|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Министерство науки и высшего образования РФ | | | | | | | | |
|  | | |  | | |  | | |
| Пермский государственный национальный  исследовательский университет | | | | | | | | |
|  | | |  | | |  | | |
|  | | ОТЧЁТ  по лабораторной работе «Разработка компилятора языка Pascal»  по дисциплине «Формальные грамматики и методы трансляции» | | | | |  | |
|  | | |  | | |  | | |
|  | Работу выполнил  студент гр. ПМИ-1,2  Валеев Р. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_ 2021 | | |  | Проверил  асис.кафедры МОВС  Пономарев Ф.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_ 2021 | | |  |
|  |  | | |  |  | | |  |
| Пермь 2021 | | | | | | | | |

# Анализ

Для решения глобальной задачи – написание компилятора для подмножества языка Pascal, требуется решить несколько подзадач, а именно, спроектировать и разработать лексический, синтаксический и семантический анализаторы, а также генератор кода.

Следующим этапом станет разработка семантического анализатора. На этапе семантического анализа происходит работа с идентификаторами и типами. Синтаксический и семантический анализ происходит параллельно, то есть во время разбора синтаксических конструкций языка также осуществляется семантическая проверка.

Таким образом, для реализации семантического анализатора будут модифицированы функции, осуществляющие синтаксический анализ, и будут добавлены структуры, хранящие информацию об идентификаторах и типах.

# Проектирование

Перед началом разработки семантического анализатора было проведено начальное проектирование. На рисунке ниже представлена диаграмма классов, отражающая обновлённую структуру компилятора

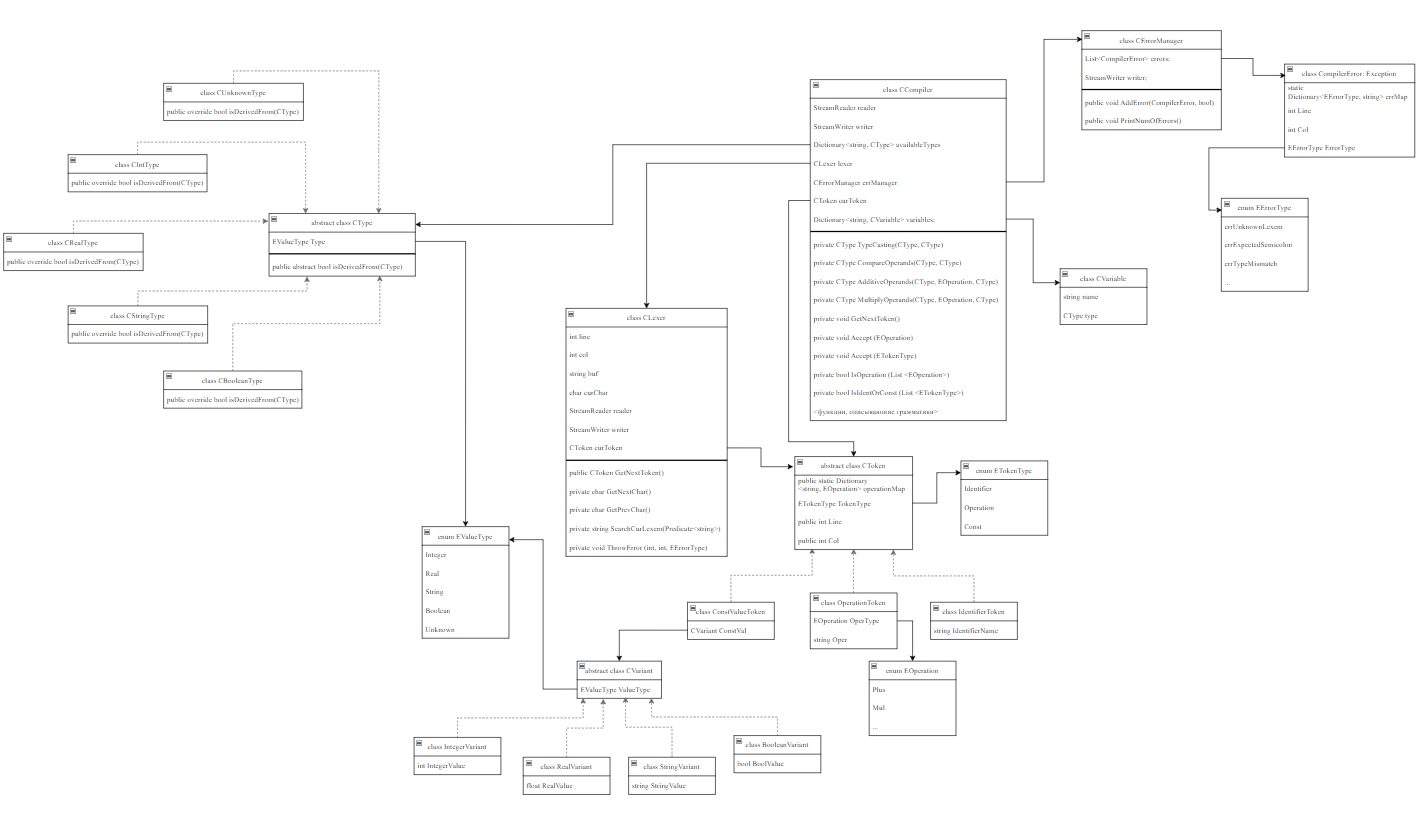


Рисунок 1 – Диаграмма классов

Информацию о допустимых типах будем хранить в коллекции Dictionary, которая позволяет обеспечить быстрый доступ к элементам по ключу. В качестве ключа будем использовать название типа, по которому можно будет получить объект соответствующего типа.

Для хранения типов был спроектирован класс CType, являющийся абстрактным. В нём хранится информация о типе. Также он имеет абстрактный метод isDerivedFrom, который принимает в качестве входного параметра CType и возвращает логическое значение. Данный метод определяет возможность приведения переданного типа к текущему экземпляру класса. Классы CIntType, CRealType, CStringType, CBooleanType, CUnknownType унаследованы от CType и переопределяют метод isDerivedFrom. Класс CUnknownType предназначен для описания неизвестного типа. Он будет использоваться в том случае, если во время анализа не удастся определить тип.

Идентификаторы также будут храниться в коллекции Dictionary. Для хранения информации о переменных был спроектирован класс CVariable, содержащий следующие поля: string Name – название переменной, CType Type – тип переменной.

Следующие методы, реализованные для анализа синтаксических конструкций, будут модифицированы:

* SimilarTypeVarPart (информация о идентификаторах добавляется в словарь переменных; проверка допустимости типов; проверка дублирующих объявлений переменной),
* AssignmentStatement (проверка объявления переменной; проверка возможности выполнения присваивания),
* Expression (проверка совместимости типов операндов для операций отношения; возвращает значение типа CType),
* SimpleExpression (проверка совместимости типов операндов для мультипликативных операций; возвращает значение типа CType),
* Term (проверка совместимости типов операндов для аддитивных операций; возвращает значение типа CType),
* Factor (возвращает значение типа CType),
* IfStatement (проверка типа выражения),
* LoopStatement (проверка типа выражения).

Проверка совместимости типов будет реализована в специальных методах CompareOperands, AdditiveOperands, MultiplyOperands.

# Разработка

После окончания этапа проектирования начался этап разработки. Сначала был реализован класс CType и все его дочерние классы, а также класс CVariable. У каждого производного от CType класса, реализуем собственную версию метода isDerivedFrom. Договоримся о следующих правилах преобразования типов:

* два одинаковых типа приводимы друг к другу;
* тип integer приводим к типу real;
* неизвестный тип может приводится к любому типу;

Информация о допустимых типах формируется перед началом этапа компиляции. Создаётся по одному объекту классов CIntType, CRealType, CStringType, CBooleanType, CUnknownType, которые добавляются в словарь availableTypes.

Начнём с модификации метода SimilarTypeVarPart, ответственного за анализ конструкции *<описание однотипных переменных>*. Так как информация о типе идентификатора будет прочитана анализатором в последнюю очередь, то создадим вспомогательный список tempListVars для добавления новых идентификаторов. При добавлении идентификатора надо проверить, что данное имя не было добавлено ранее в временный список или в список variables. В противном случае будет сформирована ошибка о том, что идентификатор был описан повторно. После того как будет прочитана информация о типе, проверяем что данный тип присутствует в availableTypes и добавляем переменную в область видимости:

string type = ((IdentifierToken)curToken).IdentifierName;

foreach (var varName in tempListVars)

{

try

{

variables.Add(varName, new CVariable(varName, availableTypes[type]));

}

catch

{

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errInType));

variables.Add(varName, new CVariable(varName,availableTypes["unknown"]));

}

}

Метод AssignmentStatement анализирует конструкцию *<оператор присваивания>*. В операторе присваивания значение правой части выражения присваивается левой части выражения. Левая часть выражения должна быть переменной, поэтому проверим, что идентификатор находится в variables:

if (variables.ContainsKey(((IdentifierToken)curToken).IdentifierName))

left = variables[((IdentifierToken)curToken).IdentifierName].Type;

else

{

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errUnknownIdent));

left = availableTypes["unknown"];

}

Если это не так, то сформируем ошибку и попытаемся нейтрализовать её путём определения типа левой части выражения как unknown. Далее проверяем, что тип правой части выражения может быть приведен к полученному типу левой части:

if (!left.isDerivedFrom(right) && left.Type != EValueType.Unknown)

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errTypeMismatch));

Методы IfStatement и LoopStatement осуществляют анализ условного оператора и оператора цикла с предусловием. После вычисления выражений конструкций if, while значение приводится к типу boolean. Если тип выражения не приводим к логическому, то формируется ошибка.

Рассмотрим методы, проводящие анализ конструкций *<выражение>*, *<простое выражение>*, *<слагаемое>*, *<множитель>*. В этих методах выполняется проверка типов операндов.

Код метода Expression:

/\* выражение \*/

CType Expression()

{

var left = SimpleExpression();

if (IsOperation(new List<EOperation>() {EOperation.Equals, EOperation.NotEquals, EOperation.Less, EOperation.Lesseqv, EOperation.Bigger, EOperation.Bigeqv}))

{

GetNextToken();

var right = SimpleExpression();

left = CompareOperands(left, right);

}

return left;

}

В left запоминается тип, полученный в результате вызова SimpleExpression. Если следующим символ будет операция отношения, то метод SimpleExpression будет вызван повторно. Результат этого вызова будет сохранён в right. Далее осуществляется проверка того, что значения операндов могут быть приведены к одному типу:

private CType CompareOperands(CType left, CType right)

{

/\* типы совпадают или левый операнд приводим к правому операнду и наоборот \*/

if (left.isDerivedFrom(right) || right.isDerivedFrom(left))

return availableTypes["boolean"];

/\* типы неприводимы друг к другу \*/

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errTypeMismatch));

return availableTypes["unknown"];

}

Если окажется, что ни один из типов не может быть приведён к другому, то будет сформирована ошибка и возвращён тип unknown. Метод Expression вернёт новое значение переменной left.

Код метода SimpleExpression:

/\* простое выражение \*/

CType SimpleExpression()

{

bool sign = false;

if (IsOperation(new List<EOperation>() { EOperation.Plus, EOperation.Min }))

{

sign = true;

GetNextToken();

}

var left = Term();

if (sign && (left.Type == EValueType.String || left.Type == EValueType.Boolean)) errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errTypeMismatch));

while (IsOperation(new List<EOperation>() { EOperation.Plus, EOperation.Min, EOperation.Or }))

{

var operation = ((OperationToken)curToken).OperType;

GetNextToken();

var right = Term();

left = AdditiveOperands(left, operation, right);

}

return left;

}

Перед вызовом метода Term, который осуществляет анализ конструкции *<слагаемое>*, проверим есть ли перед первым слагаемым знаки «+» или «-». Если знак есть, то нужно проверить правильно ли записан знак перед слагаемым. Далее пока встречается одна из аддитивных операций, то запоминаем операцию в operation и следующее слагаемое в right. Проверяем совместимость типов слагаемых:

/\* операнды аддитивных операций \*/

private CType AdditiveOperands(CType left, EOperation op, CType right)

{

/\* оба типа оказались неизвестными \*/

if (left.Type == EValueType.Unknown && right.Type == EValueType.Unknown)

return left;

/\* типы совпадают или левый операнд приводим к правому операнду и наоборот \*/

if (left.isDerivedFrom(right) || right.isDerivedFrom(left))

{

/\* привести к одному типу \*/

CType castType = TypeCasting(left, right);

if (castType.Type == EValueType.Integer

|| castType.Type == EValueType.Real)

return castType;

if (castType.Type == EValueType.String && op == EOperation.Plus)

return castType;

if (castType.Type == EValueType.Boolean && op == EOperation.Or)

return castType;

}

/\* типы неприводимы друг к другу/недопустимая операция с типом \*/

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errTypeMismatch));

return availableTypes["unknown"];

}

Метод TypeCasting преобразовывает тип одного операнда к типу другого операнда по правилам приведения. Если это невозможно, то будет возвращён тип unknown. Значение, возвращённое в результате вызова метода AdditiveOperands, присваиваем переменной left и либо возвращаем, либо продолжаем анализировать простое выражение.

Код метода Term:

/\* слагаемое \*/

CType Term()

{

var left = Factor();

while (IsOperation(new List<EOperation>() { EOperation.Mul, EOperation.Division, EOperation.Mod, EOperation.Div, EOperation.And }))

{

var operation = ((OperationToken)curToken).OperType;

GetNextToken();

var right = Factor();

left = MultiplyOperands(left, operation, right);

}

return left;

}

Проверка совместимости типов множителей осуществляется в методе MultiplyOperands. Данный метод описывается аналогично методу AdditiveOperands. Здесь лишь отметим, что при делении (/) тип возвращаемого значения всегда будет real, а операции div, mod возможны только для целочисленного типа.

Рассмотрим последний метод, подвергшийся модификации. Код метода Factor представлен ниже:

/\* множитель \*/

CType Factor ()

{

CType expr = availableTypes["unknown"];

if (IsIdentOrConst(new List<ETokenType>() { ETokenType.Const }))

{

switch (((ConstValueToken)curToken).ConstVal.ValueType)

{

case EValueType.Integer:

expr = availableTypes["integer"]; break;

case EValueType.Real:

expr = availableTypes["real"]; break;

case EValueType.String:

expr = availableTypes["string"]; break;

case EValueType.Boolean:

expr = availableTypes["boolean"]; break;

}

GetNextToken();

}

else if (IsIdentOrConst(new List<ETokenType>() { ETokenType.Identifier }))

{

if (variables.ContainsKey(((IdentifierToken)curToken).IdentifierName))

expr = variables[((IdentifierToken)curToken).IdentifierName].Type;

else

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errUnknownIdent));

GetNextToken();

}

else if (IsOperation(new List<EOperation>() { EOperation.LeftBracket }))

{

GetNextToken();

expr = Expression();

Accept(EOperation.RightBracket);

}

else if (IsOperation(new List<EOperation>() { EOperation.Not }))

{

GetNextToken();

expr = Factor();

if (expr.Type != EValueType.Boolean && expr.Type != EValueType.Unknown)

{

errManager.AddError(new CompilerError(curToken.Line, curToken.Col, EErrorType.errTypeMismatch));

expr = availableTypes["unknown"];

}

}

return expr;

}

При получении токена константы определяется тип константы. Если будет считан токен идентификатора, то название идентификатора ищется в области видимости. При считывании ключевого слова *not* осуществляется проверка того, что множитель имеет логический тип.

Рассмотрим каким образом осуществляется продолжение процесса компиляции после обнаружения первой ошибки (нейтрализация семантических ошибок). Для идентификаторов, объявленных повторно, использовался тот тип, который был описан первым. Ошибки в объявлении типа или ошибки, связанные с использованием необъявленного идентификатора, устраняются путём определения типа как неизвестного. Если операнды не могут быть приведены к одинаковому типу, то тип также становится неизвестным. Так как неизвестный тип может быть приведён к любому известному типу, то дальнейшие ошибки, связанные с правильностью определения этого типа, обнаруживаться не будут.

# Тестирование

В этом разделе протестирована работа семантического анализатора на различных входных данных.

Пример листинга корректной программы:

1. program prog1;

2. var m, n:integer;

3. Y: real;

4. flag: boolean;

5. begin

6. m := 15;

7. n := 45;

8. Y := (m + n) / 12;

9. while m <> n do

10. begin

11. if m > n then

12. m := m - n

13. else

14. n := n - m;

15. end;

16. if Y / 2 > m then

17. flag := true;

18. if flag then

19. begin

20. Y := Y / 2;

21. flag := not flag;

22. end

23. else

24. Y := m div 2;

25. end.

Количество ошибок: 0

Пример листинга некорректной программы с сообщениями об ошибках, обнаруженных семантическим анализатором:

1. program prog2;

2. var a, b, c, e: integer;

3. a, b, d, superExpr: real;

^

\*\*\*\*[Error] Code 25: Duplicate identifier\*\*\*\*

^

\*\*\*\*[Error] Code 25: Duplicate identifier\*\*\*\*

4. c: integer;

^

\*\*\*\*[Error] Code 25: Duplicate identifier\*\*\*\*

5. s: string;

6. bool: bolean;

^

\*\*\*\*[Error] Code 27: Error in type definiton\*\*\*\*

7. begin

8. a := -13;

9. b := 0.3;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

10.

11. abcd := 'abcd';

^

\*\*\*\*[Error] Code 26: Unknown identifier\*\*\*\*

12. f := 'abcd' + 1;

^

\*\*\*\*[Error] Code 26: Unknown identifier\*\*\*\*

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

13.

14. if (a > 13) and (a + b < 13.03) then

15. c := (13 - 9) / 2;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

16.

17. if ('abc' > 'bcd') then e := 13 + 0.5;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

18.

19. while a do begin

^

\*\*\*\*[Error] Code 29: Expression type must be boolean\*\*\*\*

20. a := a + 1;

21. e := e / 2;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

22. end;

23.

24. superExpr := a \* a \* a - 3 \* a \* a \* b + 3 \* a \* b \* b - b \* b \* b;

25. e := (a + b) \* (a \* a + a \* b + b \* b) div superExpr;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

26.

27. s := -'a+b' + (s - 'a');

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

28.

29. while not a do c:=c+1;

^

\*\*\*\*[Error] Code 28: Type mismatch\*\*\*\*

30.

31. while not true do c:=c-2;

32. end.

Количество ошибок: 16