Прототипы построителей промежуточных представлений исходных текстов программ, основанные на компиляторах с открытым исходным кодом

А.Н.Пустыгин, Б.А.Тарелкин, А.А.Ковалевский, Е.А.Огуречникова, А.В. Десинов, Н.А. Ошнуров, Е.В.Старцев, М.В. Зубов

Способы анализа качества ПО

- Динамический - проверка кода в процессе его исполнения
  - Отладчики для наблюдения за состоянием программной системы после её запуска
  - Профайлеры для анализа производительности и потребления памяти под нагрузкой
  - Фазеры для тестирования для всех возможных сочетаний входных данных
- Статический анализ - проверка исходных текстов ПО без его компиляции и исполнения

Достоинства статического анализа

- Не требует подготовленных или генерируемых входных данных, специально созданного окружения
- Допускает кросс-системный анализ программ, независимо от целевой программной и аппаратной платформы
- Не зависит от частоты появления событий, влияющих на функционирование ПО
- Не требует затрат ресурсов целевой платформы(программно-аппаратных стендов)

Недостатки статического анализа

- Требует обработки больших массивов символьной информации
- С увеличением объема исходных текстов время анализа может расти быстрее,чем растет объем
- Неизбежно использование упрощенных моделей, что ухудшает качество анализа
- Не все ошибки могут быть обнаружены, одновременно происходят “ложные срабатывания”

Направления использование статического анализа

- для поиска ошибок
- для верификации программного обеспечения
- для проверки соответствия исходного текста стандартам разработки
  - MISRA C/C++. стандарт разработки приложений для встроенных систем. поддерживается рядом статических анализаторов (в основном проприетарных).
  - JSF++. Для C++,стандарт безопасности объектно-ориентированных приложений.
  - ISO 26262 - стандарт функциональной безопасности для автомобильных электрических/электронных систем. (совместим с IEC 61508)

Существующие методы статического анализа ПО

- Анализ текста вручную (неэффективен в силу больших человеческих затрат, однако, дает наиболее точный результат)
- Машинная обработка текста без формирования вспомогательных данных (имеет очень ограниченное применение)
- Обработка с использованием промежуточных представлений (машинное эквивалентное представление кода)

Выполнение статического анализа без построения промежуточного представления (текстовые методы обработки)

- Методы обработки: поиск по регулярным выражениям и поиск шаблонов типовых конструкций
- Достоинство - большая эффективность на определенном круге задач, например, при построении различных метрик
- Недостаток следует из достоинства — узкий круг задач, потому что таким методом невозможно проанализировать ни структуру кода, ни тем более семантику. Такой анализ не может дать никакого представления о поведении программы при исполнении

Использование промежуточных представлений

Промежуточное представление - набор данных, создаваемый анализатором, на основе которых выполяется анализ

- Промежуточное представление эквивалентно исходному коду по заданному критерию
- Часто промежуточные представления являются типовыми внутренними наборами данных, для которых известны эффективные алгоритмы обработки

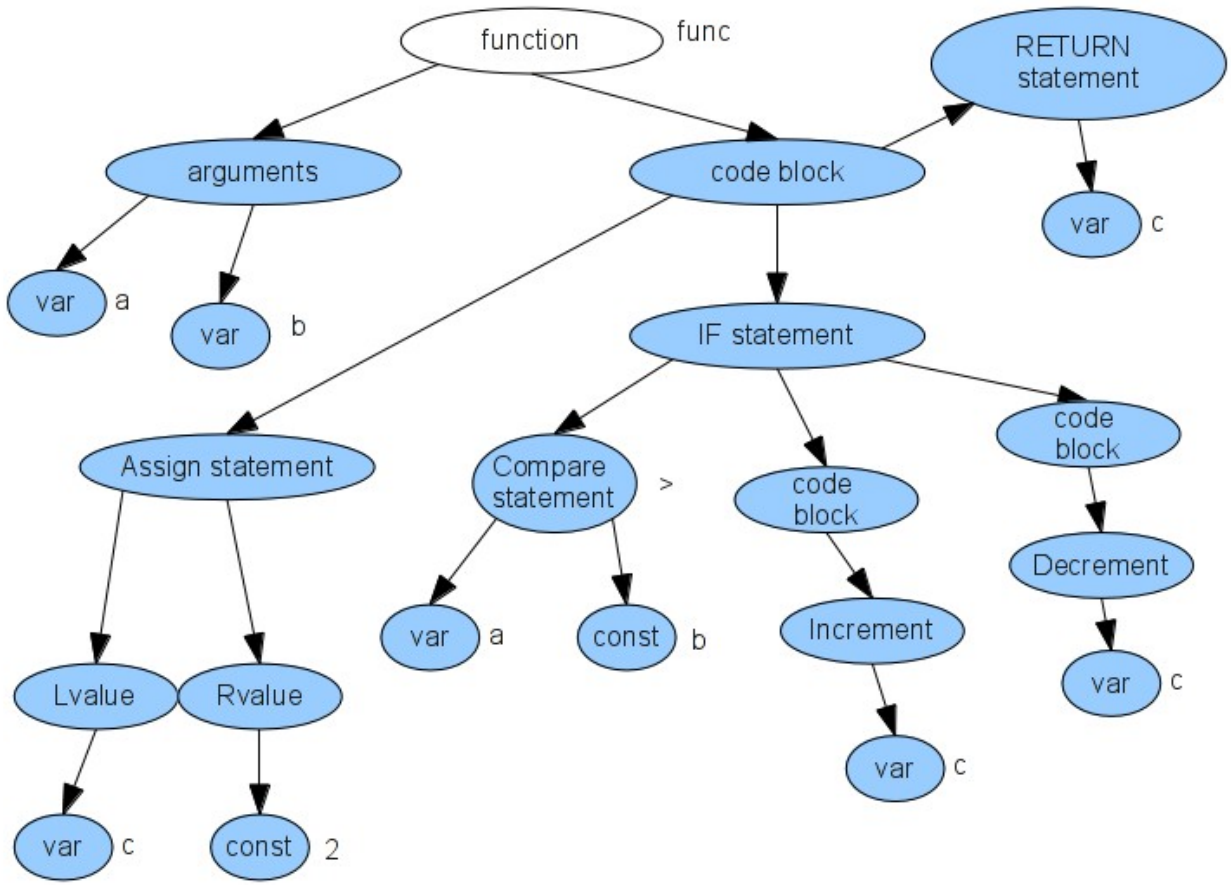
Абстрактное синтаксическое дерево AST

- Самое распространенное представление, как наиболее изученное в области компиляции.
- Компиляторы имеют под собой достаточную теоретическую и научную базу, а также постоянно развиваются. Это облегчает использование AST в статическом анализе
- При построении дерева для большого проекта целиком большой объем промежуточного представление затрудняет операции с AST

Пример исходного текста С для иллюстрации промежуточных представлений

```
    int func (int a, int b){
1:      int c=2;
2:      if (a>b) {
3:          c++;
4:      }
5:      else {
6:          c--;
7:      }
8:      return c;
    }
```

Абстрактное синтаксическое дерево для примера исходного текста С



Список существующих инструментов статического анализа, использующих AST

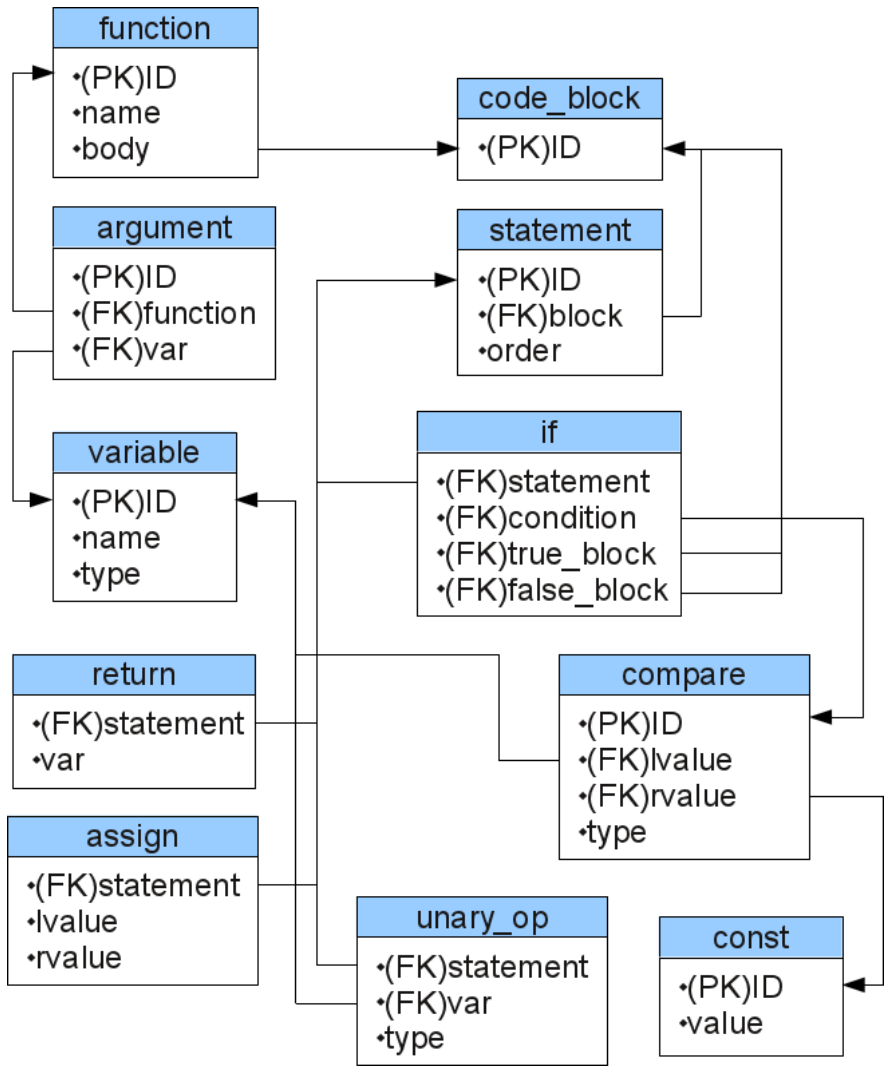
- Используется в подавляющем большинстве анализаторов:
  - Checkstyle (на основе ANTLR)
  - Compass/ROSE (на основе собственной инфраструктуры компилятора ROSE)
  - PyLint (на основе отдельного проекта logilab-astng)
  - DMS Software Reengineering Toolkit (позволяет генерировать AST для огромного количества входных языков от языков описания аппаратуры до языков программирования)

Промежуточное представление в реляционной форме

Каждый элемент грамматики языка исходного текста представляется в виде реляционного отношения. Отношения отражаются в реляционной базе данных. За счет возможности создания связей по ключам и ограничений реализуются взаимосвязи элементов грамматики.

- Реляционное представление вовсе не формируется напрямую из кода. Сначала по коду получается AST, а уже потом по нему заполняется БД.
- Основной современный проект - SemmleCode, разработка при поддержке Оксфордского университета

Реляционное промежуточное представления для примера исходного текста C - структура БД



Реляционное промежуточное представления для примера исходного текста C — содержимое таблиц БД

function		
ID	name	body
1	func	1

code_block
ID
1
2
3

statement		
ID	block	order
1	1	1
2	1	2
3	2	1
4	3	1
5	1	3

argument		
ID	function	var
1	1	1
2	1	2

const	
ID	value
1	2
2	b

if			
statement	condition	true_block	false_block
2	1	2	3

return	
statement	var
5	c

unary_op		
statemen	var	type
3	3	post++
4	3	post--

variable		
ID	name	type
1	a	int
2	b	int
3	c	int

assign		
statement	lvalue	rvalue
1	3	1

compare			
ID	lvalue	rvalue	type
1	1	2	g

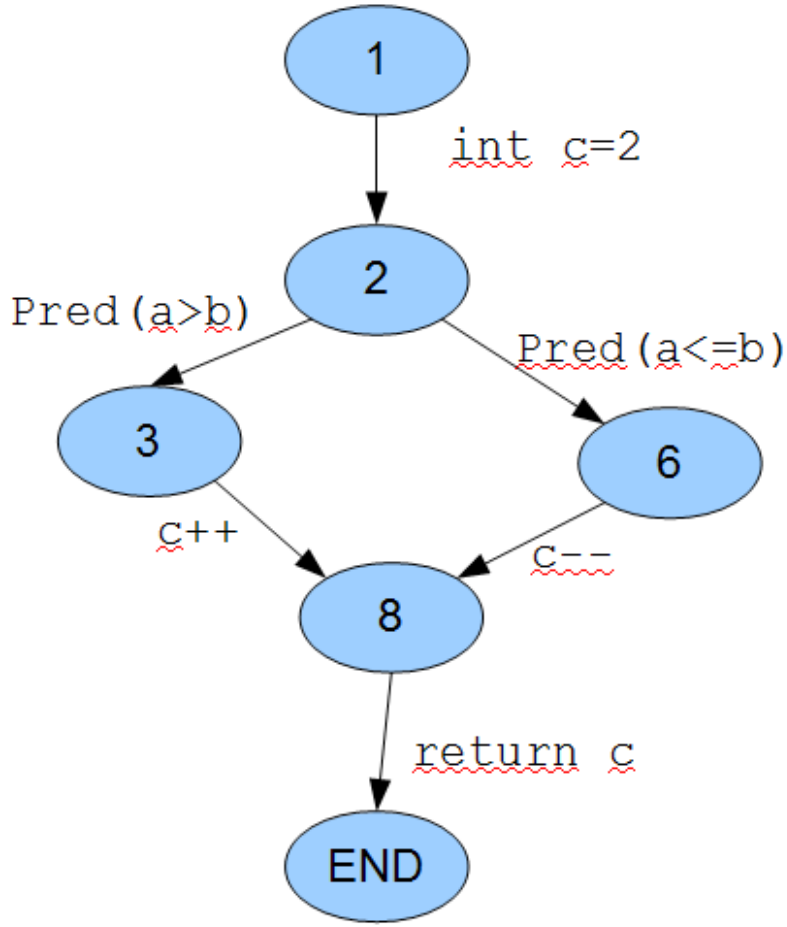
Характеристика реляционного промежуточного представления

- Большая избыточность, что свойственно всем базам данных.
- Может достигать весьма существенных объемов на больших проектах.
- Очень удобно при больших объемах, так как получение данных об узлах кода сводится к выполнению запросов.
- Для человека крайне неудобно, так как будет ручная корректировка и анализ БД неприемлем.
- Требует индивидуального проектирования БД (таблиц, индексов, ключей) для каждой грамматики,что увеличивает затраты на его получение.

Автоматное промежуточное представление

- Предназначено для анализ потока управления ПО
  Для каждой из функций программы с состояниями связываются управляющие точки программы, а с переходами - операции.
- Для верификации из автомата строятся абстрактные деревья достижимости (ART), объединяющие переходы между состояниями различных автоматов для функций (при вызове одной из другой), а также связывающие с этими состояниями логические предикаты, описывающие с определенной долей абстракции состояние переменных.
  - Пример - инструмент BLAST, который строит и использует автомат потока управления (control-flow automat).

Автоматное промежуточное представление для примера исходного текста С



Свойства автоматного промежуточного представления

- Позволяет выполнять генерацию тестовых наборов данных (test-case), обеспечивающих попадание во все возможные состояния функций (вернее их CFA).

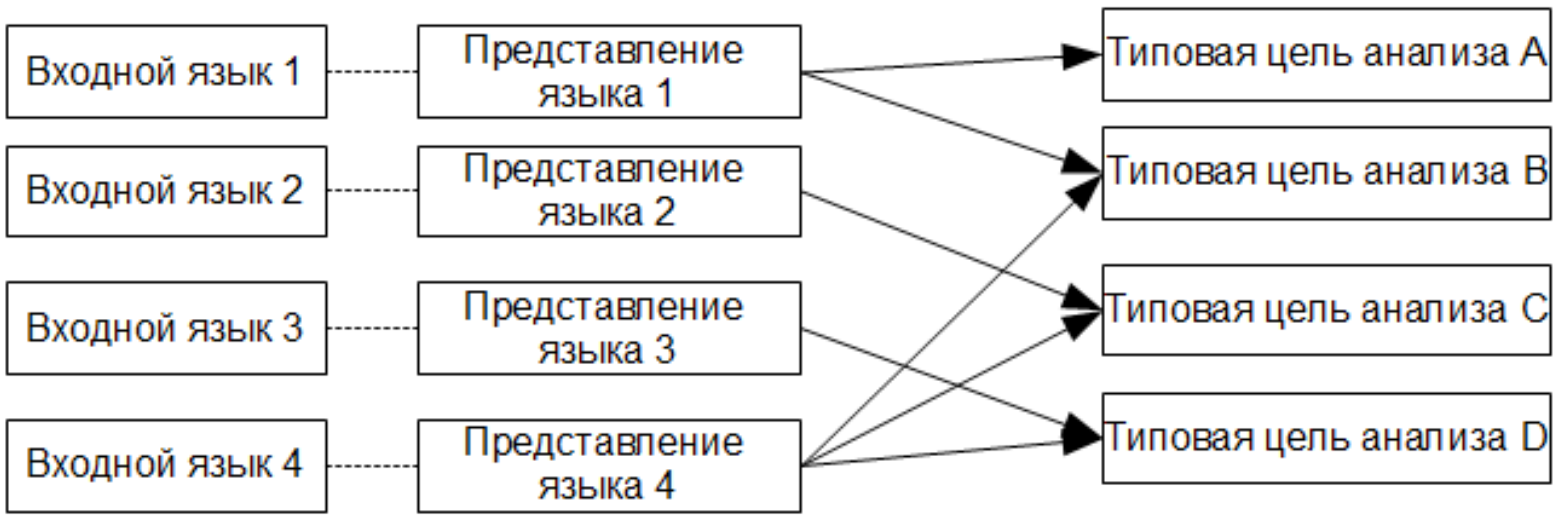
– Деревья ART строятся итерационно - counterexample-guided abstraction refinement (алгоритм уточнения абстракции по контр-примерам) (CEGAR)

– В связи с этим завершение генерации не гарантируется - это во многом зависит от конкретного случая. Зачастую, в связи с ограниченностью ресурсов.

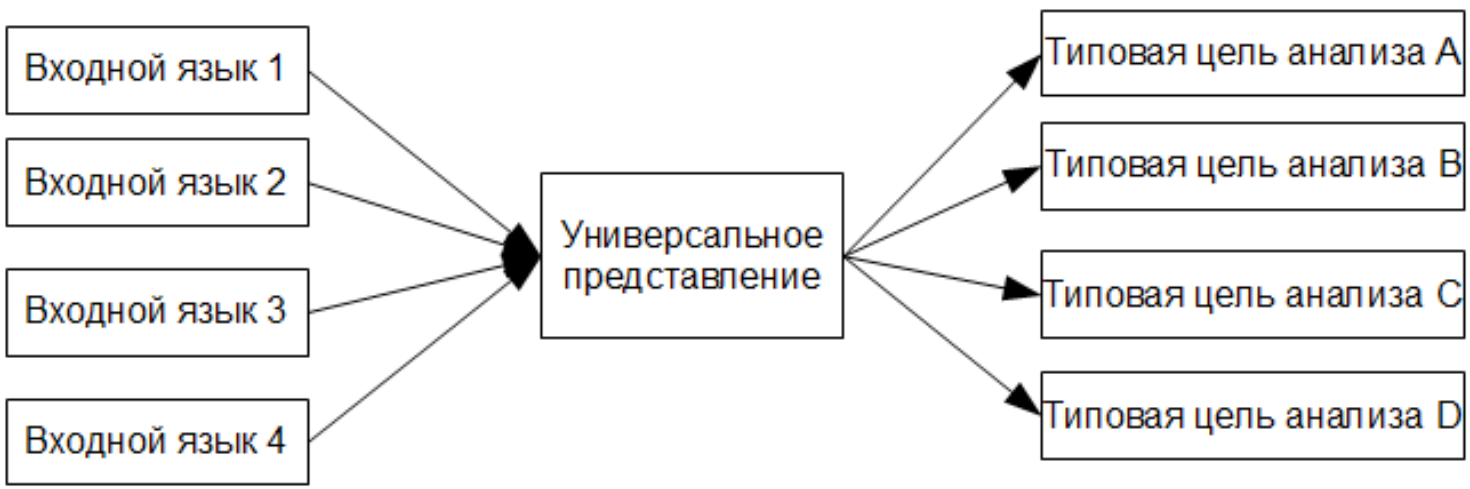
Универсальные промежуточные представления

- Универсальные промежуточные представления предназначены для построения из текстов нескольких входных языков
  - Пригодны для представления нескольких входных языков. В представлении выделяются общие для них особенности
  - Позволяют сэкономить на затратах при анализе
  - Не могут учесть особенности и специфичные конструкции для конкретных языков
- Частные представления используются для конкретных языков
  - Позволяют учесть тонкости программирования на целевом языке, которые могут быть уникальными для него
  - Ведут к большим затратам, в связи с необходимостью иметь для каждого языка свое представление

Частные промежуточные представления



Универсальные промежуточные представления



Универсальные промежуточные представления исходного текста

- Разделение фазы формирования представления (модели) и анализа этой модели сейчас слабо развито.
- Эту задачу решает универсальное представление, так как оно позволяет применить одно представление для нескольких языков.
- Благодаря этому, например, для 10 входных языков и 10 типовых целей анализа, при реализации всех 10 целей для всех 10 языков потребуется разработать 20 инструментов, а не 100 (при условие наличия единого универсального представления для всех 10 типов анализа и 10 языков)

Преимущества, обеспечиваемые многоуровневыми промежуточными представлениями

- Для разных задач анализа позволяют применять представления разной степени подробности
- На более высоких уровнях абстракции лучше использовать универсальные представления, так как имеется больше общих для целевых языков элементов
- Использование представлений различной природы на разных уровнях позволит найти компромисс между затратами на хранение и обработку
- Каждое представление хранит свой уровень абстракции, что позволит избежать избыточности. При необходимости взять данные другого уровня, для них будут использоваться другие представления.

Применение многоуровнего промежуточного представления в существующих программных инструментах

- Bauhaus project - разработка университетов Штутгарта и Бремена. Для Ada, C, C++, C#, и Java.
- Используются два представления - InterMediate Language (IML) и Resource flow graphs (RFG).

- IML является низкоуровневым представлением, содержащим информацию на синтаксическом и семантическом уровнях (описание конструкций языка).
- RFG представляет информацию о глобальных и архитектурных аспектах анализируемой системы. Это граф, включающий такие узлы как компоненты, файлы и модули.

Прототип построителя промежуточного представления для исходных текстов C/C++

Использует одноуровневое промежуточное представление в форме текстовой нотации

Рассмотрены возможные варианты получения промежуточного представления с помощью готовых инструментов.

1. Открытая библиотека VivaCore для работы с исходным текстом C и C++
2. front-end GCCXML для компилятора GCC — свободный, с открытым исходным кодом парсер для C++
3. front-end GASTA для компилятора GCC —свободный с открытым исходным кодом анализатор синтаксического дерева разбора компилятора GCC.

Недостатки вариантов получения промежуточного представления с помощью готовых инструментов

1. front-end GCCXML и *GASTA* возвращает AST только на уровне объявления операндов языка; будут отсутствовать описание содержимого тела функций, операторов ветвления, циклов и других конструкций.

2. Библиотека *VivaCore* поддерживает стандарт языка программирования *C\C++* лишь частично, в пределах текущей реализации модулей лексического и грамматического анализа библиотеки.

Внешний модуль (plug-in) для компилятора GCC

С версии *GCC* 4.5 за счет подключения динамических библиотек к компилятору во время выполнения. Интерфейс путем подписки на события, происходящие в процессе компиляции, которые определены в *GCC-plugin.h*.

```
enum plugin_event
    {
      PLUGIN_PASS_MANAGER_SETUP,    /* To hook into pass
manager.  */
      PLUGIN_FINISH_TYPE,           /* After finishing parsing a
type.  */
      PLUGIN_FINISH_UNIT,           /* Useful for summary
processing.  */
      PLUGIN_PRE_GENERICIZE,        /* Allows to see low level
AST in C and C++ frontends.  */
      PLUGIN_FINISH,                /* Called before GCC exits.
*/
      PLUGIN_INFO,                  /* Information about the
plugin. */
      PLUGIN_GGC_START,             /* Called at start of GCC
Garbage Collection. */
      PLUGIN_GGC_MARKING,           /* Extend the GGC marking.
*/
      PLUGIN_GGC_END,               /* Called at end of GGC. */
      PLUGIN_REGISTER_GGC_ROOTS,    /* Register an extra GGC
root table. */
      PLUGIN_REGISTER_GGC_CACHES,   /* Register an extra GGC
cache table. */
      PLUGIN_ATTRIBUTES,            /* Called during attribute
registration */
      PLUGIN_START_UNIT,            /* Called before processing
a translation unit.  */
      PLUGIN_PRAGMAS,               /* Called during pragma
registration. */
      /* Called before first pass from all_passes.  */
      PLUGIN_ALL_PASSES_START,
      /* Called after last pass from
all_passes.  */
      PLUGIN_ALL_PASSES_END,
      /* Called before first ipa pass.  */
      PLUGIN_ALL_IPA_PASSES_START,
      /* Called after last ipa pass.  */
      PLUGIN_ALL_IPA_PASSES_END,
      /* Allows to override pass gate
decision for current_pass.  */
      PLUGIN_OVERRIDE_GATE,
      /* Called before executing a pass.  */
      PLUGIN_PASS_EXECUTION,
      /* Called before executing subpasses of
a GIMPLE_PASS in
         execute_ipa_pass_list.  */
      PLUGIN_EARLY_GIMPLE_PASSES_START,
      /* Called after executing subpasses of
a GIMPLE_PASS in
         execute_ipa_pass_list.  */
      PLUGIN_EARLY_GIMPLE_PASSES_END,
      /* Called when a pass is first
instantiated.  */
      PLUGIN_NEW_PASS,
      PLUGIN_EVENT_FIRST_DYNAMIC    /* Dummy
event used for indexing callback
                                       array.
 */
```

Прототип функции инициализации плагина

Плагин должен экспортировать функцию инициализации и определить глобальную переменную *plugin_is_GPL_compatible*, которая показывает, что плагин был лицензирован под *GPL* – совместимой лицензией. Без данной переменной подключение плагина к GCC невозможно .

Прототип функции инициализации:

```
#include "plugin-version.h"
...
  int plugin_init (struct plugin_name_args *plugin_info,
                   struct plugin_gcc_version *version)
{
    . . .
}
```

Модули информации вызова плагина

Функция *plugin_init* вызывается сразу после загрузки плагина. Она нужна для регистрации всех обратных вызовов, требуемых плагином. Она вызывается из процедуры *compile_file* до вызова парсера. Аргументы *plugin_init* - Plugin_info – модули информации вызова

Version – версия GCC

plugin_info - структура

```
struct plugin_name_args {
    char *base_name;        /* Short name of the plugin  (filename without .so suffix). */
    const char *full_name;     /* Path to the plugin as specified with -fplugin=. */
    int argc;                  /* Number of arguments specified with -fplugin-arg-.... */
    struct plugin_argument *argv; /* Array of ARGC key-value pairs. */
    const char *version;            /* Version string provided by plugin. */
        const char *help;                   /* Help string provided by
                                plugin. */
            }
```

Компиляция и компоновка plug-in

Компиляция плагина осуществляется с помощью ключа компиляции:

```
$gcc -I`gcc -print-file-name=plugin`/include
-fPIC -shared -O2 plugin.c -o plugin.so
```

после компиляции получается динамическую библиотеку .so. В дальнейшем плагин загружается компилятором. Для подключения плагина к компилятору используется следующий ключ компиляции:

```
$gcc -fplugin=/path/to/name.so  -fplugin-arg-
name-key1[=value1]
```

Пример исходного текста C/C++ для получения промежуточного представления

```
1. bool comparison(int a, int b)
2. {
3.     bool result;
4.
5.     if (a < b)
6.         result = true;
7.     else
8.         result = false;
9.
10.     return result;
11. }
```

Промежуточное представление исходного C текста в текстовой нотации

```
function_decl ::comparison type: function_type at test.cpp:1
   decl_arguments
      parm_decl ::comparison::a type:
  integer_type at test.cpp:1
      parm_decl ::comparison::b type:
  integer_type at test.cpp:1
   function_body
      bind_expr at test.cpp:10
         statement_list
            decl_expr
               var_decl ::comparison::result
  type: boolean_type at test.cpp:3
            if_stmt at test.cpp:5
               if_cond
                  lt_expr
                     parm_decl ::comparison::a
  type: integer_type at test.cpp:1
                     parm_decl ::comparison::b
  type: integer_type at test.cpp:1
               then_clause
                  cleanup_point_expr at
  test.cpp:6
                     expr_stmt at test.cpp:6
                        convert_expr
                           modify_expr
                              var_decl
  ::comparison::result type: boolean_type at test.cpp:3
                              integer_cst
               else_clause
                  cleanup_point_expr at test.cpp:8
                     expr_stmt at test.cpp:8
                        convert_expr
                           modify_expr
                              var_decl
  ::comparison::result type: boolean_type at test.cpp:3
                              integer_cst
            return_expr at test.cpp:10
               init_expr
                  result_decl ::comparison::<unnamed>
  type: boolean_type at test.cpp:1
                  var_decl ::comparison::result type:
  boolean_type at test.cpp:3
function_decl ::comparison type: function_type at
  test.cpp:1
```

Пример исходного текста на ЯВУ Python

```
class NextClass:
    def printer(self, text):
        self.message = text
        print self.message
```

Промежуточное представление, построенное с помощью утилититы logilab-python

```
Module(simple)
  body = [
  Class(NextClass)
    bases = [
    ]
    body = [
    Function(printer)
      decorators =
      args =
      Arguments()
          args = [
          AssName(self)
          AssName(text)
          ]
          defaults = [
          ]
```

```
body = [
      Assign()
        targets = [
        AssAttr(message)
          expr =
          Name(self)
        ]
        value =
        Name(text)
      Print()
        dest =
        values = [
          Getattr(message)
            expr =
            Name(self)
          ]
      ]
    ]
```

Промежуточное xml-представление AST, полученное с помощью препарата Jython

```
<project name="SimpleProject">
  <package name="__default__">
    <module name="simple.py">
      <classdef name="NextClass">
        <def name="printer">
          <arguments>
            <arg name="self"/>
            <arg name="text"/>
          </arguments>
          <assign>
            <lvalue>
              <field name="message">
                <var name="self"/>
              </field>
            </lvalue>
            <rvalue>
              <var name="text"/>
            </rvalue>
          </assign>
          <print>
            <field name="message">
              <var name="self"/>
            </field>
          </print>
        </def>
      </classdef>
```

Спасибо за внимание