Содержание

1 Ввод выражений

1.1 Общие замечания

Система `SYMBALG` позволяет смешивать инструкции языка `Python`, которые будут выполняться непосредственно при запуске программы кодогенерации, и различные выражения (как правило алгебраические), которые будут преобразованы в AST, при необходимости проанализированы и сконвертированы в соответствующий выходной формат. Выражения, преобразуемые в AST, вводятся в обычном синтаксисе языка `Python`, но их аргументами должны быть экземпляры класса `Var` либо функции из словаря `spaceOp` (модуль `expressions.py`).

Если в выражении хотя бы один из аргументов является экземпляром класса `Var`, результатом выполнения выражения будет AST. Вы можете специально создать необходимые экземпляры класса `Var` в текущем пространстве имен, но как правило для создания AST используется стандартная функция `eval`, вызываемая в специальном контексте. Предполагается, что глобальным пространством имен будет словарь `spaceOp`, содержащий математические функции, функции скобок, и специальный экземпляр класса `EmptyVar` с индентификатором `_`, а локальным пространством имен будет специальный объект пользователя, эмулирующий словарь и создающий экземпляры класса `Var` по правилам пользователя, зависящим от специфики решаемой задачи.

Все классы, из которых собирается AST, являются наследниками класса `BaseOp`. Для всех объектов перегружены операции, возвращающие экземпляры соотвествующих классов. Скобки в выражении, используемые для задания приоритета выполнения операций, учитываются при создании AST, но при конвертации выражения в любой из форматов скобки расставляются лишь при необходимости, согласно приоритету операций. Перегруженные операции, их приоритет и названия соотвествующих классов, экземпляры которых являются узлами AST, приведены в таблице 1.

При задании выражений категорически **не рекомендуется** явно использовать имена классов реализующих узлы AST, за исключением специально оговоренных случаев (например класса `Not` или классов, реализующих скобки), поскольку имена классов могут не присутство-

операция	метод BaseOp	класс AST	приоритет	коммутативность
a^b	`__pow__`	`PowOp`	1	нет
$-a$	`__neg__`	`NegOp`	2	—
$+a$	`__pos__`	`PosOp`	2	—
$\tilde{}a$	`__inv__`	`InvOp`	2	—
$a * b$	`__mul__`	`MulOp`	3	да
a/b	`__div__`	`DivOp`	3	нет
$a//b$	`__floordiv__`	`FloordivOp`	3	нет
$a\%b$	`__mod__`	`ModOp`	3	нет
$a + b$	`__add__`	`AddOp`	4	да
$a - b$	`__sub__`	`SubOp`	4	нет
$a<<b$	`__lshift__`	`LshiftOp`	5	нет
$a>>b$	`__rshift__`	`RshiftOp`	5	нет
$a\&b$	`__and__`	`AndOp`	6	да
$a\|b$	`__or__`	`OrOp`	7	да
$a\hat{}b$	`__xor__`	`XorOp`	8	да
$a = b$	`__eq__`	`EqOp`	9	да
$a \neq b$	`__ne__`	`NeOp`	9	да
$a < b$	`__lt__`	`EqOp`	9	нет
$a \leq b$	`__le__`	`LeOp`	9	нет
$a > b$	`__gt__`	`GtOp`	9	нет
$a \geq b$	`__ge__`	`GeOp`	9	нет

Таблица 1: Таблица перегруженных операций

вать в контексте выполнения кода. Имена классов–узлов AST используются для настройки преобразования AST в различные форматы вывода и для анализа введенных выражений.

При задании выражений может использоваться специальный экземпляр класса `EmptyVar` с индентификатором `_`, содержащийся в словаре `spaceOp`. Фактически, при помощи этого объекта расширяется синтакис `Python`. Вы можете вводить свои операции, как комбинации уже существующих бинарных операций, разделенных символом `_` (например `a/_/b`). Экземпляр класса `EmptyVar` не может быть приведен к строке и не может присутствовать в итоговом выражении в формате AST.

При задании участков кода для `C++` некоторые конструкции не могут быть заданы из `Python` непосредственно. Например, код вида `p->a` или `&(A::x)` может быть сформирован лишь в виде строки, что исключает возможность анализа кода. Для подобных ситуаций специально введен класс `JoinOp`, позволяющий вводить в перемешку участки кода в виде AST и в виде строк. При ковертации, все участки сливаются без разделяющих пробелов. Например, код

```
aiv::indx<3>(a->x)
```

может быть сформирован как

скобки	(...)	[...]	{...}	|...|	<...>
класс AST	crc_bk	rm_bk	fig_bk	fabs	ang_bk
вызов через объект _	_(...)	_(..., [])	_(..., {})	_(..., "||")	_(..., "<>")
расширенный синтаксис	(%...%)	[%...%]	{%...%}	|%...%|	<%...%>

Таблица 2: Виды скобок. Последняя строка может исполььзоваться лишь при обработке функцией `expressions.parse_bk`

```
JoinOp('aiv::indx<3>(', a, '->', x, ')')
```

Непосредственное обращение к классу `JoinOp` выглядит довольно громоздко, кроме того этот класс не пристутсвует в словаре `spaceOp`. Во всех случаях рекомендуется более лаконичная запись

```
_['aiv::indx<3>(', a, '->', x, ')']
```

которая к тому же может использоваться при присваивании (слева от знака равенства) при задании участков графа зависмости численных схем.

1.2 Математические функции

В словаре `spaceOp` представлены математические функции, присутствующие в модуле `math.py` и заголовочном файле `math.h`:

```
acos acosh asin asinh atan atanh cos cosh
exp fabs floor log log10 sin sinh sqrt tan tanh
```

Математические функции являются наследниками класса `UnaryOp` с приоритетом 0. Математические функции корректно конвертируются в формат LaTeX, в частности для них корректно проставляются степени, например код `cos(x)**2` при выовде в LaTeXбудет преобразован в $\cos^2 x$.

1.3 Явное задание скобок

Для явного задания скобок могут использоваться специальные классы скобок из словаря `spaceOp`, хотя в ряде случаев более лаконичным (наглядным) может оказаться вызов метода `_.__call__`, см. таблицу 2. Все скобки преобразуются к обычным (круглым) скобкам при конвертации в `C++`/`Python`, и в соотвествующие скобки при конвертации в обычный текст или LaTeX.

1.4 Условия и логические операции and, or и not

Поскольку разбор выражения в AST реализован на основе перегрузки операций, в выражении не могут непосредственно применяться логические операции `and`, `or`, `not` и тернарный оператор `... if ... else ...`, поскольку их перегрузка невозможна. Кроме того, эти операции

актуальны в неперегруженном виде при задании выражений, например в зависимости от значений флага может выбираться тот или иной вид численной схемы еще на этапе генерации кода.

Для задания унарного оператора `not` используется класс `Not` из словаря `spaceOp` (например `Not(a<b)`), корректно конвертирующийся в форматы языков `Python` (как `not ...`) и `C++` (как `!...`).

Для задания операторов `and` и `or` используются методы `.And(...)` и `.Or(...)` базового класса `BaseOp` или операторы `&_&` и `|_|`. При их вызове возвращаются экземпляры классов `BoolAndOp` и `BoolOrOp`, реализующие узлы AST для соотвествующих операций. Поскольку приоритеты операций `|` и `&` выше приоритета операций сравнения, при использовании записи `&_&` и `|_|` желательно использовать скобки, например для задания выражения

```
a>b and c==d
```

код

```
(a>b).And(c==d)
```

или

```
(a>b)&_&(c==d)
```

корректен, и приводит к нужным результатам, а код

```
a>b &_& c==d
```

будет эквивалентен выражению `a>(b and c)==d`.

Для задания тернарных операторов `... if ... else ...` в выражении используется специальная функция `ifch` из словаря `spaceOp`. Функция может принимать три аргумента, в этом случае возвращается экземпляр класса `IfElseOp` (как узел AST), либо два аругмента, в этом случае возвращается спецальная функция, запоминающая введенные аргументы и готовая опять к приему двух или трех аргументов. Узел AST генерируется лишь при передаче трех аргументов. Например выражение

```
a if a<b else b if b<c else c if c>d else d
```

может быть задано как

```
ifch(a, a<b)(b, b<c)(c, c>d, d)
```

1.5 Задание индексов, операций вызова и доступа к атрибуту

Для использования в выражениях конструкиций типа `a[1,2]`, `b(d,1)` и `g.h` на основе бинарных операций были реализованы следующие классы:

Класс	Описание	Пример
`IndexOp(BinaryOp)`	Квадратные скобки.	`a[1,2]`
`CallOp(BinaryOp)`	Круглые скобки.	`b(d,1)`
`AttrOp(BinaryOp)`	Оператор точки	`g.h`

С помощью `IndexOp` можно указывать индексы. С помощью `CallOp` можно имитировать операцию вызова с аргументами или без. С помощью `AttrOp` можно имитировать операцию доступа к атрибуту.

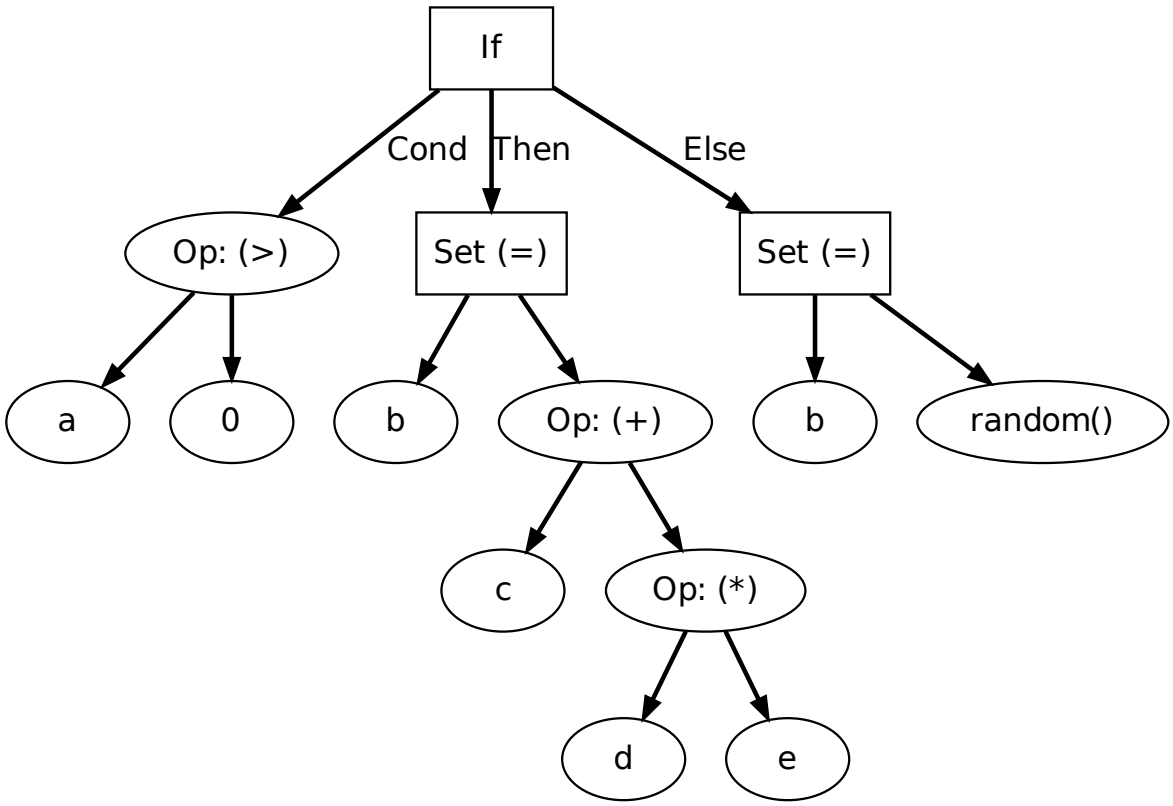

Рис. 1: Пример AST

2 Операторы(Statements)

2.1 AST, операторы, трасса выполнения программы

Система Symbalg является инструментом для кодогенерации. Как правило, код программыё на каком-либо языке программирования (в нашем случае используется язык C++) состоит из модулей и написанного внутри них текста программы.

Программный код на языке C++ представим в виде AST (Abstract Syntax Tree — Абстрактное синтаксическое дерево). AST программного кода — дерево, узлами которого являются операторы языка программирования или операции, а листьями — операнды (рис.1). Реализация операций и операндов уже описана в прошлой главе (1). В этой главе рассмотрим реализацию операторов языка программирования C++.

Оператор в императивных языках программирования (коим является C++) — это наименьшая автономная часть языка программирования; команда. В таблице 3 приведены примеры различных операторов на языке C++.

Оператор	Пример
Объявление	`int a = 0;`
Присваивание	`a = b + c;`
Последовательность операторов	`int a;` `a = b + c;`

Блок операторов	`{`
	`  int a;`
	`  a = b + c;"`
	`}`

Условный оператор	`if (a != b){`
	`  d = c + 1;`
	`} else {`
	`  d = c + 2;`
	`}`

Оператор переключения	`switch (a){`
	`  case 0:`
	`  d = c + 1;`
	`  break;`
	`  case 1:`
	`  d = c + 2;`
	`  break;`
	`}`

Цикл For	`for(int i = 0; i < N; ++i){`
	`  d+=i;`
	`}`

Цикл с предусловием	`while (a<10){`
	`  a++;`
	`}`

Цикл с постусловием	`do {`
	`  a++;`
	`while (a<10);`

Вызов подпрограммы	`IncreaseTemp(10);`
Возврат из подпрограммы	`return 0;`

Таблица 3: Различные виды операторов языка C++

Хоть AST — дерево, набор операторов также представим в виде последовательно идущего программного кода — трассы выполнения программы. Это наиболее привычный вид, который используется при написании программы человеком — последовательно строчку за строчкой добавлять операторы в трассу выполнения программы. Этот принцип взят за основу кодогенерации в системе Symbalg.

Система Symbalg реализована на языке программирования Python, и используется в рамках синтаксиса языка Python. Соответственно все использование системы должно вестись в рамках синтаксиса языка Python, не используя дополнительные файлы с псевдокодом, и ограничивая использование строковых конструкций. Python — объектно-ориентированный язык программирования, и все сущности представляют собой объекты — экземпляры классов. Операторы реализованы как классы языка Python, и их экземпляры можно создавать при помощи явного использования конструкторов. Этот метод создания AST, несомненно, является громоздким и нежелательным. Для создания AST путем наполнения трассы выполнения программы были сформулированы следующие постулаты, на которых основана идеология системы кодогенерации:

1. Хранение AST в виде дерева, узлами и листьями которого являются экземпляры классов системы Symbalg.

2. Возможность создания дерева с нуля, наполняя трассу выполнения программы в рамках синтаксиса языка Python.

3. Возможность гибко работать с уже созданным фрагментом AST, как с традиционным деревом (создание нового дерева на основе существующего(их), добавление ветвей, замена ветвей, удаление ветвей, рекурсивный перебор всех элементов дерева и т.д.).

Опишем реализацию сформулированных постулатов по пунктам.

2.2 Иереахия классов, реализующих операторы

Базовым классом для операторов является `BaseStm` и основная логика операторов описана в модуле `statement.py`. На рисунке 2 представлены классы, которые реализуют поведение операторов.

Каждый наследник класса `BaseStm` реализует один из операторов, приведенных в таблице 3. В таблице 4 представлено сопоставление операторов и классов, их реализующих.

Эта иерерахия и лежит за реализацией первого пункта постулатов из трех.

2.3 Заполнение AST путем наполнения трассы выполнения программы

Второй постулат заключается в том, чтобы существовала возможность создания AST путем набирания трассы выполнения кода программы на языке C++ в рамках синтаксиса языка Python. Первое, на чем нужно условиться, это то, что участок AST представляется в системе

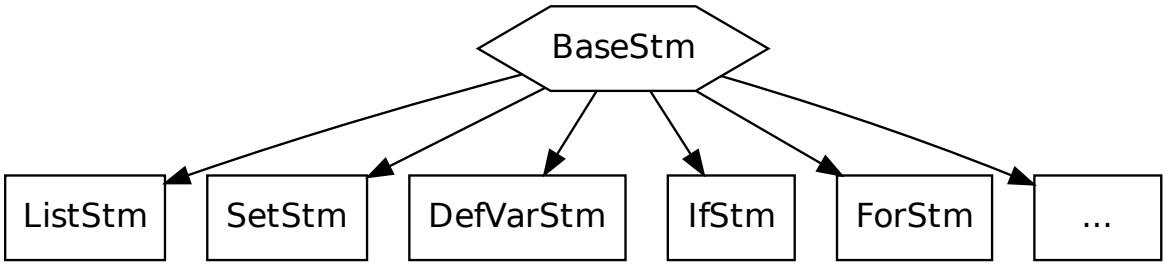

Рис. 2: Иерархия классов, реализующих операторы

Оператор	Класс, реализующий оператор	Описание класса
Объявление	`DefVarStm`	Имеет поля `name`, `type` и `value`
Присваивание	`SetStm`	Имеет поля `lvalue`, `op` и `expr`
Последовательность операторов	`ListStm`	Имеет поле `_list` — список операторов
Блок операторов	`Пока не реализован`	
Условный оператор	`IfStm`	Имеет поля `cond`, `mode`, `_then` и `_else` Может работать в двух режимах: 1) Только ветвь True (`mode==True`), 2) Ветвь True и одна ветвь Else (`mode==False`),
Оператор переключения	`Пока не реализован`	
Цикл For	`ForStm`	Имеет поля `init`, `cond`, `step` и `code`
Цикл с предусловием	`Пока не реализован`	
Цикл с постусловием	`Пока не реализован`	
Вызов подпрограммы	`Пока не реализован`	
Возврат из подпрограммы	`Пока не реализован`	

Таблица 4: Различные виды операторов языка C++

Symbalg, как экземпляр класса `ListStm`. Таким образом, в системе Symbalg, наполнение трассы выполнения программы заключается в формировании списка операторов, который хранится в экземпляре класса `ListStm`.

Для этого функционала нам нужно по особому обрабатывать код на языке Python, который будет составлять желаемый участок AST. Такая обработка кода в системе Symbalg достигается путем исполнения участка кода на языке Python в искусственно созданном пространстве имен.

В этом пространстве имен нам хотелось бы легко вводить участок AST, то есть:

- Использовать переменные внутри выражений без их изначального инициализирования
- Позволить вводить некоторые специфичные для языка C++ конструкции (циклы For, i++ и т.д) в синтаксисе языка Python
- Просто вводить AST как программный код без самостоятельного создания каких-либо объектов.

Для этого предлагается реализовать класс, эмулирующий пространство имен — `NamespaceStm`. Чтобы выполнить некий фрагмент кода в нужном пространстве имен, нужно этот фрагмент кода в виде строки передать в процедуру `exec`, а также в качестве локального пространства имен передать нужное нам пространство имен. Для того, чтобы это можно было сделать, для этого класса, реализующего пространство имен нужно создать методы `__getitem__` и `__setitem__`, которые используется в традиционных пространств имен (обычные словари) для возврата элемента по имени. Тем самым, вместо возврата элемента словаря в методе `__getitem__` мы можем обрабатывать пришедшую на вход строку как угодно и совершать любые действия. Метод `__setitem__` вызывается при конструкциях типа:

```
a = ...  # вызовет .__setitem__("a")
```

Будем различать несколько типов переменных во фрагменте кода, представляющего участок AST:

- переменная `_` — при встрече с такой переменной создастся экземпляр класса `EmptyVar`.
- переменная, начинающаяся со знака подчеркивания — переменная, которая не войдет в AST и используется как обычная переменная.
- переменная, не начинающаяся со знака подчеркивания — для нее создастся(если уже не был создан) экземпляр класса `var`.
- зарезервированная Symbalg переменная для написания некоторых операторов (например — `For`, `If`, ...). При их встрече в коде, вызовется создания экземпляра класса соответствующего оператора.

Для того, чтобы реализовать описанный функционал, был реализован декоратор `@statement`. При декорации им метода, все, что является телом этого метода передается в виде строки в процедуру `exec` с локальным пространством имен `NamespaceStm`.

Пример создания участка AST при помощи метода, декорируемого декоратором `@statement`:

```
@statement
def test():
    a[0,1,2] = w//b[2,3]
    For[i:0,N]
    a[1,3,4] = g[1,2,3]
```

При вызове полученной функции возвращается экземпляр класса `ListStm`, с именем `test`. При вызове этого метода в качестве именованных аргументов можно передавать пары "переменная": "операция". При этом во всем участке AST совершится подстановка предложенной операции вместо предложенной переменной.

В таблице 5 приведены примеры использования операторов языка C++ внутри функций, декориуемых декоратором `@statement`.

2.4 Работа с AST как с деревом

С участком AST(экземпляром класса `ListStm`) можно совершать различные действия. Для класса `ListStm` определено сложение, которое интерпретируется как послеследование одного участка AST за другим. Уже существующий экземпляр класса `ListStm` можно использовать как внутри пространства имен `NamespaceStm`, так и вне его. Использовать внутри `NamespaceStm` можно при помощи следующей конструции:

```
_[StmName()]
```

При обращении к оператору, в круглых скобках можно указывать аргументы. При их помощи можно совершать изменения в AST, которое интерпретируется этим оператором. Есть два различных вида аргументов:

- именованные аргументы, передаваемые при вызове экземпляра класса `ListStm`, используемые для макроподстановок. Например конструкция:

  ```
  StmName(a=b)
  ```

  заменит все переменные `a` внутри AST на переменную `b`

- именованный аргумент `_conv=func` — позволяет рекурсивно применять метод `func` ко всем узлам AST. Метод может как собирать информацию с AST, так и изменять его части. Пример рекурсивной функции и ее применения:

  ```
  def offset_1(x):
      if isinstance(x, IndexOp) and type(x.b) is tuple:
          x.b = tuple([i+1 for i in x.b])
      return x

  print StmName()(_conv=offset_1)
  ```

  Эта функция прибавляет единицу ко всем, встречающимся в AST, индексам (содержимым квадратных скобок)

Оператор	Пример
Объявление	`В явном виде не реализован`

Присваивание	`a = 1`
	`----------`
	`b += 1`

Последовательность операторов	`a = 1` `b += 1`
Блок операторов	`Пока не реализован`

Условный оператор	`If[a>b]`
	`c+=1`
	`Else`
	`c+=2`
	`End`

Оператор переключения	`Пока не реализован`

Цикл For	`For[counter:start, stop]`
	`B += counter`
	`End`
	`----------`
	`For[counter:start, stop, step]`
	`B /= counter`
	`End`
	`----------`
	`For[counter, start, condition, nextstep]`
	`B += counter`
	`End`
	`----------`
	`For[type, counter, start, condition, nextstep]`
	`B += counter`
	`End`

Цикл с предусловием	`Пока не реализован`
Цикл с постусловием	`Пока не реализован`
Вызов подпрограммы	`Пока не реализован`
Возврат из подпрограммы	`Пока не реализован`

Таблица 5: Пример реализации операторов языка C++