МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра информатики и вычислительного эксперимента

Мовчан Егор Витальевич

РЕАЛИЗАЦИЯ СИНТАКСИЧЕСКОГО И СЕМАНТИЧЕСКОГО АНАЛИЗАТОРА ЯЗЫКА SPYTHON В АРХИТЕКТУРЕ PASCALABC.NET

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
зав. кафедрой, к. ф.-м. н. Михалкович Станислав Станиславович

Допущено к защите:
руководитель направления 02.03.02 _____ Михалкович С. С.

Ростов-на-Дону – 2025

Содержание

Введение

Компиляция — процесс преобразования кода, написанного на языке программирования, в набор команд, непосредственно выполняемых машиной. Программа, выполняющая компиляцию, называется компилятором.

Стадии компиляции подразделяются на:

1) Фронтенд:
    а) лексический анализ;
    б) синтаксический анализ;
    в) семантический анализ.
2) Бэкенд:
    а) оптимизация программы;
    б) генерация кода.

Лексический анализ — процесс преобразования исходного текста программы из потока символов в поток лексем. Лексема (токен) — минимально значимая единица программы.

Синтаксический анализ — процесс построения синтаксического дерева программы из потока лексем.

Семантический анализ — процесс преобразования синтаксического дерева в семантическое дерево программы, содержащее метаинформацию.

Программы, выполняющие синтаксический и семантический анализ, называются соответственно синтаксическим и семантическим анализаторами.

Интерпретация — альтернативный подход к выполнению программ на языке программирования, который заключается в трансляции исходного кода в машинные команды по ходу выполнения программы. Программа, выполняющая интерпретацию, называется интерпретатором.

Компилируемые языки программирования предпочтительнее для обучения по сравнению с интерпретируемыми языками, так как они:

— гарантируют отсутствие большего числа ошибок при успешной компиляции;

— позволяют создавать более эффективные программы;

— обладают мощными инструментами отладки кода.

Система типов — набор правил, описывающих доступные типы в языке программирования, операции над ними и способы преобразования типов выражений и переменных.

В основном выделяют следующие системы типов:

— статическая типизация — тип всех выражений известен на этапе компиляции и не может меняться;

— динамическая типизация — тип переменной может меняться по ходу выполнения программы.

Статическая типизация позволяет выявлять большинство семантических ошибок в программе, что крайне важно при обучении первому языку программирования.

Python — интерпретируемый язык программирования с динамической типизацией. Он популярен среди школьников благодаря простому синтаксису. Тем не менее, у языка есть ряд особенностей, которые делают его не самым подходящим выбором в качестве первого языка программирования. К ним относятся:

— не всегда понятные сообщения об ошибках;

— выявление большинства ошибок только во время выполнения;

— низкая производительность программ;

— сложности с установкой и настройкой среды у начинающих.

Альтернативой выступает язык SPython — компилируемый язык программирования со статической типизацией и синтаксисом, максимально приближенным к синтаксису языка Python. Он реализован в архитектуре PascalABC.NET [1]. Язык SPython лишён недостатков Python, сохраняет большинство его синтаксических конструкций, а процесс его установки прост. Это делает SPython отличной заменой Python в образовательных целях.

Постановка задачи

Цель работы — реализовать подмножество языка Python со статической типизацией в архитектуре системы программирования PascalABC.NET.

Для достижения указанной цели решить следующие задачи:

1) Разработать лексический анализатор SPython с учетом синтаксиса отступов в языке Python

2) Разработать грамматику языка SPython

3) Реализовать данную грамматику в генераторе компиляторов GPPG

4) Написать парсер SPython, переводящий программу на языке SPython в синтаксическое дерево системы программирования PascalABC.NET

5) Реализовать семантический анализатор SPython

6) Разработать сценарий перевода ряда сложных конструкций SPython в AST-дерево PascalABC.NET, в числе которых локальные и глобальные переменные, параметры функций вида * и **, возможность чередовать операторы и объявления функций Python, реализацию import и from

7) Обеспечить возможность подключения модулей на PascalABC.Net на SPython и наоборот

8) Провести тестирование SPython по времени работы, сравнив с языками Python и PascalABC.NET

1 Предварительная обработка текста и лексический анализ

Синтаксис языка Python использует отступы для обозначения вложенных блоков кода, а символ перевода строки — для определения конца оператора. Однако генератор лексических анализаторов GPLEX [2], использованный в данной работе, не позволяет реализовать полноценный лексический анализатор с учётом этих особенностей. Для решения этой проблемы текст программы сначала подвергается предварительной обработке.

Предварительная обработка текста включает:

1) удаление комментариев;
2) вставку вспомогательных токенов, обозначающих:
    а) начало блока кода;
    б) конец блока кода;
    в) ошибочный отступ;
    г) конец оператора;
    д) конец программы.

В процессе предварительной обработки текст программы просматривается построчно, при этом игнорируются строки, содержащие только пробельные символы. Последовательные строки объединяются в одну логическую строку в следующих случаях:

— если предыдущая строка заканчивается символом обратной косой черты (\), указывающим на продолжение текущей строки;

— если в программе имеются незакрытые скобки, открытые до текущей строки.

В результате объединения символы обратной косой черты удаляются из текста программы. Пример кода, демонстрирующий оба случая, представлен в листинге 1.1.

Листинг 1.1 — Пример логических строк в SPython, объединённых с помощью обратной косой черты и незакрытых скобок

```
1 a = [                    # 1–5 одна логическая строка
```

```
    1,
    2,
    3
]
if a[0] == 1 and \   # 6–8 одна логическая строка
   a[1] == 2 and \
   a[2] == 3:
      print(a[0]     # 9–12 одна логическая строка
          + a[1]
          + a[2]
           )
```

Алгоритм вставки токенов, обозначающих границы блоков, включает следующие шаги:

1) Анализ текущей строки.

2) Сравнение отступа текущей строки с отступом предыдущей:

— если отступ увеличился — вставляется токен начала блока, после чего осуществляется переход к следующей строке;

— если отступ не изменился — переход к следующей строке без добавления токена;

— если отступ уменьшился — выполняется проверка завершения предыдущих блоков.

3) В случае уменьшения отступа выполняется проверка: существует ли незавершённый блок, соответствующий текущему отступу:

— если такой блок есть — вставляются токены окончания блоков в количестве, соответствующем числу закрываемых блоков;

— если нет — вставляется токен ошибочного отступа.

Токен конца оператора вставляется:

— в конец строки, если после неё не начинается новый блок кода;

— после каждого блока кода, за исключением блоков условных операторов, после которых следует альтернативная ветка.

Подробнее описание предварительной обработки текста и процесса лексического анализа приведено в работе [3].

2 Особенности построения синтаксического дерева программы

Для выполнения синтаксического анализа использовался генератор GPPG [4]. Результатом синтаксического анализа является синтаксическое дерево исходной программы, написанной на языке SPython.

2.1 Специфичные узлы синтаксического дерева

Синтаксическое дерево программ, написанных на языке SPython, строится по аналогии с синтаксическим деревом программ на PascalABC.NET. Однако некоторые конструкции, присутствующие в SPython, не имеют прямых аналогов в PascalABC.NET. В связи с этим для таких конструкций были разработаны специальные синтаксические узлы, необходимые для последующих преобразований дерева. Перед этапом построения семантического дерева эти дополнительные узлы удаляются, что позволяет использовать существующий механизм преобразования синтаксического дерева в семантическое дерево, разработанный для PascalABC.NET.

Далее приводится список этих узлов.

— generator_object
— global_statement
— return_statement
— as_statement, as_statement_list
— import_statement
— from_import_statement

Пояснения к узлам:

Узел generator_object

Узел generator_object представляет собой конструкцию генератора. Такие генераторы позволяют создавать последовательности элементов, выполняя фильтрацию и преобразования элементов другой после-

довательности. В частности, узел generator_object содержится в каждой из строк кода, приведённых в листинге 2.1.

Листинг 2.1 — Пример кода, содержащего узлы generator_object

```
1 a = [i for i in range(5, 10)]          # генератор списка (list comprehension)
2 b = {i * i for i in a}                 # генератор множества (set comprehension)
3 c = {i: -i for i in b if i > 48}       # генератор словаря (dict comprehension)
4 print(any(elem > 64 for elem in b)) # генераторный объект в функции any()
5 print(all(elem < 32 for elem in b)) # генераторный объект в функции all()
```

Узел global_statement

global_statement — узел, представляющий ключевое слово global. Он используется компилятором при разрешении имён для обозначения того, что переменная, объявленная внутри функции, не является локальной, а относится к глобальному пространству имён.

Листинг 2.2 — Пример использования global

```
1 a = 3.14
2
3 def modify_a():
4     global a
5     a = 2.7
```

Узел return_statement

return_statement — синтаксический узел, представляющий инструкцию return, используемую для возврата значения из функции. После выполнения этой инструкции функция немедленно завершает своё выполнение, и управление передаётся в точку вызова. Если функция не возвращает значение, то return используется только для досрочного завершения её выполнения, как показано в листинге 2.4.

Листинг 2.3 — Пример использования return

```
1 def min(a: float, b: float) -> float:
2     if a < b:
3         return a
4     else:
5         return b
```

Листинг 2.4 — Пример использования return, без возвращаемого значения

```
def print_tree(a: tree_node):
    if is_leaf(a):
        return
    print_tree(a.left)
    print(a.value)
    print_tree(a.right)
```

Узлы as_statement, as_statement_list

Узлы as_statement и as_statement_list являются вспомогательными и применяются при построении конструкций import_statement и from_import_statement. Они используются для задания псевдонимов импортируемым модулям и именам, импортируемым из них, позволяя использовать конструкции вида import module as alias и from module import name as alias.

Узел import_statement

Синтаксический узел import_statement представляет инструкцию import, используемую для подключения модулей в программу. При таком способе импорта все имена из модуля становятся доступны только при предварении их именем модуля (например, module.name). Кроме того, с помощью ключевого слова as можно задать псевдоним для имени модуля, что позволяет обращаться к нему только через альтернативное имя, как показано в листинге 2.5.

Листинг 2.5 — Пример использования import

```
import GraphWPF as graph

color = graph.RandomColor()
def MouseMove(x: float, y: float, mb: int):
    if mb == 1:
        graph.FillCircle(x, y, 10, color)

graph.OnMouseMove = MouseMove
```

Узел from_import_statement

Синтаксический узел from_import_statement представляет конструкцию from ... import ..., которая используется для выборочного подключения отдельных имён из модуля. При таком способе импорта выбранные имена попадают напрямую в текущую область видимости и могут использоваться без указания имени модуля. При этом к этим именам нельзя обратиться через имя самого модуля. Также возможно присваивание псевдонимов импортируемым именам с помощью ключевого слова as, что позволяет задавать альтернативные имена для импортируемых сущностей, как показано в листинге 2.6.

Листинг 2.6 — Пример использования from ... import

```
1 from GraphWPF import *
2
3 color = RandomColor()
4 def MouseMove(x: float, y: float, mb: int):
5     if mb == 1:
6         FillCircle(x, y, 10, color)
7
8 OnMouseMove = MouseMove
```

2.2 Форматированные строки

Форматированные строки (далее — *f-строки*) — способ записи строк в Python, позволяющий удобно создавать строки, в которые подставляются значения переменных и выражений с заданным форматом.

В SPython также реализованы *f-строки*. Для написания *f-строки* необходимо добавить к строковому литералу префикс f и выделить фигурными скобками выражение, значение которого необходимо подставить в строку. Если требуется задать формат, отличающийся от формата по умолчанию, его можно указать после выражения через двоеточие: «:», как показано в листинге 2.7. SPython поддерживает в *f-строках* форматы, указанные в таблице 2.1.

Листинг 2.7 — Пример использования *f-строк*

```
1 a = 11 * 16 + 10
2 print(f'{a:x}')       # ba
```

```
3  print(f'{a:X}')       # BA
4  print(f'{a}')         # 186
5  print(f'{a:d}')       # 186
6  print(f'{a:b}')       # 10111010
7  print(f'{a:.5f}')     # 186.00000
8
9  pi = 3.1415926
10 print(f'{pi}')        # 3.1415926
11 print(f'{pi:.0f}')    # 3
12 print(f'{pi:.5f}')    # 3.14159
13 print(f'{pi:.10f}')   # 3.1415926000
```

Формат	**Описание**
x, X	Для целых чисел, записывает их значение в шестнадцатеричной системе счисления (прописные/строчные буквы).
d	Для целых чисел, записывает их значение в десятичной системе счисления (значение по умолчанию).
b	Для целых чисел, записывает их значение в двоичной системе счисления.
.nf	Для чисел с плавающей точкой и целых чисел, позволяет записать значение с точностью до первых n знаков после запятой, где n удовлетворяет неравенствам $0 \leq n \leq 10$.

Таблица 2.1 — Форматы *f-строк* в SPython

Парсинг форматированных строк

Построение узла синтаксического дерева для выражения, представляющего собой *f-строку*, происходит по следующему алгоритому.

Сначала лексический анализатор находит строковые литералы с префиксом f, указывающим на начало *f-строки*. Далее полученная строка разбирается на две группы лексем: обычные текстовые фрагменты и выражения (с возможным указанием формата через двоеточие «:»), заключённые в фигурные скобки.

На этом этапе также проверяется корректность структуры *f-строки*: недопустимо использовать вложенные фигурные скобки, у каждой открывающей скобки должна быть парная ей закрывающая. Если какое-то требование нарушено, то *f-строка* считается неправильно сформированной и возникает ошибка компиляции.

После этого каждая подстрока, содержащаяся в фигурных скобках, обрабатывается отдельно. Такая подстрока может содержать:

— только выражение ({expr});
— выражение с указанием формата ({expr:fmt}).

Для выражения рекурсивно вызывается компилятор, разбирающий данное выражение отдельно, после чего его вхождение в *f-строку* заменяется на str(expr).

Если также указан формат, производится проверка его корректности. В случае, если формат не соответствует ни одному из поддерживаемых, возникает ошибка компиляции с соответствующим сообщением об ошибке. При корректном формате подстрока заменяется на вызов функции format(expr, fmt), которая преобразует значение выражения в строку в соответствии с заданным форматом.

Полученный в обоих случаях результат затем конкатенируется с остальными частями строки.

2.3 Глобальные объявления

Программы на SPython могут содержать глобальные объявления переменных и функций вперемешку с выполняемым кодом. Такие переменные и функции могут быть импортированы в другие программы. Архитектура PascalABC.NET не поддерживает такой подход: сначала должны идти глобальные объявления, а затем — выполняемый код программы.

Для корректных преобразований синтаксического дерева необходимо знать точную позицию каждого объявления, так как внутри самой программы можно использовать только переменные и функции, которые были объявлены выше по тексту (за исключением вызова функции внут-

ри другой функции, объявленной ранее, что позволяет реализовывать взаимно рекурсивные вызовы).

Поэтому для поддержки такого механизма все объявления при парсинге программы сохраняются с помощью узла declarations_as_statement как элементы типа statement вместе с выполняемым кодом. При таком подходе после завершения работы синтаксического анализатора формируется синтаксическое дерево без явных глобальных объявлений. Затем, когда информация о точном расположении объявлений в тексте уже не требуется, глобальные объявления выносятся в начало программы с помощью преобразований синтаксического дерева.

2.4 Поддержка модулей в языке SPython

Модули в языке SPython синтаксически не отличаются от обычных программ. Ключевое различие заключается в способе их использования: если файл запускается напрямую, он рассматривается как основная программа; если же он подключается к другой программе через оператор импорта, то рассматривается как модуль.

Для реализации данной функциональности в компиляторе была добавлена возможность определить, является ли текущий файл модулем или основной программой.

Процесс построения синтаксического дерева для модулей и обычных программ различается только типом корневого узла. При этом структура самой программы остаётся неизменной: все функции, объявленные внутри, доступны при подключении в качестве модуля, а все переменные, определённые на внешнем уровне, считаются глобальными и также доступны извне.

В листинге 2.8 представлен пример программы на языке SPython, которая может быть использована в качестве модуля. После подключения этого файла как модуля в другой программе, его функции и глобальные переменные станут доступными для вызова и использования.

Листинг 2.8 — Пример модуля в SPython

```
1 pi = 3.14
```

```
2
3  def fibonacci(n: int)-> bigint:
4      if n <= 1:
5          return n
6      f1 = 0bi
7      f2 = 1bi
8      for i in range(n - 1):
9          t = f1 + f2
10         f1 = f2
11         f2 = t
12     return f2
```

3 Преобразования синтаксического дерева программы

Результатом работы парсера является «сырое» синтаксическое дерево программы. Перед построением семантического дерева это синтаксическое дерево подвергается множеству преобразований с использованием шаблона визитор.

Визитор — поведенческий шаблон проектирования, упрощающий добавление нового поведения для обслуживаемых классов без их изменения [5]. Этот шаблон часто используется в компиляторах для реализации проверок корректности программ, их оптимизации и преобразования внутреннего представления.

Визиторы обходят программу последовательно. Между их вызовами существует частичный порядок . На рисунке из приложения А показаны все зависимости между визиторами. Если между соседними визиторами нет зависимости, их вызовы можно поменять местами, не нарушая логики работы компилятора.

Для понимания логики преобразований далее разобраны все визиторы, написанные для языка SPython, в порядке их вызова над синтаксическим деревом программы.

3.1 UnsupportedConstructsVisitor

Данный визитор проверяет программу на наличие следующих ошибок:

— любое использование return вне функции;

— использование return внутри функции, которая возвращает значение;

— использование return expr внутри функции, которая не возвращает значение;

— использование global вне функции;

— использование global во вложенном блоке кода внутри функции;

— использование конструкции import. . . не на глобальном уровне;

— использование конструкции from... import... не на глобальном уровне;

— объявление локальных функций.

При обнаружении любой из перечисленных ошибок компилятор выдаёт пользователю соответствующее сообщение об ошибке. Например, если запустить программу из листинга 3.1, возникнет ошибка компиляции: «конструкция «global...» недопустима во вложенном блоке кода».

Листинг 3.1 — Пример неправильного использования global

```
1 a = 10
2
3 def modify_a(flag: bool):
4     if flag:
5         global a      # Ошибка: global во вложенном блоке
6     a = 20
```

3.2 ImportToUsesVisitor

Визитор ImportToUsesVisitor посещает узлы import_statement и from_import_statement и выносит из них имена модулей в секцию Uses. Это обеспечивает подключение всех модулей, упомянутых в соответствующих операторах. Так, для программы из листинга 3.2, будут добавлены узлы, подключающие модули time и random.

Листинг 3.2 — Программа, подключающая модули time и random

```
1 a = 20
2 from time import time
3 b = time()
4 from random import random
5 c = random()
6 print(a, b, c)
```

3.3 BitwiseAssignmentDesugarVisitor

Задачей этого визитора является обеспечение поддержки кратких форм присваиваний с побитовыми операциями, а именно:

— побитовое «и» (&=);

— побитовое «или» (|=);

— побитовое «исключающее или» (^=);

— сдвиг влево (<<=);

— сдвиг вправо (>>=).

В PascalABC.NET таких операторов нет, поэтому данный визитор находит такие присваивания и заменяет их на эквивалент с обычным присваиванием. Например, выражение

$$a \&= b$$

преобразуется в

$$a = a \& b$$

и аналогично для остальных операторов.

3.4 ReturnDesugarVisitor

Данный визитор работает с узлами return_statement. Инструкция return может встретиться в двух формах:

— return

— return expr

В первом случае компилятор заменяет инструкцию вызовом функции exit(), которая досрочно завершает выполнение функции. Во втором случае, компилятор также добавляет сохранение значения expr в переменную result.

Например, после применения этого визитора к программе из листинга 3.3 получится синтаксическое дерево, аналогичное программе из листинга 3.4.

Листинг 3.3 — Программа до приминения ReturnDesugarVisitor

```
1 def foo(a: int) -> int:
2     return a + 2
3
4 def bar(b: float):
5     if b < 0:
6         return
7     print(b)
```

Листинг 3.4 — Программа после приминения ReturnDesugarVisitor

```
1 def foo(a: int) -> int:
2     result = a + 2
3     exit()
4
5 def bar(b: float):
6     if b < 0:
7         exit()
8     print(b)
```

3.5 MapDesugarVisitor

Визитор MapDesugarVisitor реализует поддержку функции map(function, sequence) из стандартной библиотеки Python. Эта функция применяет заданную функцию function ко всем элементам последовательности sequence.

Для обеспечения такого функционала данный визитор заменяет вхождения вызова функции map с двумя аргументами на другой синтаксический узел generator_object, который эквивалентен конструкции function(temp) for temp in sequence. Далее этот узел обрабатывается визитором GeneratorObjectDesugarVisitor.

3.6 GeneratorObjectDesugarVisitor

Данный визитор используется для сахарного преобразования конструкций одного из двух видов:

— function(i) for i in sequence
— function(i) for i in sequence if condition(i)

В первом случае визитор заменяет конструкцию на выражение sequence.Select(i -> function(i)), во втором — на sequence.Where(i -> condition(i)).Select(i -> function(i)).

Например, выражение map(int, input().split()) сначала преобразуется визитором MapDesugarVisitor в int(temp) for temp in input().split(), которое затем визитор GeneratorObjectDesugarVisitor преобразует в input().split().Select(temp -> int(temp)).

3.7 CompoundComparisonDesugarVisitor

Задачей данного визитора является поддержка цепочек сравнений. Пример их использования приведён в листинге 3.5.

Листинг 3.5 — Пример использования цепочек сравнений

```
1 def median(a: int, b: int, c: int) -> int:
2     if b <= a <= c or c <= a <= b:
3         return a
4     if a <= b <= c or c <= b <= a:
5         return b
6     return c
```

В общем виде цепочка сравнений записывается как:

$$expr_1\ op_1\ expr_2\ op_2\ expr_3 \ldots$$

где каждый op_i — один из операторов сравнения: $<$, $>$, $<=$, $>=$, $==$, $!=$.

Для упрощения разберём действие визитора на примере выражения:

$$e_1 < e_2 < e_3 \tag{3.1}$$

где e_i — произвольное выражение, $i = \overline{1,3}$.

Простая замена выражения 3.1 на:

$$e_1 < e_2\ and\ e_2 < e_3$$

работает неправильно, если e_2 — вызов функции, имеющей побочные эффекты (поскольку она будет вычислена дважды, хотя должна — только один раз). Для устранения этой проблемы можно сохранить значение e_2 в промежуточную переменную, объявленную перед текущим оператором. Данный подход тоже не всегда работает правильно, что демонстрирует листинг 3.6.

Листинг 3.6 — Контрпример: преждевременное вычисление выражения может вызвать ошибку

```
1 def foo(a: int, b: int, c: int) -> int:
2     if a != 0 and c / 2 < b / a < c:
3         return 42
4     return -1
```

При такой реализации выражение b / a вычислится даже при a = 0, что приведёт к ошибке времени выполнения, хотя её быть не должно.

Для корректной реализации визитор использует обобщённую функцию assign(to, from), которая принимает первым аргументом переменную по ссылке и присваивает ей значение второго аргумента. Таким образом можно перенести вычисление выражения в момент его первого использования и не вычислять его дважды.

Для данного примера визитор должен выполнить следующие шаги:

1) Перед текущим оператором добавить объявление переменной temp, соответствующей типу выражения e_2.

2) Заменить выражение 3.1 на:

$$e_1 < assign(temp, e_2)\ and\ temp < e_3$$

Визитор CompoundComparisonDesugarVisitor реализует данную замену цепочки сравнений для произвольного числа выражений и произвольных операций сравнения.

3.8 TypeCorrectVisitor

Данный визитор производит замену стандартных типов языка SPython на эквивалентные типы языка PascalABC.NET согласно таблице 3.1. Помимо этого, визитор проверяет, что пользователь не пытается использовать стандартные типы языка PascalABC.NET в программе на языке SPython. В противном случае возникает ошибка компиляции с предложением использовать правильное имя типа. Например, для программы из листинга 3.7 будет выведена следующая ошибка компиляции: «Неизвестный тип «integer», возможно, Вы имели в виду «int»».

Листинг 3.7 — Пример ошибочной программы: использование типа integer

```
1 def foo(a: integer) -> float:
2     return a * 0.5
```

Таблица 3.1 — Эквивалентность типов в языках SPython и PascalABC.NET

SPython	**PascalABC.NET**
int	Integer
float	Real
str	String
bool	Boolean
bigint	System.BigInteger
list	System.List
set	System.NewSet
dict	System.Dictionary

3.9 NameCorrectVisitor

Визитор NameCorrectVisitor отвечает за первичное разрешение имён в программе. В его задачи входит:

1) замена псевдонимов на действительные имена;

2) выбор модуля для импортированных имён;

3) определение, является ли присваивание объявлением с инициализацией;

4) проверка программы на наличие следующих ошибок:

   а) использование неизвестного имени;

   б) использование внутри global имени, которое уже присутствует в текущем пространстве имён (см. листинг 3.8);

   в) использование функции, объявленной позже (см. листинг 3.9);

   г) инициализация контейнеров пустым контейнером без указания типа элементов (см. листинг 3.10).

Помимо этого, данный визитор сохраняет для визитора RetainUsedGlobalVariablesVisitor множество переменных, которые были использованы как глобальные (переменная использована как глобальная, если она объявленна на глобальном уровне и использована внутри функций).

Примеры обрабатываемых ошибок

Ниже приведены примеры ошибок, которые обнаруживает визитор NameCorrectVisitor.

Листинг 3.8 — Использование global с локальной переменной

```
1 a = 10
2
3 def foo():
4     a = 20
5     global a   # Ошибка: имя «a» присутствует в текущем пространстве имён
```

Листинг 3.9 — Использование функции до определения

```
1 print(foo())   # Ошибка: неизвестное имя «foo»
2
3 def foo() -> int:
4     return bar()   # Внутри другой функции можно
5
6 def bar() -> int:
7     return 42
```

Листинг 3.10 — Инициализация контейнеров

```
1 a: list[int] = [1, 2, 3]   # тип a — list[int]
2 b: dict[str, str] = {}     # тип b — dict[str, str]
3 c = {4.1, 5.2, 6}          # тип c — set[float]
4 d = []                     # Ошибка: нельзя вычислить тип списка
```

Алгоритм работы

Перед вызовом этого визитора происходит заполнение словаря, в котором каждому импортируемому модулю ставится в соответствие множество глобальных имён, объявленных в нём. Далее данный словарь передаётся визитору через конструктор.

Программно визитор разделён на две части: родительский класс — SymbolTableFillingVisitor, функциональность которого ограничена динамической поддержкой таблицы символов, и сам визитор NameCorrectVisitor, который занимается выявлением ошибок и разрешением имён.

При обходе синтаксического дерева визитор поддерживает словарь текущих имён в программе и для каждого имени определяет его тип. Возможны следующие типы имён в программе:

— локальное имя;
— ключевое слово языка;
— имя глобальной переменной;
— псевдоним модуля;
— псевдоним имени из модуля;
— имя функции, объявленной до текущего места;
— имя функции, объявленной после текущего места.

Каждый вложенный блок кода создаёт новое пространство имён, в котором могут быть объявлены локальные переменные. При поиске типа имени локальные имена являются приоритетными; они действуют до конца блока кода, в котором были объявлены. Такая легковесная таблица символов для локальных имён реализована по алгоритму, описанному в главе 2.7 книги[6].

Все остальные имена считаются глобальными. Новое глобальное имя может переопределить ранее объявленное имя, за исключением случаев, когда это имя является ключевым словом языка. Пример переопределения показан в листинге 3.11.

Листинг 3.11 — Пример переопределения глобального имени

```
1  from a import foo
2  def bar():
3      foo()              # Вызов функции a.foo()
4
5  foo()                  # Вызов функции a.foo()
6  from b import foo
7  def baz():
8      foo()              # Вызов функции b.foo()
9
10 foo()                  # Вызов функции b.foo()
11
12 foo: int = 25          # Объявление переменной foo типа int
13 print(foo)             # Вывод: 25
```

Для псевдонимов визитор также сохраняет настоящее имя, а для имён из модулей — имя модуля.

Внутри контекста функции визитор проверяет, что используются только:

— локальные имена (в том числе параметры функции);

— функции, объявленные в текущей программе (в том числе те, что объявлены позже);

— глобальные имена, добавленные в текущий контекст через global...;

— ключевые слова языка.

Вне функций визитор запрещает использование функций, объявленных в программе после текущего места.

Когда визитор встречает псевдоним, он заменяет его на настоящее имя. Для имён, импортируемых из модуля, он добавляет в начало имя модуля. Таким образом, программа из листинга 3.12 преобразуется визитором в программу, приведённую в листинге 3.13.

Листинг 3.12 — Программа до разрешения имён

```
1 from m1 import a, b as c
2 import m2 as m
3
4 print(m.d)
5 print(a)
6 print(c)
```

Листинг 3.13 — Программа после разрешения имён

```
1 from m1 import a, b as c
2 import m2 as m
3
4 print(m2.d)      # Замена псевдонима модуля
5 print(m1.a)      # Добавление имени модуля
6 print(m1.b)      # Замена псевдонима имени и добавление имени модуля
```

При парсинге программы все конструкции вида some_name =... считаются присваиванием. Однако если имя some_name ещё не встречалось в программе, то это считается объявлением новой переменной с инициализацией. Данный визитор заменяет соответствующий узел

синтаксического дерева на нужный, если такое имя отсутствует в таблице символов.

В результате работы визитора программа не содержит ошибок, связанных с использованием псевдонимов: все псевдонимы заменены на настоящие имена, а у каждого импортированного имени указано, из какого модуля оно было получено.

3.10 AddForwardDeclarationsVisitor

Задачей этого визитора является добавление в начало программы forward-объявлений для каждой функции, объявленной в текущей программе. Это делается для того, чтобы внутри функции можно было вызывать функции, объявленные позже. Такое поведение характерно для программ на языке Python, кроме того, без этого невозможно реализовать взаимную рекурсию, как показано в листинге 3.14.

Листинг 3.14 — Пример взаимной рекурсии

```
1 def foo(a: int) -> int:
2     if a == 0:
3         return 0
4     return a * bar(a - 1)
5
6 def bar(b: int) -> int:
7     if b == 0:
8         return 1
9     return b + foo(b - 1)
```

Для этого данный визитор копирует заголовки всех функций, добавляет к ним атрибут forward и выносит полученные заголовки в начало программы.

3.11 KwargsFunctionDesugarVisitor

Данный визитор реализует поддержку механизма объявления функций с kwargs-аргументами. В языке Python конструкция **kwargs предназначена для передачи произвольного числа именованных аргументов в функцию.

Для поддержки этого механизма в стандартную библиотеку SPython был добавлен шаблонный класс kwargs_gen<T>, содержащий одно поле kwargs типа Dictionary<string, T>. Это поле инициализируется в конструкторе.

Рассмотрим работу визитора KwargsFunctionDesugarVisitor на примере функции sum, приведённой в листинге 3.15.

Листинг 3.15 — Пример объявления функции с kwargs-аргументами

```
1 def sum(*args: int, **my_kwargs: int) -> int:
2     res = 0
3     for arg in args:
4         res += arg
5     for k, v in my_kwargs:
6         res += v
7     return res
```

Визитор выполняет следующие шаги:

1) Объявляет глобально класс !sum, наследующийся от kwargs_gen<int>;

2) Добавляет в класс !sum определение метода sum(*args: int) -> int;

3) Заменяет исходное определение функции на определение метода, указанного в пункте (2);

4) Заменяет в (3) все вхождения имени последнего параметра на соответствующее имя поля базового класса;

5) Добавляет внешнюю функцию sum(*args: int) -> int, которая вызывает соответствующий метод класса, передавая пустой словарь;

6) Добавляет forward-объявление для функции из пункта (5).

В результате работы визитора для программы из листинга 3.15 будет сгенерирована программа, приведённая в листинге 3.16.

Листинг 3.16 — Сгенерированный псевдокод для функции sum

```
1 class !sum : kvargs_gen[int]      # Класс из пункта (а)
2     sum(*args: int) -> int        # Объявление метода из пункта (б)
3
4 sum(*args: int) -> int            # Forward-объявление из пункта (е)
5
6 !sum:sum(*args: int) -> int:      # Определение метода из пункта (в)
7     res = 0
```

```
8       for arg in args:
9           res += arg
10      for k, v in kwargs: # Замена имени согласно пункту (г)
11          res += v
12      return res
13
14  sum(*args: int) -> int:          # Функция из пункта (д)
15      (new !sum).sum(args)
```

Кроме того, визитор KwargsFunctionDesugarVisitor выполняет синтаксический контроль заголовка функции: разрешено использование не более одного kwargs-параметра, и он должен располагаться последним в списке параметров. Нарушение этого правила приводит к генерации ошибки компиляции.

Пример некорректного объявления функции с kwargs-параметрами приведён в листинге 3.17.

Листинг 3.17 — Пример некорректного объявления функции с kwargs-параметром

```
1  def foo(a: int, **b: float, *c: str):    # Ошибка: kwarg-параметр не последний
2      print(f'a = {a}')
3      print(f'b = {b}')
4      print(f'c = {c}')
```

3.12 FunctionsWithNamedParametersDesugarVisitor

Данный визитор отвечает за трансформацию вызовов функций, содержащих именованные параметры, в эквивалентную форму, пригодную для выполнения во время выполнения программы.

При обнаружении вызова функции с именованными аргументами визитор генерирует код, в котором создаётся вспомогательный объект (см. раздел 3.11). В конструктор этого объекта передаются два набора аргументов:

— список строк, соответствующий именам именованных параметров;

— список значений, переданных этим параметрам, в виде параметра params.

После создания объекта у него вызывается метод с тем же именем, что и у исходной функции. При этом позиционные аргументы (т. е. неименованные параметры) передаются напрямую в этот метод.

Например, вызов функции `sum(1, 2, 3, a=4, b=5)` преобразуется в следующий эквивалентный вызов:

```
!sum(['a', 'b'], 4, 5).sum(1, 2, 3)
```

Визитор также выполняет проверку корректности расположения именованных параметров: они должны следовать строго после всех позиционных аргументов и не могут быть перемешаны с ними. В противном случае генерируется сообщение об ошибке.

Пример некорректного использования именованных параметров приведён в листинге 3.18.

Листинг 3.18 — Пример некорректного вызова функции с именованными параметрами

```
1 # Ошибка: После именованного аргумента может идти только именованный аргумент
2 print(1, 2, 3, sep='#', 4)
```

3.13 RetainUsedGlobalVariablesVisitor

Визитор RetainUsedGlobalVariablesVisitor отвечает за перенос объявлений переменных, находящихся на внешнем уровне программы, на глобальный уровень, если они используются внутри функций.

Локальные переменные обрабатываются быстрее, чем глобальные, поэтому для повышения производительности глобальными становятся только те переменные, которые действительно используются внутри функций. Множество таких переменных передаётся визитору через конструктор от NameCorrectVisitor.

Если программа компилируется как модуль, то предполагается, что внешняя программа может использовать все имена из данного модуля. В этом случае оптимизация глобальных переменных не производится, чтобы сохранить полную доступность объявлений.

Во время обхода визитор анализирует объявления переменных на внешнем уровне. Если переменная используется внутри функций, визитор:

— выносит её объявление на глобальный уровень;

— заменяет первоначальное объявление на присваивание;

— в случае отсутствия типа при объявлении к присваиванию добавляет специальный флаг first_assignment_defines_type. Этот флаг будет использован на этапе семантического анализа для определения типа переменной по типу правой части оператора присваивания (см. раздел 4.3).

Пример переменных, которые интерпретируются как глобальные, представлен в листинге 3.19.

Листинг 3.19 — Пример глобальных переменных

```
1  a = 10              # глобальная переменная: используется в foo
2  def foo():
3      global a
4      a = 20
5
6  b = 30              # локальная переменная
7                      # (глобальная, если это модуль или будет использована внутри функции)
8  def bar():
9      b = 40          # локальная переменная функции bar
10
11 if True:
12     c = 50          # не может быть глобальной: объявлена не на верхнем уровне
```

3.14 EraseSpythonOnlyNodesVisitor

На последующих этапах компиляции синтаксическое дерево не должно содержать узлов, специфичных для языка SPython, поэтому данный визитор удаляет из дерева следующие узлы:

— global_statement

— import_statement

— from_import_statement

Для этого он заменяет их на empty_statement при обходе дерева.

3.15 TreeNodesRearrangementVisitor

Данный визитор подготавливает синтаксическое дерево к следующему этапу компиляции — построению семантического дерева. Он преобразует структуру синтаксического дерева к следующему виду:

1) forward-объявления функций программы;
2) объявления глобальных переменных;
3) объявление функции main;
4) объявления всех остальных функций программы;
5) тело программы: вызов функции main.

Функция main представляет собой обёртку над основным телом пользовательской программы, в которую помещаются все инструкции, расположенные вне функций. Таким образом, функция main служит точкой входа в программу.

Такое представление синтаксического дерева является эквивалентным исходному, однако значительно упрощает выполнение семантических проверок (см. раздел 4.3).

4 Дополнительные семантические проверки и преобразования

Основные семантические проверки и преобразования программ на языке SPython выполняются по аналогии с соответствующими этапами в языке PascalABC.NET. Однако различия в синтаксисе и семантике этих языков требуют внесения определённых изменений в общую логику анализа.

В разделе 3.9 уже был рассмотрен механизм первичного разрешения имён, реализованный с учётом этих особенностей.

Настоящая глава посвящена другим изменениям в семантической обработке, вызванным спецификой языка SPython.

4.1 Замена методов контейнеров

В языке SPython реализованы три стандартных контейнера языка Python: список list, множество set и словарь dict. Для каждого из них существует эквивалент в языке PascalABC.NET, указанный в таблице 3.1. Несмотря на одинаковое внутреннее устройство, названия методов, реализующих идентичный функционал, различаются. Например, метод добавления элемента в конец списка называется append в Python и Add в PascalABC.NET.

Использование классов-обёрток для адаптации методов привело бы к дополнительным накладным расходам и замедлению выполнения программ, так как каждый вызов метода требовал бы промежуточной переадресации. В связи с этим было принято решение реализовать другой подход — замену имён методов на этапе построения семантического дерева.

Замена имён выполняется с помощью визитора, в процессе построения семантического дерева программы. При посещении узла с вызовом метода визитор определяет тип выражения до точки. Если тип соответствует одному из контейнеров SPython, выполняется проверка имени метода. В случае отсутствия соответствующего метода возникает ошибка

компиляции. Если метод найден, его имя заменяется на соответствующее имя метода контейнера в PascalABC.NET.

Соответствие методов списков приведено в таблице 4.1.

Таблица 4.1 — Соответствие методов списка list в языках SPython и PascalABC.NET

SPython (Python)	**PascalABC.NET**
append	Add
clear	Clear
insert	Insert
remove	Remove
index	IndexOf
sort	Sort
copy	ToList

4.2 Обработка семантических ошибок

При компиляции программ на языке SPython могут возникать ситуации, когда семантические проверки, реализованные для PascalABC.NET, оказываются некорректными или недостаточными. Это обусловлено отличиями в семантике языков. В таких случаях требуется вмешательство в стандартную обработку ошибок компиляции.

Можно выделить три основные ситуации, при которых необходимо переопределение поведения генератора ошибок:

— **Ошибка в PascalABC.NET, но не ошибка в SPython**. В данной ситуации необходимо перехватить сгенерированную ошибку и преобразовать исходный код в эквивалентный код на PascalABC.NET, не нарушающий его правила.

— **Конструкция допустима в SPython, но отсутствует соответствующая проверка в PascalABC.NET**. В этом случае требуется расширение механизма семантического анализа компилятора и добавление соответствующей проверки, чтобы обеспечить правильную диагностику ошибок, специфичных для SPython.

— **Ошибка присутствует и в SPython, и в PascalABC.NET, но сопровождается различным пояснением**. В такой ситуации необходимо перехватить сообщение об ошибке, сформированное компилятором, и заменить его на более релевантное текстовое описание, соответствующее синтаксису и семантике SPython.

Подробнее данный механизм рассмотрен в работе [7].

Компилятор языка SPython генерирует понятные и адаптированные к языку сообщения о семантических ошибках. Для этого был реализован специальный механизм, позволяющий подменять внутренние типы PascalABC.NET на соответствующие типы SPython при формировании текста сообщения. Это повышает читаемость и понятность ошибок для пользователя, изучающего программирование на языке SPython.

4.3 Автоматический вывод типа глобальной переменной

В языке SPython предусмотрено два способа объявления переменных:

— явное объявление с указанием типа;

— неявное объявление с присваиванием, при котором тип переменной выводится на основе выражения в правой части оператора присваивания.

Если переменная является глобальной, её объявление выносится на глобальный уровень, а первому присваиванию устанавливается флаг first_assignment_defines_type, если явно не указан тип (см. раздел 3.13). В случае неявного объявления глобальной переменной для корректного вывода её типа необходимо определить тип выражения в правой части оператора присваивания. Однако в этом выражении могут использоваться локальные переменные (см. листинг 4.1).

Листинг 4.1 — Пример инициализации глобальной переменной с использованием локальной переменной

```
1 b = int(input())      # Локальная переменная
2 a = b                 # Глобальная переменная: используется в функции foo
3
```

```
4 def foo():
5     print(a)          # Использование глобальной переменной a
6
7 foo()
```

Для решения данной проблемы были рассмотрены два подхода:

— считать глобальными все переменные, которые используются в правой части первого присваивания другой глобальной переменной;

— определять тип глобальной переменной в том локальном контексте, где происходит её первое присваивание, и обновлять тип переменной в таблице символов.

Первый вариант приводит к увеличению количества глобальных переменных в программе. Работа с ними менее эффективна по сравнению с локальными переменными, поэтому в целях оптимизации был выбран второй подход.

Для его реализации необходимо, чтобы первое присваивание переменной обрабатывалось раньше всех остальных операций с ней. Поскольку первое присваивание всегда находится вне функций, а внутри функций переменная может уже использоваться, требуется изменить структуру программы. Для этого синтаксическое дерево преобразуется согласно описанию в разделе 3.15.

После такого преобразования каждая глобальная переменная сначала встречается при объявлении, затем — при первом присваивании. К этому моменту уже известны forward-объявления всех функций, что позволяет корректно определить тип выражений, в которых они вызываются.

В алгоритм построения семантического дерева была добавлена следующая логика: при посещении оператора присваивания проверяется флаг first_assignment_defines_type. Если он установлен, вычисляется тип выражения в правой части, и тип переменной из левой части обновляется в таблице символов. Изначально переменной присваивается тип int, чтобы предотвратить ошибку компиляции до выполнения анализа.

5 Взаимодействие SPython и PascalABC.NET посредством модулей

Компилятор SPython поддерживает написание программных модулей (см. раздел 2.4). Одной из ключевых задач при разработке компилятора являлась реализация возможности взаимодействия между языками SPython и PascalABC.NET посредством совместно используемых модулей.

Для этого при компиляции программы в виде модуля генерируется файл с расширением .pcu, представляющий собой независимое промежуточное представление, унифицированное для обоих языков. Это позволяет осуществлять подключение модуля вне зависимости от того, на каком языке он был изначально написан и на каком языке он подключается. Механизм чтения и использования .pcu-файлов одинаков для SPython и PascalABC.NET.

В листинге 5.1 показан пример использования модуля GraphWPF, написанного на языке PascalABC.NET, в программе на SPython. Данная программа создаёт окно, в котором пользователь может рисовать с помощью мыши. Пример более сложной программы, демонстрирующей расширенные возможности, приведён в приложении Б.

Листинг 5.1 — Использование модуля GraphWPF в программе на SPython

```
1  from GraphWPF import *
2
3  color = RandomColor()
4
5  def MouseMove(x: float, y: float, mb: int):
6      if mb == 1:
7          FillCircle(x, y, 10, color)
8
9
10 OnMouseMove = MouseMove
```

В листинге 5.2 представлен пример обратной ситуации: использование модуля, написанного на SPython (см. листинг 2.8), в программе на PascalABC.NET.

Листинг 5.2 — Пример использования модуля, написанного на SPython, в программе на PascalABC.NET

```
1 Uses SPythonModule;
2
3 begin
4     Println(SPythonModule.pi);                // 3.14
5     Println(SPythonModule.fibonacci(10));     // 55
6 end.
```

Таким образом, благодаря унифицированному формату .pcu-файлов, программы на языке SPython могут использовать широкую библиотеку модулей, написанных на PascalABC.NET. В свою очередь, SPython также может быть эффективно применён для написания собственных небольших модулей, требующих меньшего объёма кода по сравнению с PascalABC.NET. Обе стороны взаимодействия полностью совместимы между собой, что позволяет использовать сильные стороны каждого из языков в рамках одной программной системы.

6 Сравнение времени выполнения программ на языках SPython, Python и PascalABC.NET

При разработке компилятора особое внимание уделялось эффективности исполняемых программ. Одной из целей было обеспечить, чтобы скорость выполнения программ на языке SPython не уступала скорости эквивалентных программ, написанных на PascalABC.NET.

Для достижения этого синтаксический сахар SPython преобразовывался в эффективные, эквивалентные конструкции языка PascalABC.NET (см. разделы 3.4, 3.5, 3.6, 3.7, 3.11 и 3.12), а также минимизировалось использование глобальных переменных (см. раздел 3.13).

Настоящая глава содержит результаты замеров времени выполнения эквивалентных программ, реализованных на языках SPython, Python и PascalABC.NET. Для объективной оценки производительности рассмотрены два тестовых примера, различающихся по структуре и характеру вычислений.

6.1 Аппаратное и программное обеспечение

Для проведения замеров производительности использовался персональный компьютер со следующими характеристиками:

— Операционная система — Microsoft Windows 10 Pro, версия 10.0.19045;

— Процессор — Intel Core i7-8700K, 6 ядер, 12 потоков, частота 3,7 ГГц;

— Оперативная память — 32 ГБ установленной памяти;

— Тип системы — 64-битная архитектура (x64).

Используемое программное обеспечение и следующие версии:

— Компилятор SPython — внутренняя сборка от 8 июня 2025 года;

— Интерпретатор Python — версия 3.10.11;

— Компилятор PascalABC.NET — версия 3.10.3.

6.2 Измерение времени выполнения при сортировке случайного списка

В качестве первого теста была выбрана задача сортировки списка вещественных чисел с использованием встроенного метода сортировки, предоставляемого языком программирования.

Тексты эквивалентных программ, реализующих данную задачу, представлены в листингах 6.1 и 6.2 для языков SPython (Python) и PascalABC.NET соответственно.

Листинг 6.1 — Сортировка случайного списка на языке SPython (Python)

```
from time import time
from random import random as rand

n = int(input())
a = [rand() for _ in range(0, n)]

start = time()
a.sort()
end = time()

print(end - start)
```

Листинг 6.2 — Сортировка случайного списка на языке PascalABC.NET

```
begin
  var n := ReadInteger;
  var a := Range(0, n - 1).Select(i -> random()).ToList();

  var start := Milliseconds;
  a.sort();
  var &end := Milliseconds;
  Println(&end - start);
end.
```

Замеры времени выполнения проводились на списках трёх различных размеров: 10^6, 10^7 и 10^8 элементов. Для каждой реализации проводилось по пять запусков, после чего вычислялось среднее время выполнения. Результаты округлены до сотых долей секунды и представлены в таблице 6.1.

Таблица 6.1 — Среднее время сортировки списка случайных вещественных чисел (в секундах)

Размер списка	**SPython**	**PascalABC.NET**	**Python**
10^6	0,07	0,07	0,18
10^7	0,85	0,83	2,72
10^8	9,34	9,35	39,21

Проведённые замеры показывают, что программы на языке SPython демонстрируют сопоставимую производительность с PascalABC.NET при использовании стандартных средств языка. При этом интерпретатор Python продемонстрировал заметно худшие результаты по времени выполнения, особенно на больших объёмах данных, что подчёркивает преимущество статической компиляции.

6.3 Измерение времени выполнения при вычислении двойной суммы

В качестве второго примера для оценки производительности выбрано вычисление двойной суммы с использованием вещественных чисел. Данная задача представляет собой типичную численную нагрузку с вложенными циклами и большим числом операций с плавающей точкой. Реализация выполнена с использованием базовых средств языков без привлечения сторонних библиотек.

Тексты программ представлены в листингах 6.3 и 6.4.

Листинг 6.3 — Вычисление двойной суммы на языке SPython (Python)

```
1  from time import time
2
3  n = int(input())
4
5  start = time()
6  sm = 0.0
7  i = 1.0
8  while i < n:
9    j = 1.0
10   while j < n:
11     sm += 1.0 / i / j
12     j += 1
```

```
    i += 1
end = time()

print(end - start)
```

Листинг 6.4 — Вычисление двойной суммы на языке PascalABC.NET

```
##
var n := ReadInteger;

var start := Milliseconds;
var sm := 0.0;
var i := 1.0;
while i < n do
 begin
    var j := 1.0;
    while j < n do
    begin
        sm += 1.0 / i / j;
        j += 1;
    end;
    i += 1;
end;
var &end := Milliseconds;

print(&end - start);
```

Результаты замеров времени выполнения (в секундах) для различных значений переменной n усреднены по пяти запускам и приведены в таблице 6.2.

Таблица 6.2 — Время выполнения задачи двойной суммы

Величина n	**SPython**	**PascalABC.NET**	**Python**
$2 \cdot 10^4$	0,71	0,72	59,02
$4 \cdot 10^4$	2,88	2,88	271,53
$6 \cdot 10^4$	6,69	6,67	724,23

Результаты показывают, что программы на языках SPython и PascalABC.NET демонстрируют сопоставимую производительность. При этом эквивалентная программа на языке Python, имеющая синтаксически аналогичный код, выполняется значительно медленнее. Существен-

ное отставание Python особенно заметно при увеличении объёма вычислений.

Использование генеративного ИИ при написании работы

В процессе подготовки текста данной работы применялась языковая модель GPT. Модель использовалась для:

— проверки грамматической и орфографической корректности текста;

— редактирования формулировок с целью повышения ясности и читаемости;

— стилистического выравнивания текста в соответствии с академическими требованиями.

Заключение

В результате дипломной работы была реализована фронтенд-часть компилятора языка SPython в архитектуре PascalABC.NET, включающая генерацию кода для сложных синтаксических конструкций, первичное разрешение имён и оптимизацию глобальных переменных.

В языке реализовано большинство конструкций Python, используемых при изучении программирования, а также поддерживаются наиболее востребованные стандартные функции и библиотеки.

Компилятор генерирует эффективный код и обеспечивает пользователя понятными сообщениями об ошибках с корректным позиционированием.

Компилятор поддерживает механизм взаимодействия между языками SPython и PascalABC.NET через программные модули, что обеспечивает двухстороннюю совместимость.

Полученный язык может быть рекомендован для использования в образовательных целях при обучении программированию.

Реализация доступна в открытом репозитории по адресу:
`https://github.com/AlexanderZemlyak/pascalabcnet/tree/SPython`

Список использованных источников

1. *Михалкович, С. С.* — PascalABC.NET — современный язык программирования. — Южный федеральный университет, 2025. — Язык Pascal с расширениями и поддержкой платформы .NET. `https://pascalabc.net`.

2. *Гафф, К. Джон.* GPLEX: генератор лексических анализаторов для C#. — `https://github.com/k-john-gough/gplex`. — 2020.

3. *Мовчан, Е. В.* Реализация парсера языка SPython в архитектуре PascalABC.NET. — `https://hub.sfedu.ru/repository/material/801327971/`. — 2024.

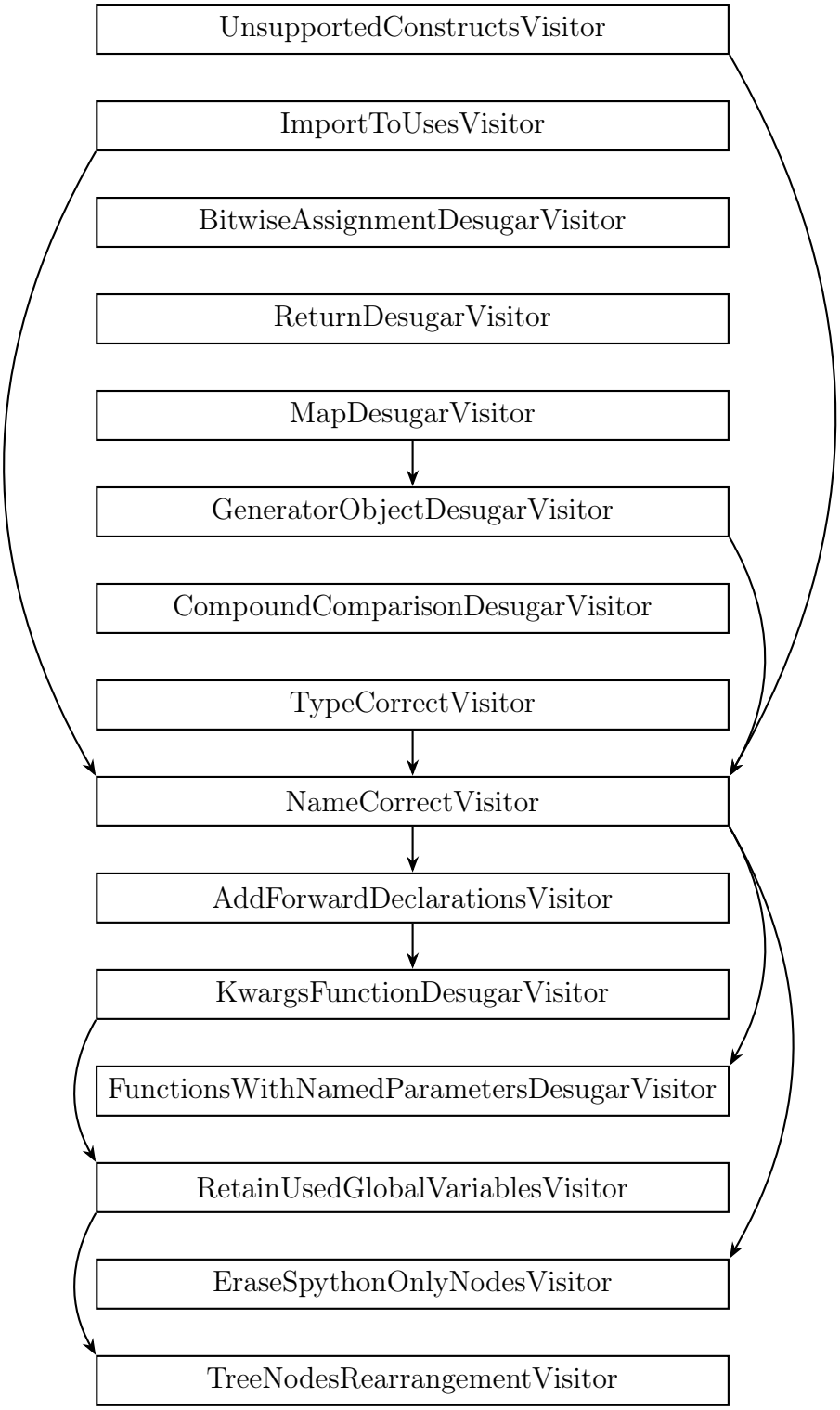
4. *Гафф, К. Джон.* GPPG: генератор GLR-парсеров для C#. — `https://github.com/k-john-gough/gppg`. — 2020.

5. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. — Санкт-Петербург: Питер, 2022. — Перевод книги «Design Patterns. Elements of Reusable Object-Oriented Software».

6. Компиляторы: принципы, технологии и инструментарий / Альфред В. Ахо, Моника Лам, Рави Сети, Джеффри Д. Ульман. — Москва: Вильямс, 2008.

7. *Крылов, В. В.* Реализация семантики языка SPython в архитектуре PascalABC.NET. — `https://hub.sfedu.ru/repository/material/801327966/`. — 2024.

Приложение А Граф зависимостей визиторов



Приложение Б Пример игры на SPython

```
from PABCSystem import Random, Inc, Milliseconds
from WPFObjects import *

CountSquares = 10
CurrentDigit = 1
Mistakes = 0
StatusRect: RectangleWPF

def DrawStatusText():
  if CurrentDigit <= CountSquares:
    StatusRect.Text = \
      f'Removed squares: {CurrentDigit-1}\tMistakes: {Mistakes}'
  else:
    StatusRect.Text = \
 f'Game over. Total time: {Milliseconds // 1000} s.\tMistakes: {Mistakes}'

def MyMouseDown(x: float, y: float, mb: int):
    ob = ObjectUnderPoint(x, y)
    if ob is RectangleWPF and ob != StatusRect:
        if ob.Number == CurrentDigit:
            ob.Destroy()
            Inc(CurrentDigit)
            DrawStatusText()
        else:
            ob.Color = Colors.Red
            Inc(Mistakes)
            DrawStatusText()

Window.Title = 'Game: destroy all squares in order'
for i in range(1, CountSquares + 1):
    x = Random(Window.Width - 50)
    y = Random(Window.Height - 100)
    ob = new RectangleWPF(x, y, 50, 50, Colors.LightGreen, 1)
    ob.FontSize = 25
    ob.Number = i
StatusRect = new RectangleWPF(0, Window.Height - 40,
                              Window.Width, 40, Colors.LightBlue)
DrawStatusText()
OnMouseDown = MyMouseDown
```