DCSinpo
28 янв 2023 в 17:38

Создаем свой собственный язык программирования с использованием LLVM. Часть 1: Лексический и синтаксический анализ

29K

Open source*, Программирование*, Компиляторы*

Из песочницы

Оглавление серии

Об авторе

С 2003 года, когда я поступил в ВУЗ, по нынешнее время я написал много различных парсеров, от простых (чтение конфигурационных файлов в ini, json и yaml форматах) до более сложных (полноценный парсер C++ подобного языка с поддержкой обобщенного программирования в виде шаблонов (схожих по сложности с тем что есть в языках C++ и D), ООП с множественным наследованием, pattern matching и др.), интерпретаторов для различных языков (от простых академических, таких как Brainfuck, до более серьезных скриптовых языков, которые были использованы в различных проектах (в основном различных pet-проектов), которые позволяли решать различные задачи от расчета значений математических функций для построения трехмерных сеток объектов с последующей их визуализацией, до интерпретаторов, позволяющих разработчику/пользователю расширять

функционал ПО без модификации исходного кода самого приложения (например создание анимаций, рисования различных объектов, элементов управления и программирование обработчиков для этих элементов управления).

Еще когда я учился в институте мне было интересно изучать различные языки программирования. Позже мне захотелось узнать, как они были устроены внутри, как они работают. Мне было интересно, как из исходных текстов которые мы даем на вход компилятору получается программа, которая может быть запущена и которая делает то, что мы запрограммировали. Что привело меня к изучению теории компиляции и дизайну языков программирования. Я прочитал книгу, которая в свое время была классикой [1], так же во время чтения ее я наткнулся на очень интересную книгу [2], которая была больше направленна не на теорию, а на практику. И тогда у меня зародилась идея написать свой собственный компилятор, для своего собственного языка программирования, но все мои попытки воплотить мою идею в жизнь сталкивались с проблемой, что для того что бы написать хороший компилятор, который мог генерировать эффективный код и быть кросс платформенным, нужны знания, которые сложно получить по учебникам (а именно создание backend компилятора, хотя в это время я уже понимал как различные конструкции языка C++ работали под «капотом», такие как виртуальные таблицы, множественное наследование и много другое). В русском интернете эта тема была не освещена вообще, а в иностранном это что-то полезное было очень сложно найти. Но в какой-то момент я наткнулся на проект под названием LLVM, который позиционировал себя как набор библиотек и инструментов, которые могут помочь в написании компиляторов. Я начал изучать официальную документацию, но ее оказалось не достаточно, т. к. она описывала только простые кейсы использования данного «комбайна», язык Kaleidoscope демонстрировал только базовые концепты, которые могли бы помочь в создании своего языка программирования. Благо на тот момент уже были созданы различные реализации языков программирования, которые использовали LLVM в качестве backend (clang, ldc, crack и несколько других реализаций), соответственно я

начал изучать как различные конструкции были реализованы в них. Так и родился simple

Simple

Изначально данный язык был задуман как обобщение всех моих знаний в теории компиляции и применении их в связке с LLVM, для создания простого языка с базовой поддержкой ООП, которые могли бы быть положены в серию статей по созданию языка программирования. Изначально исходный код был написан в итеративном подходе, пригодном для написания серии статей (т. е. наращивание функционала из статьи в статью), но после написания всего планированного функционала, проект по тем или иным причинам был заброшен, поэтому сами статьи пришлось писать с нуля (т. к. исходные версии были утеряны), а сам исходный код пришлось адаптировать (т. к. осталась только финальная версия проекта, без его промежуточных частей). Финальная версия была написана около 11 лет назад (и использовался LLVM 3.0), в новой версии был взят исходный код старого проекта и произведены следующие изменения:

1. Смена версии LLVM на 14, т. к. версии частично не совместимы, поэтому были произведены некоторые изменения, чтобы программа работала корректно с новыми реалиями;
2. Изменена структура проекта (за основу был взята структура описанная в [3]).

Сам код по большей части остался в том виде, в каком он был 11 лет назад. Т.к. сам я уже 3 года не пишу на C++, и в последние годы его использования я периодически следил за историей его развития, но не применял большую часть нововведений на практике, поэтому некоторые части кода могут не подходить под правила хорошего тона в написании современных приложений на языке C++. Но должно быть достаточным для базового понимания того, как те или иные вещи можно было бы реализовать в своем языке с использованием LLVM в качестве backend

(пожелания по улучшению просьба оставлять в комментариях или в личных сообщениях).

По моему ощущению за эти 11 лет даже с появлением некоторого количества книг по LLVM, ситуация не сильно изменилась в лучшую сторону и если кто-то хочет написать что-то серьезнее чем простой язык программирования вида Kaleidoscope, то придется все равно собирать все знания по крупицам и копаться в исходных кодах компиляторов для различных языков программирования (если это не так, то был бы признателен если бы Вы поделились ссылками и своими размышлениями по данной тематике в комментариях). Поэтому не смотря на то, что данный проект был создан давно, знания которые были собраны в нем могли бы помочь многим, кто хотел бы создать свой язык программирования и не знает с чего начать и где смотреть, как можно было бы реализовать более высокоуровневые аспекты языка.

О серии

Сам цикл будет разбит на несколько статей, в которых мы реализуем данный язык.

Мы начнем с простого подмножества в котором доступны только два типа данных int и float, пользователь может объявлять локальные переменные и описывать функции, использовать различные арифметические операции над ними, а так же будут доступны управляющие конструкции if, while, for, а так же break, continue и return. Данное подмножество будет реализовано в рамках нескольких статей начиная с лексического и синтаксического анализаторов, затем мы добавим семантический анализатор (проверка типов и корректности программы) и в заключении будет реализована генерация промежуточного кода для LLVM.

После реализации базового функционала, он будет постепенно расширяться путем добавления в язык: массивов, указателей, структур, поддержка строковых констант, поддержка классов с одиночным наследованием и динамической диспетчеризацией методов, поддержкой

конструкторов, деструкторов, а так же возможность перегрузки функций с выбором подходящей функции в зависимости от типов аргументов.

В результате у нас получится язык, на котором можно будет написать программу вида:

```
fn printLn(s1: string, s2: string) {
  print(s1);
  print(s2);
  printLn("");
}

fn printLn(s: string, i: int) {
  print(s);
  print(i);
  printLn("");
}

class Shape {
  virt draw() {
  }
}

class Square extends Shape {
  side: int = 0;

  new(s: int) {
    side = s;
  }

  impl draw() {
    printLn("Square.draw side: ", side);
  }
}

class Circle extends Shape {
```

```
  radius: int = 0;

  new(r: int) {
    radius = r;
  }

  impl draw() {
    printLn("Circle.draw radius: ", radius);
  }
}

fn main(): float {
  for let i: int = 0; i < 10; ++i {
    let p: Shape* = 0;

    if i % 2 == 0 {
      p = new Square((i + 1) * 10);
    } else {
      p = new Circle((i + 1) * 5);
    }

    p.draw();
    del p;
  }

  return 0.0;
}
```

В данных статьях я не буду углубляться в деталях в тех местах, где информацию найти достаточно просто, в таких случаях, я дам базовую информацию необходимую для понимания и дам информацию, где можно узнать более подробно про все это. В местах, где я считаю, что информацию найти сложнее, я постараюсь дать как можно больше информации и при возможности дам ссылки на первоисточники.

Для упрощения, что бы сразу можно видеть результаты работы весь генерируемый код будет запускаться в JIT, но по возможности генерацию машинного кода и исполняемых файлов можно будет добавить самим. Пример того, как это можно сделать можно посмотреть в официальном материале на сайте LLVM [4].

Замечание по генерируемому коду: в процессе написания изначальной версии я очень много времени провел изучая код, который генерировал clang для различных конструкций языка C++, поэтому на тот момент качество кода генерированного данным интерпретатором был сопоставим с тем, что генерировал clang, включая различные оптимизации по переиспользованию блоков. С тех пор прошло много времени, но код до сих пор в большинстве ситуаций совпадает с тем, что генерирует clang для программы, если ее переписать один-в-один на C++ [5].

Если данная серия будет интересна, то возможно будет ее продолжение, у меня есть несколько идей о том, что еще можно будет добавить в язык. Например это может быть:

1. Поддержка модулей (import и export);
2. Добавление области видимости для переменных и членов класса (private, protected, public);
3. Добавление множественного наследования по средством интерфейсов (или полного аналога, как в C++);
4. Вывод типов (type inference), сопоставление с образцом (pattern matching);
5. Обобщенное программирование (generics);
6. Вычисление во время компиляции (template metaprogramming или constant expressions).

Что такое компилятор?

Согласно Wikipedia компилятор — программа, переводящая написанный на языке программирования текст в набор машинных кодов.

Традиционно компилятор состоит из двух больших блоков:

1. Frontend – основная задача которого заключается в чтении исходных кодов на конкретном языке программирования, анализа его на наличие как синтаксических, так семантических ошибок, а в случае их отсутствия, генерации промежуточного представления программы, которое может быть передано в backend, для последующей обработки. Один из вариантов представления полученного после успешной отработки frontend является аннотированное AST (Abstract Syntax Tree), в котором содержится вся необходимая информация о программе;
2. Backend – основная задача данного блока заключается в генерации программы для целевой системы. Это может быть запускаемый файл для конкретной ОС, байт код для некой виртуальной машины, так же это может быть набор объектных файлов, которые могут быть позже собраны в результирующий файл после их линковки с помощью специальной программы линкера.

Такое разделение сделано не случайно, оно позволяет переиспользовать отдельные части компилятора для различных целей. Т.к. frontend отвечает за работу с исходным языком, то результат его работы можно использовать для написания различных программ, которые могут помогать решать те или иные задачи по работе с исходным кодом (например программы для форматирования текста, статический анализ программы на наличия в ней скрытых и сложных в нахождении дефектов, рефакторинга, интеллектуальных подсказок и многого другого). В то же время один раз написанный backend можно использовать для написания компиляторов для других языков программирования.

Раньше для написания backend приходилось писать генератор кода под каждую платформу и операционную систему, с появлением LLVM данная

задача упростилась, т. к. достаточно написать генератор в LLVM IR и дальше все остальное можно будет сделать через утилиты, идущие с LLVM (хочу уточнить, что какую-то часть платформозависимых функций придется все равно реализовать, например поддержку ABI для платформы, но все равно большую часть LLVM берет на себя).

Frontend в свою очередь состоит из:

1. Лексический анализатор;
2. Синтаксический анализатор (или парсер);
3. Семантический анализатор (или проверка типов);
4. Генератор промежуточного кода.

Далее мы рассмотрим более подробно первые 2 пункта, последние 2 будут более подробно описаны в следующих статьях.

Backend в свою очередь обычно состоит из:

1. Оптимизатор промежуточного кода;
2. Генерация кода для целевой платформы.

В рамках данной серии backend мы рассматривать не будем.

Лексический анализ

Лексический анализатор — часть компилятора, которая читает текст программы из входного потока и преобразовывает его в набор лексем или токенов (неделимые примитивы языка, такие как ключевые слова, идентификаторы, строковые и числовые константы, операторы и др), которые в простейшем случае представляют собой структуру содержащую информацию о типе прочитанного токена и его значение. Так же лексический анализатор часто убирает из выходного потока «шум» (конструкции языка, которые не влияют на дальнейший разбор программы, такие как комментарии и пробельные символы (в языках, где

отступы не являются значимыми для разбора программы)). Причина данного шага банальна, проще работать с представлением, которое просто использовать в машинной обработке, в котором нет никаких неоднозначностей. Например для синтаксического анализатора может быть не важно значение целочисленной константы, а важно то, что это целочисленная константа. Так же часто в языках программирование одна и та же лексическая конструкция может быть записана множеством способов. Например, в языке C++ целочисленное число 10 может быть записано множеством способов

```
10  // Десятичная система
012 // Восьмеричная
0xA // Шестнадцатеричная
0b1010 // Двоичная
```

И это только некоторые способы записи, которые после лексического анализа может быть преобразовано в структуру вида

type	value
number	10

Также лексический анализатор может иметь дело с различными кодировками и специальными конструкциями, которые в результате все равно будут приводится к унифицированному набору токенов, которые будут удобны для обработки (например различные escape и Unicode последовательности в строковых литералах некоторых языков).

Обычно все лексические конструкции можно описать в виде набора регулярных выражений, которые могут распознать каждую лексическую конструкцию. Существуют программы, которые позволяют взять на себя работу по созданию лексического анализатора, на вход таких программ подается описание лексем языка в виде регулярных выражений (обычно записываются в специальном формате, понимаемых конкретной

утилитой) и они на основе данного входного файла генерируют исходный код содержащий лексический анализатор (часто в виде конечного автомата, который понимает заданный язык) для заданного входного языка. Они доступны для многих языков программирования, например для C++ одна из таких утилит называется flex.

Более подробнее про лексический анализ, алгоритмы, которые можно применить для преобразования регулярных выражений в конечный автомат и другое, можно прочитать в [1].

Для данной серии мы реализуем лексический анализатор сами.

Для начала я покажу основные структуры, которые используются в лексическом анализаторе и других частях программы.

```
struct Name {
  const char *Id;
  int Kind;
  size_t Length;
};
```

Данная структура хранит информацию о идентификаторах и ключевых словах. В процессе своей работы лексический анализатор при обнаружении идентификатора проверяет его наличие во внутренней таблице символов и в случае его отсутствия добавляет новую запись в соответствии с типом (идентификатор или ключевое слово), а при наличии он возвращает уже имеющуюся запись из таблицы символов. Это сделано для упрощения дальнейшей работы, т. к. все идентификаторы с одним и тем же значением будут всегда возвращать одно и тоже значение, то для сравнения двух идентификаторов в дальнейшем, достаточно будет сравнить указатели и если они совпадут, то они содержат одинаковые значения.

Сама же таблица символов имеет следующий вид:

```
class NamesMap {
  bool IsInit;
  llvm::StringMap<Name> HashTable;
  Name *addName(StringRef Id, tok::TokenKind TokenCode);
public:
  NamesMap(): IsInit(false) { }
  void addKeywords();
  Name *getName(StringRef Id);
};
```

Где TokenKind обычный enum вида:

```
namespace tok {
enum TokenKind : unsigned short {
#define TOK(ID, TEXT) ID,
#include "simple/Basic/TokenKinds.def"
};
} // namespace tok
```

▾ Hidden text

```
// simple/Basic/TokenKinds.def
#ifndef TOK
#define TOK(ID, TEXT)
#endif

#ifndef KEYWORD
#define KEYWORD(ID, TEXT) TOK(ID, TEXT)
#endif

KEYWORD(Int, "int")
KEYWORD(Float, "float")
```

```
KEYWORD(Void, "void")
KEYWORD(Def, "fn")
KEYWORD(If, "if")
KEYWORD(Else, "else")
KEYWORD(For, "for")
KEYWORD(While, "while")
KEYWORD(Return, "return")
KEYWORD(Break, "break")
KEYWORD(Continue, "continue")
KEYWORD(Var, "let")
TOK(Identifier, "identifier")

TOK(IntNumber, "")
TOK(FloatNumber, "")
TOK(CharLiteral, "")
TOK(StringConstant, "")

TOK(Plus, "+")
TOK(PlusPlus, "++")
TOK(Minus, "-")
TOK(MinusMinus, "--")
TOK(Not, "!")
TOK(Tilda, "~")
TOK(Mul, "*")
TOK(Div, "/")
TOK(Mod, "%")
TOK(LShift, "<<")
TOK(RShift, ">>")
TOK(LogAnd, "&&")
TOK(LogOr, "||")
TOK(BitAnd, "&")
TOK(BitOr, "|")
TOK(BitXor, "^")
TOK(Less, "<")
TOK(Greater, ">")
TOK(LessEqual, "<=")
TOK(GreaterEqual, ">=")
```

```
  TOK(Equal, "==")
  TOK(NotEqual, "!=")
  TOK(Assign, "=")
  TOK(Comma, ",")
  TOK(Dot, ".")
  TOK(Question, "?")
  TOK(Colon, ":")
  TOK(Semicolon, ";")
  TOK(OpenParen, "(")
  TOK(CloseParen, ")")
  TOK(BlockStart, "{")
  TOK(BlockEnd, "}")
  TOK(OpenBrace, "[")
  TOK(CloseBrace, "]")

  TOK(Invalid, "")
  TOK(EndOfFile, "EOF")

  #undef KEYWORD
  #undef TOK
```

Сам токен, который возвращается в результате лексического анализа имеет вид:

```
class Token {
  friend class Lexer;

  const char *Ptr; ///< Положение токена во входном буфере
  size_t Length; ///< Длина токена в символах
  tok::TokenKind Kind; ///< Тип
  union {
    Name *Id; ///< Значение, если это идентификатор или ключевое слов
    char *Literal; ///< Значение, если это константа (например целочи
```

```
  };
  ...
};
```

Как видно из выше представленной класса, токен хранит минимальное количество информации:

1. Положение токена во входном буфере, необходимо для диагностики (в случае возникновения ошибок на дальнейших стадиях анализа программы);
2. Длина токена и его тип;
3. Значение токена (для оптимизации места используется union).

Сам же лексический анализатор имеет следующий вид:

```
class Lexer {
  SourceMgr &SrcMgr;
  DiagnosticsEngine &Diags;

  StringRef BufferStart;
  const char *CurPos;

  unsigned CurBuffer = 0;

  static NamesMap IdsMap;

  llvm::SMLoc getLoc(const char *Pos) const {
    return llvm::SMLoc::getFromPointer(Pos);
  }

public:
  Lexer(SourceMgr &SrcMgr, DiagnosticsEngine &Diags)
    : SrcMgr(SrcMgr), Diags(Diags) {
```

```
    CurBuffer = SrcMgr.getMainFileID();
    BufferStart = SrcMgr.getMemoryBuffer(CurBuffer)->getBuffer();
    CurPos = BufferStart.begin();
    IdsMap.addKeywords();
  }

  DiagnosticsEngine &getDiagnostics() const {
    return Diags;
  }

  void next(Token &Result);
};
```

Он имеет всего одну публичную значимую функцию next, которая считывает новый токен. Реализация самой функции представлена ниже:

```
void Lexer::next(Token &Result) {
  // Устанавливаем токен в качестве недействительного
  Result.Kind = tok::Invalid;

  // Сохраняем текущую позицию во входном потоке и будем использовать
  // позицию для чтения, что бы не испортить состояние лексического
  // анализватора
  const char *p = CurPos;

  // Считываем символы, пока не найдем корректный токен
  while (Result.Kind == tok::Invalid) {
    const char *tokenStart = p;
// Макрос для чтения одинарного оператора
#define CHECK_ONE(CHR, TOK) \
      case CHR: \
        Result.Length = 1; \
        Result.Kind = TOK; \
        break
// Макрос для чтения операторов вида ">" и ">=". Возвращает, Т1 если
```

```
// встретился только CH1 и T2, если следом идет CH2
#define CHECK_TWO(CH1, CH2, T1, T2) \
      case CH1: \
        if (*p == CH2) { \
          ++p; \
          Result.Length = 2; \
          Result.Kind = T1; \
        } else { \
          Result.Length = 1; \
          Result.Kind = T2; \
        } \
        break
    // Переходим к следующему символу, но запоминаем его значение и сі
    // что это за символ
    switch (char ch = *p++) {
      case 0:
        // Это конец файла, возвращаем его, но предварительно возвращ
        // положение лексического анализатора на начало этого токена
        // можно его перечитать, если это необходимо)
        --p;
        Result.Kind = tok::EndOfFile;
        break;

      case '\n': case '\r': case ' ': case '\t': case '\v': case '\f'
        // Считываем все пробельные символы в цикле, т. к. их может бы
        // много
        while (charinfo::isWhitespace(*p)) {
          ++p;
        }
        // Продолжаем искать токен
        continue;

      // Анализ на операторы состоящих из одного символа
      CHECK_ONE('~', tok::Tilda);
      CHECK_ONE('*', tok::Mul);
      CHECK_ONE('%', tok::Mod);
      CHECK_ONE('^', tok::BitXor);
```

```
CHECK_ONE(',', tok::Comma);
CHECK_ONE('?', tok::Question);
CHECK_ONE(':', tok::Colon);
CHECK_ONE(';', tok::Semicolon);
CHECK_ONE('(', tok::OpenParen);
CHECK_ONE(')', tok::CloseParen);
CHECK_ONE('{', tok::BlockStart);
CHECK_ONE('}', tok::BlockEnd);
// Анализ на операторы состоящих из одного символа или альтерна
// 2-х символов
CHECK_TWO('-', '-', tok::MinusMinus, tok::Minus);
CHECK_TWO('+', '+', tok::PlusPlus, tok::Plus);
CHECK_TWO('!', '=', tok::NotEqual, tok::Not);
CHECK_TWO('=', '=', tok::Equal, tok::Assign);
CHECK_TWO('|', '|', tok::LogOr, tok::BitOr);
CHECK_TWO('&', '&', tok::LogAnd, tok::BitAnd);

case '/':
  if (*p == '/') {
    // Одно строчный комментарий
    ++p;
    // В цикле ищем конец строки или конец файла
    while (*p && (*p != '\r' && *p != '\n')) {
      ++p;
    }
    break;
  } else if (*p == '*') {
    // Это много строчный комментарий
    unsigned Level = 1;
    ++p;

    while (*p && Level) {
      // Считываем все символы пока не найдем конец файла или п
      // выйдем из вложенного много строчного комментария
      if (*p == '/' && p[1] == '*') {
        p += 2;
        ++Level;
```

```
      // Проверяем на конец считывания комментария
      } else if (*p == '*' && p[1] == '/' && Level) {
        // Уменьшаем уровень вложенности
        p += 2;
        --Level;
      } else {
        // Продолжаем искать конец комментария
        ++p;
      }
    }

    if (Level) {
      // Комментарий не закрыт, сообщаем об ошибке
      Diags.report(getLoc(p), diag::ERR_UnterminatedBlockCommen
    }

    continue;
  } else {
    // Это обычный оператор деления
    Result.Length = 1;
    Result.Kind = tok::Div;
    break;
  }

case '<':
  if (*p == '=') {
    // Это <=
    ++p;
    Result.Length = 2;
    Result.Kind = tok::LessEqual;
    break;
  } else if (*p == '<') {
     // Это <<
    ++p;
    Result.Length = 2;
    Result.Kind = tok::LShift;
    break;
```

```
    } else {
      // Это <
      Result.Length = 1;
      Result.Kind = tok::Less;
      break;
    }

  case '>':
    if (*p == '=') {
       // Это >=
      ++p;
      Result.Length = 2;
      Result.Kind = tok::GreaterEqual;
      break;
    } else if (*p == '>') {
      // Это >>
      ++p;
      Result.Length = 2;
      Result.Kind = tok::RShift;
      break;
    } else {
      // Это >
      Result.Length = 1;
      Result.Kind = tok::Greater;
      break;
    }

  case '1': case '2': case '3': case '4': case '5': case '6': cas
  case '8': case '9':
    // Считываем целую часть числа
    while (charinfo::isDigit(*p)) {
      ++p;
    }

  case '0':
    Result.Kind = tok::IntNumber;
    // Проверяем на то, что вещественное число
```

```
if (*p == '.') {
  const char *firstDigit = p++;
  // Считываем все цифры
  while (charinfo::isDigit(*p)) {
    ++p;
  }

  // Проверяем на наличие экспоненты у числа
  if (*p == 'e' || *p == 'E') {
    ++p;
    // Проверка на наличие знака в экспоненте
    if (*p == '+' || *p == '-') {
      ++p;
    }

    // Должна быть хотя бы одна цифра после [eE][+-]
    firstDigit = p;

    while (charinfo::isDigit(*p)) {
      ++p;
    }

    // Проверяем, что в экспоненте есть хотя бы одна цифра
    if (p == firstDigit) {
      Diags.report(getLoc(p),
                   diag::ERR_FloatingPointNoDigitsInExponent)
    }
  }

  Result.Kind = tok::FloatNumber;
}

// Создаем токен на основе константы
Result.Length = (int)(p - tokenStart);
Result.Literal = new char[Result.Length + 1];
memcpy(Result.Literal, tokenStart, Result.Length);
Result.Literal[Result.Length] = 0;
```

```
        break;

      default:
        // Проверяем, что это идентификатор
        if (charinfo::isIdentifierHead(ch)) {
          // Считываем все символы идентификатора
          while (charinfo::isIdentifierBody(*p)) {
            ++p;
          }
          // Создаем токен для идентификатора или ключевого слова
          size_t length = (size_t)(p - tokenStart);
          Name *name = IdsMap.getName(StringRef(tokenStart, length));

          Result.Id = name;
          Result.Kind = (tok::TokenKind)name->Kind;
          Result.Length = name->Length;

          break;
        } else {
           // Не определенный символ
          Diags.report(getLoc(p), diag::ERR_InvalidCharacter);
          break;
        }
    }

    Result.Ptr = tokenStart;
  }

  // Обновляем текущую позицию в лексическом анализаторе
  CurPos = p;
}
```

Для упрощения реализации при возникновении ошибки, выводится сообщение об ошибке и работа программы завершается. Если бы необходимо выводить все найденные ошибки (как в большинстве

серьезных компиляторах), то можно было бы ввести дополнительный тип токена Error, и возвращать его, сам ошибочный символ игнорировать, а на последующих стадиях обрабатывать что если вернулся Error, то происходило бы восстановление от ошибок. Аналогично и с самой функцией, при большем количестве и сложности лексем языка (например поддержка Unicode, различных escape последовательностей, и различных типов строк (как в C++ или D), ее можно было бы разбить на основной цикл, а разбор конкретных лексем вынести в отдельные функции (например scanIdentitifer, scanNumber, scanHexString, scanString и т.д).

Синтаксический анализ

Синтаксический анализатор (парсер) — часть компилятора, основной целью которой является анализ потока токенов на принадлежность грамматике конкретного языка.

Часто результатом работы парсера является AST, которое содержит всю необходимую информацию для дальнейших стадий компилятора.

Данная реализация парсера была написана с помощью метода рекурсивного спуска и метода разбора выражений с приоритетом операций. Про другие варианты построения парсеров, а так же вариант использования специальных программ для генерации парсеров (например bison для C++) можно прочитать в [1] или в интернете.

Для упрощения работы с потоком токенов при разборе грамматики, мы будем использовать следующий набор классов, который позволяет работать с токенами, как с итераторами, что позволяет писать код вида

```
if (CurPos == tok::Identifier) {
  ...
  ++CurPos;
}
if (CurPos + 1 == tok::Identifier) {
```

```
  ...
}
```

Интерфейс классов:

```
class TokenStream {
  struct StreamNode {
    StreamNode(StreamNode *next, StreamNode *prev) ;

    Token Tok;         ///<  Сам токен
    StreamNode *Next; ///< Следующий элемент в списке
    StreamNode *Prev; ///< Предыдущий элемент в списке
  };

  class TokenStreamIterator {
  public:
    TokenStreamIterator &operator=(const TokenStreamIterator &other);

    bool empty() const ;

    llvm::SMLoc getLocation() const ;

    bool operator ==(tok::TokenKind tok) const;
    bool operator !=(tok::TokenKind tok) const;
    const Token &operator *() const;
    const Token *operator ->() const;
   TokenStreamIterator operator ++(int);
   TokenStreamIterator &operator ++() ;
    TokenStreamIterator operator –(int);
    TokenStreamIterator &operator --() ;
    TokenStreamIterator operator +(int count);
    TokenStreamIterator operator -(int count);

    friend class TokenStream;
```

```
  private:
    TokenStream *CurStream; ///< Родительский контейнер
    StreamNode *CurPos;     ///< Текущая позиция в контейнере

    TokenStreamIterator(TokenStream *source = nullptr,
                        StreamNode *curPos = nullptr) ;
  };
  /// Экземпляр лексического анализатора, с исходным кодом конкретног
  Lexer *Lex;
  StreamNode *Head; ///< Первый элемент в списке
  StreamNode *Tail; ///< Последний элемент в списке
  bool ScanDone;    ///< true — если считывание полностью завершено

  StreamNode *next(StreamNode *curPos);

public:
  typedef TokenStreamIterator iterator;

  TokenStream(Lexer *lexer);

  ~TokenStream();

  DiagnosticsEngine &getDiagnostics() const;

  TokenStreamIterator begin();
};
```

Для упрощение некоторых проверок в парсере есть функция check, которая проверяет то, что тип текущего токена совпадает с типом ожидаемого, а в случае расхождения выдает ошибку:

```
 void Parser::check(tok::TokenKind tok) {
  if (CurPos != tok) {
    getDiagnostics().report(CurPos.getLocation(),
```

```
                                diag::ERR_Expected,
                                tok::toString(tok),
                                tok::toString(CurPos->getKind()));
    }

    ++CurPos;
  }
```

Для упрощения работы, как и с ошибками в лексическом анализаторе, будем просто выводить сообщение об ошибке и завершать работу приложения. Но возможны различные варианты восстановления от ошибок, которые зависят от конкретной конструкции и типа ошибки, например некоторые ошибки могут быть обработаны путем добавление необходимых элементов и продолжения анализа (например отсутствие операнда у бинарного или унарного оператора). Так же это может быть вариант панического восстановления, путем пропуска токенов и поиска ближайшего синхронизирующего токена (например ";" или "}"). Про все эти методы можно более подробно прочитать в [1] или в интернете.

В процессе своей работы парсер будет возвращать AST, со следующим списком базовых типов (все типы поддерживают RTTI, который используется в LLVM [6].

```
struct TypeAST {
  enum TypeId {
    TI_Void,     ///< void
    TI_Bool,     ///< булево значение
    TI_Int,      ///< int
    TI_Float,    ///< float
    TI_Function  ///< функция
  };

  TypeAST(int typeKind);

  ...
```

```
  int TypeKind; ///< тип в иерархии типов
  static llvm::StringSet<> TypesTable; ///< список используемых типов
};
```

Данный класс является базовым для иерархии типов в дереве (например базовые типы (int, float, bool, void), тип для функций. В последующих статьях серии данный список будет расширен и будут добавлены типы для строк, символов, классов, структур, указателей и массивов.

```
struct ExprAST {
  enum ExprId {
    EI_Int,          ///< целочисленные константы
    EI_Float,        ///< константы для чисел с плавающей точкой
    EI_Id,           ///< использование переменной
    EI_Cast,         ///< преобразование типов
    EI_Unary,        ///< унарная операция
    EI_Binary,       ///< бинарная операция
    EI_Call,         ///< вызов функции
    EI_Cond          ///< тернарный оператор
  };

  ExprAST(llvm::SMLoc loc, int exprKind, TypeAST *type = nullptr) ;
  virtual ~ExprAST();

  ...

  llvm::SMLoc Loc; ///< расположение выражения в исходном файле
  int ExprKind; ///< тип в иерархии выражений
};
typedef llvm::SmallVector<ExprAST*, 4> ExprList;
```

Данный класс является базовым для иерархии для различных выражений языка, например константы различных типов, вызов функции и т. п. В последующих статьях список поддерживаемых выражений будет расширен и будут добавлены операторы для взятия адреса, разыменования, индексации элементов массива или значения хранящемуся по указателю, а так же обращения к методам и членам классов/структур.

```
struct StmtAST {
  enum StmtId {
    SI_Expr, ///< выражение
    SI_Var, ///< объявление локальной переменной
    SI_For, ///< цикл for
    SI_While, ///< цикл while
    SI_If, ///< инструкция ветвления if
    SI_Return, ///< инструкция возврата из функции return
    SI_Continue, ///< инструкция перехода к следующей итерации цикла
    SI_Break, ///< инструкция досрочного выхода из цикла break
    SI_Block ///< блок инструкций
  };

  StmtAST(llvm::SMLoc loc, int stmtKind) ;
  virtual ~StmtAST();

  ...

  llvm::SMLoc Loc; ///< расположение выражения в исходном файле
  int StmtKind; ///< тип в иерархии инструкций
};

typedef llvm::SmallVector< StmtAST*, 4 > StmtList;
```

Данный класс является базовым для иерархии различных инструкций языка, такие как условные операторы if, циклы while и for, а так же

операторы выхода из цикла break, continue, инструкция возврата из функции, объявление локальных переменных и выражений, а так же блок с набором инструкций.

```
struct SymbolAST {
  enum SymbolId {
    SI_Variable,    ///< переменная
    SI_Function,    ///< функция
    SI_Module,      ///< модуль
    SI_Parameter,   ///< параметр функции
    SI_Block        ///< блочная декларация
  };

  SymbolAST(llvm::SMLoc loc, int symbolKind, Name *id);
  virtual ~SymbolAST();

  ...

  llvm::SMLoc Loc;  ///< расположение в исходном файле
  int SymbolKind;   ///< тип в иерархии объявлений
  Name *Id;         ///< имя объявлений
};
```

Данный класс является базовым для иерархии объявлений переменных, функций, параметров функции и модуля. В последующих статьях список поддерживаемых объявлений будет расширен объявлениями структур, классов, а так же списком перегруженных функций.

Иерархия для типов

На данный момент для типов у нас будет всего два типа BuiltinTypeAST, который отвечает за базовые типы, такие как int, float, bool и void.

```
struct BuiltinTypeAST : TypeAST {
  static TypeAST *get(int type);
private:
  BuiltinTypeAST(int type) ;
};
```

и FuncTypeAST, который отвечает за прототип функции (тип возвращаемого ей значения и типы ее параметров)

```
struct ParameterAST {
  ParameterAST(TypeAST* type, Name* id);

  TypeAST* Param; ///< тип параметра
  Name* Id; ///< имя
};

typedef llvm::SmallVector< ParameterAST*, 4 > ParameterList;

struct FuncTypeAST : TypeAST {
  FuncTypeAST(TypeAST* returnType, const ParameterList& params);

  TypeAST* ReturnType; ///< тип возвращаемого значение или nullptr
  ParameterList Params; ///< список параметров функции
};
```

Иерархия выражений

Для констант целочисленных и с плавающей точкой будем использовать следующие классы:

```
struct IntExprAST : ExprAST {
  IntExprAST(llvm::SMLoc loc, int value) ;

  int Val; ///< значение
```

```
};

struct FloatExprAST : ExprAST {
  FloatExprAST(llvm::SMLoc loc, double value) ;

  double Val; ///< значение
};
```

Для использования переменных будем использовать следующий класс (на данный момент может ссылаться на параметры функции или переменные объявленные в теле функции, но в дальнейшем может ссылать и на члены и методы классов/структур:

```
struct IdExprAST : ExprAST {
  IdExprAST(llvm::SMLoc loc, Name *name) ;

  Name* Val; ///< имя ссылаемой переменной
};
```

Для преобразования типов будем использовать следующий класс (т. к. явных преобразований в языке не будет, то этот класс будет в основном использоваться как вспомогательный другими выражениями, для приведения операндов к единому типу (например int к float или int/float к bool в условных выражениях)):

```
struct CastExprAST : ExprAST {
  CastExprAST(llvm::SMLoc loc, ExprAST *expr, TypeAST *type) ;

  ExprAST* Val; ///< выражение, которое нужно преобразовать
};
```

Для выражений с одним операндом (такие как +,-, ++, -- и ~) будем использовать следующий класс:

```
struct UnaryExprAST : ExprAST {
  UnaryExprAST(llvm::SMLoc loc, int op, ExprAST *value) ;

  int Op; ///< оператор
  ExprAST* Val; ///< операнд выражения
};
```

Для выражений с двумя аргументами будем использовать следующий класс:

```
struct BinaryExprAST : ExprAST {
  BinaryExprAST(llvm::SMLoc loc, int op, ExprAST *lhs, ExprAST *rhs)

  int Op; ///< оператор
  ExprAST* LeftExpr; ///< левый оператор
  ExprAST* RightExpr; ///< правый оператор
};
```

Для тернарного оператора ? : будем использовать следующий класс:

```
struct CondExprAST : ExprAST {
  CondExprAST(llvm::SMLoc loc, ExprAST *cond, ExprAST *ifExpr,
    ExprAST* elseExpr) ;

  ExprAST* Cond; ///< условие
  ExprAST* IfExpr; ///< выражение, если условие истинно
  ExprAST* ElseExpr; ///< выражение, если условие ложно
};
```

Для выражения вызова функции будем использовать следующий класс:

```
struct CallExprAST : ExprAST {
  CallExprAST(llvm::SMLoc loc, ExprAST *callee, const ExprList &args)

  ExprAST* Callee; ///< функция для вызова
  ExprList Args; ///< список аргументов функции
};
```

Иерархия для инструкций

Для инструкций, которые содержат выражения (например вызов функции или присвоение переменной):

```
struct ExprStmtAST : StmtAST {
  ExprStmtAST(llvm::SMLoc loc, ExprAST *expr) ;

  ExprAST* Expr; ///< выражение, которое должно быть выполнено
};
```

Инструкция ветвления if:

```
struct IfStmtAST : StmtAST {
  IfStmtAST(llvm::SMLoc loc,
    ExprAST *cond,
    StmtAST *thenBody,
    StmtAST* elseBody) ;

  ExprAST* Cond; ///< условие для проверки
  /// инструкции, для выполнения в случае, если условие истинно
  StmtAST* ThenBody;
  /// инструкции, для выполнения в случае, если условие ложно
```

```
  StmtAST* ElseBody;
};
```

Цикл с предусловием:

```
struct WhileStmtAST : StmtAST {
  WhileStmtAST(llvm::SMLoc loc, ExprAST *cond, StmtAST *body) ;

  ExprAST* Cond; ///< условие для проверки
  StmtAST* Body; ///< инструкции тела цикла
};
```

Инструкция для досрочного выхода из цикла:

```
struct BreakStmtAST : StmtAST {
  BreakStmtAST(llvm::SMLoc loc) ;
};
```

Инструкция для перехода к следующей итерации цикла:

```
struct ContinueStmtAST : StmtAST {
  ContinueStmtAST(llvm::SMLoc loc) ;
};
```

Инструкция для возврата из функции:

```
struct ReturnStmtAST : StmtAST {
  ReturnStmtAST(llvm::SMLoc loc, ExprAST *expr) ;
```

```
  ExprAST* Expr; ///< значение, которое нужно вернуть (или nullptr)
};
```

Блок с инструкциями (например тело функции или цикла):

```
struct BlockStmtAST : StmtAST {
  BlockStmtAST(llvm::SMLoc loc, const StmtList &body) ;

  StmtList Body; ///< список инструкций для выполнения
};
```

Объявление локальной переменной:

```
struct DeclStmtAST : StmtAST {
  DeclStmtAST(llvm::SMLoc loc, const SymbolList &decls) ;

  SymbolList Decls; ///< список объявлений
};
```

Цикл for. Данный цикл, как и его аналог из C++, имеет три блока (для объявлений переменных цикла или инициализации цикла, условие цикла и операции, которые должны быть выполнены после каждой итерации), которые являются не обязательными, а так же обязательное тело цикла:

```
struct ForStmtAST : StmtAST {
  ForStmtAST(llvm::SMLoc loc,
    ExprAST *initExpr,
    const SymbolList &decls,
    ExprAST* cond,
    ExprAST* post,
    StmtAST* body);
```

```
  ExprAST* InitExpr; ///< выражение для инициализации переменных цикл
  SymbolList InitDecls; ///< список переменных цикла с их инициализац
  ExprAST* Cond; ///< условие цикла
  ExprAST* Post; ///< выражение для выполнения после каждой операции
  StmtAST* Body; ///< тело цикла
};
```

Иерархия для объявлений

Объявление переменной:

```
struct VarDeclAST : SymbolAST {
  VarDeclAST(llvm::SMLoc loc, TypeAST *varType, Name *id,
    ExprAST* value);

  TypeAST* ThisType; ///< тип символа
  ExprAST* Val; ///< инициализатор (может быть nullptr, если его нет)
};
```

Базовый класс для объявлений, которые могут сами содержать другие объявления (например функции, классы, структуры):

```
struct ScopeSymbol : SymbolAST {
  ScopeSymbol(llvm::SMLoc loc, int symbolKind, Name *id);

  SymbolAST* find(Name* id, int flags = 0);

  typedef std::map< Name*, SymbolAST* > SymbolMap;
  SymbolMap Decls; ///< вложенные объявления
};
```

Параметр функции:

```
struct ParameterSymbolAST : SymbolAST {
  ParameterSymbolAST(ParameterAST* param);

  ParameterAST* Param; ///< значение параметра (тип и имя)
};
```

Объявление функции:

```
struct FuncDeclAST : ScopeSymbol {
  FuncDeclAST(llvm::SMLoc loc, TypeAST *funcType, Name *id,
    StmtAST* body, int tok = tok::Def);

  TypeAST* ThisType; ///< прототип функции
  TypeAST* ReturnType;
 ///< тип возвращаемого значения
  StmtAST* Body; ///< тело функции
  int Tok; ///< токен при помощи которого была объявлена функция
};
```

Объявление модуля:

```
struct ModuleDeclAST : ScopeSymbol {
  ModuleDeclAST(DiagnosticsEngine &D, const SymbolList& decls);

  static ModuleDeclAST* load(
    llvm::SourceMgr &SrcMgr,
    DiagnosticsEngine &Diags,
    llvm::StringRef fileName
  );

  SymbolList Members; ///< список объявлений в модуле
```

```
  DiagnosticsEngine &Diag; ///< модуль диагностики
};
```

Разбор выражений

Для разбора примитивных выражений (целочисленные и вещественные константы, обращение к переменной, а также выражение в скобках) используется следующая функция:

```
/// primary-expr
///   ::= floating-point-literal
///   ::= integral-literal
///   ::= identifier
///   ::= '(' expr ')'
ExprAST *Parser::parsePrimaryExpr() {
  ExprAST *result = nullptr;

  switch (CurPos->getKind()) {
    case tok::FloatNumber:
      // Это число с плавающей точкой. Строим ветку дерева и считывае
      // следующий токен
      result = new FloatExprAST(CurPos.getLocation(),
                                strtod(CurPos->getLiteral().data(), n
      ++CurPos;
      return result;

    case tok::IntNumber:
      // Это целочисленная константа. Строим ветку дерева и считываем
      // следующий токен
      result = new IntExprAST(CurPos.getLocation(),
                              atoi(CurPos->getLiteral().data()));
      ++CurPos;
      return result;

    case tok::Identifier:
      // Это обращение к переменной или параметру функции. Строим вет
```

```
      // и считываем следующий токен
      result = new IdExprAST(CurPos.getLocation(), CurPos->getIdentif
      ++CurPos;
      return result;

    case tok::OpenParen:
      // Это выражение заключенное в скобки, считываем "(" и переходи
      // следующему токену
      ++CurPos;
      // Считываем выражение в скобках
      result = parseExpr();
      // Проверяем наличие ")"
      check(tok::CloseParen);
      return result;

    default:
      // Сообщаем об ошибке
      getDiagnostics().report(CurPos.getLocation(),
                              diag::ERR_ExpectedExpression);
      return nullptr;
  }
}
```

Следующий вид выражений, это постфиксные операции, такие как ++ и --, а так же вызов функции:

```
/// call-arguments
///   ::= assign-expr
///   ::= call-arguments ',' assign-expr
/// postfix-expr
///   ::= primary-expr
///   ::= postfix-expr '++'
///   ::= postfix-expr '--'
///   ::= postfix-expr '(' call-arguments ? ')'
```

```
ExprAST *Parser::parsePostfixExpr() {
   // Считываем выражение (константу, идентификатор или выражение в с
  ExprAST *result = parsePrimaryExpr();

  // Пытаемся найти постфиксные операторы или вызов функции
  for (;;) {
    llvm::SMLoc loc = CurPos.getLocation();

    switch (int op = CurPos->getKind()) {
      case tok::PlusPlus:
      case tok::MinusMinus:
        // Это "++" или "--". Создаем ветвь дерева в виде выражения с
        // операндами, но в качестве второго оператора указываем null
        // аналогии с постфиксными операторами в C++)
        result = new BinaryExprAST(loc, op, result, nullptr);
        // Считываем "++" или "--"
        ++CurPos;
        continue;

      case tok::OpenParen: {
        // Это вызов функции. Пропускаем "("
        ++CurPos;

        ExprList args;

        if (CurPos != tok::CloseParen) {
          // Это не ")", значит у нас есть аргументы для вызова функц
          for (;;) {
            // Считываем аргумент и добавляем его к списку аргументов
            ExprAST *arg = parseAssignExpr();
            args.push_back(arg);

            if (CurPos != tok::Comma) {
              // Если это не ",", то у нас больше нет аргументов для
              // функции
              break;
            }
```

```
            // Пропускаем ","
             ++CurPos;
           }
         }
        // Проверяем наличие ")" и строим ветку дерева для вызова функ
         check(tok::CloseParen);
         result = new CallExprAST(loc, result, args);
         continue;
       }

       default:
         // Больше нет никаких постфиксных операторов. Возвращаем
         // результирующее выражение
         return result;
     }
   }
}
```

Следующий вид выражений, это выражения с одним операндом:

```
/// unary-expr
///   ::= postfix-expr
///   ::= '+' unary-expr
///   ::= '-' unary-expr
///   ::= '++' unary-expr
///   ::= '--' unary-expr
///   ::= '~' unary-expr
///   ::= '!' unary-expr
ExprAST *Parser::parseUnaryExpr() {
  ExprAST *result = nullptr;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (int op = CurPos->getKind()) {
    case tok::Plus:
    case tok::Minus:
```

```
    case tok::PlusPlus:
    case tok::MinusMinus:
    case tok::Not:
    case tok::Tilda:
      // Пропускаем оператор и считываем следующий выражение с одним
      ++CurPos;
      result = parseUnaryExpr();
      // Строим ветку дерева для выражения с одним операндом на основ
      // имеющейся информации
      return new UnaryExprAST(loc, op, result);

    default:
      // Нет никаких унарных операторов, считываем постфиксное выраже
      return parsePostfixExpr();
  }
}
```

Следующими идут выражения с двумя аргументами. Тут уже все становится интересным. Есть два варианта реализации, для каждой группы операторов вводить отдельную функцию, которая делает разбор и анализ данного типа операций (что увеличивает глубину рекурсии и количество схожего кода, т. к. все эти операции очень похожи друг на друга), либо использовать вариант с разборам на основе приоритета операций и их ассоциативности. Мы будем использовать вариант с приоритетом операций. Для начала опишем список всех доступных нам приоритетов (они расположены в порядке возрастания приоритета операции, приоритет операций в нашем языке схож с тем, который есть в языке C++ и некоторых других):

```
enum OpPrecedenceLevel {
  OPL_Unknown = 0,         ///< Не является оператором
  OPL_Comma = 1,           ///< ,
  OPL_Assignment = 2,      ///< =
  OPL_Conditional = 3,     ///< ?
```

```
  OPL_LogicalOr = 4,      ///< ||
  OPL_LogicalAnd = 5,     ///< &&
  OPL_InclusiveOr = 6,    ///< |
  OPL_ExclusiveOr = 7,    ///< ^
  OPL_And = 8,            ///< &
  OPL_Equality = 9,       ///< ==, !=
  OPL_Relational = 10,    ///<  >=, <=, >, <
  OPL_Shift = 11,         ///< <<, >>
  OPL_Additive = 12,      ///< -, +
  OPL_Multiplicative = 13 ///< *, /, %
};
```

Теперь, когда у нас есть список всех приоритетов, мы можем описать функцию, которая на основе типа токена выдает нам приоритет данного оператора:

```
OpPrecedenceLevel getBinOpPrecedence(tok::TokenKind op) {
  switch (op) {
    case tok::Greater:
    case tok::Less:
    case tok::GreaterEqual:
    case tok::LessEqual:
      return OPL_Relational;

    case tok::LShift:
    case tok::RShift:
      return OPL_Shift;

    case tok::Comma:
      return OPL_Comma;

    case tok::Assign:
      return OPL_Assignment;

    case tok::Question:
```

```
      return OPL_Conditional;

    case tok::LogOr:
      return OPL_LogicalOr;

    case tok::LogAnd:
      return OPL_LogicalAnd;

    case tok::BitOr:
      return OPL_InclusiveOr;

    case tok::BitXor:
      return OPL_ExclusiveOr;

    case tok::BitAnd:
      return OPL_And;

    case tok::Equal:
    case tok::NotEqual:
      return OPL_Equality;

    case tok::Plus:
    case tok::Minus:
      return OPL_Additive;

    case tok::Mul:
    case tok::Div:
    case tok::Mod:
      return OPL_Multiplicative;

    default:
      return OPL_Unknown;
  }
}
```

Теперь рассмотрим реализацию самого разбора выражений с двумя операндами:

```
/// expr
///   ::= assign-expr
///   ::= expr ',' assign-expr
/// assign-expr
///   ::= cond-expr
///   ::= cond-expr '=' assign-expr
/// cond-expr
///   ::= assoc-expr
///   ::= assoc-expr '?' expr ':' cond-expr
/// op
///   ::= '||' | '&&' | '|' | '^' | '&' | '==' | '!=' | '>=' | '<=' |
///   ::= '<' | '<<' | '>>' | '-' | '+' | '*' | '/' | '%'
/// assoc-expr
///   ::= unary-expr
///   ::= assoc-expr op unary-expr
ExprAST *Parser::parseRHS(ExprAST *lhs, int maxPrec) {
  OpPrecedenceLevel newPrec = getBinOpPrecedence(CurPos->getKind());

  for (;;) {
    tok::TokenKind tok = CurPos->getKind();
    llvm::SMLoc loc = CurPos.getLocation();
    // Если у текущего оператора приоритет ниже того, максимального п
    // для данного выражения, то мы возвращаем выражение ранее получе
    // процессе разбора
    if (newPrec < maxPrec) {
      return lhs;
    }
   // Пропускаем считанный оператор
    ++CurPos;

    // Проверяем на то, что это ? : т. к. у нас для этого оператора е
    // специальная обработка
    if (newPrec == OPL_Conditional) {
```

```
    ExprAST *thenPart = nullptr;

    if (CurPos == tok::Colon) {
      // Если это ":", то возвращаем ошибку, т. к. у нас отсутствуе
      // выражение для варианта, когда условие истинно
      getDiagnostics().report(CurPos.getLocation(),
                              diag::ERR_ExpectedExpressionAfterQues
    } else {
      // Считываем выражение для варианта, когда условие истинно
      thenPart = parseExpr();
    }
   // Проверяем наличие ":" после выражения
    check(tok::Colon);
   // Считываем выражение после ":" и создаем дерево для оператора
    ExprAST *elsePart = parseAssignExpr();
    lhs = new CondExprAST(loc, lhs, thenPart, elsePart);
   // Получаем приоритет следующего токена и переходим к следующей
    newPrec = getBinOpPrecedence(CurPos->getKind());
    continue;
  }
  // Разбор выражения для правого операнда
  ExprAST *rhs = parseUnaryExpr();
  // Получение приоритета следующего токена и проверка на то, что э
  // оператор имеет правую ассоциативность
  OpPrecedenceLevel thisPrec = newPrec;
  newPrec = getBinOpPrecedence(CurPos->getKind());
  bool isRightAssoc = (thisPrec == OPL_Assignment);
  // Проверяем, что приоритет текущего оператора меньше приоритета
  // оператора, а так же если их приоритеты совпадают, то оператор
  // иметь правую ассоциативность
  if (thisPrec < newPrec || (thisPrec == newPrec && isRightAssoc))
    if (isRightAssoc) {
      // Для операторов с правой ассоциативностью перед создания де
      // текущего выражения, мы должны сперва рекурсивно вызвать и
      // дерево для правой части выражения с таким же приоритетом
      rhs = parseRHS(rhs, thisPrec);
    } else {
```

```
        // Для операторов с левой ассоциативностью мы должны сначала
        // дерево для правой части выражения, но с более высоким прио
        rhs = parseRHS(rhs, (OpPrecedenceLevel)(thisPrec + 1));
      }
      // Получаем приоритет текущего токена, для последующего разбора
      newPrec = getBinOpPrecedence(CurPos->getKind());
    }
    // Создание дерева для текущего выражения
    lhs = new BinaryExprAST(loc, tok, lhs, rhs);
  }
}
```

И в заключении для выражений у нас остались 2 функции, одна для выражений, в которых не может встречаться "," (например для обработки аргументов для вызова функции и некоторых других мест) и другая для случаев, где они допустимы:

```
ExprAST *Parser::parseAssignExpr() {
  ExprAST *lhs = parseUnaryExpr();
  return parseRHS(lhs, OPL_Assignment);
}

ExprAST *Parser::parseExpr() {
  ExprAST *lhs = parseUnaryExpr();
  return parseRHS(lhs, OPL_Comma);
}
```

Для парсинга типа переменной, параметра функции или типа возвращаемого значения будет использоваться слудующий метод:

```
/// type ::= 'int' | 'float' | 'void'
TypeAST *Parser::parseType() {
  TypeAST *type = nullptr;
```

```
  bool isVoid = false;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (CurPos->getKind()) {
    case tok::Int:
      ++CurPos;
      type = BuiltinTypeAST::get(TypeAST::TI_Int);
      break;

    case tok::Float:
      ++CurPos;
      type = BuiltinTypeAST::get(TypeAST::TI_Float);
      break;

    case tok::Void:
      ++CurPos;
      type = BuiltinTypeAST::get(TypeAST::TI_Void);
      isVoid = true;
      break;

    default:
      getDiagnostics().report(CurPos.getLocation(), diag::ERR_Invalid
      return nullptr;
  }

  if (type == BuiltinTypeAST::get(TypeAST::TI_Void)) {
    // Запрещаем использовать "void" в качестве типа, без дополнитель
    // модификаторов
    getDiagnostics().report(loc, diag::ERR_VoidAsNonPointer);
    return nullptr;
  }

  return type;
}
```

Для разбора прототипа функции мы будем использовать следующую функцию (она будет применяться в нескольких местах):

```
/// parameter ::= identifier ':' type
/// parameters-list
///   ::= parameter
///   ::= parameter-list ',' parameter
/// return-type ::= ':' type
/// func-proto ::= identifier '(' parameters-list ? ')' return-type ?
SymbolAST *Parser::parseFuncProto() {
  Name *name = nullptr;
  TypeAST *returnType = BuiltinTypeAST::get(TypeAST::TI_Void);
  ParameterList params;
  int Tok = CurPos->getKind();
  llvm::SMLoc loc = CurPos.getLocation();
  // При вызове этой функции текущий токен всегда "fn" (в дальнейшем
  // список будет расширен)
  ++CurPos;

  // После ключевого слова объявления функции всегда должен быть иден
  if (CurPos == tok::Identifier) {
    // Сохраняем имя функции для ее прототипа и переходим к следующем
    name = CurPos->getIdentifier();
    ++CurPos;
  } else {
    check(tok::Identifier);
  }
  // Проверяем наличие "("
  check(tok::OpenParen);

  //  Проверяем, что у прототипа есть параметры
  if (CurPos != tok::CloseParen) {
    // Производим анализ всех параметров функции
    for (;;) {
      Name *paramName = nullptr;
```

```
    if (CurPos == tok::Identifier) {
      // Параметр всегда должен начинаться с имени параметра. Сохра
      // его и переходим к следующему токену
      paramName = CurPos->getIdentifier();
      ++CurPos;

      // Особая обработка для "_" - анонимный параметр
      if (strcmp(paramName->Id, "_") == 0) {
        paramName = nullptr;
      }
    } else {
      // Сообщаем об ошибке
      check(tok::Identifier);
    }
    // После имени параметра всегда должно быть ":"
    check(tok::Colon);
    // Получаем тип для параметра функции
    TypeAST *type = parseType();
    // Добавляем новый параметр для прототипа на основе полученной
    // информации
    params.push_back(new ParameterAST(type, paramName));

    if (CurPos != tok::Comma) {
      // Если это не "," то заканчиваем обработку параметров
      break;
    }
    // Считываем "," и переходим к следующему токену
    ++CurPos;
  }
}
// Проверяем на наличие ")"
check(tok::CloseParen);
// Проверяем на наличие ":" и считываем тип возвращаемого значения
// Примечание: мы не допускаем ":" "void"
if (CurPos == tok::Colon) {
  ++CurPos;
  returnType = parseType();
```

```
  }
  // Строим элемент дерева для прототипа функции
  returnType = new FuncTypeAST(returnType, params);
  return new FuncDeclAST(loc, returnType, name, nullptr, Tok);
}
```

Для разбора объявления функции будет использована следующая функция:

```
/// func-decl ::= 'fn' func-proto block-stmt
SymbolList Parser::parseFuncDecl() {
  // Чтение прототипа функции
  SymbolList result;
  FuncDeclAST *decl = (FuncDeclAST *)parseFuncProto();

  if (CurPos != tok::BlockStart) {
    // Отсутствие "{" является ошибкой
    getDiagnostics().report(CurPos.getLocation(), diag::ERR_ExpectedF
  } else {
    // Считываем тело функции
    StmtAST *body = parseStmt();
    // Добавляем тело к прототипу функции
    decl->Body = body;
    // Добавляем созданное объявление функции к результирующим объявл
    result.push_back(decl);
  }

  return result;
}
```

Для разбора объявлений используется следующая функция (пока она может понимать только функции, но позже мы добавим классы и структуры):

```
/// decls
///  ::= func-decl
///  ::= decls func-decl
SymbolList Parser::parseDecls() {
  SymbolList result;
 // Производим анализ всех доступных объявлений
  for (;;) {
    SymbolList tmp;
    llvm::SMLoc loc = CurPos.getLocation();

    switch (int Tok = CurPos->getKind()) {
      case tok::Def:
        // Объявление функции. Считываем объявление функции и добавля
        // список объявлений
        tmp = parseFuncDecl();
        result.push_back(tmp.pop_back_val());
        continue;

      case tok::EndOfFile:
        break;

      default:
        break;
    }

    break;
  }

  return result;
}
```

Разбор модуля со всеми верхнеуровневыми объявлениями:

```
/// module-decl ::= decls
ModuleDeclAST *Parser::parseModule() {
  // Вызываем функцию разбора объявлений
  SymbolList decls = parseDecls();
  // Любой модуль должен заканчиваться символом конца файла (значит в
  // не разобранных конструкций
  if (CurPos != tok::EndOfFile) {
    getDiagnostics().report(CurPos.getLocation(),
                            diag::ERR_ExpectedEndOfFile);
    return nullptr;
  }
  // Конструирование дерева для модуля
  return new ModuleDeclAST(getDiagnostics(), decls);
}
```

Разбор объявлений для локальных переменных и блока инициализации в цикле "for":

```
/// var-init ::= '=' assign-expr
/// var-decl ::= identifier ':' type var-init?
/// var-decls
///   ::= var-decl
///   ::= var-decls ',' var-decl
/// decl-stmt ::= var-decls ';'
SymbolList Parser::parseDecl(bool needSemicolon) {
  SymbolList result;
  // Производим анализ списка объявлений переменных
  for (;;) {
    llvm::SMLoc loc = CurPos.getLocation();

    if (CurPos != tok::Identifier) {
      // Объявление всегда должно начинаться с идентификатора, если э
      // другое, то сообщаем об ошибке
      getDiagnostics().report(CurPos.getLocation(),
```

```
                                diag::ERR_ExpectedIdentifierInDecl);
    } else {
      // Сохраняем имя переменной
      Name *name = CurPos->getIdentifier();
      ExprAST *value = nullptr;
      // Переходим к следующему токену
      ++CurPos;
      // Проверяем на наличие ":"
      check(tok::Colon);
      // Считываем тип переменной
      TypeAST *type = parseType();

      if (CurPos == tok::Assign) {
        // Если у переменной есть инициализатор, то считываем "=" и
        // инициализирующее значение
        ++CurPos;
        value = parseAssignExpr();
      }
      // Добавляем новую переменную к списку
      result.push_back(new VarDeclAST(loc, type, name, value));

      if (CurPos != tok::Comma) {
        // Если это не ",", то список объявлений завершен
        break;
      }
      // Пропускаем ","
      ++CurPos;
    }
  }

  if (needSemicolon) {
    // Если объявление должно заканчиваться ";", то проверяем его нал
    check(tok::Semicolon);
  }
```

```
  return result;
}
```

Вспомогательная функция, которая считывает инструкцию и если это не блочный элемент (набор инструкций между "{" и "}"), то создает новый элемент дерева в виде блочного элемента:

```
StmtAST *Parser::parseStmtAsBlock() {
  // Чтение инструкции
  llvm::SMLoc loc = CurPos.getLocation();
  StmtAST *result = parseStmt();

  if (isa<BlockStmtAST>(result)) {
    // Это уже блочный элемент, возвращаем его
    return result;
  }

  // Создаем новый блочный элемент с одной инструкцией
  StmtList body;
  body.push_back(result);
  return new BlockStmtAST(loc, body);
}
```

Разбор инструкций языка:

```
/// block-stmt ::= '{' stmt* '}'
/// for-init
///   ::= 'let' decl-stmt
///   ::= expr
/// for-stmt ::= 'for' for-init? ';' expr? ';' expr? block-stmt
/// stmt
///   ::= expr? ';'
///   ::= 'let' decl-stmt
```

```
///   ::= 'if' expr block-stmt ( 'else' block-stmt )?
///   ::= 'while' expr block-stmt
///   ::= for-stmt
///   ::= 'break'
///   ::= 'continue'
///   ::= 'return' expr? ';'
///   ::= block-stmt
StmtAST *Parser::parseStmt() {
  StmtAST *result = nullptr;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (CurPos->getKind()) {
    case tok::Var:
      // Объявление локальных переменных
      ++CurPos;
      return new DeclStmtAST(loc, parseDecl(true));

    // Для упрощения анализа мы знаем с каких токенов может начинатьс
    // арифметические выражение, поэтому мы можем обработать их здесь
    case tok::Plus:
    case tok::Minus:
    case tok::PlusPlus:
    case tok::MinusMinus:
    case tok::Tilda:
    case tok::Not:
    case tok::Identifier:
    case tok::IntNumber:
    case tok::FloatNumber:
    case tok::OpenParen: {
      // Чтение выражение
      ExprAST *expr = parseExpr();
      // Инструкция с выражением должна заканчиваться ";"
      check(tok::Semicolon);
      // Создаем ветвь дерева для инструкции с арифметическим выражен
      return new ExprStmtAST(loc, expr);
    }
```

```
case tok::If: {
  // Это инструкция ветвления "if"
  check(tok::If);
  // Считываем выражение — условие
  ExprAST *expr = parseExpr();

  if (CurPos != tok::BlockStart) {
    // Ветки ветвления должны начинаться с "{", т. е. должны быть
    // элементом
    check(tok::BlockStart);
  }
  // Считываем ветку если условие истинно
  StmtAST *thenPart = parseStmtAsBlock();
  StmtAST *elsePart = nullptr;

  if (CurPos == tok::Else) {
    // Если у нас есть "else", то пропускаем его и считываем тело
    ++CurPos;

    if (CurPos != tok::BlockStart) {
      // Ветки ветвления должны начинаться с "{", т. е. должны бы
      // блочным элементом
      check(tok::BlockStart);
    }
    // Считываем ветку если условие ложно
    elsePart = parseStmtAsBlock();
  }
  // Создаем ветку дерева для инструкции ветвления "if"
  return new IfStmtAST(loc, expr, thenPart, elsePart);
}

case tok::While: {
  // Это цикл "while"
  check(tok::While);
  // Считываем выражение — условие цикла
  ExprAST *expr = parseExpr();
```

```
  if (CurPos != tok::BlockStart) {
    // Тело цикла должно быть блочным элементом
    check(tok::BlockStart);
  }
  // Считываем тело цикла и создаем ветку дерева для инструкции "
  result = parseStmtAsBlock();
  return new WhileStmtAST(loc, expr, result);
}

case tok::For: {
  // Это цикл "for"
  check(tok::For);

  ExprAST *initExpr = nullptr, *condExpr = nullptr, *postExpr = n
  SymbolList decls;

  if (CurPos != tok::Semicolon) {
    // У цикла есть блок инициализации цикла
    if (CurPos == tok::Var) {
       // Это объявление переменных цикла
      ++CurPos;
      decls = parseDecl(true);
    } else {
      // Это обычное выражение
      initExpr = parseExpr();
      check(tok::Semicolon);
    }
  } else {
    // Блок инициализации всегда должен заканчиваться ";"
    check(tok::Semicolon);
  }

  if (CurPos != tok::Semicolon) {
    // У нас есть условие цикла, считываем его
    condExpr = parseExpr();
  }
  // После условия цикла всегда должна быть ";"
```

```
  check(tok::Semicolon);

  if (CurPos != tok::BlockStart) {
    // У нас есть выражение, которое должно быть выполнено после
    // итерации
    postExpr = parseExpr();
  }

  if (CurPos != tok::BlockStart) {
    // Тело цикла всегда должно быть блочным элементом
    check(tok::CloseParen);
  }
  // Чтение тела цикла и конструирование ветви дерева для инструк
  // "for"
  result = parseStmtAsBlock();
  return new ForStmtAST(loc, initExpr, decls, condExpr, postExpr,
                        result);
}

case tok::Break:
  // Считываем "break" ";" и создаем ветку дерева для инструкции
  // досрочного выхода из цикла "break"
  check(tok::Break);
  check(tok::Semicolon);
  return new BreakStmtAST(loc);

case tok::Continue:
  // Считываем "continue" ";" и создаем ветку дерева для инструкц
  // перехода к следующей итерации цикла "continue"
  check(tok::Continue);
  check(tok::Semicolon);
  return new ContinueStmtAST(loc);

case tok::Return: {
  // Это инструкция возврата из функции
  check(tok::Return);
```

```
  if (CurPos == tok::Semicolon) {
    // Если это ";", то у нас нет выражение, возможно это выход и
    // функции с типом возвращаемого значения "void". Считываем "
    // строим ветвь дерева для "return"
    ++CurPos;
    return new ReturnStmtAST(loc, nullptr);
  }
  // Считываем выражение для возврата
  ExprAST *expr = parseExpr();
  // После выражения всегда должен быть ";". Считываем и строим в
  // дерева для "return"
  check(tok::Semicolon);
  return new ReturnStmtAST(loc, expr);
}

case tok::BlockStart: {
   // Это блочный элемент
  StmtList stmts;
  // Считываем "{"
  ++CurPos;
  // И считываем все инструкции, пока мы не встретим "}" или коне
  while (CurPos != tok::BlockEnd && CurPos != tok::EndOfFile) {
    result = parseStmt();
    stmts.push_back(result);
  }
  // Проверяем на наличие "}" и создаем ветвь дерева для блочного
  // элемента
  check(tok::BlockEnd);
  return new BlockStmtAST(loc, stmts);
}

case tok::Semicolon:
  // Это пустое выражение, просто ";". Строим дерево для пустого
  // выражения
  ++CurPos;
  return new ExprStmtAST(loc, nullptr);
```

```
    default:
      // Все остальные случаи являются ошибкой
      getDiagnostics().report(CurPos.getLocation(),
                              diag::ERR_InvalidStatement);
      return nullptr;
  }
}
```

Для упрощения объявления прототипов функций для внутреннего использования (для подключения функций из языка C++) будем использовать следующую функцию:

```
SymbolAST *parseFuncProto(llvm::StringRef Proto) {
  llvm::SourceMgr SrcMgr;
  DiagnosticsEngine Diags(SrcMgr);
  // Создаем буфер для лексического анализа и инициализируем его пере
  // строкой
  std::unique_ptr<llvm::MemoryBuffer> Buff =
    llvm::MemoryBuffer::getMemBuffer(Proto);
  // Добавляем файл для чтения
  SrcMgr.AddNewSourceBuffer(std::move(Buff), llvm::SMLoc());

  Lexer Lex(SrcMgr, Diags);
  Parser P(&Lex);
  // Парсим прототип функции
  return P.parseFuncProto();
}
```

Для разбора файла используется следующая функция:

```
ModuleDeclAST* ModuleDeclAST::load(SourceMgr &SrcMgr,
                                   DiagnosticsEngine &Diags,
```

```
                                    StringRef fileName) {
  llvm::ErrorOr<std::unique_ptr<llvm::MemoryBuffer>>
    FileOrErr = llvm::MemoryBuffer::getFile(fileName);

  if (std::error_code BufferError = FileOrErr.getError()) {
    llvm::WithColor::error(llvm::errs(), "simple")
      << "Error reading " << fileName << ": "
      << BufferError.message() << "\n";
  }

  // Добавляем файл для чтения
  SrcMgr.AddNewSourceBuffer(std::move(*FileOrErr),
                            llvm::SMLoc());

  Lexer Lex(SrcMgr, Diags);
  Parser P(&Lex);

  // Парсим содержимое модуля
  return P.parseModule();
}
```

Заключение

Статья получилась объемной, но надеюсь тем людям, которые прочитали ее до конца, она оказалась полезной и даст стимул, в написании своих языков программирования или даст толчок для присоединения к сообществам разработчиков уже имеющихся.

Полный исходный код доступен в репозитории github (в коде присутствует полная версия программы включая семантический анализ и кодогенерацию для данного подмножества, код содержит комментарии (за исключением тех, что были добавлены в статье), а так же содержит много документирующих комментариях для Doxygen). Позже будет загружена версия и для полной реализации языка, поэтому нетерпеливые смогут покопаться в них, а остальным до встречи в продолжении серии.

Полезные ссылки

1. Compilers: Principles, Techniques, and Tools by Alfred V. Aho (Author), Ravi Sethi (Author), Jeffrey D. Ullman (Author)
2. Let's Build a Compiler, by Jack Crenshaw
3. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries By Min-Yih Hsu
4. Kaleidoscope: Compiling to Object Code
5. https://llvm.godbolt.org/
6. How to set up LLVM-style RTTI for your class hierarchy

Теги: llvm, компиляторы, c++

Хабы: Open source, Программирование, Компиляторы

Создаем свой собственный язык программирования с использованием LLVM. Часть 2: Семантический анализ

7.8K

Open source*, Программирование*, Компиляторы*

В предыдущей статье мы закончили на том, что мы написали лексический и синтаксический анализаторы для нашего учебного языка. В данной статье мы продолжим начатое и рассмотрим следующую стадию анализа исходного кода программы — семантический анализ.

Оглавление серии

Семантический анализ

Основная задача семантического анализа заключается в проверки того, что программа корректна с точки зрения языка, например:

1. Все переменные в программе объявлены;
2. Все выражения совершаются над корректными типами;
3. Если в программе используется безусловный/условный переход, то метка, на которую совершается переход должна существовать;
4. Функция возвращает значение корректного типа.

Замечание. Нужно понимать, что семантический анализ не может выловить все ошибки в программе, а только те их типы, которые были заложены разработчиком компилятора или описаны в спецификации языка. Например в языке C++ компилятор не сможет обнаружить все ошибки работы с памятью, даже если разработчик компилятора внес специальную обработку для выявления ошибок такого типа. Но в Rust ошибки такого типа исключаются на уровне самого языка.

Все семантические правила обычно описываются в спецификации конкретного языка и описывают критерии нахождения не корректных программ на данном языке.

Область видимости (Scope)

Область видимости — это часть программы, в пределах которой идентификатор, объявленный как имя некоторой программной сущности (обычно — переменной, типа данных или функции), остаётся связанным с этой сущностью, то есть позволяет посредством себя обратиться к ней.

Часто в спецификации языка описаны правила, для формирования областей видимости, например может быть глобальная область видимости, область видимости класса, область видимости функции и др. Обычно области видимости образуют иерархию и для поиска сущности, на которую ссылается идентификатор в конкретном месте программы, может понадобиться поиск этого идентификатора во всех уровнях этой иерархии.

В simple есть следующие виды областей видимости:

1. Модуль (все объявления объявленные на самом верхнем уровне файла);
2. Класс/Структура (более подробно будут рассмотрены в последующих частях серии);
3. Функция;

4. Блок в функции.

Так же в simple объявления не могут перекрывать другие объявление доступные в данной области видимости (за исключением перегрузки функций (рассмотри в последующих частях серии)). В simple, любые переменная объявленная в функции может быть использована в любом месте после ее объявления, любое имя объявленное в модуле может быть использовано в любом месте в этом модуле (т. е. разрешено использовать имя класса до его объявления, если оно находится в этом же модуле).

Более подробно про области видимости можно почитать в [1].

Для работы с областью видимости будем использовать следующий класс:

```
struct Scope {
  Scope(DiagnosticsEngine &D) ;
  Scope(Scope* enclosed) ;
  Scope(ModuleDeclAST* thisModule) ;
 // Поиск идентификатора во всех доступных в данной точке областях ви
  SymbolAST* find(Name* id);
  // Поиск идентификатора – члена класса или структуры
  SymbolAST* findMember(Name* id, int flags = 0);
  // Создать новую область видимости
  Scope* push();
  // Создать новую область видимости на основе объявления
  Scope* push(ScopeSymbol* sym);
  // Закрыть текущую область видимости (возвращает родительский Scope
  Scope* pop();
 // Воссоздать список областей видимости для иерархии классов
  static Scope* recreateScope(Scope* scope, SymbolAST* sym);
  // Очистка списка областей видимости и оставить только область види
  // самого модуля
  static void clearAllButModule(Scope* scope);
 // Вывод сообщения об ошибке
```

```
  template <typename... Args>
  void report(SMLoc Loc, unsigned DiagID, Args &&...Arguments) {
    Diag.report(Loc, DiagID, std::forward<Args>(Arguments)...);
  }

  Scope* Enclosed; ///< родительский Scope
  ModuleDeclAST* ThisModule; ///< родительский модуль
  /// символ для текущей области видимости (например функция или клас
  SymbolAST* CurScope;
  /// функция к которой принадлежит область видимости
  FuncDeclAST* EnclosedFunc;
  StmtAST* BreakLoc; ///< цикл для инструкции break
  StmtAST* ContinueLoc; ///< цикл для инструкции continue
  LandingPadAST* LandingPad; ///< нужно для генерации кода (см. описа
  DiagnosticsEngine &Diag; ///< модуль диагностики
};
```

Ниже приведу реализацию данного класса:

▾ Hidden text

```
SymbolAST* Scope::find(Name* id) {
  Scope* s = this;

  // Если идентификатор не указан, то это глобальная область
  // видимости
  if (!id) {
    return ThisModule;
  }

  // Циклический поиск объявления во всех доступных из данной
  // областях видимости
  for ( ; s; s = s->Enclosed) {
```

```
    if (s->CurScope) {
      SymbolAST* sym = s->CurScope->find(id);

      if (sym) {
        return sym;
      }
    }
  }

  return nullptr;
}

SymbolAST* Scope::findMember(Name* id, int flags) {
  if (CurScope) {
    return CurScope->find(id, flags);
  }

  return nullptr;
}

Scope* Scope::push() {
  Scope* s = new Scope(this);
  return s;
}

Scope* Scope::push(ScopeSymbol* sym) {
  Scope* s = push();
  s->CurScope = sym;
  return s;
}

Scope* Scope::pop() {
  Scope* res = Enclosed;
  delete this;
  return res;
}
```

```
Scope* Scope::recreateScope(Scope* scope, SymbolAST* sym) {
  Scope* p = scope;

  // Поиск области видимости самого верхнего уровня (модуля)
  while (p->Enclosed) {
    p = p->Enclosed;
  }

  // Создаем список все родительских объявлений
  SymbolList symList;
  SymbolAST* tmp = sym;

  while (tmp->Parent) {
    symList.push_back(tmp);
    tmp = tmp->Parent;
  }

  // Воссоздаем все области видимости в обратном порядке объяв
  // сущностей. Нужно для поиска в имени в иерархии классов
  for (SymbolList::reverse_iterator it = symList.rbegin(),
       end = symList.rend(); it != end; ++it) {
    p = p->push((ScopeSymbol*)*it);
  }

  // Возвращаем созданный Scope
  return p;
}

void Scope::clearAllButModule(Scope* scope) {
  Scope* p = scope;

  while (p->Enclosed) {
    p = p->pop();
  }
}
```

Проверка типов

Суть проверки типов заключается в том, что бы проверить, что все операции выполняемые в программе производятся с операндами корректных типов.

Проверка типов бывает:

1. Статическая (данная проверка происходит на этапе компиляции программы);
2. Динамическая (данная проверка происходит на этапе выполнения программы).

Более подробно можно почитать в [2].

В simple у нас будет статическая проверка и все типы будут определяться во время анализа исходного кода программы. В последующих частях мы добавим классы с поддержкой виртуальных функций, при вызове которых результирующая функция будет разрешаться во время выполнения программы.

Семантический анализ типов

Для начала рассмотрим какие именно функции члены иерархии типов будут отвечать за семантический анализ этих ветвей AST:

```
struct TypeAST {
  ...

  /// Производит семантический анализ для типа
  virtual TypeAST* semantic(Scope* scope) = 0;
  /// Проверка на то, что данный тип может быть преобразован в "newTy
  virtual bool implicitConvertTo(TypeAST* newType) = 0;
  /// Проверка на то, что данный тип совпадает с "otherType"
```

```
  bool equal(TypeAST* otherType);
  /// Сгенерировать и поместить в буфер декорированную строку для
  /// данного типа (реализуется в потомках)
  virtual void toMangleBuffer(llvm::raw_ostream& output) = 0;
  /// Сгенерировать декорированную строку для данного типа
  void calcMangle();


  llvm::StringRef MangleName; ///< декорированная строка с именем дан
};
```

Если посмотреть на приведенный выше код, то можно увидеть, что функция semantic возвращает TypeAST*, это нужно для того, что бы при необходимости можно было бы вернуть новый тип в замен старого. Например в simple можно объявить переменную с типом A и на момент построения дерева мы не можем сказать, какой именно тип будет иметь данная переменная, поэтому во время парсинга будет создан экземпляр типа QualifiedTypeAST, который во время семантического анализа будет заменен типом StructTypeAST или ClassTypeAST (все эти типы будут рассмотрены в последующих частях серии).

Так же можно увидеть, что существуют функции для создания декорированного имени, данная функция генерирует уникальное имя типа в системе и если два объекта производные от TypeAST имеют одинаковое декорированное имя, то эти типы совпадают. Так же декорация имен нужна для создания уникальных имен функций, например функция

```
fn check(s: string, a: float, b: float)
```

после декорации будет иметь следующие имя

```
_P5checkPKcff
```

Более подробнее про декорирование имен (Name mangling) можно посмотреть тут [3].

Теперь мы можем рассмотреть реализацию семантического анализа для иерархии типов:

▾ Hidden text

```
// Хранит множество всех уникальных декорированных строк
StringSet< > TypeAST::TypesTable;

void TypeAST::calcMangle() {
  // Ничего не делаем, если декорированное имя для типа уже за
  if (!MangleName.empty()) {
    return;
  }

  // Для большинства типов 128 байт будет достаточно
  llvm::SmallString< 128 > s;
  llvm::raw_svector_ostream output(s);

  // Пишем декорированное имя в буфер и добавляем в множество
  // т. к. StringRef требует выделенный блок памяти со строкой
  toMangleBuffer(output);
  TypesTable.insert(output.str());

  // Устанавливаем внутреннее состояние MangleName на основе з
  // созданной в множестве имен
  StringSet< >::iterator pos = TypesTable.find(s);
  assert(pos != TypesTable.end());
  MangleName = pos->getKeyData();
```

```
}

bool TypeAST::equal(TypeAST* otherType) {
  // Два типа эквивалентны, если их "MangleName" совпадают, дл
  // сложных типов это может быть гораздо быстрее, чем сравнив
  // их структуру
  assert(!MangleName.empty() && !otherType->MangleName.empty()
  return MangleName == otherType->MangleName;
}

TypeAST* BuiltinTypeAST::semantic(Scope* scope) {
  // Для базовых типов для проверки семантики достаточно произ
  // декорирование имени типа
  calcMangle();
  return this;
}

bool BuiltinTypeAST::implicitConvertTo(TypeAST* newType) {
  // Список разрешенных преобразований
  static bool convertResults[TI_Float + 1][TI_Float + 1] = {
    // void  bool   int    float
    { false, false, false, false }, // void
    { false, true,  true,  true  }, // bool
    { false, true,  true,  true  }, // int
    { false, true,  true,  true  }, // float
  };

  // Только базовые типы могут быть преобразованы друг в друга
  if (newType->TypeKind > TI_Float) {
    return false;
  }

  return convertResults[TypeKind][newType->TypeKind];
}

void BuiltinTypeAST::toMangleBuffer(llvm::raw_ostream& output)
  switch (TypeKind) {
```

```
    case TI_Void : output << "v"; break;
    case TI_Bool : output << "b"; break;
    case TI_Int : output << "i"; break;
    case TI_Float : output << "f"; break;
    default: assert(0 && "Should never happen"); break;
  }
}

TypeAST* FuncTypeAST::semantic(Scope* scope) {
  if (!ReturnType) {
    // Если у функции не был задан тип возвращаемого значения,
    // установить как "void"
    ReturnType = BuiltinTypeAST::get(TypeAST::TI_Void);
  }

  // Произвести семантический анализ для типа возвращаемого
  // значения
  ReturnType = ReturnType->semantic(scope);

  // Произвести семантический анализ для всех параметров
  for (ParameterList::iterator it = Params.begin(),
       end = Params.end(); it != end; ++it) {
    (*it)->Param = (*it)->Param->semantic(scope);
  }

  calcMangle();
  return this;
}

bool FuncTypeAST::implicitConvertTo(TypeAST* newType) {
  return false;
}

void FuncTypeAST::toMangleBuffer(llvm::raw_ostream& output) {
  // Добавляем "v", если функция возвращает "void"
  if (Params.empty()) {
    output << "v";
```

```
      return;
    }

    ParameterList::iterator it = Params.begin(), end = Params.er

    // Произвести декорацию имен для всех параметров
    for ( ; it != end; ++it) {
      (*it)->Param->toMangleBuffer(output);
    }
  }
```

На данной стадии у нас есть только несколько базовых типов и семантический анализ для иерархии типов достаточно прост, мы просто генерируем декорированные имена для каждого типа. Но в последующих статьях с добавлением более сложных типов семантический анализ усложнится и список преобразований между типами будет так же расширен.

Семантический анализ выражений

Для начала рассмотрим какие именно функции члены иерархии выражений будут отвечать за семантический анализ этих ветвей AST:

```
struct ExprAST {
  ...
  /// Проверка, что это целочисленная константа
  bool isIntConst() { return ExprKind == EI_Int; }
  /// Проверка, что это константное выражение
  bool isConst() {
    return ExprKind == EI_Int || ExprKind == EI_Float;
  }
  /// Проверяем, что константное выражение имеет истинное значение
```

```
  virtual bool isTrue();
  /// Проверка на то, что данное выражение может быть использовано
  /// в качестве значения с левой стороны от "="
  virtual bool isLValue();
  /// Произвести семантический анализ выражения
  virtual ExprAST *semantic(Scope *scope) = 0;
  /// Сделать копию ветви дерева
  virtual ExprAST *clone() = 0;

  TypeAST *ExprType;  ///< тип выражения
};
```

Так же, как и у типов, ExprAST после семантического анализа может быть изменено (например можно сделать оптимизацию и произвести вычисление константных выражений, но в нашем случае мы будем использовать это для замены некоторых выражений их аналогами, что бы упростить генерацию кода).

Рассмотрим более подробно сам семантический анализ для выражений:

▾ Hidden text

```
bool ExprAST::isTrue() {
  return false;
}

bool ExprAST::isLValue() {
  return false;
}

bool IntExprAST::isTrue() {
  return Val != 0;
}
```

```
ExprAST* IntExprAST::semantic(Scope* ) {
  return this;
}

ExprAST* IntExprAST::clone() {
  return new IntExprAST(Loc, Val);
}

bool FloatExprAST::isTrue() {
  return Val != 0.0;
}

ExprAST* FloatExprAST::semantic(Scope* ) {
  return this;
}

ExprAST* FloatExprAST::clone() {
  return new FloatExprAST(Loc, Val);
}

bool IdExprAST::isLValue() {
  return true;
}

ExprAST* IdExprAST::semantic(Scope* scope) {
  // Если "ThisSym" задан, то семантический анализ над данным
  // выражением уже был ранее завершен
  if (!ThisSym) {
    // Ищем объявление в текущей области видимости
    ThisSym = scope->find(Val);

    if (!Val) {
      return this;
    }

    if (!ThisSym) {
      // Объявление не найдено, возвращаем ошибку
```

```
      scope->report(Loc, diag::ERR_SemaUndefinedIdentifier,
                    Val->Id);
      return nullptr;
    }
    // Устанавливаем тип данного выражения в соответствии с ти
    // объявленной переменной
    ExprType = ThisSym->getType();
  }

  return this;
}

ExprAST* IdExprAST::clone() {
  return new IdExprAST(Loc, Val);
}

ExprAST* CastExprAST::semantic(Scope* scope) {
  if (SemaDone) {
    return this;
  }

  // Проверяем, что тип был корректно задан и что это не "void
  assert(ExprType != nullptr && "Type for cast not set");
  if (ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaCastToVoid);
    return nullptr;
  }

  // Производим семантический анализ выражения для преобразова
  Val = Val->semantic(scope);

  // Проверяем, что тип исходного выражения корректный
  if (!Val->ExprType || Val->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaCastToVoid);
    return nullptr;
  }
```

```
  // Запрещаем преобразования функций
  if (isa<FuncTypeAST>(ExprType)
      || isa<FuncTypeAST>(Val->ExprType)) {
    scope->report(Loc, diag::ERR_SemaFunctionInCast);
    return nullptr;
  }

  // Проверяем, что типы совместимы
  if (!Val->ExprType->implicitConvertTo(ExprType)) {
    scope->report(Loc, diag::ERR_SemaInvalidCast);
    return nullptr;
  }

  SemaDone = true;
  return this;
}

ExprAST* CastExprAST::clone() {
  return new CastExprAST(Loc, Val->clone(), ExprType);
}
```

Т.к. все выражения с одним операндом могут быть переписаны в виде эквивалентного выражения с двумя операндами, то для упрощения кодогенерации при семантическом анализе мы проверяем корректность операнда и возвращаем эквивалентный аналог:

- Hidden text

```
ExprAST* UnaryExprAST::semantic(Scope* scope) {
  // Проверяем, что операнд был корректно задан
  if (!Val) {
```

```
  assert(0 && "Invalid expression value");
  return nullptr;
}

// Производим семантический анализ для операнда
Val = Val->semantic(scope);

// Проверяем корректность типа операнда, т. к. он может быть
// "void"
if (!Val->ExprType || Val->ExprType->isVoid()) {
  scope->report(Loc, diag::ERR_SemaOperandIsVoid);
  return nullptr;
}

// Исключаем операции над булевыми значениями
if (Val->ExprType->isBool() && (Op == tok::Plus
                                || Op == tok::Minus)) {
  scope->report(Loc, diag::ERR_SemaInvalidBoolForUnaryOperar
  return nullptr;
}

// Список замен:
//  +Val to Val
//  -Val to 0 - Val
//  ~intVal to intVal ^ -1
//  ++id to id = id + 1
//  --id to id = id - 1
//  !Val to Val == 0

ExprAST* result = this;

// Проверяем тип оператора, для необходимой замены
switch (Op) {
  // "+" - noop
  case tok::Plus: result = Val; break;

  case tok::Minus:
```

```
    // Преобразовываем в 0 - Val с учетом типа Val
    if (Val->ExprType->isFloat()) {
      result = new BinaryExprAST(Val->Loc, tok::Minus,
                                 new FloatExprAST(Val->Loc,
                                 Val);
    } else {
      result = new BinaryExprAST(Val->Loc, tok::Minus,
                                 new IntExprAST(Val->Loc, 0)
                                 Val);
    }
    break;

  case tok::Tilda:
    // ~ можно применять только к целочисленным выражениям
    if (!Val->ExprType->isInt()) {
      scope->report(Loc,
                    diag::ERR_SemaInvalidOperandForComplemet
      return nullptr;
    } else {
      // Преобразуем в Val ^ -1
      result = new BinaryExprAST(Val->Loc, tok::BitXor, Val,
                                 new IntExprAST(Val->Loc, -1
      break;
    }

  case tok::PlusPlus:
  case tok::MinusMinus: {
      // "++" и "--" можно вызывать только для IdExprAST в
      // качестве операнда и только для целочисленного типа
      if (!Val->ExprType->isInt() || !Val->isLValue()) {
        scope->report(Loc,
                      diag::ERR_SemaInvalidPostfixPrefixOper
        return nullptr;
      }

      // Необходимо заменить "++" id или "--" id на id = id
      // или id = id + -1
```

```
        ExprAST* val = Val;
        ExprAST* valCopy = Val->clone();
        result = new BinaryExprAST(Val->Loc, tok::Assign,
          val,
          new BinaryExprAST(Val->Loc, tok::Plus,
            valCopy,
            new IntExprAST(Val->Loc,
                           (Op == tok::PlusPlus) ? 1 : -1)));
      }
      break;

    case tok::Not:
      // Заменяем на Val == 0
      result = new BinaryExprAST(Val->Loc, tok::Equal, Val,
                                 new IntExprAST(Val->Loc,
        0));
      break;

    default:
      // Никогда не должно произойти
      assert(0 && "Invalid unary expression");
      return nullptr;
  }

  if (result != this) {
    // Т.к. старое выражение было заменено, очищаем память и
    // производим семантический анализ нового выражения
    Val = nullptr;
    delete this;
    return result->semantic(scope);
  }

  return result;
}

ExprAST* UnaryExprAST::clone() {
```

```
  return new UnaryExprAST(Loc, Op, Val->clone());
}
```

Рассмотрим семантический анализ выражений с двумя операндами:

▾ Hidden text

```
ExprAST* BinaryExprAST::semantic(Scope* scope) {
  // Семантический анализ уже был произведен ранее
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ операнда с левой стороны
  LeftExpr = LeftExpr->semantic(scope);

  // Проверяем на "++" или "--", т. к. эти операции имеют толь
  // один операнд
  if (Op == tok::PlusPlus || Op == tok::MinusMinus) {
    // "++" и "--" можно вызывать только для IdExprAST в качес
    // операнда и только для целочисленного типа
    if (!LeftExpr->isLValue() || !LeftExpr->ExprType->isInt())
      scope->report(Loc,
                    diag::ERR_SemaInvalidPostfixPrefixOperand)
      return nullptr;
    }

    // Устанавливаем результирующий тип выражения
    ExprType = LeftExpr->ExprType;
    return this;
  }
```

```
// Производим семантический анализ операнда с правой сторонь
RightExpr = RightExpr->semantic(scope);

// Проверяем, что оба операнда имеют корректные типы
if (!LeftExpr->ExprType || !RightExpr->ExprType) {
  scope->report(Loc,
                diag::ERR_SemaUntypedBinaryExpressionOperanc
  return nullptr;
}

// "," имеет специальную обработку и тип выражения совпадает
// типом операнда с правой стороны
if (Op == tok::Comma) {
  ExprType = RightExpr->ExprType;
  return this;
}

// Исключаем операции, если хотя бы один операнд имеет тип "
if (LeftExpr->ExprType->isVoid()
    || RightExpr->ExprType->isVoid()) {
  scope->report(Loc, diag::ERR_SemaOperandIsVoid);
  return nullptr;
}

// Проверка на операторы сравнения, т. к. их результат всегд
// будет иметь тип "bool"
switch (Op) {
  case tok::Less:
  case tok::Greater:
  case tok::LessEqual:
  case tok::GreaterEqual:
  case tok::Equal:
  case tok::NotEqual:
    // Если левый операнд "bool", то сначала конвертируем ег
    // "int"
    if (LeftExpr->ExprType->isBool()) {
```

```
      LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
        BuiltinTypeAST::get(TypeAST::TI_Int));
      LeftExpr = LeftExpr->semantic(scope);
    }

    // Операнды для "<", "<=", ">", ">=", "==" и "!=" всегда
    // должны иметь одинаковые типы, если они отличаются, то
    // нужно сделать преобразование
    if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
      RightExpr = new CastExprAST(RightExpr->Loc, RightExpr,
                                  LeftExpr->ExprType);
      RightExpr = RightExpr->semantic(scope);
    }

    // Результирующий тип выражения — "bool"
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Bool);
    return this;

  case tok::LogOr:
  case tok::LogAnd:
    // Для логических операций оба операнда должны
    // конвертироваться в "bool"
    if (!LeftExpr->ExprType->implicitConvertTo(
        BuiltinTypeAST::get(TypeAST::TI_Bool))
      || !RightExpr->ExprType->implicitConvertTo(
        BuiltinTypeAST::get(TypeAST::TI_Bool))) {
      scope->report(Loc, diag::ERR_SemaCantConvertToBoolean)
      return nullptr;
    }

    // Результирующий тип выражения — "bool"
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Bool);
    return this;

  default:
    // Остальные варианты обрабатываем ниже
    break;
```

```
}

// Если левый операнд "bool", то сначала конвертируем его в
if (LeftExpr->ExprType == BuiltinTypeAST::get(TypeAST::TI_Bo
  LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
    BuiltinTypeAST::get(TypeAST::TI_Int));
  LeftExpr = LeftExpr->semantic(scope);
}

// Результирующий тип выражения будет совпадать с типом левс
// операнда
ExprType = LeftExpr->ExprType;

// Для "=" тоже есть специальная обработка
if (Op == tok::Assign) {
  // Если типы левого и правого операнда отличаются, то нужн
  // сделать преобразование
  if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
    RightExpr = new CastExprAST(RightExpr->Loc, RightExpr,
                                LeftExpr->ExprType);
    RightExpr = RightExpr->semantic(scope);
  }

  // Проверяем, что операнд с левой стороны является адресом
  if (!LeftExpr->isLValue()) {
    scope->report(Loc, diag::ERR_SemaMissingLValueInAssignme
    return nullptr;
  }

  // Выражение корректно, завершаем анализ
  return this;
}

// Если операнды имеют различные типы, то нужно произвести
// преобразования
if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
  // Если операнд с правой стороны имеет тип "float", то
```

```
    // результат операции тоже будет "float"
    if (RightExpr->ExprType->isFloat()) {
      ExprType = RightExpr->ExprType;
      LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
                                 RightExpr->ExprType);
      LeftExpr = LeftExpr->semantic(scope);
    } else {
      // Преобразуем операнд с правой стороны к типу операнда
      // левой стороны
      RightExpr = new CastExprAST(RightExpr->Loc, RightExpr,
                                  LeftExpr->ExprType);
      RightExpr = RightExpr->semantic(scope);
    }
  }

  // "int" и "float" имеют отличный набор допустимых бинарных
  // операций
  if (ExprType == BuiltinTypeAST::get(TypeAST::TI_Int)) {
    // Проверяем допустимые операции над "int"
    switch (Op) {
      case tok::Plus:
      case tok::Minus:
      case tok::Mul:
      case tok::Div:
      case tok::Mod:
      case tok::BitOr:
      case tok::BitAnd:
      case tok::BitXor:
      case tok::LShift:
      case tok::RShift:
        return this;

      default:
        // Никогда не должны сюда попасть, если только нет оши
        // в парсере
        assert(0 && "Invalid integral binary operator");
        return nullptr;
```

```
    }
  } else {
    // Проверяем допустимые операции над "float"
    switch (Op) {
      case tok::Plus:
      case tok::Minus:
      case tok::Mul:
      case tok::Div:
      case tok::Mod:
      case tok::Less:
        return this;

      default:
        // Сообщаем об ошибке, т. к. данная операция не допуст
        scope->report(Loc,
          diag::ERR_SemaInvalidBinaryExpressionForFloatingPoir
        return nullptr;
    }
  }
}

ExprAST* BinaryExprAST::clone() {
  return new BinaryExprAST(Loc, Op, LeftExpr->clone(),
    RightExpr ? RightExpr->clone() : nullptr);
}
```

Семантический анализ тернарного оператор:

- Hidden text

```
bool CondExprAST::isLValue() {
  return IfExpr->isLValue() && ElseExpr->isLValue();
}

ExprAST* CondExprAST::semantic(Scope* scope) {
  if (SemaDone) {
    return this;
  }

  // Производим семантический анализ условия и всех операндов
  Cond = Cond->semantic(scope);
  IfExpr = IfExpr->semantic(scope);
  ElseExpr = ElseExpr->semantic(scope);

  // Проверяем, что условие не является "void"
  if (Cond->ExprType == nullptr || Cond->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaConditionIsVoid);
    return nullptr;
  }

  // Проверяем, что условие может быть преобразовано в "bool"
  if (!Cond->ExprType->implicitConvertTo(
    BuiltinTypeAST::get(TypeAST::TI_Bool))) {
    scope->report(Loc, diag::ERR_SemaCantConvertToBoolean);
    return nullptr;
  }

  // Результирующий тип совпадает с тем, что задан в ветке с
  // выражением, если условие истинно
  ExprType = IfExpr->ExprType;

  // Если обе части имеют одинаковые типы, то больше семантиче
  // анализ завершен
  if (IfExpr->ExprType->equal(ElseExpr->ExprType))
    SemaDone = true;
    return this;
```

```
  }

  // Исключаем вариант, когда один из операндов имеет тип "voi
  // но разрешаем если оба операнда имеют тип "void"
  if (!IfExpr->ExprType || !ElseExpr->ExprType ||
    IfExpr->ExprType->isVoid() || ElseExpr->ExprType->isVoid()
    scope->report(Loc, diag::ERR_SemaOperandIsVoid);
    return nullptr;
  }

  // Приводим типы к единому
  ElseExpr = new CastExprAST(ElseExpr->Loc, ElseExpr, ExprType
  ElseExpr = ElseExpr->semantic(scope);
  SemaDone = true;

  return this;
}

ExprAST* CondExprAST::clone() {
  return new CondExprAST(Loc, Cond->clone(), IfExpr->clone(),
                         ElseExpr->clone());
}
```

Семантический анализ вызова функции:

- Hidden text

```
static SymbolAST* resolveFunctionCall(Scope *scope, SymbolAST*
                                      CallExprAST* args) {
  // Проверяем, что это функция
  if (isa<FuncDeclAST>(func)) {
```

```
    FuncDeclAST* fnc = static_cast< FuncDeclAST* >(func);
    FuncTypeAST* type = static_cast< FuncTypeAST* >(fnc->ThisT

    // Количество аргументов должно совпадать с количеством
    // параметров у функции
    if (args->Args.size() != type->Params.size()) {
      scope->report(args->Loc,
                    diag::ERR_SemaInvalidNumberOfArgumentsInCa
      return nullptr;
    }

    ExprList::iterator arg = args->Args.begin();
    ParameterList::iterator it = type->Params.begin();

    // Проверяем все аргументы
    for (ParameterList::iterator end = type->Params.end();
         it != end; ++it, ++arg) {
      // Проверяем, что аргумент может быть использован для вы
      // и диагностируем об ошибке, если нет
      if (!(*arg)->ExprType->implicitConvertTo((*it)->Param))
        scope->report(args->Loc,
                      diag::ERR_SemaInvalidTypesOfArgumentsInC
        return nullptr;
      }
    }

    // Вызов функции может быть произведен с данными аргумента
    return func;
  }

  return nullptr;
}

ExprAST* CallExprAST::semantic(Scope* scope) {
  if (ExprType) {
    return this;
  }
```

```
// Производим семантический анализ выражения до "("
Callee = Callee->semantic(scope);

// Мы можем использовать только IdExprAST в качестве выражен
// для вызова
if (isa<IdExprAST>(Callee)) {
  SymbolAST* sym = ((IdExprAST*)Callee)->ThisSym;

  // Идентификатор может ссылаться только на функцию
  if (isa<FuncDeclAST>(sym)) {
    TypeAST* returnType = nullptr;

    // Производим семантический анализ для всех аргументов
    // функции
    for (ExprList::iterator arg = Args.begin(), end = Args.e
         arg != end; ++arg) {
      *arg = (*arg)->semantic(scope);
    }

    // Ищем функцию для вызова
    if (SymbolAST* newSym = resolveFunctionCall(scope, sym,
                                                this)) {
      FuncDeclAST* fnc = static_cast< FuncDeclAST* >(newSym)
      FuncTypeAST* type = static_cast< FuncTypeAST* >(
        fnc->ThisType);
      ExprList::iterator arg = Args.begin();
      ParameterList::iterator it = type->Params.begin();

      // Производим сопоставление аргументов и параметров
      for (ParameterList::iterator end = type->Params.end();
           it != end; ++it, ++arg) {
        // Если тип аргумента отличается от типа параметра,
        // производим преобразование типа
        if (!(*arg)->ExprType->equal((*it)->Param)) {
          ExprAST* oldArg = (*arg);
          *arg = new CastExprAST(oldArg->Loc, oldArg->clone(
```

```
                                    (*it)->Param);
            *arg = (*arg)->semantic(scope);
            delete oldArg;
          }
        }

        // Определяем тип возвращаемого значения и устанавлива
        // тип для результата вызова функции
        if (!returnType) {
          ExprType = ((FuncDeclAST*)newSym)->ReturnType;
        } else {
          ExprType = returnType;
        }

        // Устанавливаем объявление функции для вызова для
        // дальнейшей работы
        CallFunc = newSym;
        return this;
      }
    }
  }
  // Диагностируем ошибку
  scope->report(Loc, diag::ERR_SemaInvalidArgumentsForCall);
  return nullptr;
}

ExprAST* CallExprAST::clone() {
  ExprList exprs;
  ExprList::iterator it = Args.begin();
  ExprList::iterator end = Args.end();

  for ( ; it != end; ++it) {
    exprs.push_back((*it)->clone());
  }
```

```
    return new CallExprAST(Loc, Callee->clone(), exprs);
  }
```

Семантический анализ инструкций

Для начала рассмотрим какие именно функции члены иерархии инструкций будут отвечать за семантический анализ этих ветвей AST:

```
struct StmtAST {
  ...
  /// Проверяет есть ли выход из функции в данной ветви дерева
  virtual bool hasReturn();
  /// Проверяет есть ли инструкция выхода из цикла или возврат из фун
  /// данной ветви дерева
  virtual bool hasJump();
  /// Произвести семантический анализ для инструкции
  StmtAST* semantic(Scope* scope);
  /// Произвести семантический анализ для инструкции (должна быть реа
  // в потомках,  вызывается только через "semantic"
  virtual StmtAST* doSemantic(Scope* scope);

  int SemaState; ///< стадия семантического анализа для текущей инстр
};
```

Так же, как и у типов и выражений, StmtAST после семантического анализа может быть изменено (например одна инструкция может быть реализована через другую).

Один из важных классов для работы семантического анализа (и генерации кода) является LandingPadAST (более подробное описание зачем он нужен рассмотрим в следующей части серии, на данный

момент приведу только те члены класса, которые нужны для семантического анализа):

```
struct LandingPadAST {
  LandingPadAST* Prev; ///< родительский LandingPadAST
  StmtAST* OwnerBlock; ///< блок к которому относится данный LandingP
  int Breaks; ///< количество инструкций выхода из цикла в ветви дере
  /// количество инструкций возврата из функции в ветви дерева
  int Returns;
  /// количество инструкций перехода к следующей итерации цикла в вет
  int Continues;
  bool IsLoop; ///< LandingPadAST находится частью цикла
};
```

Рассмотрим более подробно сам семантический анализ для инструкций:

▾ Hidden text

```
bool StmtAST::hasReturn() {
  return false;
}

bool StmtAST::hasJump() {
  return false;
}

StmtAST* StmtAST::semantic(Scope* scope) {
  // Защита от повторного вызова
  if (SemaState > 0) {
    return this;
  }
```

```
  ++SemaState;
  return doSemantic(scope);
}

StmtAST* StmtAST::doSemantic(Scope* ) {
  assert(0 && "StmtAST::semantic should never be reached");
  return this;
}

StmtAST* ExprStmtAST::doSemantic(Scope* scope) {
  if (Expr) {
    // Проверка семантики выражения
    Expr = Expr->semantic(scope);

    // После окончания семантического анализа хранимого выраже
    // у него должен быть задан тип
    if (!Expr->ExprType) {
      scope->report(Loc, diag::ERR_SemaNoTypeForExpression);
      return nullptr;
    }
  }

  return this;
}

bool BlockStmtAST::hasReturn() {
  return HasReturn;
}

bool BlockStmtAST::hasJump() {
  return HasJump;
}

StmtAST* BlockStmtAST::doSemantic(Scope* scope) {
  // Для блока мы должны создать новую область видимости
  ThisBlock = new ScopeSymbol(Loc, SymbolAST::SI_Block, nullpt
  Scope* s = scope->push((ScopeSymbol*)ThisBlock);
```

```
// Создать LandingPadAST для данного блока
LandingPad = new LandingPadAST(s->LandingPad);
LandingPad->OwnerBlock = this;
s->LandingPad = LandingPad;

ExprList args;

// Проверяем все ветви дерева принадлежащие данному блоку
for (StmtList::iterator it = Body.begin(), end = Body.end();
     it != end; ++it) {
  // Если в предыдущей инструкции был "break", "continue" ил
  // "return", то диагностируем об ошибке (предотвращаем
  // появление кода, который не может быть достижимым
  if (HasJump) {
    scope->report(Loc, diag::ERR_SemaDeadCode);
    return nullptr;
  }

  // Проверяем семантику вложенной инструкции
  *it = (*it)->semantic(s);

  // Проверяем, что это "break", "continue" или "return"
  if ((*it)->isJump()) {
    HasJump = true;

    // Проверяем, что это "return"
    if ((*it)->hasReturn()) {
      HasReturn = true;
    }
  } else {
    // Это обычная инструкция, но все равно проверяем, налич
    // "break", "continue" или "return" в дочерних ветках
    // инструкции
    HasJump = (*it)->hasJump();
    HasReturn = (*it)->hasReturn();
  }
```

```
  }

  // Удаляем область видимости
  s->pop();

  return this;
}

StmtAST* DeclStmtAST::doSemantic(Scope* scope) {
  // Производим семантический анализ для всех объявлений
  for (SymbolList::iterator it = Decls.begin(), end = Decls.er
       it != end; ++it) {
    (*it)->semantic(scope);
    (*it)->semantic2(scope);
    (*it)->semantic3(scope);
    (*it)->semantic4(scope);
    (*it)->semantic5(scope);
  }

  return this;
}

bool BreakStmtAST::hasJump() {
  return true;
}

StmtAST* BreakStmtAST::doSemantic(Scope* scope) {
  // Проверяем, что мы находимся в цикле, т. к. "break" может
  // только в цикле
  if (!scope->BreakLoc) {
    scope->report(Loc, diag::ERR_SemaInvalidJumpStatement);
    return nullptr;
  }

  // Запоминаем местоположение точки, куда нужно перевести
  // управление для выхода из цикла
  BreakLoc = scope->LandingPad;
```

```
  ++BreakLoc->Breaks;
  return this;
}

// ContinueStmtAST implementation
bool ContinueStmtAST::hasJump() {
  return true;
}

StmtAST* ContinueStmtAST::doSemantic(Scope* scope) {
  // Проверяем, что мы находимся в цикле, т. к. "continue" мож
  // быть только в цикле
  if (!scope->ContinueLoc) {
    scope->report(Loc, diag::ERR_SemaInvalidJumpStatement);
    return nullptr;
  }

  // Запоминаем местоположение точки, куда нужно перевести
  // управление для перехода к следующей итерации цикла
  ContinueLoc = scope->LandingPad;
  ++ContinueLoc->Continues;
  return this;
}

bool ReturnStmtAST::hasReturn() {
  return true;
}

bool ReturnStmtAST::hasJump() {
  return true;
}

StmtAST* ReturnStmtAST::doSemantic(Scope* scope) {
  assert(scope→LandingPad);
  // Сохраняем местоположения точки, куда нужно перевести
  // управление для выхода из функции
  ReturnLoc = scope->LandingPad;
```

```
  ++ReturnLoc->Returns;
  // Проверка наличия возвращаемого значения
  if (Expr) {
    // Если тип возвращаемого значения функции "void", то
    // сигнализируем об ошибке
    if (!scope->EnclosedFunc->ReturnType
        || scope->EnclosedFunc->ReturnType->isVoid()) {
      scope->report(Loc, diag::ERR_SemaReturnValueInVoidFuncti
      return nullptr;
    }

    // Производим семантический анализ возвращаемого значения
    Expr = Expr->semantic(scope);

    // Если тип возвращаемого значения не совпадает с типом
    // возвращаемого значения самой функции, то мы должны
    // произвести преобразование типов
    if (!scope->EnclosedFunc->ReturnType->equal(Expr->ExprType
      Expr = new CastExprAST(Loc, Expr,
                             scope->EnclosedFunc->ReturnType);
      Expr = Expr->semantic(scope);
    }

    return this;
  }

  // У нас нет выражения для возврата. Проверяем, что тип
  // возвращаемого значения самой функции "void" и сигнализиру
  // об ошибке, если он отличен от "void"
  if (scope->EnclosedFunc->ReturnType
      && !scope->EnclosedFunc->ReturnType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaReturnVoidFromFunction);
    return nullptr;
  }

  return this;
}
```

```
bool WhileStmtAST::hasReturn() {
  // Всегда возвращаем "false", т. к. может быть 0 итераций
  return false;
}

StmtAST* WhileStmtAST::doSemantic(Scope* scope) {
  // Производим семантический анализ условия цикла
  Cond = Cond->semantic(scope);

  // Условие цикла должно иметь не "void" тип
  if (!Cond->ExprType || Cond->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaConditionIsVoid);
    return nullptr;
  }

  // Проверяем, что условие цикла может быть преобразовано в "
  if (!Cond->ExprType->implicitConvertTo(
    BuiltinTypeAST::get(TypeAST::TI_Bool))) {
    scope->report(Loc, diag::ERR_SemaCantConvertToBoolean);
    return nullptr;
  }

  // Делаем копии для точек возврата из цикла и перехода к
  // следующей итерации
  StmtAST* oldBreak = scope->BreakLoc;
  StmtAST* oldContinue = scope->ContinueLoc;

  // Устанавливаем данный цикл в качестве точек возврата из ци
  // и перехода к следующей итерации
  scope->BreakLoc = this;
  scope->ContinueLoc = this;
  // Создаем новую LandingPadAST для всех вложенных инструкций
  LandingPad = new LandingPadAST(scope->LandingPad);
  LandingPad->IsLoop = true;
  scope->LandingPad = LandingPad;
```

```
  // Производим семантический анализ тела цикла
  Body = Body->semantic(scope);

  // Восстанавливаем предыдущие значения для точек возврата
  scope->BreakLoc = oldBreak;
  scope->ContinueLoc = oldContinue;
  scope->LandingPad = LandingPad->Prev;

  if (PostExpr) {
    // Производим семантический анализ для "PostExpr", который
    // создан в процессе конвертации цикла "for" в цикл "while
    PostExpr = PostExpr->semantic(scope);
  }

  return this;
}

StmtAST* ForStmtAST::doSemantic(Scope* scope) {
  // Заменяем цикл "for" эквивалентным аналогом цикла "while",
  // бы упростить генерацию кода, но предварительно проверив
  // семантику
  // {
  //   init
  //   while (cond) {
  //     body
  //     continueZone: post
  //   }
  // }

  StmtAST* init = nullptr;

  // Если в цикле задано выражение для инициализации или объяв
  // переменные цикла, то нужно создать на их основе
  // соответствующие инструкции
  if (InitExpr) {
    init = new ExprStmtAST(Loc, InitExpr);
  } else if (!InitDecls.empty()) {
```

```
    init = new DeclStmtAST(Loc, InitDecls);
  }

  if (Cond) {
    // У нас есть условие выхода из цикла
    StmtList stmts;

    // Если у нас есть блок инициализации цикла, то добавляем
    // к телу новой конструкции
    if (init) {
      stmts.push_back(init);
    }

    // Создаем новый цикл "while" на основе данного цикла "for
    // добавляем его к списку инструкций в блоке
    WhileStmtAST* newLoop = new WhileStmtAST(Loc, Cond, Body);
    newLoop->PostExpr = Post;
    stmts.push_back(newLoop);

    // Очищаем и удаляем данную ветку
    InitExpr = nullptr;
    InitDecls.clear();
    Cond = nullptr;
    Post = nullptr;
    Body = nullptr;
    delete this;

    // Создаем новы блочный элемент для нового цикла и произвс
    // его семантический анализ
    StmtAST* res = new BlockStmtAST(Loc, stmts);
    return res->semantic(scope);
  } else {
    // У нас нет условия выхода из цикла
    StmtList stmts;

    // Если у нас есть блок инициализации цикла, то добавляем
    // к телу новой конструкции
```

```
    if (init) {
      stmts.push_back(init);
    }

    // Создаем новый цикл "while" на основе данного цикла "for
    // добавляем его к списку инструкций в блоке
    WhileStmtAST* newLoop = new WhileStmtAST(Loc,
                                             new IntExprAST(Lo
                                             Body);
    newLoop->PostExpr = Post;
    stmts.push_back(newLoop);

    // Очищаем и удаляем данную ветку
    InitExpr = nullptr;
    InitDecls.clear();
    Cond = nullptr;
    Post = nullptr;
    Body = nullptr;
    delete this;

    // Создаем новы блочный элемент для нового цикла и произво
    // его семантический анализ
    StmtAST* res = new BlockStmtAST(Loc, stmts);
    return res->semantic(scope);
  }
}

bool IfStmtAST::hasReturn() {
  if (!ElseBody) {
    return false;
  }

  // Возвращаем "true" только если обе ветки имеют инструкции
  // возврата из функции
  return ThenBody->hasReturn() && ElseBody->hasReturn();
}
```

```
bool IfStmtAST::hasJump() {
  if (!ElseBody) {
    return false;
  }

  // Возвращаем "true" только если обе ветки имеют инструкции
  // возврата из функции или выхода из цикла
  return ThenBody->hasJump() && ElseBody->hasJump();
}

StmtAST* IfStmtAST::doSemantic(Scope* scope) {
  // Производим семантический анализ условия
  Cond = Cond->semantic(scope);

  // Запрещаем условия с типом "void"
  if (!Cond->ExprType
      || Cond->ExprType == BuiltinTypeAST::get(TypeAST::TI_Voi
    scope->report(Loc, diag::ERR_SemaConditionIsVoid);
    return nullptr;
  }

  // Проверяем, что условие может быть преобразовано в "bool"
  if (!Cond->ExprType->implicitConvertTo(
    BuiltinTypeAST::get(TypeAST::TI_Bool))) {
    scope->report(Loc, diag::ERR_SemaCantConvertToBoolean);
    return nullptr;
  }

  // Создаем новый LandingPadAST
  LandingPad = new LandingPadAST(scope->LandingPad);
  scope->LandingPad = LandingPad;

  // Производим семантический анализ ветки, если условие истин
  ThenBody = ThenBody->semantic(scope);

  // Производим семантический анализ ветки, если условие ложно
  // если она есть
```

```
    if (ElseBody) {
      ElseBody = ElseBody->semantic(scope);
    }

    // Восстанавливаем старый LandingPadAST
    scope->LandingPad = LandingPad->Prev;

    return this;
  }
```

Семантический анализ для объявлений

Для начала рассмотрим какие именно функции члены иерархии объявлений будут отвечать за семантический анализ этих ветвей AST:

```
struct SymbolAST {
  /// Получить тип объявления
  virtual TypeAST *getType();
  /// Произвести 1 фазу семантического анализа
  void semantic(Scope *scope);
  /// Произвести 2 фазу семантического анализа
  void semantic2(Scope *scope);
  /// Произвести 3 фазу семантического анализа
  void semantic3(Scope *scope);
  /// Произвести 4 фазу семантического анализа
  void semantic4(Scope *scope);
  /// Произвести 5 фазу семантического анализа
  void semantic5(Scope *scope);

  /// Произвести 1 фазу семантического анализа (должна быть переопред
  /// потомках, вызывается только через "semantic")
  virtual void doSemantic(Scope *scope);
  /// Произвести 2 фазу семантического анализа (может быть переопреде
```

```
    /// потомках, вызывается только через "semantic2")
    virtual void doSemantic2(Scope *scope);
    /// Произвести 3 фазу семантического анализа (может быть переопреде
    /// потомках, вызывается только через "semantic3")
    virtual void doSemantic3(Scope *scope);
    /// Произвести 4 фазу семантического анализа (может быть переопреде
    /// потомках, вызывается только через "semantic4")
    virtual void doSemantic4(Scope *scope);
    /// Произвести 5 фазу семантического анализа (может быть переопреде
    /// потомках, вызывается только через "semantic5")
    virtual void doSemantic5(Scope *scope);

    /// Поиск дочернего объявления ("flags" — 1 если не нужен поиск в
    /// родительских классах)
    virtual SymbolAST *find(Name *id, int flags = 0);
    /// область видимости, в которой было объявлено данное объявление
    SymbolAST *Parent;
    int SemaState;   ///< текущая стадия семантического анализа
};
```

Из-за правил поиска идентификаторов в simple, весь семантический анализ был разбит на 5 фаз:

1. Создание списка всех объявлений в области видимости;
2. Разрешение типов базовых классов;
3. Разрешение типов для имен переменных и функций членов структур и классов;
4. Построение таблиц виртуальных функций, конструкторов и деструкторов;
5. Анализ функций и их тел.

Разбивка семантики на несколько фаз позволяет упростить грамматику языка, т. к. не нужно вводить дополнительные конструкции для

объявления и определения (т. е. введение имени в программу (что бы оно могло быть использовано) и описание ее реализации (т. е. для классов это описание всех его функций и переменных членов, а для функций ее тело). Например в C++ можно сперва объявить класс (написать "class A;"), что позволяет использовать имя этого класса в других объявлениях (например в качестве параметра функции), а потом в другой части исходного кода произвести определение — описать все функции и переменные члены класса. При разбивке семантического анализа на части, такие разграничения объявления и определения на разные сущности просто не нужны, т. к. даже циклические зависимости могут быть спокойно разрешены, т. к. в каждой фазе происходят действия, которые позволят продолжить анализ на следующих стадиях.

Ниже рассмотрим сам семантический анализ для объявлений:

▾ Hidden text

```
TypeAST* SymbolAST::getType() {
  assert(0 && "SymbolAST::getType should never be reached");
  return nullptr;
}

void SymbolAST::semantic(Scope* scope) {
  // Пропускаем семантический анализ для этой фазы, если она у
  // была завершена
  if (SemaState > 0) {
    return;
  }

  // Произвести семантический анализ и переход к следующей ста
  doSemantic(scope);
  ++SemaState;
}
```

```
void SymbolAST::semantic2(Scope* scope) {
  // Пропускаем семантический анализ для этой фазы, если она у
  // была завершена
  assert(SemaState >= 1);
  if (SemaState > 1) {
    return;
  }

  // Произвести семантический анализ и переход к следующей ста
  doSemantic2(scope);
  ++SemaState;
}

void SymbolAST::semantic3(Scope* scope) {
  // Пропускаем семантический анализ для этой фазы, если она у
  // была завершена
  assert(SemaState >= 2);
  if (SemaState > 2) {
    return;
  }

  // Произвести семантический анализ и переход к следующей ста
  doSemantic3(scope);
  ++SemaState;
}

void SymbolAST::semantic4(Scope* scope) {
  // Пропускаем семантический анализ для этой фазы, если она у
  // была завершена
  assert(SemaState >= 3);
  if (SemaState > 3) {
    return;
  }

  // Произвести семантический анализ и переход к следующей ста
  doSemantic4(scope);
  ++SemaState;
```

```
}

void SymbolAST::semantic5(Scope* scope) {
  // Пропускаем семантический анализ для этой фазы, если она у
  // была завершена
  assert(SemaState >= 4);
  if (SemaState > 4) {
    return;
  }

  // Произвести семантический анализ и переход к следующей ста
  doSemantic5(scope);
  ++SemaState;
}

void SymbolAST::doSemantic(Scope* ) {
  // Данная функция обязательна для реализации в дочерних клас
  assert(0 && "SymbolAST::semantic should never be reached");
}

void SymbolAST::doSemantic2(Scope* ) {
  // По умолчанию игнорируем данную фазу
}

void SymbolAST::doSemantic3(Scope* ) {
  // По умолчанию игнорируем данную фазу
}

void SymbolAST::doSemantic4(Scope* ) {
  // По умолчанию игнорируем данную фазу
}

void SymbolAST::doSemantic5(Scope* ) {
  // По умолчанию игнорируем данную фазу
}

SymbolAST* SymbolAST::find(Name* id, int flags) {
```

```
  // В базовой реализации только проверяем имя самой сущности
  if (id == Id) {
    return this;
  }

  return nullptr;
}

TypeAST* VarDeclAST::getType() {
  return ThisType;
}

void VarDeclAST::doSemantic(Scope* scope) {
  // Исключаем повторные объявления переменных
  if (scope->find(Id)) {
    scope->report(Loc, diag::ERR_SemaIdentifierRedefinition);
    return;
  }
  // Добавляем переменную в список объявленных переменных в те
  // области видимости
  ((ScopeSymbol*)scope->CurScope)->Decls[Id] = this;
  Parent = scope->CurScope;
  // Так же добавляем переменную к списку объявленных переменн
  // функции, если она была объявлена в функции (нужны для
  // генерации кода)
  if (scope->EnclosedFunc) {
    scope->EnclosedFunc->FuncVars.push_back(this);
  }
  // Производим семантический анализ для типа переменной
  ThisType = ThisType->semantic(scope);
}

void VarDeclAST::doSemantic3(Scope* scope) {
  // Проверяем наличие инициализатора
  if (Val) {
    // Производим семантический анализ инициализирующего выраж
    Val = Val->semantic(scope);
```

```
    // Запрещаем использования выражений с типов "void" в
    // инициализации
    if (!Val->ExprType || Val->ExprType->isVoid()) {
      scope->report(Loc, diag::ERR_SemaVoidInitializer);
      return;
    }

    // Если типы не совпадают, то добавляем преобразование
    if (!Val->ExprType->equal(ThisType)) {
      Val = new CastExprAST(Loc, Val, ThisType);
      Val = Val->semantic(scope);
    }
  }
}

ScopeSymbol::~ScopeSymbol() {
}

SymbolAST* ScopeSymbol::find(Name* id, int /*flags*/) {
  // Производим поиск в объявленных в данном блоке объявлениях
  SymbolMap::iterator it = Decls.find(id);

  if (it == Decls.end()) {
    return nullptr;
  }

  return it->second;
}

TypeAST* ParameterSymbolAST::getType() {
  return Param->Param;
}

void ParameterSymbolAST::doSemantic(Scope* ) {
}
```

```
SymbolAST* ParameterSymbolAST::find(Name* id, int ) {
  if (id == Param->Id) {
    return this;
  }

  return nullptr;
}

TypeAST* FuncDeclAST::getType() {
  return ThisType;
}

void FuncDeclAST::doSemantic(Scope* scope) {
  // Производим семантический анализ для прототипа функции
  ThisType = ThisType->semantic(scope);
  Parent = scope->CurScope;
  // Настраиваем тип возвращаемого значения
  ReturnType = ((FuncTypeAST*)ThisType)->ReturnType;

  // Отдельная проверка для функции "main"
  if (Id->Length == 4 && memcmp(Id->Id, "main", 4) == 0) {
    FuncTypeAST* thisType = (FuncTypeAST*)ThisType;

    // Должна не иметь параметров
    if (thisType->Params.size()) {
      scope->report(Loc, diag::ERR_SemaMainParameters);
      return;
    }

    // Должна возвращать "float"
    if (ReturnType != BuiltinTypeAST::get(TypeAST::TI_Float))
      scope->report(Loc, diag::ERR_SemaMainReturnType);
      return;
    }
  }

  // Проверяем, что идентификатор еще не был объявлен ранее
```

```
  if (SymbolAST* fncOverload = scope->findMember(Id, 1)) {
    scope->report(Loc, diag::ERR_SemaFunctionRedefined, Id->Ic
    return;
  }

  // Добавляем функцию к списку объявлений
  ((ScopeSymbol*)Parent)->Decls[Id] = this;
}

void FuncDeclAST::doSemantic5(Scope* scope) {
  FuncTypeAST* func = (FuncTypeAST*)ThisType;
  // Проверяем наличие тела функции (если его нет, то это прот
  if (Body) {
    // Создаем новую область видимости и устанавливаем текущую
    // функции в данной области видимости
    Scope* s = scope->push(this);
    s->EnclosedFunc = this;

    // Производим проверку всех параметров функции
    for (ParameterList::iterator it = func->Params.begin(),
         end = func->Params.end(); it != end; ++it) {
      ParameterAST* p = *it;
      // Особая обработка для именованных параметров
      if (p->Id) {
        // Запрещаем переопределение
        if (find(p->Id)) {
          scope->report(Loc, diag::ERR_SemaIdentifierRedefinit
          return;
        }

        // Для каждого параметра в прототипе, создаем отдельну
        // переменную в самой функции
        SymbolAST* newSym = new ParameterSymbolAST(p);
        Decls[p->Id] = newSym;
        FuncVars.push_back(newSym);
      }
    }
```

```
    // Устанавливаем новый LandingPadAST
    LandingPadAST* oldLandingPad = s->LandingPad;
    LandingPad = new LandingPadAST();
    s->LandingPad = LandingPad;

    // Производим семантический анализ тела функции
    Body = Body->semantic(s);

    // Восстанавливаем старый LandingPadAST
    s->LandingPad = oldLandingPad;

    // Проверяем, что функция с типов возвращаемого значения
    // отличным от "void" вернуло значение
    if (!ReturnType->isVoid() && !Body->hasReturn()) {
      scope->report(Loc,
                    diag::ERR_SemaMissingReturnValueInFunctior
      return;
    }

    // Удаляем область видимости для данной функции
    s->pop();
  }
}

// Функции, которые будут доступны из simple
extern "C" void lle_X_printDouble(double val) {
  outs() << val;
}

extern "C" void lle_X_printLine() {
  outs() << "\n";
}

/// Добавить прототип функции и связать ее с функцией C++
FuncDeclAST* addDynamicFunc(const char* protoString,
                            const char* newName,
```

```
                                ModuleDeclAST* modDecl,
                                void* fncPtr) {
  // Разбор прототипа функции
  FuncDeclAST* func = (FuncDeclAST*)parseFuncProto(protoString
  // Добавляем функцию и указываем, что ее компиляция уже была
  // произведена
  func->CodeValue = Function::Create(
    (FunctionType*)func->ThisType->getType(),
    Function::ExternalLinkage,
    Twine(newName),
    getSLContext().TheModule
  );
  func->Compiled = true;
  modDecl->Members.insert(modDecl->Members.begin(), func);

  // Делаем функцию доступной из LLVM
  ExitOnError ExitOnErr;

  ExitOnErr(getJIT().addSymbol(newName, fncPtr));

  return func;
}

void initRuntimeFuncs(ModuleDeclAST* modDecl) {
  addDynamicFunc("fn print(_: float)", "lle_X_printDouble",
                 modDecl, (void*)lle_X_printDouble);
  addDynamicFunc("fn printLn()", "lle_X_printLine",
                 modDecl, (void*)lle_X_printLine);
}

void ModuleDeclAST::semantic() {
  Scope s(this);

  // Инициализация runtime функций
  initRuntimeFuncs(this);

  // Производим семантический анализ всех базовых типов
```

```
for (int i = TypeAST::TI_Void; i <= TypeAST::TI_Float; ++i)
  BuiltinTypeAST::get(i)->semantic(&s);
}

// Производим семантический анализ для всех объявлений
// (кроме функций)
for (SymbolList::iterator it = Members.begin(),
     end = Members.end(); it != end; ++it) {
  if (!isa<FuncDeclAST>(*it))
    (*it)->semantic(&s);
}

// Производим семантический анализ всех функций
for (SymbolList::iterator it = Members.begin(),
     end = Members.end(); it != end; ++it) {
  if (isa<FuncDeclAST>(*it))
    (*it)->semantic(&s);
}

// Запускаем 2-ю фазу семантического анализа для всех объявл
for (SymbolList::iterator it = Members.begin(),
     end = Members.end(); it != end; ++it) {
  (*it)->semantic2(&s);
}

// Запускаем 3-ю фазу семантического анализа для всех объявл
for (SymbolList::iterator it = Members.begin(),
     end = Members.end(); it != end; ++it) {
  (*it)->semantic3(&s);
}

// Запускаем 4-ю фазу семантического анализа для всех объявл
for (SymbolList::iterator it = Members.begin(),
     end = Members.end(); it != end; ++it) {
  (*it)->semantic4(&s);
}
```

```
  // Запускаем 5-ю фазу семантического анализа для всех объявл
  for (SymbolList::iterator it = Members.begin(),
       end = Members.end(); it != end; ++it) {
    (*it)->semantic5(&s);
  }
}
```

Заключение

В данной части мы рассмотрели семантический анализ программы на языке simple, произвели проверку ее корректности и подготовили наше промежуточное представление к заключительной части — генерации кода.

В статье мы научились изменять изначальное представление программы его аналогом, что позволит упростить многие части генерации кода (например автоматическое создание преобразований типов, где они были необходимы, позволит не задумываться о преобразовании типов на стадии генерации кода, т. к. этим будет заниматься отдельный класс). Автоматическая генерация и модификация дерева может быть полезна при решении многих задач в проектировании своего языка. В последующих частях мы еще не раз будем использовать данный подход, например для генерации кода для вызова конструкторов и деструкторов для объектов класса, генерации конструкторов по умолчанию, а так же автоматический вызов конструкторов и деструкторов для родительских классов. В более сложных языках программирования данный подход может быть применен для генерации замыканий, итераторов и многих других высокоуровневых конструкциях.

Разбиение семантического анализа на фазы так же позволяет делать язык более выразительным и дает возможность убрать из него многие

ограничения, которые при однопроходном варианте приходилось бы решать другими способами.

Полный исходный код доступен на gihub. В следующей статье мы закончим реализацию базовой версии языка simple и добавим генерацию кода для LLVM IR.

Полезные ссылки

1. https://en.wikipedia.org/wiki/Scope_(computer_science)
2. https://en.wikipedia.org/wiki/Type_system
3. https://en.wikipedia.org/wiki/Name_mangling

Теги: llvm, компиляторы, c++

Хабы: Open source, Программирование, Компиляторы

Создаем свой собственный язык программирования с использованием LLVM. Часть 3: Генерация кода

6.3K

Open source*, Программирование*, Компиляторы*

В предыдущих статьях мы рассмотрели и реализовали лексический и синтаксический анализаторы, а так же реализовали семантический анализ для нашего учебного языка, что дало нам основу. В данной статье мы продолжим начатое и реализуем генерацию кода для LLVM IR.

Оглавление серии

LLVM

LLVM это набор компиляторов и инструментов, которые могут быть использованы для создания фронтендов для любого языка программирования и бэкендов для любой архитектуры набора команд. LLVM разработан вокруг языко-независимого промежуточного представления (LLVM IR), который служит в качестве портативного, высокоуровневого языка ассемблера, который может быть оптимизирован при помощи различных преобразований.

Программа LLVM состоит из модулей, которые являются единицами трансляции исходной программы. Каждый модуль содержит функции,

глобальные переменные и элементы таблицы символов. Модули могут быть объединены с использованием линковщика LLVM.

Определение функции представляет собой прототип самой функции (ее имя, типы ее параметров, тип возвращаемого значения и некоторые другие необязательные части, такие как способ линковки, ее видимость, способ передачи параметров и др.), а также ее тела, которое может состоять из одного или нескольких базовых блоков.

Базовый блок — является последовательностью операций (инструкций), которые образуют логику базового блока. Любой базовый блок должен завершаться инструкцией терминатором (условный или безусловный переход, возврат из функции и др.).

Система типов LLVM поддерживает большое количество типов данных, которые делятся на (рассмотрим только те типы, которые мы будем использовать в процессе реализации нашего учебного языка):

1. Целые числа различной разрядности (например i1, i8, i32, i64);
2. Числа с плавающей точкой (например float, double);
3. void;
4. Указатели (ptr);
5. Массивы (например [20 x i32]);
6. Структуры (например { i32, double, i64 });
7. Функции (например i32 (i32, double)).

Основная особенность LLVM IR состоит в том, что он записан в формате SSA (Static Single Assignment), при которой каждому регистру значение может быть присвоено только один раз. Что упрощает анализ потока данных программы и производить различные оптимизации.

Но такая форма накладывает некоторые ограничения, если в программе есть переменные, которые могут изменяться, для решения этой

проблемы используется инструкция phi, которая присваивает значение на основе того, из какого блока был переход в блок содержащую инструкцию phi. Например:

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

Более подробнее про LLVM IR можно почитать в [1], а как можно все это применить в [2].

Далее мы рассмотрим генерацию кода для различных элементов языка (мы будем использовать JIT, реализацию которого можно будет посмотреть в полном исходном коде проекта на github, либо почитать в [3]).

Генерация кода для типов

Для начала посмотрим, что изменилось в описании типа для генерации кода:

```
struct TypeAST {
   /// Получить тип LLVM для данной ветки дерева
  virtual llvm::Type* getType() = 0;

  llvm::Type* ThisType; ///< сгенерированный тип LLVM
};
```

Сама же генерация для веток дерева, которые относятся к типам очень простая, нужно всего сопоставить тип языка к аналогичному типу в LLVM:

Hidden text

```
Type* BuiltinTypeAST::getType() {
  static Type* builtinTypes[] = {
    Type::getVoidTy(getGlobalContext()),
    Type::getInt1Ty(getGlobalContext()),
    Type::getInt32Ty(getGlobalContext()),
    Type::getDoubleTy(getGlobalContext()),
    Type::getInt8Ty(getGlobalContext()),
    Type::getInt8PtrTy(getGlobalContext())
  };

  if (ThisType) {
    return ThisType;
  }

  return ThisType = builtinTypes[TypeKind];
}

Type* FuncTypeAST::getType() {
  // Генерируем тип только один раз
  if (ThisType) {
    return ThisType;
  }

  // Если ReturnType не указан, то указываем "void"
  Type* returnType = (ReturnType ? ReturnType->getType() :
    Type::getVoidTy(getGlobalContext()));

  if (Params.empty()) {
    // Возвращаем функцию без параметров
    return ThisType = FunctionType::get(returnType, false);
  }

  std::vector< Type* > params;
```

```
  // Создаем список параметров
  for (ParameterList::iterator it = Params.begin(),
    end = Params.end(); it != end; ++it) {
    params.push_back((*it)->Param->getType());
  }

  // Создаем тип для функции с параметрами
  return ThisType = FunctionType::get(returnType, params, fals
}
```

Генерация кода для выражений

Рассмотрим изменения необходимые для генерации кода выражений:

```
struct ExprAST {
  /// Генерация кода для выражения, которое может быть использовано в
  /// части "="
  virtual llvm::Value *getLValue(SLContext &Context);
  /// Генерация кода для выражения, которое может быть использовано в
  /// части "="
  virtual llvm::Value *getRValue(SLContext &Context) = 0;
};
```

Зачем нужны две функции для lvalue и rvalue? Рассмотрим на примере выражения

```
i
```

rvalue этого выражения будет значение этой переменной, а lvalue будет адрес этой переменной в памяти, который может быть использована для присвоения нового значения.

Рассмотрим функции, которые будут использоваться во время генерации очень часто:

Hidden text

```
/// Сгенерировать код для целочисленной константы
ConstantInt* getConstInt(uint64_t value) {
  return ConstantInt::get(Type::getInt32Ty(getGlobalContext()),
                          value, true);
}

/// Конвертировать тип выражения в "bool"
/// \param[in] val — значение для конвертации
/// \param[in] type — тип изначального выражения
/// \param[in] builder — конструктор LLVM IR
Value* promoteToBool(Value* val,
                     TypeAST* type,
                     IRBuilder< >& builder) {
  if (type == BuiltinTypeAST::get(TypeAST::TI_Bool)) {
    // Это уже "bool"
    return val;
  }

  if (type->isInt()) {
    // Для целочисленного типа генерируем сравнение с 0
    return builder.CreateICmpNE(val,
                                ConstantInt::get(
                                  type->getType(),
                                  0)
                               );
  } else {
```

```
    assert(type->isFloat());
    // Для числа с плавающей точкой генерируем сравнение с 0.0
    return builder.CreateFCmpUNE(val, ConstantFP::get(
      Type::getDoubleTy(getGlobalContext()), 0.0));
  }
}
```

Рассмотрим реализацию генерации кода для самих выражений:

```
Value* ExprAST::getLValue(SLContext& ) {
  assert(0 && "");
  return nullptr;
}

Value* IntExprAST::getRValue(SLContext& ) {
  // Создать целочисленную константу нужного типа с нужным
  // значением
  return ConstantInt::get(ExprType->getType(), Val, true);
}

Value* FloatExprAST::getRValue(SLContext& ) {
  // Создать константу для числа с плавающей точкой нужного ти
  // нужным значением
  return ConstantFP::get(ExprType->getType(), Val);
}

Value* IdExprAST::getLValue(SLContext& Context) {
  // Получить адрес переменной (сам адрес сгенерирует SymbolAS
  // на который ссылается переменная)
  return ThisSym->getValue(Context);
}

Value* IdExprAST::getRValue(SLContext& Context) {
  if (isa<FuncDeclAST>(ThisSym)) {
    // Особая обработка для функций
    return ThisSym->getValue(Context);
  }

  // Для остальных вариантов нужно сгенерировать инструкцию "l
```

```
  return Context.TheBuilder->CreateLoad(
    ExprType->getType(),
    ThisSym->getValue(Context),
    StringRef(Val->Id, Val->Length)
  );
}

Value* CastExprAST::getRValue(SLContext& Context) {
  // Сначала проверяем, что это преобразование в целочисленный
  if (ExprType->isInt()) {
    if (Val->ExprType->isBool()) {
      // Если исходный тип "bool", то дополняем 0
      return Context.TheBuilder->CreateZExt(
        Val->getRValue(Context),
        ExprType->getType()
      );
    }

    assert(Val->ExprType->isFloat());
    // Генерируем преобразование "float" в "int"
    return Context.TheBuilder->CreateFPToSI(
      Val->getRValue(Context),
      ExprType->getType()
    );
  }

  if (ExprType->isBool()) {
    // Для преобразования в "bool"
    return promoteToBool(Val->getRValue(Context), Val->ExprTyp
      *Context.TheBuilder);
  } else if (Val->ExprType->isInt()) {
    // Преобразование "int" в "float"
    return Context.TheBuilder->CreateSIToFP(
      Val->getRValue(Context),
      ExprType->getType()
    );
  } else if (Val->ExprType->isBool()) {
```

```
    // Преобразование "bool" в "float"
    return Context.TheBuilder->CreateUIToFP(
      Val->getRValue(Context),
      ExprType->getType()
    );
  }

  assert(0 && "should never be reached");
  return nullptr;
}

Value* UnaryExprAST::getRValue(SLContext& Context) {
  assert(0 && "Should never happen");
  return nullptr;
}

Value* BinaryExprAST::getRValue(SLContext& Context) {
  // Сначала необходимо проверить все специальные случаи

  // =
  if (Op == tok::Assign) {

    // Генерируем код для правой части выражения
    Value* right = RightExpr->getRValue(Context);
    // Получаем адрес по которому нужно сохранить значение
    Value* res = LeftExpr->getLValue(Context);

    // Генерируем инструкцию "store"
    Context.TheBuilder->CreateStore(right, res);
    return right;
  }

  // ,
  if (Op == tok::Comma) {
    // Генерируем код для левого и правого операнда
    LeftExpr->getRValue(Context);
    Value* rhs = RightExpr->getRValue(Context);
```

```
  // Возвращаем правый операнд
  return rhs;
}

// Постфиксная версия операторов ++ и --
if (Op == tok::PlusPlus || Op == tok::MinusMinus) {
  // Получаем адрес переменной, а так же ее значение
  Value* var = LeftExpr->getLValue(Context);
  Value* val = LeftExpr->getRValue(Context);

  if (Op == tok::PlusPlus) {
    // Создать целочисленную константу 1 и прибавить ее к
    // загруженному ранее значению
    Value* tmp = getConstInt(1);
    tmp = Context.TheBuilder->CreateAdd(val, tmp, "inctmp");
    // Сохранить новое значение и вернуть старое значение
    Context.TheBuilder->CreateStore(tmp, var);
    return val;
  } else {
    // Создать целочисленную константу -1 и прибавить ее к
    // загруженному ранее значению
    Value* tmp = getConstInt(~0ULL);
    tmp = Context.TheBuilder->CreateAdd(val, tmp, "dectmp");
    // Сохранить новое значение и вернуть старое значение
    Context.TheBuilder->CreateStore(tmp, var);
    return val;
  }
}

// ||
if (Op == tok::LogOr) {
  // Псевдокод для оператора ||

  //if (bool result = (left != 0))
  //  return result;
  //else
  //  return (right != 0);
```

```
// Генерация кода для левого операнда
Value* lhs = LeftExpr->getRValue(Context);
// Произвести преобразование в "bool"
lhs = promoteToBool(lhs, LeftExpr->ExprType,
                    *Context.TheBuilder);

// Создать блоки для ветки "false" и "result" (куда будет
// возвращено управление для продолжения работы программы)
BasicBlock* condBB = Context.TheBuilder->GetInsertBlock();
BasicBlock* resultBB = BasicBlock::Create(
  getGlobalContext(),
  "result"
);
BasicBlock* falseBB = BasicBlock::Create(
  getGlobalContext(),
  "false"
);

// Создать условный переход к "result" или "false" в
// зависимости от истинности левого операнда
Context.TheBuilder->CreateCondBr(lhs, resultBB, falseBB);

// Добавить блок "false" к функции (для которой генерируем
// и переходим к генерации кода для нее
Context.TheFunction->getBasicBlockList().push_back(falseBB
Context.TheBuilder->SetInsertPoint(falseBB);
// Генерация кода для правого операнда
Value* rhs = RightExpr->getRValue(Context);
// Обновляем блок "false", т. к. код для правого операнда
// ее сменить
falseBB = Context.TheBuilder->GetInsertBlock();
// Произвести преобразование в "bool"
rhs = promoteToBool(
  rhs,
  RightExpr->ExprType,
  *Context.TheBuilder
```

```
  );
  // Производим безусловный переход к "result"
  Context.TheBuilder->CreateBr(resultBB);

  // Добавить блок "result" к функции (для которой генерируе
  // и переходим к генерации кода для нее
  Context.TheFunction->getBasicBlockList().push_back(resultB
  Context.TheBuilder->SetInsertPoint(resultBB);

  // Создаем PHI значение, которое будет содержать значение
  // зависимости от того какая ветка была выбрана при обрабо
  // изначального условия
  PHINode* PN = Context.TheBuilder->CreatePHI(
    Type::getInt1Ty(getGlobalContext()),
    2
  );

  PN->addIncoming(lhs, condBB);
  PN->addIncoming(rhs, falseBB);

  return PN;
}

// &&
if (Op == tok::LogAnd) {
  // Псевдокод для оператора &&

  //if (left != 0 && right != 0)
  //    return true;
  //else
  //  return false;

  // Генерируем код для левой операнда
  Value* lhs = LeftExpr->getRValue(Context);
  // Произвести преобразование в "bool"
  lhs = promoteToBool(
    lhs,
```

```
    LeftExpr->ExprType,
    *Context.TheBuilder
  );

  // Создать блоки для ветки "true" и "result" (куда будет
  // возвращено управление для продолжения работы программы)
  BasicBlock* condBB = Context.TheBuilder->GetInsertBlock();
  BasicBlock* resultBB = BasicBlock::Create(
    getGlobalContext(),
    "result"
  );
  BasicBlock* trueBB = BasicBlock::Create(
    getGlobalContext(),
    "true"
  );

  // Создать условный переход к "true" или "result" в
  // зависимости от истинности левого операнда
  Context.TheBuilder->CreateCondBr(lhs, trueBB, resultBB);

  // Добавить блок "true" к функции (для которой генерируем
  // и переходим к генерации кода для нее
  Context.TheFunction->getBasicBlockList().push_back(trueBB)
  Context.TheBuilder->SetInsertPoint(trueBB);
  // Генерация кода для правого операнда
  Value* rhs = RightExpr->getRValue(Context);
  // Обновляем блок "true", т. к. код для правого операнда м
  // ее сменить
  trueBB = Context.TheBuilder->GetInsertBlock();
  // Произвести преобразование в "bool"
  rhs = promoteToBool(
    rhs,
    RightExpr->ExprType,
    *Context.TheBuilder
  );
  // Производим безусловный переход к "result"
  Context.TheBuilder->CreateBr(resultBB);
```

```
  // Добавить блок "result" к функции (для которой генерируе
  // код) и переходим к генерации кода для нее
  Context.TheFunction->getBasicBlockList().push_back(resultB
  Context.TheBuilder->SetInsertPoint(resultBB);

  // Создаем PHI значение, которое будет содержать значение
  // зависимости от того какая ветка была выбрана при обрабо
  // изначального условия
  PHINode* PN = Context.TheBuilder->CreatePHI(
    Type::getInt1Ty(getGlobalContext()),
    2
  );

  PN->addIncoming(lhs, condBB);
  PN->addIncoming(rhs, trueBB);

  return PN;
}

// Генерируем код для обоих операндов
Value* lhs = LeftExpr->getRValue(Context);
Value* rhs = RightExpr->getRValue(Context);

// Проверяем тип выражения на то, что это "int"
if (LeftExpr->ExprType == BuiltinTypeAST::get(TypeAST::TI_Ir
  // Генерация кода в соответствии с операцией
  switch (Op) {
    case tok::Plus:
      return Context.TheBuilder->CreateAdd(lhs, rhs, "addtmp
    case tok::Minus:
      return Context.TheBuilder->CreateSub(lhs, rhs, "subtmp
    case tok::Mul:
      return Context.TheBuilder->CreateMul(lhs, rhs, "multmp
    case tok::Div:
      return Context.TheBuilder->CreateSDiv(lhs, rhs, "divtm
    case tok::Mod:
```

```
      return Context.TheBuilder->CreateSRem(lhs, rhs, "remtm
    case tok::BitOr:
      return Context.TheBuilder->CreateOr(lhs, rhs, "ortmp")
    case tok::BitAnd:
      return Context.TheBuilder->CreateAnd(lhs, rhs, "andtmp
    case tok::BitXor:
      return Context.TheBuilder->CreateXor(lhs, rhs, "xortmp
    case tok::LShift:
      return Context.TheBuilder->CreateShl(lhs, rhs, "shltmp
    case tok::RShift:
      return Context.TheBuilder->CreateAShr(lhs, rhs, "shrtm
    case tok::Less:
      return Context.TheBuilder->CreateICmpSLT(lhs, rhs, "cm
    case tok::Greater:
      return Context.TheBuilder->CreateICmpSGT(lhs, rhs, "cm
    case tok::LessEqual:
      return Context.TheBuilder->CreateICmpSLE(lhs, rhs, "cm
    case tok::GreaterEqual:
      return Context.TheBuilder->CreateICmpSGE(lhs, rhs, "cm
    case tok::Equal:
      return Context.TheBuilder->CreateICmpEQ(lhs, rhs, "cmp
    case tok::NotEqual:
      return Context.TheBuilder->CreateICmpNE(lhs, rhs, "cmp
    default:
      assert(0 && "Invalid integral binary operator");
      return nullptr;
  }
}

// Генерация кода в соответствии с операцией
switch (Op) {
  case tok::Plus:
    return Context.TheBuilder->CreateFAdd(lhs, rhs, "addtmp"
  case tok::Minus:
    return Context.TheBuilder->CreateFSub(lhs, rhs, "subtmp"
  case tok::Mul:
    return Context.TheBuilder->CreateFMul(lhs, rhs, "multmp"
```

```
    case tok::Div:
      return Context.TheBuilder->CreateFDiv(lhs, rhs, "divtmp"
    case tok::Mod:
      return Context.TheBuilder->CreateFRem(lhs, rhs, "remtmp"
    case tok::Less:
      return Context.TheBuilder->CreateFCmpULT(lhs, rhs, "cmpt
    case tok::Greater:
      return Context.TheBuilder->CreateFCmpUGT(lhs, rhs, "cmpt
    case tok::LessEqual:
      return Context.TheBuilder->CreateFCmpULE(lhs, rhs, "cmpt
    case tok::GreaterEqual:
      return Context.TheBuilder->CreateFCmpUGE(lhs, rhs, "cmpt
    case tok::Equal:
      return Context.TheBuilder->CreateFCmpUEQ(lhs, rhs, "cmpt
    case tok::NotEqual:
      return Context.TheBuilder->CreateFCmpUNE(lhs, rhs, "cmpt
    default:
      assert(0 && "Invalid floating point binary operator");
      return nullptr;
  }
}

/// Генерация кода для тернарного оператора (в отличии от "if"
/// данный оператор возвращает результирующее значение и может
/// быть использован в других выражениях)
/// \param[in] Context - контекст
/// \param[in] Cond – условное выражение
/// \param[in] IfExpr – выражение, если условие истинно
/// \param[in] ElseExpr – выражение, если условие ложно
/// \param[in] isLValue - true – если нужен адрес (или lvalue)
Value* generateCondExpr(
  SLContext& Context,
  ExprAST* Cond,
  ExprAST* IfExpr,
  ExprAST* ElseExpr,
  bool isLValue) {
  // Генерация кода для условия
```

```
Value* cond = Cond->getRValue(Context);
// Произвести преобразование в "bool"
cond = promoteToBool(cond, Cond->ExprType, *Context.TheBuild

// Создаем блоки для веток "then", "else" и "ifcont"
BasicBlock* thenBB = BasicBlock::Create(
  getGlobalContext(),
  "then",
  Context.TheFunction
);
BasicBlock* elseBB = BasicBlock::Create(
  getGlobalContext(),
  "else"
);
BasicBlock* mergeBB = BasicBlock::Create(
  getGlobalContext(),
  "ifcont"
);

// Создать условный переход на "then" или "else", в зависимо
// от истинности условия
Context.TheBuilder->CreateCondBr(cond, thenBB, elseBB);
// Set insert point to a then branch
Context.TheBuilder->SetInsertPoint(thenBB);

// Генерация кода для ветки "then"
Value* thenValue;

// Генерируем lvalue или rvalue
if (isLValue) {
  thenValue = IfExpr->getLValue(Context);
} else {
  thenValue = IfExpr->getRValue(Context);
}

// Создаем безусловный переход на "ifcont" и обновляем блок
// "then", т. к. он мог смениться
```

```
Context.TheBuilder->CreateBr(mergeBB);
thenBB = Context.TheBuilder->GetInsertBlock();

// Добавить блок "else" к функции (для которой генерируем ко
// и переходим к генерации кода для нее
Context.TheFunction->getBasicBlockList().push_back(elseBB);
Context.TheBuilder->SetInsertPoint(elseBB);

Value* elseValue;

// Генерируем lvalue или rvalue
if (isLValue) {
  elseValue = ElseExpr->getLValue(Context);
} else {
  elseValue = ElseExpr->getRValue(Context);
}

// Создаем безусловный переход на "ifcont" и обновляем блок
// "else", т. к. он мог смениться
Context.TheBuilder->CreateBr(mergeBB);
elseBB = Context.TheBuilder->GetInsertBlock();

// Создаем безусловный переход на "ifcont" и обновляем блок
// "else", т. к. он мог смениться
Context.TheFunction->getBasicBlockList().push_back(mergeBB);
Context.TheBuilder->SetInsertPoint(mergeBB);

// Если результат выражения имеет тип "void", то мы больше н
// не делаем
if (!IfExpr->ExprType || IfExpr->ExprType->isVoid()) {
  return nullptr;
}

// Создаем PHI значение, которое будет содержать значение в
// зависимости от того какая ветка была выбрана при обработк
// изначального условия
PHINode* PN;
```

```
  if (isLValue) {
    // Для lvalue тип выражения является указателем
    PN = Context.TheBuilder->CreatePHI(
      PointerType::get(
        getGlobalContext(),
        Context.TheTarget->getProgramAddressSpace()
      ),
      2
    );
  } else {
    PN = Context.TheBuilder->CreatePHI(
      IfExpr->ExprType->getType(),
      2
    );
  }

  PN->addIncoming(thenValue, thenBB);
  PN->addIncoming(elseValue, elseBB);

  return PN;
}

Value* CondExprAST::getLValue(SLContext& Context) {
  return generateCondExpr(Context, Cond, IfExpr, ElseExpr, tru
}

Value* CondExprAST::getRValue(SLContext& Context) {
  return generateCondExpr(Context, Cond, IfExpr, ElseExpr, fal
}

Value* CallExprAST::getRValue(SLContext& Context) {
  Value* callee = nullptr;
  std::vector< Value* > args;
  ExprList::iterator it = Args.begin();

  assert(isa<FuncDeclAST>(CallFunc));
```

```
    FuncDeclAST* funcDecl = (FuncDeclAST*)CallFunc;
    Type* funcRawType = CallFunc->getType()->getType();
    assert(isa<FunctionType>(funcRawType));
    FunctionType* funcType = static_cast<FunctionType*>(funcRawT

    // Получаем адрес функции
    callee = CallFunc->getValue(Context);

    // Производим генерацию кода для каждого аргумента функции и
    // добавляем их к списку аргументов функции
    for (ExprList::iterator end = Args.end(); it != end; ++it) {
      Value* v = (*it)->getRValue(Context);
      args.push_back(v);
    }

    // Генерируем вызов функции
    return Context.TheBuilder->CreateCall(funcType, callee, args
  }
```

Генерация кода для инструкций

Рассмотрим изменения необходимые для генерации кода инструкций:

```
struct StmtAST {
  /// Генерация кода для инструкции
  virtual llvm::Value* generateCode(SLContext& Context);
};
struct BlockStmtAST : StmtAST {
  /// Генерация кода для инструкции
  llvm::Value* generateCode(SLContext& Context);
  /// Генерация кода для вложенных интсрукций
  /// \param[in] Context - контекст
  /// \param[in] it — начало
```

```
  /// \param[in] end – конец
  /// \note Вспомогательная функция для генерации кода для блока, нуж
  ///  т. к. некоторые конструкции нуждаются в специальной обработке
  ///  (например конструкторы и деструкторы)
  llvm::Value* generatePartialCode(SLContext& Context, StmtList::iter
    StmtList::iterator end);
};
```

Если язык, который вы хотите реализовать, достаточно сложный и поддерживает большое количество управляющих конструкций, то при генерации кода нужно принять решение, как сделать так что бы генерация кода была с одной стороны простой, а с другой стороны эффективной. Это особенно актуально, если язык поддерживает элементы ООП. Например если взять язык C++, то если мы где-то в программе создали экземпляр класса, у которого есть деструктор, то компилятор должен добавить вызов деструктора для объекта во всех местах, где данный объект выходит из своей области видимости (более подробно этот кейс мы рассмотрим в одной из следующих частей серии, в которой мы добавим поддержку классов).

Для упрощения генерации кода для инструкций и управляющих конструкций мы будем использовать следующую структуру:

```
struct LandingPadAST {
  /// Получить переменную, которая будет хранить возвращаемое значени
  llvm::Value* getReturnValue();

  /// значение для возврата (нудно использовать getReturnValue)
  llvm::Value* ReturnValue;
  llvm::BasicBlock* BreakLoc; ///< блок для возврата по "break"
  llvm::BasicBlock* ContinueLoc; ///< блок для возврата по "continue"
  llvm::BasicBlock* ReturnLoc; ///< блок для возврата по "return"
  /// блок для продолжения нормального течения функции
```

```
  llvm::BasicBlock* FallthroughLoc;
};
```

Во время генерации кода для тела функции мы будем следовать такому подходу, что если тип возвращаемого ей значения отличен от "void", то мы будем создавать переменную, в которой мы будем хранить результат этой функции, а во всех местах использования "return" мы будем сохранять в эту переменную значение для возврата и делать безусловный переход на блок возврата из функции, который уже будет возвращать значение этой переменной (clang придерживается такого же подхода). Те кто писали на C, могли видеть такой подход, когда в функции было создание большого количества различных ресурсов (различных дескрипторов, выделение блоков памяти и т. п.), то для того что бы не производить очистку всех этих ресурсов в каждом месте с "return", создавался отдельный блок с меткой, который занимался очисткой всех этих ресурсов и управление с помощью "goto" передавалось на эту метку.

Например при данном подходе, функция main на языке C имеющая вид:

```
int main() {
  for (int i = 0; i < 10; ++i) {
    if (i == 5) {
      break;
    }

    if (i == 2) {
      continue;
    }

    if (i == 7) {
      return 1;
    }
```

```
    printf("%d", i);
  }

  return 0;
}
```

могла бы быть представлена в виде:

```
int main() {
  int result;
  {
    int i = 0;
loopcond:
    if (i < 10) {
      goto loopbody;
    } else {
      goto loopend;
    }
loopbody:
    if (i == 5) {
      goto loopend;
    }
    if (i == 2) {
      goto postbody;
    }
    if (i == 7) {
      result = 1;
      goto return_point;
    }
    printf("%d", i);
postbody:
    ++i;
    goto loopcond;
loopend:
  }
```

```
  result = 0;
return_point:
  return result;
}
```

Замечание: В коде ниже применены некоторые оптимизации, которые позволяют убрать генерацию кода для блоков, которые никогда не будут вызваны. Например при генерации инструкций ветвления или циклов, часть кода может быть убрана, если истинность или ложность условия известна на этапе компиляции (Обрабатываются только простые выражения, вида 1, но не 1 + 2. Что бы сделать доступными более сложные конструкции, можно реализовать вычисление константных выражений на этапе семантического анализа).

Теперь рассмотрим генерацию кода для инструкций более подробно:

<details>
<summary>Hidden text</summary>

```
Value* StmtAST::generateCode(SLContext& Context) {
  assert(0 && "StmtAST::generateCode should never be reached")
  return nullptr;
}

Value* ExprStmtAST::generateCode(SLContext& Context) {
  if (Expr) {
    // Генерация кода для выражения
    return Expr->getRValue(Context);
  }

  return nullptr;
}

llvm::Value* BlockStmtAST::generatePartialCode(SLContext& Cont
  StmtList::iterator it, StmtList::iterator end) {
```
</details>

```
// Инициализируем блоки "break", "continue" и "return" для
// LandingPadAST текущего блока на основе родительского блок
LandingPadAST* parent = LandingPad->Prev;
LandingPad->BreakLoc = parent->BreakLoc;
LandingPad->ContinueLoc = parent->ContinueLoc;
LandingPad->ReturnLoc = parent->ReturnLoc;

// Создаем новый блок FallthroughLoc (этот блок может быть
// использован в дочерних инструкциях, в качестве точки возв
// (например для выхода из цикла))
LandingPad->FallthroughLoc = BasicBlock::Create(
  getGlobalContext(),
  "block"
);

// Генерация кода для всех вложенных инструкций
for (; it != end; ++it) {
  // Сохраняем FallthroughLoc (т. к. при использовании его
  // вложенная инструкция должна будет его обнулить)
  BasicBlock* fallthroughBB = LandingPad->FallthroughLoc;
  lastFallThrough = LandingPad->FallthroughLoc;

  // Генерируем код для вложенной инструкции
  (*it)->generateCode(Context);

  // Проверяем была ли FallthroughLoc использована во вложен
  // инструкции или нет
  if (!LandingPad->FallthroughLoc) {
    // Была. Поэтому нужно добавить данный блок в конец функ
    // как ее часть и задать его в качестве точки для вставк
    // нового кода
    Context.TheFunction->getBasicBlockList().push_back(
      fallthroughBB
    );
    Context.TheBuilder->SetInsertPoint(fallthroughBB);

    // Записать в FallthroughLoc новый созданный блок, для
```

```
    // последующего использования во вложенных инструкциях
    LandingPad->FallthroughLoc = BasicBlock::Create(
      getGlobalContext(),
      "block"
    );
  }
}

// Делаем подсчет количество "break" "continue" и "return"
// инструкций в данном блоке (в последующих статьях будет
// ясно зачем они нужны)
parent->Breaks += LandingPad->Breaks;
parent->Continues += LandingPad->Continues;
parent->Returns += LandingPad->Returns;

// Проверяем, что текущий блок имеет инструкцию завершения б
// (условный или безусловный переход)
if (!Context.TheBuilder->GetInsertBlock()->getTerminator())
  // Нет. Генерируем инструкцию перехода на FallthroughLoc
  // родительского блока и помечаем, ее как использованную
  // (путем присвоения nullptr)
  Context.TheBuilder->CreateBr(parent->FallthroughLoc);
  parent->FallthroughLoc = nullptr;

  // Если FallthroughLoc, которую мы создали для вложенных
  // инструкций не была использована, то производим ее очист
  if (!LandingPad->FallthroughLoc->hasNUsesOrMore(1)) {
    delete LandingPad->FallthroughLoc;
  }
// Проверяем, что  FallthroughLoc был использован во вложенн
// инструкциях
} else if (LandingPad->FallthroughLoc->hasNUsesOrMore(1)) {
  // Был использован, нужно добавить его в конец функции,
  // установить его в качестве места вставки нового кода, а
  // также создаем безусловный переход на FallthroughLoc
  // родительского блока, а так же помечаем его как
  // использованную (путем присвоения nullptr)
```

```
    Context.TheFunction->getBasicBlockList().push_back(
      LandingPad->FallthroughLoc
    );
    Context.TheBuilder->SetInsertPoint(LandingPad->Fallthrough
    Context.TheBuilder->CreateBr(parent->FallthroughLoc);
    parent->FallthroughLoc = nullptr;
  } else {
    // Нет. Производим ее очистку
    delete LandingPad->FallthroughLoc;
  }

  return nullptr;
}

Value* BlockStmtAST::generateCode(SLContext& Context) {
  return generatePartialCode(Context, Body.begin(), Body.end()
}

Value* DeclStmtAST::generateCode(SLContext& Context) {
  // Генерируем код для каждого объявления
  for (SymbolList::iterator it = Decls.begin(), end = Decls.er
    it != end; ++it) {
    (*it)->generateCode(Context);
  }

  return nullptr;
}

Value* LandingPadAST::getReturnValue() {
  LandingPadAST* prev = this;

  // Ищем LandingPadAST самого верхнего уровня (блок тела само
  // функции)
  while (prev->Prev) {
    prev = prev->Prev;
  }
```

```
  // Возвращаем (адрес переменной) возвращаемое значение функц
  return prev->ReturnValue;
}

Value* BreakStmtAST::generateCode(SLContext& Context) {
  // Создать безусловный переход на метку "break" текущего цик
  Context.TheBuilder->CreateBr(BreakLoc->BreakLoc);
  return nullptr;
}

Value* ContinueStmtAST::generateCode(SLContext& Context) {
  // Создать безусловный переход на метку "continue" текущего
  Context.TheBuilder->CreateBr(ContinueLoc->ContinueLoc);
  return nullptr;
}

Value* ReturnStmtAST::generateCode(SLContext& Context) {
  // Проверяем, что у нас есть возвращаемое значение
  if (Expr) {
    // Генерируем код для выражения с возвращаемым значением
    Value* retVal = Expr->getRValue(Context);
    // Создаем инструкцию "store" для сохранения возвращаемого
    // значения в переменной, для хранения результата выполнен
    // функции
    Context.TheBuilder->CreateStore(
      retVal,
      ReturnLoc->getReturnValue()
    );
    // Создаем безусловный переход на метку с выходом из функц
    Context.TheBuilder->CreateBr(ReturnLoc->ReturnLoc);
    return nullptr;
  }

  // Создаем безусловный переход на метку с выходом из функции
  Context.TheBuilder->CreateBr(ReturnLoc->ReturnLoc);
  return nullptr;
}
```

```
Value* WhileStmtAST::generateCode(SLContext& Context) {
  LandingPadAST* prev = LandingPad->Prev;
  LandingPad->ReturnLoc = prev->ReturnLoc;

  // У нас есть оптимизация для варианта, если значение услови
  // цикла известно на этапе компиляции (но мы не производим
  // вычисление константных выражений вида 1 + 2, рассматривае
  // только выражения вида 1)
  if (Cond->isConst()) {
    // Если условие имеет значение "false", то не генерируем к
    // вообще
    if (!Cond->isTrue()) {
      return nullptr;
    }

    // Создаем новые блоки для тела цикла и выхода из цикла, и
    // устанавливаем их в качестве меток для "break" и "contin
    BasicBlock* bodyBB =
    LandingPad->ContinueLoc = BasicBlock::Create(
      getGlobalContext(),
      "loopbody",
      Context.TheFunction
    );
    BasicBlock* endBB = LandingPad->BreakLoc = BasicBlock::Cre
      getGlobalContext(),
      "loopend"
    );

    if (PostExpr) {
      // Для циклов, которые были сделаны из цикла "for" и у
      // которых есть блок для изменения переменных циклов, на
      // нужно создать новый блок для этого кода и мы должны
      // установить его в качестве метки "continue"
      LandingPad->ContinueLoc = BasicBlock::Create(
        getGlobalContext(),
        "postbody"
```

```
  );
}

// Создать безусловный переход на тело цикла и установить
// данный блок в качестве точки для генерации кода
Context.TheBuilder->CreateBr(bodyBB);
Context.TheBuilder->SetInsertPoint(bodyBB);

// Устанавливаем точку для возврата из цикла в зависимости
// того, есть ли блок для изменения переменных цикла или н
if (PostExpr) {
  LandingPad->FallthroughLoc = LandingPad->ContinueLoc;
} else {
  LandingPad->FallthroughLoc = bodyBB;
}

// Генерируем код тела цикла
Body->generateCode(Context);

// У нас есть специальная обработка для циклов "for" с бло
// для изменения переменных цикла
if (PostExpr) {
  // Добавляем блок в конец функции и устанавливаем его в
  // качестве точки генерации кода
  Context.TheFunction->getBasicBlockList().push_back(
    LandingPad->ContinueLoc
  );
  Context.TheBuilder->SetInsertPoint(LandingPad->ContinueL

  // Генерируем код для выражения изменения переменных цик
  PostExpr->getRValue(Context);
  // Генерируем безусловный переход на тело цикла
  Context.TheBuilder->CreateBr(bodyBB);
}

// Добавляем блок для возврата из цикла в конец функции и
// устанавливаем его в качестве точки генерации кода
```

```
  Context.TheFunction->getBasicBlockList().push_back(endBB);
  Context.TheBuilder->SetInsertPoint(endBB);
  prev->Returns += LandingPad->Returns;

  // Создаем безусловный переход на точку возврата родительс
  // блока и помечаем ее как использованную (путем присвоени
  // nullptr)
  Context.TheBuilder->CreateBr(prev->FallthroughLoc);
  prev->FallthroughLoc = nullptr;

  return nullptr;
}

// Создаем новые блоки для условия цикла, его тела и выхода
// него, и устанавливаем их в качестве меток для "break"
// и "continue"
BasicBlock* condBB = LandingPad->ContinueLoc = BasicBlock::C
  getGlobalContext(),
  "loopcond",
  Context.TheFunction
);
BasicBlock* bodyBB = BasicBlock::Create(
  getGlobalContext(),
  "loopbody"
);
BasicBlock* endBB = LandingPad->BreakLoc = BasicBlock::Creat
  getGlobalContext(),
  "loopend"
);

if (PostExpr) {
  // Для циклов, которые были сделаны из цикла "for" и у кот
  // есть блок для изменения переменных циклов, нам нужно со
  // новый блок для этого кода и мы должны установить его в
  // качестве метки "continue"
  LandingPad->ContinueLoc = BasicBlock::Create(
    getGlobalContext(),
```

```
    "postbody"
  );
}

// Создаем безусловный переход на блок с условием цикла и
// устанавливаем его в качестве точки генерации кода
Context.TheBuilder->CreateBr(condBB);
Context.TheBuilder->SetInsertPoint(condBB);

// Генерируем код для условия цикла
Value* cond = Cond->getRValue(Context);
// Произвести преобразование в "bool"
cond = promoteToBool(cond, Cond->ExprType, *Context.TheBuild
// Создать условный переход на тело цикла или на выход из ци
// в зависимости от истинности условия цикла
Context.TheBuilder->CreateCondBr(cond, bodyBB, endBB);

// Создать безусловный переход на тело цикла и установить да
// блок в качестве точки для генерации кода
Context.TheFunction->getBasicBlockList().push_back(bodyBB);
Context.TheBuilder->SetInsertPoint(bodyBB);

// Устанавливаем точку для возврата из цикла в зависимости с
// того, есть ли блок для изменения переменных цикла или нет
if (PostExpr) {
  LandingPad->FallthroughLoc = LandingPad->ContinueLoc;
} else {
  LandingPad->FallthroughLoc = condBB;
}

// Генерируем код тела цикла
Body->generateCode(Context);

// У нас есть специальная обработка для циклов "for" с блоко
// для изменения переменных цикла
if (PostExpr) {
  // Добавляем блок в конец функции и устанавливаем его в
```

```
    // качестве точки генерации кода
    Context.TheFunction->getBasicBlockList().push_back(
      LandingPad->ContinueLoc
    );
    Context.TheBuilder->SetInsertPoint(LandingPad->ContinueLoc

    // Генерируем код для выражения изменения переменных цикла
    PostExpr->getRValue(Context);
    // Генерируем безусловный переход на условие цикла
    Context.TheBuilder->CreateBr(condBB);
  }

  // Добавляем блок для возврата из цикла в конец функции и
  // устанавливаем его в качестве точки генерации кода
  Context.TheFunction->getBasicBlockList().push_back(endBB);
  Context.TheBuilder->SetInsertPoint(endBB);
  prev->Returns += LandingPad->Returns;

  // Создаем безусловный переход на точку возврата родительскс
  // блока и помечаем ее как использованную (путем присвоения
  // nullptr)
  Context.TheBuilder->CreateBr(prev->FallthroughLoc);
  prev->FallthroughLoc = nullptr;

  return nullptr;
}

llvm::Value* ForStmtAST::generateCode(SLContext& Context) {
  assert(0 && "ForStmtAST::semantic should never be reached");
  return nullptr;
}

llvm::Value* IfStmtAST::generateCode(SLContext& Context) {
  // Инициализируем блоки "break", "continue" и "return" для
  // LandingPadAST текущего блока на основе родительского блок
  LandingPadAST* prev = LandingPad->Prev;
  LandingPad->ReturnLoc = prev->ReturnLoc;
```

```
LandingPad->FallthroughLoc = prev->FallthroughLoc;
LandingPad->ContinueLoc = prev->ContinueLoc;
LandingPad->BreakLoc = prev->BreakLoc;

// У нас есть оптимизация для варианта, если значение услови
// известно на этапе компиляции (но мы не производим вычисле
// константных выражений вида 1 + 2, рассматриваем только
// выражения вида 1)
if (Cond->isConst()) {
  // Если выражение истинно
  if (Cond->isTrue()) {
    // Генерируем код для ветки "then"
    ThenBody->generateCode(Context);

    // Делаем подсчет количество "break" "continue" и "retur
    // инструкций в данном блоке (в последующих статьях буде
    // ясно зачем они нужны)
    prev->Returns += LandingPad->Returns;
    prev->Breaks += LandingPad->Breaks;
    prev->Continues += LandingPad->Continues;

    if (!Context.TheBuilder->GetInsertBlock()->getTerminator
      // Если еще нет условного или безусловного перехода, т
      // создаем безусловный переход на точку возврата
      // родительского блока и помечаем ее как использованну
      // (путем присвоения nullptr)
      Context.TheBuilder->CreateBr(prev->FallthroughLoc);
      prev->FallthroughLoc = nullptr;
    }

    return nullptr;
  }

  // Если выражение ложно и у нас есть ветка "else"
  if (ElseBody) {
    // Генерируем код для ветки "else"
    ElseBody->generateCode(Context);
```

```
    // Делаем подсчет количество "break" "continue" и "retur
    // инструкций в данном блоке (в последующих статьях буде
    // ясно зачем они нужны)
    prev->Returns += LandingPad->Returns;
    prev->Breaks += LandingPad->Breaks;
    prev->Continues += LandingPad->Continues;

    if (!Context.TheBuilder->GetInsertBlock()->getTerminator
      // Если еще нет условного или безусловного перехода, т
      // создаем безусловный переход на точку возврата
      // родительского блока и помечаем ее как использованну
      // (путем присвоения nullptr)
      Context.TheBuilder->CreateBr(prev->FallthroughLoc);
      prev->FallthroughLoc = nullptr;
    }
  }

  return nullptr;
}

// Генерируем код для условного выражения
Value* cond = Cond->getRValue(Context);
// Произвести преобразование в "bool"
cond = promoteToBool(cond, Cond->ExprType, *Context.TheBuild

// Создаем блок для веток "then" и блока за инструкцией ветв
BasicBlock* thenBB = BasicBlock::Create(
  getGlobalContext(),
  "thenpart",
  Context.TheFunction
);
BasicBlock* elseBB = nullptr;
BasicBlock* endBB = BasicBlock::Create(
  getGlobalContext(),
  "ifcont"
);
```

```
// Проверяем наличие ветки "else"
if (ElseBody) {
  // Создаем блок для ветки "else"
  elseBB = BasicBlock::Create(getGlobalContext(), "elsepart"
  // Создаем условный переход на "then" или "else" в зависим
  // от истинности условия
  Context.TheBuilder->CreateCondBr(cond, thenBB, elseBB);
} else {
  // Создаем условный переход на "then" или блок за инструкц
  // ветвления в зависимости от истинности условия
  Context.TheBuilder->CreateCondBr(cond, thenBB, endBB);
}

// Устанавливаем точку для генерации кода на блок "then"
Context.TheBuilder->SetInsertPoint(thenBB);

// Устанавливаем точку для FallthroughLoc
LandingPad->FallthroughLoc = endBB;

// Генерируем код для ветви "then"
ThenBody->generateCode(Context);

// Устанавливаем точку для FallthroughLoc (т. к. она могла б
// обнулена при генерации кода для ветки "then")
LandingPad->FallthroughLoc = endBB;

// Проверяем наличие ветки "else"
if (ElseBody) {
  // Добавляем блок "else" в конец функции и устанавливаем е
  // качестве точки генерации кода
  Context.TheFunction->getBasicBlockList().push_back(elseBB)
  Context.TheBuilder->SetInsertPoint(elseBB);

  // Генерируем код для ветви "else"
  ElseBody->generateCode(Context);
}
```

```
  // Проверяем был ли использован блок, который мы создали для
  // продолжения выполнения после инструкции ветвления
  if (endBB->hasNUsesOrMore(1)) {
    // Добавляем данный блок в конец функции и устанавливаем е
    // качестве точки генерации кода
    Context.TheFunction->getBasicBlockList().push_back(endBB);
    Context.TheBuilder->SetInsertPoint(endBB);

    // Создаем безусловный переход на точку возврата родительс
    // блока и помечаем ее как использованную (путем присвоени
    // nullptr)
    Context.TheBuilder->CreateBr(prev->FallthroughLoc);
    prev->FallthroughLoc = nullptr;
  } else {
    // Производим очистку
    delete endBB;
  }

  // Делаем подсчет количество "break" "continue" и "return"
  // инструкций в данном блоке (в последующих статьях будет
  // ясно зачем они нужны)
  prev->Returns += LandingPad->Returns;
  prev->Breaks += LandingPad->Breaks;
  prev->Continues += LandingPad->Continues;

  return nullptr;
}
```

Генерация кода для объявлений

Рассмотрим изменения которые необходимы для генерации кода для объявлений:

```
struct SymbolAST {
  /// Генерация объявления символа (например переменной)
  virtual llvm::Value *getValue(SLContext &Context);
  /// Генерация кода для символа (тела функции, инициализаторов и т.
  virtual llvm::Value *generateCode(SLContext &Context);
};
```

Рассмотрим саму генерацию кода для объявлений:

Hidden text

```
Value* SymbolAST::getValue(SLContext& ) {
  assert(0 && "SymbolAST::getValue should never be reached");
  return nullptr;
}

Value* SymbolAST::generateCode(SLContext& Context) {
  assert(0 && "SymbolAST::generateCode should never be reached
  return nullptr;
}

Value* VarDeclAST::generateCode(SLContext& Context) {
  assert(SemaState >= 5);
  // Получаем адрес переменной (т. к. память под саму переменн
  // уже должна была выделена ранее во время генерации кода дл
  // функции)
  Value* val = getValue(Context);


  // Если у нас есть выражение инициализации, то нужно ее
  // произвести
  if (Val) {
    // Генерация кода для инициализирующего выражения
```

```
    Value* init = Val->getRValue(Context);

    // Создание инструкции "store"
    return Context.TheBuilder->CreateStore(init, val);
  }

  return val;
}

Value* VarDeclAST::getValue(SLContext& Context) {
  // Генерируем код для объявления только один раз
  if (CodeValue) {
    return CodeValue;
  }

  // Создаем инструкцию "alloca" для переменной
  CodeValue = Context.TheBuilder->CreateAlloca(
    ThisType->getType(),
    nullptr,
    StringRef(Id->Id, Id->Length)
  );
  return CodeValue;
}

llvm::Value* ParameterSymbolAST::getValue(SLContext& Context)
  // Генерируем код для объявления только один раз
  if (Param->CodeValue) {
    return Param->CodeValue;
  }

  // Создаем инструкцию "alloca" для параметра функции
  Param->CodeValue = Context.TheBuilder->CreateAlloca(
    Param->Param->getType()
  );
  return Param->CodeValue;
}
```

```
llvm::Value* ParameterSymbolAST::generateCode(SLContext& Conte
  assert(SemaState >= 5);
  // Мы только генерируем адрес для переменной, все остальное
  // необходимости будет сгенерировано в  FuncDeclAST
  return getValue(Context);
}

Value* FuncDeclAST::getValue(SLContext& Context) {
  // Генерируем код для объявления только один раз
  if (CodeValue) {
    return CodeValue;
  }

  SmallString< 128 > str;
  raw_svector_ostream output(str);

  // Производим генерацию имени функции (для "main" у нас есть
  // специальная обработка)
  if (!(Id->Length == 4 && memcmp(Id->Id, "main", 4) == 0)) {
    // Генерация уникального декорированного имени функции
    output << "_P" << Id->Length << StringRef(Id->Id, Id->Leng
    ThisType->toMangleBuffer(output);
  } else {
    output << "main";
  }

  // Создать функцию с внешним видом связанности для данного
  // объявления
  CodeValue = Function::Create((FunctionType*)ThisType->getTyp
    Function::ExternalLinkage, output.str(), nullptr);
  Context.TheModule->getFunctionList().push_back(CodeValue);

  return CodeValue;
}

Value* FuncDeclAST::generateCode(SLContext& Context) {
  if (Compiled) {
```

```
  return CodeValue;
}

assert(SemaState >= 5);
// Создать объявление функции (ее прототип)
getValue(Context);

BasicBlock* oldBlock = Context.TheBuilder->GetInsertBlock();

Function::arg_iterator AI = CodeValue->arg_begin();
ParameterList::iterator PI =
  ((FuncTypeAST*)ThisType)->Params.begin();
ParameterList::iterator PE =
  ((FuncTypeAST*)ThisType)->Params.end();

// Создать блок "entry" для тела функции и установить его в
// качестве точки генерации кода
BasicBlock* BB = BasicBlock::Create(
  getGlobalContext(),
  "entry",
  CodeValue
);
Context.TheBuilder->SetInsertPoint(BB);

// Выделение памяти для всех объявленных переменных
for (std::vector< SymbolAST* >::iterator it = FuncVars.begir
  end = FuncVars.end(); it != end; ++it) {
  (*it)->getValue(Context);
}

// Произвести генерацию кода для всех параметров
for ( ; PI != PE; ++PI, ++AI) {
  ParameterAST* p = *PI;

  // Нам важны только именованные параметры
  if (p->Id) {
    // Указание имени параметра
```

```
    AI->setName(StringRef(p->Id->Id, p->Id->Length));
    // Генерация инструкции "alloca", если это необходимо
    if (!p->CodeValue) {
      p->CodeValue = Context.TheBuilder->CreateAlloca(
        p->Param->getType()
      );
    }
    // Генерируем инструкцию "store" для копирования значени
    // переданного в функцию в качестве параметра в переменн
    // выделенную под этот параметр
    Context.TheBuilder->CreateStore(AI, p->CodeValue);
  }
}

// Если тип возвращаемого значения функции отличается от "vo
// то выделяем память для переменной, которая будет хранить
// значение данной функции
if (!ReturnType->isVoid()) {
  LandingPad->ReturnValue = Context.TheBuilder->CreateAlloca
    ReturnType->getType(),
    nullptr,
    "return.value"
  );
}

// Создать блок для возврата из функции и установить его в
// качестве FallthroughLoc
LandingPad->ReturnLoc = BasicBlock::Create(
  getGlobalContext(),
  "return.block"
);
LandingPad->FallthroughLoc = LandingPad->ReturnLoc;

Function* oldFunction = Context.TheFunction;
Context.TheFunction = CodeValue;

// Генерация кода для тела функции
```

```
Body->generateCode(Context);

Context.TheFunction = oldFunction;

// Добавляем условный переход на блок выхода из функции, есл
// он еще не был сгенерирован
if (!Context.TheBuilder->GetInsertBlock()->getTerminator())
  Context.TheBuilder->CreateBr(LandingPad->ReturnLoc);
}

// Добавить блок для возврата из функции в конец функции и
// установить его в качестве точки генерации кода
CodeValue->getBasicBlockList().push_back(LandingPad->ReturnL
Context.TheBuilder->SetInsertPoint(LandingPad->ReturnLoc);

if (!ReturnType->isVoid()) {
  // Тип возвращаемого значение не "void". Создаем инструкци
  // "load" для загрузки возвращаемого значения
  Value* ret = Context.TheBuilder->CreateLoad(
    ReturnType->getType(),
    LandingPad->ReturnValue
  );
  // Генерация инструкции возврата из функции
  Context.TheBuilder->CreateRet(ret);
} else {
  // Генерация инструкции возврата из функции для "void"
  Context.TheBuilder->CreateRetVoid();
}

// Восстановление предыдущей точки генерации кода (если нужн
if (oldBlock) {
  Context.TheBuilder->SetInsertPoint(oldBlock);
}

// Проверка кода функции и оптимизация (если она включена)
verifyFunction(*CodeValue, &llvm::errs());
```

```
  Compiled = true;

  return CodeValue;
}

void ModuleDeclAST::generateCode() {
  SLContext& Context = getSLContext();

  // Генерируем код для всех объявлений
  for (SymbolList::iterator it = Members.begin(),
    end = Members.end(); it != end; ++it) {
    (*it)->generateCode(Context);
  }

  // Печать кода модуля на консоль
  Context.TheModule->dump();

  if (!OutputFilename.empty()) {
    std::error_code errorInfo;
    raw_fd_ostream fd(OutputFilename.c_str(), errorInfo);

    if (errorInfo) {
      llvm::errs()
        << "Can't write to \""
        << OutputFilename.c_str()
        << "\" file\n";
    }

    // Печать кода модуля в результирующий файл, если он был з
    Context.TheModule->print(fd, 0);
  }

  // Получить адрес JIT функции для "main", которую можно вызв
  // для запуска приложения
  MainPtr = (double (*)())(intptr_t)getJITMain();
}
```

Заключение

В данной части мы рассмотрели генерацию кода для LLVM IR, рассмотрели базовые концепции LLVM и LLVM IR. Так же были рассмотрены некоторые концепции, которые позволяют упростить генерацию кода и производить различные оптимизации на их основе.

Полный исходный код доступен на github. В следующих статьях мы будем расширять возможности подмножества нашего учебного языка и добавим конструкции более высокого уровня, такие как:

1. Указатели и массивы;
2. Структуры и классы;
3. Наследование и динамическая диспетчеризация методов объектов класса;
4. А так же перегрузку функций.

Полезные ссылки

1. https://llvm.org/docs/LangRef.html
2. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html
3. https://llvm.org/docs/tutorial/BuildingAJIT1.html

Теги: llvm, компиляторы, c++

Хабы: Open source, Программирование, Компиляторы

Создаем свой собственный язык программирования с использованием LLVM. Часть 4: Поддержка составных типов

4K

Open source*, Программирование*, Компиляторы*

В предыдущей статье мы закончили на том, что реализовали полностью законченное подмножество нашего учебного языка, в котором есть целые и вещественные числа, функции и множество управляющих конструкций, такие как: циклы, операторы ветвления и некоторые другие. В этой части мы продолжим расширять данный язык и добавим в него: строки, указатели, массивы и структуры, а так же операции для работы с ними.

Оглавление серии

1. Лексический и синтаксический анализ
2. Семантический анализ
3. Генерация кода
4. **Поддержка составных типов**
5. Поддержка классов и перегрузки функций

Введение

Для начала опишем, что именно мы хотим получить в результате.

1. Добавить два базовых типа "char" и "string";
2. Добавить поддержку двух новых типов констант:

- Строковые литералы (например "Hello, world");

- Символьные литералы (например 'd').

3. Массивы (например int[10]) и операции получения элемента массива по индексу;
4. Указатели (например int*) и операции над ними (такие как: ++/--, & и *), а так же поддержка индексации;
5. Структуры (например struct A { a: int; b: float; }), а так же возможность обращаться к членам структуры через оператор ".".

Для реализации всего этого нам нужно будет внести изменения во все 4 части нашего интерпретатора: лексический, синтаксический и семантический анализ, а так же в генерацию кода.

Примечание: В статье будут приведены только изменения, которые нужно внести для реализации всех вышеперечисленных компонентов, полный исходный код можно посмотреть в репозитории на github.

Примечание: В языке составные типы считаются с правой стороны, т. е. для "int*[2][3]" тип будет массив из 3 элементов типа массив из 2 элементов типа указателей на int. Что может быть не совсем удобным для использования в операторах индексирования, т. к. мы сможем записать a[2][1], но не a[1][2] (т. к. для второго индекса максимальное к-во элементов — 2, не 3).

Лексический и синтаксический анализ

Первое, что нам нужно сделать это добавить новые типы лексем в include/Basic/TokenKinds.def.

Новые ключевые слова:

```
KEYWORD(String, "string")
KEYWORD(Char, "char")
KEYWORD(Struct, "struct")
```

```
KEYWORD(New, "new")
KEYWORD(Delete, "del")
```

Новые литералы:

```
TOK(CharLiteral, "")
TOK(StringConstant, "")
```

Новые операторы:

```
TOK(Dot, ".")
TOK(OpenBrace, "[")
TOK(CloseBrace, "]")
```

Так же нам необходимо внести изменения в описание Name:

```
struct Name {
  ...
  static Name *New;    ///< ключевое слово "new"
  static Name *Delete; ///< ключевое слово "del"
};
```

Так же нам нужно инициализировать эти переменные в Lexer во время инициализации ключевых слов:

```
void NamesMap::addKeywords() {
  if (IsInit) {
    return;
  }
```

```
#define KEYWORD(NAME, TEXT)                                    \
  addName(StringRef(TEXT), tok::NAME);
#include "simple/Basic/TokenKinds.def"

  Name::New = getName("new");
  Name::Delete = getName("delete");

  IsInit = true;
}
```

Объявление этих переменных необходимы, т. к. в дальнейшем мы будем использовать их для того, что бы связать системные функции выделения блока памяти и освобождения блока памяти с этими именами, позже мы будем использовать их и для конструкторов и деструкторов классов.

Следующее, что нам нужно сделать, это добавить поддержку новых операторов и литералов в лексическом анализаторе (поддержка новых ключевых слов дополнительных доработок в нем не требует):

▾ Hidden text

```
void Lexer::next(Token &Result) {
  // Устанавливаем токен в качестве недействительного
  Result.Kind = tok::Invalid;

  // Сохраняем текущую позицию во входном потоке и будем
  // использовать эту позицию для чтения, что бы не испортить
  // состояние лексического анализатора
  const char *p = CurPos;

  // Считываем символы, пока не найдем корректный токен
  while (Result.Kind == tok::Invalid) {
    const char *tokenStart = p;
    ...
```

```
// Переходим к следующему символу, но запоминаем его значе
// смотрим что это за символ
switch (char ch = *p++) {
  ...
  // Анализ новых операторов состоящих из одного символа
  CHECK_ONE('[', tok::OpenBrace);
  CHECK_ONE(']', tok::CloseBrace);
  CHECK_ONE('.', tok::Dot);

  ...
  case '\'': {
    // Это символьный литерал
    char resultCh = ' ';
    // Проверяем на закрытие литерала, и если есть какой-л
    // символ, то сохраняем его
    if (*p != '\'') {
      resultCh = *p;
      ++p;
    }
    // Проверяем, что литерал имеет закрывающую '
    if (*p != '\'') {
      Diags.report(getLoc(p),
                   diag::ERR_UnterminatedCharOrString);
    }

    ++p;
    // Создаем токен на основе полученной информации
    Result.Length = (int)(p - tokenStart);
    Result.Chr = resultCh;
    Result.Kind = tok::CharLiteral;
    break;
  }

  case '"': {
    // Это строковый литерал
    llvm::SmallString<32> str;
    // Считываем, пока у нас во входном потоке еще есть да
```

```
while (*p != 0) {
  // Проверяем, что это "
  if (*p == '"') {
    // Проверяем, что это не "
    if (p[1] != '"') {
      // Заканчиваем чтение строкового литерала
      break;
    }

    // Заменяем "" на "
    p += 2;
    str.push_back('"');
  } else if (*p == '\\') {
    // Проверяем на эскейп последовательность
    ++p;
    // Обрабатываем только \n
    if (*p == 'n') {
      str.push_back('\n');
      ++p;
     // и \\
    } else if (*p == '\\') {
      str.push_back('\\');
      ++p;
    } else {
      Diags.report(getLoc(p),
                   diag::ERR_InvalidEscapeSequence);
    }
  } else if (*p == '\r' || *p == '\n') {
    // Мы встретили конец строки в литерале, сообщаем
    // ошибке
    Diags.report(getLoc(p), diag::ERR_NewLineInString)
  } else {
    // Добавляем символ к строке
    str.push_back(*p);
    ++p;
  }
}
```

```
        // Проверяем, что у нас есть завершающий символ "
        if (*p != '"') {
          Diags.report(getLoc(p),
                       diag::ERR_UnterminatedCharOrString);
        }
        // Пропускаем "
        ++p;
        // Создаем токен на основе полученной информации
        Result.Kind = tok::StringConstant;
        Result.Length = str.size();
        Result.Literal = new char[Result.Length + 1];
        memcpy(Result.Literal, str.c_str(), Result.Length);
        Result.Literal[Result.Length] = 0;
        break;
      }
      ...
    }
    Result.Ptr = tokenStart;
  }

  // Обновляем текущую позицию в лексическом анализаторе
  CurPos = p;
}
```

В синтаксическом анализаторе нам нужно будет добавить поддержку всех новых конструкций языка.

Поддержка новых литералов:

- Hidden text

```
/// primary-expr
///   ::= floating-point-literal
///   ::= integral-literal
///   ::= '.'? identifier
///   ::= char-literal
///   ::= string-literal
///   ::= '(' expr ')'
ExprAST *Parser::parsePrimaryExpr() {
  ExprAST *result = nullptr;

  switch (CurPos->getKind()) {
    ...
    case tok::CharLiteral:
      // Это символьная константа. Строим ветку дерева и счить
      // следующий токен
      result = new IntExprAST(CurPos.getLocation(),
                              CurPos->getChar());
      ++CurPos;
      return result;
    case tok::StringConstant:
      // Это строковая константа. Строим ветку дерева и считыв
      // следующий токен
      result = new StringExprAST(CurPos.getLocation(),
                                 CurPos->getLiteral());
      ++CurPos;
      return result;
    case tok::Dot:
      // Идентификаторы могут начинаться с ".", проверяем этот
      // случай
      if ((CurPos + 1) != tok::Identifier) {
        check(tok::Identifier);
      }
      // Сам идентификатор будет разобран в parsePostfixExpr,
      // нам нужно только указать, что это глобальная область
      // видимости
      return new IdExprAST(CurPos.getLocation(), nullptr);
```

```
      ...
  }
}
```

Так же необходимо добавить поддержку новых постфиксных операторов, такие как индексация в массиве и доступ к члену структуры:

▾ Hidden text

```
/// call-arguments
///   ::= assign-expr
///   ::= call-arguments ',' assign-expr
/// postfix-expr
///   ::= primary-expr
///   ::= postfix-expr '++'
///   ::= postfix-expr '--'
///   ::= postfix-expr '(' call-arguments ? ')'
///   ::= postfix-expr '[' expr ']'
///   ::= postfix-expr '.' identifier
ExprAST *Parser::parsePostfixExpr() {
  ExprAST *result = parsePrimaryExpr();

  for (;;) {
    llvm::SMLoc loc = CurPos.getLocation();

    switch (int op = CurPos->getKind()) {
      ...
      case tok::OpenBrace: {
        // Это индексация элемента в массиве. Пропускаем [
        check(tok::OpenBrace);
        // Считываем выражение
```

```
        ExprAST *expr = parseExpr();
        // Проверяем на наличие ]
        check(tok::CloseBrace);
        // Создаем ветку дерева
        result = new IndexExprAST(loc, result, expr);
        continue;
      }

      case tok::Dot: {
        // Это обращение к члену структуры или класса, их може
        // несколько, поэтому обрабатываем их в цикле
        while (CurPos == tok::Dot) {
          // Пропускаем .
          ++CurPos;

          Name *name = nullptr;
          // Если это идентификатор, то сохраняем его
          if (CurPos == tok::Identifier) {
            name = CurPos->getIdentifier();
          }
          // Считываем идентификатор или диагностируем об ошиб
          check(tok::Identifier);
          // Создаем ветку дерева
          result = new MemberAccessExprAST(loc, result, name);
        }

        continue;
      }

      default:
        return result;
    }
  }
}
```

В синтаксическом анализе выражений с одним операндом нам нужно добавить поддержку новых операторов, такие как: взятие адреса, разыменование указателей, а так же выделение и освобождение блока памяти:

Hidden text

```
/// unary-expr
///   ::= postfix-expr
///   ::= '+' unary-expr
///   ::= '-' unary-expr
///   ::= '++' unary-expr
///   ::= '--' unary-expr
///   ::= '~' unary-expr
///   ::= '!' unary-expr
///   ::= '*' unary-expr
///   ::= '&' unary-expr
///   ::= 'del' unary-expr
///   ::= 'new' type ('[' assign-expr ']')?
ExprAST *Parser::parseUnaryExpr() {
  ExprAST *result = nullptr;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (int op = CurPos->getKind()) {
    ...
    case tok::Mul:
      // Это разыменование указателя. Пропускаем "*" и считыва
      // следующее выражение с одним операндом
      ++CurPos;
      result = parseUnaryExpr();
      // Создаем ветку дерева
      return new DerefExprAST(loc, result);

    case tok::BitAnd:
      // Это операция взятия адреса. Пропускаем "&" и считывае
```

```
    // следующее выражение с одним операндом
    ++CurPos;
    result = parseUnaryExpr();
    // Создаем ветку дерева
    return new AddressOfExprAST(loc, result);

  case tok::Delete: {
    // Это операция освобождения памяти. Пропускаем "del" и
    // считываем следующее выражение с одним операндом
    ++CurPos;
    result = parsePostfixExpr();
    // Создаем ветку дерева
    return new DeleteExprAST(loc, result);
  }

  case tok::New: {
    // Это операция выделения памяти. Пропускаем "new"
    ++CurPos;
    // Считываем тип
    TypeAST *type = parseType();
    ExprList args;
    ExprAST *dynamicSize = nullptr;

    // Проверяем "[" expression "]" , т.к. "[" integral-lite
    // может быть обработано ранее в parseType
    if (CurPos == tok::OpenBrace) {
      // Проверяем, что слева от "[" у нас нету структуры ил
      // класса, т. к. мы их не поддерживаем
      if (isa<QualifiedTypeAST>(type)) {
        getDiagnostics().report(CurPos.getLocation(),
                                diag::ERR_DynArrayAggregate)
      }
      // Пропускаем "[" и считываем выражение между "[" и "]
      ++CurPos;
      dynamicSize = parseAssignExpr();
      // Проверяем наличие "]"
      check(tok::CloseBrace);
```

```
      }
      // Создаем ветку дерева
      return new NewExprAST(loc, type, dynamicSize, args);
    }

    default:
      return parsePostfixExpr();
  }
}
```

Так же нужно добавить поддержку новых типов в объявлениях:

▾ Hidden text

```
/// basic-type
///  ::= 'int'
///  ::= 'float'
///  ::= 'void'
///  ::= 'char'
///  ::= 'string'
///  ::= '.'? identifier ('.' identifier)*
/// type
///   ::= basic-type
///   ::= type '*'
///   ::= type '[' integral-literal ']'
TypeAST *Parser::parseType() {
  TypeAST *type = nullptr;
  bool isVoid = false;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (CurPos->getKind()) {
```

```
...

case tok::Char:
  // "char"
  ++CurPos;
  type = BuiltinTypeAST::get(TypeAST::TI_Char);
  break;

case tok::String:
  // "string"
  ++CurPos;
  type = BuiltinTypeAST::get(TypeAST::TI_String);
  break;

case tok::Dot:
case tok::Identifier: {
  // Это должно быть полное имя, например A.B или .A
  Name *name = nullptr;
  QualifiedName qualName;

  if (CurPos == tok::Identifier) {
    // Сохраняем идентификатор, если это был он (т. к. мог
    // быть ".")
    name = CurPos->getIdentifier();
    ++CurPos;
  }
  // Добавляем ранее считанный идентификатор или "." в спи
  qualName.push_back(name);
  // Полный идентификатор может состоять из серии
  // идентификаторов разделенных ".". Считываем их все
  while (CurPos == tok::Dot) {
    ++CurPos;

    // Сохраняем идентификатор для последующего использова
    if (CurPos == tok::Identifier) {
      name = CurPos->getIdentifier();
    }
```

```
      // Проверяем, что мы встретили идентификатор и переход
      // следующему токену
      check(tok::Identifier);
      // Добавляем идентификатор к полному имени
      qualName.push_back(name);
    }
    // Создаем ветку дерева
    type = new QualifiedTypeAST(qualName);
    break;
  }

  default:
    getDiagnostics().report(CurPos.getLocation(),
                            diag::ERR_InvalidType);
    return nullptr;
}
// После простого типа мы можем иметь составные типы
for (;;) {
  switch (CurPos->getKind()) {
    case tok::OpenBrace: {
      // Это "[", но исключаем вариант с "void["
      if (isVoid) {
        getDiagnostics().report(loc, diag::ERR_VoidAsNonPoir
        return nullptr;
      }
      // Пропускаем "["
      ++CurPos;

      int dim = 0;

      if (CurPos == tok::IntNumber) {
        // Это обычный целочисленный литерал. Сохраняем его
        // значение
        dim = atoi(CurPos->getLiteral().data());
      } else {
        // Считаем, что это "[" expression "]", которое може
        // считано позже или выдана ошибка. Поэтому мы
```

```
          // возвращаемся к предыдущему токену и завершаем раз
          // типа
          --CurPos;
          break;
        }
       // Считываем integral-literal и "]"
        check(tok::IntNumber);
        check(tok::CloseBrace);
       // Создаем ветку дерева
        type = new ArrayTypeAST(type, dim);
        continue;
      }

      case tok::Mul:
        // Это указатель. Переходим к следующему токену и созд
        // ветку дерева
        ++CurPos;
        type = new PointerTypeAST(type, false);
        continue;

      default:
        break;
    }

    break;
  }

  if (type == BuiltinTypeAST::get(TypeAST::TI_Void)) {
    getDiagnostics().report(loc, diag::ERR_VoidAsNonPointer);
    return nullptr;
  }

  return type;
}
```

В разборе прототипа функции нам нужно добавить поддержку двух специальных функций "new" и "del" (на данный момент они будут использоваться только для системных функций, но в дальнейшем их использование будет расширено):

Hidden text

```
/// parameter ::= identifier ':' type
/// parameters-list
///   ::= parameter
///   ::= parameter-list ',' parameter
/// return-type ::= ':' type
/// func-proto
///   ::= identifier '(' parameters-list ? ')' return-type ?
SymbolAST *Parser::parseFuncProto() {
  Name *name = nullptr;
  TypeAST *returnType = BuiltinTypeAST::get(TypeAST::TI_Void);
  ParameterList params;
  int Tok = CurPos->getKind();
  llvm::SMLoc loc = CurPos.getLocation();
  // При вызове этой функции текущий токен всегда "fn" (в даль
  // этот список будет расширен)
  ++CurPos;

  // После ключевого слова объявления функции всегда должен бы
  // идентификатор
  if (CurPos == tok::Identifier) {
    // Сохраняем имя функции для ее прототипа и переходим к
    // следующему токену
    name = CurPos->getIdentifier();
    ++CurPos;
  // Специальная обработка для "new" в качестве имени функции
  } else if (CurPos == tok::New) {
    ++CurPos;
    name = Name::New;
```

```
  // Специальная обработка для "del" в качестве имени функции
  } else if (CurPos == tok::Delete) {
    ++CurPos;
    name = Name::Delete;
  } else {
    check(tok::Identifier);
  }

  ...
}
```

В объявлениях мы должны добавить поддержку структур:

- Hidden text

```
/// decls
///  ::= func-decl
///  ::= decls func-decl
///  ::= struct-decl
///
/// struct-decl
///   ::= 'struct' identifier '{' decl-stmt * '}'
SymbolList Parser::parseDecls() {
  SymbolList result;
  // Производим анализ всех доступных объявлений
  for (;;) {
    SymbolList tmp;
    llvm::SMLoc loc = CurPos.getLocation();

    switch (int Tok = CurPos->getKind()) {
      ...
```

```
    case tok::Struct: {
      // Объявление структуры
      ++CurPos;
      Name *name = nullptr;

      // Сохраняем имя структуры для дальнейшего использован
      if (CurPos == tok::Identifier) {
        name = CurPos->getIdentifier();
      }

      // Проверяем наличие идентификатора и "{"
      check(tok::Identifier);
      check(tok::BlockStart);

      // Считываем все переменные, объявленные как ее члены
      while (CurPos != tok::BlockEnd) {
        SymbolList tmp2 = parseDecl(true, true);
        tmp.append(tmp2.begin(), tmp2.end());
      }

      // Проверяем наличие "}" и создаем ветку дерева
      check(tok::BlockEnd);
      result.push_back(new StructDeclAST(loc, name, tmp));
      continue;
    }

    case tok::EndOfFile:
      break;

    default:
      break;
  }

  break;
}
```

```
  return result;
}
```

В разборе объявлений переменных необходимо добавить поддержку того, что эти переменные могут быть объявлены как члены структур:

▾ Hidden text

```
/// var-init ::= '=' assign-expr
/// var-decl ::= identifier ':' type var-init?
/// var-decls
///   ::= var-decl
///   ::= var-decls ',' var-decl
/// decl-stmt ::= var-decls ';'
SymbolList Parser::parseDecl(bool needSemicolon,
                             bool isClassMember) {
  SymbolList result;
  // Производим анализ списка объявлений переменных
  for (;;) {
    llvm::SMLoc loc = CurPos.getLocation();

    if (CurPos != tok::Identifier) {
      // Объявление всегда должно начинаться с идентификатора,
      // это что-то другое, то сообщаем об ошибке
      getDiagnostics().report(CurPos.getLocation(),
                              diag::ERR_ExpectedIdentifierInDe
    } else {
      // Сохраняем имя переменной
      Name *name = CurPos->getIdentifier();
      ExprAST *value = nullptr;
      // Переходим к следующему токену
```

```
    ++CurPos;
    // Проверяем на наличие ":"
    check(tok::Colon);
    // Считываем тип переменной
    TypeAST *type = parseType();

    if (CurPos == tok::Assign) {
      // Запрещаем инициализаторы для массивов и структур/кл
      if (isa<ArrayTypeAST>(type) || isa<QualifiedTypeAST>(t
        getDiagnostics().report(
          CurPos.getLocation(),
          diag::ERR_AggregateOrArrayInitializer
        );
        return result;
      }

      // Если у переменной есть инициализатор, то считываем
      // инициализирующее значение
      ++CurPos;
      value = parseAssignExpr();
    }
    // Добавляем новую переменную к списку
    result.push_back(new VarDeclAST(loc, type, name, value,
                                    isClassMember));

    if (CurPos != tok::Comma) {
      // Если это не ",", то список объявлений завершен
      break;
    }
    // Пропускаем ","
    ++CurPos;
  }
}

if (needSemicolon) {
  // Если объявление должно заканчиваться ";", то проверяем
  // наличие
```

```
      check(tok::Semicolon);
    }

    return result;
  }
```

И последняя правка, которую нужно произвести это добавить поддержку новых операторов в разборе инструкций:

- Hidden text

Изменения в структуре AST

Для поддержки всех новых конструкций в языке для начала нужно внести некоторые правки в базовые класс корневых элементов AST. Для типов:

```
struct TypeAST {
  enum TypeId {
    TI_Void,     ///< void
    TI_Bool,     ///< булево значение
    TI_Int,      ///< int
    TI_Float,    ///< float
    TI_Char,     ///< char
    TI_String,   ///< string
    TI_Pointer,  ///< указатель
    TI_Array,    ///< массив
    TI_Struct,   ///< структура
    TI_Function, ///< функция
    TI_Qualified ///< квалифицированное имя (полное имя класса или ст
  };
  ...
```

```
  /// Проверка на то, что это тип строки
  bool isString() {
    return TypeKind == TI_String;
  }
  /// Проверка на то, что это тип символа
  bool isChar() {
    return TypeKind == TI_Char;
  }
  /// Проверка на то, что это тип структуры или класса
  bool isAggregate() {
    return TypeKind == TI_Struct;
  }

  /// Получить SymbolAST связанный с данным типом (доступен только дл
  ///   StructTypeAST)
  virtual SymbolAST* getSymbol();
  ...
};
```

Для выражений:

```
struct ExprAST {
  /// Expression's identifiers
  enum ExprId {
    EI_Int,          ///< целочисленные константы
    EI_Float,        ///< константы для чисел с плавающей точкой
    EI_Char,           ///< символьная константа
    EI_String,         ///< строковая константа
    EI_Proxy,          ///< прокси выражение
    EI_Id,           ///< использование переменной
    EI_Cast,         ///< преобразование типов
    EI_Index,          ///< операция индексирования
    EI_MemberAccess,  ///< доступ к члену структуры или класса
    /// доступ к члену структуры или класса, если экземпляр является
    /// указателем
```

```
    EI_PointerAccess,
    EI_Deref,          ///< разыменование указателя
    EI_AddressOf,      ///< операция взятия адреса
    EI_Unary,        ///< унарная операция
    EI_Binary,       ///< бинарная операция
    EI_Call,         ///< вызов функции
    EI_New,            ///< оператор выделение памяти
    EI_Delete,         ///< оператор очистки выделенной ранее памяти
    EI_Cond          ///< тернарный оператор
  };

  ...
  /// Проверка на то, что выражение является константой
  bool isConst() {
    return ExprKind == EI_Int
      || ExprKind == EI_Float
      || ExprKind == EI_String;
  }
  /// Получить SymbolAST соответствующий агрегату
  virtual SymbolAST *getAggregate(Scope *scope);
  /// Может ли значение быть null
  virtual bool canBeNull();
  ...
};
```

И для символов (объявлений):

```
struct SymbolAST {
  enum SymbolId {
    SI_Variable,     ///< переменная
    SI_Struct,       ///< структура
    SI_Function,     ///< функция
    SI_Module,       ///< модуль
    SI_Parameter,    ///< параметр функции
    SI_Block         ///< блочная декларация
```

```
  };
  ...
  /// Проверка на то, что символ является структурой или классом
  bool isAggregate() {
    return SymbolKind == SI_Struct;
  }
  /// Проверка на то, что необходим this (true - член class/struct)
  virtual bool needThis();
  /// Проверка на то, что данный символ содержит тип \c type
  /// в качестве дочерней сущности
  /// \note Позволяет обнаружить кольцевые ссылки
  virtual bool contain(TypeAST *type);
  /// Может ли данный символ быть null
  /// \note Должна возвращать true только для переменных с типом указ
  virtual bool canBeNull();
 ...
};
```

Дальше рассмотрим правки, которые нужно внести в описание некоторых ранее рассматриваемых веток дерева, а так же описание новых веток:

▾ Hidden text

```
struct BuiltinTypeAST : TypeAST {
  ...
  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Void ||
      T->TypeKind == TI_Bool ||
      T->TypeKind == TI_Int ||
      T->TypeKind == TI_Float ||
      T->TypeKind == TI_Char ||
      T->TypeKind == TI_String;
```

```
  }
  ...
};

/// Ветка дерева для типа "массив"
struct ArrayTypeAST : TypeAST {
  ArrayTypeAST(TypeAST* next, int dim) ;

  TypeAST* semantic(Scope* scope);
  bool implicitConvertTo(TypeAST* newType);
  void toMangleBuffer(llvm::raw_ostream& output);

  llvm::Type* getType();

  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Array;
  }

  TypeAST* Next; ///< базовый тип
  int Dim; ///< размер массива
};

/// Ветка дерева для типа "указатель"
struct PointerTypeAST : TypeAST {
  PointerTypeAST(TypeAST* next, bool isConst) ;

  TypeAST* semantic(Scope* scope);
  bool implicitConvertTo(TypeAST* newType);
  void toMangleBuffer(llvm::raw_ostream& output);

  llvm::Type* getType();

  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Pointer;
  }

  TypeAST* Next; ///< базовый тип
```

```
  bool IsConstant; ///< true - если значение является константой
};

/// Ветка дерева для типа "структура"
struct StructTypeAST : TypeAST {
  StructTypeAST(SymbolAST* thisDecl) ;

  TypeAST* semantic(Scope* scope);
  bool implicitConvertTo(TypeAST* newType);
  void toMangleBuffer(llvm::raw_ostream& output);
  SymbolAST* getSymbol();

  llvm::Type* getType();

  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Struct;
  }

  SymbolAST* ThisDecl; ///< объявление самой структуры
};

/// Квалифицированное имя
typedef llvm::SmallVector< Name*, 4 > QualifiedName;

/// Ветка дерева для квалифицированного имени
/// \note Вспомогательный класс, который будет заменен на тип
/// оригинал после семантического анализа (например на
/// StructTypeAST)
struct QualifiedTypeAST : TypeAST {
  QualifiedTypeAST(const QualifiedName& qualName);

  TypeAST* semantic(Scope* scope);
  bool implicitConvertTo(TypeAST* newType);
  void toMangleBuffer(llvm::raw_ostream& output);

  llvm::Type* getType();
```

```
  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Qualified;
  }

  QualifiedName QualName; ///< квалифицированное имя
};

/// Ветка дерева для строковой константы
struct StringExprAST : ExprAST {
  StringExprAST(llvm::SMLoc loc, llvm::StringRef str) ;

  bool isTrue();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_String;
  }

  llvm::SmallString< 32 > Val; ///< хранимое значение
};

/// Ветка для прокси выражения
/// \note Некоторые выражения могут копировать дочерние выраже
/// во время семантического анализа. Этот класс позволяет
/// заблокировать это поведение, путем копирования только себя
/// а дочернее выражение оставляет без изменений
struct ProxyExprAST : ExprAST {
  ProxyExprAST(llvm::SMLoc loc, ExprAST *orig) ;

  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getRValue(SLContext& Context);
```

```
  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_Proxy;
  }

  ExprAST* OriginalExpr; ///< вложенное выражение
};

/// Ветка дерева для оператора "new"
struct NewExprAST : ExprAST {
  NewExprAST(llvm::SMLoc loc, TypeAST *type, ExprAST *dynamicS
    const ExprList& args) ;

  ~NewExprAST();

  bool canBeNull();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_New;
  }

  TypeAST* NewType; ///< тип, под который нужно выделить памят
  ExprList Args; ///< аргументы для конструктора
  ExprAST* CallExpr; ///< вызов функции allocMem
  ExprAST* DynamicSize; ///< выражение с динамическим размером
  /// размер для выделения памяти одного элемента
  IntExprAST* SizeExpr;
  /// true – если необходимо произвести преобразование после
  /// выделения памяти
  bool NeedCast;
};

/// Ветка дерева для оператора "del"
struct DeleteExprAST : ExprAST {
```

```
  DeleteExprAST(llvm::SMLoc loc, ExprAST *val) ;

  ~DeleteExprAST();

  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_Delete;
  }

  ExprAST* Val; ///< выражение для очистки памяти
  ExprAST* DeleteCall; ///< вызов функции freeMem
};

struct IdExprAST : ExprAST {
  ...
  SymbolAST* getAggregate(Scope *scope);
  bool canBeNull();
  ...
};

/// Ветка дерева для выражения взятия индекса
struct IndexExprAST : ExprAST {
  IndexExprAST(llvm::SMLoc loc, ExprAST *left, ExprAST *right)

  ~IndexExprAST();

  bool isLValue();
  SymbolAST* getAggregate(Scope *scope);
  bool canBeNull();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getLValue(SLContext& Context);
```

```
  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_Index;
  }

  ExprAST* Left; ///< индексируемое значение
  ExprAST* Right; ///< индекс
};

/// Ветка дерева для доступа к члену класса/структуры
struct MemberAccessExprAST : ExprAST {
  MemberAccessExprAST(llvm::SMLoc loc, ExprAST *val,
                      Name *memberName) ;

  ~MemberAccessExprAST();

  bool isLValue();
  SymbolAST* getAggregate(Scope *scope);
  bool canBeNull();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getLValue(SLContext& Context);
  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_MemberAccess;
  }

  ExprAST* Val; ///< экземпляр класса/структуры
  Name* MemberName; ///< имя члена класса/структуры
  /// символ с объявлением члена класса/структуры
  SymbolAST* ThisSym;
  /// символ с объявлением класса/структуры
  SymbolAST* ThisAggr;
  bool SemaDone; ///< true – если семантический анализ был зав
```

```
};

/// Ветка дерева для доступа к члену класса/структуры через
/// указатель
struct PointerAccessExprAST : ExprAST {
  PointerAccessExprAST(llvm::SMLoc loc,
    ExprAST *val,
    Name *memberName) ;

  ~PointerAccessExprAST();

  bool isLValue();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getLValue(SLContext& Context);
  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_PointerAccess;
  }

  ExprAST* Val; ///< экземпляр класса/структуры
  Name* MemberName; ///< имя члена класса/структуры
  /// символ с объявлением члена класса/структуры
  SymbolAST* ThisSym;
};

/// Ветка дерева для разыменования указателя
struct DerefExprAST : ExprAST {
  DerefExprAST(llvm::SMLoc loc, ExprAST *val) ;

  ~DerefExprAST();

  bool isLValue();
  SymbolAST* getAggregate(Scope *scope);
  bool canBeNull();
```

```
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getLValue(SLContext& Context);
  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_Deref;
  }

  ExprAST* Val; ///< выражение для разыменования
};

/// Ветка дерева для операции взятия адреса
struct AddressOfExprAST : ExprAST {
  AddressOfExprAST(llvm::SMLoc loc, ExprAST *val) ;

  ~AddressOfExprAST();

  SymbolAST* getAggregate(Scope *scope);
  bool canBeNull();
  ExprAST* semantic(Scope* scope);
  ExprAST* clone();

  llvm::Value* getRValue(SLContext& Context);

  static bool classof(const ExprAST *E) {
    return E->ExprKind == EI_AddressOf;
  }

  ExprAST* Val; ///< выражение, адрес которого необходимо полу
};

/// Ветка для объявления переменной или члена класса/структурь
struct VarDeclAST : SymbolAST {
  VarDeclAST(llvm::SMLoc loc, TypeAST *varType, Name *id,
    ExprAST* value, bool inClass) ;
```

```
  ...
  bool needThis();
  bool canBeNull();
  bool contain(TypeAST* type);
  ...
  int OffsetOf; ///< индекс поля в классе/структуре
  bool NeedThis; ///< true – если это член класса/структуры
};

/// Ветка для объявления структуры
struct StructDeclAST : ScopeSymbol {
  StructDeclAST(llvm::SMLoc loc, Name *id,
                const SymbolList &vars) ;

  ~StructDeclAST();

  TypeAST* getType();
  void doSemantic(Scope* scope);
  void doSemantic3(Scope* scope);
  bool contain(TypeAST* type);

  llvm::Value* generateCode(SLContext& Context);
  llvm::Value* getValue(SLContext& Context);

  static bool classof(const SymbolAST *T) {
    return T->SymbolKind == SI_Struct;
  }

  SymbolList Vars; ///< список объявлений членов структуры
  TypeAST* ThisType; ///< тип данного объявления
};

/// Ветка для параметра функции
struct ParameterSymbolAST : SymbolAST {
  ...
  bool canBeNull();
```

```
  ...
};
```

Изменения в семантике для AST

Рассмотрим основные семантические правила, которые добавлены в язык с появлением новых типов и операций.

1. Тип char является однобайтным целочисленным типом, который представляет собой один символ в строке;
2. Строка — является указателем типа char, с тем отличием от обычных указателей, что элементы не могут быть изменены (поэтому поддерживает ограниченный набор операций над указателями). Переменные данного типа могут быть инициализированы строковыми литералами;
3. Указатель — является адресом на начала блока памяти и поддерживает операции ++/--, индексации, +/- с целочисленной константой, а так же операцию разыменование указателя *. Переменные данного типа могут быть инициализированы значением указателя или массива совместимого типа или константой 0. Указатель на char может быть преобразован в строку;
4. Массивы — блок памяти определенного размера, состоящих из элементов одного и того же типа. Поддерживают только операцию индексирования. Не могут иметь инициализаторов. Массив char может быть преобразован в строку. Данный тип не может быть использован в качестве возвращаемого значения для функции;
5. Структуры — именованный тип содержащий набор полей различных типов. Поддерживает только операцию обращения к собственным полям. Данный тип не может быть использован в качестве параметра и возвращаемого значения для функции. Также структуры не могут

иметь циклические ссылки (за исключением случая циклических ссылок через указатели);

6. Операция обращения к полям структуры может применяться к переменным с типом указателям на структуру без дополнительного разыменования указателя;
7. Массивы могут быть преобразованы к указателям, если они имеют общий базовый тип;
8. Операция взятия адреса может быть применена только к lvalue значениям;
9. Операции new и del могут быть использованы для выделения и очистки блоков памяти.

Далее рассмотрим изменения в семантики более подробно.

Семантика типов:

▾ Hidden text

```
SymbolAST* TypeAST::getSymbol() {
  return nullptr;
}

TypeAST *BuiltinTypeAST::get(int type) {
  static TypeAST *builtinTypes[] = {
    new BuiltinTypeAST(TI_Void),
    new BuiltinTypeAST(TI_Bool),
    new BuiltinTypeAST(TI_Int),
    new BuiltinTypeAST(TI_Float),
    new BuiltinTypeAST(TI_Char),
    new BuiltinTypeAST(TI_String)
  };

  assert(type >= TI_Void && type <= TI_String);
```

```
  return builtinTypes[type];
}

bool BuiltinTypeAST::implicitConvertTo(TypeAST* newType) {
  // Список разрешенных преобразований
  static bool convertResults[TI_String + 1][TI_String + 1] = {
    // void  bool   int    float  char   string
    { false, false, false, false, false, false }, // void
    { false, true,  true,  true,  true,  false }, // bool
    { false, true,  true,  true,  true,  false }, // int
    { false, true,  true,  true,  false, false }, // float
    { false, true,  true,  true,  true,  false }, // char
    { false, true,  false, false, false, true  }  // string
  };
  // Только базовые типы могут быть преобразованы друг в друга
  if (newType->TypeKind > TI_String) {
    return false;
  }

  return convertResults[TypeKind][newType->TypeKind];
}

void BuiltinTypeAST::toMangleBuffer(llvm::raw_ostream& output)
  switch (TypeKind) {
    case TI_Void : output << "v"; break;
    case TI_Bool : output << "b"; break;
    case TI_Int : output << "i"; break;
    case TI_Float : output << "f"; break;
    case TI_Char : output << "c"; break;
    case TI_String : output << "PKc"; break;
    default: assert(0 && "Should never happen"); break;
  }
}

TypeAST* ArrayTypeAST::semantic(Scope* scope) {
  // Проверить семантику базового типа
  Next = Next->semantic(scope);
```

```
  // создаем MangleName
  calcMangle();
  return this;
}

bool ArrayTypeAST::implicitConvertTo(TypeAST* newType) {
  TypeAST* next1 = Next;
  TypeAST* next2 = nullptr;

  // char[] можно преобразовать в string
  if (Next->isChar() && newType->isString()) {
    return true;
  }

  // Если newType не массив или указатель, то мы запрещаем
  // преобразование
  if (!(isa<PointerTypeAST>(newType)
        || isa<ArrayTypeAST>(newType))) {
    return false;
  }

  // Получаем базовый тип для указателей или массивов
  if (isa<PointerTypeAST>(newType)) {
    next2 = ((PointerTypeAST*)newType)->Next;
  } else if (isa<ArrayTypeAST>(newType)) {
    ArrayTypeAST* arrayType = (ArrayTypeAST*)newType;

    // Для массивов нужно проверить еще и размеры
    if (arrayType->Dim != Dim) {
      return false;
    }

    next2 = arrayType->Next;
  }

  // Разрешаем преобразование любого массива в void*
  if (next2->isVoid()) {
```

```
    return true;
  }

  // Допустимы только преобразования самого верхнего уровня
  return next1->equal(next2);
}

void ArrayTypeAST::toMangleBuffer(llvm::raw_ostream& output) {
  output << "A" << Dim << "_";
  Next->toMangleBuffer(output);
}

TypeAST* PointerTypeAST::semantic(Scope* scope) {
  // Проверить семантику базового типа
  Next = Next->semantic(scope);
  // создаем MangleName
  calcMangle();
  return this;
}

bool PointerTypeAST::implicitConvertTo(TypeAST* newType) {
  // Разрешаем преобразование указателя в bool
  if (newType->isBool()) {
    return true;
  }

  // Разрешаем преобразование char* в string
  if (Next->isChar() && newType->isString()) {
    return true;
  }

  // Запрещаем любое преобразование в тип отличный от указател
  if (!isa<PointerTypeAST>(newType)) {
    return false;
  }

  PointerTypeAST* ptr2 = (PointerTypeAST*)newType;
```

```
  // Получаем базовый тип для newType
  TypeAST* next2 = ptr2->Next;

  // Разрешаем преобразование, если типы совпадают
  if (Next->equal(next2)) {
    return true;
  }

  TypeAST* next1 = Next;

  // Разрешаем преобразование указателя любого типа в void*
  if (next2->isVoid()) {
    return true;
  }

  return false;
}

void PointerTypeAST::toMangleBuffer(llvm::raw_ostream& output)
  output << "P";
  Next->toMangleBuffer(output);
}

void mangleAggregateName(llvm::raw_ostream& output,
                         SymbolAST* thisSym) {
  assert(thisSym && thisSym->isAggregate());
  SymbolAST* sym = thisSym;

  output << "N";

  // Создаем строку вида N Длина Имя E для классов/структур ил
  // N Длина1 Имя1 Длина2 Имя2 E для вложенных классов/структу
  // Например:
  //   A.B.C будет N1A1B1CE
  //   A будет N1AE

  for ( ; ; ) {
```

```
    if (!sym || !sym->isAggregate()) {
      output << "E";
      return;
    }

    output << sym->Id->Length << StringRef(sym->Id->Id,
                                           sym->Id->Length);
    sym = sym->Parent;
  }
}

TypeAST* StructTypeAST::semantic(Scope* scope) {
  calcMangle();
  return this;
}

bool StructTypeAST::implicitConvertTo(TypeAST* newType) {
  return false;
}

void StructTypeAST::toMangleBuffer(llvm::raw_ostream& output)
  mangleAggregateName(output, ThisDecl);
}

SymbolAST* StructTypeAST::getSymbol() {
  return ThisDecl;
}

TypeAST* FuncTypeAST::semantic(Scope* scope) {
  if (!ReturnType) {
    // Если у функции не был задан тип возвращаемого значения,
    // установить как "void"
    ReturnType = BuiltinTypeAST::get(TypeAST::TI_Void);
  }

  // Произвести семантический анализ для типа возвращаемого
  // значения
```

```
  ReturnType = ReturnType->semantic(scope);

  // Произвести семантический анализ для всех параметров
  for (ParameterList::iterator it = Params.begin(),
       end = Params.end(); it != end; ++it) {
    (*it)->Param = (*it)→Param→semantic(scope);

    // Запрещаем использовать классы и структуры в качестве
    // параметров функции
    if ((*it)->Param->isAggregate()) {
      scope->report(SMLoc(),
                    diag::ERR_SemaAggregateAsFunctionParameter
      return nullptr;
    }
  }

  // Запрещаем использование массивов и классов/структур в кач
  // возвращаемого значения функций
  if (ReturnType && (ReturnType->isAggregate()
                     || isa<ArrayTypeAST>(ReturnType))) {
    scope->report(SMLoc(),
                  diag::ERR_SemaArrayOrAggregateAsReturnType);
    return nullptr;
  }

  calcMangle();
  return this;
}

TypeAST* QualifiedTypeAST::semantic(Scope* scope) {
  SymbolAST* sym = nullptr;

  // Мы должны найти объявление по полному имени
  for (QualifiedName::iterator it = QualName.begin(),
       end = QualName.end(); it != end; ++it) {

    if (sym) {
```

```
      // Ищем символ в качестве дочернего
      sym = sym->find((*it));

      if (!sym) {
        scope->report(SMLoc(), diag::ERR_SemaUndefinedIdentifi
                      (*it)->Id);
        return nullptr;
      }
    } else {
      // Ищем имя в текущей области видимости
      sym = scope->find((*it));

      if (!sym) {
        scope->report(SMLoc(), diag::ERR_SemaUndefinedIdentifi
                      (*it)->Id);
        return nullptr;
      }
    }
  }

  // Возвращаем тип найденного символа
  return sym->getType();
}

bool QualifiedTypeAST::implicitConvertTo(TypeAST* ) {
  assert(0 && "QualifiedTypeAST::implicitConvertTo should neve
         " be reached");
  return false;
}

void QualifiedTypeAST::toMangleBuffer(llvm::raw_ostream& ) {
  assert(0 && "QualifiedTypeAST::toMangleBuffer should never b
         "reached");
}
```

Семантика выражений:

Hidden text

```
SymbolAST* ExprAST::getAggregate(Scope *) {
  assert(0 && "ExprAST::getAggregaet should never be reached")
  return nullptr;
}

bool ExprAST::canBeNull() {
  return false;
}

bool StringExprAST::isTrue() {
  return true;
}

ExprAST* StringExprAST::semantic(Scope* ) {
  return this;
}

ExprAST* StringExprAST::clone() {
  return new StringExprAST(Loc, StringRef(Val));
}

ExprAST* ProxyExprAST::semantic(Scope* scope) {
  // Проверяем, что было использовано для выражения, которое н
  // заменяет себя после семантического анализа
  ExprAST* tmp = OriginalExpr->semantic(scope);
  assert(tmp == OriginalExpr);
  ExprType = OriginalExpr->ExprType;
  return this;
}

ExprAST* ProxyExprAST::clone() {
```

```
  return new ProxyExprAST(Loc, OriginalExpr);
}

bool NewExprAST::canBeNull() {
  return true;
}

ExprAST* NewExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ для типа
  NewType = NewType->semantic(scope);

  // Проверяем на то, что это массив с фиксированным размером
  if (isa<ArrayTypeAST>(NewType) && !DynamicSize) {
    // Мы выделяем память как для массива, но результирующий т
    // будет указателем
    ArrayTypeAST* arrayType = (ArrayTypeAST*)NewType;
    ExprType = new PointerTypeAST(arrayType->Next, false);
    ExprType = ExprType->semantic(scope);
  } else {
    // Результирующий тип будет указателем
    ExprType = new PointerTypeAST(NewType, false);
    ExprType = ExprType->semantic(scope);
  }

  // Создаем новое выражение для хранения размера NewType на
  // целевой платформе и оборачиваем его в прокси, т. к.
  // реальный размер будет известен только во время генерации
  SizeExpr = new IntExprAST(Loc, 0);
  ExprAST* proxy = new ProxyExprAST(Loc, SizeExpr);

  // Для массивов с динамическим размером, мы должны посчитать
  // актуальный размер
```

```
  if (DynamicSize) {
    proxy = new BinaryExprAST(Loc, tok::Mul, DynamicSize, prox
  }

  // Создаем вызов функции allocMem для выделения блока памяти
  // нужного размера
  ExprList args;
  args.push_back(proxy);
  CallExpr = new CallExprAST(Loc, new IdExprAST(Loc, Name::New
                             args);

  // Производим семантический анализ только что созданного выз
  // функции
  CallExpr = CallExpr->semantic(scope);

  return this;
}

ExprAST* NewExprAST::clone() {
  ExprList exprs;
  ExprList::iterator it = Args.begin();
  ExprList::iterator end = Args.end();

  // Делаем клонирование всех аргументов (если они есть)
  for ( ; it != end; ++it) {
    exprs.push_back((*it)->clone());
  }

  return new NewExprAST(Loc, NewType, DynamicSize, exprs);
}

ExprAST* DeleteExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (DeleteCall) {
    return this;
  }
```

```
  // Производим семантический анализ выражения для очистки
  Val = Val->semantic(scope);

  // Запрещаем удаление выражений с типом void
  if (!Val->ExprType || Val->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaCantDeleteVoid);
    return nullptr;
  }

  // Мы можем производить очистку только для указателей
  if (!isa<PointerTypeAST>(Val->ExprType)) {
    scope->report(Loc, diag::ERR_SemaCantDeleteNonPointer);
    return nullptr;
  }

  PointerTypeAST* ptrType = (PointerTypeAST*)Val->ExprType;

  ExprAST* deleteArg = Val->clone();
  ExprList args;

  // Создаем список аргументов для вызова freeMem
  // Замечание: Это может быть либо Val, либо результат вызова
  // деструктора
  args.push_back(deleteArg);

  // Создание вызова freeMem и проверка его семантики
  DeleteCall = new CallExprAST(
    Loc,
    new IdExprAST(Loc, Name::Delete),
    args
  );
  DeleteCall = DeleteCall->semantic(scope);
  // Результат операции будет void
  ExprType = BuiltinTypeAST::get(TypeAST::TI_Void);

  return this;
}
```

```
ExprAST* DeleteExprAST::clone() {
  return new DeleteExprAST(Loc, Val->clone());
}

bool IdExprAST::isLValue() {
  // Исключаем неизменяемые указатели, т. к. мы их не можем
  // изменить
  if (isa<PointerTypeAST>(ExprType)) {
    return !((PointerTypeAST*)ExprType)->IsConstant;
  }

  return true;
}

SymbolAST* IdExprAST::getAggregate(Scope *) {
  return ThisSym;
}

bool IdExprAST::canBeNull() {
  return ThisSym->canBeNull();
}

bool IndexExprAST::isLValue() {
  // Мы не можем изменять строки
  if (Left->ExprType->isString()) {
    return false;
  }

  return true;
}

SymbolAST* IndexExprAST::getAggregate(Scope *scope) {
  // Мы не можем получить символ для агрегата, если это не
  // структура или класс
  if (!ExprType->isAggregate()) {
    scope->report(Loc, diag::ERR_SemaNonAggregateDotOperand);
```

```
    return nullptr;
  }
  // Возвращаем объявление выражения
  return ExprType->getSymbol();
}

bool IndexExprAST::canBeNull() {
  // Если тип результирующего выражения указатель, то он может
  // быть null
  if (isa<PointerTypeAST>(ExprType)) {
    return true;
  }

  return false;
}

ExprAST* IndexExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ обоих операндов
  Left = Left->semantic(scope);
  Right = Right->semantic(scope);

  // Мы можем индексировать только массивы, указатели и строки
  if (Left->ExprType && !isa<ArrayTypeAST>(Left->ExprType)
    && !isa<PointerTypeAST>(Left->ExprType)
    && !Left->ExprType->isString()) {
    scope->report(Loc, diag::ERR_SemaNonIndexableType);
    return nullptr;
  }

  // Произвести преобразование типа индекса к int, если это
  // необходимо
  if (!Right->ExprType->isInt()) {
```

```
    Right = new CastExprAST(
      Right->Loc,
      Right,
      BuiltinTypeAST::get(TypeAST::TI_Int)
    );
    Right = Right->semantic(scope);
  }

  // Определяем тип результата работы выражения
  if (isa<ArrayTypeAST>(Left->ExprType)) {
    ExprType = ((ArrayTypeAST*)Left->ExprType)->Next;
  } else if (isa<PointerTypeAST>(Left->ExprType)) {
    ExprType = ((PointerTypeAST*)Left->ExprType)->Next;
  } else {
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Char);
  }

  return this;
}

ExprAST* IndexExprAST::clone() {
  return new IndexExprAST(Loc, Left->clone(), Right->clone());
}

bool MemberAccessExprAST::isLValue() {
  return true;
}

SymbolAST* MemberAccessExprAST::getAggregate(Scope *) {
  return ThisSym;
}

bool MemberAccessExprAST::canBeNull() {
  return Val->canBeNull();
}

ExprAST* MemberAccessExprAST::semantic(Scope* scope) {
```

```
// Исключаем повторный запуск семантического анализа
if (SemaDone) {
  return this;
}

// Проверяем семантику левого операнда
Val = Val->semantic(scope);

// Специальная обработка для символа в глобальной области
// видимости
if (isa<IdExprAST>(Val) && !((IdExprAST*)Val)->Val) {
  // Получить объявление модуля
  SymbolAST* moduleSym = scope->find(0);
  assert(moduleSym);
  // Поиск объявления в модуле
  SymbolAST* thisSym = moduleSym->find(MemberName);

  // Диагностика ошибки, если символ не найден
  if (!thisSym) {
    scope->report(Loc, diag::ERR_SemaUndefinedMember,
                  MemberName->Id);
    return nullptr;
  }

  // Заменяем MemberAccessExprAST на IdExprAST с предопредел
  // значениями
  IdExprAST* newExpr = new IdExprAST(Loc, MemberName);

  newExpr->ExprType = thisSym->getType();

  newExpr->ThisSym = thisSym;
  delete this;
  return newExpr;
}

// Проверяем на корректность типа операнда
if (!Val->ExprType) {
```

```
    scope->report(Loc, diag::ERR_SemaInvalidOperandForMemberAc
    return nullptr;
  }

  // Если Val является указателем на агрегат, то нам нужно
  // произвести разыменование указателя. Т.е. в семантики C++,
  // мы считаем agg->a эквивалентом (*agg).a
  if (isa<PointerTypeAST>(Val->ExprType) &&
    ((PointerTypeAST*)Val->ExprType)->Next->isAggregate()) {
    // Создание разыменования указателя и проверка его семанти
    Val = new DerefExprAST(Loc, Val);
    Val = Val->semantic(scope);
  }

  // Val должно быть lvalue
  if (!Val->isLValue()) {
    scope->report(Loc, diag::ERR_SemaNonLValueForMemberAccess)
    return nullptr;
  }

  // Получить агрегат
  SymbolAST* aggSym = Val->getAggregate(scope);

  // Запрещаем использование A.b, где A — класс или структура.
  // Это должно быть a.b, где a экземпляр A
  if (isa<IdExprAST>(Val) && aggSym && aggSym->isAggregate())
    scope->report(Loc, diag::ERR_SemaMemberAccessOnAggregateTy
    return nullptr;
  }

  // Исключаем операции над не агрегатами
  if (!aggSym || !aggSym->getType()->isAggregate()) {
    scope->report(Loc, diag::ERR_SemaNonAggregateDotOperand);
    return nullptr;
  }

  // Получить объявление агрегата
```

```
  if (!aggSym->isAggregate()) {
    aggSym = aggSym->getType()->getSymbol();
  }

  // Поиск символа в агрегате
  ThisSym = aggSym->find(MemberName);
  ThisAggr = aggSym;

  // Проверка на то, что данный агрегат имеет член с таким име
  if (!ThisSym) {
    scope->report(Loc, diag::ERR_SemaUndefinedMember,
                  MemberName->Id);
    return nullptr;
  }

  // Запрет конструкции вида a.A, где A это агрегат
  if (ThisSym->isAggregate()) {
    scope->report(Loc,
                  diag::ERR_SemaAggregateTypeAsMemberAccessOpe
    return nullptr;
  }

  // Установка типа результата операции
  ExprType = ThisSym->getType();
  SemaDone = true;

  return this;
}

ExprAST* MemberAccessExprAST::clone() {
  return new MemberAccessExprAST(Loc, Val->clone(), MemberName
}

bool PointerAccessExprAST::isLValue() {
  return true;
}
```

```
ExprAST* PointerAccessExprAST::semantic(Scope* scope) {
  // Переписываем expr->member в виде (*expr).member, производ
  // семантический анализ и удаляем старое выражение
  ExprAST* newExpr = new DerefExprAST(Loc, Val);
  newExpr = new MemberAccessExprAST(Loc, newExpr, MemberName);
  Val = nullptr;
  delete this;
  return newExpr->semantic(scope);
}

ExprAST* PointerAccessExprAST::clone() {
  return new PointerAccessExprAST(Loc, Val->clone(), MemberNam
}

bool DerefExprAST::isLValue() {
  return true;
}

SymbolAST* DerefExprAST::getAggregate(Scope *scope) {
  // Мы не можем получить символ для агрегата, если это не
  // структура или класс
  if (!ExprType->isAggregate()) {
    scope->report(Loc, diag::ERR_SemaNonAggregateDotOperand);
    return nullptr;
  }
  // Возвращаем объявление выражения
  return ExprType->getSymbol();
}

bool DerefExprAST::canBeNull() {
  return true;
}

ExprAST* DerefExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (ExprType) {
    return this;
```

```
}

// Проверяем семантику операнда
Val = Val->semantic(scope);

// Проверяем, что это операция взятия адреса
if (isa<AddressOfExprAST>(Val)) {
  // Удаляем * и &
  ExprAST* result = ((AddressOfExprAST*)Val)->Val->clone();
  delete this;
  return result->semantic(scope);
}

// Проверяем на корректность типов
if (Val->ExprType) {
  // Специальный случай для строк
  if (Val->ExprType->isString()) {
    // Запрет разыменования константных строк
    if (Val->isConst()) {
      scope->report(Loc, diag::ERR_SemaConstStringDeRef);
      return nullptr;
    }
    // Устанавливаем тип результата операции, как char
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Char);
    return this;
  // Запрещаем работу с не указателями
  } else if (!isa<PointerTypeAST>(Val->ExprType)) {
    scope->report(Loc, diag::ERR_SemaUnDereferencableType);
    return nullptr;
  }
} else {
  scope->report(Loc, diag::ERR_SemaUnDereferencableType);
  return nullptr;
}
// Установка типа для результата операции
ExprType = ((PointerTypeAST*)Val->ExprType)->Next;
return this;
```

```
}

ExprAST* DerefExprAST::clone() {
  return new DerefExprAST(Loc, Val->clone());
}

SymbolAST* AddressOfExprAST::getAggregate(Scope *scope) {
  return Val->getAggregate(scope);
}

bool AddressOfExprAST::canBeNull() {
  // Специальная проверка для идентификаторов
  if (isa<IdExprAST>(Val) && !Val->canBeNull()) {
    return false;
  }

  return true;
}

ExprAST* AddressOfExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (ExprType) {
    return this;
  }

  // Проверяем семантику для операнда
  Val = Val->semantic(scope);

  // Проверяем корректность типа операнда
  if (!Val->ExprType) {
    scope->report(Loc, diag::ERR_SemaAddressExpressionNoType);
    return nullptr;
  }

  // Проверяем на разыменование
  if (isa<DerefExprAST>(Val)) {
    // Удаляем & и *
```

```
    ExprAST* result = ((DerefExprAST*)Val)->Val->clone();
    delete this;
    return result->semantic(scope);
  }

  // Мы можем взять адрес только у lvalue
  if (!Val->isLValue()) {
    scope->report(Loc, diag::ERR_SemaAddressOfNonLValue);
    return nullptr;
  }

  // Тип выражения будет указателем на тип операнда
  ExprType = new PointerTypeAST(Val->ExprType, false);
  ExprType = ExprType->semantic(scope);

  return this;
}

ExprAST* AddressOfExprAST::clone() {
  return new AddressOfExprAST(Loc, Val->clone());
}

ExprAST* CastExprAST::semantic(Scope* scope) {
  ...

  // Запрещаем преобразование в массив
  if (isa<ArrayTypeAST>(ExprType)) {
    scope->report(Loc, diag::ERR_SemaArrayInCast);
    return nullptr;
  }

  // Проверяем на указатель и их совместимость
  if (isa<PointerTypeAST>(ExprType) &&
    !Val->ExprType->implicitConvertTo(ExprType)) {
    scope->report(Loc, diag::ERR_SemaIncompatiblePointerTypes)
    return nullptr;
  }
```

```
  // Проверяем, что типы совместимы
  if (!Val->ExprType->implicitConvertTo(ExprType)) {
    scope->report(Loc, diag::ERR_SemaInvalidCast);
    return nullptr;
  }

  SemaDone = true;
  return this;
}

ExprAST* UnaryExprAST::semantic(Scope* scope) {
  // Проверяем, что операнд был корректно задан
  if (!Val) {
    assert(0 && "Invalid expression value");
    return nullptr;
  }

  // Производим семантический анализ для операнда
  Val = Val->semantic(scope);

  // Проверяем корректность типа операнда, т. к. он может быть
  // "void"
  if (!Val->ExprType || Val->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaOperandIsVoid);
    return nullptr;
  }

  // Запрещаем операции на классами и структурами
  if (Val->ExprType->isAggregate()) {
    scope->report(Loc, diag::ERR_SemaAggregateAsExpression);
    return nullptr;
  }

  // Для строк разрешаем только ++/--
  if (Val->ExprType->isString() && (Op != tok::MinusMinus
                                    && Op != tok::PlusPlus)) {
```

```
    scope->report(Loc,
                  diag::ERR_SemaInvalidUnaryExpressionForStrin
    return nullptr;
  }

  // Запрещаем операции над массивами
  if (isa<ArrayTypeAST>(Val->ExprType)) {
    scope->report(Loc,
                  diag::ERR_SemaInvalidUnaryExpressionForArray
    return nullptr;
  }

  // Для указателей разрешаем только ++/--
  if (isa<PointerTypeAST>(Val->ExprType)
    && (Op != tok::MinusMinus && Op != tok::PlusPlus)) {
    scope->report(Loc,
                  diag::ERR_SemaInvalidUnaryExpressionForPoint
    return nullptr;
  }

  // Исключаем операции над булевыми значениями
  if (Val->ExprType->isBool() && (Op == tok::Plus
                                  || Op == tok::Minus)) {
    scope->report(Loc, diag::ERR_SemaInvalidBoolForUnaryOperan
    return nullptr;
  }

  // Список замен:
  //  +Val to Val
  //  -Val to 0 - Val
  //  ~intVal to intVal ^ -1
  //  ++id to id = id + 1
  //  --id to id = id - 1
  //  !Val to Val == 0

  ExprAST* result = this;
```

```
// Проверяем тип оператора, для необходимой замены
switch (Op) {
  // "+" - noop
  case tok::Plus: result = Val; break;

  case tok::Minus:
    // Преобразовываем в 0 - Val с учетом типа Val
    if (Val->ExprType->isFloat()) {
      result = new BinaryExprAST(
        Val->Loc,
        tok::Minus,
        new FloatExprAST(Val->Loc, 0),
        Val
      );
    } else {
      result = new BinaryExprAST(
        Val->Loc,
        tok::Minus,
        new IntExprAST(Val->Loc, 0),
        Val
      );
    }
    break;

  case tok::Tilda:
    // ~ можно применять только к целочисленным выражениям
    if (!Val->ExprType->isInt()) {
      scope->report(Loc,
                    diag::ERR_SemaInvalidOperandForComplemet
      return nullptr;
    } else {
      // Преобразуем в Val ^ -1
      result = new BinaryExprAST(
        Val->Loc,
        tok::BitXor,
        Val,
        new IntExprAST(Val->Loc, -1)
```

```
      );
      break;
    }

  case tok::PlusPlus:
  case tok::MinusMinus:
    if ((!Val->ExprType->isInt()
      && !isa<PointerTypeAST>(Val->ExprType)
      && !Val->ExprType->isString()) || !Val->isLValue()) {
      scope->report(Loc,
                    diag::ERR_SemaInvalidPostfixPrefixOperan
      return nullptr;
    } else {
      // Специальная обработка для IndexExprAST и MemberAcce
      // и строк
      if (isa<IndexExprAST>(Val) || isa<MemberAccessExprAST>
        || Val->ExprType->isString()) {
        ExprType = Val->ExprType;
        return this;
      }
      // Специальная обработка для указателей
      if (isa<PointerTypeAST>(Val->ExprType)) {
        // Заменяем ++ ptr или -- ptr на ptr = &ptr[1] или
        // ptr = &ptr[-1]
        ExprAST* val = Val;
        ExprAST* valCopy = Val->clone();
        result = new BinaryExprAST(
          Val->Loc,
          tok::Assign,
          val,
          new AddressOfExprAST(
            Val->Loc,
            new IndexExprAST(
              Val->Loc,
              valCopy,
              new IntExprAST(
                Val->Loc,
```

```
                (Op == tok::PlusPlus) ? 1 : -1
              )
            )
          )
        );
      } else {
        // Необходимо заменить "++" id или "--" id на id = i
        // or id = id + -1
        ExprAST* val = Val;
        ExprAST* valCopy = Val->clone();
        result = new BinaryExprAST(
          Val->Loc,
          tok::Assign,
          val,
          new BinaryExprAST(
            Val->Loc,
            tok::Plus,
            valCopy,
            new IntExprAST(
              Val->Loc,
              (Op == tok::PlusPlus) ? 1 : -1
            )
          )
        );
      }
    }
    break;

  case tok::Not:
    // Заменяем на Val == 0
    result = new BinaryExprAST(
      Val->Loc,
      tok::Equal,
      Val,
      new IntExprAST(Val->Loc, 0)
    );
    break;
```

```
    default:
      // Никогда не должно произойти
      assert(0 && "Invalid unary expression");
      return nullptr;
  }

  if (result != this) {
    // Т.к. старое выражение было заменено, очищаем память и
    // производим семантический анализ нового выражения
    Val = nullptr;
    delete this;
    return result->semantic(scope);
  }

  return result;
}

ExprAST* BinaryExprAST::semantic(Scope* scope) {
  // Семантический анализ уже был произведен ранее
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ операнда с левой стороны
  LeftExpr = LeftExpr->semantic(scope);

  // Проверяем на "++" или "--", т. к. эти операции имеют толь
  // один операнд
  if (Op == tok::PlusPlus || Op == tok::MinusMinus) {
    // Проверка на корректности типов для операнда ++/--
    if (!LeftExpr->isLValue() || (!LeftExpr->ExprType->isInt()
      && !isa<PointerTypeAST>(LeftExpr->ExprType)
      && !LeftExpr->ExprType->isString())) {
      scope->report(Loc, diag::ERR_SemaInvalidPostfixPrefixOpe
      return nullptr;
    }
```

```
  // Устанавливаем результирующий тип выражения
  ExprType = LeftExpr->ExprType;
  return this;
}

// Производим семантический анализ операнда с правой стороны
RightExpr = RightExpr->semantic(scope);

// Проверяем, что оба операнда имеют корректные типы
if (!LeftExpr->ExprType || !RightExpr->ExprType) {
  scope->report(Loc,
                diag::ERR_SemaUntypedBinaryExpressionOperand
  return nullptr;
}

// Запрещаем работу с агрегатами
if (LeftExpr->ExprType->isAggregate()
    || RightExpr->ExprType->isAggregate()) {
  scope->report(Loc, diag::ERR_SemaAggregateAsExpression);
  return nullptr;
}

// "," имеет специальную обработку и тип выражения совпадает
// типом операнда с правой стороны
if (Op == tok::Comma) {
  ExprType = RightExpr->ExprType;
  return this;
}

// Исключаем операции, если хотя бы один операнд имеет тип "
if (LeftExpr->ExprType->isVoid()
  || RightExpr->ExprType->isVoid()) {
  scope->report(Loc, diag::ERR_SemaOperandIsVoid);
  return nullptr;
}
```

```
// Проверка на операторы сравнения, т. к. их результат всегд
// будет иметь тип "bool"
switch (Op) {
  case tok::Less:
  case tok::Greater:
  case tok::LessEqual:
  case tok::GreaterEqual:
  case tok::Equal:
  case tok::NotEqual:
    // Запрет использование строк, указателей и массивов в
    // качестве операндов для операций сравнения
    if (LeftExpr->ExprType->isString()
      || isa<PointerTypeAST>(LeftExpr->ExprType)
      || isa<ArrayTypeAST>(LeftExpr->ExprType)
      || RightExpr->ExprType->isString()
      || isa<PointerTypeAST>(RightExpr->ExprType)
      || isa<ArrayTypeAST>(RightExpr->ExprType)) {
      scope->report(Loc, diag::ERR_SemaInvalidTypeForCompari
      return nullptr;
    }

    // Если левый операнд "bool", то сначала конвертируем ег
    // "int"
    if (LeftExpr->ExprType->isBool()) {
      LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
        BuiltinTypeAST::get(TypeAST::TI_Int));
      LeftExpr = LeftExpr->semantic(scope);
    }

    // Если левый операнд имеет тип char, то преобразовываем
    // в int
    if (LeftExpr->ExprType->isChar()) {
      LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
        BuiltinTypeAST::get(TypeAST::TI_Int));
      LeftExpr = LeftExpr->semantic(scope);
    }
```

```
    // Операнды для "<", "<=", ">", ">=", "==" и "!=" всегда
    // должны иметь одинаковые типы, если они отличаются, то
    // нужно сделать преобразование
    if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
      // Perform conversion
      RightExpr = new CastExprAST(
        RightExpr->Loc,
        RightExpr,
        LeftExpr->ExprType
      );
      RightExpr = RightExpr->semantic(scope);
    }

    // Результирующий тип выражения — "bool"
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Bool);
    return this;

  case tok::LogOr:
  case tok::LogAnd:
    // Для логических операций оба операнда должны конвертир
    // в "bool"
    if (!LeftExpr->ExprType->implicitConvertTo(
        BuiltinTypeAST::get(TypeAST::TI_Bool)
      ) || !RightExpr->ExprType->implicitConvertTo(
        BuiltinTypeAST::get(TypeAST::TI_Bool)
      )
    ) {
      scope->report(Loc, diag::ERR_SemaCantConvertToBoolean)
      return nullptr;
    }

    // Результирующий тип выражения — "bool"
    ExprType = BuiltinTypeAST::get(TypeAST::TI_Bool);
    return this;

  default:
    // Остальные варианты обрабатываем ниже
```

```
    break;
}

// Если левый операнд "bool", то сначала конвертируем его в
if (LeftExpr->ExprType == BuiltinTypeAST::get(TypeAST::TI_Bo
  LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
    BuiltinTypeAST::get(TypeAST::TI_Int));
  LeftExpr = LeftExpr->semantic(scope);
}

// Результирующий тип выражения будет совпадать с типом лево
// операнда
ExprType = LeftExpr->ExprType;

// Для "=" тоже есть специальная обработка
if (Op == tok::Assign) {
  // Если тип левого операнда указатель, то нужно проверить
  // правый операнд
  if (isa<PointerTypeAST>(ExprType)) {
    // Проверяем на то, что это целочисленная константа и он
    // равна 0
    if (RightExpr->isIntConst()
      && ((IntExprAST*)RightExpr)->Val != 0) {
      scope->report(Loc, diag::ERR_SemaPointerInitializatior
      return nullptr;
    // Проверяем, что операнды совместимы по типам
    } else if (!RightExpr->ExprType->implicitConvertTo(
      ExprType)) {
      scope->report(Loc, diag::ERR_SemaPointerInitializatior
      return nullptr;
    } else {
      // Производим преобразование
      RightExpr = new CastExprAST(
        RightExpr->Loc,
        RightExpr,
        LeftExpr->ExprType
      );
```

```
      RightExpr = RightExpr->semantic(scope);
    }
  } else {
    // Если типы левого и правого операнда отличаются, то ну
    // сделать преобразование
    if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
      RightExpr = new CastExprAST(
        RightExpr->Loc,
        RightExpr,
        LeftExpr->ExprType);
      RightExpr = RightExpr->semantic(scope);
    }
  }

  // Проверяем, что операнд с левой стороны является адресом
  if (!LeftExpr->isLValue()) {
    scope->report(Loc, diag::ERR_SemaMissingLValueInAssignme
    return nullptr;
  }

  // Выражение корректно, завершаем анализ
  return this;
}

// Запрет массивов в других выражениях
if (isa<ArrayTypeAST>(LeftExpr->ExprType)
  || isa<ArrayTypeAST>(RightExpr->ExprType)) {
  scope->report(Loc,
                diag::ERR_SemaCantUseArrayInBinaryExpressior
  return nullptr;
}

// Для указателей и строк мы можем использовать только
// определенный набор операций
if (isa<PointerTypeAST>(LeftExpr->ExprType)
  || LeftExpr->ExprType->isString()
  || isa<PointerTypeAST>(RightExpr->ExprType)
```

```
    || RightExpr->ExprType->isString()) {
    // Если это не "-" или "+", то это ошибка
    if (Op != tok::Plus && Op != tok::Minus) {
      scope->report(Loc,
        diag::ERR_SemaPointerOrStringInBinaryExpression);
      return nullptr;
    }

    // Проверяем, что левый операнд является строкой или
    // указателем
    if (isa<PointerTypeAST>(LeftExpr->ExprType)
      || LeftExpr->ExprType->isString()) {
      // Правый операнд может быть только целочисленным выраже
      if (!RightExpr->ExprType->isInt()) {
        scope->report(Loc,
                      diag::ERR_SemaPointerArithmeticsForNonIn
        return nullptr;
      }

      ExprType = LeftExpr->ExprType;
      return this;
    } else {
      scope->report(Loc,
                    diag::ERR_SemaPointerArithmeticsForNonInt)
      return nullptr;
    }
  }

  // Если левый операнд имеет тип char то преобразуем его в in
  if (LeftExpr->ExprType->isChar()) {
    LeftExpr = new CastExprAST(LeftExpr->Loc, LeftExpr,
      BuiltinTypeAST::get(TypeAST::TI_Int));
    LeftExpr = LeftExpr->semantic(scope);
    ExprType = LeftExpr->ExprType;
  }

  // Если операнды имеют различные типы, то нужно произвести
```

```
// преобразования
if (!LeftExpr->ExprType->equal(RightExpr->ExprType)) {
  // Если операнд с правой стороны имеет тип "float", то
  // результат операции тоже будет "float"
  if (RightExpr->ExprType->isFloat()) {
    ExprType = RightExpr->ExprType;
    LeftExpr = new CastExprAST(
      LeftExpr->Loc,
      LeftExpr,
      RightExpr->ExprType
    );
    LeftExpr = LeftExpr->semantic(scope);
  } else {
    // Преобразуем операнд с правой стороны к типу операнда
    // левой стороны
    RightExpr = new CastExprAST(
      RightExpr->Loc,
      RightExpr,
      LeftExpr->ExprType
    );
    RightExpr = RightExpr->semantic(scope);
  }
}

// "int" и "float" имеют отличный набор допустимых бинарных
// операций
if (ExprType == BuiltinTypeAST::get(TypeAST::TI_Int)) {
  // Проверяем допустимые операции над "int"
  switch (Op) {
    case tok::Plus:
    case tok::Minus:
    case tok::Mul:
    case tok::Div:
    case tok::Mod:
    case tok::BitOr:
    case tok::BitAnd:
    case tok::BitXor:
```

```
      case tok::LShift:
      case tok::RShift:
        return this;

      default:
        // Никогда не должны сюда попасть, если только нет оши
        // в парсере
        assert(0 && "Invalid integral binary operator");
        return nullptr;
    }
  } else {
    // Проверяем допустимые операции над "float"
    switch (Op) {
      case tok::Plus:
      case tok::Minus:
      case tok::Mul:
      case tok::Div:
      case tok::Mod:
      case tok::Less:
        return this;

      default:
        // Сообщаем об ошибке, т. к. данная операция не допуст
        scope->report(Loc,
          diag::ERR_SemaInvalidBinaryExpressionForFloatingPoir
        return nullptr;
    }
  }
}
```

Семантика объявлений:

Hidden text

```
bool SymbolAST::needThis() {
  return false;
}

bool SymbolAST::canBeNull() {
  return true;
}

bool SymbolAST::contain(TypeAST* type) {
  return false;
}

bool VarDeclAST::needThis() {
  return NeedThis;
}

void VarDeclAST::doSemantic(Scope* scope) {
  // Исключаем повторные объявления переменных
  if (needThis()) {
    // Но разрешаем переопределять переменные в родителях
    if (scope->findMember(Id, 1)) {
      scope->report(Loc, diag::ERR_SemaIdentifierRedefinition)
      return;
    }
  } else {
    if (scope->find(Id)) {
      scope->report(Loc, diag::ERR_SemaIdentifierRedefinition)
      return;
    }
  }

  ...
}
```

```
void VarDeclAST::doSemantic3(Scope* scope) {
  // Проверяем наличие инициализатора
  if (Val) {
    // Производим семантический анализ инициализирующего выраж
    Val = Val->semantic(scope);

    // Если это структура или класс, то запрещаем инициализаци
    if (ThisType->isAggregate()) {
      scope->report(Loc, diag::ERR_SemaConstructionCallInStruc
      return;
    }

    // Запрещаем использования выражений с типов "void" в
    // инициализации
    if (!Val->ExprType || Val->ExprType->isVoid()) {
      scope->report(Loc, diag::ERR_SemaVoidInitializer);
      return;
    }

    // Специальная обработка для указателей
    if (isa<PointerTypeAST>(ThisType)) {
      // Выражение для инициализации имеет тип int
      if (Val->ExprType->isInt()) {
        // Разрешаем только присвоение 0
        if (Val->isIntConst() && ((IntExprAST*)Val)->Val != 0)
          scope->report(Loc, diag::ERR_SemaPointerInitializati
        }

        return;
      // Проверяем, что типы совместимы
      } else if (!Val->ExprType->implicitConvertTo(ThisType))
        scope->report(Loc, diag::ERR_SemaPointerInitializatior
        return;
      }
    }
```

```
    // Если типы не совпадают, то добавляем преобразование
    if (!Val->ExprType->equal(ThisType)) {
      Val = new CastExprAST(Loc, Val, ThisType);
      Val = Val->semantic(scope);
    }
  }
}

bool VarDeclAST::contain(TypeAST* type) {
  // Проверяем, что типы совпадают
  if (ThisType->equal(type)) {
    return true;
  }

  return false;
}

bool VarDeclAST::canBeNull() {
  // Если это не указатель, то возвращаем false
  if (!isa<PointerTypeAST>(ThisType)) {
    return false;
  }
  // Возвращаем true, т. к. указатель может содержать значение
  return true;
}

TypeAST* StructDeclAST::getType() {
  return ThisType;
}

void StructDeclAST::doSemantic(Scope* scope) {
  // Запрещаем переопределение
  if (scope->find(Id)) {
    scope->report(Loc, diag::ERR_SemaIdentifierRedefinition);
    return;
  }
```

```
  // Добавляем новую область видимости
  Scope* s = scope->push(this);
  // Производим семантику объявления типа
  ThisType = ThisType->semantic(s);

  // Добавление объявления в родительскую область видимости и
  // сохраняем объявление родителя
  ((ScopeSymbol*)scope->CurScope)->Decls[Id] = this;
  Parent = scope->CurScope;

  // Удаляем область видимости для данного объявления
  s->pop();
}

void StructDeclAST::doSemantic3(Scope* scope) {
  // Создаем новую область видимости
  Scope* s = scope->push(this);
  int curOffset = 0;

  // Проверка всех дочерних объявлений
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    VarDeclAST* var = (VarDeclAST*)(*it);

    // Производим семантическую проверку для члена структуры
    // и получаем индекс для данного члена
    var->semantic(s);
    var->OffsetOf = curOffset++;
  }

  // Проверка на наличие циклических зависимостей
  if (contain(ThisType)) {
    scope->report(Loc, diag::ERR_SemaCricularReference, Id->Ic
    return;
  }

  // Удаляем область видимости для данного объявления
```

```
  s->pop();
}

bool StructDeclAST::contain(TypeAST* type) {
  // Проверяем все переменные члены структуры на циклы
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
   it != end; ++it) {
    if ((*it)->contain(type)) {
      return true;
    }
  }

  return false;
}

bool ParameterSymbolAST::canBeNull() {
  // Если параметр не является указателем, то возвращаем false
  if (!isa<PointerTypeAST>(Param->Param)) {
    return false;
  }
  // Возвращаем true, т. к. указатель может содержать значение
  return true;
}

// Функции, которые будут доступны из simple
extern "C" void lle_X_printInt(int val) {
  outs() << val;
}

extern "C" void lle_X_printDouble(double val) {
  outs() << val;
}

extern "C" void lle_X_printChar(char val) {
  outs() << val;
}
```

```
extern "C" void lle_X_printString(char* str) {
  outs() << str;
}

extern "C" void lle_X_printLine(char* str) {
  outs() << str << "\n";
}

extern "C" void* lle_X_new(int size) {
  return malloc(size);
}

extern "C" void lle_X_delete(void* block) {
  free(block);
}

void initRuntimeFuncs(ModuleDeclAST* modDecl) {
  addDynamicFunc(
    "fn printChar(_: char)",
    "lle_X_printChar",
    modDecl,
    (void*)lle_X_printChar
  );
  addDynamicFunc(
    "fn printInt(_: int)",
    "lle_X_printInt",
    modDecl,
    (void*)lle_X_printInt
  );
  addDynamicFunc(
    "fn printFloat(_: float)",
    "lle_X_printDouble",
    modDecl,
    (void*)lle_X_printDouble
  );
  addDynamicFunc(
    "fn printString(_: string)",
```

```
    "lle_X_printString",
    modDecl,
    (void*)lle_X_printString
  );
  addDynamicFunc(
    "fn printLn(_: string)",
    "lle_X_printLine",
    modDecl,
    (void*)lle_X_printLine
  );
  addDynamicFunc(
    "fn new(_: int) : void*",
    "lle_X_new",
    modDecl,
    (void*)lle_X_new
  );
  addDynamicFunc(
    "fn delete(_: void*)",
    "lle_X_delete",
    modDecl,
    (void*)lle_X_delete
  );
}

void ModuleDeclAST::semantic() {
  Scope s(this);

  // Инициализация runtime функций
  initRuntimeFuncs(this);

  // Производим семантический анализ всех базовых типов
  for (int i = TypeAST::TI_Void; i <= TypeAST::TI_String; ++i)
    BuiltinTypeAST::get(i)->semantic(&s);
  }
```

```
    ...
}
```

GetElementPtr

Для того, что бы мы могли получать доступ к элементу массива или полю в структуре, в LLVM существует инструкция GetElementPtr. Она используется для получения адреса дочернего элемента составного типа. Эта инструкция производит только расчет адреса и не обращается к самой памяти.

Синтаксис этой инструкции имеет вид (ниже будет приведен только один из способов ее записи, более подробнее можно прочитать в [1]:

```
<result> = getelementptr <ty>, ptr <ptrval>{, [inrange] <ty> <idx>}*
```

Первый аргумент всегда является тип, который будет использоваться за основу при расчете. Второй всегда будет указателем или вектором указателей, который содержит базовый адрес с которого нужно произвести расчет. После чего идет список индексов, которые зависят от индексируемого типа. Первый индекс всегда индексирует значение указателя из второго аргумента, следующий индекс индексирует значение типа, на который указывают (не обязательно значение, на которое прямо указывает, т. к. первый индекс может быть отличным от 0) и т. д. Первый индексируемый тип должен быть значением указателя, последующие могут быть массивом, структурой или вектором. Но последующие индексируемые типы не могут быть указателями, т. к. для начала нужно загрузить его значение, для продолжения вычисления.

Например, для получения адреса 3 элемента в массиве %arr из 10 элементов типа i32, инструкцию GetElementPtr можно записать следующим образом:

```
%1 = getelementptr inbounds [10 x i32], ptr %arr, i64 0, i64 3
```

Если %arr является двумерным массивом 10x10, то для получения адреса 2 элемента внутри 1 элемента массива можно записать:

```
%1 = getelementptr inbounds [10 x [10 x i32]], ptr %arr, i64 0, i64 1
```

Что также может быть записано в виде:

```
%1 = getelementptr inbounds [10 x [10 x i32]], ptr %arr, i64 0, i64 1
%2 = getelementptr inbounds [10 x i32], ptr %1, i64 0, i64 2
```

Замечание: В нашей реализации мы будем использовать более длинный вариант состоящий из 2-х инструкций.

Аналогично и со структурами. Для структуры вида %struct.C = type { i32, i32 }, получение второго поля, можно записать:

```
%1 = getelementptr inbounds %struct.C, ptr %c, i32 0, i32 1
```

Более подробно про особенности использования инструкцию можно почитать в [2], а примеры использования можно почитать в [3].

Изменения в генерации кода

Генерация кода для типов будет иметь следующий вид:

▾ Hidden text

```
Type* BuiltinTypeAST::getType() {
  static Type* builtinTypes[] = {
    Type::getVoidTy(getGlobalContext()),
    Type::getInt1Ty(getGlobalContext()),
    Type::getInt32Ty(getGlobalContext()),
    Type::getDoubleTy(getGlobalContext()),
    Type::getInt8Ty(getGlobalContext()),
    PointerType::get(
      getGlobalContext(),
      getSLContext().TheTarget->getProgramAddressSpace()
    )
  };

  if (ThisType) {
    return ThisType;
  }

  return ThisType = builtinTypes[TypeKind];
}

Type* ArrayTypeAST::getType() {
  // Генерируем тип только один раз
  if (ThisType) {
    return ThisType;
  }

  // Генерируем тип для массива
  return ThisType = ArrayType::get(Next->getType(), Dim);
}

Type* PointerTypeAST::getType() {
```

```
  // Генерируем тип только один раз
  if (ThisType) {
    return ThisType;
  }

  // Генерируем тип для указателя
  return ThisType = PointerType::get(
    getGlobalContext(),
    getSLContext().TheTarget->getProgramAddressSpace()
  );
}

/// Сгенерировать строку с именем агрегата
/// \param[in] output — результат
/// \param[in] thisSym — агрегат имя которого нужно сгенериров
/// \note Генерирует строку вида A.B.C для вложенных агрегатов
void calcAggregateName(llvm::raw_ostream& output,
                       SymbolAST* thisSym) {
  assert(thisSym && thisSym->isAggregate());
  SymbolAST* sym = thisSym;

  // Для генерируем результирующую строку для всех вложенных
  // агрегатов
  for ( ; ; ) {
    output << StringRef(sym->Id->Id, sym->Id->Length);
    sym = sym->Parent;

    if (!sym || !sym->isAggregate()) {
      return;
    }

    output << ".";
  }
}

Type* StructTypeAST::getType() {
  // Генерируем тип только один раз
```

```
  if (ThisType) {
    return ThisType;
  }

  llvm::SmallString< 128 > s;
  llvm::raw_svector_ostream output(s);

  // Для структур имя типа всегда начинается с "struct."
  output << "struct.";
  calcAggregateName(output, ThisDecl);

  // Замечание: нужно присвоить тип агрегата сейчас, т. к.
  // дочерние объявления могут ссылаться на саму структуру
  ThisType = StructType::create(getGlobalContext(), output.str

  std::vector< Type* > vars;
  StructDeclAST* structDecl = (StructDeclAST*)ThisDecl;

  // Создаем список всех вложенных объявлений
  for (SymbolList::iterator it = structDecl->Vars.begin(),
    end = structDecl->Vars.end(); it != end; ++it) {
    vars.push_back(((VarDeclAST*)(*it))->ThisType->getType());
  }

  // Если в структуре нет ни одного члена, то добавляем поле
  // размером в один байт, что бы размер типа отличался от 0
  if (vars.empty()) {
    vars.push_back(Type::getInt8Ty(getGlobalContext()));
  }

  // Установить список дочерних объявлений для структуры и вер
  // полученный тип
  ((StructType*)ThisType)->setBody(vars);
  return ThisType;
}

Type* QualifiedTypeAST::getType() {
```

```
  assert(0 && "QualifiedTypeAST::getType should never be reach
  return nullptr;
}
```

Генерация кода для выражений будет иметь следующий вид:

▾ Hidden text

```
/// Конвертировать тип выражения в "bool"
/// \param[in] val — значение для конвертации
/// \param[in] type — тип изначального выражения
/// \param[in] builder — конструктор LLVM IR
Value* promoteToBool(Value* val, TypeAST* type,
                     IRBuilder< >& builder) {
  if (type == BuiltinTypeAST::get(TypeAST::TI_Bool)) {
    // Это уже "bool"
    return val;
  }

  if (type->isInt() || type->isChar()) {
    // Для целочисленных типов генерируем сравнение с 0
    return builder.CreateICmpNE(
      val,
      ConstantInt::get(type->getType(), 0)
    );
  } else if (isa<PointerTypeAST>(type) || type->isString()) {
    // Для указателей или строк производим сравнение с null
    return builder.CreateICmpNE(val,
      ConstantPointerNull::get((PointerType*)type->getType()))
  } else {
    assert(type->isFloat());
```

```
    // Для числа с плавающей точкой генерируем сравнение с 0.0
    return builder.CreateFCmpUNE(val, ConstantFP::get(
      Type::getDoubleTy(getGlobalContext()), 0.0));
  }
}

Value* StringExprAST::getRValue(SLContext& Context) {
  // Создаем глобальную строку
  GlobalValue* val = Context.TheBuilder->CreateGlobalString(Va
  std::vector< Value* > idx;

  idx.push_back(getConstInt(0));
  idx.push_back(getConstInt(0));

  // Создаем инструкцию GetElementPtr для получения адреса ран
  // созданной константы
  return Context.TheBuilder->CreateInBoundsGEP(
    val->getValueType(),
    val,
    idx
  );
}

Value* ProxyExprAST::getRValue(SLContext& Context) {
  return OriginalExpr->getRValue(Context);
}

Value* NewExprAST::getRValue(SLContext& Context) {
  // Нам необходимо определить размер типа и заменить 0, котор
  // указали во время семантического анализа, реальным значени
  uint64_t allocSize = Context.TheTarget->getTypeAllocSize(
    NewType->getType());
  SizeExpr->Val = (int)allocSize;
  // Генерируем код для выражения вызова функции выделения бло
  // памяти
  Value* val = CallExpr->getRValue(Context);
```

```
  if (NeedCast) {
    // Необходимо произвести преобразование типов
    return Context.TheBuilder->CreateBitCast(
      val,
      ExprType->getType()
    );
  }

  return val;
}

Value* DeleteExprAST::getRValue(SLContext& Context) {
  Value* val = DeleteCall->getRValue(Context);
  return val;
}

Value* IndexExprAST::getLValue(SLContext& Context) {
  // У нас есть 2 отличающихся друг от друга варианта: для мас
  // и для указателей
  if (isa<ArrayTypeAST>(Left->ExprType)) {
    // Получаем массив как lvalue и значение самого индекса
    Value* val = Left->getLValue(Context);
    Value* index = Right->getRValue(Context);

    std::vector< Value* > idx;

    idx.push_back(getConstInt(0));
    idx.push_back(index);
    // Создаем инструкцию GetElementPtr
    return Context.TheBuilder->CreateInBoundsGEP(
      Left->ExprType->getType(),
      val,
      idx
    );
  } else {
    // Для указателя мы должны получить его rvalue и значение
    // самого индекса
```

```
    Value* val = Left->getRValue(Context);
    Value* index = Right->getRValue(Context);

    std::vector< Value* > idx;

    idx.push_back(index);
    // Создаем инструкцию GetElementPtr
    return Context.TheBuilder->CreateInBoundsGEP(
      ExprType->getType(),
      val,
      idx
    );
  }
}

Value* IndexExprAST::getRValue(SLContext& Context) {
  Value* val = getLValue(Context);
  return Context.TheBuilder->CreateLoad(ExprType->getType(), v
}

Value* MemberAccessExprAST::getLValue(SLContext& Context) {
  // Для функций просто возвращаем ее тип
  if (isa<FuncTypeAST>(ExprType)) {
    return ThisSym->getValue(Context);
  }

  // Получаем lvalue для левой части (this)
  Value* val = Val->getLValue(Context);

  // Генерирум индекс на основе индекса члена класса/структурь
  Value* index = getConstInt(((VarDeclAST*)ThisSym)->OffsetOf)

  std::vector< Value* > idx;

  idx.push_back(getConstInt(0));
  idx.push_back(index);
```

```
  // Создаем инструкцию GetElementPtr
  return Context.TheBuilder->CreateGEP(
    ThisSym->Parent->getType()->getType(),
    val,
    idx
  );
}

Value* MemberAccessExprAST::getRValue(SLContext& Context) {
  // Получаем lvalue для левой части (this)
  Value* val = getLValue(Context);

  // Для функций возвращаем полученное значение
  if (isa<FuncTypeAST>(ExprType)) {
    return val;
  }

  // Для rvalue необходимо создать инструкцию "load"
  return Context.TheBuilder->CreateLoad(ExprType->getType(), v
}

Value* PointerAccessExprAST::getLValue(SLContext& Context) {
  assert(0 && "PointerAccessExprAST::getLValue should never be
         "reached");
  return nullptr;
}

Value* PointerAccessExprAST::getRValue(SLContext& Context) {
  assert(0 && "PointerAccessExprAST::getRValue should never be
         "reached");
  return nullptr;
}

Value* DerefExprAST::getLValue(SLContext& Context) {
  // Проверяем, что это lvalue
  if (Val->isLValue()) {
    // Получаем lvalue выражения и создаем инструкцию "load"
```

```
    Value* val = Val->getLValue(Context);
    return Context.TheBuilder->CreateLoad(
      PointerType::get(
        getGlobalContext(),
        Context.TheTarget->getProgramAddressSpace()
      ),
      val
    );
  } else {
    // Для функций и бинарных выражений генерируем rvalue
    return Val->getRValue(Context);
  }
}

Value* DerefExprAST::getRValue(SLContext& Context) {
  // Получаем lvalue
  Value* val = getLValue(Context);

  // Для функций возвращаем полученное значение
  if (isa<FuncTypeAST>(ExprType)) {
    return val;
  }

  // Для rvalue необходимо создать инструкцию "load"
  return Context.TheBuilder->CreateLoad(ExprType->getType(), v
}

Value* AddressOfExprAST::getRValue(SLContext& Context) {
  // Для rvalue выражения получения адреса, мы должны вернуть
  // lvalue операнда
  // Замечание: Данная операция не имеет lvalue
  return Val->getLValue(Context);
}

Value* CastExprAST::getRValue(SLContext& Context) {
  // Сначала проверяем, что это преобразование в целочисленный
  if (ExprType->isInt()) {
```

```
    if (Val->ExprType->isBool() || Val->ExprType->isChar()) {
      // Если исходный тип "bool" или "char", то дополняем 0
      return Context.TheBuilder->CreateZExt(
        Val->getRValue(Context),
        ExprType->getType()
      );
    }

    assert(Val->ExprType->isFloat());
    // Генерируем преобразование "float" в "int"
    return Context.TheBuilder->CreateFPToSI(
      Val->getRValue(Context),
      ExprType->getType()
    );
  }

  if (ExprType->isBool()) {
    // Для преобразования в "bool"
    return promoteToBool(Val->getRValue(Context), Val->ExprTyp
      *Context.TheBuilder);
    // Для преобразования в "char"
  } else if (ExprType->isChar()) {
    // Если это "bool", то дополняем 0
    if (Val->ExprType->isBool()) {
      return Context.TheBuilder->CreateZExt(
        Val->getRValue(Context),
        ExprType->getType()
      );
    }

    assert(Val->ExprType->isInt());
    // Обрезаем целочисленное значение
    return Context.TheBuilder->CreateTrunc(
      Val->getRValue(Context),
      ExprType->getType()
    );
  // Проверяем преобразование в строку
```

```
} else if (ExprType->isString()) {
  // Проверяем, что это массив
  if (isa<ArrayTypeAST>(Val->ExprType)) {
    // Получаем lvalue исходного значения
    Value* val = Val->getLValue(Context);
    Value* tmp = getConstInt(0);

    std::vector< Value* > idx;

    idx.push_back(tmp);
    idx.push_back(tmp);

    // Проверяем, что значение является объявление переменнс
    if (isa<AllocaInst>(val)) {
      AllocaInst *alloca = (AllocaInst*)val;
      // Создаем инструкцию GetElementPtr
      return Context.TheBuilder->CreateGEP(
        alloca->getAllocatedType(),
        val,
        idx
      );
    } else {
      assert(0);
    }
  } else {
    // Получаем rvalue значения
    Value* val = Val->getRValue(Context);
    // Создаем преобразование в новый тип
    return Context.TheBuilder->CreateBitCast(
      val,
      ExprType->getType()
    );
  }
// Проверка на преобразования к указателю
} else if (isa<PointerTypeAST>(ExprType)) {
  // Проверяем на приведение указателя
  if (isa<PointerTypeAST>(Val->ExprType)) {
```

```
    // Генерируем rvalue
    return Val->getRValue(Context);
  // Проверяем на приведение массива
  } else if (isa<ArrayTypeAST>(Val->ExprType)) {
    // Получаем lvalue исходного значения
    Value* val = Val->getLValue(Context);
    Value* tmp = getConstInt(0);

    std::vector< Value* > idx;

    idx.push_back(tmp);
    idx.push_back(tmp);

    // Проверяем, что значение является объявление переменнс
    if (isa<AllocaInst>(val)) {
      AllocaInst *alloca = (AllocaInst*)val;
      // Создаем инструкцию GetElementPtr
      tmp = Context.TheBuilder->CreateGEP(
        alloca->getAllocatedType(),
        val,
        idx
      );
    } else {
      assert(0);
    }

    PointerTypeAST* ptrType = (PointerTypeAST*)ExprType;

    if (ptrType->Next->isVoid()) {
      // Создаем преобразование
      tmp = Context.TheBuilder->CreateBitCast(
        tmp,
        ptrType->getType()
      );
    }

    return tmp;
```

```
    } else {
      // Генерируем rvalue
      Value* val = Val->getRValue(Context);
      // Создаем преобразование
      return Context.TheBuilder->CreateBitCast(
        val,
        ExprType->getType()
      );
    }
  } else if (Val->ExprType->isInt() || Val->ExprType->isChar()
    // Преобразование "int" или "char" во "float"
    return Context.TheBuilder->CreateSIToFP(
      Val->getRValue(Context),
      ExprType->getType()
    );
  } else if (Val->ExprType->isBool()) {
    // Преобразование "bool" в "float"
    return Context.TheBuilder->CreateUIToFP(
      Val->getRValue(Context),
      ExprType->getType()
    );
  }

  assert(0 && "should never be reached");
  return nullptr;
}

Value* UnaryExprAST::getRValue(SLContext& Context) {
  assert(Op == tok::PlusPlus || Op == tok::MinusMinus);
  assert(isa<IndexExprAST>(Val) || isa<MemberAccessExprAST>(Va
    || ExprType->isString());
  // Получаем lvalue
  Value* val = Val->getLValue(Context);
  // Создаем инструкцию "load" и константу 1 или -1
  Value* res = Context.TheBuilder->CreateLoad(
    ExprType->getType(),
    val
```

```
  );
  Value* tmp = getConstInt((Op == tok::PlusPlus) ? 1ULL : ~0UL

  if (isa<PointerTypeAST>(ExprType) || ExprType->isString()) {
    Type *resType;

    // Проверяем, что тип выражения указатель
    if (isa<PointerTypeAST>(ExprType)) {
      PointerTypeAST *ptrType = (PointerTypeAST*)ExprType;

      // Определяем тип для результата
      if (ptrType->Next->isVoid()) {
        resType = Type::getInt8Ty(getGlobalContext());
      } else {
        resType = ptrType->Next->getType();
      }
    } else {
      // Это строка, результат будет символом
      resType = Type::getInt8Ty(getGlobalContext());
    }

    // Для указателей создаем инструкцию GetElementPtr
    res = Context.TheBuilder->CreateGEP(resType, res, tmp);
  } else {
    // Создаем инструкцию "add"
    res = Context.TheBuilder->CreateAdd(res, tmp);
  }

  // Создаем инструкцию "store" для сохранения результата
  Context.TheBuilder->CreateStore(res, val);
  return res;
}

Value* BinaryExprAST::getRValue(SLContext& Context) {
  // Сначала необходимо проверить все специальные случаи

  // =
```

```
if (Op == tok::Assign) {
  // Генерируем код для правой части выражения
  Value* right;

  // Специальная обработка для указателей
  if (isa<PointerTypeAST>(ExprType)) {
    // Если это константа 0, то конвертируем ее в null
    if (RightExpr->ExprType->isInt()) {
      right = ConstantPointerNull::get(
        (PointerType*)ExprType->getType()
      );
    } else {
      // Создаем rvalue
      right = RightExpr->getRValue(Context);
    }
  } else {
    // Создаем rvalue
    right = RightExpr->getRValue(Context);
  }

  // Получаем адрес по которому нужно сохранить значение
  Value* res = LeftExpr->getLValue(Context);

  // Генерируем инструкцию "store"
  Context.TheBuilder->CreateStore(right, res);
  return right;
}

// ,
if (Op == tok::Comma) {
  // Генерируем код для левого и правого операнда
  LeftExpr->getRValue(Context);
  Value* rhs = RightExpr->getRValue(Context);
  // Возвращаем правый операнд
  return rhs;
}
```

```
// Постфиксная версия операторов ++ и --
if (Op == tok::PlusPlus || Op == tok::MinusMinus) {
  // Получаем адрес переменной, а так же ее значение
  Value* var = LeftExpr->getLValue(Context);
  Value* val = nullptr;

  // Специальная обработка для IndexExprAST и MemberAccessEx
  if (isa<IndexExprAST>(LeftExpr)
      || isa<MemberAccessExprAST>(LeftExpr)) {
    // Создаем инструкцию "load"
    val = Context.TheBuilder->CreateLoad(
      ExprType->getType(),
      var
    );
  } else {
    // Генерируем rvalue
    val = LeftExpr->getRValue(Context);
  }

  if (!LeftExpr->ExprType->isInt()) {
    // Специальная обработка для указателей, т.к мы должны
    // создать инструкцию GetElementPtr
    Value* tmp = getConstInt(
      (Op == tok::PlusPlus) ? 1ULL : ~0ULL
    );
    Type *resType;

    // Проверяем, что тип выражения указатель
    if (isa<PointerTypeAST>(ExprType)) {
      PointerTypeAST *ptrType = (PointerTypeAST*)ExprType;

      // Определяем тип для результата
      if (ptrType->Next->isVoid()) {
        resType = Type::getInt8Ty(getGlobalContext());
      } else {
        resType = ptrType->Next->getType();
      }
```

```
    } else {
      // Это строка, результат будет символом
      resType = Type::getInt8Ty(getGlobalContext());
    }

    // Создаем инструкцию GetElementPtr
    tmp = Context.TheBuilder->CreateGEP(resType, val, tmp);
    // Создаем инструкцию "store" и возвращаем старое значен
    Context.TheBuilder->CreateStore(tmp, var);
    return val;
  } else {
    if (Op == tok::PlusPlus) {
      // Создать целочисленную константу 1 и прибавить ее к
      // загруженному ранее значению
      Value* tmp = getConstInt(1);
      tmp = Context.TheBuilder->CreateAdd(val, tmp, "inctmp"
      // Сохранить новое значение и вернуть старое значение
      Context.TheBuilder->CreateStore(tmp, var);
      return val;
    } else {
      // Создать целочисленную константу -1 и прибавить ее к
      // загруженному ранее значению
      Value* tmp = getConstInt(~0ULL);
      tmp = Context.TheBuilder->CreateAdd(val, tmp, "dectmp"
      // Сохранить новое значение и вернуть старое значение
      Context.TheBuilder->CreateStore(tmp, var);
      return val;
    }
  }
}

...

// Специальная обработка для +/- над указателями
if (isa<PointerTypeAST>(ExprType)) {
  // Получить значение для левой и правой части
  Value* ptr = LeftExpr->getRValue(Context);
```

```
    Value* index = RightExpr->getRValue(Context);

    // Для минуса нужно создать инструкцию "sub"
    if (Op == tok::Minus) {
      index = Context.TheBuilder->CreateSub(getConstInt(0), ir
    }

    // Определяем тип значения для загрузки
    PointerTypeAST *ptrType = (PointerTypeAST*)ExprType;
    Type *resType = ptrType->Next->isVoid()
      ? Type::getInt8Ty(getGlobalContext())
      : ptrType->Next->getType();

    // Создаем инструкцию GetElementPtr
    return Context.TheBuilder->CreateGEP(resType, ptr, index);
  }

  // Это обычное выражение с двумя операндами

  // Генерируем код для обоих операндов
  Value* lhs = LeftExpr->getRValue(Context);
  Value* rhs = RightExpr->getRValue(Context);
  ...
}
```

Правки для объявлений:

▾ Hidden text

```
Value* VarDeclAST::generateCode(SLContext& Context) {
  assert(SemaState >= 5);
```

```
// Получаем адрес переменной (т. к. память под саму переменн
// должна была выделена ранее во время генерации кода для фу
Value* val = getValue(Context);

// Если у нас есть выражение инициализации, то нужно ее прои
if (Val) {
  Value* init;

  // Специальная обработка для указателей
  if (isa<PointerTypeAST>(ThisType)) {
    // Для целочисленной константы 0, создаем null
    if (Val->isIntConst()) {
      init = ConstantPointerNull::get(
        (PointerType*)ThisType->getType()
      );
    } else {
      // Специальная обработка для массивов
      if (isa<ArrayTypeAST>(Val->ExprType)) {
        // Получить lvalue выражения для инициализации
        init = Val->getLValue(Context);
        Value* tmp = getConstInt(0);

        std::vector< Value* > idx;

        idx.push_back(tmp);
        idx.push_back(tmp);

        // Проверяем, что значение является объявление перем
        if (isa<AllocaInst>(init)) {
          AllocaInst *alloca = (AllocaInst*)init;
          // Создаем инструкцию GetElementPtr
          init = Context.TheBuilder->CreateGEP(
            alloca->getAllocatedType(),
            init,
            idx
          );
        } else {
```

```
            assert(0);
          }
        } else {
          // Генерация кода для инициализирующего выражения
          init = Val->getRValue(Context);
        }
      }
    } else {
      // Генерация кода для инициализирующего выражения
      init = Val->getRValue(Context);
    }

    // Создание инструкции "store"
    return Context.TheBuilder->CreateStore(init, val);
  }

  return val;
}

Value* StructDeclAST::getValue(SLContext& Context) {
  // Генерируем тип для исходного типа
  ThisType->getType();
  return nullptr;
}

Value* StructDeclAST::generateCode(SLContext& Context) {
  assert(SemaState >= 5);
  // Генерируем тип для исходного типа
  ThisType->getType();
  return nullptr;
}
```

Заключение

В данной части мы добавили в наш учебный язык поддержку новых типов, таких как:

1. Строки;
2. Указатели;
3. Массивы;
4. Структуры.

Так же мы добавили новые операции в язык, такие как:

1. Операция разыменования указателей ("*" для одного операнда);
2. Операция взятия адреса ("&" для одного операнда);
3. Операция индексации (обращения к конкретному элементу массива);
4. Операция обращения к полю структуры;
5. Операторы "new" и "del" для выделения и очистки блока памяти.

Полный исходный код доступен на github (в папке tests есть несколько примеров, которые тестируют различные части языка). В дальнейшем мы рассмотрим, как добавить в наш язык поддержку классов с наследованием и динамической диспетчеризации методов, а так же перегрузку функций.

Полезные ссылки

1. https://llvm.org/docs/LangRef.html#getelementptr-instruction
2. https://llvm.org/docs/GetElementPtr.html
3. https://blog.yossarian.net/2020/09/19/LLVMs-getelementptr-by-example

Теги: llvm, компиляторы, c++

Хабы: Open source, Программирование, Компиляторы

Создаем свой собственный язык программирования с использованием LLVM. Часть 5: Поддержка классов и перегрузки функций

4.4K

Open source*, Программирование*, Компиляторы*

В предыдущей статье мы закончили на том, что добавили в наш учебный язык поддержку строк, указателей, массивов, структур, а так же операции для работы с ними. В этой части мы продолжим расширять дынный язык и добавим в него: классы с динамической диспетчеризации методов, одиночным наследованием и поддержку перегрузки функций на основе их параметров.

Оглавление серии

Введение

Для начала рассмотри кратко, что именно будет реализовано в результате:

1. Добавим новый тип для классов, который в отличие от структур имеет дополнительные свойства:

- Может содержать функции члены (методы), которые могут работать с переменным членами класса или производить различные другие действия;
- Могут иметь родительский класс или структуру, от которых класс может наследовать поведение;
- Методы класса могут быть виртуальными. И тогда при вызове этих методов через переменные типа указателя на экземпляр этого класса, реализация метода будет выбираться во время выполнения программы, что позволяет вызывать корректные методы даже при вызове этих методов через указатель на объект родительского класса;
- Могут иметь специальные методы класса — конструкторы (инициализируют начальное состояние объекта) и деструкторы (производит очистку состояния объекта), причем деструкторы могут быть виртуальными.

2. Добавим поддержку автоматического вызова конструкторов и деструкторов класса для их объектов при выходе объектов класса из области их видимости;
3. Поддержку переиспользования имен функций путем создания набора перегруженных функций, которые отличаются типами и/или количеством параметров.

Далее будем последовательно рассматривать все правки, которые нужно внести в каждую из частей нашего интерпретатора.

Лексический и синтаксический анализ

Единственные изменения, которые нужно применить в лексическом анализаторе, это:

Добавить поддержку новых ключевых слов в include/Basic/Name.h

```
KEYWORD(Class, "class")
KEYWORD(Extends, "extends")
KEYWORD(This, "this")
KEYWORD(Super, "super")
KEYWORD(Virtual, "virt")
KEYWORD(Override, "impl")
```

Добавить новые переменные для работы с некоторыми из новых ключевых слов (для использования их в семантическом анализе для служебных методов и специальных конструкций, например для вызова методов базового класса):

▾ Hidden text

```
struct Name {
  ...
  static Name *Super;  ///< Имя для ключевого слова "super"
  static Name *This;   ///< Имя для ключевого слова "this"
  static Name *Ctor;   ///< Имя для конструктора
  static Name *Dtor;   ///< Имя для деструктора
};
// lib/Basic/Name.cpp"
Name *Name::Super = nullptr;
Name *Name::This = nullptr;
Name *Name::Ctor = nullptr;
Name *Name::Dtor = nullptr;

// lib/Lexer/Lexer.cpp
void NamesMap::addKeywords() {
  if (IsInit) {
    return;
  }
```

```
#define KEYWORD(NAME, TEXT)                                    \
  addName(StringRef(TEXT), tok::NAME);
#include "simple/Basic/TokenKinds.def"

  Name::This = getName("this");
  Name::Super = getName("super");
  Name::New = getName("new");
  Name::Delete = getName("delete");
  Name::Ctor = addName("__ctor", tok::Identifier);
  Name::Dtor = addName("__dtor", tok::Identifier);

  IsInit = true;
}
```

Правок в синтаксическом анализаторе будет побольше.

Необходимо добавить поддержку новых выражений для указания экземпляра текущего класса или класса родителя:

▾ Hidden text

```
/// primary-expr
///   ::= floating-point-literal
///   ::= integral-literal
///   ::= '.'? identifier
///   ::= char-literal
///   ::= string-literal
///   ::= '(' expr ')'
///   ::= 'this'
///   ::= 'super'
ExprAST *Parser::parsePrimaryExpr() {
```

```
  ExprAST *result = nullptr;

  switch (CurPos->getKind()) {
    ...
    case tok::Super:
      result = new IdExprAST(CurPos.getLocation(), Name::Super
      ++CurPos;
      return result;

    case tok::This:
      result = new IdExprAST(CurPos.getLocation(), Name::This)
      ++CurPos;
      return result;
     ...
  }
}
```

Поддержка создания экземпляра класса через оператор "new":

▾ Hidden text

```
/// unary-expr
///   ::= postfix-expr
///   ::= '+' unary-expr
///   ::= '-' unary-expr
///   ::= '++' unary-expr
///   ::= '--' unary-expr
///   ::= '~' unary-expr
///   ::= '!' unary-expr
///   ::= '*' unary-expr
///   ::= '&' unary-expr
```

```
///   ::= 'del' unary-expr
///   ::= new-expr
/// new-expr
///   ::= 'new' type [' assign-expr ']'
///   ::= 'new' type ('(' call-arguments? ')') ?
ExprAST *Parser::parseUnaryExpr() {
  ExprAST *result = nullptr;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (int op = CurPos->getKind()) {
    ...
    case tok::New: {
      // Это операция выделения памяти. Пропускаем "new"
      ++CurPos;
      // Считываем тип
      TypeAST *type = parseType();
      ExprList args;
      ExprAST *dynamicSize = nullptr;

      // Проверяем "[" expression "]" , т. к. "[" integral-lit
      // "]" может быть обработано ранее в parseType
      if (CurPos == tok::OpenBrace) {
        // Проверяем, что слева от "[" у нас нету структуры ил
        // класса, т. к. мы их не поддерживаем
        if (isa<QualifiedTypeAST>(type)) {
          getDiagnostics().report(CurPos.getLocation(),
                                  diag::ERR_DynArrayAggregate)
        }
       // Пропускаем "[" и считываем выражение между "[" и "]"
        ++CurPos;
        dynamicSize = parseAssignExpr();
        // Проверяем наличие "]"
        check(tok::CloseBrace);
      }
      // Если тип является полным (может быть именем класса ил
      // структуры), а так же текущий токен является "(", то н
      // считать аргументы для конструктора
```

```
      if (isa<QualifiedTypeAST>(type) && CurPos == tok::OpenPa
        // Пропустить "("
        ++CurPos;

        // Проверяем наличие аргументов (может быть вызов
        // конструктора без аргументов)
        if (CurPos != tok::CloseParen) {
          for (;;) {
            // Считываем выражение аргумента
            ExprAST *arg = parseAssignExpr();
            args.push_back(arg);
            // Проверяем на наличие ","
            if (CurPos != tok::Comma) {
              // Нет, завершаем разбор аргументов
              break;
            }
            // Считываем "," и переходим к следующему аргумент
            ++CurPos;
          }
        }
        // Проверяем на наличие ")"
        check(tok::CloseParen);
      }
      // Создаем ветку дерева
      return new NewExprAST(loc, type, dynamicSize, args);
    }

    default:
      return parsePostfixExpr();
  }
}
```

Небольшая правка в разборе прототипа функции, для поддержки нового конструктора:

Hidden text

```
SymbolAST *Parser::parseFuncProto() {
  ...
  return new FuncDeclAST(loc, returnType, name, nullptr, false
}
```

Добавляем поддержку методов класса:

Hidden text

```
/// func-decl ::= func-decl-kwd func-proto block-stmt
/// func-decl-kwd
///   ::= 'fn'
///   ::= 'virt'
///   ::= 'impl'
SymbolList Parser::parseFuncDecl(bool isClassMember)) {
  // Чтение прототипа функции
  SymbolList result;
  FuncDeclAST *decl = (FuncDeclAST *)parseFuncProto();

  if (CurPos != tok::BlockStart) {
    // Отсутствие "{" является ошибкой
    getDiagnostics().report(CurPos.getLocation(),
                            diag::ERR_ExpectedFuncBody);
  } else {
```

```
    // Считываем тело функции
    StmtAST *body = parseStmt();
    // Добавляем тело к прототипу функции
    decl->Body = body;
    // Сохраняем флаг того, что это метод класса или нет
    decl->NeedThis = isClassMember;
    // Добавляем созданное объявление функции к результирующим
    // объявлениям
    result.push_back(decl);
  }

  return result;
}
```

Поддержка классов и специальных функций:

- Hidden text

```
/// decl
///  ::= func-decl
///  ::= dtor-decl
///  ::= struct-decl
///  ::= class-decl
///
/// decls
///  ::= decl
///  ::= decls decl
///
/// struct-decl
///  ::= 'struct' identifier '{' decl-stmt * '}'
///
```

```
/// class-decl
///  ::= 'class' identifier ( 'extends' type ) ? '{' decls '}'
///
/// dtor-decl ::= ('virt' | 'impl')? 'del' '(' ')' block-stmt
/// ctor-decl
///   ::= 'new' '(' parameters-list ? ')' base-ctor-call ? blo
/// base-ctor-call ::= ':' 'super' '(' call-arguments ? ')'
SymbolList Parser::parseDecls(bool isClassMember) {
  SymbolList result;
  // Производим анализ всех доступных объявлений
  for (;;) {
    SymbolList tmp;
    llvm::SMLoc loc = CurPos.getLocation();

    switch (int Tok = CurPos->getKind()) {
      // Проверяем на виртуальные функции
      case tok::Virtual:
      case tok::Override:
        // Диагностируем об ошибке, если мы не разбираем класс
        if (!isClassMember) {
          getDiagnostics().report(CurPos.getLocation(),
            diag::ERR_VirtOverAsNonClassMember);
          return result;
        }
        // Пропускаем ключевое слово "virt" или "impl"
        ++CurPos;
        // Если это не "del", то мы должны обработать как обыч
        // функцию
        if (CurPos != tok::Delete) {
          --CurPos;
          tmp = parseFuncDecl(isClassMember);
          result.push_back(tmp.pop_back_val());
          continue;
        }
      // Это "del" (может быть виртуальным)
      case tok::Delete: {
        // Диагностируем об ошибке, если мы не разбираем класс
```

```
    if (!isClassMember) {
      getDiagnostics().report(CurPos.getLocation(),
        diag::ERR_UnexpectedDestructorDecl);
      return result;
    }
    // Проверяем наличие "del" "(" ")"
    check(tok::Delete);
    check(tok::OpenParen);
    check(tok::CloseParen);
    // Диагностируем об ошибке, если отсутствует тело
    // деструктора
    if (CurPos != tok::BlockStart) {
      getDiagnostics().report(CurPos.getLocation(),
                              diag::ERR_ExpectedDestructor
      return result;
    }
    // Считываем тело деструктора
    StmtAST *body = parseStmt();
    ParameterList params;
    // Создаем тип функции для деструктора и создание ветв
    // дерева
    TypeAST *funcType = new FuncTypeAST(
      BuiltinTypeAST::get(TypeAST::TI_Void),
      params
    );
    result.push_back(
        new FuncDeclAST(
          loc,
          funcType,
          Name::Dtor,
          body,
          true,
          Tok
        )
    );
    continue;
  }
```

```
case tok::Def:
  // Объявление функции. Считываем объявление функции и
  // добавляем в список объявлений
  tmp = parseFuncDecl(isClassMember);
  result.push_back(tmp.pop_back_val());
  continue;
// Проверяем на конструктор
case tok::New: {
  // Диагностируем об ошибке, если мы не разбираем класс
  if (!isClassMember) {
    getDiagnostics().report(CurPos.getLocation(),
      diag::ERR_UnexpectedConstructorDecl);
    return result;
  }

  ExprAST *superCtorCall = nullptr;
  ParameterList params;
  // Пропускаем "new" и проверяем наличие "("
  ++CurPos;
  check(tok::OpenParen);
  // Проверяем на наличие параметров
  if (CurPos != tok::CloseParen) {
    for (;;) {
      Name *paramName = nullptr;
      // Проверяем на наличие имени параметра
      if (CurPos == tok::Identifier) {
        // Сохраняем имя параметра переходим к следующем
        // токену
        paramName = CurPos->getIdentifier();
        ++CurPos;

        // Проверяем на то, что это анонимный параметр
        if (strcmp(paramName->Id, "_") == 0) {
          paramName = nullptr;
        }
      } else {
```

```
        // Диагностика ошибки, т. к. отсутствует
        // идентификатор
        check(tok::Identifier);
      }
      // Проверяем ":" и считываем тип параметра
      check(tok::Colon);

      TypeAST *type = parseType();
      // Добавляем параметр к списку параметров
      params.push_back(new ParameterAST(type, paramName)
      // Проверяем на наличие ","
      if (CurPos != tok::Comma) {
        // Нет, завершаем разбор параметров
        break;
      }
      // Считываем "," и переходим к следующему параметр
      ++CurPos;
    }
  }
  // Проверяем на наличие ")"
  check(tok::CloseParen);
  // Проверяем на наличие ":", для вызова конструктора
  // родительского класса
  if (CurPos == tok::Colon) {
    check(tok::Colon);
    // Диагностируем ошибку в случае отсутствия "super"
    if (CurPos != tok::Super) {
      getDiagnostics().report(CurPos.getLocation(),
        diag::ERR_ExpectedSuperAfterNew);
      return result;
    } else {
       // Пропускаем "super"
      ++CurPos;
      // Проверяем на наличие "(", для аргументов
      // конструктора
      if (CurPos != tok::OpenParen) {
        getDiagnostics().report(CurPos.getLocation(),
```

```
    diag::ERR_ExpectedFuncArguments);
  return result;
}
// Пропускаем ","
++CurPos;

ExprList args;

// Проверяем на наличие аргументов для конструктор
if (CurPos != tok::CloseParen) {
  for (;;) {
    // Считываем выражения для аргумента и добавля
    // его к списку аргументов
    ExprAST *arg = parseAssignExpr();
    args.push_back(arg);

    // Проверяем на наличие ","
    if (CurPos != tok::Comma) {
      // Нет, завершаем разбор аргументов
      break;
    }
    // Считываем "," и переходим к следующему
    // аргументу
    ++CurPos;
  }
}
// Проверяем на наличие "("
check(tok::CloseParen);
// Создаем выражение для вызова конструктора
// родительского класса
superCtorCall = new CallExprAST(
  loc,
  new MemberAccessExprAST(
    loc,
    new IdExprAST(loc, Name::Super),
    Name::Ctor
  ),
```

```
      args);
  }
}
// Проверка на наличие тела конструктора
if (CurPos != tok::BlockStart) {
  getDiagnostics().report(CurPos.getLocation(),
    diag::ERR_ExpectedConstructorBody);
  return result;
}
// Считываем тело конструктора
StmtAST *body = parseStmt();

// Создаем функцию с телом вида:
// {
//   opt super.__ctor(args);
//   body
// }
StmtList ctorBody;
// Добавляем вызов конструктора родительского класса (
// он был указан)
if (superCtorCall) {
  ctorBody.push_back(new ExprStmtAST(loc, superCtorCal
}
// Создаем ветку для объявления конструктора
ctorBody.push_back(body);
body = new BlockStmtAST(loc, ctorBody);
TypeAST *funcType = new FuncTypeAST(
  BuiltinTypeAST::get(TypeAST::TI_Void),
  params
);
result.push_back(
    new FuncDeclAST(
      loc,
      funcType,
      Name::Ctor,
      body,
      true,
```

```
        tok::New
      )
  );
  continue;
}
...
// Считываем объявление класса
case tok::Class: {
  // Пропускаем "class"
  ++CurPos;
  Name *name = nullptr;
  TypeAST *baseType = nullptr;

  if (CurPos == tok::Identifier) {
    // Сохраняем имя класса, если оно есть
    name = CurPos->getIdentifier();
  }
  // Проверяем наличие идентификатора или диагностируем
  // ошибке
  check(tok::Identifier);
  // Проверяем наличия указания родительского класса
  if (CurPos == tok::Extends) {
    // Пропускаем "extends"
    check(tok::Extends);
    // Диагностируем об ошибки, если это не идентификатс
    // точка
    if (CurPos != tok::Identifier && CurPos != tok::Dot)
      getDiagnostics().report(CurPos.getLocation(),
        diag::ERR_ExpectedQualifiedNameAsBase);
      return result;
    }
    // Считываем тип базового класса
    baseType = parseType();
  }
  // Проверяем на наличие тела класса
  check(tok::BlockStart);
  // Считываем все объявления внутри класса
```

```
        while (CurPos != tok::BlockEnd && CurPos != tok::EndOf
          SymbolList tmp2 = parseDecls(true);
          tmp.append(tmp2.begin(), tmp2.end());
        }
        // Проверяем наличие "}" и создаем ветвь дерева
        check(tok::BlockEnd);
        result.push_back(new ClassDeclAST(loc, name, baseType,
        continue;
      }
      // Проверяем на выходы из чтения объявлений
      case tok::BlockEnd:
      case tok::EndOfFile:
        break;

      default:
        if (isClassMember) {
          // Если это объявление класса, то считываем объявлен
          // переменных
          SymbolList tmp2 = parseDecl(true, isClassMember);
          result.append(tmp2.begin(), tmp2.end());
          continue;
        }
        break;
    }

    break;
  }

  return result;
}
```

Правка для разбора объявления модуля, для поддержки новой сигнатуры функции:

Hidden text

```
/// module-decl ::= decls
ModuleDeclAST *Parser::parseModule() {
  SymbolList decls = parseDecls(false);

  if (CurPos != tok::EndOfFile) {
    getDiagnostics().report(CurPos.getLocation(),
                            diag::ERR_ExpectedEndOfFile);
    return nullptr;
  }

  return new ModuleDeclAST(getDiagnostics(), decls);
}
```

Поддержка инициализации объектов класса с помощью вызова конструктора класса:

Hidden text

```
/// var-init
///   ::= '=' assign-expr
///   ::= '(' call-arguments ? ')'
/// var-decl ::= identifier ':' type var-init?
/// var-decls
///   ::= var-decl
///   ::= var-decls ',' var-decl
/// decl-stmt ::= var-decls ';'
SymbolList Parser::parseDecl(bool needSemicolon, bool isClassM
  SymbolList result;
```

```
// Производим анализ списка объявлений переменных
 for (;;) {
   llvm::SMLoc loc = CurPos.getLocation();

   if (CurPos != tok::Identifier) {
     // Объявление всегда должно начинаться с идентификатора,
     // это что-то другое, то сообщаем об ошибке
     getDiagnostics().report(CurPos.getLocation(),
                             diag::ERR_ExpectedIdentifierInDe
   } else {
     // Сохраняем имя переменной
     Name *name = CurPos->getIdentifier();
     ExprAST *value = nullptr;
    // Переходим к следующему токену
     ++CurPos;
     // Проверяем на наличие ":"
     check(tok::Colon);
     // Считываем тип переменной
     TypeAST *type = parseType();

     if (CurPos == tok::Assign) {
       // Запрещаем инициализаторы для массивов и структур/кл
       if (isa<ArrayTypeAST>(type) || isa<QualifiedTypeAST>(t
         getDiagnostics().report(CurPos.getLocation(),
           diag::ERR_AggregateOrArrayInitializer);
         return result;
       }
       // Если у переменной есть инициализатор, то считываем
       // инициализирующее значение
       ++CurPos;
       value = parseAssignExpr();
     }
     // Проверяем, что тип является именем класса и есть "("
     if (isa<QualifiedTypeAST>(type) && CurPos == tok::OpenPa
       // Пропускаем "("
       ++CurPos;
```

```
  ExprList args;

  // Проверяем на наличие аргументов для конструктора
  if (CurPos != tok::CloseParen) {
    for (;;) {
      // Считываем выражения для аргумента и добавляем е
      // списку аргументов
      ExprAST *arg = parseAssignExpr();
      args.push_back(arg);

      // Проверяем на наличие ","
      if (CurPos != tok::Comma) {
        // Нет, завершаем разбор аргументов
        break;
      }
      // Считываем "," и переходим к следующему аргумент
      ++CurPos;
    }
  }
  // Проверяем на наличие ")"
  check(tok::CloseParen);
  // Создаем вызов конструктора класса
  value = new CallExprAST(
    loc,
    new MemberAccessExprAST(
      loc,
      new IdExprAST(loc, name),
      Name::Ctor
    ),
    args);
}
// Добавляем новое объявление переменной
result.push_back(
  new VarDeclAST(loc, type, name, value, isClassMember)
);
// Проверяем на наличие ","
if (CurPos != tok::Comma) {
```

```
        // Нет, завершаем разбор объявлений переменных
        break;
      }
      // Считываем "," и переходим к следующему объявлению
      ++CurPos;
    }
  }
  // Считываем ";", если нужно
  if (needSemicolon) {
    check(tok::Semicolon);
  }

  return result;
}
```

Добавляем поддержку новых выражений:

Hidden text

```
/// block-stmt ::= '{' stmt* '}'
/// for-init
///   ::= 'let' decl-stmt
///   ::= expr
/// for-stmt ::= 'for' for-init? ';' expr? ';' expr? block-stm
/// stmt
///   ::= expr? ';'
///   ::= 'let' decl-stmt
///   ::= 'if' expr block-stmt ( 'else' block-stmt )?
///   ::= 'while' expr block-stmt
///   ::= for-stmt
///   ::= 'break'
```

```
///   ::= 'continue'
///   ::= 'return' expr? ';'
///   ::= block-stmt
StmtAST *Parser::parseStmt() {
  StmtAST *result = nullptr;
  llvm::SMLoc loc = CurPos.getLocation();

  switch (CurPos->getKind()) {
    ...
    case tok::Plus:
    case tok::Minus:
    case tok::PlusPlus:
    case tok::MinusMinus:
    case tok::Tilda:
    case tok::Not:
    case tok::Identifier:
    case tok::IntNumber:
    case tok::FloatNumber:
    case tok::Super:
    case tok::This:
    case tok::OpenParen:
    case tok::Delete:
    case tok::Mul:
    case tok::BitAnd:
    case tok::New:
    case tok::Dot: {
      ExprAST *expr = parseExpr();
      check(tok::Semicolon);
      return new ExprStmtAST(loc, expr);
    }
    ...
  }
}
```

Семантический анализ

Для начала рассмотрим правки, которые нужно внести в структуре дерева:

Добавить поддержку нового типа:

```
struct TypeAST {
  enum TypeId {
    TI_Void,     ///< void
    TI_Bool,     ///< булево значение
    TI_Int,      ///< int
    TI_Float,    ///< float
    TI_Char,     ///< char
    TI_String,   ///< string
    TI_Pointer,  ///< указатель
    TI_Array,    ///< массив
    TI_Class,    ///< класс
    TI_Struct,   ///< структура
    TI_Function, ///< функция
    TI_Qualified ///< квалифицированное имя (полное имя класса или ст
  };
  ...
  /// Проверка на то, что это тип структуры или класса
  bool isAggregate() {
    return TypeKind == TI_Class || TypeKind == TI_Struct;
  }
  ...
  /// Проверка на то, что данный тип является базовым для type
  virtual bool isBaseOf(TypeAST* type);
};
```

Для объявлений:

```
struct SymbolAST {
  enum SymbolId {
    SI_Variable,     ///< переменная
    SI_Struct,       ///< структура

    SI_Class,        ///< класс
    SI_Function,     ///< функция
    SI_Module,       ///< модуль
    SI_Parameter,    ///< параметр функции
    SI_OverloadSet, ///< набор перегруженных функций
    SI_Block         ///< блочная декларация
    SI_Variable,     ///< variable declaration
    SI_Struct,       ///< structure declaration
    SI_Function,     ///< function declaration
    SI_Module,       ///< module declaration
    SI_Parameter,    ///< function's parameter declaration
    SI_Block         ///< scope declaration
  };
  ...
  /// Проверка на то, что символ является структурой или классом
  bool isAggregate() {
    return SymbolKind == SI_Struct || SymbolKind == SI_Class;
  }
  ...
};
```

Добавить новую ветку для типов:

▾ Hidden text

```
struct ClassTypeAST : TypeAST {
  ClassTypeAST(SymbolAST* thisDecl) ;

  TypeAST* semantic(Scope* scope);
```

```
  bool implicitConvertTo(TypeAST* newType);
  void toMangleBuffer(llvm::raw_ostream& output);
  bool isBaseOf(TypeAST* type);
  SymbolAST* getSymbol();

  llvm::Type* getType();

  static bool classof(const TypeAST *T) {
    return T->TypeKind == TI_Class;
  }

  SymbolAST* ThisDecl; ///< Объявление данного класса
};
```

Поддержку методов класса:

- Hidden text

```
struct FuncTypeAST : TypeAST {
 ...
  bool HasThis; ///< true – Если это метод класса
};
```

Поддержка вызова конструкторов и деструкторов для вызовов "new" и "del":

- Hidden text

```
struct MemberAccessExprAST : ExprAST {
  MemberAccessExprAST(llvm::SMLoc loc, ExprAST *val, Name *mem
  MemberAccessExprAST(llvm::SMLoc loc, SymbolAST *aggrSym,
    ExprAST* forcedThis, Name* memberName) ;
  ...
  /// true – если "this" задан извне (используется для вызова
  /// конструкторов и деструкторов после вызова операторов
  /// "new" и "del")
  bool ForceThis;
};
```

Поддержка вызовов деструкторов в блоках:

▾ Hidden text

```
struct BlockStmtAST : StmtAST {
  ~BlockStmtAST() {
    for (StmtList::iterator it = Body.begin(), end = Body.end(
         it != end; ++it) {
      delete *it;
    }

    delete ThisBlock;
    delete LandingPad;

    for (ExprList::iterator it = CleanupList.begin(),
         end = CleanupList.end(); it != end; ++it) {
      delete *it;
    }
  }
 ...
```

```
  /// Список объектов для которых нужно вызвать деструкторы пс
  /// из блока
  ExprList CleanupList;
};
```

Поддержка объявления классов:

▾ Hidden text

```
/// Класс для хранения информации о виртуальных методах
struct VTableAST {
  VTableAST(SymbolAST* parent)
    : Parent(parent),
      CurOffset(0),
      CodeValue(nullptr),
      VTblType(nullptr) {
  }

  /// Проверка на то, что функция существует в таблице виртуал
  /// методов
  /// \param[in] func – функция для поиска
  bool isExist(SymbolAST* func);
  /// Создать копию таблицы виртуальных методов родительского
  VTableAST* clone(SymbolAST* parent);
  /// Добавить новый метод в таблицу виртуальных методов или
  /// заменить версию родительского класса
  void addOrReplace(Scope *scope, SymbolAST* func);
  /// Сгенерировать код для таблицы виртуальных методов
  /// \param[in] Context - контекст
  llvm::Value* generateCode(SLContext& Context);
```

```
  SymbolAST* Parent; ///< Класс владелец
  /// Таблица виртуальных методов
  typedef std::map< Name*, SymbolAST* > SymbolMap;
  SymbolMap Decls; ///< Объявленные виртуальные методы
  int CurOffset; ///< Последний свободный слот в таблице
  llvm::Value* CodeValue; ///< Сгенерированная таблица
  llvm::Type* VTblType; ///< Тип для сгенерированной таблицы

private:
  /// Конструктор
  /// \param[in] other — таблица для копирования
  /// \param[in] parent — класс владелец
  VTableAST(const VTableAST& other, SymbolAST* parent)
    : Parent(parent),
      Decls(other.Decls),
      CurOffset(other.CurOffset),
      CodeValue(nullptr),
      VTblType(nullptr) {
  }
};
```

Создать ветку дерева для класса:

```
struct ClassDeclAST : ScopeSymbol {
  /// Конструктор
  /// \param[in] loc — расположение в исходном коде
  /// \param[in] id — имя класса
  /// \param[in] baseClass — родительский класс (может быть nu
  /// \param[in] vars — список членов класса
  ClassDeclAST(llvm::SMLoc loc, Name *id, TypeAST *baseClass,
    const SymbolList& vars)
    : ScopeSymbol(loc, SI_Class, id),
      BaseClass(baseClass),
      Vars(vars),
      ThisType(nullptr),
      Ctor(false),
      Dtor(false),
      VTbl(nullptr),
      OwnVTable(false) {
```

```
  ThisType = new ClassTypeAST(this);
}

~ClassDeclAST() {
  for (SymbolList::iterator it = Vars.begin(), end = Vars.er
    it != end; ++it) {
    delete *it;
  }

  if (OwnVTable) {
    delete VTbl;
  }
}

TypeAST* getType();
void doSemantic(Scope* scope);
void doSemantic2(Scope* scope);
void doSemantic3(Scope* scope);
void doSemantic4(Scope* scope);
void doSemantic5(Scope* scope);
bool contain(TypeAST* type);

llvm::Value* getValue(SLContext& Context);
llvm::Value* generateCode(SLContext& Context);

SymbolAST* find(Name* id, int flags = 0);

static bool classof(const SymbolAST *T) {
  return T->SymbolKind == SI_Class;
}

TypeAST* BaseClass; ///< базовый класс
SymbolList Vars; ///< список членов класса
TypeAST* ThisType; ///< сгенерированный тип
bool Ctor; ///< true – если был объявлен хотя бы один констр
bool Dtor; ///< true – если был объявлен деструктор
/// таблица виртуальных методов (может быть nullptr)
```

```
  VTableAST* VTbl;
  /// true – если имеет свою VTable (отличную от родительского
  /// класса)
  bool OwnVTable;
};
```

Поддержку методов класса:

▾ Hidden text

```
struct FuncDeclAST : ScopeSymbol {
  FuncDeclAST(llvm::SMLoc loc, TypeAST *funcType, Name *id,
    StmtAST* body, bool inClass, int tok = tok::Def) ;
  ...
  /// Это объявление виртуального метода
  bool isVirtual() {
    return Tok == tok::Virtual;
  }
  /// Это объявление переопределенного виртуального метода
  bool isOverride() {
    return Tok == tok::Override;
  }
  /// Конструктор класса
  bool isCtor() {
    return Id == Name::Ctor;
  }
  /// Деструктор класса
  bool isDtor() {
    return Id == Name::Dtor;
  }
```

```
  bool needThis();
  ...
  bool NeedThis; ///< true — метод класса
  int OffsetOf; ///< индекс метода в таблице виртуальных метод
  SymbolAST* AggregateSym; ///< родительский класс
};
```

Поддержка перегрузки функций:

▾ Hidden text

```
struct OverloadSetAST : SymbolAST {
  OverloadSetAST(Name* name);

  /// Добавить новую функцию к списку перегруженных функций
  void push(Scope *scope, SymbolAST* func);

  OverloadSetAST* clone();

  static bool classof(const SymbolAST *T) {
    return T->SymbolKind == SI_OverloadSet;
  }

  SymbolList Vars; ///< список всех перегруженных функций
  /// набор уникальных перегрузок, для исключения дублей
  llvm::StringSet< > OverloadsUsed;
};
```

Правки для LandingPadAST (более подробно будем рассматривать дальше):

```
struct LandingPadAST {
  LandingPadAST(LandingPadAST* prev, bool needCleanup);
  LandingPadAST();
  ...
  /// Получить переменную со статусом очистки
  llvm::Value* getCleanupValue();
  ...
  bool NeedCleanup; ///< true – если нужно произвести очистку
  /// статус очистки (необходимо вызывать getCleanupValue)
  llvm::Value* CleanupValue;
  llvm::BasicBlock* CleanupLoc; ///< блок очистки
};
```

Семантика типов

Для уже ранее созданных веток дерева для типов, нужно добавить поддержку семантической обработки нового функционала — наследования и классов:

▾ Hidden text

```
bool TypeAST::isBaseOf(TypeAST* ) {
  return false;
}

TypeAST* ArrayTypeAST::semantic(Scope* scope) {
  // Проверить семантику базового типа
  Next = Next->semantic(scope);

  // Запрещаем массивы объектов классов
  if (isa<ClassTypeAST>(Next)) {
```

```
    scope->report(SMLoc(), diag::ERR_SemaArrayOfClassUnsupport
    return nullptr;
  }

  // создаем MangleName
  calcMangle();
  return this;
}


bool PointerTypeAST::implicitConvertTo(TypeAST* newType) {
  // Разрешаем преобразование указателя в bool
  if (newType->isBool()) {
    return true;
  }

  // Разрешаем преобразование char* в string
  if (Next->isChar() && newType->isString()) {
    return true;
  }

  // Запрещаем любое преобразование в тип отличный от указател
  if (!isa<PointerTypeAST>(newType)) {
    return false;
  }

  PointerTypeAST* ptr2 = (PointerTypeAST*)newType;
  // Получаем базовый тип для newType
  TypeAST* next2 = ptr2->Next;

  // Разрешаем преобразование, если типы совпадают
  if (Next->equal(next2)) {
    return true;
  }

  // Если оба типа на которые указывают указатели являются
  // агрегатами, то нужно проверить, что тип, в который нужно
```

```
  // произвести преобразование является базовым для текущего
  if (Next->isAggregate() && next2->isAggregate()) {
    return next2->isBaseOf(Next);
  }

  TypeAST* next1 = Next;

  // Разрешаем преобразование указателя любого типа в void*
  if (next2->isVoid()) {
    return true;
  }

  return false;
}

bool StructTypeAST::isBaseOf(TypeAST* type) {
  // Только классы могут иметь родительский класс или структур
  if (!isa<ClassTypeAST>(type)) {
    return false;
  }

  ClassTypeAST* classType = (ClassTypeAST*)type;
  ClassDeclAST* classDecl = (ClassDeclAST*)classType->ThisDecl

  // Если этот тип и базовый совпадают, то возвращаем "true"
  if (classDecl->BaseClass && this->equal(classDecl->BaseClass
    return true;
  }

  // Если класс имеет базовый класс, то вызываем метод повторн
  // но для родительского класса
  if (classDecl->BaseClass) {
    return isBaseOf(classDecl->BaseClass);
  }
```

```
  return false;
}
```

Добавляем тип для классов:

▾ Hidden text

```
TypeAST* ClassTypeAST::semantic(Scope* scope) {
  calcMangle();
  return this;
}

bool ClassTypeAST::implicitConvertTo(TypeAST* newType) {
  if (newType->equal(this)) {
    return true;
  }

  return false;
}

void ClassTypeAST::toMangleBuffer(llvm::raw_ostream& output) {
  mangleAggregateName(output, ThisDecl);
}

bool ClassTypeAST::isBaseOf(TypeAST* type) {
  // Только классы могут иметь родительский класс или структур
  if (!isa<ClassTypeAST>(type)) {
    return false;
  }

  ClassTypeAST* classType = (ClassTypeAST*)type;
```

```
  ClassDeclAST* classDecl = (ClassDeclAST*)classType->ThisDecl

  // Если этот тип и родительский совпадают, то возвращаем "tr
  if (classDecl->BaseClass && this->equal(classDecl->BaseClass
    return true;
  }

  // Если класс имеет базовый класс, то вызываем метод повторн
  // но для родительский класса
  if (classDecl->BaseClass) {
    return isBaseOf(classDecl->BaseClass);
  }

  return false;
}

SymbolAST* ClassTypeAST::getSymbol() {
  return ThisDecl;
}
```

Для функций нужно добавить поддержку методов класса:

▾ Hidden text

```
void FuncTypeAST::toMangleBuffer(llvm::raw_ostream& output) {
  // Добавляем "v", если у функции нет параметров
  if (Params.empty()) {
    output << "v";
    return;
  }
```

```
  ParameterList::iterator it = Params.begin(), end = Params.er
  // Пропускаем параметр для "this"
  if (HasThis) {
    ++it;
  }
  // Добавляем "v", если у функции нет параметров, кроме "this
  if (it == end) {
    output << "v";
    return;
  }
  // Произвести декорацию имен для всех параметров
  for ( ; it != end; ++it) {
    (*it)->Param->toMangleBuffer(output);
  }
}
```

Семантика выражений для поддержки классов и перегрузки функций

Поддержка классов для оператора "new":

- Hidden text

```
ExprAST* NewExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ для типа
  NewType = NewType->semantic(scope);

  // Проверяем на то, что это массив с фиксированным размером
```

```
if (isa<ArrayTypeAST>(NewType) && !DynamicSize) {
  // Мы выделяем память как для массива, но результирующий т
  // будет указателем
  ArrayTypeAST* arrayType = (ArrayTypeAST*)NewType;
  ExprType = new PointerTypeAST(arrayType->Next, false);
  ExprType = ExprType->semantic(scope);
} else {
  // Результирующий тип будет указателем
  ExprType = new PointerTypeAST(NewType, false);
  ExprType = ExprType->semantic(scope);
}

// Создаем новое выражение для хранения размера NewType на ц
// платформе и оборачиваем его в прокси, т. к. реальный разм
// будет известен только во время генерации кода
SizeExpr = new IntExprAST(Loc, 0);
ExprAST* proxy = new ProxyExprAST(Loc, SizeExpr);

if (DynamicSize) {
  proxy = new BinaryExprAST(Loc, tok::Mul, DynamicSize, prox
}

// Создаем вызов функции allocMem для выделения блока памяти
// нужного размера
ExprList args;
args.push_back(proxy);
CallExpr = new CallExprAST(
  Loc,
  new IdExprAST(Loc, Name::New),
  args
);

// Для классов мы должны произвести вызов конструктора, если
// есть
if (isa<ClassTypeAST>(NewType)) {
  ClassDeclAST* classDecl = (ClassDeclAST*)NewType->getSymbo
```

```
    if (classDecl->Ctor) {
      // Если класс имеет конструктор, то нужно сгенерировать
      // вызов
      // Замечание: в качестве "this" для вызова конструктора,
      // должны использовать результат вызова "allocMem"
      CallExpr = new CallExprAST(
        Loc,
        new MemberAccessExprAST(
          Loc,
          classDecl,
          CallExpr,
          Name::Ctor
        ),
        Args);
      // Нам не нужно производить преобразование типа, т. к.
      // конструктор вернет указатель нужного типа
      NeedCast = false;
    }
  }

  // Производим семантический анализ только что созданного выз
  // функции
  CallExpr = CallExpr->semantic(scope);

  return this;
}
```

Поддержка классов для оператора "del":

- Hidden text

```
ExprAST* DeleteExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (DeleteCall) {
    return this;
  }

  // Производим семантический анализ выражения для очистки
  Val = Val->semantic(scope);

  // Запрещаем удаление выражений с типом void
  if (!Val->ExprType || Val->ExprType->isVoid()) {
    scope->report(Loc, diag::ERR_SemaCantDeleteVoid);
    return nullptr;
  }

  // Мы можем производить очистку только для указателей
  if (!isa<PointerTypeAST>(Val->ExprType)) {
    scope->report(Loc, diag::ERR_SemaCantDeleteNonPointer);
    return nullptr;
  }

  PointerTypeAST* ptrType = (PointerTypeAST*)Val->ExprType;

  ExprAST* deleteArg = Val->clone();
  ExprList args;

  // Для классов с деструкторами, мы должны произвести его выз
  if (isa<ClassTypeAST>(ptrType->Next)) {
    ClassDeclAST* classDecl =
      (ClassDeclAST*)ptrType->Next->getSymbol();

    if (classDecl->Dtor) {
      // Создаем вызов деструктора
      // Замечание: В качестве "this" мы используем результат
      // кодогенерации для Val
      deleteArg = new CallExprAST(
```

```
        Loc,
        new MemberAccessExprAST(
          Loc,
          classDecl,
          deleteArg,
          Name::Dtor
        ),
        args);
    }
  }

  // Создаем список аргументов для вызова freeMem
  // Замечание: Это может быть либо Val, либо результат вызова
  // деструктора
  args.push_back(deleteArg);

  // Создание вызова freeMem и проверка его семантики
  DeleteCall = new CallExprAST(
    Loc,
    new IdExprAST(Loc, Name::Delete),
    args
  );
  DeleteCall = DeleteCall->semantic(scope);

  ExprType = BuiltinTypeAST::get(TypeAST::TI_Void);

  return this;
}
```

Поддержка обращения к членам класса без явного указания объекта класса в его методах:

Hidden text

```
ExprAST* IdExprAST::semantic(Scope* scope) {
  // Если "ThisSym" задан, то семантический анализ над данным
  // выражением уже был ранее завершен
  if (!ThisSym) {
    // Ищем объявление в текущей области видимости
    ThisSym = scope->find(Val);

    if (!Val) {
      return this;
    }

    if (!ThisSym) {
      // Объявление не найдено, возвращаем ошибку
      scope->report(Loc, diag::ERR_SemaUndefinedIdentifier, Va
      return nullptr;
    }
    // Если это член класса, то мы должны произвести специальн
    // обработку
    if (ThisSym->needThis()) {
      // Заменить выражение на this . a
      IdExprAST* thisExpr = new IdExprAST(Loc, Name::This);
      PointerAccessExprAST* memberExpr =
        new PointerAccessExprAST(Loc, thisExpr, Val);
      // Производим семантический анализ нового выражения и уд
      // старое
      ExprAST* resultExpr = memberExpr->semantic(scope);
      delete this;
      return resultExpr;
    }
    // Устанавливаем тип данного выражения в соответствии с ти
    // объявленной переменной (за исключением набора перегруже
    // функций, т. к. реальный тип мы можем узнать только во в
    // вызова функции)
```

```
    if (!isa<OverloadSetAST>(ThisSym)) {
      ExprType = ThisSym->getType();
    }
  }

  return this;
}
```

Поддержка новых возможностей для обращения к методам и переменным агрегатов:

▾ Hidden text

```
ExprAST* MemberAccessExprAST::semantic(Scope* scope) {
  // Исключаем повторный запуск семантического анализа
  if (SemaDone) {
    return this;
  }
  // Проверяем семантику левого операнда
  Val = Val->semantic(scope);

  // Специальная обработка для символа в глобальной области
  // видимости
  if (isa<IdExprAST>(Val) && !((IdExprAST*)Val)->Val) {
    // Получить объявление модуля
    SymbolAST* moduleSym = scope->find(0);
    assert(moduleSym);
    // Поиск объявления в модуле
    SymbolAST* thisSym = moduleSym->find(MemberName);

    // Диагностика ошибки, если символ не найден
```

```
  if (!thisSym) {
    scope->report(Loc, diag::ERR_SemaUndefinedMember,
                  MemberName->Id);
    return nullptr;
  }

  // Заменяем MemberAccessExprAST на IdExprAST с
  // предопределенными значениями
  IdExprAST* newExpr = new IdExprAST(Loc, MemberName);

  // Устанавливаем тип найденного символа (за исключением
  // набора перегруженных функций, т. к. реальный тип мы мож
  // узнать только во время вызова функции)
  if (!isa<OverloadSetAST>(thisSym)) {
    newExpr->ExprType = thisSym->getType();
  }

  newExpr->ThisSym = thisSym;
  delete this;
  return newExpr;
}

// Специальная обработка для вариантов, когда "this", был яв
// указан извне
if (ForceThis) {
  // Проверяем, что это агрегат
  if (!ThisAggr->isAggregate()) {
    scope->report(Loc, diag::ERR_SemaNonAggregateForFocedThi
    return nullptr;
  }

  // Пытаемся найти символ в агрегате
  ThisSym = ThisAggr->find(MemberName);
  // Диагностируем об ошибке, в случае его отсутствия
  if (!ThisSym) {
    scope->report(Loc, diag::ERR_SemaUndefinedMember,
                  MemberName->Id);
```

```
    return nullptr;
  }
  // Устанавливаем тип найденного символа (за исключением на
  // перегруженных функций, т. к. реальный тип мы можем узна
  // только во время вызова функции)
  if (!isa<OverloadSetAST>(ThisSym)) {
    ExprType = ThisSym->getType();
  }

  SemaDone = true;
  return this;
}

// Проверяем на корректность типа операнда
if (!Val->ExprType) {
  scope->report(Loc, diag::ERR_SemaInvalidOperandForMemberAc
  return nullptr;
}

// Если Val является указателем на агрегат, то нам нужно
// произвести разыменование указателя. Т.е. в семантики C++,
// мы считаем agg->a эквивалентом (*agg).a
if (isa<PointerTypeAST>(Val->ExprType) &&
  ((PointerTypeAST*)Val->ExprType)->Next->isAggregate()) {
  Val = new DerefExprAST(Loc, Val);
  Val = Val->semantic(scope);
}

// Val должно быть lvalue
if (!Val->isLValue()) {
  scope->report(Loc, diag::ERR_SemaNonLValueForMemberAccess)
  return nullptr;
}

// Получить агрегат
SymbolAST* aggSym = Val->getAggregate(scope);
```

```
// Запрещаем использование A.b, где A — класс или структура.
// Это должно быть a.b, где a экземпляр A
if (isa<IdExprAST>(Val) && aggSym && aggSym->isAggregate())
  // Check special case for base class function or variable
  scope->report(Loc, diag::ERR_SemaMemberAccessOnAggregateTy
  return nullptr;
}

// Исключаем операции над не агрегатами
if (!aggSym || !aggSym->getType()->isAggregate()) {
  scope->report(Loc, diag::ERR_SemaNonAggregateDotOperand);
  return nullptr;
}

// Получить объявление агрегата
if (!aggSym->isAggregate()) {
  aggSym = aggSym->getType()->getSymbol();
}

// Поиск символа в агрегате
ThisSym = aggSym->find(MemberName);
ThisAggr = aggSym;

// Проверка на то, что данный агрегат имеет член с таким име
if (!ThisSym) {
  scope->report(Loc, diag::ERR_SemaUndefinedMember,
                MemberName->Id);
  return nullptr;
}

// Запрет конструкции вида a.A, где A это агрегат
if (ThisSym->isAggregate()) {
  scope->report(Loc,
                diag::ERR_SemaAggregateTypeAsMemberAccessOpe
  return nullptr;
}
// Устанавливаем тип результата (за исключением набора
```

```
  // перегруженных функций, т. к. реальный тип мы можем узнать
  // только во время вызова функции)
  if (!isa<OverloadSetAST>(ThisSym)) {
    ExprType = ThisSym->getType();
  }

  SemaDone = true;

  return this;
}
```

Поддержка клонирования с учетом новой логики работы обращению к членам агрегата:

▾ Hidden text

```
ExprAST* MemberAccessExprAST::clone() {
  if (ForceThis) {
    return new MemberAccessExprAST(Loc, ThisAggr, Val->clone()
                                   MemberName);
  } else {
    return new MemberAccessExprAST(Loc, Val->clone(), MemberNa
  }
}
```

Выбор функции для вызова при перегрузке функций

Отдельно рассмотрим метод определения того, какая функция из набора перегруженных функций должна быть выбрана, при вызове. Для этого пройдем по всем шагам, которые нам нужны для этого.

Для начала нам нужно объявить два типа перечислений.

1. Для указания того что функция или параметр функции является лучшим, худшим вариантом из 2-х предоставленных или выбор не может быть сделан (например варианты равнозначны);
2. Для типа преобразования на основе которого уже можно будет производить сравнение параметров.

```
enum class OverloadCompareKind {
  Better = -1,      // функция/параметр является лучшим кандидатом
  Unknown = 0, // определить нельзя
  Worse = 1        // функция/параметр является худшим кандидатом
};

enum class OverloadConversionType {
  Same = 0,                  ///< типы параметра и аргумента идентичны
  /// преобразование является приведения указателя к указателю на "vo
  PointerToVoid = 1,
  /// преобразование является приведение дочернего типа к родительско
  DerivedToBase = 2,
  /// базовое преобразование языка (например int -> float)
  Implicit = 3,
};
```

Дальше нам нужен механизм для определения того, какой тип преобразования необходим для преобразования типа аргумента для вызова к типу параметра функции:

Hidden text

```
static OverloadConversionType getConversionLevel(
  TypeAST *arg, TypeAST *param) {
  // Типы совпадают
  if (arg->equal(param)) {
    return OverloadConversionType::Same;
  }

  // Проверка для указателей
  if (isa<PointerTypeAST>(param)) {
    TypeAST* ptr1 = ((PointerTypeAST*)param)->Next;
    TypeAST* ptr2 = ((PointerTypeAST*)arg)->Next;

    // Проверяем на преобразование указателя в указатель на "v
    if (ptr1->isVoid()) {
      return OverloadConversionType::PointerToVoid;
    }
    // Если это агрегатный тип, то необходимо проверить привед
    // родительскому типу
    if (ptr1->isAggregate() && ptr1->isBaseOf(ptr2)) {
      return OverloadConversionType::DerivedToBase;
    }
  }
  // Базовое преобразование
  return OverloadConversionType::Implicit;
}
```

Дальше нам нужен механизм проверки одного параметра 2-х кандидатов:

Hidden text

```
static OverloadCompareKind compareConversion(
  TypeAST *arg,
  TypeAST *left,
  TypeAST *right
) {
  // Получаем типа преобразования для каждого кандидата
  OverloadConversionType
    Level1 = getConversionLevel(arg, left),
    Level2 = getConversionLevel(arg, right);
  // Проверяем особые варианты, если типы преобразования не
  // совпадают для обоих кандидатов
  if (Level1 != Level2) {
    // Если типы аргумента и параметра 1-го кандидата совпада
    // то 1-й кандидат лучше
    if (Level1 == OverloadConversionType::Same) {
      return OverloadCompareKind::Better;
    }
    // Если типы аргумента и параметра 2-го кандидата совпада
    // то 1-й кандидат хуже
    if (Level2 == OverloadConversionType::Same) {
      return OverloadCompareKind::Worse;
    }
  }

  // Сравниваем типы преобразований обоих кандидатов с
  // преобразованием указателя в указатель на "void"
  bool ConversionToVoid1 =
    Level1 == OverloadConversionType::PointerToVoid;
  bool ConversionToVoid2 =
    Level2 == OverloadConversionType::PointerToVoid;

  // Проверяем, что один из кандидатов имеет лучшее преобразов
  // чем преобразование указателя к указателю на "void"
```

```
if (ConversionToVoid1 != ConversionToVoid2) {
  // Если 2-й кандидат является преобразованием указателя к
  // указателю на "void", то 1-й кандидат является лучшим,
  // иначе 1-й кандидат хуже
  return ConversionToVoid2
    ? OverloadCompareKind::Better
    : OverloadCompareKind::Worse;
}

// Сравниваем типы преобразований обоих кандидатов с
// преобразованием указателя на дочерний класс в указатель н
// родительский класс
bool ConversionToBase1 =
  Level1 == OverloadConversionType::DerivedToBase;
bool ConversionToBase2 =
  Level2 == OverloadConversionType::DerivedToBase;

// Проверяем вариант, когда типы преобразований обоих кандид
// является преобразованием дочернего класса к родительскому
// классу
if (ConversionToBase1 && ConversionToBase1 == ConversionToBa
  TypeAST *to1 = ((PointerTypeAST*)left)->Next;
  TypeAST *to2 = ((PointerTypeAST*)right)->Next;

  // Если оба типа совпадают, то мы не можем однозначно опре
  // какой из кандидатов лучше
  if (to1->equal(to2)) {
    return OverloadCompareKind::Unknown;
  }
  // Замечание: Если у нас есть иерархия типов C -> B -> A,
  // преобразование C -> B лучше, чем преобразование C -> A
  // Проверяем на C -> A
  if (to1->isBaseOf(to2)) {
    return OverloadCompareKind::Worse;
  }
  // Это C -> B
  return OverloadCompareKind::Better;
```

```
  }
  // Мы не можем однозначно определить какой из кандидатов луч
  return OverloadCompareKind::Unknown;
}
```

Теперь, когда мы можем сравнивать преобразование для параметра обоих кандидатов между собой, мы можем написать алгоритм, для сравнения двух функций между собой:

▾ Hidden text

```
static bool isMoreSpecialized(
  CallExprAST *Call, SymbolAST* func1, SymbolAST* func2) {
  // Получаем типы обоих кандидатов
  FuncTypeAST* type1 = (FuncTypeAST*)((FuncDeclAST*)func1)->Th
  FuncTypeAST* type2 = (FuncTypeAST*)((FuncDeclAST*)func2)->Th

  // Получаем список аргументы и типы для вызова
  ExprList::iterator arg = Call->Args.begin();
  ParameterList::iterator it1 = type1->Params.begin();
  ParameterList::iterator it2 = type2->Params.begin();
  ParameterList::iterator end = type1->Params.end();

  // Если это функции методы класса, то игнорируем "this" при
  // сравнении кандидатов
  if (type1->HasThis) {
    ++arg;
    ++it1;
    ++it2;
  }
```

```
  // Проверяем все параметры кандидатов
  bool IsWorse = false;
  bool IsBetter = false;

  for ( ; it1 != end; ++arg, ++it1, ++it2) {
    OverloadCompareKind Kind = compareConversion(
      (*arg)->ExprType,
      (*it1)->Param,
      (*it2)->Param);
    // Определяем является ли текущий параметром первого канди
    // лучшим или худшим вариантом для вызова
    if (Kind == OverloadCompareKind::Better) {
      IsBetter = true;
    } else if (Kind == OverloadCompareKind::Worse) {
      IsWorse = true;
    }
  }

  // Если кандидат имеет хотя бы одно преобразование лучше, че
  // кандидат и не имеет ни одного преобразования хуже, то это
  // лучший кандидат для вызова
  if (IsBetter && !IsWorse) {
    return true;
  }

  // Либо хуже, либо оба кандидаты равнозначны
  return false;
}
```

Теперь после подготовительных работ, мы можем реализовать выбор функции для вызова, если эта функция имеет несколько перегруженных версий по параметрам:

Hidden text

```
/// Предикат для сортировки кандидатов на вызов
struct OverloadSetSort: std::binary_function<
  SymbolAST*, SymbolAST*, bool
> {
  OverloadSetSort(CallExprAST *args): Args(args) {}
  OverloadSetSort(const OverloadSetSort &right): Args(right.Ar

  bool operator()(SymbolAST* left, SymbolAST* right) const {
    // Мы должны переместить все более специализированные функ
    // начало. Для этого мы должны произвести обмен элементов
    // только, если левый кандидат лучше, чем 2-й и 2-й не луч
    // чем первый
    return isMoreSpecialized(Args, left, right) &&
      !isMoreSpecialized(Args, right, left);
  }

private:
  CallExprAST* Args;
};

static SymbolAST* resolveFunctionCall(
  Scope *scope, SymbolAST* func, CallExprAST* args) {
  // Набор потенциальных кандидатов для вызова
  SymbolList listOfValidOverloads;

  // У нас есть специальная обработка для варианта, когда есть
  // только один кандидат для вызова
  if (isa<FuncDeclAST>(func)) {
    FuncDeclAST* fnc = static_cast< FuncDeclAST* >(func);
    FuncTypeAST* type = static_cast< FuncTypeAST* >(fnc->ThisT

    // Количество аргументов должно совпадать с количеством
    // параметров у функции
```

```
  if (args->Args.size() != type->Params.size()) {
    scope->report(args->Loc,
                  diag::ERR_SemaInvalidNumberOfArgumentsInCa
    return nullptr;
  }

  ExprList::iterator arg = args->Args.begin();
  ParameterList::iterator it = type->Params.begin();

  // Если это вызов метода класса, то игнорируем параметр дл
  // "this"
  if (isa<MemberAccessExprAST>(args->Callee)) {
    ++it;
    ++arg;
  }

  // Проверяем все аргументы
  for (ParameterList::iterator end = type->Params.end();
    it != end; ++it, ++arg) {
    // Проверяем, что аргумент может быть использован для вы
    // и диагностируем об ошибке, если нет
    if (!(*arg)->ExprType->implicitConvertTo((*it)->Param))
      scope->report(args->Loc,
                    diag::ERR_SemaInvalidTypesOfArgumentsInC
      return nullptr;
    }
  }

  // Вызов функции может быть произведен с данными аргумента
  return func;
}

// Это набор перегруженных функций, необходимо произвести вы
// лучшего кандидата для вызова
OverloadSetAST* overloadSet = (OverloadSetAST*)func;
size_t numArgs = args->Args.size();
```

```
// Проверяем все функции из набора
for (SymbolList::iterator it = overloadSet->Vars.begin(),
  end = overloadSet->Vars.end(); it != end; ++it) {
  FuncDeclAST* fnc = static_cast< FuncDeclAST* >(*it);
  FuncTypeAST* type = static_cast< FuncTypeAST* >(fnc->ThisT

  // Если количество параметров и аргументов не совпадают, т
  // игнорируем данную функцию-кандидат
  if (numArgs != type->Params.size()) {
    continue;
  }

  ExprList::iterator arg = args->Args.begin();
  ParameterList::iterator param = type->Params.begin();

  // Если это вызов метода класса, то игнорируем параметр дл
  // "this"
  if (isa<MemberAccessExprAST>(args->Callee)) {
    ++param;
    ++arg;
  }

  // Помечаем кандидат как пригодный и полное совпадение тип
  // аргументов и параметров
  bool isValid = true;
  bool isExact = true;

  // Проверяем каждый аргумент
  for (ParameterList::iterator end = type->Params.end();
    param != end; ++param, ++arg) {
    // Проверяем, что типы совпадают
    if (!(*arg)->ExprType->equal((*param)->Param)) {
      // Типы не совпадают, помечаем это
      isExact = false;

      // Проверяем, что есть доступное преобразование для ти
      if (!(*arg)->ExprType->implicitConvertTo((*param)->Par
```

```
        // Кандидат не пригоден для вызова, помечаем и игнор
        // все последующие аргументы
        isValid = false;
        break;
      }
    }
  }

  // Предпочитаем полное совпадение по типам
  if (isExact) {
    return fnc;
  }

  // Добавляем фукнции-кандидаты, которые пригодны для вызов
  if (isValid) {
    listOfValidOverloads.push_back(fnc);
  }
}

// Если у нас больше чем 1 кандидат, необходимо это обработа
if (listOfValidOverloads.size() > 1) {
  // Сортируем все кандидаты в порядке их пригодности
  std::sort(
    listOfValidOverloads.begin(),
    listOfValidOverloads.end(),
    OverloadSetSort(args)
  );

  SymbolList::iterator it1 = listOfValidOverloads.begin();
  SymbolList::iterator it2 = listOfValidOverloads.begin() +

  // Если 1-я функция-кандидат лучше, чем 2-я и 2-я не лучше
  // то необходимо удалить из набора все функции, кроме 1-й.
  // Иначе сообщаем об ошибке ниже
  if (isMoreSpecialized(args, *it1, *it2) &&
    !isMoreSpecialized(args, *it2, *it1)) {
    listOfValidOverloads.erase(it2, listOfValidOverloads.enc
```

```
    }
  }

  // Если у нас осталось больше 1-й функции, то сообщаем об ош
  if (listOfValidOverloads.size() > 1) {
    scope->report(args->Loc, diag::ERR_SemaAmbiguousCall);
    return nullptr;
  // Если у нас не осталось ни 1-й функции, то сообщаем об оши
  } else if (listOfValidOverloads.empty()) {
    scope->report(args->Loc,
                  diag::ERR_SemaInvalidNumberOfArgumentsInCall
    return nullptr;
  }

  // Осталась только одна, возвращаем ее
  return listOfValidOverloads.pop_back_val();
}

ExprAST* CallExprAST::semantic(Scope* scope) {
  if (ExprType) {
    return this;
  }

  // Производим семантический анализ выражения до "("
  Callee = Callee->semantic(scope);

  // Мы можем использовать только IdExprAST и MemberAccessExpr
  // в качестве выражения для вызова
  if (isa<IdExprAST>(Callee) || isa<MemberAccessExprAST>(Calle
    SymbolAST* sym = (isa<IdExprAST>(Callee)
      ? ((IdExprAST*)Callee)->ThisSym
      : ((MemberAccessExprAST*)Callee)->ThisSym);

    // Val должна ссылаться на функцию или набор перегруженных
    // функций
    if (isa<FuncDeclAST>(sym) || isa<OverloadSetAST>(sym)) {
      TypeAST* returnType = nullptr;
```

```
// Специальная обработка для методов класса
if (isa<MemberAccessExprAST>(Callee)) {
  MemberAccessExprAST* memAccess =
    (MemberAccessExprAST*)Callee;

  // Необходимо добавить аргумент для параметра "this"
  ExprAST* thisArg = memAccess->Val->clone();
  Args.insert(Args.begin(), thisArg);

  // Для методов, у которых предопределен тип возвращаем
  // значения мы также должны указать тип результата
  if (memAccess->ForceThis) {
    returnType = memAccess->Val->ExprType;
  }
}

// Производим семантический анализ для всех аргументов
// функции
for (ExprList::iterator arg = Args.begin(), end = Args.e
  arg != end; ++arg) {
  *arg = (*arg)->semantic(scope);
}

// Ищем функцию для вызова
if (SymbolAST* newSym = resolveFunctionCall(scope, sym,
  FuncDeclAST* fnc = static_cast< FuncDeclAST* >(newSym)
  FuncTypeAST* type =
    static_cast< FuncTypeAST* >(fnc->ThisType);

  ExprList::iterator arg = Args.begin();
  ParameterList::iterator it = type->Params.begin();

  // Для вызова метода класса игнорируем аргумент для
  // параметра "this"
  if (isa<MemberAccessExprAST>(Callee)) {
    ++arg;
```

```
        ++it;
      }

      // Производим сопоставление аргументов и параметров
      for (ParameterList::iterator end = type->Params.end();
        it != end; ++it, ++arg) {
        // Если тип аргумента отличается от типа параметра,
        // производим преобразование типа
        if (!(*arg)->ExprType->equal((*it)->Param)) {
          ExprAST* oldArg = (*arg);
          *arg = new CastExprAST(
            oldArg->Loc,
            oldArg->clone(),
            (*it)->Param
          );
          *arg = (*arg)->semantic(scope);
          delete oldArg;
        }
      }

      // Определяем тип возвращаемого значения и устанавлива
      // тип для результата вызова функции
      if (!returnType) {
        ExprType = ((FuncDeclAST*)newSym)->ReturnType;
      } else {
        ExprType = returnType;
      }

      // Устанавливаем объявление функции для вызова для
      // дальнейшей работы
      CallFunc = newSym;
      return this;
    }
  }
}

// Диагностируем ошибку
```

```
    scope->report(Loc, diag::ERR_SemaInvalidArgumentsForCall);
    return nullptr;
  }

  ExprAST* CallExprAST::clone() {
    ExprList exprs;
    ExprList::iterator it = Args.begin();
    ExprList::iterator end = Args.end();

    // Для методов члена мы должны игнорировать аргумент для
    // параметра "this", т. к. он будет повторно добавлен во
    // время проверки семантики
    if (isa<MemberAccessExprAST>(Callee)) {
      ++it;
    }

    for ( ; it != end; ++it) {
      exprs.push_back((*it)->clone());
    }

    return new CallExprAST(Loc, Callee->clone(), exprs);
  }
```

Поддержка автоматического вызова деструкторов, при выходе объектов класса у которых есть деструктор, из области видимости их объявления. Для упрощения генерации кода любое объявление объекта класса, у которого есть деструктор, в середине блока создает новый блок, в котором в начале блока будет объявление объекта, а в конце — вызов его деструктора. Ниже код, который будет отвечать за это:

- Hidden text

```
/// Проверка на то, что для инструкции нужно создать новый бло
/// \param[in] oldStmt — текущая инструкция
/// \param[in] scope — область видимости
bool needPromoteBodyToBlock(StmtAST* oldStmt, Scope* scope) {
  // Только объявления могут создавать новые блоки
  if (!isa<DeclStmtAST>(oldStmt)) {
    return false;
  }

  // Производим семантический анализ для инструкции, т. к. нам
  // необходимо, что бы тип переменной был уже известен
  DeclStmtAST* declStmt = (DeclStmtAST*)oldStmt->semantic(scop
  SymbolList::iterator it = declStmt->Decls.begin();
  assert(isa<VarDeclAST>(*it));
  VarDeclAST* var = (VarDeclAST*)*it;

  // Только объекты классов могут создавать новые блоки
  if (isa<ClassTypeAST>(var->ThisType)) {
    ClassDeclAST* classDecl =
      (ClassDeclAST*)var->ThisType->getSymbol();

    // Только объекты классов с деструкторами могут создавать
    // новые блоки
    if (classDecl->Dtor) {
      return true;
    }
  }

  // Создавать новый блок не нужно
  return false;
}

StmtAST* BlockStmtAST::doSemantic(Scope* scope) {
  // Для блока мы должны создать новую область видимости
  ThisBlock = new ScopeSymbol(Loc, SymbolAST::SI_Block, nullpt
  Scope* s = scope->push((ScopeSymbol*)ThisBlock);
```

```
// Создать LandingPadAST для данного блока
// Замечание: Позже мы должны установить NeedCleanup, если
// это будет необходимо
LandingPad = new LandingPadAST(s->LandingPad, false);
LandingPad->OwnerBlock = this;
LandingPad->NeedCleanup = s->LandingPad->NeedCleanup;
s->LandingPad = LandingPad;

bool atStart = true; // Необходимо для создания новых блоков
ExprList args;

// Проверяем все ветви дерева принадлежащие данному блоку
for (StmtList::iterator it = Body.begin(), end = Body.end();
     it != end; ++it) {
  // Если в предыдущей инструкции был "break", "continue" ил
  // "return", то диагностируем об ошибке (предотвращаем
  // появление кода, который не может быть достижимым
  if (HasJump) {
    scope->report(Loc, diag::ERR_SemaDeadCode);
    return nullptr;
  }

  // Проверяем, что должен быть создан новый блок
  if (needPromoteBodyToBlock(*it, s)) {
    // В данном блоке есть деструкторы, которые должны быть
    // вызваны
    LandingPad->NeedCleanup = true;

    // Проверяем, что это середина блока
    if (!atStart) {
      // Создаем новый блок и переносим в него все инструкци
      // текущей и до конца блока (удаляя их из текущего бло
      StmtList body(it, end);
      BlockStmtAST* newBlockStmt = new BlockStmtAST(Loc, boc
      Body.erase(it, end);
      Body.push_back(newBlockStmt);
```

```
      // Помечаем блок, как автоматически созданный и произв
      // семантический анализ для него, а так же устанавлива
      // флаги HasJump и HasReturn
      newBlockStmt->IsPromoted = true;
      newBlockStmt->semantic(s);
      HasJump = newBlockStmt->HasJump;
      HasReturn = newBlockStmt->HasReturn;
      break;
    }

    DeclStmtAST* decls = (DeclStmtAST*)*it;

    for (SymbolList::iterator it2 = decls->Decls.begin(),
      end2 = decls->Decls.end(); it2 != end2; ++it2) {
      VarDeclAST* var = (VarDeclAST*)*it2;
      // Создаем вызов деструктора для данного объекта
      ExprAST* cleanupExpr = new CallExprAST(
        Loc,
        new MemberAccessExprAST(
          Loc,
          new IdExprAST(Loc, var->Id),
          Name::Dtor),
        args);
      // Произвести семантический анализ для вызова деструкт
      cleanupExpr = cleanupExpr->semantic(s);
      // Добавляем вызов деструктора к списку блока очистки
      CleanupList.push_back(cleanupExpr);
    }
  } else if (!isa<DeclStmtAST>(*it)) {
    // Это не объявление, устанавливаем atStart в false
    atStart = false;
  }

  // Проверяем семантику вложенной инструкции
  *it = (*it)->semantic(s);

  // Проверяем, что это "break", "continue" или "return"
```

```
    if ((*it)->isJump()) {
      HasJump = true;

      // Проверяем, что это "return"
      if ((*it)->hasReturn()) {
        HasReturn = true;
      }
    } else {
      // Это обычная инструкция, но все равно проверяем, налич
      // "break", "continue" или "return" в дочерних ветках
      // инструкции
      HasJump = (*it)->hasJump();
      HasReturn = (*it)->hasReturn();
    }
  }

  // Если нам нужно произвести очистку ресурсов
  if (LandingPad->NeedCleanup) {
    // Получаем родительскую функцию и устанавливаем флаг
    // NeedCleanup
    FuncDeclAST* fncDecl = (s->EnclosedFunc);
    fncDecl->LandingPad->NeedCleanup = true;

    if (!LandingPad->Prev->OwnerBlock) {
      // Если предыдущий LandingPadAST не является блоком,
      // то устанавливаем флаг NeedCleanup в true
      LandingPad->Prev->NeedCleanup = true;
    } else {
      // Предыдущий LandingPadAST является блоком, то нужно
      // установить флаг NeedCleanup только если есть наличие
      // "break", "continue" или "return" в дочерних ветках
      // инструкции
      if (LandingPad->Returns || LandingPad->Continues ||
        LandingPad->Breaks) {
        LandingPad->Prev->NeedCleanup = true;
      }
    }
```

```
  }

  // Удаляем область видимости
  s->pop();

  return this;
}
```

Так же нужно внести небольшие правки в циклах "while" и "if", которые связаны с вызовом деструкторов:

▾ Hidden text

```
StmtAST* WhileStmtAST::doSemantic(Scope* scope) {
  ...
  // Создаем новую LandingPadAST для всех вложенных инструкций
  LandingPad = new LandingPadAST(scope->LandingPad, false);
  LandingPad->NeedCleanup = scope->LandingPad->NeedCleanup;
  ...
}

StmtAST* IfStmtAST::doSemantic(Scope* scope) {
  ...
  // Создаем новый LandingPadAST
  LandingPad = new LandingPadAST(scope->LandingPad, false);
  LandingPad->NeedCleanup = scope->LandingPad->NeedCleanup;
  ...
}
```

Семантика объявлений

Для объявлений переменных или членов класса нам нужно произвести особую обработку для вызова конструкторов по умолчанию, если для них не заданы инициализаторы и есть конструктор:

▾ Hidden text

```
void VarDeclAST::doSemantic3(Scope* scope) {
  // При отсутствии инициализатора переменной, мы генерируем
  // инициализатор по умолчанию
  if (!Val) {
    // Отключаем стандартную обработку, если это член класса и
    // структуры
    if (needThis()) {
      return;
    }

    // Игнорируем, если это массив
    if (isa<ArrayTypeAST>(ThisType)) {
      return;
    }

    // У нас есть специальная обработка для агрегатов
    if (ThisType->isAggregate()) {
      if (isa<ClassTypeAST>(ThisType)) {
        ClassDeclAST* thisClass =
          (ClassDeclAST*)ThisType->getSymbol();
        // Для классов с конструктором, необходимо сгенерирова
        // вызов конструктора по умолчанию
        if (thisClass->Ctor) {
          ExprList args;
          Val = new CallExprAST(
            Loc,
            new MemberAccessExprAST(
```

```
            Loc,
            new IdExprAST(Loc, Id),
            Name::Ctor),
          args);
        // проверка семантики для вызова конструктора
        Val = Val->semantic(scope);
      }
    }

    return;
  // Специальная обработка для указателей и строк
  } else if (isa<PointerTypeAST>(ThisType) ||
    ThisType->isString())
    Val = new IntExprAST(Loc, 0);
  // Специальная обработка для "int"
  else if (ThisType->isInt())
    Val = new IntExprAST(Loc, 0);
  // Специальная обработка для "char"
  else if (ThisType->isChar()) {
    Val = new IntExprAST(Loc, 0);
    Val = new CastExprAST(
      Loc,
      Val,
      BuiltinTypeAST::get(TypeAST::TI_Char)
    );
  } else {
    // Инициализация для "float"
    Val = new FloatExprAST(Loc, 0.0);
  }

  // Производим семантический анализ для созданного
  // инициализирующего выражения
  Val = Val->semantic(scope);

  return;
}
```

```
  // Производим семантический анализ инициализирующего выражен
  Val = Val->semantic(scope);

  if (isa<ClassTypeAST>(ThisType)) {
    return;
  }
  ...
}
```

Проверка циклов в объявлениях:

▾ Hidden text

```
bool VarDeclAST::contain(TypeAST* type) {
  // Возвращаем true, если тип совпадает или type является
  // родительским классом для типа данного объекта
  if (ThisType->equal(type) || type->isBaseOf(ThisType)) {
    return true;
  }

  return false;
}

bool VarDeclAST::canBeNull() {
  // "super" или тип отличный от указателя не могут быть nullp
  if (Id == Name::Super || !isa<PointerTypeAST>(ThisType)) {
    return false;
  }
```

```
  return true;
}
```

Специальная обработка для очистки ресурсов для OverloadSetAST:

▾ Hidden text

```
ScopeSymbol::~ScopeSymbol() {
  for (SymbolMap::iterator it = Decls.begin(), end = Decls.enc
    it != end; ++it) {
    if (isa<OverloadSetAST>(it->second)) {
      delete it->second;
    }
  }
}
```

Запрет использования объектов класса в полях в структурах:

▾ Hidden text

```
void StructDeclAST::doSemantic3(Scope* scope) {
  // Создаем новую область видимости
  Scope* s = scope->push(this);
  int curOffset = 0;

  // Проверка всех дочерних объявлений
```

```
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    VarDeclAST* var = (VarDeclAST*)(*it);

    // Производим семантическую проверку для члена структуры
    // и получаем индекс для данного члена
    var->semantic(s);
    var->OffsetOf = curOffset++;

    // Запрещаем использование объектов класса в качестве поле
    // структур
    if (isa<ClassTypeAST>(var->getType())) {
      scope->report(Loc, diag::ERR_SemaClassAsStructMember);
      return;
    }
  }

  // Проверка на наличие циклических зависимостей
  if (contain(ThisType)) {
    scope->report(Loc, diag::ERR_SemaCricularReference, Id->Ic
    return;
  }

  // Удаляем область видимости для данного объявления
  s->pop();
}
```

Поддержка динамической диспетчеризации вызовов методов класса:

- Hidden text

```
// Проверка на то, что метод уже существует в таблице виртуаль
// методов
bool VTableAST::isExist(SymbolAST* func) {
  assert(isa<FuncDeclAST>(func));
  FuncDeclAST* fncDecl = (FuncDeclAST*)func;
  SymbolMap::iterator pos = Decls.find(func->Id);

  // Если ничего не найдено, то возвращаем false
  if (pos == Decls.end()) {
    return false;
  }

  // Проверка на то, что найденный символ является функцией ил
  // набором перегруженных функций
  if (isa<FuncDeclAST>(pos->second)) {
    // Функция
    FuncDeclAST* existDecl = (FuncDeclAST*)pos->second;

    // Если типы функций совпадают, то возвращаем true
    if (existDecl->ThisType->equal(fncDecl->ThisType)) {
      return true;
    }
    // Такой функции еще нет таблице
    return false;
  } else {
    // Это набор перегруженных функций
    assert(isa<OverloadSetAST>(pos->second));
    OverloadSetAST* existDecls = (OverloadSetAST*)pos->second;

    // Пытаемся найти функцию в наборе перегруженных функций
    return existDecls->OverloadsUsed.find(
      fncDecl->ThisType->MangleName
    ) != existDecls->OverloadsUsed.end();
  }
}
// Добавить или заменить функцию в таблице виртуальных методов
```

```
void VTableAST::addOrReplace(Scope* scope, SymbolAST* func) {
  assert(isa<FuncDeclAST>(func));
  FuncDeclAST* fncDecl = (FuncDeclAST*)func;
  SymbolMap::iterator pos = Decls.find(func->Id);

  if (pos == Decls.end()) {
    // Функция не найдена, добавляем ее в таблицу виртуальных
    // методов
    fncDecl->OffsetOf = CurOffset++;
    Decls[fncDecl->Id] = func;
    return;
  } else {
    // Проверяем, что это набор перегруженных функций
    if (isa<OverloadSetAST>(pos->second)) {
      OverloadSetAST* overloadSet = (OverloadSetAST*)pos->secc

      // Проверяем была ли данная функция добавлена в таблицу
      // нет
      if (overloadSet->OverloadsUsed.find(
          fncDecl->ThisType->MangleName
        ) != overloadSet->OverloadsUsed.end()) {
        // Ранее была добавлена. Находим старый слот и заменяе
        // новой
        for (SymbolList::iterator it = overloadSet->Vars.begir
          end = overloadSet->Vars.end(); it != end; ++it) {
          FuncDeclAST* oldFnc = (FuncDeclAST*)*it;
          if (oldFnc->ThisType->equal(fncDecl->ThisType)) {
            // Замена старой функции в слоте
            fncDecl->OffsetOf = oldFnc->OffsetOf;
            (*it) = fncDecl;

            // Проверяем, что типы возвращаемого значения совп
            if (!fncDecl->ReturnType->equal(oldFnc->ReturnType
              scope->report(
                fncDecl->Loc,
                diag::ERR_SemaVirtualFunctionReturnTypeOverloa
                fncDecl->Id->Id,
```

```
            Parent->Id->Id
          );
        }
        return;
      }
    }
  } else {
    // Это новая функция. Добавляем ее
    fncDecl->OffsetOf = CurOffset++;
    overloadSet->push(scope, func);
    return;
  }
} else {
  // Это обычная функция
  assert(isa<FuncDeclAST>(pos->second));
  FuncDeclAST* oldFnc = (FuncDeclAST*)pos->second;

  // Проверка на то, что прототипы функций совпадают
  if (oldFnc->ThisType->equal(fncDecl->ThisType)) {
    // Замена старой функции в слоте
    fncDecl->OffsetOf = oldFnc->OffsetOf;
    Decls[fncDecl->Id] = func;

    // Проверяем, что типы возвращаемого значения совпада
    if (!fncDecl->ReturnType->equal(oldFnc->ReturnType)) {
      scope->report(
        fncDecl->Loc,
        diag::ERR_SemaVirtualFunctionReturnTypeOverload,
        fncDecl->Id->Id,
        Parent->Id->Id
      );
    }
    return;
  } else {
    // Функции еще нет в таблице, создаем набор перегружен
    // функций с именем данной функции и добавляем ее в сл
    OverloadSetAST* newSet = new OverloadSetAST(fncDecl->I
```

```
        fncDecl->OffsetOf = CurOffset++;
        newSet->push(scope, oldFnc);
        newSet->push(scope, fncDecl);
        Decls[fncDecl->Id] = newSet;
        return;
      }
    }
  }
}
// Клонирование таблицы виртуальных методов
VTableAST* VTableAST::clone(SymbolAST* parent) {
  return new VTableAST(*this, parent);
}
```

Семантический анализ для классов:

- Hidden text

```
/// Генерация списка инициализирующих выражений по умолчанию д
/// полей класса
/// \param[in] classDecl — класс, для которого нужно произвест
///   генерацию
ExprList generateDefaultInitializer(ClassDeclAST* classDecl) {
  ExprList result;

  // Проверяем каждое объявление на наличие инициализирующих
  // выражений
  for (SymbolList::iterator it = classDecl->Vars.begin(),
    end = classDecl->Vars.end(); it != end; ++it) {
    // Проверяем только объявление переменных членов класса
    if (isa<VarDeclAST>(*it)) {
```

```
      VarDeclAST* var = (VarDeclAST*)*it;

      // Если есть значение для инициализации, то генерируем
      // инструкцию присвоения
      if (var->Val) {
        ExprAST* expr = new BinaryExprAST(
          classDecl->Loc,
          tok::Assign,
          new IdExprAST(classDecl->Loc, var->Id),
          var->Val->clone()
        );
        result.push_back(expr);
      }
      // Специальная проверка для объектов класса
      else if (isa<ClassTypeAST>(var->ThisType)) {
        ClassDeclAST* innerDecl =
          (ClassDeclAST*)var->ThisType->getSymbol();
        // Если у класса есть конструктор, то нужно сгенериров
        // его вызов
        if (innerDecl->Ctor) {
          // var.__ctor()
          ExprAST* expr = new MemberAccessExprAST(classDecl->L
            new IdExprAST(classDecl->Loc, (*it)->Id), Name::Ct
          ExprList args;
          expr = new CallExprAST(classDecl->Loc, expr, args);
          result.push_back(expr);
        }
      }
    }
  }

  return result;
}

/// Генерация конструктора по умолчанию для класса (если это
/// необходимо)
/// \param[in] classDecl — класс для которого нужно сгенериров
```

```
///  конструктор
FuncDeclAST* generateDefaultConstructor(ClassDeclAST* classDec
  // Получить список выражений для инициализации переменных чл
  // класса
  ExprList valsInit = generateDefaultInitializer(classDecl);

  ParameterList params;
  StmtList body;
  StmtAST* ctorCall = nullptr;

  // Проверяем родительский класс на наличие конструкторов
  if (classDecl->BaseClass &&
    isa<ClassTypeAST>(classDecl->BaseClass)) {
    ClassDeclAST* baseClass =
      (ClassDeclAST*)classDecl->BaseClass->getSymbol();
    // Если есть конструктор, то нужно произвести его вызов
    if (baseClass->Ctor) {
      // super.__ctor()
      ExprAST* expr = new MemberAccessExprAST(classDecl->Loc,
        new IdExprAST(classDecl->Loc, Name::Super), Name::Ctor
      ExprList args;
      expr = new CallExprAST(classDecl->Loc, expr, args);
      ctorCall = new ExprStmtAST(classDecl->Loc, expr);
    }
  }

  // Если есть хотя бы одна переменная с инициализирующим выра
  // нужно создать инструкции для
  if (!valsInit.empty()) {
    for (ExprList::iterator it = valsInit.begin(),
      end = valsInit.end(); it != end; ++it) {
      body.push_back(new ExprStmtAST(classDecl->Loc, *it));
    }
  }

  // Если нет ни таблицы виртуальных методов, ни конструктора
  // родительского класса, ни переменных для инициализации, то
```

```
  // возвращаем nullptr
  if (!classDecl->VTbl && body.empty() && !ctorCall) {
    return nullptr;
  }

  StmtList ctorBody;

  // Добавляем вызов конструктора базового класса
  if (ctorCall) {
    ctorBody.push_back(ctorCall);
  }

  // Добавляем инициализацию членов класса
  ctorBody.push_back(new BlockStmtAST(classDecl->Loc, body));

  // Создаем конструктор по умолчанию для класса
  FuncTypeAST* ctorType = new FuncTypeAST(nullptr, params);
  FuncDeclAST* ctorDecl = new FuncDeclAST(
    classDecl->Loc,
    ctorType,
    Name::Ctor,
    new BlockStmtAST(classDecl->Loc, ctorBody),
    true,
    tok::New
  );
  return ctorDecl;
}

/// Генерация действий по умолчанию для вызова деструкторов
/// переменных членов класса
/// \param[in] classDecl — класс для которого нужно произвести
///  генерацию
StmtList generateCleanupList(ClassDeclAST* classDecl) {
  StmtList result;

  // Проверяем каждое объявление и генерирурем вызовы деструкт
  // для тех членов класса, у которых они есть (деструкторы дс
```

```
  // вызваться в обратном порядке)
  for (SymbolList::reverse_iterator it = classDecl->Vars.rbegi
    end = classDecl->Vars.rend(); it != end; ++it) {
    // Пропускаем все объявления отличные от переменных
    if (!isa<VarDeclAST>(*it)) {
      continue;
    }

    VarDeclAST* var = (VarDeclAST*)*it;

    // Пропускаем все переменные не являющимеся объектами клас
    if (!isa<ClassTypeAST>(var->ThisType)) {
      continue;
    }

    ClassDeclAST* innerDecl =
      (ClassDeclAST*)var->ThisType->getSymbol();
    // Если класс имеет деструктор, но необходимо сгенерироват
    // его вызов
    if (innerDecl->Dtor) {
      // var.__dtor()
      ExprAST* expr = new MemberAccessExprAST(classDecl->Loc,
        new IdExprAST(classDecl->Loc, (*it)->Id), Name::Dtor);
      ExprList args;
      expr = new CallExprAST(classDecl->Loc, expr, args);
      result.push_back(new ExprStmtAST(classDecl->Loc, expr));
    }
  }

  return result;
}
```

Для корректной работы программы нам необходимо:

1. Для всех переменных членов класса у которых есть конструкторы, мы должны сгенерировать код для вызова их конструкторов, что бы они все были корректно инициализированы. Поэтому мы должны добавить их инициализацию для каждого конструктора, который объявлен в классе;
2. Для всех переменных членов класса у которых есть деструкторы, мы должны произвести очистку ресурсов путем вызова нужных деструкторов. Поэтому мы должны добавить очистку ресурсов в деструктор класса;
3. Если п.1 или п.2 соблюдены, но у самого класса нет ни одного явно объявленного конструктора или деструктора (или если у класса есть виртуальные методы), то для данного класса должны быть автоматически сгенерированы конструктор/деструктор.

Все это можно сделать либо на этапе генерации кода, либо на этапе семантического анализа (путем добавления к исходному дереву, полученному после синтаксического анализа, новых веток, которые будут производить все необходимые действия). Мы остановимся на варианте модификации исходного дерева, т. к. это позволит использовать уже готовую инфраструктуру генерации кода и избежать усложнения генерации кода.

Ниже рассмотрим семантический анализ для классов с учетом всех этих дополнений:

Hidden text

```
TypeAST* ClassDeclAST::getType() {
  return ThisType;
}

void ClassDeclAST::doSemantic(Scope* scope) {
  // Запрещаем переопределение
```

```
  if (scope->find(Id)) {
    scope->report(Loc, diag::ERR_SemaIdentifierRedefinition);
    return;
  }

  // Создаем новую область видимости
  Scope* s = scope->push(this);

  // Производим семантический анализ для типа текущего класса
  ThisType = ThisType->semantic(s);
  // Добавляем объявление в родительскую область видимости
  ((ScopeSymbol*)scope->CurScope)->Decls[Id] = this;
  Parent = scope->CurScope;

  // Производим семантический анализ для всех вложенных агрега
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    if ((*it)->isAggregate()) {
      (*it)->semantic(s);
    }
  }

  // Удаляем область видимости
  s->pop();
}

void ClassDeclAST::doSemantic2(Scope* scope) {
  // Создаем новую область видимости
  Scope* s = scope->push(this);

  // Если у класса есть родительский класс, то нужно это обраб
  if (BaseClass) {
    // Произвести семантический анализ типа родительского клас
    BaseClass = BaseClass->semantic(s);

    // Тип родительского класса должен быть агрегатом
    if (!BaseClass->isAggregate()) {
```

```
      scope->report(Loc, diag::ERR_SemaInvalidBaseClass, Id->I
      return;
    }

    // Проверяем на циклические зависимости
    if (BaseClass->equal(ThisType) ||
      ThisType->isBaseOf(BaseClass)) {
      scope->report(Loc, diag::ERR_SemaCricularReference, Id->
      return;
    }
  }

  // Удаляем область видимости
  s->pop();
}

void ClassDeclAST::doSemantic3(Scope* scope) {
  // Создаем новую область видимости
  Scope* s = scope->push(this);

  // Проверяем все методы класс на специальные методы
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    // Обрабатываем только методы класса
    if (!isa<FuncDeclAST>(*it)) {
      continue;
    }

    FuncDeclAST* fncDecl = (FuncDeclAST*)*it;

    // Если это конструктор, то устанавливаем флаг, что класс
    // имеет конструктор
    if (fncDecl->isCtor()) {
      Ctor = true;
      continue;
    }
```

```
  // Если это деструктор, то устанавливаем флаг, что класс
  // имеет деструктор
  if (fncDecl->isDtor()) {
    Dtor = true;
    continue;
  }
}

// Производим семантический анализ для каждой объявленной
// переменной члена класса
for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
  it != end; ++it) {
  // Обрабатываем только переменные члены
  if (!isa<VarDeclAST>(*it)) {
    continue;
  }

  // Производим семантический анализ объявления
  (*it)->semantic(s);

  // Игнорируем не объекты классов
  if (!isa<ClassTypeAST>((*it)->getType())) {
    continue;
  }

  ClassDeclAST* classDecl =
    (ClassDeclAST*)((*it)->getType()->getSymbol());

  // Для того, что бы корректно сгенерировать конструкторы и
  // деструкторы необходимо убедится, что все переменные чле
  // класса являющимися объектами классов должны пройти 4 ст
  // семантического анализа, если это не так, то запускаем
  // семантический анализ всех нужных стадий
  if (classDecl->SemaState < 4) {
    // Воссоздаем область видимости для данного члена класса
    Scope* memberScope = Scope::recreateScope(scope, classDe
```

```
    // Запускаем 2, 3 и 4 стадии семантического анализа
    classDecl->semantic2(memberScope);
    classDecl->semantic3(memberScope);
    classDecl->semantic4(memberScope);

    // Очищаем все лишние области видимости, оставляем тольк
    // область видимости модуля
    Scope::clearAllButModule(memberScope);
  }
}

// Проверяем на наличие циклических зависимостей
if (contain(ThisType)) {
  scope->report(Loc, diag::ERR_SemaCricularReference, Id->Ic
  return;
}

// Убеждаемся, что родительский класс знает все о своих
// конструкторах
if (BaseClass && isa<ClassTypeAST>(BaseClass)) {
  ClassDeclAST* baseDecl = (ClassDeclAST*)BaseClass->getSymb

  // Проверяем, что 4 проход семантического анализа был запу
  // ранее
  if (baseDecl->SemaState < 4) {
    // Нет. Воссоздаем область видимости для данного члена к
    Scope* parenScope = Scope::recreateScope(scope, baseDecl

    // Запускаем 2, 3 и 4 стадии семантического анализа
    baseDecl->semantic2(parenScope);
    baseDecl->semantic3(parenScope);
    baseDecl->semantic4(parenScope);

    // Очищаем все лишние области видимости, оставляем тольк
    // область видимости модуля
    Scope::clearAllButModule(parenScope);
  }
```

```
}

ExprList valsInit;

// Если есть конструктор, то генерируем список инициализирую
// выражений для переменных членов класса
if (Ctor) {
  valsInit = generateDefaultInitializer(this);
}

// Теперь мы можем произвести семантический анализ для метод
// класса
for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
  it != end; ++it) {
  // Пропускаем все не методы класса
  if (!isa<FuncDeclAST>(*it)) {
    continue;
  }

  FuncDeclAST* fncDecl = (FuncDeclAST*)*it;

  // Специальная обработка для конструкторов
  if (fncDecl->isCtor()) {
    StmtList initBody;

    // Генерируем блок инициализации переменных членов класс
    // на основе ранее сгенерированого списка
    for (ExprList::iterator it = valsInit.begin(),
      end = valsInit.end(); it != end; ++it) {
      initBody.push_back(new ExprStmtAST(Loc, (*it)->clone()
    }

    // Получаем тело конструктора
    BlockStmtAST* fncBody = (BlockStmtAST*)fncDecl->Body;

    // Проверяем наличие вызова конструктора родительского к
    if (isa<ExprStmtAST>(fncBody->Body[0])) {
```

```
      // Мы должны добавить блок инициализации сразу после в
      // конструктора родительского класса
      fncBody->Body.insert(
        fncBody->Body.begin() + 1,
        new BlockStmtAST(fncDecl->Loc, initBody)
      );
    } else {
      // Мы должны добавить блок инициализации в самое начал
      // тела конструктора
      fncBody->Body.insert(
        fncBody->Body.begin(),
        new BlockStmtAST(fncDecl->Loc, initBody)
      );
    }
  // Специальная обработка для деструктора
  } else if (fncDecl->isDtor()) {
    // Пробуем сгенерировать список очистки для переменных
    // членов класса
    StmtList defaultCleanup = generateCleanupList(this);

    if (!defaultCleanup.empty()) {
      // Если список не пуст, то нужно добавить очистку в
      // конец тела деструктора
      BlockStmtAST* fncBody = (BlockStmtAST*)fncDecl->Body;
      fncBody->Body.insert(
        fncBody->Body.end(),
        defaultCleanup.begin(),
        defaultCleanup.end()
      );
    }
  }

  // Проверка наличия родительского класса
  if (BaseClass && isa<ClassTypeAST>(BaseClass)) {
    Name* memberId = (*it)->Id;

    // Пытаемся найти данное определение в родительском клас
```

```
      SymbolAST* foundInBase =
        ((ClassTypeAST*)BaseClass)->ThisDecl->find(memberId);

      // Если символ найден и это не конструктор или деструктс
      // то нужно произвести специальную обработку
      if (foundInBase && memberId != Name::Ctor &&
        memberId != Name::Dtor) {
        if (isa<FuncDeclAST>(foundInBase)) {
          // Это метод класса, копируем его в список объявлени
          // данного класса для поддержки перегрузки функций
          Decls[memberId] = foundInBase;
        } else if (isa<OverloadSetAST>(foundInBase)) {
          // Это набор перегруженных функций. Клонируем его дл
          // поддержки перегрузки функций из родительского кла
          Decls[memberId] = ((OverloadSetAST*)foundInBase)->cl
        }
      }
    }

    // Производим семантический анализ метода класса
    (*it)->semantic(s);
  }

  // Если ранее были созданы инициализирующие выражения, то
  // удаляем их
  if (!valsInit.empty()) {
    for (ExprList::iterator it = valsInit.begin(),
      end = valsInit.end(); it != end; ++it) {
      delete *it;
    }
  }

  // Удаляем область видимости
  s->pop();
}

void ClassDeclAST::doSemantic4(Scope* scope) {
```

```
// Создаем новую область видимости
Scope* s = scope->push(this);

// Если VTable была создана в данном классе, не в родительск
// классе
bool vtableOwner = true;
VTableAST* vtbl = nullptr;
int curOffset = 0; // Текущий слот для переменных
// true — если нужно сгенерировать деструктор
bool generateDtor = false;
bool baseHasVtbl = false;
StmtList defaultDtorBody;

// Если класс не имеет явных деструкторов, то пробуем создат
// список для очистки переменных членов класса
if (!Dtor) {
  defaultDtorBody = generateCleanupList(this);
}

// Пытаемся найти VTable в родительском классе
if (BaseClass) {
  // Проверяем, что это класс
  if (isa<ClassTypeAST>(BaseClass)) {
    ClassDeclAST* baseDecl =
      (ClassDeclAST*)BaseClass->getSymbol();

    if (baseDecl->VTbl) {
      // Базовый класс имеет свою таблиц виртуальных методов
      // должны взять Vtable за основу и установить все нужн
      // флаги
      vtableOwner = false;
      vtbl = baseDecl->VTbl;
      baseHasVtbl = true;
    }

    // Если у класса нет деструктора, но родительский класс
    // деструктор то мы должны сгенерировать деструктор
```

```
    if (!Dtor && baseDecl->Dtor) {
      generateDtor = true;
    }

    // Увеличиваем слот для переменных членов, т. к. 1 слот
    // экземпляр родительского класса
    ++curOffset;
  } else {
    // Это структура. Увеличиваем слот для переменных членов
    // т. к. 1 слот – экземпляр родительского класса
    ++curOffset;
  }
}

// Если у класса нет деструктора и есть переменные для очист
// то мы должны сгенерировать деструктор
if (!Dtor && !defaultDtorBody.empty()) {
  generateDtor = true;
}

// Если VTable еще не было установленно (т. е. у родительскс
// класса нет виртуальных методов), то создаем свою таблицу
// виртуальных методов
if (!vtbl) {
  vtbl = new VTableAST(this);
}

// Проверяем все объявление на наличие виртуальных методов
for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
  it != end; ++it) {
  // Пропускаем не методы
  if (!isa<FuncDeclAST>(*it)) {
    continue;
  }

  FuncDeclAST* fncDecl = (FuncDeclAST*)*it;
```

```
// Пропускаем конструкторы
if (fncDecl->isCtor()) {
  continue;
}

// Проверяем, существует ли метод в таблице виртуальных
// методов или нет
if (vtbl->isExist(fncDecl)) {
  // Присутствует. Это должно быть "impl"
  if (!fncDecl->isOverride()) {
    scope->report(Loc, diag::ERR_SemaVirtualFunctionExists
    return;
  }
} else {
  // Отсутствует. Это должно быть "virt"
  if (fncDecl->isOverride()) {
    scope->report(Loc,
                  diag::ERR_SemaOverrideFunctionDoesntExis
    return;
  }
}

// Если это виртуальный или переопределенный метод, то нам
// нужно его добавить
if (fncDecl->isVirtual() || fncDecl->isOverride()) {
  // Если у нас нет своей VTable, то клонируем таблицу
  // виртуальных методов родительского класса и берем ее
  // за основу
  if (vtableOwner == false) {
    vtbl = vtbl->clone(this);
    vtableOwner = true;
  }

  // Добавляем или перезаписываем функцию в таблице
  // виртуальных методов
  vtbl->addOrReplace(scope, fncDecl);
}
```

```
}

// Если VTable имеет хотя бы 1 виртуальный метод, то мы долж
// его добавить
if (vtbl->CurOffset > 0) {
  VTbl = vtbl;
  OwnVTable = vtableOwner;

  // Если родительский класс не имеет своей таблицы виртуаль
  // методов, то мы должны добавить слот для нее
  if (!baseHasVtbl) {
    ++curOffset;
  }
}

// Теперь мы должны рассчитать слот для каждой переменной
// члена класса
for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
  it != end; ++it) {
  if (isa<VarDeclAST>(*it)) {
    ((VarDeclAST*)(*it))->OffsetOf = curOffset++;
  }
}

// Если нет явного конструктора, то пробуем создать конструк
// по умолчанию
if (!Ctor) {
  // Пробуем создать конструктор по умолчанию
  SymbolAST* ctorDecl = generateDefaultConstructor(this);

  // Конструктор был сгенерирован. Производим его семантичес
  // анализ и добавляем его к списку объявлений данного клас
  if (ctorDecl) {
    ctorDecl->semantic(s);
    Vars.push_back(ctorDecl);
    Ctor = true;
  }
```

```
}

// Проверяем нужен ли нам деструктор или нет
if (generateDtor) {
  // Генерируем деструктор с пустым телом. Все необходимое
  // будет сгенерировано во время семантического анализа
  ParameterList params;
  FuncTypeAST* dtorType = new FuncTypeAST(
    BuiltinTypeAST::get(TypeAST::TI_Void),
    params
  );
  SymbolAST* dtorDecl = new FuncDeclAST(
    Loc,
    dtorType,
    Name::Dtor,
    new BlockStmtAST(Loc, defaultDtorBody),
    true,
    tok::Def
  );

  // Производим семантический анализ деструктора
  dtorDecl->semantic(s);

  // Если деструктор базового класса был виртуальным, то нам
  // нужно его заменить
  if (VTbl && VTbl->isExist(dtorDecl)) {
    ((FuncDeclAST*)dtorDecl)->Tok = tok::Override;

    // Если у нас нет своей VTable, то клонируем таблицу
    // виртуальных методов родительского класса и берем ее
    // за основу
    if (!vtableOwner) {
      VTbl = VTbl->clone(this);
    }

    // Заменяем деструтор в таблице виртуальных методов
    VTbl->addOrReplace(scope, dtorDecl);
```

```
    }

    Vars.push_back(dtorDecl);
    Dtor = true;
  }

  // Удаляем область видимости
  s->pop();
}

void ClassDeclAST::doSemantic5(Scope* scope) {
  // Создаем новую область видимости
  Scope* s = scope->push(this);

  // Запускаем 2-ю стадию семантического анализа для всех доче
  // объявлений
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    (*it)->semantic2(s);
  }

  // Запускаем 3-ю и 4-ю стадии семантического анализа для все
  // дочерних объявлений
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    (*it)->semantic3(s);
    (*it)->semantic4(s);
  }

  // Запускаем финальную 5-ю стадию семантического анализа для
  // всех дочерних объявлений
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    (*it)->semantic5(s);
  }

  // Удаляем область видимости
```

```
  s->pop();
}

SymbolAST* ClassDeclAST::find(Name* id, int flags) {
  // Ищем объявление в данном классе
  SymbolAST* res = ScopeSymbol::find(id, flags);

  // Если нашли, то возвращаем его
  if (res) {
    return res;
  }

  // Если нужно, то пытаемся найти в родительском классе
  if ((flags & 1) == 0) {
    if (BaseClass && BaseClass->isAggregate()) {
      return BaseClass->getSymbol()->find(id, flags);
    }
  }

  return res;
}

bool ClassDeclAST::contain(TypeAST* type) {
  // Пробуем найти циклические зависимости в членах класса
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    if ((*it)->contain(type)) {
      return true;
    }
  }

  // Если есть родительский класс, то необходимо проверить нал
  // циклов в нем
  if (BaseClass && BaseClass->isAggregate()) {
    return BaseClass->getSymbol()->contain(type);
  }
```

```
  return false;
}
```

Для параметров функции необходимо обновить функцию член класса canBeNull:

Hidden text

```
bool ParameterSymbolAST::canBeNull() {
  // Если это "this" или не указатель, то параметр не может бы
  // "null"
  if (Param->Id == Name::This || !isa<PointerTypeAST>(Param->P
    return false;
  }

  // Это указатель, может быть "null"
  return true;
}
```

Семантика для функций также должна быть обновлена, т. к. нам необходимо добавить поддержку конструкторов, деструкторов, а так же поддержку перегрузки функций, так же нам нужно учитывать, что в методах класса могут быть специальные параметры и переменные, которые отвечают за "this" и "super":

Hidden text

```
bool FuncDeclAST::needThis() {
  return NeedThis;
}

void FuncDeclAST::doSemantic(Scope* scope) {
  // Методы класса имеют специальную обработку
  if (needThis()) {
    assert(isa<ClassDeclAST>(scope->CurScope));
    ClassDeclAST* classDecl = ((ClassDeclAST*)scope->CurScope)
    // Создаем параметр для "this", как неизменяемый указатель
    // на объект класса
    ParameterAST* thisParam = new ParameterAST(
      new PointerTypeAST(classDecl->ThisType, true),
      Name::This);
    // Получаем тип текущей функции
    FuncTypeAST* thisType = (FuncTypeAST*)ThisType;
    // Указываем, что у функции есть "this"
    thisType->HasThis = true;
    // Добавляем "this" в качестве первого параметра функции
    thisType->Params.insert(thisType->Params.begin(), thisPara
    AggregateSym = classDecl;

    // Добавляем особую обработку только, если у функции уже е
    // тело
    if (Body) {
      BlockStmtAST* body = (BlockStmtAST*)Body;

      // Проверяем, что это конструктор
      if (isCtor()) {
        // Проверяем, что у нас есть вызов конструктор
        // родительского класса, но у нас либо нет родительскс
        // класса, либо он не имеет конструктора
        if (!body->Body.empty() &&
          isa<ExprStmtAST>(*body->Body.begin())) {
          // Сообщаем об ошибке, если нет родительского класса
          // он есть, но это структура
```

```
      if (!classDecl->BaseClass ||
        !isa<ClassTypeAST>(classDecl->BaseClass)) {
        scope->report(Loc,
          diag::ERR_SemaBaseClassNoConstructorToCall);
        return;
      }

      ClassDeclAST* baseDecl =
        (ClassDeclAST*)classDecl->BaseClass->getSymbol();

      // Сообщаем об ошибке, если родительский класс не им
      // конструктора
      if (!baseDecl->Ctor) {
        scope->report(Loc,
          diag::ERR_SemaBaseClassNoConstructorToCall);
        return;
      }
    } else {
      // У конструктора пока нет вызова конструктора
      // родительского класса, мы должны сгенерировать выз
      // конструктора по умолчанию, если необходимо
      if (classDecl->BaseClass &&
        isa<ClassTypeAST>(classDecl->BaseClass)) {
        // Родительский класс есть
        ClassDeclAST* baseDecl =
          (ClassDeclAST*)classDecl->BaseClass->getSymbol()

        // Проверяем, что у родительского класса есть
        // конструктор
        if (baseDecl->Ctor) {
          // Генерируем вызов конструктора по умолчанию
          // родительского класса
          ExprList args;
          ExprStmtAST* exprStmt = new ExprStmtAST(
            Loc,
            new CallExprAST(
              Loc,
```

```
                new MemberAccessExprAST(
                  Loc,
                  new IdExprAST(Loc, Name::Super),
                  Name::Ctor),
                args));
            // Добавляем вызов конструктора родительского кл
            // как 1-ю инструкцию в тело функции
            // Замечание: объявление переменной "super" буде
            // добавлен позже
            body->Body.insert(body->Body.begin(), exprStmt);
          }
        }
      }
    // Проверка деструктора
    } else if (isDtor()) {
      // Если у класса есть родительский класс и у него есть
      // деструктор, то мы должны добавить его вызов
      if (classDecl->BaseClass &&
        isa<ClassTypeAST>(classDecl->BaseClass)) {
        ClassDeclAST* baseDecl =
          (ClassDeclAST*)classDecl->BaseClass->getSymbol();

        // Проверка на то, что родительский класс имеет дест
        if (baseDecl->Dtor) {
          // Генерируем вызов деструктора базового класса
          ExprList args;
          ExprStmtAST* exprStmt = new ExprStmtAST(
            Loc,
            new CallExprAST(
              Loc,
              new MemberAccessExprAST(
                Loc,
                new IdExprAST(Loc, Name::Super),
                Name::Dtor),
              args));
          // Добавляем вызов деструктора в тело деструктора
          // качестве последней инструкции
```

```
            body->Body.push_back(exprStmt);
          }
        }
      }

      // Если у класса есть родительский класс, то нам нужно
      // объявить "super", как ссылка на экземпляр родительскс
      // класса
      if (classDecl->BaseClass) {
        // Создаем объявление переменной "super", с типом
        // неизменяемого указателя на родительский класс
        VarDeclAST* superVar = new VarDeclAST(
          Loc,
          new PointerTypeAST(classDecl->BaseClass, true),
          Name::Super,
          new IdExprAST(Loc, Name::This),
          false);
        SymbolList tmp;
        // Добавляем объявление как 1-ю инструкцию в теле
        // конструктора
        tmp.push_back(superVar);
        body->Body.insert(
          body->Body.begin(),
          new DeclStmtAST(Loc, tmp)
        );
      }
    }
  }

  // Производим семантический анализ для прототипа функции
  ThisType = ThisType->semantic(scope);
  Parent = scope->CurScope;
  // Настраиваем тип возвращаемого значения
  ReturnType = ((FuncTypeAST*)ThisType)->ReturnType;

  // Отдельная проверка для функции "main"
  if (Id->Length == 4 && memcmp(Id->Id, "main", 4) == 0) {
```

```
    FuncTypeAST* thisType = (FuncTypeAST*)ThisType;

    // Должна не иметь параметров
    if (thisType->Params.size()) {
      scope->report(Loc, diag::ERR_SemaMainParameters);
      return;
    }

    // Должна возвращать "float"
    if (ReturnType != BuiltinTypeAST::get(TypeAST::TI_Float))
      scope->report(Loc, diag::ERR_SemaMainReturnType);
      return;
    }
  }

  // Ищем объявление функции в текущей области видимости
  if (SymbolAST* fncOverload = scope->findMember(Id, 1)) {
    // Проверяем, что это объявление функции
    if (isa<FuncDeclAST>(fncOverload)) {
      // Создаем набор перегруженных функций
      OverloadSetAST* newOverloadSet = new OverloadSetAST(Id);

      // Устанавливаем родительскую область видимости и добавл
      // обе функции в него
      newOverloadSet->Parent = Parent;
      newOverloadSet->push(scope, fncOverload);
      newOverloadSet->push(scope, this);

      // Заменяем старое объявление в родительской области
      // видимости на только что созданный набор
      ((ScopeSymbol*)Parent)->Decls[Id] = newOverloadSet;
      return;
    }

    // Проверяем, что это набор перегруженных функций
    if (isa<OverloadSetAST>(fncOverload)) {
      // Да. Добавляем новую функции в набор
```

```
      ((OverloadSetAST*)fncOverload)->push(scope, this);
      return;
    }

    // Выдаем ошибку, т. к. имя было использовано для другого
    // объявления
    scope->report(Loc, diag::ERR_SemaFunctionRedefined, Id->Ic
    return;
  }

  // Добавляем функцию к списку объявлений
  ((ScopeSymbol*)Parent)->Decls[Id] = this;
}
```

Объявление для набора перегруженных функций:

▾ Hidden text

```
void OverloadSetAST::push(Scope *scope, SymbolAST* func) {
  assert(isa<FuncDeclAST>(func));
  FuncDeclAST* funcDecl = (FuncDeclAST*)func;

  // Если у функции еще нет уникального сгенерируемого имени,
  // генерируем его
  if (!funcDecl->ThisType->MangleName.empty()) {
    funcDecl->ThisType->calcMangle();
  }

  // Пытаемся добавить функцию в список используемых перегрузс
  std::pair<llvm::StringSet< >::iterator, bool> res =
    OverloadsUsed.insert(funcDecl->ThisType->MangleName);
```

```
  // Проверяем была ли она добавлена ранее или нет
  if (!res.second) {
    // Да. Просматриваем все функции в наборе перегруженных фу
    for (SymbolList::iterator it = Vars.begin(), end = Vars.er
      it != end; ++it) {
      FuncDeclAST* oldFn = (FuncDeclAST*)(*it);

      // Проверяем на совпадение протатипов
      if (oldFn->ThisType->equal(funcDecl->ThisType)) {
        // Проверяем, что это функция объявленная в дочернем к
        // которая заменяет аналогичную функцию в родительском
        // классе
        if (oldFn->Parent != func->Parent) {
          // Заменяем функцию из базового класса
          (*it) = func;
          return;
        }

        // Это дублирующая функция, выходим из цикла и сообщае
        // ошибке
        break;
      }
    }

    // Сообщаем об ошибке
    scope->report(
      funcDecl->Loc,
      diag::ERR_SemaDuplicateFunctions,
      funcDecl->Id->Id
    );
    return;
  }

  // Добавляем функцию в набор
  Vars.push_back(func);
}
```

```
OverloadSetAST* OverloadSetAST::clone() {
  // Создаем новый набор перегруженных функций
  OverloadSetAST* newSet = new OverloadSetAST(Id);

  // Переносим все значимые поля в копию
  newSet->Parent = Parent;
  newSet->Vars.assign(Vars.begin(), Vars.end());
  newSet->OverloadsUsed = OverloadsUsed;

  // Возвращаем копию
  return newSet;
}
```

Т.к. мы добавили поддержку перегрузки функций на основе их параметров, то мы можем заменить все системные функции и использовать для них перегрузку функций:

▾ Hidden text

```
void initRuntimeFuncs(ModuleDeclAST* modDecl) {
  addDynamicFunc(
    "fn print(_: char)",
    "lle_X_printChar",
    modDecl,
    (void*)lle_X_printChar
  );
  addDynamicFunc(
    "fn print(_: int)",
    "lle_X_printInt",
    modDecl,
```

```
    (void*)lle_X_printInt
  ;
  addDynamicFunc(
    "fn print(_: float)",
    "lle_X_printDouble",
    modDecl,
    (void*)lle_X_printDouble
  );
  addDynamicFunc(
    "fn print(_: string)",
    "lle_X_printString",
    modDecl,
    (void*)lle_X_printString
  );
  addDynamicFunc(
    "fn printLn(_: string)",
    "lle_X_printLine",
    modDecl,
    (void*)lle_X_printLine
  );
  addDynamicFunc(
    "fn new(_: int) : void*",
    "lle_X_new",
    modDecl,
    (void*)lle_X_new
  );
  addDynamicFunc(
    "fn delete(_: void*)",
    "lle_X_delete",
    modDecl,
    (void*)lle_X_delete
  );
}
```

Генерация кода

В генерации кода для типов у нас появился новый тип — ClassTypeAST:

Hidden text

```
Type* ClassTypeAST::getType() {
  // Исключаем повторную генерацию
  if (ThisType) {
    return ThisType;
  }

  llvm::SmallString< 128 > s;
  llvm::raw_svector_ostream output(s);

  // Для классов мы всегда добавляем "class." в начале имени
  output << "class.";
  calcAggregateName(output, ThisDecl);

  // Замечание: Мы должны создать новый тип и инициализировать
  // ThisType, т. к. поля класса могут ссылаться на этот тип
  ThisType = StructType::create(getGlobalContext(), output.str

  std::vector< Type* > vars;
  ClassDeclAST* classDecl = (ClassDeclAST*)ThisDecl;
  VTableAST* baseVtbl = nullptr;
  Type* baseType = nullptr;

  // Проверяем, есть ли у данного класса родительский класс ил
  if (classDecl->BaseClass) {
    // Проверяем, что родительский класс является классом, а н
    // структурой
    if (isa<ClassTypeAST>(classDecl->BaseClass)) {
      // Сохраняем таблицу виртуальных методов родительского к
      // если она есть
```

```
    ClassDeclAST* baseDecl =
      (ClassDeclAST*)classDecl->BaseClass->getSymbol();
    baseVtbl = baseDecl->VTbl;
  }

  // Сохраняем тип родительского класса
  baseType = classDecl->BaseClass->getType();
}

// Если у класса есть таблица виртуальных методов, а у
// родительского класса ее не было, то у нас для этого есть
// специальная обработка
if (classDecl->VTbl && !baseVtbl) {
  // Добавляем слот для таблицы виртуальных методов
  // Замечание: Мы хотим, что бы таблица виртуальных методов
  // всегда была первым слотом в структуре класса
  vars.push_back(
    PointerType::get(
      getGlobalContext(),
      getSLContext().TheTarget->getProgramAddressSpace()
    )
  );
}

// Если у класса есть родительский класс, то добавляем его к
// отдельный слот
// Пример:
//   class A { int a; }
//   class B : A { int b; }
// будет
//   class.A = { int32 }
//   class.B = { class.A, int32 }
if (baseType) {
  vars.push_back(baseType);
}

// Мы должны обработать все дочерние объявления
```

```
  for (SymbolList::iterator it = classDecl->Vars.begin(),
    end = classDecl->Vars.end(); it != end; ++it) {
    // Добавляем в результирующую структуру только переменные
    if (isa<VarDeclAST>(*it)) {
      vars.push_back(((VarDeclAST*)(*it))->ThisType->getType()
    }
  }

  // Если у класса 0 членов, то мы должны добавить одно поле
  // размером в 1 байт, иначе размер класса будет равен 0, что
  // нам не нужно
  if (vars.empty()) {
    vars.push_back(Type::getInt8Ty(getGlobalContext()));
  }

  // Добавляем все только что созданные члены к телу ранее
  // созданной структуры класса
  ((StructType*)ThisType)->setBody(vars);
  return ThisType;
}
```

Одна из частых операций, которая понадобится нам при генерации кода для поддержки классов и наследования, это операция приведения указателя дочернего класса в указатель родительского класса. Поэтому для этого мы будем использовать следующую вспомогательную функцию:

▾ Hidden text

```
/// \param[in] Context - контекст
/// \param[in] val — указатель для преобразования
```

```
/// \param[in] var — объявление класса к которому нужно
///   преобразовать
/// \param[in] agg — объявление класса из которого нужно
///   преобразовать
/// \param[in] canBeNull - true — если указатель может быть
///   nullptr
Value* convertToBase(
  SLContext& Context, Value* val, SymbolAST* var,
  SymbolAST* agg, bool canBeNull) {
  ClassDeclAST* thisDecl = (ClassDeclAST*)agg;
  SymbolAST* baseDecl = thisDecl->BaseClass->getSymbol();
  bool baseHasVTable = false;

  // Если родительский класс является классом, а не структурой
  // то нужно определить имеет ли он таблицу виртуальных метод
  // или нет
  if (isa<ClassDeclAST>(baseDecl)) {
    ClassDeclAST* baseClassDecl = (ClassDeclAST*)baseDecl;
    baseHasVTable = baseClassDecl->VTbl != nullptr;
  }

  // Если родительский класс не имеет собственной таблицы
  // виртуальных методов, то нам нужно произвести специальную
  // обработку для приведения типов
  if (thisDecl->VTbl && baseHasVTable == false) {
    // Если значение может быть nullptr, то нам нужно произвес
    // дополнительные проверки
    if (canBeNull) {
      // Сейчас val содержит указатель на дочерний класс. Нам
      // произвести проверку для безопасного преобразования,
      // если он равен nullptr
      Value* tmp = ConstantPointerNull::get(
        (PointerType*)val->getType()
      );
      // Сравниваем значение val с nullptr
      tmp = Context.TheBuilder->CreateICmpNE(val, tmp);
```

```
// Создаем блоки для проверки
BasicBlock* thenBlock = BasicBlock::Create(
  getGlobalContext(),
  "then",
  Context.TheFunction
);
BasicBlock* elseBlock = BasicBlock::Create(
  getGlobalContext(),
  "else"
);
BasicBlock* contBlock = BasicBlock::Create(
  getGlobalContext(),
  "ifcont"
);

// Производим условный переход в зависимости от результа
// сравнения с nullptr
tmp = Context.TheBuilder->CreateCondBr(tmp, thenBlock,
                                       elseBlock);
Context.TheBuilder->SetInsertPoint(thenBlock);

// 1. Корректируем указатель и пропускаем таблицу виртуа
// методов
val = Context.TheBuilder->CreateGEP(
  Type::getInt8Ty(getGlobalContext()),
  val,
  getConstInt(
    Context.TheTarget->getTypeAllocSize(
      PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace
      )
    )
  )
);

// 2. Получаем реальный адрес родительского класса
```

```
if (var != baseDecl) {
  // Тип результата не совпадаем с родительским классом
  // текущего класса, значит результирующий класс являет
  // родительским классом для класса результата, произво
  // преобразование еще раз
  val = convertToBase(Context, val, var, baseDecl, canBe
}

// Сохраняем реальное значение для ветки "then", т. к. с
// могло измениться
thenBlock = Context.TheBuilder->GetInsertBlock();
// Генерируем безусловный переход на "ifcont"
Context.TheBuilder->CreateBr(contBlock);

// Замечание: elseBlock не содержит никакого кода, тольк
// безусловный переход на "ifcont"
Context.TheFunction->getBasicBlockList().push_back(elseB
Context.TheBuilder->SetInsertPoint(elseBlock);

Context.TheBuilder->CreateBr(contBlock);

// В качестве текущего блока устанавливаем "ifcont"
Context.TheFunction->getBasicBlockList().push_back(contB
Context.TheBuilder->SetInsertPoint(contBlock);

// Создаем узел PHI с результатом операции
PHINode* PN = Context.TheBuilder->CreatePHI(val->getType

// Добавляем значения для PHI, либо результат операции
// в "then", либо nullptr
PN->addIncoming(val, thenBlock);
PN->addIncoming(
  ConstantPointerNull::get((PointerType*)val->getType())
  elseBlock
);

// Возвращаем созданный узел PHI
```

```
      return PN;
    } else {
      // Значение указателя не может быть nullptr, поэтому нам
      // нужны дополнительные проверки

      // 1. Корректируем указатель и пропускаем таблицу виртуа
      // методов
      val = Context.TheBuilder->CreateGEP(
        Type::getInt8Ty(getGlobalContext()),
        val,
        getConstInt(
          Context.TheTarget->getTypeAllocSize(
            PointerType::get(
              getGlobalContext(),
              getSLContext().TheTarget->getProgramAddressSpace
            )
          )
        )
      );

      // 2. Получаем реальный адрес родительского класса
      if (var != baseDecl) {
        // Тип результата не совпадаем с родительским классом
        // текущего класса, значит результирующий класс являет
        // родительским классом для класса результата, произво
        // преобразование еще раз
        return convertToBase(Context, val, var, baseDecl, canB
      }

      // Возвращаем результат преобразования
      return val;
    }
  } else {
    // У нас либо нет таблицы виртуальных методов, либо оба кл
    // имеют ее

    // Если класс класс результата и класс родителя совпадают,
```

```
    // возвращаем полученное значение "как есть"
    if (var == baseDecl) {
      return val;
    }

    // Тип результата не совпадаем с родительским классом теку
    // класса, значит результирующий класс является родительск
    // классом для класса результата, производим преобразовани
    // еще раз
    return convertToBase(Context, val, var, baseDecl, canBeNul
  }
}
```

Для оператора обращения к члену класса нам нужно добавить поддержку того, что само объявление принадлежит родительскому классу, а не самому объекту:

▾ Hidden text

```
Value* MemberAccessExprAST::getLValue(SLContext& Context) {
  // Для функций просто возвращаем ее тип
  if (isa<FuncTypeAST>(ExprType)) {
    return ThisSym->getValue(Context);
  }

  // Получаем lvalue для левой части (this)
  Value* val = Val->getLValue(Context);

  // Если член класса принадлежит родительскому классу, то нуж
  // произвести преобразование типов
  if (ThisSym->Parent != ThisAggr) {
```

```
    val = convertToBase(Context, val, ThisSym->Parent, ThisAgg
                        false);
  }

  // Генерирум индекс на основе индекса члена класса/структурь
  Value* index = getConstInt(((VarDeclAST*)ThisSym)->OffsetOf)

  std::vector< Value* > idx;

  idx.push_back(getConstInt(0));
  idx.push_back(index);

  // Создаем инструкцию GetElementPtr
  return Context.TheBuilder->CreateGEP(
    ThisSym->Parent->getType()->getType(),
    val,
    idx
  );
}

Value* CastExprAST::getRValue(SLContext& Context) {
  ...
  // Проверка на преобразования к указателю
  } else if (isa<PointerTypeAST>(ExprType)) {
    // Проверяем на приведение указателя
    if (isa<PointerTypeAST>(Val->ExprType)) {
      // Получаем rvalue для выражения
      Value* val = Val->getRValue(Context);

      // Получаем типы на которые указывают оба указателя
      TypeAST* next1 = ((PointerTypeAST*)ExprType)->Next;
      TypeAST* next2 = ((PointerTypeAST*)Val->ExprType)->Next;

      // У нас есть специальная обработка для классов и приведд
      // родительскому классу
      if (next1->isAggregate() && next1->isBaseOf(next2)) {
        // Получаем объявление для обоих классов
```

```
        SymbolAST* s1 = next1->getSymbol();
        SymbolAST* s2 = next2->getSymbol();

        // Производим преобразование
        return convertToBase(Context, val, s1, s2, Val->canBeN
      }

      // Возвращаем "как есть"
      return val;
    // Проверяем на приведение массива
    } else if (isa<ArrayTypeAST>(Val->ExprType)) {
    ...
}
```

Вызов функции или методов класса с учетом динамической диспетчеризации:

▾ Hidden text

```
Value* CallExprAST::getRValue(SLContext& Context) {
  Value* callee = nullptr;
  std::vector< Value* > args;
  ExprList::iterator it = Args.begin();
  Value* forcedReturn = nullptr;

  assert(isa<FuncDeclAST>(CallFunc));
  FuncDeclAST* funcDecl = (FuncDeclAST*)CallFunc;
  Type* funcRawType = CallFunc->getType()->getType();
  assert(isa<FunctionType>(funcRawType));
  FunctionType* funcType = static_cast<FunctionType*>(funcRawT
```

```
// Проверяем, является ли функция виртуальным методом или не
if (funcDecl->isVirtual() || funcDecl->isOverride()) {
  // Это виртуальный метод, производим динамическую
  // диспетчиризацию
  assert(isa<MemberAccessExprAST>(Callee));
  MemberAccessExprAST* memAccess = (MemberAccessExprAST*)Cal

  // Проверяем является ли Callee указателем или нет
  if (isa<DerefExprAST>(memAccess->Val)) {
    // Это разыменование указателя
    DerefExprAST* derefExpr = (DerefExprAST*)memAccess->Val;

    // Проверяем, что это "super", т. к. для "super" вызов
    // должен быть обычным, а не динамическим
    if ((isa<IdExprAST>(derefExpr->Val) &&
      ((IdExprAST*)derefExpr->Val)->Val == Name::Super)) {
      // Это "super" в качестве параметра "this", делаем выз
      // не виртуальным
      callee = CallFunc->getValue(Context);
    } else {
      // Этот вызов должен быть динамическим

      // Получаем "this"
      callee = memAccess->Val->getLValue(Context);

      // Получаем тип указателя
      PointerType* ptrType = PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace()
      );

      // Загружаем таблицу виртуальных методов
      callee = Context.TheBuilder->CreateLoad(ptrType, calle
      // Получаем виртуальный метод из таблицы по его индекс
      callee = Context.TheBuilder->CreateGEP(
        ptrType,
        callee,
```

```
        getConstInt(((FuncDeclAST*)CallFunc)->OffsetOf)
      );
      // Получаем адрес реального метода для вызова
      callee = Context.TheBuilder->CreateLoad(ptrType, calle
    }
  } else {
    // Был ли "this" задан явно извне или нет
    if (memAccess->ForceThis) {
      // Этот вызов должен быть динамическим

      // Получаем "this"
      callee = memAccess->Val->getRValue(Context);

      // Получаем тип указателя
      PointerType* ptrType = PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace()
      );

      // Загружаем таблицу виртуальных методов
      callee = Context.TheBuilder->CreateLoad(ptrType, calle
      // Получаем виртуальный метод из таблицы по его индекс
      callee = Context.TheBuilder->CreateGEP(
        ptrType,
        callee,
        getConstInt(((FuncDeclAST*)CallFunc)->OffsetOf)
      );
      // Получаем адрес реального метода для вызова
      callee = Context.TheBuilder->CreateLoad(ptrType, calle
    } else {
      // Это обычная функция, получаем ее адрес
      callee = CallFunc->getValue(Context);
    }
  }
} else {
  // Это обычная функция, получаем ее адрес
  callee = CallFunc->getValue(Context);
```

```
}

// Для методов классов у нас есть специальная обработка
if (isa<MemberAccessExprAST>(Callee)) {
  MemberAccessExprAST* memAccess = (MemberAccessExprAST*)Cal
    // Был ли "this" задан явно извне или нет
  if (!memAccess->ForceThis) {
    // Нет. Получаем lvalue для левого операнда
    Value* val = (*it)->getLValue(Context);
    MemberAccessExprAST* memAccess = (MemberAccessExprAST*)C

    // Производим преобразование указателя класса в указател
    // родительский класс, если это необходимо
    if (memAccess->ThisSym->Parent != memAccess->ThisAggr) {
      val = convertToBase(
        Context,
        val,
        memAccess->ThisSym->Parent,
        memAccess->ThisAggr,
        false
      );
    }

    // Добавляем аргумент для "this"
    args.push_back(val);
    ++it;
  } else {
    // Специальная обработка для случае когда "this" был явн
    // задан извне
    MemberAccessExprAST* memAccess = (MemberAccessExprAST*)C
    assert(memAccess->ThisSym->Parent == memAccess->ThisAggr
    // Получаем rvalue и устанавливаем тип результат вызова
    forcedReturn = (*it)->getRValue(Context);

    // Добавляем аргумент для "this"
    args.push_back(forcedReturn);
    ++it;
```

```
    }
  }

  // Производим генерацию кода для каждого аргумента функции и
  // добавляем их к списку аргументов функции
  for (ExprList::iterator end = Args.end(); it != end; ++it) {
    Value* v = (*it)->getRValue(Context);
    args.push_back(v);
  }

  // Если у нас тип результата вызова был указан явно
  if (forcedReturn) {
    // "this" был явно задан извне. Генерируем вызов функции,
    // качестве результата возвращаем значение полученное ране
    Context.TheBuilder->CreateCall(funcType, callee, args);
    return forcedReturn;
  }

  // Генерируем вызов функции
  return Context.TheBuilder->CreateCall(funcType, callee, args
}
```

Генерация кода для инструкций

Т.к. в функции в различных местах ее тела могут объявляться переменные, которые могут быть объектами классов у которых есть деструктор, то необходимо для этих переменных делать вызов деструкторов этих объектов, что бы произвести очистку ресурсов. Так же язык поддерживает различные управляющие конструкции, которые могут усложнить эту задачу. Рассмотрим варианты решения данной задачи на следующем примере программы на языке C++ (A — класс, у которого есть объявленный деструктор, который нужно вызвать для очистки ресурсов занимаемых данным объектом):

Hidden text

```
void foo(int z) {
  A a;
  for (int i = 0; i < 10; ++i) {
    A b;
    if (i == 4) {
      continue;
    }
    if (i == 6) {
      break;
    }
    if (i == z) {
      return;
    }
  }
}
```

Если решить поставленную задачу "в лоб", то данная программа может быть переписана в виде следующего псевдокода:

Hidden text

```
void foo(int z) {
  A a;
  {
    int i = 0;

loopcond:
    if (i < 10) {
```

```
      goto loopbody;
    } else {
      goto loopend;
    }

loopbody:
    A b;

    if (i == 4) {
      b.~A();
      goto loopcont;
    }

    if (i == 6) {
      b.~A();
      goto loopend;
    }

    if (i == z) {
      b.~A();
      a.~A();
      return;
    }

loopcont:
    ++i;
    goto loopcond;

loopend:
  }

  a.~A();
  return;
}
```

Как видно из примера выше, то получившийся код содержит много дублирующей информации, которая для более сложных функций будет только увеличивать количество дублируемого кода, а так же увеличивает сложность генерации кода для данной функции.

В одной из предыдущих частей я показывал оптимизацию для функций у которых есть возвращаемое значение, если развить данную идею и ввести некоторые дополнения, то генерация кода для сложных функций может быть значительно упрощена, а так же дублирование кода будет полностью исключено.

Опишем список дополнений:

1. Если в функции есть переменные для которых нужно будет вызвать деструкторы, то мы вводим новую переменную cleanupValue;
2. Для каждого блока, где мы имеем переменные с деструкторами (и если в нем есть "break", "continue" или "return"), мы будем создавать новую метку CleanupLoc, на которую будет передаваться управление в не зависимости от того, были ли вызваны инструкции изменения стандартного потока выполнения программы или нет. После этой метки мы производим вызов деструкторов всех объектов объявленных в этом блоке, для которых их нужно вызвать, в обратном порядке их объявления. После этого мы создаем инструкцию switch, которая делает следующее:

- Если значение cleanupValue равно 0, то мы переводим управление на FallthorughLoc родительского блока;
- Если значение cleanupValue равно 2, то мы переводим управление на ContinueLoc родительского блока (доступно только в циклах);
- Если значение cleanupValue равно 3, то мы переводим управление на BreakLoc родительского блока (доступно только в циклах);
- Во всех остальных случаях переводим управление на CleanupLoc родительской функции, либо на ReturnLoc (если это блок самого

верхнего уровня, у которого есть переменные для очистки).

3. Если блок, для которого мы генерируем код является телом цикла (и в теле цикла есть "break", "continue" или "return"), то после всего кода цикла (и перед выполнения кода по метке CleanupLoc), мы должны сделать следующее:

- Записать в cleanupValue значение 0;
- Перейти на метку CleanupLoc.

4. Для каждой инструкции "continue", если должна произведена очистка в данном блоке, мы должны будем сделать следующее:

- Записать в cleanupValue значение 2;
- Перейти на метку CleanupLoc.

5. Для каждой инструкции "break", если должна произведена очистка в данном блоке, мы должны будем сделать следующее:

- Записать в cleanupValue значение 2;
- Перейти на метку CleanupLoc.

6. Для каждой инструкции "return", если должна произведена очистка в данном блоке, мы должны будем сделать следующее:

- Записать в cleanupValue значение 1;
- Перейти на метку CleanupLoc.

Ниже рассмотрим псевдокод исходной функции, если к ней применить алгоритм приведенный выше:

Hidden text

```
void foo(int z) {
  int cleanupValue;
  A a;
  {
    int i = 0;

loopcond:
    if (i < 10) {
      goto loopbody;
    } else {
      goto loopend;
    }

loopbody:
    A b;

    if (i == 4) {
      cleanupValue = 2;
      goto cleanupLoc2;
    }

    if (i == 6) {
      cleanupValue = 3;
      goto cleanupLoc2;
    }

    if (i == z) {
      cleanupValue = 1;
      goto cleanupLoc2;
    }

    cleanupValue = 0;
    goto cleanupLoc2;

cleanupLoc2:
    b.~A();
```

```
    switch (cleanupValue) {
      case 0: goto loopcont;
      case 2: goto loopcont;
      case 3: goto loopend;
      default: goto cleanupLoc1;
    }

loopcont:
    ++i;
    goto loopcond;

loopend:
  }

cleanupLoc1:
  a.~A();
  goto returnLoc;

returnLoc:
  return;
}
```

Из приведенного кода выше видно, что несмотря на то, что он стал немного громоздким, но исчезло дублирование вызовов деструкторов и код стал более структурированным (Clang генерирует схожий код).

Примечание: В коде ниже есть еще одна оптимизация, которая исключает генерацию конструкций вида:

```
  br label1;
label1:
  br label2;
label2:
```

```
  br label3;
label3:
  br someLabel;
```

данная оптимизация приведет к генерации кода вида (все пустые блоки будет позже удалены после генерации тела всей функции):

```
  br someLabel;
label1:
label2:
label3:
```

Рассмотрим реализацию алгоритма описанного выше:

Hidden text

```
llvm::Value* BlockStmtAST::generatePartialCode(SLContext& Cont
  StmtList::iterator it, StmtList::iterator end) {
  // Инициализируем блоки "break", "continue" и "return" для
  // LandingPadAST текущего блока на основе родительского блок
  LandingPadAST* parent = LandingPad->Prev;
  LandingPad->BreakLoc = parent->BreakLoc;
  LandingPad->ContinueLoc = parent->ContinueLoc;
  LandingPad->ReturnLoc = parent->ReturnLoc;

  // CleanupLoc должна быть задана по разному, в зависимости с
  // того, нужно ли произвести очистку ресурсов или нет
  if (!LandingPad->NeedCleanup || CleanupList.empty()) {
    // Нет. Установить на основе родительского блока
    LandingPad->CleanupLoc = parent->CleanupLoc;
  } else {
    // Да. Создать новый блок
```

```
  LandingPad->CleanupLoc = BasicBlock::Create(getGlobalConte
}

// Создаем новый блок FallthroughLoc (этот блок может быть
// использован в дочерних инструкциях, в качестве точки возв
// (например для выхода из цикла))
LandingPad->FallthroughLoc = BasicBlock::Create(
  getGlobalContext(),
  "block"
);

// Нам нужно сохранять состояние последнего обработанного бл
// а именно:
// 1. Инструкция, которая была сгенерирвана последней;
// 2. Последний использованный блок для возврата;
// 3. Последняя точка вставки кода.
StmtAST* lastStmt = nullptr;
BasicBlock* lastFallThrough = nullptr;
BasicBlock* lastInsertionPoint = nullptr;

// Генерация кода для всех вложенных инструкций
for (; it != end; ++it) {
  // Сохраняем FallthroughLoc (т. к. при использовании его
  // вложенная инструкция должна будет его обнулить)
  BasicBlock* fallthroughBB = LandingPad->FallthroughLoc;
  lastFallThrough = LandingPad->FallthroughLoc;

  // Генерируем код для вложенной инструкции
  lastStmt = *it;
  lastStmt->generateCode(Context);

  // Проверяем была ли FallthroughLoc использована во вложен
  // инструкции или нет
  if (!LandingPad->FallthroughLoc) {
    // Была. Поэтому нужно добавить данный блок в конец функ
    // как ее часть и задать его в качестве точки для вставк
    // нового кода
```

```
    lastInsertionPoint = Context.TheBuilder->GetInsertBlock(
    Context.TheFunction->getBasicBlockList().push_back(
      fallthroughBB
    );
    Context.TheBuilder->SetInsertPoint(fallthroughBB);

    // Записать в FallthroughLoc новый созданный блок, для
    // последующего использования во вложенных инструкциях
    LandingPad->FallthroughLoc = BasicBlock::Create(
      getGlobalContext(),
      "block"
    );
  }
}

// Делаем подсчет количество "break" "continue" и "return"
// инструкций в данном блоке (в последующих статьях будет
// ясно зачем они нужны)
parent->Breaks += LandingPad->Breaks;
parent->Continues += LandingPad->Continues;
parent->Returns += LandingPad->Returns;

// Проверяем нужно ли нам генерировать вызовы деструкторов и
// нет
if (LandingPad->NeedCleanup) {
  // Нужно

  // Если FallthroughLoc, которую мы создали для вложенных
  // инструкций не была использована, то производим ее очист
  if (!LandingPad->FallthroughLoc->hasNUsesOrMore(1)) {
    delete LandingPad->FallthroughLoc;
  }

  // Если список переменных для очистки пуст, то нам ничего
  // нужно делать
  if (CleanupList.empty()) {
    // Проверяем, что текущий блок имеет инструкцию завершен
```

```
  // блока (условный или безусловный переход)
  if (!Context.TheBuilder->GetInsertBlock()->getTerminator
    // Генерируем инструкцию перехода на FallthroughLoc
    // родительского блока и помечаем, ее как использованн
    // (путем присвоения nullptr)
    Context.TheBuilder->CreateBr(parent->FallthroughLoc);
    parent->FallthroughLoc = nullptr;
    return nullptr;
  }

  return nullptr;
}

// Проверяем, что текущий блок имеет инструкцию завершения
// или безусловный переход)
if (!Context.TheBuilder->GetInsertBlock()->getTerminator()
  // Проверяем была ли последняя сгенерированная инструкци
  // блоком, который был создан на основе объявления
  // объекта класса
  if (lastStmt && isa<BlockStmtAST>(lastStmt) &&
    ((BlockStmtAST*)lastStmt)->IsPromoted) {
    // Создаем безусловный переход на блок очистки
    Context.TheBuilder->CreateBr(LandingPad->CleanupLoc);
  } else {
    // Необходимо установить значение 0 в переменную стату
    // очистки, а так же создать безусловный переход на бл
    // очистки
    Context.TheBuilder->CreateStore(
      getConstInt(0),
      parent->getCleanupValue()
    );
    Context.TheBuilder->CreateBr(LandingPad->CleanupLoc);
  }
}

// Добавляем блок для очистки в конец функции и устанавлив
// ее, как точку для генерации кода
```

```
Context.TheFunction->getBasicBlockList().push_back(
  LandingPad->CleanupLoc
);
Context.TheBuilder->SetInsertPoint(LandingPad->CleanupLoc)
LandingPad->CleanupLoc->setName("cleanup.block");

// Генерируем вызовы деструкторов для всех созданных объек
// в данном блоке (мы должны их вызвать в обратном порядке
// их объявления)
for (ExprList::reverse_iterator it = CleanupList.rbegin(),
  end = CleanupList.rend(); it != end; ++it) {
  (*it)->getRValue(Context);
}

// Проверяем были ли у нас "break", "continue" и "return"
// инструкции в данном блоке или нет
if (!LandingPad->Returns && !LandingPad->Breaks &&
  !LandingPad->Continues) {
  // Нет, их не было. Генерируем инструкцию перехода на
  // FallthroughLoc родительского блока и помечаем, ее как
  // использованную (путем присвоения nullptr)
  Context.TheBuilder->CreateBr(parent->FallthroughLoc);
  parent->FallthroughLoc = nullptr;
} else {
  // Да, есть. Определяем имеет ли родительский блок Clean
  // или нет
  if (parent->CleanupLoc) {
    // Да. Определяем был ли блок создан из объявления объ
    // или нет
    if (IsPromoted) {
      // Да.  Генерируем инструкцию перехода на Fallthroug
      // родительского блока
      Context.TheBuilder->CreateBr(parent->FallthroughLoc)
    } else {
      // Нет. Нам нужно создать "switch", который будет
      // перенаправлять либо на родительский блок очистки,
      // либо на родительский блок FallthroughLoc, в
```

```
      // зависимости от статуса очистки
      Value* val = Context.TheBuilder->CreateLoad(
        Type::getInt32Ty(getGlobalContext()),
        LandingPad->getCleanupValue()
      );
      SwitchInst* switchBB = Context.TheBuilder->CreateSwi
        val,
        parent->CleanupLoc
      );

      switchBB->addCase(getConstInt(0), parent->Fallthroug
    }
  } else {
    // Это блок очистки самого высокого уровня
    LandingPadAST* prev = parent;

    // Пытаемся найти предыдущий блок у которого есть блок
    // очистки
    while (prev->Prev) {
      if (prev->CleanupLoc) {
        break;
      }

      prev = prev->Prev;
    }

    // Определяем имеет ли родительский блок CleanupLoc ил
    if (prev->CleanupLoc) {
      // Да. Нам нужно создать "switch", который будет
      // перенаправлять либо на родительский блок очистки,
      // либо на родительский блок FallthroughLoc, в завис
      // от статуса очистки, так же мы могли иметь вызовы
      // инструкций "break", "continue", которые тоже долж
      // быть добавлены в "switch"
      Value* val = Context.TheBuilder->CreateLoad(
        Type::getInt32Ty(getGlobalContext()),
        LandingPad->getCleanupValue()
```

```
    );
    SwitchInst* switchBB = Context.TheBuilder->CreateSwi
      val,
      prev->CleanupLoc
    );

    switchBB->addCase(getConstInt(0), parent->Fallthroug

    // Если есть "continue", то добавляем новую ветку в
    // "switch" с переходом на ContinueLoc
    if (LandingPad->Continues) {
      switchBB->addCase(getConstInt(2), parent->Continue
    }

    // Если есть "break", то добавляем новую ветку в
    // "switch" с переходом на BreakLoc
    if (LandingPad->Breaks) {
      switchBB->addCase(getConstInt(3), parent->BreakLoc
    }
  } else {
    // Проверяем были ли у нас "break", "continue" и "re
    // инструкции в данном блоке или нет
    if (LandingPad->Returns || LandingPad->Breaks ||
      LandingPad->Continues) {
      // Проверяем, что это не были "break" и "continue"
      // блок возврата из функции отличается от блока
      // FallthroughLoc родительского блока
      if (prev->ReturnLoc == parent->FallthroughLoc &&
        !LandingPad->Breaks && !LandingPad->Continues) {
        // Генерируем безусловный переход на выход из фу
        Context.TheBuilder->CreateBr(prev->ReturnLoc);
      } else {
        // Нам нужно создать "switch", который будет
        // перенаправлять либо на выход из функции, либо
        // на родительский блок FallthroughLoc, в зависи
        // от статуса очистки, так же мы могли иметь выз
        // инструкций "break", "continue", которые тоже
```

```
      // быть добавлены в "switch"
      Value* val = Context.TheBuilder->CreateLoad(
        Type::getInt32Ty(getGlobalContext()),
        LandingPad->getCleanupValue()
      );
      SwitchInst* switchBB =
        Context.TheBuilder->CreateSwitch(
          val,
          prev->ReturnLoc
        );

      switchBB->addCase(
        getConstInt(0),
        parent->FallthroughLoc
      );

      // Если есть "continue", то добавляем новую ветк
      // "switch" с переходом на ContinueLoc
      if (LandingPad->Continues) {
        switchBB->addCase(
          getConstInt(2),
          parent->ContinueLoc
        );
      }

      // Если есть "break", то добавляем новую ветку в
      // "switch" с переходом на BreakLoc
      if (LandingPad->Breaks) {
        switchBB->addCase(getConstInt(3), parent->Brea
      }
    }
  } else {
    // Нет, их не было. Генерируем инструкцию перехода
    // FallthroughLoc родительского блока и помечаем,
    // как использованную (путем присвоения nullptr)
    Context.TheBuilder->CreateBr(parent->FallthroughLc
  }
```

```
      }
    }

    parent->FallthroughLoc = nullptr;
  }
} else {
  // У нас нет блока для очистки

  // Проверяем, что текущий блок имеет инструкцию завершения
  // блока (условный или безусловный переход)
  if (!Context.TheBuilder->GetInsertBlock()->getTerminator()
    // Получаем количество инструкций в текущем блоке
    const unsigned u =
      Context.TheBuilder->GetInsertBlock()->size();

    // У нас есть оптимизация для варианта, когда блок пуст
    if (u == 0) {
      // Получить текущий блок
      BasicBlock *thisBlock = Context.TheBuilder->GetInsertB

      // Если на этот блок есть только одна ссылка
      if (thisBlock->hasNUses(1)) {
        // Получаем блок, который ссылается на этот, а так ж
        // инструкцию терминатор из него
        BasicBlock *prevBlock = thisBlock->getSinglePredeces
        Instruction *prevTerminator = prevBlock->getTerminat

        // Если родительский блок цикл, то нам необходимо
        // заменить все использования текущего блока на
        // FallthroughLoc родительского блока
        if (parent->IsLoop) {
          thisBlock->replaceAllUsesWith(parent->FallthroughL
        }

        // Если FallthroughLoc, которую мы создали для вложе
        // инструкций не была использована, то производим ее
        // очистку
```

```
      if (!LandingPad->FallthroughLoc->hasNUsesOrMore(1))
        delete LandingPad->FallthroughLoc;
      }

      return nullptr;
    }
  }

  // Генерируем инструкцию перехода на FallthroughLoc
  // родительского блока и помечаем, ее как использованную
  // (путем присвоения nullptr)
  Context.TheBuilder->CreateBr(parent->FallthroughLoc);
  parent->FallthroughLoc = nullptr;

  // Если FallthroughLoc, которую мы создали для вложенных
  // инструкций не была использована, то производим ее очи
  if (!LandingPad->FallthroughLoc->hasNUsesOrMore(1)) {
    delete LandingPad->FallthroughLoc;
  }
// Проверяем, что  FallthroughLoc был использован во вложе
// инструкциях
} else if (LandingPad->FallthroughLoc->hasNUsesOrMore(1))
  // Был использован, нужно добавить его в конец функции,
  // установить его в качестве места вставки нового кода,
  // также создаем безусловный переход на FallthroughLoc
  // родительского блока, а так же помечаем его как
  // использованную (путем присвоения nullptr)
  Context.TheFunction->getBasicBlockList().push_back(
    LandingPad->FallthroughLoc
  );
  Context.TheBuilder->SetInsertPoint(LandingPad->Fallthrou
  Context.TheBuilder->CreateBr(parent->FallthroughLoc);
  parent->FallthroughLoc = nullptr;
} else {
  // Нет. Производим ее очистку
  delete LandingPad->FallthroughLoc;
}
```

```
  }

  return nullptr;
}

Value* LandingPadAST::getCleanupValue() {
  LandingPadAST* prev = this;

  // Находим корневой LandingPadAST
  while (prev->Prev) {
    prev = prev->Prev;
  }

  // Возвращаем ее CleanupValue
  return prev->CleanupValue;
}

Value* BreakStmtAST::generateCode(SLContext& Context) {
  // Есть ли у нас блок для очистки?
  if (BreakLoc->NeedCleanup) {
    // Да. Нужно сохранить значение 3 в переменную статус очис
    // и произвести безусловный переход на CleanupLoc
    Context.TheBuilder->CreateStore(
      getConstInt(3),
      BreakLoc->getCleanupValue()
    );
    Context.TheBuilder->CreateBr(BreakLoc->CleanupLoc);
    return nullptr;
  }

  // Создать безусловный переход на  BreakLoc
  Context.TheBuilder->CreateBr(BreakLoc->BreakLoc);
  return nullptr;
}

Value* ContinueStmtAST::generateCode(SLContext& Context) {
  // Есть ли у нас блок для очистки?
```

```
  if (ContinueLoc->NeedCleanup) {
    // Да. Нужно сохранить значение 2 в переменную статус очис
    // и произвести безусловный переход на CleanupLoc
    Context.TheBuilder->CreateStore(
      getConstInt(2),
      ContinueLoc->getCleanupValue()
    );
    Context.TheBuilder->CreateBr(ContinueLoc->CleanupLoc);
    return nullptr;
  }

  // Создать безусловный переход на  ContinueLoc
  Context.TheBuilder->CreateBr(ContinueLoc->ContinueLoc);
  return nullptr;
}

Value* ReturnStmtAST::generateCode(SLContext& Context) {
  // Есть ли у нас блок для очистки?
  if (ReturnLoc->NeedCleanup) {
    // Да. Нужно сохранить значение 1 в переменную статус очис
    // и произвести безусловный переход на CleanupLoc
    Context.TheBuilder->CreateStore(
      getConstInt(1),
      ReturnLoc->getCleanupValue()
    );

    // Проверяем, что у нас есть возвращаемое значение
    if (Expr) {
      // Генерируем код для выражения с возвращаемым значением
      Value* retVal = Expr->getRValue(Context);
      // Создаем инструкцию "store" для сохранения возвращаемо
      // значения в переменной, для хранения результата выполн
      // функции
      Context.TheBuilder->CreateStore(
        retVal,
        ReturnLoc->getReturnValue()
      );
```

```
    }

    // Создаем безусловный переход на CleanupLoc
    Context.TheBuilder->CreateBr(ReturnLoc->CleanupLoc);
    return nullptr;
  }
  ...
}

llvm::Value* IfStmtAST::generateCode(SLContext& Context) {
  // Инициализируем блоки "break", "continue" и "return", а та
  // же блок очистки для LandingPadAST текущего блока на основ
  // родительского блока
  LandingPadAST* prev = LandingPad->Prev;
  LandingPad->CleanupLoc = prev->CleanupLoc;
  LandingPad->ReturnLoc = prev->ReturnLoc;
  LandingPad->FallthroughLoc = prev->FallthroughLoc;
  LandingPad->ContinueLoc = prev->ContinueLoc;
  LandingPad->BreakLoc = prev->BreakLoc;
  ...
}
```

Генерация кода для объявлений

Поддержка объектов класса в объявлениях:

- Hidden text

```
Value* VarDeclAST::generateCode(SLContext& Context) {
  assert(SemaState >= 5);
  // Получаем адрес переменной (т. к. память под саму переменн
  // уже должна была выделена ранее во время генерации кода дл
```

```
  // функции)
  Value* val = getValue(Context);

  // Специальная обработка для классов с конструктором
  if (Val && isa<ClassTypeAST>(ThisType)) {
    Val->getRValue(Context);
    return val;
  }
  ...
}
```

Таблица виртуальных методов:

▾ Hidden text

```
Value* VTableAST::generateCode(SLContext& Context) {
  // Исключаем повторную генерацию кода
  if (CodeValue) {
    return CodeValue;
  }

  // Создаем таблицу виртуальных методов (все слоты будут запо
  // позже)
  std::vector< Constant* > vtblEntries(CurOffset);

  // Обрабатываем все методы и наборы перегруженных методов в
  // таблице виртуальных методов
  for (SymbolMap::iterator it = Decls.begin(), end = Decls.end
    it != end; ++it) {
    if (isa<OverloadSetAST>(it->second)) {
      // Это набор перегруженных функций
```

```
    OverloadSetAST* overloadSet = (OverloadSetAST*)it->secor

    // Обрабатываем все методы в наборе перегруженных функци
    for (SymbolList::iterator it2 = overloadSet->Vars.begin(
      end2 = overloadSet->Vars.end(); it2 != end2; ++it2) {
      FuncDeclAST* fnc = (FuncDeclAST*)*it2;
      // Добавляем метод в соответствующий слот
      vtblEntries[fnc->OffsetOf] = ConstantExpr::getBitCast(
        (Function*)fnc->getValue(Context),
        PointerType::get(
          getGlobalContext(),
          getSLContext().TheTarget->getProgramAddressSpace()
        )
      );
    }
  } else {
    // Это обычная функция
    FuncDeclAST* fnc = (FuncDeclAST*)it->second;
    // Добавляем метод в соответствующий слот
    vtblEntries[fnc->OffsetOf] = ConstantExpr::getBitCast(
      (Function*)fnc->getValue(Context),
      PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace()
      )
    );
  }
}

llvm::SmallString< 128 > s;
llvm::raw_svector_ostream output(s);

// Генерируем имя для таблицы виртуальных методов для класса
output << "_PTV";
mangleAggregateName(output, Parent);

// Создаем массив указателей
```

```
  ArrayType* tableType = ArrayType::get(
    PointerType::get(
      getGlobalContext(),
      getSLContext().TheTarget->getProgramAddressSpace()
    ),
    CurOffset
  );
  // Добавляем таблицу виртуальных методов к списку глобальных
  // объявлений
  GlobalVariable* val =
    (GlobalVariable*)Context.TheModule->getOrInsertGlobal(
      output.str(),
      tableType
    );
  // Инициализируем виртуальную таблицу данными
  val->setInitializer(ConstantArray::get(tableType, vtblEntrie
  // Сохраняем тип для этой таблицы
  VTblType = tableType;

  // Возвращаем только что сгенерированую глобальную переменну
  return CodeValue = val;
}
```

Генерация кода для объявлений классов:

- Hidden text

```
llvm::Value* ClassDeclAST::getValue(SLContext& Context) {
  // Генерируем тип для класса
  ThisType->getType();
```

```
  // Генерируем тип для родительского класса, если он есть
  if (BaseClass) {
    BaseClass->getType();
  }

  return nullptr;
}

llvm::Value* ClassDeclAST::generateCode(SLContext& Context) {
  assert(SemaState >= 5);
  // Генерируем код для класса
  getValue(Context);

  // Генерируем таблицу виртуальных методов, если она нужна
  if (VTbl) {
    VTbl->generateCode(Context);
  }

  // Генерация кода для всех классов/структур и методов
  // объявленных в классе
  for (SymbolList::iterator it = Vars.begin(), end = Vars.end(
    it != end; ++it) {
    if (!isa<VarDeclAST>(*it)) {
      (*it)->generateCode(Context);
    }
  }

  return nullptr;
}
```

В генерации кода функций у нас появилась специальная обработка для конструкторов, деструкторов, а также поддержка статуса очистки (если в

функции существуют объявления объектов класса с деструкторами, которые должны быть автоматически вызваны при выходе из блоков):

Hidden text

```
Value* FuncDeclAST::getValue(SLContext& Context) {
  // Генерируем код для объявления только один раз
  if (CodeValue) {
    return CodeValue;
  }

  SmallString< 128 > str;
  raw_svector_ostream output(str);

  // Производим генерацию имени функции (для "main" у нас есть
  // специальная обработка)
  if (!(Id->Length == 4 && memcmp(Id->Id, "main", 4) == 0)) {
    // Генерация функций для методов класса чуть отличается
    if (needThis()) {
      // Для методов класса необходимо к имени функции добавит
      // полное имя агрегата
      assert(AggregateSym);
      output << "_P";
      mangleAggregateName(output, AggregateSym);
      output << Id->Length << StringRef(Id->Id, Id->Length);
      ThisType->toMangleBuffer(output);
    } else {
      // Генерация уникального декорированного имени функции
      output << "_P" << Id->Length << StringRef(Id->Id, Id->Le
      ThisType->toMangleBuffer(output);
    }
  } else {
    output << "main";
  }
```

```
  // Создать функцию с внешним видом связанности для данного
  // объявления
  CodeValue = Function::Create((FunctionType*)ThisType->getTyp
    Function::ExternalLinkage, output.str(), nullptr);
  Context.TheModule->getFunctionList().push_back(CodeValue);

  return CodeValue;
}

Value* FuncDeclAST::generateCode(SLContext& Context) {
  ...
  // Создать блок для возврата из функции и установить его в
  // качестве FallthroughLoc
  LandingPad->ReturnLoc = BasicBlock::Create(
    getGlobalContext(),
    "return.block"
  );
  LandingPad->FallthroughLoc = LandingPad->ReturnLoc;

  // Создаем переменную со статусом очистки, если она необходи
  if (LandingPad->NeedCleanup) {
    LandingPad->CleanupValue = Context.TheBuilder->CreateAlloc
      Type::getInt32Ty(getGlobalContext()),
      0U,
      "cleanup.value"
    );
  }

  Function* oldFunction = Context.TheFunction;
  Context.TheFunction = CodeValue;

  if (isCtor()) {
    // Специальная обработка для конструктора
    assert(isa<BlockStmtAST>(Body));
    assert(isa<ClassDeclAST>(Parent));
    BlockStmtAST* blockStmt = (BlockStmtAST*)Body;
    ClassDeclAST* classDecl = (ClassDeclAST*)Parent;
```

```
VTableAST* parentVtbl = nullptr;

StmtList::iterator it = blockStmt->Body.begin();
StmtList::iterator end = blockStmt->Body.end();

// Если у класса есть родительский класс, то мы должны
// произвести специальную обработку
if (classDecl->BaseClass) {
  // 1-я инструкция в теле конструкция – объявление переме
  // "super". Генерируем ее и переходим к следующей инстру
  (*it)->generateCode(Context);
  ++it;

  // Если родительский класс является классом, а не структ
  // то мы должны произвести специальную обработку
  if (isa<ClassTypeAST>(classDecl->BaseClass)) {
    ClassDeclAST* baseDecl =
      (ClassDeclAST*)classDecl->BaseClass->getSymbol();
    parentVtbl = baseDecl->VTbl;

    // Проверяем, что у родительского класса есть конструк
    if (baseDecl->Ctor) {
      // Он есть. Генерируем код для его вызова и переходи
      // следующей инструкции
      (*it)->generateCode(Context);
      ++it;
    }
  }
}

// Если у класса есть таблица виртуальных методов и она
// отличается от таблицы класса родителя, то мы должны
// сделать специальную обработку
if (classDecl->VTbl != parentVtbl) {
  SymbolAST* thisParam = find(Name::This);
  assert(thisParam != nullptr);
  // Генерируем код для параметра "this" и производим загр
```

```
    // параметра
    Value* val = thisParam->getValue(Context);

    val = Context.TheBuilder->CreateLoad(
      PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace()
      ),
      val
    );

    // Генерируем код для таблицы виртуальных методов
    Value* vtblVal = classDecl->VTbl->generateCode(Context);

    std::vector< Value* > idx;

    idx.push_back(getConstInt(0));
    idx.push_back(getConstInt(0));

    // Генерируем инструкцию GetElementPtr для загрузки адре
    // таблицы виртуальных методов из глобальной переменной
    vtblVal = Context.TheBuilder->CreateInBoundsGEP(
      classDecl->VTbl->VTblType,
      vtblVal,
      idx
    );
    // Сохраняем указатель на таблицу виртуальных методов в
    // экземпляре класса
    val = Context.TheBuilder->CreateStore(vtblVal, val);
  }

  // Генерируем код для оставшейся части тела функции
  blockStmt->generatePartialCode(Context, it, end);
} else if (isDtor()) {
  // У нас есть специальная обработка для деструкторов
  assert(isa<BlockStmtAST>(Body));
  assert(isa<ClassDeclAST>(Parent));
```

```
BlockStmtAST* blockStmt = (BlockStmtAST*)Body;
ClassDeclAST* classDecl = (ClassDeclAST*)Parent;

// Если у класса есть родительский класс и это класс, а не
// структура, то нам нужно произвести специальную обработк
if (classDecl->BaseClass &&
  isa<ClassTypeAST>(classDecl->BaseClass)) {
  ClassDeclAST* baseDecl =
    (ClassDeclAST*)classDecl->BaseClass->getSymbol();

  // Обрабатываем случай, когда у родительского класса ест
  // деструктор
  if (baseDecl->Dtor) {
    // Получаем первую и последнюю инструкции тела деструк
    StmtList::iterator it = blockStmt->Body.begin();
    StmtList::iterator end = blockStmt->Body.end();
    // Последняя инструкция будет вызов деструктора
    // родительского класса, генерацию которого мы произве
    // позже, поэтому мы должны исключить его из генерации
    --end;

    // Для деструктора мы должны создать новый блок
    // для FallthroughLoc, т. к. он может быть использован
    // в generatePartialCode
    BasicBlock* falthroughBB = BasicBlock::Create(
      getGlobalContext()
    );
    LandingPad->FallthroughLoc = falthroughBB;

    // Генерируем код для всех инструкций, кроме вызова
    // деструктора родительского класса
    blockStmt->generatePartialCode(Context, it, end);

    // Либо добавляем блок FallthroughLoc, либо удаляем в
    // зависимости от того был ли он использован или нет
    if (falthroughBB->hasNUsesOrMore(1)) {
      CodeValue->getBasicBlockList().push_back(falthroughB
```

```
    Context.TheBuilder->SetInsertPoint(falthroughBB);
  } else {
    delete falthroughBB;
  }

  // Устанавливаем в качестве FallthroughLoc блок Return
  LandingPad->FallthroughLoc = LandingPad->ReturnLoc;

  // Текущая инструкция для генерации — вызов деструктор
  // родительского класса. Но перед его вызовом нам
  // необходимо сгенерировать установить в качестве табл
  // виртуальных методов таблицу родительского класса
  // (если они отличаются, но родительский класс ее имее
  if (classDecl->VTbl != baseDecl->VTbl && baseDecl->VTb
    SymbolAST* thisParam = find(Name::This);
    assert(!thisParam);
    // Генерируем код для параметра "this" и производим
    // загрузку параметра
    Value* val = thisParam->getValue(Context);
    val = Context.TheBuilder->CreateLoad(
      PointerType::get(
        getGlobalContext(),
        getSLContext().TheTarget->getProgramAddressSpace
      ),
      val
    );
    // Генерируем код для таблицы виртуальных методов
    // родительского класса
    Value* vtblVal = baseDecl->VTbl->generateCode(Contex

    std::vector< Value* > idx;

    idx.push_back(getConstInt(0));
    idx.push_back(getConstInt(0));

    // Генерируем инструкцию GetElementPtr для загрузки
    // таблицы виртуальных методов родительского класса
```

```
        // глобальной переменной
        vtblVal = Context.TheBuilder->CreateInBoundsGEP(
          baseDecl->VTbl->VTblType,
          vtblVal,
          idx
        );
        // Сохраняем указатель на таблицу виртуальных методс
        // экземпляре класса
        val = Context.TheBuilder->CreateStore(vtblVal, val);
      }

      // Генерация кода для вызова деструктора родительскогс
      // класса
      (*end)->generateCode(Context);
    } else {
      // У нас нет никаких специальных обработок. Генерируем
      // код "как есть"
      Body->generateCode(Context);
    }
  } else {
    // У нас нет никаких специальных обработок. Генерируем
    // код "как есть"
    Body->generateCode(Context);
  }
} else {
  // Генерация кода для тела функции
  Body->generateCode(Context);
}

Context.TheFunction = oldFunction;

// Добавляем условный переход на блок выхода из функции, есл
// он еще не был сгенерирован
if (!Context.TheBuilder->GetInsertBlock()->getTerminator())
  Context.TheBuilder->CreateBr(LandingPad->ReturnLoc);
}
```

```
// Добавить блок для возврата из функции в конец функции и
// установить его в качестве точки генерации кода
CodeValue->getBasicBlockList().push_back(LandingPad->ReturnL
Context.TheBuilder->SetInsertPoint(LandingPad->ReturnLoc);

if (!ReturnType->isVoid()) {
  // Тип возвращаемого значение не "void". Создаем инструкци
  // "load" для загрузки возвращаемого значения
  Value* ret = Context.TheBuilder->CreateLoad(
    ReturnType->getType(),
    LandingPad->ReturnValue
  );
  // Генерация инструкции возврата из функции
  Context.TheBuilder->CreateRet(ret);
} else {
  // Генерация инструкции возврата из функции для "void"
  Context.TheBuilder->CreateRetVoid();
}

// Восстановление предыдущей точки генерации кода (если нужн
if (oldBlock) {
  Context.TheBuilder->SetInsertPoint(oldBlock);
}

Function::BasicBlockListType &blocksList =
  CodeValue->getBasicBlockList();

// Небольшая оптимизация. Мы удаляем все блоки из функции, у
// которых нет инструкции завершения
for (Function::BasicBlockListType::iterator it =
  blocksList.begin(), lastBlock = blocksList.end();
  it != lastBlock; ) {
  if (!it->getTerminator()) {
    Function::BasicBlockListType::iterator cur = it;

    ++it;
```

```
      blocksList.erase(cur);
    } else {
      ++it;
    }
  }

  // Проверка кода функции и оптимизация (если она включена)
  verifyFunction(*CodeValue, &llvm::errs());

  Compiled = true;

  return CodeValue;
}
```

Заключение

В данной части мы добавили в наш учебный язык поддержку классов с одиночным наследованием, перегрузку функции в зависимости от типов принимаемых аргументов, а так же поддержку динамической диспетчеризации методов класса. Так же мы рассмотрели генерацию конструкторов и деструкторов по умолчанию, генерацию таблицы виртуальных методов, автоматический вызов конструкторов и деструкторов для объектов класса, как членов других классов, так и в теле функций.

Полный исходный код доступен на github.

Вот мы и подошли концу серии. Надеюсь, что она была полезна тем, кто решил создать свой собственный язык программирования.

Возможно, если серия будет интересна, то она получит продолжение. Всем успехов в мире разработки языков программирования коммерческих и для души.