|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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.

До текущего момента, разработанный нами синтаксический анализатор, функционировал только до обнаружения первой синтаксической ошибки. То есть обнаружение неправильной синтаксической конструкции приводило к прекращению процесса компиляции. Действия, позволяющие продолжить процесс анализа текста программы после нахождения ошибки, называют нейтрализацией ошибки.

Требуется модифицировать наш компилятор таким образом, чтобы после обнаружения ошибки, процесс компиляция не прекращался.

# Проектирование

Основная идея нейтрализации синтаксических ошибок заключается в следующем: после обнаружения ошибки пропустить какую-то часть программы до того момента, с которого можно возобновить нормальный анализ.

При обнаружении ошибки будем пропускать символы до тех пор, пока не встретим символ, ожидаемый в рамках этой конструкции, либо пока не встретим символ, который ожидается после всей анализируемой конструкции в целом. Все правила, описывающие синтаксис языка, представлены в виде БНФ. В ходе проектирования синтаксического анализатора для каждой БНФ уже была описана своя функция. Поэтому предстоит лишь дополнить имеющиеся функции.

Работа с ошибками осуществляется в классе ErrorManager, который позволяет внести ошибку в список ошибок и/или вывести её в выходной файл. Экземпляр этого класса является одним из атрибутов класса компилятора CCompiler. Для правильного позиционирования ошибок в тексте программы было принято решение хранить позицию в самих в токенах.

# Разработка

Для нейтрализации синтаксических ошибок нужно разместить блоки try-catch в ключевых местах анализа БНФ, а именно, в тех местах, где ожидается определённая конструкция языка. В ходе разработки синтаксического анализатора был реализован метод Accept(), который сравнивает ожидаемый символ с текущим. Если ожидаемый и текущий символы не совпадают, то генерируется исключение, иначе считывается следующий символ. Исключение, сгенерированное в ходе вызова метода Accept(), ловится блоком catch либо в том месте, где вызывалась эта проверка, либо в другой БНФ, если символы, идущие после этой конструкции, могут отличаться в зависимости от контекста.

Если находится не соответствующий анализируемой конструкции символ, то нужно пропустить определённое число символов. Для этого используется метод SkipTo:

/\* пропустить символы пока не встретится один из ожидаемых символов или null \*/

void SkipTo(List<ETokenType> idents, List<EOperation> opers)

{

while (curToken != null && !IsOperation(opers) && !IsIdentOrConst(idents))

GetNextToken();

}

Рассмотрим каким образом была реализована нейтрализация синтаксических ошибок в конструкциях *<составной оператор>* и *<условный оператор>*.

Тело метода CompoundStatement()

try

{

Accept(EOperation.Begin);

}

catch (CompilerError)

{

SkipTo(new List<ETokenType> { }, new List<EOperation> { EOperation.Begin, EOperation.End });

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

GetNextToken();

}

try

{

Statement();

}

catch (CompilerError)

{

SkipTo(new List<ETokenType> { }, new List<EOperation> { EOperation.Semicolon, EOperation.End });

}

while (IsOperation(new List<EOperation>() { EOperation.Semicolon }))

{

GetNextToken();

try

{

Statement();

}

catch (CompilerError)

{

SkipTo(new List<ETokenType> { }, new List<EOperation> { EOperation.Semicolon, EOperation.End });

}

}

Accept(EOperation.End);

Первая возможная ситуация – отсутствие символа ‘begin’. В таком случае будем пропускать следующие символы до тех пор, пока не встретим символы ‘begin’ или ‘end’. Если ‘begin’ нашёлся, то считаем также следующий символ для того, чтобы можно было анализировать конструкцию *<оператор>.* Следующая ситуация, которая вероятно может произойти – это ошибка в одном из операторов. При возникновении такой ошибки будем пытаться восстановить анализ после считывания символа ‘;’ или символа ‘end’. После завершения анализа операторов обязательно должен встретиться символ ‘end’. Так как после ‘end’ могут встретиться разные символы (‘;’, ‘.’), то при возникновении ошибки она будет пробросана в объемлющую БНФ (из которой была вызвана данная БНФ) .

Тело метода IfStatement():

GetNextToken();

try

{

var expr = Expression();

if (!expr.isDerivedFrom(availableTypes["boolean"]) && expr.Type != EValueType.Unknown)

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

}

catch (CompilerError)

{

SkipTo(new List<ETokenType> { ETokenType.Identifier }, new List<EOperation>

{ EOperation.Then, EOperation.Begin, EOperation.If, EOperation.While, EOperation.Semicolon, EOperation.End });

}

try

{

Accept(EOperation.Then);

}

catch (CompilerError)

{

SkipTo(new List<ETokenType> { ETokenType.Identifier }, new List<EOperation>

{ EOperation.Then, EOperation.Begin, EOperation.If, EOperation.While, EOperation.Semicolon, EOperation.End });

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

GetNextToken();

}

Statement();

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

{

GetNextToken();

Statement();

}

Сначала проверяется, что после анализа конструкции *<выражение>* не было получено ошибок. В таких конструкциях, как *<выражение>*, *<простое выражение>*, *<слагаемое>,* *<множитель>* нет блоков try-catch, так как при их анализе мы не знаем до каких символов нужно пропустить, чтобы возобновить анализ конструкции. При возникновении ошибки в одной из перечисленных выше конструкций надо пропустить или до символа ‘then’, или до одного из символов начала описания оператора, или до символов ‘;’ и ‘end’. При отсутствии ‘then’ также будем пропускать до этих же символов. Так как анализ условного выражения может быть вызван только в составном операторе, то ошибки в операторах будут проанализированы в конструкции *<составной оператор>.*

# Тестирование

В этом разделе приведены тесты на нейтрализацию синтаксических ошибок

1. Нейтрализация ошибок в разделе объявления переменных

1. program prog;

2. var 1m, b,, n:integer;

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

3. s string;

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

4. flag boolean

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

5. , Y: real;

^

\*\*\*\*[Error] Code 4: ';' expected\*\*\*\*

6. 5: integer;

^

\*\*\*\*[Error] Code 10: 'begin' expected\*\*\*\*

7. begin

8.

9. end.

Количество ошибок: 6

2. Нейтрализация ошибок в разделе операторов

1. program prog;

2. var m, n:integer;

3. Y: real;

4. zabyl: \*6;

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

5. flag : boolean;

6. begin

7. m = 15;

^

\*\*\*\*[Error] Code 14: ':=' expected\*\*\*\*

8. n := (52+652;

^

\*\*\*\*[Error] Code 15: ')' expected\*\*\*\*

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

10. while m <> n

11. begin

^

\*\*\*\*[Error] Code 17: 'do' expected\*\*\*\*

12. if m := n then

^

\*\*\*\*[Error] Code 28: Expression type must be boolean\*\*\*\*

^

\*\*\*\*[Error] Code 16: 'then' expected\*\*\*\*

^

\*\*\*\*[Error] Code 14: ':=' expected\*\*\*\*

13. m := m - n;

14. else

^

\*\*\*\*[Error] Code 23: Error in statement\*\*\*\*

15. n :5= n - m;

16. end;

17. if Y / 2 != m then

^

\*\*\*\*[Error] Code 18: Illegal character\*\*\*\*

18. flag := true;

19. if flag

20. begin

^

\*\*\*\*[Error] Code 16: 'then' expected\*\*\*\*

21. Y := Y / 2;

22. flag := not flag;

23. end

24. else

25. Y : m div 2

^

\*\*\*\*[Error] Code 14: ':=' expected\*\*\*\*

26. end.

Количество ошибок: 11

Примечание к примеру: ошибка в строке 17 – лексическая ошибка, ошибка в строке 12 (код 28) – семантическая.

3. Ошибка в выражении цикла с предусловием; отсутствие ‘end’ в составном операторе, отсутствие символа ‘.’ после ‘end’

1. program prog;

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 == 5 do

^

\*\*\*\*[Error] Code 29: Syntax error\*\*\*\*

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. else

^

\*\*\*\*[Error] Code 23: Error in statement\*\*\*\*

23. Y : m div 2;

24. end;

25. end

^

\*\*\*\*[Error] Code 5: '.' expected\*\*\*\*

Количество ошибок: 3

4. Ошибка в операторе присваивания; ошибка в выражении цикла с предусловием; ошибка, связанная с отсутствием ‘begin’ в составном операторе

1. program prog;

2. var m, n:integer;

3. Y: real;

4. flag: boolean;

5. begin

6. 5 := 15;

^

\*\*\*\*[Error] Code 23: Error in statement\*\*\*\*

7. n := 45;

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

9. while m = 5) do

^

\*\*\*\*[Error] Code 17: 'do' expected\*\*\*\*

10. if m > n then

11. m := m - n

12. else

13. n := n - m;

14. end;

^

\*\*\*\*[Error] Code 5: '.' expected\*\*\*\*

15. if Y / 2 > m then

16. flag := true;

17. if flag then

18. begin

19. Y := Y / 2;

20. flag := not flag;

21. else

22. Y := m div 2;

23. end;

24. end.

Количество ошибок: 3

5. Пропущенный begin в разделе операторов (в выводе также семантические ошибки)

1. program prog;

2. var m, n:integer;

3. Y: real;

4. flag: boolean;

5.

6. m := 15;

^

\*\*\*\*[Error] Code 24: Duplicate identifier\*\*\*\*

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

7. n := 45;

^

\*\*\*\*[Error] Code 24: Duplicate identifier\*\*\*\*

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

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

^

\*\*\*\*[Error] Code 24: Duplicate identifier\*\*\*\*

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

^

\*\*\*\*[Error] Code 4: ';' expected\*\*\*\*

^

\*\*\*\*[Error] Code 24: Duplicate identifier\*\*\*\*

^

\*\*\*\*[Error] Code 9: ':' expected\*\*\*\*

^

\*\*\*\*[Error] Code 1: Ident expected\*\*\*\*

9. while m = 5) do

^

\*\*\*\*[Error] Code 10: 'begin' expected\*\*\*\*

10. if m > n then

11. m := m - n

12. else

13. n := n - m;

14. end;

^

\*\*\*\*[Error] Code 5: '.' expected\*\*\*\*

15. if Y / 2 > m then

16. flag := true;

17. if flag then

18. begin

19. Y := Y / 2;

20. flag := not flag;

21. else

22. Y := m div 2;

23. end;

24. end.

Количество ошибок: 14

На этом примере отчётливо видно, что нейтрализация ошибок имеет свой недостаток – появление наведённых ошибок.