).

Входные данные:

const LIMIT = 10000;

var n,i,j,s,lim,c,d : integer

begin

for i:=1 to LIMIT do begin

s:=1;

for j:=2 to lim do begin

c:=i mod j;

d:=i div j;

if c = 0 then begin

s:=0;

end;

end;

end;

end.

Выходные данные:

9 ident(LIMIT) 36 constVal(10000) 24

12 ident(n) 27 ident(i) 27 ident(j) 27 ident(s) 27 ident(lim) 27 ident(c) 27 ident(d) 48 16

6

10 ident(i) 42 constVal(1) 13 ident(LIMIT) 8 6

ident(s) 42 constVal(1) 24

10 ident(j) 42 constVal(2) 13 ident(lim) 8 6

ident(c) 42 ident(i) 30 ident(j) 24

ident(d) 42 ident(i) 29 ident(j) 24

2 ident(c) 36 constVal(0) 4 6

ident(s) 42 constVal(0) 24

7 24

7 24

7 24

7 31

На данной итерации лексического анализатора все ключевые слова перерабатываются программой в enum-значения, где у каждого ключевого слова есть свой идентификатор. Для удобства идентификаторы и константы помечаются словами ident и constVal соответственно, при этом в скобках содержится значение идентификатора или же значение константы.

Тест 2. Все конструкции языка Pascal для моего варианта.

Ниже отображены все конструкции языка Pascal моего варианта. Стоит отметить, что в текущем тестовом примере нарушен синтаксис языка, но это не имеет значения в рамках тестирования лексического анализатора.

Входные данные:

program test2;

procedure printBin (n:integer,s:string);

var k:boolean;

begin

k:=true;

for n:=1 to n do begin

k:= not k;

if k=true then

k:=false;

else

k:=true;

end;

end;

type beverage = (coffee, tea, milk, water, coke, limejuice);

var a: string;

b,c: real;

d: boolean;

e: integer;

begin

e:=e +(1-1)\*1/ 0.01;

e:=e div 10;

e:=e mod 10;

if e>10 and e<=9 or e<5 and not e>=3 then e=3;

case e of

1: e:=2;

2: e:=3;

end;

Выходные данные:

25 ident(test2) 24

15 ident(printBin) 46 ident(n) 48 16 27 ident(s) 48 17 47 24

12 ident(k) 48 19 24

6

ident(k) 42 constVal(true) 24

10 ident(n) 42 constVal(1) 13 ident(n) 8 6

ident(k) 42 45 ident(k) 24

2 ident(k) 36 constVal(true) 4

ident(k) 42 constVal(false) 24

3

ident(k) 42 constVal(true) 24

7 24

7 24

26 ident(beverage) 36 46 ident(coffee) 27 ident(tea) 27 ident(milk) 27 ident(water) 27 ident(coke) 27 ident(limejuice) 47 24

12 ident(a) 48 17 24

ident(b) 27 ident(c) 48 18 24

ident(d) 48 19 24

ident(e) 48 16 24

6

ident(e) 42 ident(e) 32 46 constVal(1) 33 constVal(1) 47 34 constVal(1) 35 constVal(0.010000) 24

ident(e) 42 ident(e) 29 constVal(10) 24

ident(e) 42 ident(e) 30 constVal(10) 24

2 ident(e) 39 constVal(10) 43 ident(e) 38 constVal(9) 44 ident(e) 37 constVal(5) 43 45 ident(e) 40 constVal(3) 4 ident(e) 36 constVal(3) 24

20 ident(e) 21

constVal(1) 48 ident(e) 42 constVal(2) 24

constVal(2) 48 ident(e) 42 constVal(3) 24

7 24

Тест 3. Программа паскаль, содержащая ошибки.

Входные данные:

PROGRAM TeSt2;

i:=54;

j:=12..0;

k:=12.0;

l:=00.12;

m:=005;

01eq:=1;

e1313:=4;

Выходные данные:

26 ident(test2)

25 ident(i) 43 constVal(54)

25 ident(j) 43 0

25 ident(k) 43 constVal(12,000000)

25 ident(l) 43 0

25 ident(m) 43 0

25 0 43 constVal(1)

25 ident(e1313) 43 constVal(4) 25

Некорректный идентификатор (Строка: 3, Позиция: 4)

Некорректный идентификатор (Строка: 5, Позиция: 4)

Некорректный идентификатор (Строка: 6, Позиция: 4)

Некорректный идентификатор (Строка: 7, Позиция: 1)

Как можно заметить, некорректные токены были помечены 0, при этом для каждого такого токена было выведено сообщение об ошибке.

Синтаксический анализатор.

Для реализации синтаксического анализатора необходимо изначально определить все используемые конструкции языка Pascal. Приведем эти конструкции ниже в БНФ нотации.

1. <программа>::=**program** <имя>(<имя файла>{,<имя файла>});<блок>.
2. <имя файла>::=<имя>
3. <имя>::=<буква>{<буква>|<цифра>}
4. <блок>::=<раздел типов><раздел переменных><раздел процедур и функций><составной оператор>
5. <раздел типов>::=<пусто>|**type** <определение типа>; {<определение типа>;}
6. <определение типа>::=<имя>=<тип>
7. <тип>::=<простой тип>
8. <простой тип>::=<перечислимый тип>|<ограниченный тип>|<имя типа>
9. <перечислимый тип>::=(<имя>{,<имя>})
10. <ограниченный тип>::=<константа>..<константа>
11. <имя типа>::=<имя>
12. <раздел переменных>::= **var** <описание однотипных переменных>;{<описание однотипных переменных>;}|<пусто>
13. <описание однотипных переменных>::=<имя>{,<имя>}:<тип>
14. <раздел процедур и функций>::={<описание процедуры или функции>;}
15. <описание процедуры или функции>::=<описание процедуры>
16. <описание процедуры>::=<заголовок процедуры><блок>
17. <заголовок процедуры>::= **procedure** <имя>;| **procedure** <имя> (<раздел формальных параметров>{;<раздел формальных параметров>});
18. <раздел формальных параметров>::=<группа параметров>| **var** <группа параметров>| **procedure** <имя>{,<имя>}
19. <группа параметров>::=<имя>{,<имя>}:<имя типа>
20. <оператор>::=<непомеченный оператор>
21. <непомеченный оператор>::=<простой оператор>|<сложный оператор>
22. <простой оператор>::=<оператор присваивания>|<оператор процедуры>
23. <оператор присваивания>::= <переменная>:=<выражение>
24. <переменная>::=<полная переменная>| <компонента переменной>|<указанная переменная>
25. <полная переменная>::=<имя переменной>
26. <имя переменной>::=<имя>
27. <выражение>::=<простое выражение>|<простое выражение> <операция отношения><простое выражение>
28. <операция отношения>::==|<>|<|<=|>=|>|**in**
29. <простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}
30. <аддитивная операция>::=+|-|**or**
31. <слагаемое>::=<множитель>{<мультипликативная операция> <множитель>}
32. <мультипликативная операция>::=\*|/|**div**|**mod**|**and**
33. <множитель>::=<переменная>|<константа без знака>| (<выражение>)| not <множитель>
34. <константа без знака>::=<число без знака>|<строка>|<имя константы>|nil
35. <оператор процедуры>::=<имя процедуры>|<имя процедуры>(<фактический параметр>{,<фактический параметр>})
36. <имя процедуры>::=<имя>
37. <фактический параметр>::=<выражение>|<переменная>|<имя процедуры>
38. <пустой оператор>::=<пусто>
39. <пусто>::=
40. <сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>|<оператор присоединения>
41. <составной оператор>::= **begin** <33 >{;<оператор>} **end**
42. <выбирающий оператор>::=<условный оператор>| <оператор варианта>
43. <условный оператор>::= **if** <выражение> **then** <оператор>| **if** <выражение> **then** <оператор> **else** <оператор>
44. <оператор варианта>::= **case** <выражение> **of** <элемент списка вариантов>{;<элемент списка вариантов>} **end**
45. <элемент списка вариантов>::=<список меток варианта>: <оператор>|<пусто>
46. <список меток варианта>::=<метка варианта>{,<метка варианта>}
47. <оператор цикла>::=<цикл с предусловием>|<цикл с постусловием>|<цикл с параметром>
48. <оператор цикла>::=<цикл с предусловием>
49. <цикл с предусловием>::= while <выражение> do <оператор>

На основе вышеописанных конструкций был построен лексический анализатор. Далее разберем основные принципы построение такого анализатора:

1. Если конструкция L содержит 1 терминальный символ T, то обработка L – это accept(T).
2. Если конструкция L содержит 1 нетерминальный символ N, то обработка – это вызов обработки N.
3. Если L = L1 L2 L3, то обработка – последовательный вызов L1, L2, L3.
4. Если L = L1 | L2, то L1 и L2 должны отличаться первым символом, по этому первому символу принимается решение, вызывать обработку L1 или L2.
5. Если L = [L1] L2, то L1 и L2 должны отличаться первым символом. Если текущий символ совпадает с первым символом L1, то вызываем обработку L1, далее в любом случае вызываем обработку L2.
6. Если L= {L1}, то обработка – вызов обработки L1 до тех пор, пока текущий символ совпадает с первым символом конструкции L1.

Исключением из этих правил является обработка вызова процедуры. Эта проблема решается на этапе реализации семантического анализатора.

Нейтрализация ошибок

Нейтрализация ошибок происходит, когда синтаксический анализатор обнаруживает ключевое слово/идентификатор/константу в некорректном месте. При обнаружении неподходящего токена выводится соответственное сообщение об ошибке в консоль с указанием необходимого ключевого слова или идентификатора/константы. Также в консоль выводится позиция ошибки.

После обнаружения ошибки синтаксический анализатор пытается найти следующее ключевое слово, начиная с которого можно дальше проводить синтаксический анализ. Список из таких ключевых слов (followers) передается в качестве параметра в метод Accept(). При этом, чем глубже находится синтаксическая конструкция, тем больше followers передается как параметр. Список followers изначально создается в стеке в специальной структуре, который пополняется новыми ключевыми словами по мере спуска вглубь БНФ конструкций. Стратегия подбора подходящих followers заключается в выборе наиболее часто встречающихся ключевых слов после какого-либо ошибочного участка БНФ-конструкции. Например, если синтаксический анализатор не нашел ключевое слово “if” в условном операторе, то пропустит все последующие токены до ключевого слова “then/else”.

Стоит отметить, что нельзя гарантировать 100% корректность работы синтаксического анализатора, а, следовательно и реализовать алгоритм, работающий в 100% случаев.

Тестирование.

1. Две ошибки

PROGRAM OK;

var n,p1,p2,p3,p4:integer;

begin

p1:=n div 1000;

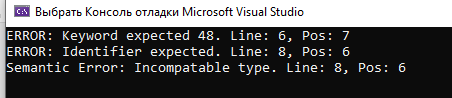
p4:=n mod 10;

p2:=(n di 100) mod 10;

p3:=(n div 10) mod 10;

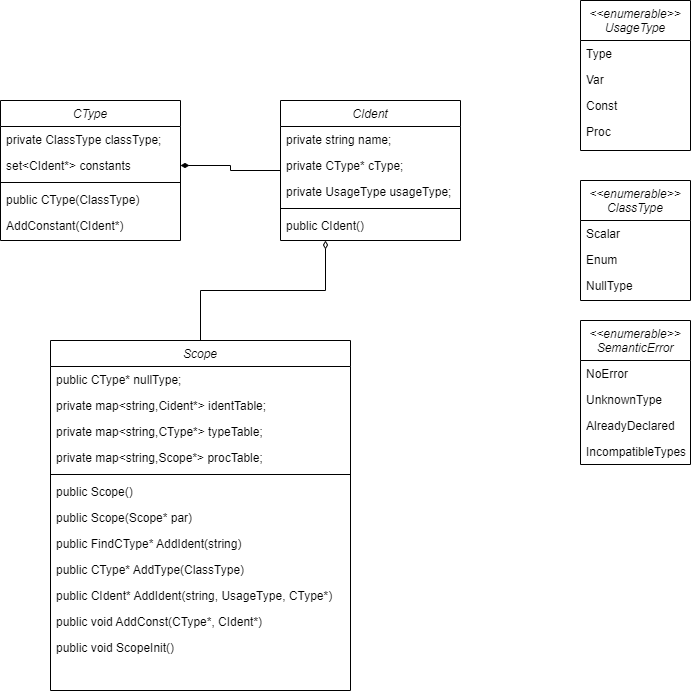
if p1+-p4=p2+p3 then p3:=1 else p1:=1

end.



1. Привести другие примеры

Семантический анализатор. Первая итерация.



Данный рисунок демонстрирует диаграмму классов семантического анализатора (1 итерация). В первой итерации реализована работа анализатора, за исключением обработки процедур.

При инициализации фиктивной области действия создаются базовые типы (integer,real,boolean,string) с указанием типа использования (Type/Var/Const/Proc) и ссылкой на область памяти CType. Область памяти CType хранит в себе информацию о классе типа (scalar/enum), а также ссылки на константы. Таким образом, принадлежность идентификатора к какому-либо типу будет определяться посредством сравнения ссылок CType. Константы true, false типа boolean также добавляются в фиктивную область действия, они имеют ссылку на CType идентичную ссылке типа boolean на свой CType. Таким образом можно инициализировать дополнительные константы MinInteger, MaxInteger, и т.д. при необходимости.

Также создается отдельная ссылка на CType с нулевым значением. Эта ссылка будет использоваться для выявления семантических ошибок.

Большинство методов синтаксического анализатора на данном этапе возвращает ссылку на область памяти CType, отведенную для конкретного типа real/boolean/string/integer или для пользовательского типа данных.

В контексте определенных методов синтаксического анализатора происходит сравнение ссылок CType для нахождения возможных семантических ошибок. Выделим основные правила:

1. Если CType\* t1 = CType\* t2, значит, что типы эквиваленты, возвращаем CType\* t1.
2. Если CType\* t1 = CType(integer) and CType\* t2 =CType(real), то, в большинстве случаев эти типы эквивалентны. Возвращаем t2. Исключение – целочисленные операции div/mod. Помимо этого, вещественной переменной можно присваивать значение целочисленное, но не наоборот.
3. Если CType\* t1 = CType(noneType), то t1 – ошибочный тип. Возвращаем t1, пробрасывая ошибочный тип по дереву БНФ на уровень выше.
4. Если CType\* t2 = NULL, CType\* t1 !=NULL, значит, ссылка t2 была не определена. Возвращаем тип t1.

Ссылки CType\* также необходимы для выделения доступных операций типов. Таким образом:

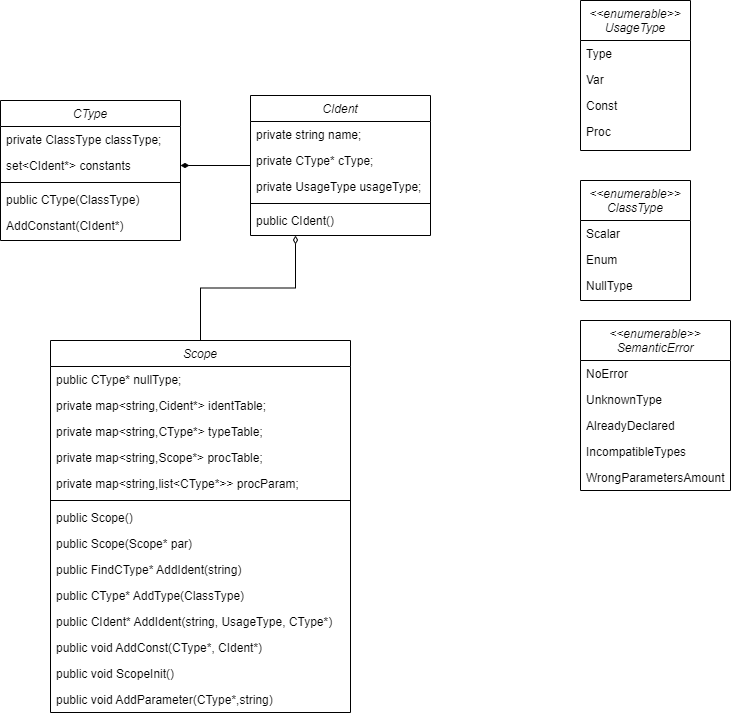
1. Real, Integer поддерживает: «+», «-», «\*», «/», операторы сравнения.
2. Boolean поддерживает: «or», «and», операторы сравнения.

Семантический анализатор. Вторая итерация.

В рамках второй итерации был реализован Метод ThrowSemanticError осуществляющий вывод найденных ошибок. Всего есть 5 типов различных семантических ошибок:

1. NoError – отсутствие ошибки
2. UnknownVar – не объявленная переменная
3. AlreadyDeclared – переменная уже объявлена
4. IncompatibleTypes – несовместимые типы
5. WrongParametersAmount – несоответствие количества параметров процедуры.

Диаграмма классов второй итерации реализации семантического анализатора:



Здесь у каждой области видимости Scope есть словарь, где в качестве ключа используется имя процедуры, а в качестве значения – ссылка на область видимости этой процедуры.

Помимо этого введен словарь procParam, отвечающий за список параметров определенной процедуры. Благодаря этому словарю можно отследить порядок и количество параметров, которые необходимо передать в определенную процедуру.

Генерация кода.

Генерация кода - это автоматическое создание программного кода специальным приложением, при котором по заданным условиям полностью или частично формируется исходный код программы. В качестве целевого языка, в который выполняется трансляция был выбран Assembler MASM. MASM (Microsoft Macro Assembler) — ассемблер для процессоров семейства x86.

Выбор MASM был обусловлен разработкой компилятора на ОС windows в Visual Studio. Visual Studio способна из коробки запускать MASM-код, в том числе в этой IDE есть встроенные средства для дебага программы (просмотра изменения содержимого регистров). Помимо этого, MASM имеет встроенные функции вывода в консоль – crt\_printf, где можно вывести значение целочисленной/вещественной переменной или строковой константы. Для подключении необходимых библиотек MASM необходимо было установить MASM SDK.

Первая проблема, с которой я столкнулся на этапе изучения ассемблера – вывод в консоль. Без использования MASM SDK вывод в консоль базировался на вызове 21-ого прерывания в коде программы, но вывести таким образом значения переменной в необходимом формате было невозможно.

Таким образом, использовалась функция crt\_printf , подобная функции printf. В качестве параметров она принимала формат вывода (целочисленный/дробный) и соответствующую переменную.

Далее рассмотрим использующиеся разделы assembler:

1. **.data**. Раздел описания всех использующихся переменных и констант (служебных и пользовательских). Таким образом, здесь были описаны форматы вывода (например: FormatInteger db '%d ', 0) , булевые константы (например: FALSE equ 0). Описание констант, использовавшихся в ходе выполнения программы помечалось строкой const и порядковым номером константы (например: const1 real8 0.100000 ). Переменные же описываются практически аналогичным образом. Используемые типы данных: dd (double word) для integer, занимает 4 байта; real8 для real, занимает 8 байтов.
2. **.model**. Указывает на используемую модель памяти, в этой же секции подключаются MASM библиотеки. В текущем генераторе кода используется плоская модель памяти, где код и данные используют одно и то же адресное пространство.
3. **.code**. Секция кода. Рассмотрим основные команды, на основе которых производились вычисления.

Вычисление выражений проводилось на основе стека с использованием операций стандартных арифметических операций (сложение, умножение, деление, вычитание).

fld a ; положить в стек float-переменную a

fild a ; положить в стек int-переменную a

fmul произвести умножение элементов стека

fdiv произвести деление между элементами стека

fsub произвести вычитание между элементами стека

fadd произвести сложение между элементами стека

fstp a – достать верхний элемент стека в float-переменную a

fstip a – достать верхний элемент стека в int-переменную a

Для использования условных операторов и циклов было необходимо использовать метки. Для перемещения между этими метками была использован fcompp, осуществляющая вычитание двух верхних элементов в стеке и заполнение регистров C0-C3, на основе которых выполнялся переход по меткам при помощи команд je (если равно), jne (если не равно), jb (если меньше), jbe (если меньше или равно), ja (если больше), jae (если больше и равно). Стоит отметить, что использовались именно такие операторы перехода, а не jl, jg и т.д.

В конце программы необходимо объявить о ее завершении вызовом invoke crt\_\_exit, 0.

Ниже рассмотрим процесс построения генератора. Во-первых, был создан класс CGenerator, осуществляющий генерацию кода. В определенных местах синтаксического анализатора вызывался метод ChangeGenerationStage, изменяющий состояние генератора. Изменение состояние генератора осуществлялась при начале и окончании чтения выражения, цикла, if-else конструкции. В определенном состоянии генератор выполнял определенные функции. Например, при изменении состояния генератора на VAR – поток лексем из синтаксического анализатора направлялся в генератор, который преобразовывал их в переменные согласно типу и записывал их в секцию .data.

При изменении состояния генератора на Expression осуществлялось перенаправление лексем из синтаксического анализатора в генератор, на основе которых строилась обратная польская запись, необходимая для вычисления выражений на стеке. Далее все операции по вычислению выражения были записаны генератором на языке ассемблера.

Наиболее проблемной задачей оказалась реализация условных конструкций в циклах и в условных операторах. В рамках настоящей работы условное выражение, которое способен представить ассемблер, состоит из одного сравнения. Конструкции while, if-else подразумевают использование условных и безусловных переходов с использованием меток. Таким образом, изначально был введен счетчик меток, чтобы не было коллизии.

Цикл while на языке ассемблер представляется следующим образом:

Jump L1

L2: тело цикла

L1: условие цикла. Условный переход на метку L2.

Также было необходимо учитывать возможность того, что циклы могут быть вложены друг в друга. Для отслеживания вложенности был использован стек whileStack. Таким образом, при записи вычислений выражений, они будут записаны в соответствующей области видимости.

Аналогичным образом было решено поступить с оператором if-else.

Таким образом, конструкции while и if-else могут иметь несколько уровней вложенности.

Чтобы реализовать вложенность while в if-else и наоборот, использовался порядковый номер лейбла конструкции – чем больше номер, тем больше вложенность.

Стоит отметить, что работа класса генератора осуществляется до обнаружения первой ошибки – в противном случае поток лексем не перенаправляется в этот класс, а его состояние остается в состоянии «ошибка».

Последним этапом построения генератора является создание исполняемого файла. Для этого необходимо собрать приложение при помощи bat файла со следующим наполнением:

C:\masm32\bin\ml /c /Cx /coff program.asm

C:\masm32\bin\link program.obj /SUBSYSTEM:console /out: program.exe

На выходе имеем листинг программы, объектный файл и исполняемый exe. Для выполнения программы необходимо запустить exe-файл в консоли.

start cmd /k "program.exe"

Выполнение этих файлов из с++ программы возможно при использовании system. Таким образом, изначально осуществляется сборка программы и формирование exe-файла, а затем запуск этого файла.

Ниже разберем небольшие алгоритмы, написанные на pascal и преобразованные в код на assembler.

1. Вычисление НОД двух чисел. Алгоритм Евклида.

program firstOne;

var a,b,d:integer;

r:real;

flag:boolean;

begin

a:=42;

b:=35;

while (a <> b) do begin

if (a>b) then

a:=a-b

else

b:=b-a;

end;

writeln(a)

end.

.386

.model flat, C

option casemap :none

include \masm32\include\msvcrt.inc

includelib \masm32\lib\msvcrt.lib

.data

FormatInteger db '%d ', 0

FormatDecimal db '%f ', 0

FALSE equ 0

TRUE equ

a dd ?

b dd ?

d dd ?

r real8 ?

flag dd FALSE

const1 dd 42

const2 dd 35

.code

start:

fild const1

fistp a

fild const2

fistp b

jmp L1

L2:

fild a

fild b

fcompp

fstsw ax

sahf

jae L3

fild a

fild b

fsub

fistp a

jmp L4

L3:

fild b

fild a

fsub

fistp b

L4:

L1:

fild a

fild b

fcompp

fstsw ax

sahf

jne L2

invoke crt\_printf, addr FormatInteger, a

invoke crt\_\_exit, 0

end start

Вывод:



1. Вычисление чисел Фибоначчи (до 100)

program firstOne;

var a,b,d:integer;

r:real;

flag:boolean;

begin

a:=1;

b:=1;

while (b<100) do begin

writeln(b);

d:=b;

b:=a+b;

a:=d;

end

end.

.386

.model flat, C

option casemap :none

include \masm32\include\msvcrt.inc

includelib \masm32\lib\msvcrt.lib

.data

FormatInteger db '%d ', 0

FormatDecimal db '%f ', 0

FALSE equ 0

TRUE equ

a dd ?

b dd ?

d dd ?

r real8 ?

flag dd FALSE

const1 dd 1

const2 dd 1

const3 dd 100

.code

start:

fild const1

fistp a

fild const2

fistp b

jmp L1

L2:

invoke crt\_printf, addr FormatInteger, b

fild b

fistp d

fild a

fild b

fadd

fistp b

fild d

fistp a

L1:

fild b

fild const3

fcompp

fstsw ax

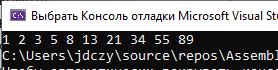
sahf

ja L2

invoke crt\_\_exit, 0

end start

Вывод:



1. Подсчет значений функции. (y=2x^2-x/2+4)

Вывод:

program firstOne;

var step,from,toVal,func:real;

begin

step:=0.1;

from:=0.5;

toVal:=1.5;

while (from<toVal) do begin

func:=2\*from\*from-from/2+4;

writeln(func);

from:=from+step;

end

end.

.386

.model flat, C

option casemap :none

include \masm32\include\msvcrt.inc

includelib \masm32\lib\msvcrt.lib

.data

FormatInteger db '%d ', 0

FormatDecimal db '%f ', 0

FALSE equ 0

TRUE equ

step real8 ?

from real8 ?

toval real8 ?

func real8 ?

const1 real8 0.100000

const2 real8 0.500000

const3 real8 1.500000

const4 dd 2

const5 dd 2

const6 dd 4

.code

start:

fld const1

fstp step

fld const2

fstp from

fld const3

fstp toval

jmp L1

L2:

fild const4

fld from

fmul

fld from

fmul

fld from

fild const5

fdiv

fsub

fild const6

fadd

fstp func

invoke crt\_printf, addr FormatDecimal, func

fld from

fld step

fadd

fstp from

L1:

fld from

fld toval

fcompp

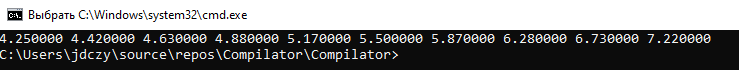
fstsw ax

sahf

ja L2

invoke crt\_\_exit, 0

end start



Сравним полученные результаты с настоящими:



Как видно, результаты идентичны