Министерство образования и науки Российской Федерации

Марийский Государственный Технический Университет

Кафедра ИиСП

**Пояснительная записка**

**к курсовой работе**

**по дисциплине**

**“Теория языков программирования и**

**методов трансляции”**

**Выполнили:** ст. группы ПС-41

Жданов А.В.,

Татаринов Т.Н.,

Шайхутдинов Р.Г.

**Проверила:**

Нехорошкова Л.Г.

Йошкар-Ола

2009 г.

Оглавление

Структура компилятора 3

Абстрактное синтаксическое дерево 5

Создание узлов6

Обработка ошибок7

Синтаксический анализатор 8

Грамматика языка 8

Структура анализатора11

Управление памятью13

Генерация кода 14

Структура генератора 14

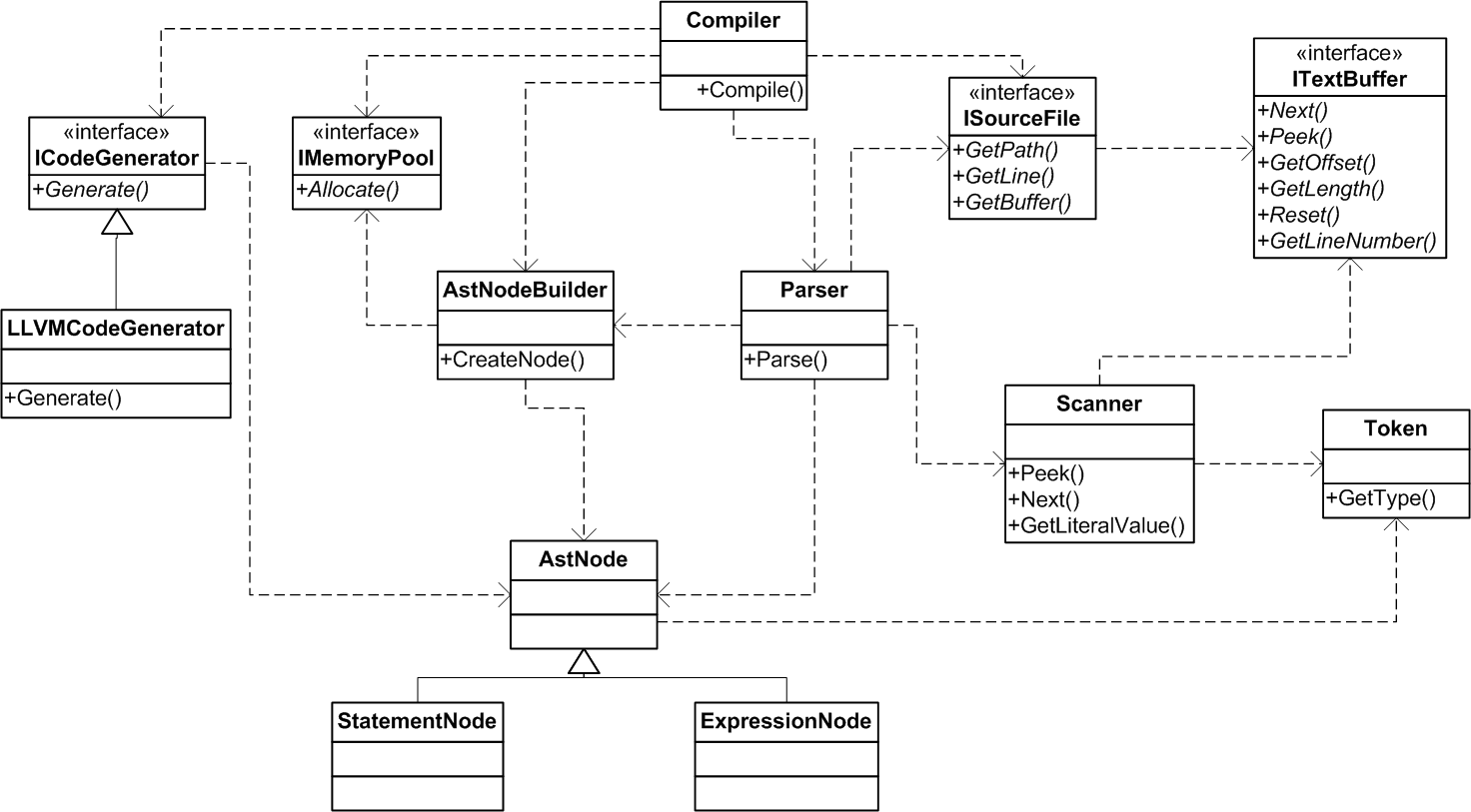
LLVM15

Процесс генерации19

Процесс разработки21

Список литературы22

**Структура компилятора**



Компилятор

LLVM IR

Кодогенератор

Синтаксическое дерево

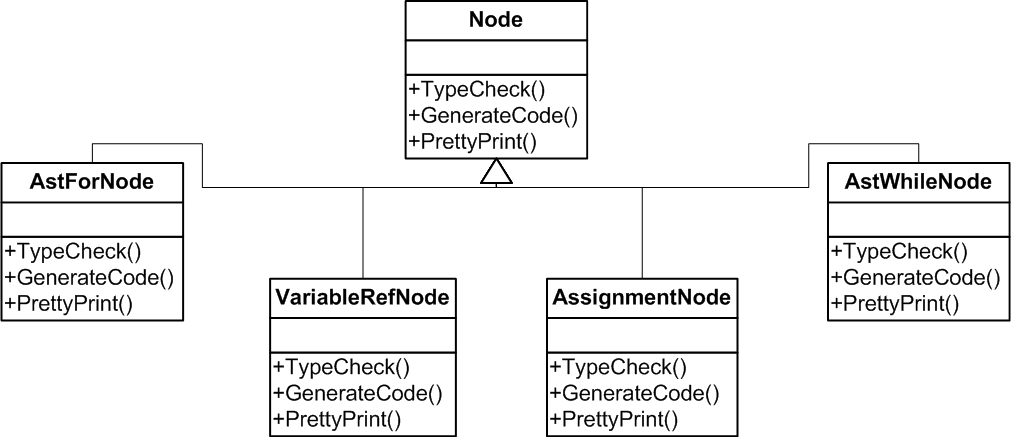
Синтаксический анализатор

Контейнер с ошибками

Сканер

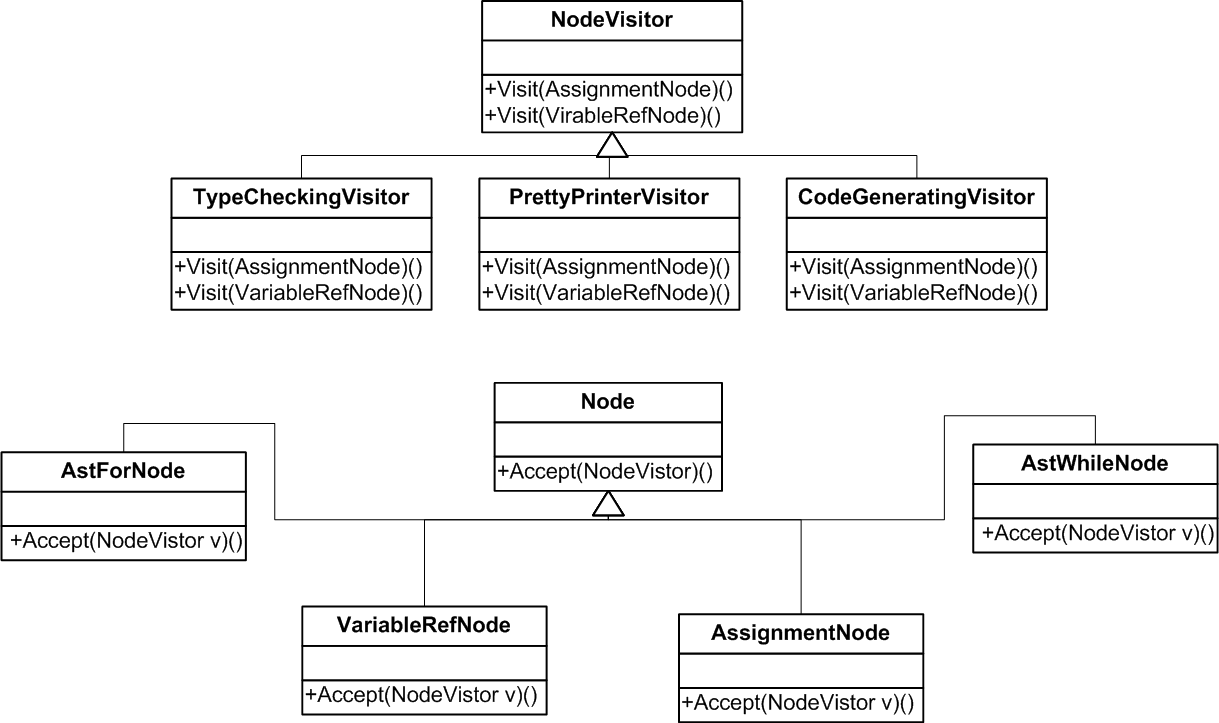
**Общая схема работы**

Компилятор представляет программу в виде абстрактного синтаксического дерева. Над ним он должен выполнять операции “статического семантического анализа”, например, проверять, что все переменные определены. Еще ему нужно генерировать код. Аналогично можно было бы определить операции контроля типов, оптимизации кода, анализа потока выполнения, проверки того, что каждой переменной было присвоено конкретное значение перед первым использованием и так далее. Более того, абстрактные синтаксические деревья могли бы служить для красивой печати программы, реструктурирования кода и вычисления различных метрик программы.

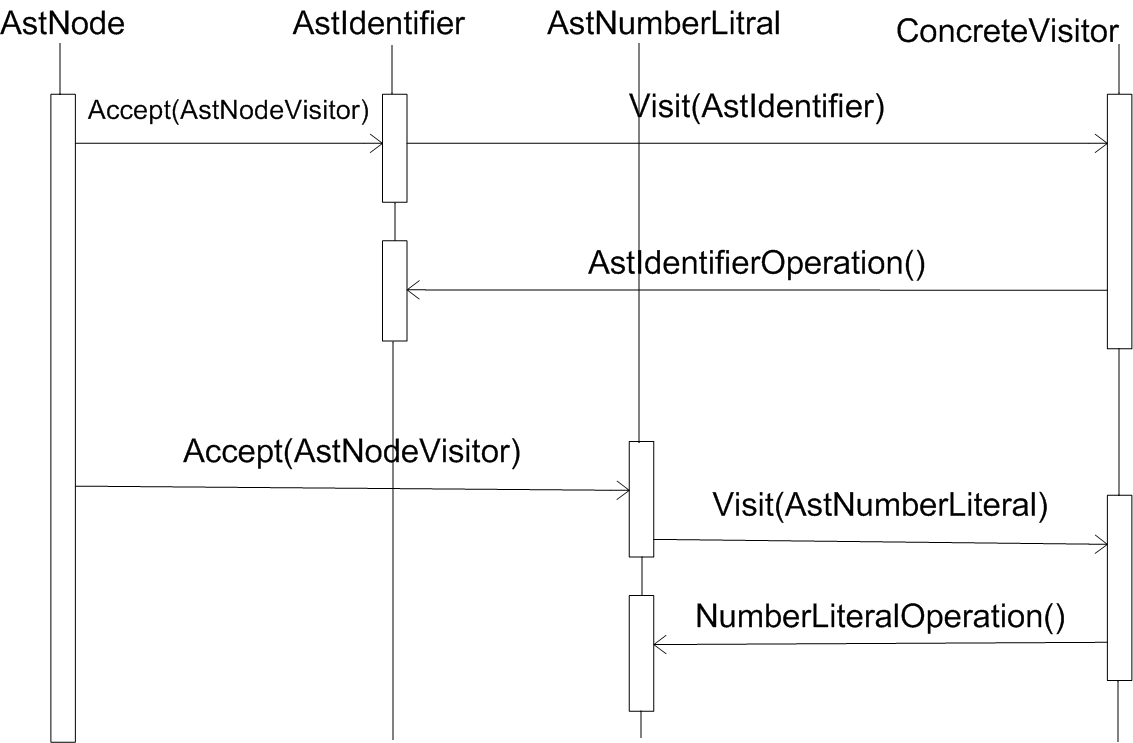


В большинстве таких операций узлы дерева, представляющие операторы присваивания, следует рассматривать иначе, чем узлы, представляющие переменные и арифметические выражения. Поэтому один класс создан для операторов присваивания, другой – для доступа к переменным, третий–для арифметических выражений и т.д. На диаграмме классов представлена часть иерархии AstNode.Возникает задача раскидать все операции по классам различных узлов, необходимо решить эту задачу так чтобы код было легко модифицировать и сопровождать. Вряд ли кто-нибудь разберется в программе, если код, отвечающий за проверку типов, будет перемешан с кодом, реализующим красивую печать или анализ потока выполнения. Кроме того, добавление любой новой операции потребует перекомпиляции всех классов. Оптимальный вариант – наличие возможности добавлять операции по отдельности и отсутствие зависимости классов узлов от применяемых к ним операциям.

И того, и другого можно добиться, если поместить взаимосвязанные операции из каждого классам в отдельный объект, называемый посетителем, и передавать его элементам абстрактного синтаксического дерева по мере обхода. “Принимая” посетителя, элемент посылает ему запрос, в котором содержится, в частности, класс элемента. Кроме того, в запросе присутствует в виде аргумента и сам элемент. Посетителю в данной ситуации предстоит выполнить операцию над элементом, ту самую, которая наверняка находилась бы в классе элемента.



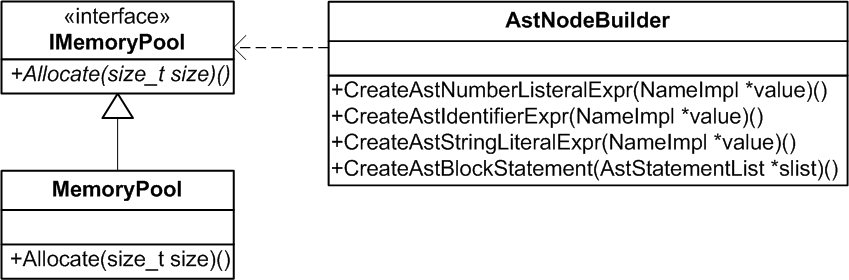
Например, компилятор, который не использует посетителей, мог бы сгенерировать код для “While”вызвав Generate у данного узла, тогда бы каждый узел дерева который может быть использован в “While”должен был реализовать операцию Generateпутем рекурсивного вызова для своих компонентов. Если же генератор генерирует код посредством посетителей то ему достаточно создать объект класса CodeGeneratorи для дерева операцию Accept, передав ей этот объект в качестве аргумента. Каждый узел должен был реализовать Acceptпутем обращения к посетителю: узел,соответствующий оператору While, вызывает операцию посетителя Visit(this). То что раньше было Generateв классеWhile, стало Visit(AstWhileStatement \*) в классе CodeGenerator.Чтобы посетители могли заниматься не только генерацией, нам нужен абстрактный класс посетителя AstVisitor, являющийся родителем для всех посетителей синтаксического дерева.



Приложение которому нужно печать дерево в красивом виде определило бы продклассы AstVisitor, так что нам не пришлось бы добавлять зависящий от приложения код в классы узлов. Паттерн посетитель инкапсулирует операции, выполняемые на каждой фазе компиляции, в классе Visitor, ассоциированном с этой фазой.

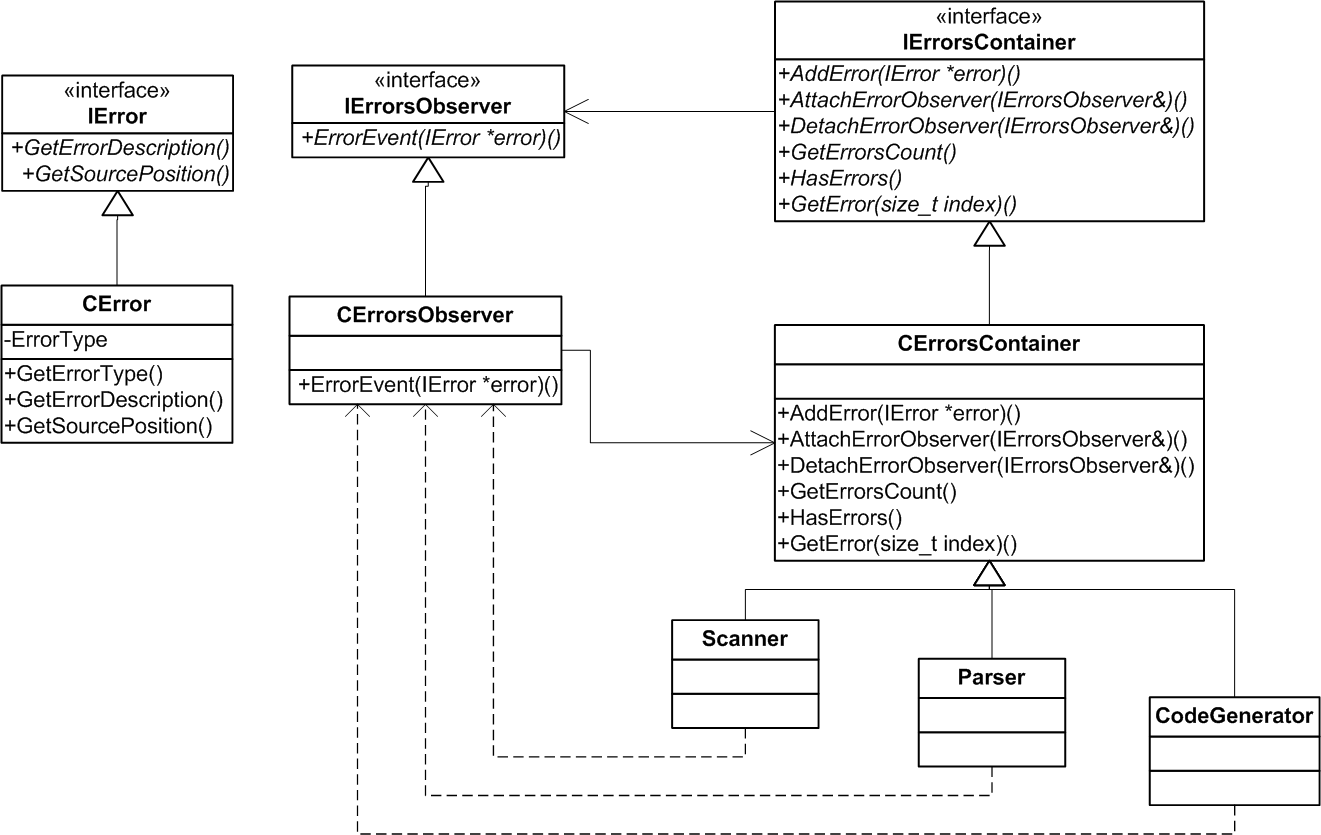
**Создание узлов дерева**

В связи тем что у нас имеется большое количество узлов, необходиммеханизм позволяющих производить их конструирование.



**Обработка ошибок**

Ввиду того что самым верхним классом во всей иерархии классов компилятора является класс Compiler, необходимо было разработать модель обработки ошибок обеспечивающую их передачу с нижших классов в иерархии вверх по цепочке, и так чтобы между классами участвующими в передачи не установилось зависимости.



**Грамматика**

**Грамматикой G[Z]** называется конечное непустое множество правил. Z – это символ, который должен встретиться в левой части хотя бы одного правила. Он называется **начальным символом***.* Все символы, которые встречаются в левых и правых частях правил, образуют **словарьV**. Если из контекста ясно какой символ является начальным символом Z, часто пишется G вместо G[Z].

%token\_prefix Token\_

%token NumberConstant "number constant"

%token StringConstant "string constant"

%token LParen "("

%token RParen ")"

%token LBracket "["

%token RBracket "]"

%token LBrace "{"

%token RBrace "}"

%token Semicolon ";"

%token char "char"

%token break "break"

%token float "float"

%token int "int"

%token double "double"

%token main "main"

%token GT ">"

%token GTE ">="

%token NE "!="

%token Equals "=="

%token LT "<"

%token LTE "<="

%token Id "identifier"

%token Mult "\*"

%token Div "/"

%token Comma ","

%token Add "+"

%token Assign "="

%token Sub "-"

%token if "if"

%token else "else"

%token for "for"

%token do "do"

%token while "while"

%token break "break"

%token read "read"

%token write "write"

%token void "void"

%token CharConstant "char constant"

%start program

program ::= global\_var\_declarationsmain\_function ;

global\_var\_declarations ::= declaration Semicolon global\_var\_declarations ;

global\_var\_declarations ::= ;

variable ::= Id array ;

declaration ::= type variable ;

array ::= LBracket expression RBracketarray\_next ;

array ::= ;

array\_next ::= LBracket expression RBracket ;

array\_next ::= ;

block ::= LBrace statements RBrace ;

block ::= Semicolon ;

main\_function ::= main LParenRParen block ;

type ::= int ;

type ::= char ;

type ::= float ;

type ::= double ;

statements ::= statement statements ;

statements ::= ;

statement ::= if\_statement ;

statement ::= for\_statement ;

statement ::= while\_statement ;

statement ::= do\_while\_statement ;

statement ::= assignment\_statement Semicolon ;

statement ::= read\_statement Semicolon ;

statement ::= write\_statement Semicolon ;

statement ::= break\_statement Semicolon ;

for\_statement ::= for LParenfor\_part Semicolon condition Semicolon for\_partRParen block ;

while\_statement ::= while LParen condition RParen block ;

do\_while\_statement ::= do block while LParen condition RParen ;

read\_statement ::= read LParenread\_argsRParen ;

write\_statement ::= write LParenwrite\_argsRParen ;

if\_statement ::= if LParen condition RParen block else\_block ;

break\_statement ::= break ;

condition ::= expression comparison expression ;

else\_block ::= else block ;

else\_block ::= ;

for\_part ::= assignment\_statement ;

for\_part ::= ;

assignment\_statement ::= assignment Comma assignment\_statement ;

assignment\_statement ::= assignment ;

assignment ::= variable Assign expression ;

read\_args ::= variable ;

read\_args ::= variable Comma read\_args ;

write\_args ::= expression ;

write\_args ::= expression Comma write\_args ;

comparison ::= GT ;

comparison ::= LT ;

comparison ::= GTE ;

comparison ::= LTE ;

comparison ::= NE ;

comparison ::= Equals ;

expression ::= term ;

expression ::= add\_expression ;

expression ::= sub\_expression ;

add\_expression ::= expression Add term ;

sub\_expression ::= expression Sub term ;

term ::= unary\_expression ;

term ::= mult\_expression ;

term ::= div\_expression ;

mult\_expression ::= unary\_expressionMult term ;

div\_expression ::= unary\_expressionDiv term ;

unary\_expression ::= unary\_operator factor ;

unary\_expression ::= factor ;

factor ::= variable ;

factor ::= NumberConstant ;

factor ::= LParen expression RParen ;

unary\_operator ::= Sub ;

factor ::= StringConstant ;

factor ::= CharConstant ;

**Парсер**

Парсер разбирая поток лексем от сканера, формирует ASTдерево с помощью класса AstNodeBuilder, который позволяет создать узлы абстрактного синтаксического дерева. Далее сформированное дерево подается на вход классу CodeGenerator,который занимается генерацией кода.

Таблицы синтаксического разбора для парсера генерируются с помощю утилиты qlalr, которая принимает на вход файл с грамматикой. Для генерации используется LALR(1) грамматика.

LALR – это восходящий алгоритм синтаксического разбора.

Представляет собой расширение алгоритма SLR(1). В ряде случаев работает тогда, когда построение SLR(1) таблицы разбора для данной грамматики невозможно из-за конфликтов сдвиг-приведение или приведение-приведение. Таким образом, класс грамматик, разбираемых по LALR(1) шире, чем класс SLR(1)-грамматик.

Буквы "LA" в сокращении означают "lookahead".Буквы"LR" указывает на то, что для нее существует МП автомат, который начинает просмотр слева направо (Left), распознает правило, когда добирается до самого правого (Rightmost) символа, выводимого из этого правила и может обнаружить любую основу просмотром k-го количества символов, расположенных правее последнего входного символа, выводимого из основы. На практике чаще всего к=1.

Алгоритм собственно разбора (исполнения анализатора по входному потоку) одинаков и у LALR(1), и у SLR(1) - и, шире, у LR(0). Различаются только алгоритмы построения таблицы разбора по грамматике в процессе генерации анализатора.

Таблицы:

* spell

Содержит отладочную информацию о лексемах и правилах.

* lhs

Содержит закодированное значение левой части сооветсующего правила. При свертке это значение преобразуется функциией nt\_action и добавляется на вершину стека состояний.

* rhs

Содержит количество терминальных и нетеринальных символов в левой части соответствующего правила. При свертке i-го правила со стека состояний выталкивется rhs[i] элементов.

* goto\_default ?
* action\_index ?
* action\_info ?
* action\_check ?

staticinlineintnt\_action (intstate, intnt)

{

constint \*constgoto\_index = &action\_index [GOTO\_INDEX\_OFFSET];

constint \*constgoto\_check = &action\_check [GOTO\_CHECK\_OFFSET];

constintyyn = goto\_index [state] + nt;

if (yyn< 0 || goto\_check [yyn] != nt)

returngoto\_default [nt];

constint \*constgoto\_info = &action\_info [GOTO\_INFO\_OFFSET];

returngoto\_info [yyn];

}

Функция nt\_action определяет какое значение необходимо поместить в стек состояний при свертке.

staticinlineintt\_action (intstate, inttoken)

{

constintyyn = action\_index [state] + token;

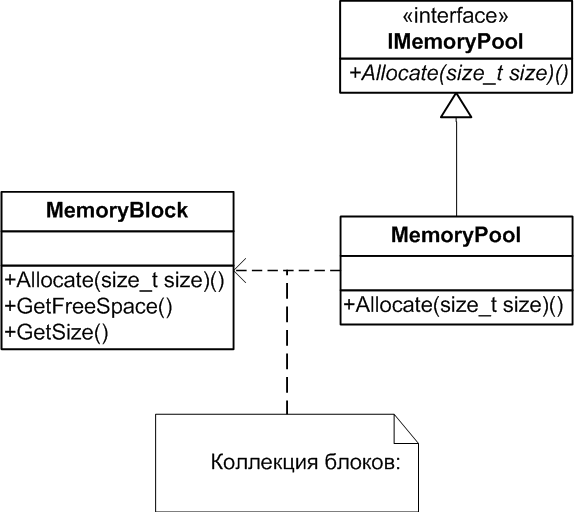
if (yyn< 0 || action\_check [yyn] != token)

return - action\_default [state];

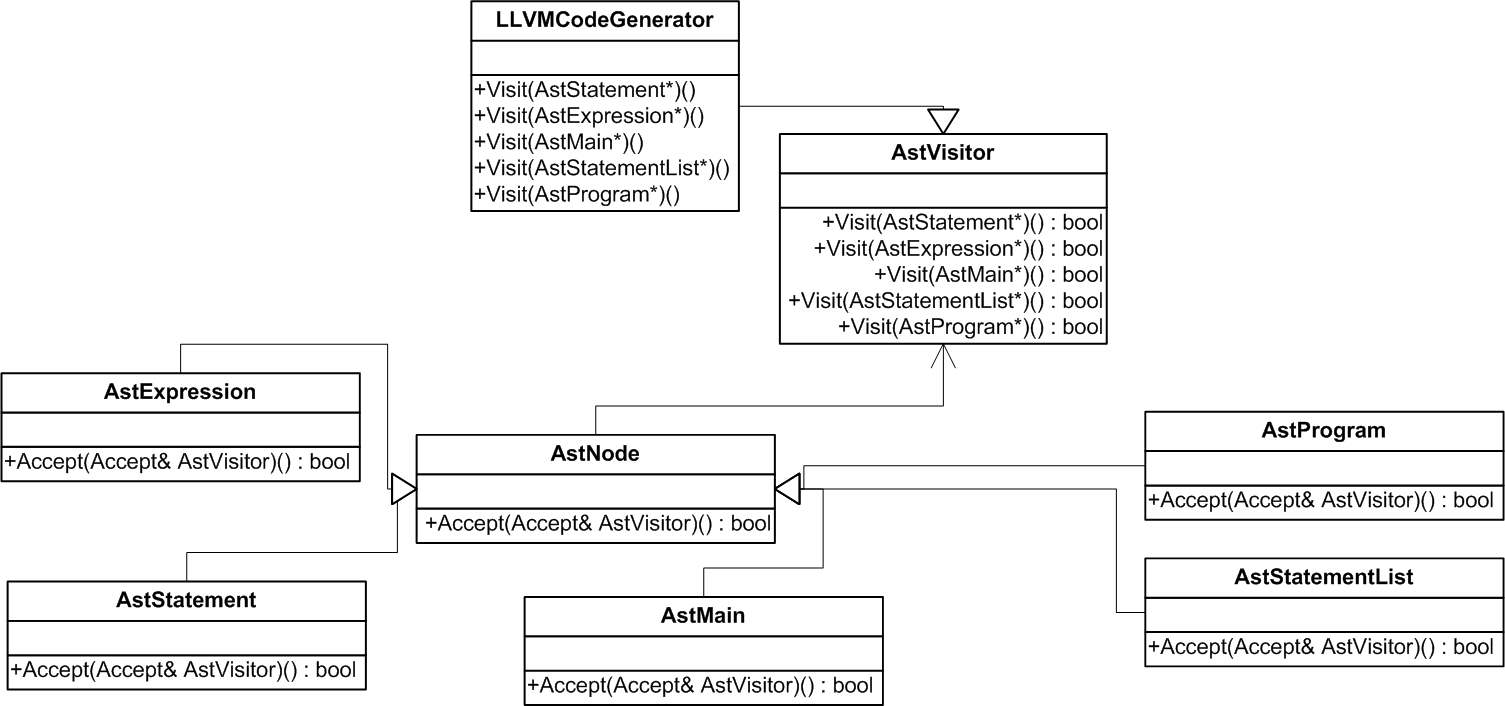
returnaction\_info [yyn];

}

**Управление памятью**



**Генератор**

****

Генератор создает программу в промежуточном представлении кода LLVM.

**LLVM**

LLVM (LowLevelVirtualMachine) — это универсальная система анализа, трансформации и оптимизации программ или, как её называют разработчики, «compilerinfrastucture».

LLVM — не просто очередной академический проект. Его история началась в 2000 году в Университете Иллинойса, а теперь LLVM используют такие гиганты индустрии как Apple и Adobe. В частности, на LLVM основана подсистема OpenGL в MacOS X 10.5, а iPhone SDK использует GCC с бэкэндом на LLVM. Apple является одним из основных спонсоров проекта, а вдохновитель LLVM — Крис Латтнер — теперь работает в Apple.

В основе LLVM лежит промежуточное представление кода (intermediaterepresentation, IR), над которым можно производить трансформации во время компиляции, компоновки (linking) и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM поддерживает генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, IA-64, Alpha.

LLVM написана на C++ и портирована на большинство \*nix-систем и Windows. Система имеет модульную структуру и может расширяться дополнительными алгоритмами трансформации (compilerpasses) и кодогенераторами для новых аппаратных платформ. Пользовательский фронтенд, как правило, линкуется с LLVM и использует C++ API для генерации кода и его преобразований. Однако LLVM включает в себя и standalone утилиты.

LLVM IR одним предложением можно охарактеризовать как типизированный трёхадресный код в SSA-форме. Далее следует его описание.

***В LLVM поддерживаются следующие примитивные типы:***

* Целые числа произвольной разрядности:

i1 ; булево значение — 0 или 1  
i32 ; 32-разрядное целое  
i17   
i256

Генерация машинного кода для типов очень большой разрядности не поддерживается. Например, для x86 вам придётся ограничиться i64, а для x86-64 и других 64-разрядных платформ — 128-битными целыми. Но для промежуточного представления никаких ограничений нет.  
Числа считаются представленными в дополнительном коде. Различий между знаковыми и беззнаковыми целыми на уровне типов не делается: в тех случаях, когда это имеет значение, с ними работают разные инструкции.

* Числа с плавающей точкой: float, double, а также ряд типов, специфичных для конкретной платформы (например, x86\_fp80).
* void — пустое значение.

***Производные типы:***

* Указатели

тип\*  
i32\* ; указатель на 32-битное целое

* Массивы

[число элементов x тип]  
[10 x i32]  
[8 x double]

* Структуры

{ i32, i32, double }

* Вектор — специальный тип для упрощения SIMD-операций. Вектор состоит из 2n значений примитивного типа — целого или с плавающей точкой.

<число элементов x тип >  
< 4 x float>

* Функции

i32 (i32, i32)  
float ({ float, float }, { float, float })

Система типов рекурсивна, поэтому можно использовать многомерные массивы, массивы структур, указатели на структуры и функции, и т. д.

***Операции***

Большинство инструкций в LLVM принимают два аргумента (операнда) и возвращают одно значение (трёхадресный код). Значения определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные — @. Локальные значения также называют регистрами, а LLVM — виртуальной машиной с бесконечным числом регистров. Пример:

%sum = add i32 %n, 5  
%diff = subdouble %a, %b  
%z = add<4 x float> %v1, %v2 ; поэлементное сложение  
%cond = icmpeq %x, %y ; Сравнение целых чисел. Результатимеетi1.  
%success = calli32 @puts(i8\* %str)

Тип операндов всегда указывается явно, и однозначно определяет тип результата. Операнды арифметических инструкций должны иметь одинаковый тип, но сами инструкции «перегружены» для любых числовых типов и векторов.

LLVM поддерживает полный набор арифметических операций, побитовых логических операций и операций сдвига, а также специальные инструкции для работы с векторами.

LLVM IR строго типизирован, поэтому не обойтись без приведений типов, которые явно кодируются специальными инструкциями. Набор из 9 инструкций покрывает всевозможные приведения между различными числовыми типами: целыми и с плавающей точкой, со знаком и без, различной разрядности и пр. Кроме этого есть инструкции преобразования между целыми и указателями, а так же инструкция bitcast, которая приведёт всё ко всему, но за результат вы отвечаете сами.

При компиляции IRкаждый узел дерева выражения кроме листьев (констант и переменных) заменяется промежуточным значением-регистром — результатом инструкции, операндами которой являются дочерние узлы.

; x = (a + b) \* c - d / e   
%tmp1 = add float %a, %b   
%tmp2 = mul float %tmp1, %c   
%tmp3 = fdiv float %d, %e   
%x = sub float %tmp2, %tmp3

***SSA***

SSA (staticsingleassignmentform) — это такая форма промежуточного представления кода, в которой любое значение присваивается только один раз. Таким образом, нельзя написать:

%z = sum i32 %x, %y   
%z = sum i32 %z, 5

Новое значение должно получить новое имя:

%z.1 = sum i32 %z, 5

Код в SSA-форме удобно рассматривать не как линейную последовательность инструкций, а как граф потока управления (controlflowgraph, CFG). Вершины этого графа — так называемые базовые блоки (basicblocks), содержащие последовательность инструкций, заканчивающуюся инструкцией-терминатором, явно передающей управление в другой блок. Базовые блоки в LLVM обозначаются метками, а терминаторами являются следующие инструкции:

* retтипзначение — возврат значения из функции
* br i1 условие, labelметка\_1, labelметка\_2 — условный переход. Например:

define float @max(float %x, float %y)   
{   
    %cond = fcmpogt float %x, %y   
    br i1 %cond, label %IfTrue, label %IfFalse  
IfTrue:   
    ret float %x   
IfFalse:   
    ret float %y   
}

Есть также форма безусловного перехода:

brlabelметка

* switch — обобщение br, позволяет организовать таблицу переходов:

switch i32 %n, label %Default, [i32 0, label %IfZero i32 5, label %IfFive]

* invoke и unwind — используются для организации исключений, в этой статье останавливаться на них мы не будем.
* unreachable — специальная инструкция, показывающая компилятору, что выполнение никогда не достигнет этой точки. Например, эта инструкция может быть вставлена после вызова системной функции завершения процесса.

***Вывод***

Промежуточное представление LLVM достаточно близко соответствует коду на низкоуровневых процедурных языках вроде Си. Транслятор Си на основе LLVM будет достаточно прост и прямолинеен, но при этом сгенерированный им машинный код по производительности сможет тягаться с последними версиями GCC.

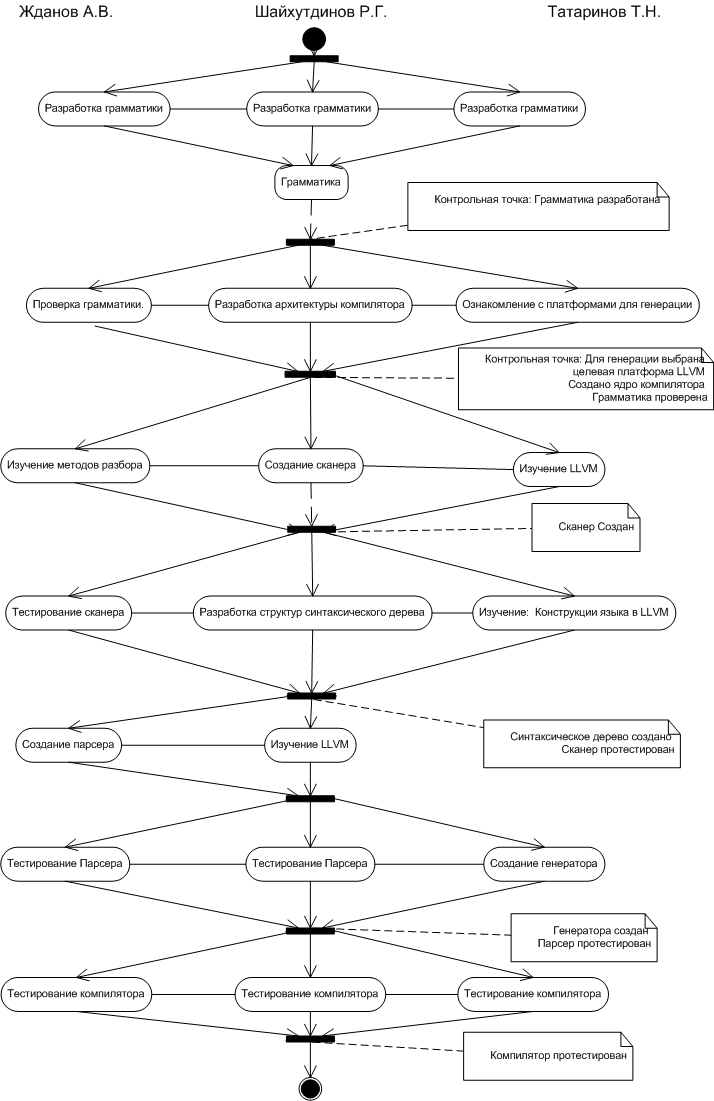
При трансляции высокоуровневых языков — объектно-ориентированных, функциональных, динамических — придётся выполнить гораздо больше промежуточных преобразований, а также написать специализированныйрантайм. Но и в этом случае LLVM снимает с разработчика компилятора проблемы кодогенерации для конкретной платформы, берёт на себя большинство независимых от языка оптимизаций — и делает их качественно. Помимо этого, мы получаем готовую инфраструктуру для JIT-компиляции и возможность link-time оптимизации между различными языками, компилируемыми в LLVM.

LLVM пытается достичь баланса между удобством и гибкостью, не навязывая какую-то конкретную парадигму программирования, не ограничивая систему типов.

**Процесс генерации кода**

Генератор получает на входвыходной поток, в который будет записываться промежуточный LLVM код, и AST-дерево, созданное парсером. Далее начинается разбор дерева. Генератор заходит в каждый узел дерева и добавляет по необходимости элементы в «программный стек» (стек элементов для дальнейшей обработки). По ходу разбора «программный стек» раскручивается, и в выходной поток записывается код – результат генерации. Декларация, к примеру, происходит следующим образом. Генератор заходит в узел AstDeclaration, далее для определения является ли эта переменная массивом и её имени он попадает в AstVariableExpr. Потом генератор следует в узел AstIdentifierExpr, в котором в «программный стек» добавляется элемент с типом Token::Token\_Id и значением имени переменной. Затем если существует узел AstArrayExpr, генератор заходит в него и определяет количество элементов в массиве после прохода узлаAstNumberLiteralExpr. Возвращаемся в функцию обработки AstDeclaration, извлекается верхний элемент из «программного стека», выполняется сериализация в выходной поток промежуточного LLVMкода, вид которого зависит от типа переменной.

Таким образом происходит генерация кода для всех узлов дерева.

**Процесс разработки**

**Список литературы**

1) [Компиляторы: принципы, технологии, инструментарий, Ахо,Лам,Сети,Ульман](javascript:%20location.href%20=%20'/cgi-bin/act.cgi?what=add_kart&id=1988230&gotop='%20+%20escape(location.href);)

2) [Современное проектированиенаC++. Серия "C++In-Depth"](http://www.williamspublishing.com/Books/5-8459-0571-0.html), Вильямс

# 3) Приемы объектно-ориентированного проектирования. Паттерны проектирования,[Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес](http://www.ozon.ru/context/detail/id/2457392/#persons)

4) http://habrahabr.ru/blogs/programming/47878/ - “Обзор LLVM”, Руслан Хайров