**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«Поволжский государственный технологический университет»**

**(ФГБОУ ВПО «ПГТУ» )**

Кафедра ИиСП

**Пояснительная записка**

к курсовой работе

по дисциплине «Теория языков программирования и методов трансляции»

**Выполнили:**

ст. группы ПС-41

Ванясин Н.В.,

Сушенцов А.О.,

Орлов К.А.

**Проверила:**

Нехорошкова Л.Г.

Йошкар-Ола

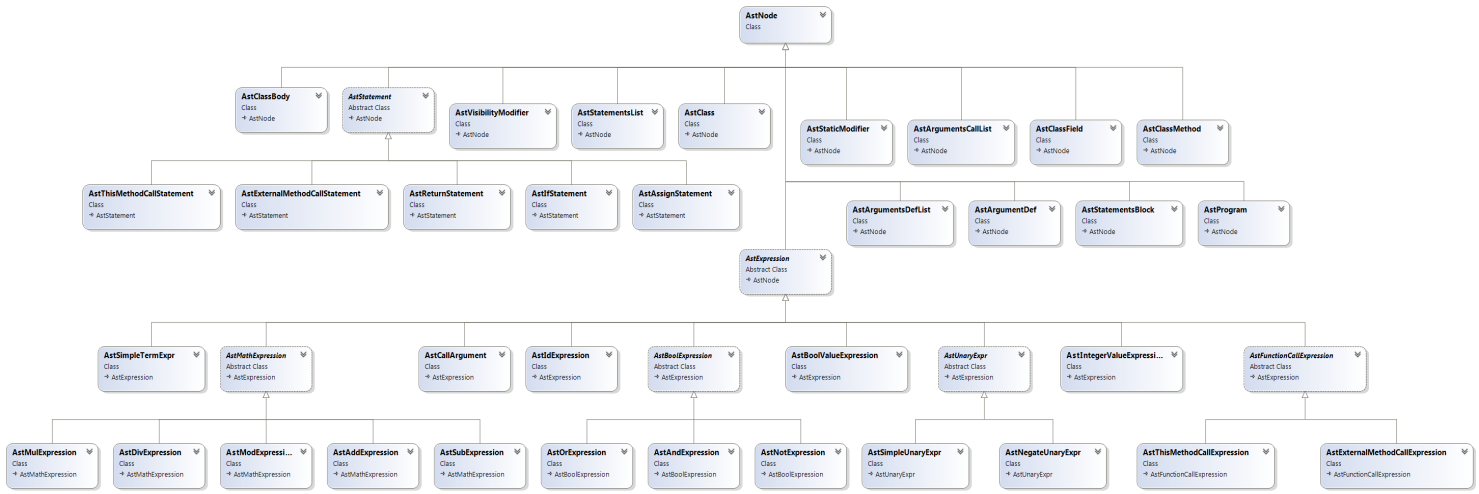
2014

Компилятор – программа, представляющая введенный пользователем текст в виде исполняемого файла или модуля для выполнения на ЭВМ. Трансляция обычно осуществляется при помощи представления исходного текста в виде абстрактного синтаксического дерева. Основные этапы компиляции:

1. Преобразование исходного текста в абстрактное синтаксическое дерево
2. Выполнение статических проверок для того, чтобы не допустить генерацию ошибочного кода
3. Генерация исполняемого кода

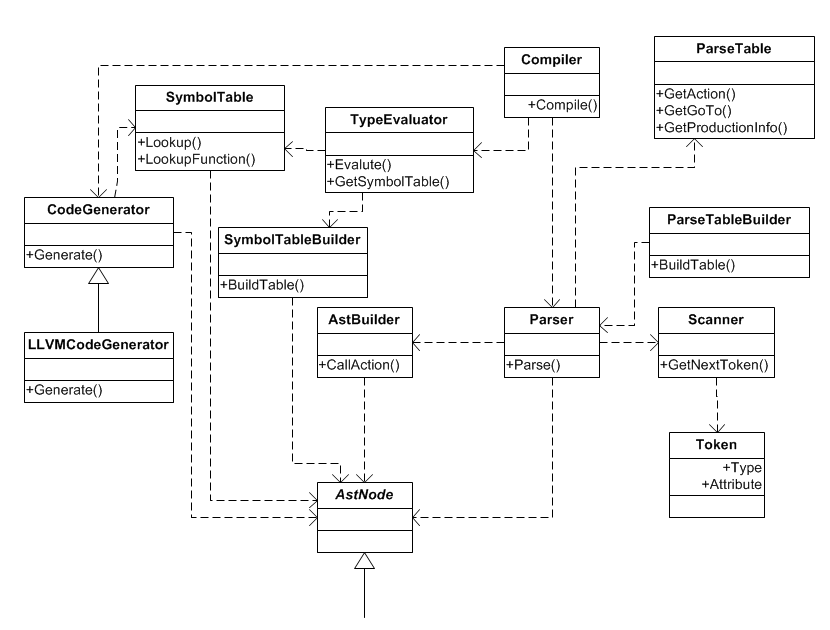
Абстрактные синтаксические деревья могли бы служить для красивого вывода программы, реструктурирования кода и вычисления различных метрик программы.

В большинстве из вышеперечисленных операций узлы дерева, представляющие собой операторы присваивания, рассматривают иначе, чем узлы, с переменными или арифметическими выражениями. Поэтому создается по классу на каждую из вышеописанных сущностей.



На диаграмме классов представлена иерархия AstNode. Необходимо распределить все операции по классам различных узлов так, чтобы код было легко модифицировать и сопровождать. Добавление любой новой операции потребует перекомпиляции всех классов. Оптимальный вариант – возможность добавлять операции по отдельности и отсутствие зависимости классов узлов от применяемых к ним операций.

И того, и другого можно добиться, если поместить взаимосвязанные операции из каждого класса 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.



# Диаграмма взаимодействия основных классов



# Таблица символов

Таблицей символов называется класс для определения типа, размерности и области видимости переменных, констант и функций в программе. Используется для семантических проверок и во время кодогенерации. Таблица организована в виде дерева объектов класса «SymbolsScope» (область видимости). Каждый экзмепляр представляет собой контейнер для символов, объявленных в конкретной области видимости. Таблица может иметь неограниченый уровень вложенности областей видимости.



# 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].

# %namespace PySharpGrammarCheck

# %token

# ID INTEGER\_VALUE BOOL CLASS ELSE FALSE WHILE

# IF INT PRIVATE PUBLIC RETURN STATIC TRUE

# COMMA ASSIGNMENT LEFT\_PAREN RIGHT\_PAREN

# LEFT\_BRACE RIGHT\_BRACE LEFT\_BRACKET

# RIGHT\_BRACKET PLUS MINUS MULTIPLICATION DIV MOD

# NOT AND OR LINE\_END

# BLOCK\_START BLOCK\_END PASS DOT EOF

# LT GT GTE LTE EQUAL NOT\_EQUAL

# %start PS\_PROGRAM

# %%

# PS\_PROGRAM : CLASS\_DEF EOF

# ;

# CLASS\_DEF : CLASS ID CLASS\_BODY

# ;

# CLASS\_BODY : BLOCK\_START CLASS\_DECLARATIONS BLOCK\_END

# ;

# CLASS\_DECLARATIONS : FIELD\_DECLARATION CLASS\_DECLARATIONS

# | METHOD\_DECLARATION CLASS\_DECLARATIONS

# | /\* eps \*/

# ;

# FIELD\_DECLARATION : VISIBILITY\_MODIFIER STATIC\_MODIFIER TYPE\_DEFINITION ID LINE\_END

# ;

# VISIBILITY\_MODIFIER : PUBLIC

# | PRIVATE

# ;

# STATIC\_MODIFIER : STATIC

# | /\* eps \*/

# ;

# TYPE\_DEFINITION : BOOL

# | INT

# | INT LEFT\_BRACKET INTEGER\_VALUE RIGHT\_BRACKET

# ;

# METHOD\_DECLARATION : METHOD\_HEADER BLOCK\_START STATEMENTS\_BLOCK BLOCK\_END

# ;

# METHOD\_HEADER : VISIBILITY\_MODIFIER STATIC\_MODIFIER TYPE\_DEFINITION ID METHOD\_ARGS

# ;

# METHOD\_ARGS : LEFT\_PAREN ARGUMENTS\_DEFINITION RIGHT\_PAREN

# | LEFT\_PAREN RIGHT\_PAREN

# ;

# ARGUMENTS\_DEFINITION : ARGUMENT\_DEFINITION

# | ARGUMENT\_DEFINITION COMMA ARGUMENTS\_DEFINITION

# ;

# ARGUMENT\_DEFINITION : TYPE\_DEFINITION ID

# ;

# STATEMENTS\_BLOCK : STATEMENTS

# | PASS LINE\_END

# ;

# 

# STATEMENTS : STATEMENT STATEMENTS\_S

# ;

# STATEMENTS\_S : STATEMENTS

# | /\* eps \*/

# ;

# STATEMENT : FUNC\_CALL LINE\_END

# | IF\_STATEMENT

# | WHILE\_STATEMENT

# | ASSIGN\_STATEMENT LINE\_END

# | RETURN OR\_TEST LINE\_END

# ;

# ASSIGN\_STATEMENT : ID ASSIGNMENT OR\_TEST

# | ID ARRAY ASSIGNMENT OR\_TEST

# | ID ASSIGNMENT ARRAY\_INITIALIZER

# ;

# ARRAY\_INITIALIZER : LEFT\_BRACE INTEGER\_VALUE\_LIST RIGHT\_BRACE ;

# INTEGER\_VALUE\_LIST : INTEGER\_VALUE

# | INTEGER\_VALUE COMMA INTEGER\_VALUE\_LIST

# ;

# IF\_STATEMENT : IF\_THEN\_STATEMENT

# | IF\_THEN\_STATEMENT ELSE BLOCK\_START STATEMENTS\_BLOCK BLOCK\_END

# ;

# IF\_THEN\_STATEMENT : IF LEFT\_PAREN OR\_TEST RIGHT\_PAREN BLOCK\_START STATEMENTS\_BLOCK BLOCK\_END

# ;

# WHILE\_STATEMENT : WHILE LEFT\_PAREN OR\_TEST RIGHT\_PAREN BLOCK\_START STATEMENTS\_BLOCK BLOCK\_END

# ;

# EXPRESSION : TERM

# | ADD\_EXPRESSION

# | SUB\_EXPRESSION

# ;

# EXPRESSION\_LIST : EXPRESSION

# | EXPRESSION COMMA EXPRESSION\_LIST

# ;

# 

# TERM : UNARY\_EXPRESSION

# | MUL\_EXPRESSION

# | DIV\_EXPRESSION

# | MOD\_EXPRESSION

# ;

# MUL\_EXPRESSION : UNARY\_EXPRESSION MULTIPLICATION TERM

# ;

# DIV\_EXPRESSION : UNARY\_EXPRESSION DIV TERM

# ;

# MOD\_EXPRESSION : UNARY\_EXPRESSION MOD TERM

# ;

# UNARY\_EXPRESSION : MINUS SIMPLE\_TERM

# | SIMPLE\_TERM

# ;

# 

# SIMPLE\_TERM : LEFT\_PAREN OR\_TEST RIGHT\_PAREN

# | ID

# | ID ARRAY

# | INTEGER\_VALUE

# | BOOL\_VALUE

# | FUNC\_CALL

# | STRING\_LITERAL

# ;

# ARRAY : LEFT\_BRACKET EXPRESSION\_LIST RIGHT\_BRACKET

# ;

# 

# ADD\_EXPRESSION : EXPRESSION PLUS TERM

# ;

# SUB\_EXPRESSION : EXPRESSION MINUS TERM

# ;

# FUNC\_CALL : THIS\_METHOD\_CALL

# | EXTERNAL\_METHOD\_CALL

# ;

# THIS\_METHOD\_CALL : ID CALL\_ARGS

# ;

# EXTERNAL\_METHOD\_CALL : ID DOT ID CALL\_ARGS

# ;

# CALL\_ARGS : LEFT\_PAREN CALL\_ARGS\_LIST RIGHT\_PAREN

# | LEFT\_PAREN RIGHT\_PAREN

# ;

# CALL\_ARGS\_LIST : OR\_TEST

# | OR\_TEST COMMA CALL\_ARGS\_LIST

# ;

# BOOL\_VALUE : TRUE

# | FALSE

# ;

# OR\_TEST : AND\_TEST

# | AND\_TEST OR OR\_TEST

# ;

# AND\_TEST : NOT\_TEST

# | NOT\_TEST AND AND\_TEST

# ;

# NOT\_TEST : NOT NOT\_TEST

# | EXPRESSION

# | COMPARISON

# ;

# COMPARISON : SIMPLE\_TERM LT SIMPLE\_TERM

# | SIMPLE\_TERM GT SIMPLE\_TERM

# | SIMPLE\_TERM LTE SIMPLE\_TERM

# | SIMPLE\_TERM GTE SIMPLE\_TERM

# | SIMPLE\_TERM EQUAL SIMPLE\_TERM

# | SIMPLE\_TERM NOT\_EQUAL SIMPLE\_TERM

# ;

# Парсер

Парсер, разбирая поток лексем от сканера, формирует AST с помощью класса AstBuilder, 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 parseTableMaker, которая принимает на вход файл с грамматикой. Для генерации используется SLR(1) грамматика.

S в “SLR” означает “simple”. Буквы "LR" указывают на то, что для грамматики существует МП автомат, который начинает просмотр слева направо, распознает правило, когда добирается до самого правого 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 у LR(0). Различаются только алгоритмы построения таблицы разбора по грамматике в процессе генерации анализатора.

# LLVM

LLVM (Low Level Virtual Machine) — 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 (intermediate representation, 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 (compiler passes) 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.

В LLVM поддерживаются следующие примитивные типы:

* Целые числа произвольной разрядности:

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

* Числа с плавающей точкой:

float, double

* 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 }

* 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.

Представление LLVM достаточно близко соответствует коду на низкоуровневых процедурных языках вроде Си. При трансляции высокоуровневых языков — объектно-ориентированных, функциональных, динамических — придётся выполнить гораздо больше промежуточных преобразований, а также написать специализированный рантайм. 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.

## Реализованный функционал

1. Типы: int, bool, string, массивы int любой размерности
2. Конструкции:

- объявление функций, с аргументами и возвращаемым значением

- присваивание значений

- if , if – else

- while

- арифметические выражения любой сложности, операторы: + - \* div mod

- булева логика: &&, ||, !

- операции сравнения >, <, >=, <=, ==, !=

- операции чтения и записи в элемент массива

- инициилизация одномерного массива списком целых чисел

1. Встроенные функции:

- Печать чисел

- Печать строк

- Печать булевых выражений (выводится 1 либо 0)

- Ввод чисел

1. Семантические проверки

- Проверка типов

- Обнаружение недостижимых участков кода

- Проверка, что все возможные пути выполнения имеют выход из функции

- Проверки корректности вызова функций (количество, порядок и типа аргументов)

- Обнаружение неиспользованных переменных и функций

1. Интерфейс:

- подсветка ключевых слов

- переход к позиции в тексте по клику на ошибку

# Литература

1. Ronald Mak. Writing Compilers and Interpreters: A Modern Software Engineering Approach Using Java®, Third Edition
2. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.