Министерство образования и науки Российской федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования

«Уральский федеральный университет

имени первого Президента России Б.Н. Ельцина»

Институт математики и компьютерных наук
Кафедра вычислительной математики.

РАЗРАБОТКА КОМПИЛЯТОРА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ЯЗЫКА ОБЩЕГО НАЗНАЧЕНИЯ

"Допущен к защите"

_____

"__"_____2012 г.

Курсовая работа
студента гр. КН - 301
Колмогорцева Егора Николаевича

Научный руководитель
Корнев Дмитрий Васильевич
ассистент кафедры
вычислительной математики

Екатеринбург
2012

РЕФЕРАТ

Колмогорцев Е. Н. РАЗРАБОТКА КОМПИЛЯТОРА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ЯЗЫКА ОБЩЕГО НАЗНАЧЕНИЯ, курсовая работа: страниц 23, рисунков 5, приложений 2.

Ключевые слова: КОМПИЛЯТОР, ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, ГРАММАТИКА, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ТРАНСЛЯЦИЯ, КОДОГЕНЕРАЦИЯ, BINSON, FLEX, LLVM.

Цель работы: разработка объектно-ориентированного языка с множественным наследованием и неявным приведением типов и его компилятора.

В работе описывается процесс разработки компилятора, производящего лексический, синтаксический, семантический анализ и генерацию промежуточного платформонезависимого кода.

Содержание

1. Введение

Высокоуровневые языки программирования предоставляют абстракции, позволяющие программисту описывать алгоритмы и структуры данных при помощи более кратких и простых для понимания конструкций. Помимо этого высокоуровневый язык может полностью брать на себя часть задач, ранее решавшихся программистами, таких как управление памятью, проверка согласованности типов и параллельное выполнение кода. Благодаря высокоуровневым языкам возможно портирование программ на другие архитектуры процессоров и операционные системы без значительных изменений в коде.

Таким образом, компилятор высокоуровневого языка берет на себя значительную часть работы, выполняемой программистом на низкоуровневом языке. Важной проблемой является оптимизация кода, позволяющая компилятору генерировать код, достигающий, или даже превосходящий по скорости, код вручную написанный и оптимизированный профессиональными программистами на низкоуровневых языках.

Это делает компилятор сам по себе довольно большой и сложной программой. Компилятор должен принимать потенциально бесконечное множество исходных программ, соответствующих спецификации языка и переводить их в эквивалентную программу на целевом языке. Ошибка в компиляторе может привести к ошибкам во всех скомпилированных им программах, что налагает на программистов, разрабатывающих компилятор, особую ответственность.

Проектирование и разработка компиляторов является одной из тех областей, в который теория оказывает особо сильное влияние на практику. Для решения различных подзадач активно применяются математические модели, так для описания лексических единиц программы используются конечные автоматы и регулярные выражения. Более фундаментальная модель — контекстно-

свободные грамматики [1], применяющиеся для описания синтаксических структур.

Целью данной работы является разработка объектно-ориентированного языка с множественным наследованием и неявным приведением типов и его компилятора.

2. Общая архитектура компиляторов

Компиляция начинается с лексического анализа. Лексический анализатор получает на вход исходный текст программы, который можно считать потоком символов исходного алфавита, и разбивает из на значащие группы, называемые лексемами. Каждой лексеме сопоставляется токен, имеющий атрибут — значение. Затем токен передается синтаксическому анализатору.

Синтаксический анализатор задается контекстно-свободной грамматикой и строит по токенам абстрактное дерево синтаксиса (AST — Abstract Syntax Tree) — древовидное промежуточное представление, эквивалентное изначальному, но более удобное для последующей работы.

Далее следует семантический анализ, проверяющий корректность программы с точки зрения семантики языка (согласование типов, контроль за использованием необъявленных переменных и функций и т. п.).

На данном этапе возможна непосредственная генерация байткода для целевой платформы, но многие компиляторы, поддерживающие компиляцию для нескольких машинных архитектур, сначала генерируют низкоуровневое промежуточное представление программы, над которым выполняются машинно-независимые оптимизации перед трансляцией в целевой машинный язык.

Завершающими стадиями становятся машинно-зависимая оптимизация и генерация кода целевой машины. Схематическое изображение фаз компилятора представлено на Рисунке 1.

Рисунок 1. Фазы компилятора.

3. Лексический анализ

При разработке компилятора для построения лексического анализатора использовался генератор лексических анализаторов Flex [2]. Flex позволяет определить анализатор, указав регулярное выражение для каждого токена. Такое описание называют языком Flex, а сам генератор — компилятором Flex. По файлу с описанием анализатора lex.l компилятор Flex генерирует файл lex.yy.c, представляющий собой программу на языке C, реализующую конечный детерминированный автомат, распознающий лексемы языка и генерирующий по ним токены. Каждый токен в Flex кодируется уникальным числом — идентификатором, позволяющим различать токены между собой. Атрибут токена передается в глобальной переменной yylval.

Программа на языке Flex имеет следующую структуру:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

В разделе объявлений находятся объявления констант, переменных, регулярные определения — определения регулярных выражений, которые могут быть использованы в разделе правил трансляции.

Регулярные определения в файле lex.l:

```
digit [0-9]
char  [a-zA-Z_]
nchar [a-zA-A_0-9]
ws    [ \t]
nl    [\r\n]
```

Правила трансляции состоят из регулярного выражения, задающего лексему и произвольного кода на языке C либо C++, выполняющегося при распознавании лексемы.

Приведем пример, как выделяется в создаваемом языке идентификаторы, целые и десятичные числа:

```
{char}{nchar}*      {
                        yylval->sval = new std::string(yytext);
                        return token::IDENTIFIER;
                    }
{digit}+\.{digit}+  {
                        yylval->dval = std::atof(yytext);
                        return token::DOUBLENUMBER;
                    }
{digit}+            {
                        yylval->ival = std::atoi(yytext);
                        return token::INTNUMBER;
                    }
```

При анализе Lex всегда отдает предпочтение наиболее длинному подходящему правилу. Это, например, позволяет корректно выделять число в десятичной записи как единую лексему, а не как два целых числа и точку между ними.

Полный текст лексического анализатора приведен в Приложении 1.

5. Синтаксический анализ

В каждом языке программирования имеются правила, определяющие корректность конструкций языка. Зачастую эти правила удобно описывать при помощи контекстно-свободных грамматик, так так грамматики обеспечивают точную и простую для понимания и модернизации спецификацию языка программирования. Имеется три основных типа синтаксических анализаторов грамматик: универсальные, нисходящие и восходящие.

Универсальные анализаторы, такие так алгоритмы Кока-Янгера-Касами [3] и Эрли [4], могут быть использованы для разбора любой грамматики, однако эти методы слишком неэффективны для использования в коммерческих компиляторах.

При нисходящем анализе происходит построение дерева разбора для входной строки начиная с корня, новые узлы добавляются в дерево в порядке обхода в грубину. Методы нисходящего анализа (метод рекурсивного спуска, анализ на основе простого и операторного предшествования [5]) популярны при ручной реализации синтаксического анализатора. Они работают с классом грамматик, называемых LL-грамматиками, LL здесь означает что сканирование входного потока происходит слева направо и строится левый вывод грамматики.

Восходящий анализ можно рассматривать как построение дерева разбора начиная с листьев и заканчивая корнем. Наиболее популярным типом восходящего синтаксического анализа является тип «перенос-свертка», применяющийся для анализа LR-грамматик. При анализе «перенос-свертка» для хранения символов грамматики используется стек, а для хранения оставшейся части строки — входной буффер. Изначально стек пуст, а во входном буффере находится вся анализируемая строка. В процессе анализа, входная строка просматривается слева направо. Руководствуясь правилами из построенной по грамматике таблицы, синтаксический анализатор переносит символы в стек, до тех пор, пока не будет готов выполнить свертку по одному из правил грамматики.

Анализ завершается, если в стеке находится стартовый символ, а входной буффер пуст, либо если обнаружена ошибка.

Таким образом, алгоритм исполнения анализатора (прохода по разбираемой строке) достаточно прост и эффективен, основную сложность представляет построение таблицы разбора по грамматике. Для этого существуют алгоритм LR(1) [1] (1 — количество символов предпросмотра, используемых на каждом шаге для принятия решения о дальнейших действиях), его расширение — SLR(1) [1] и расширение SLR(1) — LALR(1) [1]. На практике наиболее часто используются LALR-анализаторы.

Синтаксические анализаторы для LR-грамматик могут создаваться с помощью автоматизированных инструментов — генераторов. В качестве такого генератора будем использовать GNU Bison [6] — генератор LALR парсеров, обычно используемый совместно с Flex. По файлу с описанием грамматики parser.y bison генерирует файл parser.tab.c, представляющий собой программу на языке C, реализующую синтаксический LALR-анализатор, соответствующий определенной грамматике.

Bison — программа имеет структуру, подобную Flex - программе:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

В разделе объявлений находятся объявления токенов грамматики, используемые в lex.l, типы данных атрибутов терминалов и нетерминалов.

```
%union {
     double dval;
     int ival;
     std::string* sval;
     ...
}
```

```
%token <sval> IDENTIFIER "identifier"
%token <dval> DOUBLENUMBER "double number"
%token <ival> INTNUMBER "int number"
```

Так в создаваемом языке определены токены для идентификаторов, целых и вещественных чисел, использовавшиеся в lex.l.

При разработке грамматики иногда удобно допускать неоднозначность для более краткого и естественного описания конструкций. В таких случаях при LR-анализе происходит конфликт перенос/свертка (в одном из состояний парсер не может определить переносить очередной символ в стек или сворачивать по одному из правил) или свертка/свертка (существует более одного возможного правила для свертки). Bison сообщает о количестве и типах конфликтов и может сгенерировать подробный отчет. Для разрешения конфликтов возможно вручную определить действие, которое должен осуществить парсер. Например для арифметических операций при следующем определении возникают конфликты перенос/свертка:

```
expr
     :  expr PLUS expr
     { $$ = new BinaryOpASTNode("+", $1, $3); }
     | expr MINUS expr
     { $$ = new BinaryOpASTNode("-", $1, $3); }
     | expr TIMES expr
     { $$ = new BinaryOpASTNode("*", $1, $3); }
     | expr SLASH expr
     { $$ = new BinaryOpASTNode("/", $1, $3); }
     | INTNUMBER
     { $$ = new IntNumberASTNode($1); }
     | DOUBLENUMBER
     { $$ = new DoubleNumberASTNode($1); }
     ;
```

Их возможно разрешить, определив приоритет операций, для чего в Bison предусмотрены стандартные операторы:

```
%left PLUS MINUS;

%left TIMES SLASH;
```

В bison через $$ определен атрибут нетерминала в левой части правила вывода, а через $n — атрибут n-го терминала или нетерминала в правой части. При генерации bison заменит заменит их на соответствующие имена атрибутов и позиции в стеке. Например первое приведенное правило транслируется так:

```
{ (yyval.ast) = new BinaryOpASTNode("+", (yysemantic_stack_[(3) - (1)].ast), (yysemantic_stack_[(3) - (3)].ast)); }
```

В грамматику языка входят:

- выражения
  - переменные
  - целые и десятичные числа
  - обращение элементу массива по индексу
- операторы
  - арифметические
  - сравнения
  - присваивания
- объявления переменных и массивов
- функции
  - объявления
  - определения
  - вызов
- объявления классов
  - наследование
  - поля
  - методы
  - конструкторы

Полный текст грамматики представлен в Приложении 2.

5. LLVM и кодогенерация

LLVM [7] (Low Level Virtual Machine) - набор инструментов для компиляции, кодогенерации, трансформации и оптимизации. LLVM реализует виртуальную машину с RISC — подобными инструкциями. Ассемблер LLVM является целевым языком для разрабатываемого в ходе работы компилятора. LLVM предоставляет API для C++, Python и OCaml.

В ассемблере LLVM используется трехадресный код и SSA-представление (*Static single assignment form*), представление, в котором каждой переменной (регистру) значение может быть присвоено лишь единожды. Таким образом виртуальную машину LLVM можно представить как машину с бесконечным количеством регистров. Главная цель такого представления — облегчение различных оптимизаций, проводимых компилятором.

Ассемблер LLVM строго типизован, существуют целочисленные и вещественные типы различной разрядности, указатели, структуры и операции приведения типов. Возможно выделение памяти на стеке. Инструкции условного перехода так же выполнены в виде трехадресного кода, в них указывается проверяемое условие и две метки блоков кода.

Кодогенерация в компиляторе осуществляется после полного завершения лексического анализа. Каждый узел абстрактного синтаксического дерева является объектом класса-потомка базового класса ASTNode, в котором определен метод codegen(). Реализация данного метода в каждом из классов-потомков отвечает за генерацию кода, соответствующего данному узлу и всем дочерним узлам. Типичная реализация метода codegen() вызывает codegen() у дочерних узлов, дополняет его своим кодом и возвращает результат. Таким образом при кодогенерации абстрактное дерево синтаксиса обходится в глубину и происходит трансляция в следующее промежуточное представление - ассемблер LLVM. В корне дерева получившийся код выводится и работа компилятора завершается.

Приведем пример генерации кода для цикла for:

```
BasicBlock* ForASTNode::codegen() const
{
    BasicBlock* for_cond = BasicBlock::Create(
        getGlobalContext(), "for_cond", currentFunction
    );
    BasicBlock* for_end = BasicBlock::Create(
        getGlobalContext(), "for_end", currentFunction
    );
    begin_->codegen();
    builder->CreateBr(for_cond);

    BasicBlock* body = body_->codegen();
    step_->codegen();
    builder->CreateBr(for_cond);

    builder->SetInsertPoint(for_cond);
    builder->CreateCondBr(
        cond_->codegen(),
        body,
        for_end
    );
    builder->SetInsertPoint(for_end);
    return for_end;
}
```

В данном коде ассемблер, сгенерированный блоком инициализации цикла for добавляется к текущему блоку и создаются новые блоки для условия цикла, тела цикла и завершения. В ассемблере LLVM блоки являются обособленными конструкциями, каждая из которых должна явно передавать управление следующему блоку, или заканчиваться инструкцией ret, возвращающей управление из функции. Рассмотрим схему таких блоков для цикла for:

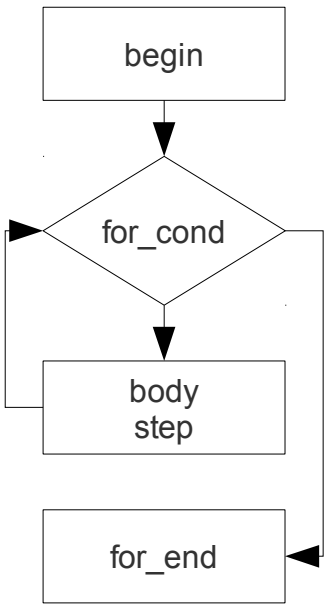

Рисунок 2.

Отличие данных блоков от меток, используемых в ассемблере x86, состоит в том, что все переходы (стрелки на схеме) должны быть явно описаны. Так мы не можем опустить безусловный переход от начального блока к условию цикла.

Приведем программу на разработанном языке и сгенерированный по ней ассемблер LLVM:

```
def main() of int
{
    sum of int;
    sum = 0;
    i of int;

    for(i = 0; i < 10; i = i + 1) {
        sum = sum + i;
    }

    printi(sum);
    return 0;
}
```

Сгенерированный ассемблер:

```
define i64 @main() {
main_entry:
  br label %bb

bb:                                               ; preds = %main_entry
  %sum_alloca = alloca i64
  store i64 0, i64* %sum_alloca
  %i_alloca = alloca i64
  store i64 0, i64* %i_alloca
  br label %for_cond

for_cond:                                         ; preds = %bb1, %bb
  %0 = load i64* %i_alloca
  %1 = icmp slt i64 %0, 10
  br i1 %1, label %bb1, label %for_end

bb1:                                              ; preds = %for_cond
  %4 = load i64* %sum_alloca
  %5 = load i64* %i_alloca
  %6 = add i64 %4, %5
  store i64 %6, i64* %sum_alloca
  %7 = load i64* %i_alloca
  %8 = add i64 %7, 1
  store i64 %8, i64* %i_alloca
  br label %for_cond

for_end:                                          ; preds = %for_cond
  %2 = load i64* %sum_alloca
  %3 = call i64 @printi(i64 %2)
  ret i64 0
}
```

6. Детали реализации компилятора

6.1 Переменные и операторы

В языке введена строгая типизация, каждая переменная должна быть явно объявлена с указанием типа. Память для переменных выделяется на стеке. При кодогенерации переменных стоит различать так называемые lvalue и rvalue обращения. Lvalue обращение происходит при присваивании переменной нового значения, rvalue - в остальных случаях. На уровне LLVM различие состоит в том, что в случае lvalue должен возвращаться указатель на стек, а при rvalue обращении — значение переменной, загруженное из стека в регистр. При кодогенерации операторов при необходимости производится неявное приведение типов.

6.2 Массивы

Массивы выделяются в динамической памяти, которая должна быть явно освобождена. Фактически массив представляет собой указатель на выделенный участок памяти. Благодаря этому массивы можно передавать в функции и возвращать из функций

6.3 Ввод/вывод

Для ввода/вывода предусмотрены обертки над функциями libc, которые компилируются в байткод LLVM, представляющий собой отдельную библиотеку. Данная библиотека линкуется компилятором к генерируемому коду. Таким образом можно менять реализацию данной библиотеки без изменений в компиляторе.

6.4 ООП

объект представляет собой структуру, состоящую из полей, объявленных в классе. Обращения к полям объекта компилятор транслирует в обращения к соответствующим полям структуры. Методы представляют собой функции, которым первым аргументом передается указатель на структуру, соответствующую

классу, в котором объявлен метод. Виртуальные методы заносятся в таблицу виртуальных методов класса — массив, содержащий указатели на функции.

6.4.1. Множественное наследование.

При наследовании объект класса-потомка содержит структуры классов-родителей в порядке, указанном при наследовании и собственные поля в конце. При приведении типа к одному из родительских, просто берется соответствующая часть структуры. При переопределении виртуальных методов, в таблице класса-наследника меняются соответствующие им указатели. При наследовании класса от нескольких классов с виртуальными методами, изменения вносятся в таблицы для каждого из них. Приведем пример программы на разработанном языке и схематичное изображение объектов соответствующих классов (стрелка указывает на таблицу виртуальных методов для класса):

```
class A {
    a of int;
    virtual def a() of int {
        printi(a);
        return 0;
    }
}
```

Рисунок 3.

Объект класса A

```
class B {
    b of int;
    virtual def b() of int {
        printi(2);
        return 0;
    }
}
```

Рисунок 4.

Объект класса B

```
class C : A, B {
    c of int;
    virtual def b() of int {
        printi(3);
        return 0;
    }
    virtual def c() of int {
        printi(4);
        return 0;
    }
}
```

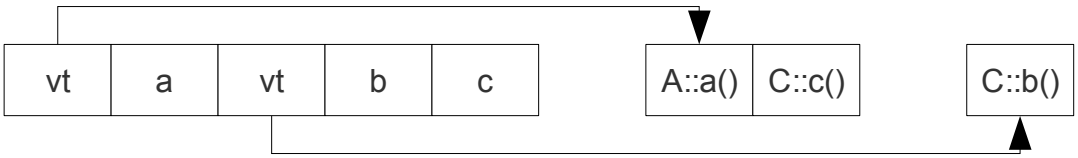

Рисунок 5.

Объект класса C.

6.4.2. Неявное приведение типов .

Компилятор поддерживает систему типов, возможно получение объекта, отвечающего за описание типа по строковому имени. Пользовательские типы (классы) добавляются во время синтаксического анализа, частично совмещенного с семантическим, до кодогенерации.

Приведем структуру, описывающую тип:

```
struct LangType
{
    std::string type_name;
    Type* llvm_type;
```

```
    PointerType* llvm_ptr_type;
    size_t size;
    std::map<std::string, std::string> methods;
    std::map<std::string, Value*> fields;
    std::map<LangType*, vtable_t> parent_vtables;
    vtable_t vtable;
    std::map<LangType*, size_t> castOffsets;
    std::vector<LangType*> parents;

    LangType* classOfField(const std::string& field) const;
    LangType* classOfMethod(const std::string& name) const;
    Value* castTo(Value* object, LangType* target_type) const;
    Value* downcast(Value* object, LangType* object_type) const;
};
```

Методы classOfField и classOfMethod отвечают за обход в ширину родительских классов и поиск класса, в котором определено унаследованное поле или метод. Это необходимо для приведения типа объекта, вызывающего родительский метод или обращающегося к родительскому полю.

Для приведения типов хранится хеш castOffsets, содержащий смещения указателя на начало объекта для каждого родительского класса (в том числе для нескольких уровней наследования). Метод castTo осуществляет так называемый upcasting — приведение объекта типа-наследника к объекту типа-родителя путем поиска смещения в castOffsets, при необходимости генерации инструкций для смещения указателя и смены типа объекта. Фрагмент кода, ответственный за данные действия:

```
if(offset == 0) {
    return builder->CreateBitCast(
        object, target_type->llvm_type
    );
}
```

```
    Value* i8ptr = builder->CreateBitCast(
            object,
            Type::getInt8PtrTy(getGlobalContext())
    );

    return builder->CreateBitCast(
        builder->CreateGEP(
            i8ptr, ConstantInt::get(
                getGlobalContext(), APInt(64, offset)
            )
        ), target_type->llvm_type
    );
```

Такое приведение типов происходит неявно, при присваивании объекту родительского типа объекта типа-наследника. Другой тип приведения типов, downcasting — приведение объекта родительского типа к объекту типа-наследника будет рассмотрен ниже.

6.4.3. Вызовы виртуальных методов .

Во время исполнения сгенерированной программы, при создании объекта класса, имеющего виртуальную таблицу, вызывается специальная функция, устанавливающая в объекте указатели на соответствующие виртуальные таблицы. Ассемблерный код данной функции для объектов класса C из раздела 6.4.1:

```
define void @__C_vtables_constructor(%C* %__this_C) {
__C_vtables_constructor_entry:
  %0 = bitcast %C* %__this_C to i8***
   store i8** getelementptr inbounds ([1 x i8*]* @__C_C_vtable, i32 0, i32 0), i8*** %0
  %1 = bitcast %C* %__this_C to i8*
  %2 = getelementptr i8* %1, i64 9
```

```
  %3 = bitcast i8* %2 to %B*
  %4 = bitcast %B* %3 to i8***
   store i8** getelementptr inbounds ([1 x i8*]* @__C_B_vtable,
i32 0, i32 0), i8*** %4
  ret void
}
```

При вызове виртуального метода, происходит приведение типа объекта к классу, в котором впервые определен данный метод и вызов функции. Во всех переопределенных виртуальных методах в начале метода вставляется код, выполняющий приведение объекта класса, в котором данный метод был обьявлен впервые, к текущему классу (downcasting). Данный код не может вызвать ошибку, так как объект, фактически переданный такому методу должен быть объектом текущего класса, приведенным к родительскому. Такие преобразования необходимы из-за того, что мы не можем на стадии компиляции определить фактический тип объекта, вызывающего виртуальный метод. Дополним пример из раздела 6.4.1 функцией main().

```
def main() of int
{
    c of C;
    b of B;
    b = c;
    b->b();
    return 0;
}
```

При присваивании объекта типа C объекту типа B, произойдет приведение его типа к B. При вызове b->b() вызовется метод b() из класса C, находящийся в виртуальной таблице объекта b. Однако уже в самом методе C::b() будет осуществлено обратное приведение к C.

Список литературы

[1] Компиляторы. Принципы, технологии, инструменты / авт. Альфред Ахо, Рави Сети, Джеффри Ульман

[2] Flex [В Интернете] // flex: The Fast Lexical Analyzer - http://flex.sourceforge.net

[3] An efficient recognition and syntax-analysis algorithm for context-free languages. / авт. T. Kasami (1965). // Scientific report AFCRL-65-758, Air Force Cambridge Research Lab,Bedford, MA.

[4] An efficient context-free parsing algorithm / авт. J. Earley // *Communications of the Association for Computing Machinery*, 13:2:94-102, 1970.

[5] Языки, грамматики, распознаватели / авт. Замятин А.П., Шур А.М.

[6] GNU Bison [В Интернете] // Bison - GNU parser generator - http://www.gnu.org/software/bison

[7] LLVM [В Интернете] // The LLVM Compiler Infrastructure - http://llvm.org

Приложение 1. Лексический анализатор.

```
%{
#include <iostream>
#include "parser.hpp"
#include "driver.hpp"

typedef yy::Parser::token token;

#define yyterminate() return token::END
%}

%option noyywrap
%option debug

digit [0-9]
char  [a-zA-Z_]
nchar [a-zA-Z_0-9]
ws    [ \t]
nl    [\r\n]

%%

"->"                return token::ARROW;
"<"                 return token::LT;
"<="                return token::LE;
">"                 return token::GT;
">="                return token::GE;
"=="                return token::EQ;
"!="                return token::NE;
"!"                 return token::NOT;
"+"                 return token::PLUS;
"-"                 return token::MINUS;
"*"                 return token::TIMES;
"/"                 return token::SLASH;
```

```
"("                     return token::LPAREN;
")"                     return token::RPAREN;
"="                     return token::ASSIGN;
","                     return token::COMMA;
";"                     return token::SEMICOLON;
":"                     return token::COLON;
"if"                    return token::IF;
"else"                  return token::ELSE;
"while"                 return token::WHILE;
"for"                   return token::FOR;
"def"                   return token::DEF;
"of"                    return token::OF;
"extern"                return token::EXTERN;
"class"                 return token::CLASS;
"return"                return token::RETURN;
"delete"                return token::DELETE;
"virtual"               return token::VIRT;
"{"                     return token::BLOCK_BEGINS;
"}"                     return token::BLOCK_ENDS;
"["                     return token::LBRACKET;
"]"                     return token::RBRACKET;

{char}{nchar}*          {
                           yylval->sval = new std::string(yytext);
                           return token::IDENTIFIER;
                        }
-?{digit}+\.{digit}+    {
                           yylval->dval = std::atof(yytext);
                           return token::DOUBLENUMBER;
                        }
-?{digit}+              {
                           yylval->ival = std::atoi(yytext);
                           return token::INTNUMBER;
                        }
({nl}|{ws})+            /* ignore */
.                       { driver.error("invalid character"); }
```

```
%%

void Driver::scan_begin()
{
    yy_flex_debug = trace_scanning;
    if (filename == "-")
        yyin = stdin;
    else if (!(yyin = fopen (filename.c_str (), "r")))
    {
        error(std::string ("cannot open ") + filename);
        exit(1);
    }
}

void Driver::scan_end()
{
    fclose(yyin);
}
```

Приложение 2. Синтаксический анализатор (грамматика).

```
%skeleton "lalr1.cc"
%defines
%define parser_class_name "Parser"

%code requires {
#include <string>
#include "ast.hpp"
class Driver;
}

%parse-param { Driver& driver }
%lex-param   { Driver& driver }

%debug
%error-verbose

%union {
    double dval;
    int ival;
    std::string* sval;
    ArgumentsContainer* params;
    ArgumentASTNode* arg;
    ASTNodeContainer* nodes;
    std::vector<std::string>* strs;
    ASTNode* ast;
    NumberASTNode* num;
    AbstractVariableASTNode* var;
    VariableDeclASTNode* decl;
    BinaryOpASTNode* binOp;
    ClassDefASTNode* classDef;
    ClassBodyASTNode* classBody;
    FuncCallASTNode* call;
    ReturnASTNode* ret;
```

```
    FuncDefASTNode* func;
    ConstructorDefASTNode* constrDef;
    BlockASTNode* block;
    UnitASTNode* unit;
    ForASTNode*  for_node;
    IfASTNode*  if_node;
    WhileASTNode*  while_node;
}

%code {
#include "driver.hpp"
}

%token END 0 "end of file"
%token LT
%token LE
%token GE
%token GT
%token EQ
%token NE
%token NOT
%token PLUS
%token MINUS
%token TIMES
%token SLASH
%token LPAREN
%token RPAREN
%token BLOCK_BEGINS
%token BLOCK_ENDS
%token ARROW
%token ASSIGN
%token COMMA
%token CLASS
%token SEMICOLON
%token COLON
%token DEF
```

```
%token EXTERN
%token RETURN
%token VIRT
%token DELETE
%token WHILE
%token IF
%token ELSE
%token OF
%token FOR
%token LBRACKET
%token RBRACKET
%token <sval> IDENTIFIER "identifier"
%token <dval> DOUBLENUMBER "double number"
%token <ival> INTNUMBER "int number"
%type <var> variable
%type <ret> return;
%type <ast> expr
%type <ast> var_delete
%type <decl> var_decl
%type <call> func_call
%type <for_node> for_statement
%type <while_node> while_statement
%type <if_node> if_statement
%type <func> func_prototype
%type <func> func_definition
%type <block> code_block
%type <block> statements
%type <classDef> class_definition
%type <constrDef> constructor_def
%type <classBody> class_body
%type <nodes>  expressions
%type <arg> param
%type <params> prototype_params
%type <unit> unit
%type <unit> program
%type <strs> parents
```

```
%type <strs> parents_
%%

%right ASSIGN;
%left EQ NE;
%left LT LE GT GE;
%left PLUS MINUS;
%left TIMES SLASH;
%right NOT;
%left ARROW;
%left LBRACKET;

%start program;

program
       : unit { $$ = $1; driver.result = $$; driver.result-
>codegen(); }
    ;

unit
    : /* Nothing */         { $$ = new UnitASTNode(); }
    | unit class_definition { $$ = $1; $$->append($2); }
    | unit func_definition  { $$ = $1; $$->append($2); }
    ;

class_definition
         : CLASS IDENTIFIER parents BLOCK_BEGINS class_body
BLOCK_ENDS
      { $$ = new ClassDefASTNode(*$2, $3, $5); }
    ;

parents
    : /* Nothing */  { $$ = new std::vector<std::string>(); }
    | COLON parents_ { $$ = $2; }
    ;

parents_
```

```
    : IDENTIFIER
    { $$ = new std::vector<std::string>(); $$->push_back(*$1);}
    | parents_ COMMA IDENTIFIER
    { $$ = $1; $$->push_back(*$3); }
    ;

class_body
    : /* Nothing */
      { $$ = new ClassBodyASTNode(); }
    | class_body var_decl SEMICOLON
      { $$ = $1; $$->addField($2); }
    | class_body constructor_def
      { $$ = $1; $$->setConstructor($2); }
    | class_body func_definition
      { $$ = $1; $$->addMethod($2); }
    | class_body VIRT func_definition
      { $$ = $1; $$->addMethod($3, true); }
    ;

code_block
    : BLOCK_BEGINS statements BLOCK_ENDS { $$ = $2; }
    ;

statements
     : /* Nothing */                                  { $$ = new
BlockASTNode(); }
     | statements var_decl SEMICOLON                  { $$ = $1; $$-
>append($2); }
     | statements var_delete SEMICOLON                { $$ = $1; $$-
>append($2); }
     | statements expr SEMICOLON                      { $$ = $1; $$-
>append($2); }
     | statements return SEMICOLON                    { $$ = $1; $$-
>append($2); }
     | statements for_statement                       { $$ = $1; $$-
>append($2); }
     | statements if_statement                        { $$ = $1; $$-
>append($2); }
```

```
      | statements while_statement                { $$ = $1; $$->append($2); }
    ;

for_statement
        : FOR LPAREN expressions SEMICOLON expr SEMICOLON expressions RPAREN code_block
      { $$ = new ForASTNode($3, $5, $7, $9); }
    ;

if_statement
    : IF LPAREN expr RPAREN code_block ELSE code_block
      { $$ = new IfASTNode($3, $5, $7); }
    | IF LPAREN expr RPAREN code_block
      { $$ = new IfASTNode($3, $5); }
    ;

while_statement
    : WHILE LPAREN expr RPAREN code_block
      { $$ = new WhileASTNode($3, $5); }
    ;

var_decl
    : IDENTIFIER OF IDENTIFIER LBRACKET INTNUMBER RBRACKET
    { $$ = new VariableDeclASTNode(*$1, *$3, $5); }
    | IDENTIFIER OF IDENTIFIER
    { $$ = new VariableDeclASTNode(*$1, *$3); }
    | IDENTIFIER OF IDENTIFIER LPAREN expressions RPAREN
    { $$ = new ConstructorASTNode(*$1, *$3, $5); }
    ;

var_delete
    : DELETE IDENTIFIER { $$ = new VariableDeleteASTNode(*$2); }

expr
    : NOT expr
    { $$ = new UnaryOpASTNode("!", $2); }
```

```
    | expr LT expr
    { $$ = new BinaryOpASTNode("<", $1, $3); }
    | expr LE expr
    { $$ = new BinaryOpASTNode("<=", $1, $3); }
    | expr GT expr
    { $$ = new BinaryOpASTNode(">", $1, $3); }
    | expr GE expr
    { $$ = new BinaryOpASTNode(">=", $1, $3); }
    | expr EQ expr
    { $$ = new BinaryOpASTNode("==", $1, $3); }
    | expr NE expr
    { $$ = new BinaryOpASTNode("!=", $1, $3); }
    | expr PLUS expr
    { $$ = new BinaryOpASTNode("+", $1, $3); }
    | expr MINUS expr
    { $$ = new BinaryOpASTNode("-", $1, $3); }
    | expr TIMES expr
    { $$ = new BinaryOpASTNode("*", $1, $3); }
    | expr SLASH expr
    { $$ = new BinaryOpASTNode("/", $1, $3); }
    | expr ASSIGN expr
    { $$ = new AssignmentASTNode("=", $1, $3); }
    | INTNUMBER
    { $$ = new IntNumberASTNode($1); }
    | DOUBLENUMBER
    { $$ = new DoubleNumberASTNode($1); }
    | LPAREN expr RPAREN
    { $$ = $2; }
    | variable ARROW IDENTIFIER LPAREN expressions RPAREN
    { $$ = new MethodCallASTNode($1, *$3, $5); }
    | func_call
    { $$ = $1; }
    | variable
    { $$ = $1; }
    ;
```

```
return
    : RETURN expr { $$ = new ReturnASTNode($2); }
    ;

variable
    : variable LBRACKET expr RBRACKET
    { $$ = new ArraySubscriptingASTNode($1, $3); }
    | variable ARROW IDENTIFIER
    { $$ = new FieldASTNode($1, *$3) ; }
    | IDENTIFIER
    { $$ = new VariableASTNode(*$1); }
    ;

func_call
        : IDENTIFIER LPAREN expressions RPAREN { $$ = new
FuncCallASTNode(*$1, $3); }
    ;

expressions
    : /* */
    { $$ = new ASTNodeContainer(); }
    | expr
    { $$ = new ASTNodeContainer(); $$->append($1); }
    | expressions COMMA expr
    { $$ = $1; $$->append($3); }
    ;

func_prototype
    : IDENTIFIER LPAREN RPAREN OF IDENTIFIER
      { $$ = new FuncDefASTNode(*$1, *$5); }
     | IDENTIFIER LPAREN RPAREN OF IDENTIFIER LBRACKET INTNUMBER
RBRACKET
      { $$ = new FuncDefASTNode(*$1, *$5, true); }
    | IDENTIFIER LPAREN prototype_params RPAREN OF IDENTIFIER
      { $$ = new FuncDefASTNode(*$1, *$6, false, $3); }
      | IDENTIFIER LPAREN prototype_params RPAREN OF IDENTIFIER
LBRACKET INTNUMBER RBRACKET
```

```
      { $$ = new FuncDefASTNode(*$1, *$6, true, $3); }
    ;

prototype_params
    : param
      { $$ = new ArgumentsContainer(); $$->append($1); }
    | prototype_params COMMA param
      { $$ = $1; $$->append($3); }
    ;

param
    : IDENTIFIER OF IDENTIFIER LBRACKET INTNUMBER RBRACKET
    { $$ = new ArgumentASTNode(*$1, *$3, $5); }
    | IDENTIFIER OF IDENTIFIER
    { $$ = new ArgumentASTNode(*$1, *$3); }
    ;

func_definition
        : DEF func_prototype code_block      { $$ = $2; $$-
>addBody($3); }
        | EXTERN func_prototype SEMICOLON { $$ = $2; $$-
>makeExtern(); }
    ;

constructor_def
    : DEF IDENTIFIER LPAREN RPAREN code_block
      { $$ = new ConstructorDefASTNode(*$2, $5); }
    | DEF IDENTIFIER LPAREN prototype_params RPAREN code_block
      { $$ = new ConstructorDefASTNode(*$2, $6, $4); }
    ;
%%
void yy::Parser::error(const yy::location& l, const std::string&
m)
{
    driver.error(m);
}
```