Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка компилятора языка TinyC»

Студент **ИУ7-21М** _____ **Дубовицкая О.Н.**
(Подпись, дата)

Руководитель _____ **Ступников А.А.**
(Подпись, дата)

2024 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ-7
(Индекс)
_____ И. В. Рудаков
(И.О.Фамилия)
« _____ » _____ 2024 г.

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине **«Конструирование компиляторов»**

Студент группы **ИУ7-21М**

Дубовицкая Ольга Николаевна

Тема курсового проекта

«Разработка компилятора языка TinyC»

Направленность КП

учебная

Источник тематики

КП кафедры

График выполнения проекта: 25% к 6 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

Техническое задание:

Описать грамматику языка TinyC, выделить её основные ключевые составляющие. Разработать компилятор языка TinyC на языке Python, использующий библиотеку ANTLR для синтаксического анализа входного потока данных и построения AST-дерева. Для последующих преобразований необходимо использовать LLVM, переводящий абстрактное дерево в IR (Intermediate Representation).

Оформление научно-исследовательской работы:

Расчётно-пояснительная записка на **20-30** листах формата А4 должна содержать постановку задачи, введение, аналитическую, конструкторскую, технологическую части, заключение, список литературы.

Дата выдачи задания « _____ » _____ 2024 г.

Руководитель курсового проекта _____ **А.А. Ступников**
(Подпись, дата) (И.О.Фамилия)

Студент _____ **О.Н. Дубовицкая**
(Подпись, дата) (И.О.Фамилия)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компилятор – программа, переводящая написанный на языке программирования текст в набор машинных кодов [1].

В процессе преобразования команд выполняется оптимизация кода и анализ ошибок, что позволяет улучшить производительность и избежать некоторых сбоев при выполнении программы [2].

Целью данной курсовой работы является разработка компилятора языка TinyC. Компилятор должен выполнять чтение текстового файла, содержащего код на языке TinyC, и генерировать на выходе LLVM IR программы, пригодное для запуска.

Для достижения цели необходимо решить следующие задачи:

– проанализировать грамматику языка TinyC;

– изучить существующие средства для анализа исходных кодов программ, системы для генерации низкоуровневого кода, запуск которого возможен на большинстве из используемых платформ и операционных систем;

– реализовать прототип компилятора языка TinyC:

– провести тестирование компилятора.

Глава 1. Аналитическая часть

1.1 Составляющие компилятора

Компилятор состоит из трёх основных компонентов [2]:

1) **Frontend** – компонент, который отвечает за первичную обработку исходного кода и создание внутреннего представления программы. Данный компонент обычно состоит из *препроцессора*, *лексического*, *синтаксического* и *семантического анализаторов*, *генератора промежуточного представления*.

2) **Middle-end** – компонент, который занимается машинно-независимой оптимизацией и преобразованием промежуточного представления программы, полученного от Frontend. Данный компонент может включать в себя различные оптимизации, такие как удаление недостижимого кода, упрощение выражений, выстраивание функций. Middle-end также может выполнять анализ зависимостей, определение времени жизни переменных и иные анализы, необходимые для оптимизации программы. После завершения работы Middle-end создаёт промежуточное представление программы, которое будет передано Backend для генерации машинного кода.

3) **Backend** – компонент, который выполняет преобразование промежуточного представления, полученного от Middle-end, в программу на языке целевой платформы (ассемблер или машинный код); отвечает за генерацию целевого кода, который может быть выполнен на конкретной аппаратной платформе или виртуальной машине. Данный компонент анализирует промежуточное представление и генерирует соответствующий машинный код для целевой архитектуры. Backend также может выполнять дополнительные оптимизации, специфичные для целевой архитектуры, такие как распределение регистров и сокращение размера кода. После завершения работы Backend генерирует исполняемый файл, который может быть запущен на целевой аппаратной платформе.

На рисунке 1.1 представлена схема слоёв инфраструктуры компилятора.

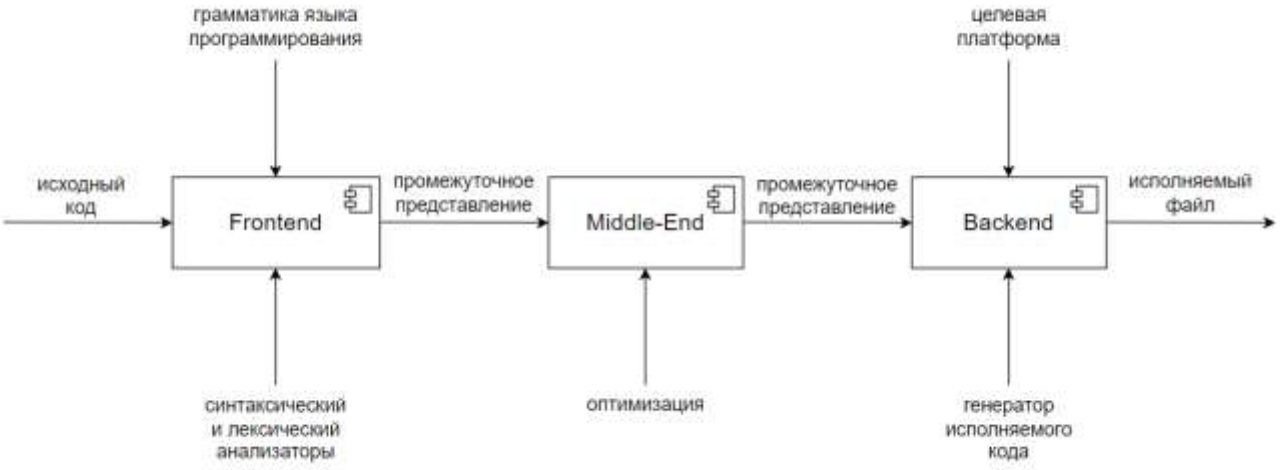

Рисунок 1.1 – Слои компилятора

В данной работе функции Middle-end и Backend компилятора будут осуществляться библиотекой LLVM, поэтому рассмотрим более подробно Fronted составляющих компилятора.

1.1.1 Препроцессор

Препроцессор компилятора – это компонент компилятора, который выполняет предварительную обработку исходного кода перед фазой фронтенда. Его задача заключается в обработке директив препроцессора и внесении соответствующих изменений в исходный код.

Препроцессоры создают входной поток информации для компилятора и выполняют следующие функции:

1) обработка макросов;
2) включение файлов;
3) обработка языковых расширений.

Препроцессор предоставляет набор директив, которые позволяют включать или исключать определённые части исходного кода, задавать макросы для замены текста и включать заголовочные файлы. Примером директив языка Си являются **include**, **define**, **pragma**. После работы препроцессора изменённый исходный код программы подаётся на вход лексического анализатора.

В данном проекте препроцессор не используется ввиду его избыточности.

1.1.2 Лексический анализатор

Лексический анализатор выполняет первичную обработку исходного кода, разбивая его на лексемы.

Цель лексического анализатора – превратить поток символов в токены (этот процесс называется *«токенизацией»*). Выполняется группировка определённых терминальных символов в лексемы. Задаются конкретные правила в виде регулярных выражений, детерминированных конечных автоматов, грамматик.

Его основные функции включают:

– удаление пробелов и комментариев,

– сборку последовательности цифр, формирующих константу,

– распознавание идентификаторов и ключевых слов.

Как правило, лексический анализатор находится между синтаксическим анализатором и входящим потоком и взаимодействует с ними таким образом, как показано на рисунке 1.2: анализатор считывает символы из входного потока, группирует их в лексемы и передаёт последующим стадиям компиляции токены, образуемые этими лексемами.

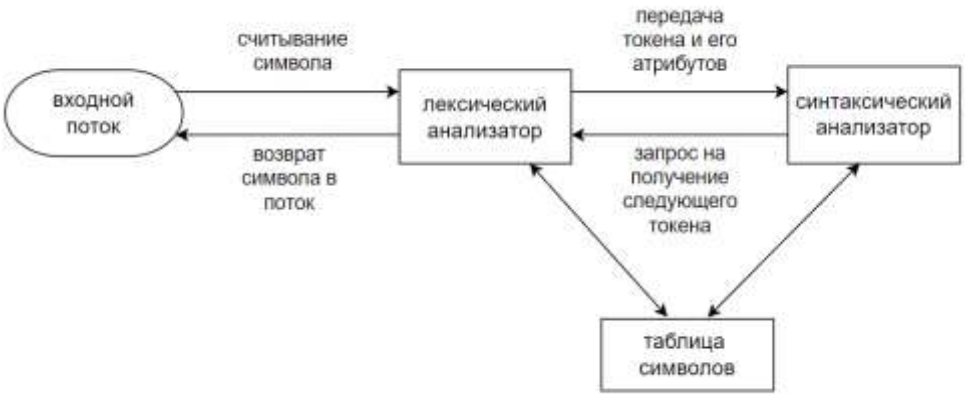

Рисунок 1.2 – Лексический анализатор

Лексический анализ может представляться как один из этапов *синтаксического анализа*. Обнаружение лексических ошибок, таких как недопустимые символы, ошибки идентификаторов или числовых констант, также является частью этого процесса.

1.1.3 Синтаксический анализатор

Иерархический анализ называется разбором (*«parsing»*) или синтаксическим анализом, который включает группировку токенов исходной программы в грамматические фразы, используемые компилятором. Обычно они представляются в виде дерева разбора.

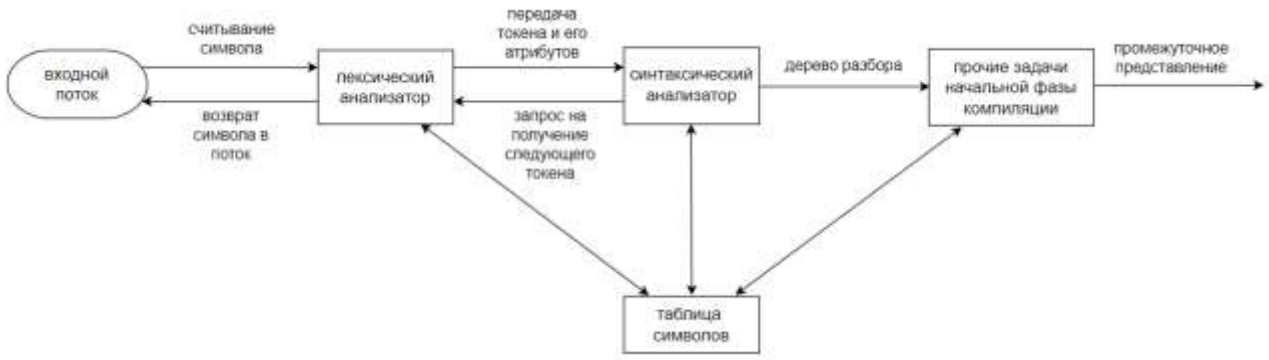

Рисунок 1.3 – Синтаксический анализатор

Как правило, это представление выражается в виде абстрактного синтаксического дерева, где каждый внутренний узел является оператором, а дочерние – его аргументами. Среди них можно выделить несколько групп связанных объектов:

– *элементы арифметических выражений* – каждый узел представляет собой операцию и содержит её аргументы;

– *элементы системы типов* – базовые типы (числовые, строковые, структуры и т.п.), указатели, массивы и функции;

– *выражения пяти типов* – арифметические, блочные и управляющие выражения, условные конструкции, циклы.

Разбор начинается со *стартового нетерминала*.

Условные конструкции описывают конструкцию «if», включающую в себя арифметическое выражение условия, выражение, выполняемое в случае его истинности, и альтернативное опциональное выражение.

Конструкции циклов (включают в себя «while», «do while» и «for») описывают арифметическое выражение условия и выражение, исполняемое в цикле.

Управляющие выражения – «break», «continue», «return» и т.д.

Блочные выражения – последовательность других выражений. Они преимущественно используются в качестве тел функций, условных выражений и циклов.

Полученная грамматическая структура используется на последующих этапах компиляции для анализа исходной программы и генерации кода для целевой платформы.

Синтаксический анализ выявляет синтаксические ошибки, относящиеся к нарушению структуры программы.

1.1.4 Семантический анализ

Семантический анализатор выполняет проверку семантики исходного кода, включая правильное использование типов данных, правила области видимости и согласованность операций.

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода.

На этом этапе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

Основными задачами семантического анализатора являются:

– установление семантической связи между различными частями программы;

– выявление потенциальных ошибок и несоответствия типов.

Как правило, семантический анализатор разделяется на ряд более мелких, каждый из которых предназначен для конкретной конструкции. Соответствующий семантический анализатор вызывается синтаксическим анализатором, как только он распознаёт синтаксическую единицу, требующую обработки.

Взаимодействие семантических анализаторов между собой достигается посредством информации, хранящейся в структурах данных, например, в таблице символов.

1.1.5 Генерация кода

Последний этап – генерация кода. Он начинается в тот момент, когда во все системные таблицы занесена необходимая информация. В этом случае компилятор переходит к построению соответствующей программы в машинном коде. Код генерируется при обходе дерева разбора, построенного на предыдущих этапах.

Для получения машинного кода необходимо осуществить:

– *генерацию промежуточного кода* – относится к последней фазе Frontend компилятора;

– *генерацию машинного кода* – относится к Middle-end и Backend компилятора.

После оптимизации промежуточного представления для каждого узла дерева генерируется соответствующий операции узла код на целевой платформе. В процессе анализа кода программы данные связываются с именами переменных. При выполнении генерации кода предполагается, что вход генератора не содержит ошибок.

Результат данного этапа – код, пригодный для исполнения на целевой платформе.

1.2 Методы реализации лексического и синтаксического анализаторов

Лексический и синтаксический анализаторы являются ключевыми компонентами компилятора, ответственными за преобразование исходного кода программы в промежуточное представление, которое необходимо для дальнейшей обработки.

Существуют два подхода для реализации лексического и синтаксического анализаторов:

– с использованием стандартных алгоритмов анализа;

– с привлечением готовых инструментов генерации.

1.2.1 Методы лексического анализа

В качестве стандартных алгоритмов анализа могут быть использованы LL-анализатор и LR-анализатор. Рассмотрим их подробнее [3].

LL-анализатор – это метод разбора для определённого подмножества контекстно-свободных, которые известны как LL-грамматики. Буква L в термине «LL-анализатор» обозначает, что входная строка обрабатывается с начала до конца и генерируется её левосторонний вывод. Если анализатор использует предварительный просмотр на k токенов при разборе входных данных, его называют LL(k)-анализатор.

LL(1)-анализаторы очень широко распространены, поскольку они просматривают входные данные только на один шаг вперёд, чтобы определить, какое грамматическое правило применить.

LR-анализатор читает входной поток слева направо и создаёт самую правую продукцию контекстно-свободной грамматики. Термин LR(k)-анализатор также используется. Здесь k означает количество непрочитанных символов предварительного просмотра во входном потоке, на основе которых принимаются решения в процессе анализа.

Синтаксис многих языков программирования определяется грамматикой LR(1) или аналогичной грамматикой. LR-анализатор обрабатывает код сверху вниз, поскольку он пытается создать продукцию верхнего уровня грамматики из листьев.

1.2.2 Генераторы лексического анализатора

Наиболее популярными из подобного рода генераторов являются Lex, Flex и ANTLR4. Рассмотрим их подробнее.

Lex (Lexical Analyzer) – стандартный инструмент для получения лексических анализаторов в операционных системах Unix, который обычно используется совместно с генератором синтаксических анализаторов Yacc.

В результате обработки входного потока получается исходный файл на языке Си.

Lex-файл разделяется на три блока (блок определений, правил и кода на Си), разделённых строками, содержащими по два символа процента [4].

В блоке определений задаются макросы и заголовочные файлы. Блок правил описывает шаблоны, представляющие собой регулярные выражения, и ассоциирует их с вызовами. Блок кода содержит операторы и функции на Си, которые копируются в генерируемый файл.

Flex (Fast Lexical Analyzer) – инструмент, который заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность [5].

ANTLR (ANother Tool for Language Recognition) – генератор лексических и синтаксических анализаторов, который позволяет создавать анализаторы на таких языках программирования, как Java, C#, Python, JavaScript, Go, C++, Swift, PHP [6].

ANTLR генерирует классы нисходящего рекурсивного синтаксического анализатора на основе правил, заданных в виде РБНФ грамматики (расширенная форма Бэкуса-Наура).

Также ANTLR позволяет строить и обходить деревья синтаксического анализа с использованием паттернов посетитель или слушатель.

1.2.3 Генераторы синтаксического анализатора

Для создания синтаксических анализаторов используются такие инструменты, как Yacc/Bison, Coco/R и описанный ранее ANTLR.

Yacc – стандартный генератор синтаксических парсеров в Unix системах.

Bison – аналог генератора Yacc для GNU систем [7].

Coco/R – генератор логических и синтаксических анализаторов. Лексические анализаторы работают по принципу конечных автоматов, а синтаксические используют рекурсивный спуск. Данный генератор поддерживает такие языки программирования, как C++, C#, Java и другие, и распространяется на условиях слегка смягчённой GNU General Public License [8].

1.3 Генерация исполняемого кода

Инструменты генерации исполняемого кода играют ключевую роль в процессоре компиляции программ. Они отвечают за трансляцию абстрактного синтаксического дерева (AST-дерева) или промежуточного представления (IR) программы в исполняемый машинный код или байткод, который может быть исполнен на целевой аппаратуре или виртуальной машине.

Генерация машинного кода относится к процессу преобразования исходного кода программы в низкоуровневый бинарный код, который может быть исполнен непосредственно аппаратурой процессора. Генерация машинного кода приводит к созданию исполняемых файлов, которые могут быть непосредственно запущены на целевой платформе без дополнительной обработки.

Виртуальная машина – это абстрактная вычислительная машина, которая исполняет программный код, представленный в виде некоторого промежуточного представления – байткода. Виртуальные машины обычно используются для исполнения программного кода, который был предварительно скомпилирован в промежуточное представление, а не в машинный код конкретной архитектуры процессора [9]. Примерами виртуальных машин являются Java Virtual Machine (JVM), Common Language Runtime (CLR) и V8 (WASM) для веб-приложений.

LLVM (Low Level Virtual Machine) представляет собой инфраструктуру для разработки компиляторов и связанных инструментов. Он включает в себя мощный оптимизатор и генератор машинного кода, который может работать с различными архитектурами процессоров. LLVM генерирует промежуточное представление LLVM IR, которое затем транслируется в машинный код для целевой архитектуры. Он также поддерживает генерацию байткода для виртуальных машин, таких как JVM и WASM [10].

GCC (GNU Compiler Collection) – это другая широко используемая инфраструктура для разработки компиляторов. В качестве промежуточного представления используется GIMPLE. GCC генерирует машинный код для

различных архитектур процессоров и поддерживает генерацию байткода для некоторых виртуальных машин [11].

Java ASM (Java bytecode manipulation framework) – это библиотека на языке Java, предназначенная для манипулирования байткодом Java. Её можно использовать для модификации существующих классов или для динамической генерации классов непосредственно в форме байткода. ASM предоставляет некоторые распространенные преобразования байт-кода и алгоритмы анализа, на основе которых могут быть созданы пользовательские сложные преобразования и инструменты анализа кода. ASM предлагает функциональность, аналогичную другим фреймворкам байт-кода Java, но ориентирован на производительность.

ILAsm (Intermediate Language Assembler) – это инструмент для создания исполняемых программ для платформы CLR (Common Language Runtime), использующей байткод Intermediate Language (IL). Он позволяет разработчикам создавать IL код вручную и компилировать его в исполняемый байткод с помощью утилиты компиляции IL.

WASM – это низкоуровневый бинарный формат, предназначенный для исполнения в веб-браузерах. Binaryen – это инструментарий для манипулирования и оптимизации байткода WebAssembly. Он обеспечивает высокую производительность и безопасность и может быть использован для исполнения приложений на различных языках программирования в веб-среде.

1.4 Причины для выбора ANTLR4

В качестве лексического и синтаксического анализатора будет использован ANTLR4 (ANother Tool for Language Recognition). Этот выбор обосновывается рядом преимуществ и особенностей данного инструмента:

- поддержка генерации лексических и синтаксических анализаторов для широкого спектра языков программирования (Java, C#, Python, JavaScript, Go, C++, Swift, PHP и другие);
- удобный и интуитивно понятный синтаксис для описания грамматик языков программирования;

– автоматическая генерация синтаксического дерева;

– широкая и активная пользовательская база и развитое сообщество разработчиков.

По сравнению с такими генераторами, как Flex + Bison и Coco/R, ANTLR4 имеет преимущество в виде поддержки интегрированных инструментов в рамках IntelliJ IDEA и Eclipse и наличия встроенных инструментов отладки.

Также, с точки зрения лицензии, ANTLR является одним из наиболее гибких инструментов, предоставляя BSD-подобную лицензию на использование.

1.5 Причины для выбора LLVM

В качестве генератора машинного кода в данной работе будет использоваться именно LLVM. Этот выбор обоснован следующими факторами:

– поддержка большого количества целевых платформ;

– поддержка библиотек на различных языках (C, C++, Rust, Python и другие);

– поддержка основных типов данных – целые числа, числа с плавающей точкой различных точностей, массивы, структуры, функции;

– автоматическая оптимизация сгенерированного промежуточного представления;

– широкая и активная пользовательская база и развитое сообщество разработчиков;

– наличие интерпретатора промежуточного представления.

Рассмотрим данный генератор более подробно.

LLVM (Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. В его основе лежит платформонезависимая система кодирования машинных инструкций – байткод LLVM IR (Intermediate Representation).

LLVM может создавать байткод для множества платформ, включая ARM, x86, x86-64, GPU от AMD и Nvidia и другие. В проекте есть генераторы кода для

множества языков, а для компиляции LLVM IR в код платформы используется clang.

В состав LLVM входит также интерпретатор LLVM IR, способный исполнять код без компиляции в код платформы [12].

Некоторые проекты имеют собственные LLVM-компиляторы, например, LLVM-версия GCC.

LLVM поддерживает целые числа произвольной разрядности, числа с плавающей точкой, массивы, структуры и функции. Большинство инструкций в LLVM принимает два аргументов (операнда) и возвращает одно значение (трёхадресный код).

Значения в LLVM определяются текстовым идентификатором. Локальные значения обозначаются префиксом %, а глобальные – @. 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. LLVM IR строго типизирован, поэтому существуют операции приведения типов, которые явно кодируются специальными инструкциями. Кроме того, существуют инструкции преобразования между целыми числами и указателями, а также универсальная инструкция для приведения типов bitcast.

Помимо значений регистров в LLVM есть работа с памятью. Значения в памяти адресуются типизированными указателями. Обратиться к ней можно с помощью двух инструкций: load и store. Инструкция alloca выделяет память в стеке. Она автоматически освобождается при выходе из функции при помощи инструкций ret или unwind.

Для вычисления адресов элементов массивов и структур с правильной типизацией используется инструкция getelementptr. Она только вычисляет адрес без обращения к памяти, принимает произвольное количество индексов и может разыменовывать структуры любой вложенности.

Выводы

В данном разделе приведён обзор основных фаз компиляции, описана каждая из них. Также рассмотрены основные инструменты для реализации лексического и синтаксического анализаторов и для генерации машинного кода.

В результате проведённого обзора в качестве генератора лексического и синтаксического анализаторов был выбран ANTLR4, а в качестве генератора машинного кода – LLVM.

Глава 2. Конструкторская часть

2.1 IDEF0

Концептуальная модель разрабатываемого компилятора в нотации IDEF0 представлена на рисунке 2.1

Рисунок 2.1 – Концептуальная модель системы в нотации IDEF0

Для уточнения деталей разрабатываемого компилятора на рисунке 2.2 представлена детализированная IDEF0 диаграмма.

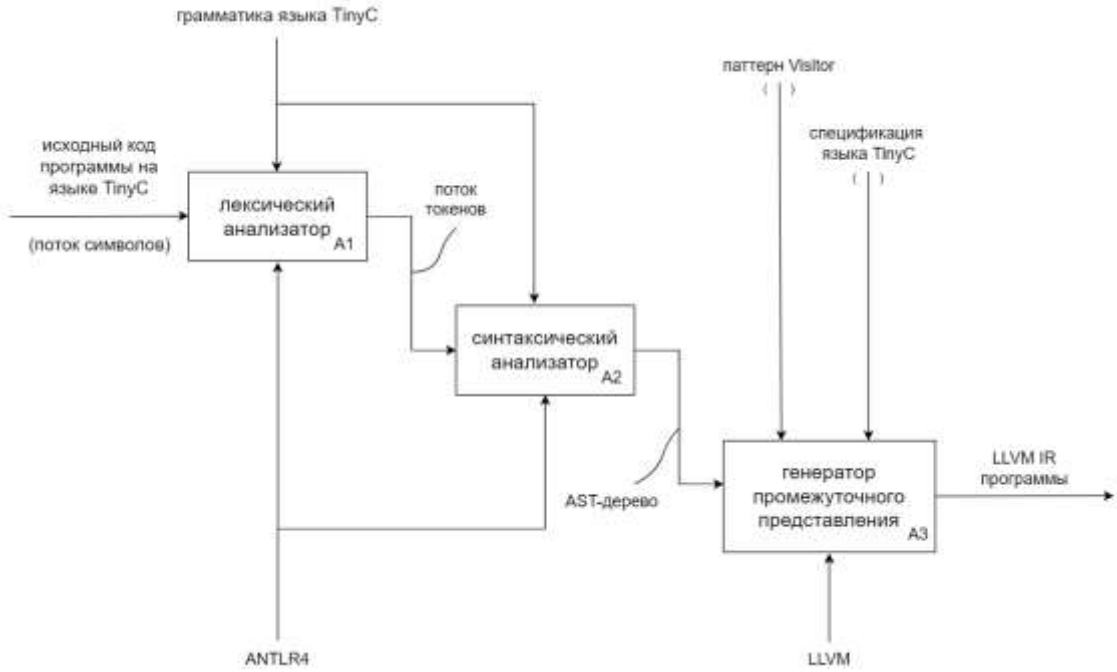

Рисунок 2.2 – Детализированная IDEF0 диаграмма компилятора языка программирования TinyC

2.2 Язык TinyC

TinyC – это мини-язык программирования, который является уменьшенной версией языка программирования C. Он разработан для использования в ограниченных средах, где требуется минимальный размер кода и низкие накладные расходы.

Основные особенности языка TinyC:

– TinyC содержит только основные элементы языка C, что делает его очень простым для изучения и использования;

– код на TinyC занимает меньше места, чем код на полной версии языка C, благодаря чему он идеально подходит для ограниченных платформ и устройств;

– исполнение программ на TinyC требует меньше ресурсов, чем на языках программирования более высокого уровня, что делает его хорошим выбором для встраиваемых систем и микроконтроллеров;

– TinyC поддерживает стандартные библиотеки языка C, что обеспечивает совместимость с другими программами и библиотеками;

– TinyC может быть использован на различных платформах, что делает его универсальным языком программирования для различных задач;

– TinyC обладает встроенной оптимизацией кода, что позволяет создавать быстрые и эффективные программы даже на ограниченных ресурсах.

В целом, TinyC является лёгким, компактным и эффективным языком программирования, который подходит для разработки простых и быстрых программ на ограниченных платформах. Он способен предоставить следующие функции:

– объявление переменных и констант,

– арифметические операции (сложение, вычитание, умножение, деление),

– логические операции (AND, OR, NOT),

– условные операторы (if, else),

– циклы (for, while, do-while),
– функции (объявление, вызов),
– пользовательские типы данных (структуры, объединения),
– массивы,
– указатели,
– ввод и вывод данных (printf, scanf),
– модули и библиотеки,
– работа с файлами.

В рамках данного проекта за основу взята грамматика языка C (приложение А), дальнейшая обработка которой будет адаптирована под язык TinyC.

2.3 Лексический и синтаксический анализаторы

Лексический и синтаксический анализаторы в данной работе генерируются с помощью ANTLR. На вход данного инструмента поступает грамматика языка в формате ANTLR4 (файл с расширением .g4).

В результате работы создаются файлы, содержащие классы лексера и парсера, а также вспомогательные файлы и классы для их работы. Также генерируются шаблоны классов для обхода дерева разбора, которое получается в результате работы парсера.

На вход лексера подаётся текст программы, преобразованный в поток символов. На выходе получается поток токенов, который затем подаётся на вход парсера. Результатом его работы является дерево разбора.

Ошибки, возникающие в ходе работы лексера и парсера, выводятся в стандартный поток ввода-вывода.

2.4 Семантический анализ

Абстрактное синтаксическое дерево можно обойти двумя способами: применяя паттерн *Listener* или *Visitor*.

Listener позволяет обходить дерево в глубину и вызывает обработчики соответствующих событий при входе и выходе из узла дерева.

Visitor предоставляет возможность более гибко обходить построенное дерево и решить, какие узлы и в каком порядке нужно посетить. Таким образом, для каждого узла реализуется метод его посещения. Обход начинается с точки входа в программу (корневого узла).

В данном проекте был использован паттерн Visitor для обхода абстрактного синтаксического дерева. Данный поход обеспечивает гибкость и позволяет точно контролировать порядок посещения узлов дерева, что удобно при реализации различных анализов и преобразований.

2.5 Обход синтаксического дерева

Исходная программа преобразуется в синтаксическое дерево при помощи кода, сгенерированного ANTLR4 для описанной грамматики. Обход всех узлов данного представления позволяет сгенерировать LLVM IR.

2.6 Генерация LLVM IR

Сгенерировать промежуточное представление LLVM можно путём обхода всех узлов синтаксического дерева. Каждый узел может создавать новые инструкции, блоки, функции и т.п. в зависимости от его типа и дочерних узлов.

Выводы

В текущем разделе была представлена концептуальная модель в нотации IDEF0, рассмотрены основные особенности языка программирования TinyC и приведена его грамматика, описаны принципы работы лексического и синтаксического анализаторов, идеи семантического анализа, обхода синтаксического дерева и генерации LLVM IR.

Глава 3. Технологическая часть

3.1 Обоснование выбора средств программной реализации

В качестве языка программирования был выбран Python, ввиду нескольких причин:

– на момент реализации был накоплен опыт в использовании данного языка программирования при выполнении других проектов и лабораторных работ;

– в Python есть библиотеки для работы с ANTLR4 и LLVM:

antlr4-python3-runtime предоставляет инструменты для разбора и анализа текста с использованием ANTLR4, а *llvmlite* предоставляет интерфейс для работы с низкоуровневым компилятором LLVM, позволяя генерировать и оптимизировать код на языке ассемблера LLVM.

3.2 Сгенерированные классы анализаторов

В результате работы ANTLR генерируются следующие файлы:

– *TinyC.interp* и *TinyCLexer.interp* содержат информацию о внутренннем представлении грамматики TinyC и лексера TinyC. Они используются ANTLR4 для построения парсера и лексера.

– *TinyC.tokens* и *TinyCLexer.tokens* содержат список токенов, которые могут быть распознаны парсером и лексером TinyC. Каждый токен имеет уникальный номер и имя.

– *TinyCListener.py* и *TinyCVisitor.py* содержат классы, которые могут быть использованы для обхода дерева синтаксического анализа, созданного парсером *TinyCParser*.

– TinyCParser.py содержит сгенерированный парсер для грамматики TinyC. Он используется для анализа входного текста и построения дерева разбора.

– *TinyCLexer.py* содержит сгенерированный лексер для грамматики TinyC. Он используется для токенизации входного текста и передачи токенов парсеру для дальнейшего анализа.

3.3 Тестирование программы

Для тестирования программы были написаны программы на языке TinyC. Тестирование проводится в соответствии со следующими шагами:

1) Для тестовой программы создаётся заполненная структура LLVM модуля, которая записывается в текстовом формате в .ll файл.

Данное действие выполняется с помощью ввода в терминал команды для выполнения программы на языке Python с использованием файла *llvm_ir.py* и файла тестовой программы.

Например, команда может иметь вид:

python llvm_ir.py example.tc

2) Полученный на предыдущем шаге файл с LLVM IR кодом компилируется и выполняется с помощью библиотеки llvmlite.

Данное действие выполняется с помощью ввода в терминал команды для выполнения программы на языке Python с использованием файла *executor.py* и полученного на предыдущем шаге .ll файла.

Например, команда может иметь вид:

python executor.py example.ll

3) После проведения шагов 1 и 2 проверяется корректность результата, выведенного на экран терминала.

Поскольку в рамках рассматриваемой грамматики было предусмотрено использование функций вывода, выполнение шага 3 окажется возможным. О правильной работе тестовой программы можно судить после сравнения результатов, который ожидаются и который выводится на экран после шага 3.

3.4 Примеры работы программы

Для лучшего понимания функционала и изучения возможностей программы в рамках Приложения В на соответствующих листингах приведены примеры её работы. Все примеры разбиты по парам: *«код на языке TinyC – соответствующий ему файл промежуточного представления»*.

Примеры представлены по следующим темам:

1) обратный связанный список (листинги 1.1 и 1.2),
2) ввод и вывод (листинги 2.1 и 2.2),
3) арифметические операции (листинги 3.1 и 3.2),
4) логические операции и условный оператор (листинги 4.1 и 4.2),
5) типы данных (листинги 5.1 и 5.2),
6) функция, одномерный массив и цикл for (листинги 6.1 и 6.2),
7) двумерный массив, символ, строка и цикл for (листинги 7.1 и 7.2),
8) структура (листинги 8.1 и 8.2).

ЗАКЛЮЧЕНИЕ

В рамках текущей курсовой работы рассмотрены основные части компилятора, алгоритмы и способы их реализации. Также были рассмотрены инструменты для создания лексического и синтаксического анализаторов и для генерации машинного кода.

Был разработан прототип компилятора языка TinyC, использующий ANTLR4 для синтаксического анализа входного потока данных и построения AST-дерева, и LLVM для последующих преобразований, переводящих абстрактное дерево в IR.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19781-83 // Вычислительная техника. Терминология: Справочное пособие. Выпуск 1 / Рецензент канд. техн. наук Ю. П. Селиванов. – М.: Издательство стандартов, 1989. – 168 с. – 55 000 экз. – ISBN 5-7050-0155-X.;
2. АХО А.В., ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.
3. Sippu Seppo, Soisalon-Soininen Eljas. Parsing Theory: Volume II LR (k) and LL (k) Parsing. — Springer Science & Business Media, 2013. — Т. 20.
4. Lesk M.E., Schmidt E. Lex: A lexical analyzer generator. – Murray Hill, NJ : Bell Laboratories, 1975. – С. 1-13.
5. Sampath P. et al. How to test program generators? A case study using flex // Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007). – IEEE, 2007. – С. 80-92.
6. What is ANTLR? [Электронный ресурс]. – Режим доступа: https://www.antlr.org/ (Дата обращения: 28.04.2024)
7. Donnelly C. BISON the YACC-compatible parser generator // Technical report, Free Software Foundation. – 1988.
8. Hanspeter Mössenböck, Löberbauer Markus, Albrecht Wöß. The Compiler Generator Coco/R. – 2018. – Режим доступа: https://ssw.jku.at/Research/Projects/Coco/ (дата обращения: 28.04.2024).
9. Appel Andrew Wilson. A Runtime System // Journal of Lisp and Symbolic Computation 3. — 1990. — С. 343–380.
10. Foundation LLVM. The LLVM Compiler Infrastructure Project. — 2024. — Режим доступа: https://llvm.org/ (дата обращения: 28.04.2024)
11. Vichare Abhijat. The Conceptual Structure of GCC // Indian Institute of Technology, Bombay. — 2008.
12. The LLVM Compiler Infrastructure Project [Электронный ресурс]. – Режим доступа: https://llvm.org/ (Дата обращения: 28.04.2024).

ПРИЛОЖЕНИЕ А

```
grammar TinyC;

compilationUnit :   translationUnit? EOF;

translationUnit:   externalDeclaration | translationUnit externalDeclaration;

externalDeclaration:   functionDefinition | declaration | ';';

functionDefinition :   declarationSpecifiers? declarator declarationList? compoundStatement;

declarationList:   declaration | declarationList declaration;

declaration
    :   declarationSpecifiers initDeclaratorList ';'
    |   declarationSpecifiers ';'
    |   staticAssertDeclaration
    ;

declarationSpecifiers :   declarationSpecifier+;

declarationSpecifiers2:   declarationSpecifier+;

declarationSpecifier
    :   storageClassSpecifier
    |   typeSpecifier
    |   typeQualifier
    |   functionSpecifier
    |   alignmentSpecifier
    ;

initDeclaratorList:   initDeclarator | initDeclaratorList ',' initDeclarator;

initDeclarator:   declarator | declarator '=' initializer;

storageClassSpecifier
    :   'typedef'
    |   'extern'
    |   'static'
    |   '_Thread_local'
    |   'auto'
    |   'register'
    ;

typeSpecifier
    :   ('void'
    |   'char'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    |   'signed'
    |   'unsigned'
    |   '_Bool'
    |   '_Complex'
    |   '__m128'
    |   '__m128d'
    |   '__m128i')
    |   '__extension__' '(' ('__m128' | '__m128d' | '__m128i') ')'
    |   atomicTypeSpecifier
    |   structOrUnionSpecifier
    |   enumSpecifier
    |   typedefName
    |   '__typeof__' '(' constantExpression ')' // GCC extension
    |   typeSpecifier pointer
    ;

structOrUnionSpecifier
    :   structOrUnion Identifier? '{' structDeclarationList '}'
    |   structOrUnion Identifier
    ;

structOrUnion:   'struct' | 'union';
```

```
structDeclarationList:   structDeclaration | structDeclarationList structDeclaration;

structDeclaration
    :   specifierQualifierList structDeclaratorList? ';'
    |   staticAssertDeclaration
    ;

specifierQualifierList
    :   typeSpecifier specifierQualifierList?
    |   typeQualifier specifierQualifierList?
    ;

structDeclaratorList
    :   structDeclarator
    |   structDeclaratorList ',' structDeclarator
    ;

structDeclarator:   declarator | declarator? ':' constantExpression;

primaryExpression
    :   Identifier
    |   Constant
    |   StringLiteral+
    |   '(' expression ')'
    |   genericSelection
    |   '__extension__'? '(' compoundStatement ')' // Blocks (GCC extension)
    |   '__builtin_va_arg' '(' unaryExpression ',' typeName ')'
    |   '__builtin_offsetof' '(' typeName ',' unaryExpression ')'
    ;

genericSelection :   '_Generic' '(' assignmentExpression ',' genericAssocList ')';

genericAssocList:   genericAssociation | genericAssocList ',' genericAssociation;

genericAssociation
    :   typeName ':' assignmentExpression
    |   'default' ':' assignmentExpression
    ;

postfixExpression
    :   primaryExpression
    |   postfixExpression '[' expression ']'
    |   postfixExpression '(' argumentExpressionList? ')'
    |   postfixExpression '.' Identifier
    |   postfixExpression '->' Identifier
    |   postfixExpression '++'
    |   postfixExpression '--'
    |   '(' typeName ')' '{' initializerList '}'
    |   '(' typeName ')' '{' initializerList ',' '}'
    |   '__extension__' '(' typeName ')' '{' initializerList '}'
    |   '__extension__' '(' typeName ')' '{' initializerList ',' '}'
    ;

argumentExpressionList:   assignmentExpression | argumentExpressionList ',' assignmentExpression;

unaryExpression
    :   postfixExpression
    |   '++' unaryExpression
    |   '--' unaryExpression
    |   unaryOperator castExpression
    |   'sizeof' unaryExpression
    |   'sizeof' '(' typeName ')'
    |   '_Alignof' '(' typeName ')'
    |   '&&' Identifier // GCC extension address of label
    ;

unaryOperator :   '&' | '*' | '+' | '-' | '~' | '!';

castExpression
    :   '(' typeName ')' castExpression
    |   '__extension__' '(' typeName ')' castExpression
    |   unaryExpression
    |   DigitSequence // for
    ;
```

```
multiplicativeExpression
    :   castExpression
    |   multiplicativeExpression '*' castExpression
    |   multiplicativeExpression '/' castExpression
    |   multiplicativeExpression '%' castExpression
    ;

additiveExpression
    :   multiplicativeExpression
    |   additiveExpression '+' multiplicativeExpression
    |   additiveExpression '-' multiplicativeExpression
    ;

shiftExpression
    :   additiveExpression
    |   shiftExpression '<<' additiveExpression
    |   shiftExpression '>>' additiveExpression
    ;

relationalExpression
    :   shiftExpression
    |   relationalExpression '<' shiftExpression
    |   relationalExpression '>' shiftExpression
    |   relationalExpression '<=' shiftExpression
    |   relationalExpression '>=' shiftExpression
    ;

equalityExpression
    :   relationalExpression
    |   equalityExpression '==' relationalExpression
    |   equalityExpression '!=' relationalExpression
    ;

andExpression:   equalityExpression | andExpression '&' equalityExpression;

exclusiveOrExpression:   andExpression | exclusiveOrExpression '^' andExpression;

inclusiveOrExpression:   exclusiveOrExpression | inclusiveOrExpression '|' exclusiveOrExpression;

logicalAndExpression:   inclusiveOrExpression | logicalAndExpression '&&' inclusiveOrExpression;

logicalOrExpression:   logicalAndExpression | logicalOrExpression '||' logicalAndExpression;

conditionalExpression :   logicalOrExpression ('?' expression ':' conditionalExpression)?;

assignmentExpression
    :   conditionalExpression
    |   unaryExpression assignmentOperator assignmentExpression
    |   DigitSequence // for
    ;

assignmentOperator :   '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' | '^=' | '|=';

expression:   assignmentExpression | expression ',' assignmentExpression;

constantExpression :   conditionalExpression;

enumSpecifier
    :   'enum' Identifier? '{' enumeratorList '}'
    |   'enum' Identifier? '{' enumeratorList ',' '}'
    |   'enum' Identifier
    ;

enumeratorList:   enumerator | enumeratorList ',' enumerator;

enumerator:   enumerationConstant | enumerationConstant '=' constantExpression;

enumerationConstant :   Identifier;

atomicTypeSpecifier :   '_Atomic' '(' typeName ')';

typeQualifier
    :   'const'
    |   'restrict'
    |   'volatile'
    |   '_Atomic'
    ;
```

```
functionSpecifier
    :   ('inline'
    |   '_Noreturn'
    |   '__inline__' // GCC extension
    |   '__stdcall')
    |   gccAttributeSpecifier
    |   '__declspec' '(' Identifier ')'
    ;

alignmentSpecifier
    :   '_Alignas' '(' typeName ')'
    |   '_Alignas' '(' constantExpression ')'
    ;

declarator :   pointer? directDeclarator gccDeclaratorExtension*;

directDeclarator
    :   Identifier
    |   '(' declarator ')'
    |   directDeclarator '[' typeQualifierList? assignmentExpression? ']'
    |   directDeclarator '[' 'static' typeQualifierList? assignmentExpression ']'
    |   directDeclarator '[' typeQualifierList 'static' assignmentExpression ']'
    |   directDeclarator '[' typeQualifierList? '*' ']'
    |   directDeclarator '(' parameterTypeList ')'
    |   directDeclarator '(' identifierList? ')'
    |   Identifier ':' DigitSequence  // bit field
    ;

gccDeclaratorExtension
    :   '__asm' '(' StringLiteral+ ')'
    |   gccAttributeSpecifier
    ;

gccAttributeSpecifier :   '__attribute__' '(' '(' gccAttributeList ')' ')';

gccAttributeList
    :   gccAttribute (',' gccAttribute)*
    |   // empty
    ;

gccAttribute
    :   ~(',' | '(' | ')') // relaxed def for "identifier or reserved word"
        ('(' argumentExpressionList? ')')?
    |   // empty
    ;

nestedParenthesesBlock
    :   (   ~('(' | ')')
        |   '(' nestedParenthesesBlock ')'
        )*
    ;

pointer
    :   '*' typeQualifierList?
    |   '*' typeQualifierList? pointer
    |   '^' typeQualifierList? // Blocks language extension
    |   '^' typeQualifierList? pointer // Blocks language extension
    ;

typeQualifierList:   typeQualifier | typeQualifierList typeQualifier;

parameterTypeList:   parameterList | parameterList ',' '...';

parameterList:   parameterDeclaration | parameterList ',' parameterDeclaration;

parameterDeclaration
    :   declarationSpecifiers declarator
    |   declarationSpecifiers2 abstractDeclarator?
    ;

identifierList:   Identifier | identifierList ',' Identifier;

typeName :   specifierQualifierList abstractDeclarator?;

abstractDeclarator:   pointer | pointer? directAbstractDeclarator gccDeclaratorExtension*;
```

```
directAbstractDeclarator
    :   '(' abstractDeclarator ')' gccDeclaratorExtension*
    |   '[' typeQualifierList? assignmentExpression? ']'
    |   '[' 'static' typeQualifierList? assignmentExpression ']'
    |   '[' typeQualifierList 'static' assignmentExpression ']'
    |   '[' '*' ']'
    |   '(' parameterTypeList? ')' gccDeclaratorExtension*
    |   directAbstractDeclarator '[' typeQualifierList? assignmentExpression? ']'
    |   directAbstractDeclarator '[' 'static' typeQualifierList? assignmentExpression ']'
    |   directAbstractDeclarator '[' typeQualifierList 'static' assignmentExpression ']'
    |   directAbstractDeclarator '[' '*' ']'
    |   directAbstractDeclarator '(' parameterTypeList? ')' gccDeclaratorExtension*
    ;

typedefName :   Identifier;

initializer
    :   assignmentExpression
    |   '{' initializerList '}'
    |   '{' initializerList ',' '}'
    ;

initializerList
    :   designation? initializer
    |   initializerList ',' designation? initializer
    ;

designation :   designatorList '=';

designatorList:   designator | designatorList designator;

designator:   '[' constantExpression ']' | '.' Identifier;

staticAssertDeclaration :   '_Static_assert' '(' constantExpression ',' StringLiteral+ ')' ';';

statement
    :   labeledStatement
    |   compoundStatement
    |   expressionStatement
    |   selectionStatement
    |   iterationStatement
    |   jumpStatement
    |   ('__asm' | '__asm__') ('volatile' | '__volatile__') '(' (logicalOrExpression (','
logicalOrExpression)*)? (':' (logicalOrExpression (',' logicalOrExpression)*)?)* ')' ';'
    ;

labeledStatement
    :   Identifier ':' statement
    |   'case' constantExpression ':' statement
    |   'default' ':' statement
    ;

compoundStatement :   '{' blockItemList? '}';

blockItemList:   blockItem | blockItemList blockItem;

blockItem:   statement | declaration;

expressionStatement :   expression? ';';

selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
    |   'switch' '(' expression ')' statement
    ;

iterationStatement
    :   While '(' expression ')' statement
    |   Do statement While '(' expression ')' ';'
    |   For '(' forCondition ')' statement
    ;

forCondition
        :   forDeclaration ';' forExpression? ';' forExpression?
        |   expression? ';' forExpression? ';' forExpression?
        ;
```

```
forDeclaration:   declarationSpecifiers initDeclaratorList | declarationSpecifiers;

forExpression:   assignmentExpression | forExpression ',' assignmentExpression;

jumpStatement
    :   'goto' Identifier ';'
    |   'continue' ';'
    |   'break' ';'
    |   'return' expression? ';'
    |   'goto' unaryExpression ';' // GCC extension
    ;

Auto : 'auto';
Break : 'break';
Case : 'case';
Char : 'char';
Const : 'const';
Continue : 'continue';
Default : 'default';
Do : 'do';
Double : 'double';
Else : 'else';
Enum : 'enum';
Extern : 'extern';
Float : 'float';
For : 'for';
Goto : 'goto';
If : 'if';
Inline : 'inline';
Int : 'int';
Long : 'long';
Register : 'register';
Restrict : 'restrict';
Return : 'return';
Short : 'short';
Signed : 'signed';
Sizeof : 'sizeof';
Static : 'static';
Struct : 'struct';
Switch : 'switch';
Typedef : 'typedef';
Union : 'union';
Unsigned : 'unsigned';
Void : 'void';
Volatile : 'volatile';
While : 'while';

Alignas : '_Alignas';
Alignof : '_Alignof';
Atomic : '_Atomic';
Bool : '_Bool';
Complex : '_Complex';
Generic : '_Generic';
Imaginary : '_Imaginary';
Noreturn : '_Noreturn';
StaticAssert : '_Static_assert';
ThreadLocal : '_Thread_local';

LeftParen : '(';
RightParen : ')';
LeftBracket : '[';
RightBracket : ']';
LeftBrace : '{';
RightBrace : '}';

Less : '<';
LessEqual : '<=';
Greater : '>';
GreaterEqual : '>=';
LeftShift : '<<';
RightShift : '>>';

Plus : '+';
PlusPlus : '++';
Minus : '-';
MinusMinus : '--';
Star : '*';
```

```
Div : '/';
Mod : '%';

And : '&';
Or : '|';
AndAnd : '&&';
OrOr : '||';
Caret : '^';
Not : '!';
Tilde : '~';

Question : '?';
Colon : ':';
Semi : ';';
Comma : ',';

Assign : '=';

StarAssign : '*=';
DivAssign : '/=';
ModAssign : '%=';
PlusAssign : '+=';
MinusAssign : '-=';
LeftShiftAssign : '<<=';
RightShiftAssign : '>>=';
AndAssign : '&=';
XorAssign : '^=';
OrAssign : '|=';

Equal : '==';
NotEqual : '!=';

Arrow : '->';
Dot : '.';
Ellipsis : '...';

Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

fragment
IdentifierNondigit
    :   Nondigit
    |   UniversalCharacterName
    ;

fragment
Nondigit :   [a-zA-Z_];

fragment
Digit:   [0-9];

fragment
UniversalCharacterName:   '\\u' HexQuad | '\\U' HexQuad HexQuad;

fragment
HexQuad :   HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit;

Constant
    :   IntegerConstant
    |   FloatingConstant
    |   CharacterConstant
    ;

fragment
IntegerConstant
    :   DecimalConstant IntegerSuffix?
    |   OctalConstant IntegerSuffix?
    |   HexadecimalConstant IntegerSuffix?
    |   BinaryConstant
    ;
fragment
BinaryConstant :'0' [bB] [0-1]+;
```

```
fragment
DecimalConstant :   NonzeroDigit Digit*;

fragment
OctalConstant :   '0' OctalDigit*;

fragment
HexadecimalConstant :   HexadecimalPrefix HexadecimalDigit+;

fragment
HexadecimalPrefix:   '0' [xX];

fragment
NonzeroDigit :   [1-9];

fragment
OctalDigit :   [0-7];

fragment
HexadecimalDigit :   [0-9a-fA-F];

fragment
IntegerSuffix
    :   UnsignedSuffix LongSuffix?
    |   UnsignedSuffix LongLongSuffix
    |   LongSuffix UnsignedSuffix?
    |   LongLongSuffix UnsignedSuffix?
    ;

fragment
UnsignedSuffix :   [uU];

fragment
LongSuffix :   [lL];

fragment
LongLongSuffix :   'll' | 'LL';

fragment
FloatingConstant:   DecimalFloatingConstant | HexadecimalFloatingConstant;

fragment
DecimalFloatingConstant
    :   FractionalConstant ExponentPart? FloatingSuffix?
    |   DigitSequence ExponentPart FloatingSuffix?
    ;

fragment
HexadecimalFloatingConstant
    :   HexadecimalPrefix HexadecimalFractionalConstant BinaryExponentPart FloatingSuffix?
    |   HexadecimalPrefix HexadecimalDigitSequence BinaryExponentPart FloatingSuffix?
    ;

fragment
FractionalConstant:   DigitSequence? '.' DigitSequence | DigitSequence '.';

fragment
ExponentPart:   'e' Sign? DigitSequence | 'E' Sign? DigitSequence;

fragment
Sign :   '+' | '-';

DigitSequence :   Digit+;

fragment
HexadecimalFractionalConstant
    :   HexadecimalDigitSequence? '.' HexadecimalDigitSequence
    |   HexadecimalDigitSequence '.'
    ;

fragment
BinaryExponentPart:   'p' Sign? DigitSequence | 'P' Sign? DigitSequence;


fragment
HexadecimalDigitSequence :   HexadecimalDigit+;
```

```
fragment
FloatingSuffix :   'f' | 'l' | 'F' | 'L';

fragment
CharacterConstant
    :   '\'' CCharSequence '\''
    |   'L\'' CCharSequence '\''
    |   'u\'' CCharSequence '\''
    |   'U\'' CCharSequence '\''
    ;

fragment
CCharSequence :   CChar+;

fragment
CChar:   ~['\\\r\n] | EscapeSequence;

fragment
EscapeSequence
    :   SimpleEscapeSequence
    |   OctalEscapeSequence
    |   HexadecimalEscapeSequence
    |   UniversalCharacterName
    ;

fragment
SimpleEscapeSequence :   '\\' ['"?abfnrtv\\];

fragment
OctalEscapeSequence
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
    |   '\\' OctalDigit OctalDigit OctalDigit
    ;

fragment
HexadecimalEscapeSequence:   '\\x' HexadecimalDigit+;

StringLiteral :   EncodingPrefix? '"' SCharSequence? '"';

fragment
EncodingPrefix:   'u8' | 'u' | 'U' | 'L';

fragment
SCharSequence :   SChar+;

fragment
SChar
    :   ~["\\\r\n]
    |   EscapeSequence
    |   '\\\n'   // Added line
    |   '\\\r\n' // Added line
    ;

ComplexDefine :   '#' Whitespace? 'define'  ~[#]* -> skip;

Include :   '#include' Whitespace* ~[\r\n]* -> skip;

AsmBlock :   'asm' ~'{'* '{' ~'}'* '}' -> skip;

LineAfterPreprocessing :   '#line' Whitespace* ~[\r\n]* -> skip;

LineDirective
    :   '#' Whitespace? DecimalConstant Whitespace? StringLiteral ~[\r\n]* -> skip;
PragmaDirective
    :   '#' Whitespace? 'pragma' Whitespace ~[\r\n]* -> skip;

Whitespace :   [ \t]+ -> skip;

Newline:   (   '\r' '\n'? | '\n') -> skip;

BlockComment :   '/*' .*? '*/' -> skip;

LineComment :   '//' ~[\r\n]* -> skip;
```

ПРИЛОЖЕНИЕ В

Листинг 1.1. Обратный связанный список

```
int printf(const char *format,...);
int nullptr = 0;
struct Node
{
    int data;
    struct Node* next;
};
void addNode(struct Node* pre, struct Node* next)
{
    pre->next = next;
}
void reverseLinkedList(struct Node** head)
{
    struct Node* prev = nullptr;
    struct Node* current = *head;
    struct Node* next;

    while (current != nullptr)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;
}

int main()
{
    printf("\n=== Linked list ===\n");

    struct Node n1, n2, n3;
    n1.data = 3;
    n1.next = nullptr;
    n2.data = 9;
    n2.next = nullptr;
    n3.data = 27;
    n3.next = nullptr;

    addNode(&n1, &n2);
    addNode(&n2, &n3);

    struct Node* head = &n1;
    printf("Original linked list:\n");
    struct Node* cur = head;
    do
    {
        printf(" data = %d\n", cur->data);
        cur = cur->next;
    }while(cur!=nullptr);

    reverseLinkedList(&head);
    printf("\nReversed linked list:\n");
    cur = head;
    do
    {
        printf(" data = %d\n", cur->data);
        cur = cur->next;
    }while(cur!=nullptr);

    return 0;
}
```

Листинг 1.2. Файл промежуточного представления для листинга 1.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""
%"Node" = type {i32, %"Node"*}
declare i32 @"printf"(i8* %".1", ...)
@"nullptr" = internal global i32 0
define void @"addNode"(%"Node"* %".1", %"Node"* %".2")
{
entry:
  %".4" = getelementptr %"Node", %"Node"* %".1", i32 0, i32 1
  %".5" = load %"Node"*, %"Node"** %".4"
  store %"Node"* %".2", %"Node"** %".4"
  ret void
}
define void @"reverseLinkedList"(%"Node"** %".1")
{
entry:
  %".3" = load i32, i32* @"nullptr"
  %".4" = inttoptr i32 %".3" to %"Node"*
  %".5" = alloca %"Node"*
  store %"Node"* %".4", %"Node"** %".5"
  %".7" = load %"Node"*, %"Node"** %".1"
  %".8" = alloca %"Node"*
  store %"Node"* %".7", %"Node"** %".8"
  %".10" = alloca %"Node"*
  br label %"entryloop_do"
entryloop_do:
  br label %"entry.loop_cond"
entry.loop_cond:
  %".13" = load i32, i32* @"nullptr"
  %".14" = load %"Node"*, %"Node"** %".8"
  %".15" = ptrtoint %"Node"* %".14" to i32
  %".16" = icmp ne i32 %".15", %".13"
  br i1 %".16", label %"entry.loop_body", label %"entry.loop_end"
entry.loop_body:
  %".18" = load %"Node"*, %"Node"** %".10"
  %".19" = load %"Node"*, %"Node"** %".8"
  %".20" = getelementptr %"Node", %"Node"* %".19", i32 0, i32 1
  %".21" = load %"Node"*, %"Node"** %".20"
  store %"Node"* %".21", %"Node"** %".10"
  %".23" = load %"Node"*, %"Node"** %".8"
  %".24" = getelementptr %"Node", %"Node"* %".23", i32 0, i32 1
  %".25" = load %"Node"*, %"Node"** %".24"
  %".26" = load %"Node"*, %"Node"** %".5"
  store %"Node"* %".26", %"Node"** %".24"
  %".28" = load %"Node"*, %"Node"** %".5"
  %".29" = load %"Node"*, %"Node"** %".8"
  store %"Node"* %".29", %"Node"** %".5"
  %".31" = load %"Node"*, %"Node"** %".8"
  %".32" = load %"Node"*, %"Node"** %".10"
  store %"Node"* %".32", %"Node"** %".8"
  br label %"entry.loop_update"
entry.loop_end:
  %".36" = load %"Node"*, %"Node"** %".1"
  %".37" = load %"Node"*, %"Node"** %".5"
  store %"Node"* %".37", %"Node"** %".1"
  ret void
entry.loop_update:
  br label %"entry.loop_cond"
}
define i32 @"main"()
{
entry:
  %".2" = alloca [22 x i8]
  store [22 x i8] c"\0a=== Linked list ===\0a\00", [22 x i8]* %".2"
  %".4" = getelementptr [22 x i8], [22 x i8]* %".2", i32 0, i32 0
  %".5" = call i32 (i8*, ...) @"printf"(i8* %".4")
```

```
  %".6" = alloca %"Node"
  %".7" = alloca %"Node"
  %".8" = alloca %"Node"
  %".9" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 0
  %".10" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 0
  %".11" = load i32, i32* %".10"
  store i32 3, i32* %".10"
  %".13" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 0
  %".14" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 1
  %".15" = load %"Node"*, %"Node"** %".14"
  %".16" = load i32, i32* @"nullptr"
  %".17" = inttoptr i32 %".16" to %"Node"*
  store %"Node"* %".17", %"Node"** %".14"
  %".19" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 0
  %".20" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 0
  %".21" = load i32, i32* %".20"
  store i32 9, i32* %".20"
  %".23" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 0
  %".24" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 1
  %".25" = load %"Node"*, %"Node"** %".24"
  %".26" = load i32, i32* @"nullptr"
  %".27" = inttoptr i32 %".26" to %"Node"*
  store %"Node"* %".27", %"Node"** %".24"
  %".29" = getelementptr %"Node", %"Node"* %".8", i32 0, i32 0
  %".30" = getelementptr %"Node", %"Node"* %".8", i32 0, i32 0
  %".31" = load i32, i32* %".30"
  store i32 27, i32* %".30"
  %".33" = getelementptr %"Node", %"Node"* %".8", i32 0, i32 0
  %".34" = getelementptr %"Node", %"Node"* %".8", i32 0, i32 1
  %".35" = load %"Node"*, %"Node"** %".34"
  %".36" = load i32, i32* @"nullptr"
  %".37" = inttoptr i32 %".36" to %"Node"*
  store %"Node"* %".37", %"Node"** %".34"
  %".39" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 0
  %".40" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 0
  call void @"addNode"(%"Node"* %".6", %"Node"* %".7")
  %".42" = getelementptr %"Node", %"Node"* %".7", i32 0, i32 0
  %".43" = getelementptr %"Node", %"Node"* %".8", i32 0, i32 0
  call void @"addNode"(%"Node"* %".7", %"Node"* %".8")
  %".45" = getelementptr %"Node", %"Node"* %".6", i32 0, i32 0
  %".46" = alloca %"Node"*
  store %"Node"* %".6", %"Node"** %".46"
  %".48" = alloca [23 x i8]
  store [23 x i8] c"Original linked list:\0a\00", [23 x i8]* %".48"
  %".50" = getelementptr [23 x i8], [23 x i8]* %".48", i32 0, i32 0
  %".51" = call i32 (i8*, ...) @"printf"(i8* %".50")
  %".52" = load %"Node"*, %"Node"** %".46"
  %".53" = alloca %"Node"*
  store %"Node"* %".52", %"Node"** %".53"
  br label %"entryloop_do"
entryloop_do:
  %".56" = load %"Node"*, %"Node"** %".53"
  %".57" = getelementptr %"Node", %"Node"* %".56", i32 0, i32 0
  %".58" = load i32, i32* %".57"
  %".59" = alloca [12 x i8]
  store [12 x i8] c" data = %d\0a\00", [12 x i8]* %".59"
  %".61" = getelementptr [12 x i8], [12 x i8]* %".59", i32 0, i32 0
  %".62" = call i32 (i8*, ...) @"printf"(i8* %".61", i32 %".58")
  %".63" = load %"Node"*, %"Node"** %".53"
  %".64" = load %"Node"*, %"Node"** %".53"
  %".65" = getelementptr %"Node", %"Node"* %".64", i32 0, i32 1
  %".66" = load %"Node"*, %"Node"** %".65"
  store %"Node"* %".66", %"Node"** %".53"
  br label %"entry.loop_cond"
entry.loop_cond:
  %".69" = load i32, i32* @"nullptr"
  %".70" = load %"Node"*, %"Node"** %".53"
  %".71" = ptrtoint %"Node"* %".70" to i32
  %".72" = icmp ne i32 %".71", %".69"
```

```
  br i1 %".72", label %"entry.loop_body", label %"entry.loop_end"
entry.loop_body:
  %".74" = load %"Node"*, %"Node"** %".53"
  %".75" = getelementptr %"Node", %"Node"* %".74", i32 0, i32 0
  %".76" = load i32, i32* %".75"
  %".77" = alloca [12 x i8]
  store [12 x i8] c" data = %d\0a\00", [12 x i8]* %".77"
  %".79" = getelementptr [12 x i8], [12 x i8]* %".77", i32 0, i32 0
  %".80" = call i32 (i8*, ...) @"printf"(i8* %".79", i32 %".76")
  %".81" = load %"Node"*, %"Node"** %".53"
  %".82" = load %"Node"*, %"Node"** %".53"
  %".83" = getelementptr %"Node", %"Node"* %".82", i32 0, i32 1
  %".84" = load %"Node"*, %"Node"** %".83"
  store %"Node"* %".84", %"Node"** %".53"
  br label %"entry.loop_update"
entry.loop_end:
  %".88" = load %"Node"*, %"Node"** %".46"
  call void @"reverseLinkedList"(%"Node"** %".46")
  %".90" = alloca [24 x i8]
  store [24 x i8] c"\0aReversed linked list:\0a\00", [24 x i8]* %".90"
  %".92" = getelementptr [24 x i8], [24 x i8]* %".90", i32 0, i32 0
  %".93" = call i32 (i8*, ...) @"printf"(i8* %".92")
  %".94" = load %"Node"*, %"Node"** %".53"
  %".95" = load %"Node"*, %"Node"** %".46"
  store %"Node"* %".95", %"Node"** %".53"
  br label %"entry.loop_endloop_do"
entry.loop_update:
  br label %"entry.loop_cond"
entry.loop_endloop_do:
  %".98" = load %"Node"*, %"Node"** %".53"
  %".99" = getelementptr %"Node", %"Node"* %".98", i32 0, i32 0
  %".100" = load i32, i32* %".99"
  %".101" = alloca [12 x i8]
  store [12 x i8] c" data = %d\0a\00", [12 x i8]* %".101"
  %".103" = getelementptr [12 x i8], [12 x i8]* %".101", i32 0, i32 0
  %".104" = call i32 (i8*, ...) @"printf"(i8* %".103", i32 %".100")
  %".105" = load %"Node"*, %"Node"** %".53"
  %".106" = load %"Node"*, %"Node"** %".53"
  %".107" = getelementptr %"Node", %"Node"* %".106", i32 0, i32 1
  %".108" = load %"Node"*, %"Node"** %".107"
  store %"Node"* %".108", %"Node"** %".53"
  br label %"entry.loop_end.loop_cond"
entry.loop_end.loop_cond:
  %".111" = load i32, i32* @"nullptr"
  %".112" = load %"Node"*, %"Node"** %".53"
  %".113" = ptrtoint %"Node"* %".112" to i32
  %".114" = icmp ne i32 %".113", %".111"
  br i1 %".114", label %"entry.loop_end.loop_body", label %"entry.loop_end.loop_end"
entry.loop_end.loop_body:
  %".116" = load %"Node"*, %"Node"** %".53"
  %".117" = getelementptr %"Node", %"Node"* %".116", i32 0, i32 0
  %".118" = load i32, i32* %".117"
  %".119" = alloca [12 x i8]
  store [12 x i8] c" data = %d\0a\00", [12 x i8]* %".119"
  %".121" = getelementptr [12 x i8], [12 x i8]* %".119", i32 0, i32 0
  %".122" = call i32 (i8*, ...) @"printf"(i8* %".121", i32 %".118")
  %".123" = load %"Node"*, %"Node"** %".53"
  %".124" = load %"Node"*, %"Node"** %".53"
  %".125" = getelementptr %"Node", %"Node"* %".124", i32 0, i32 1
  %".126" = load %"Node"*, %"Node"** %".125"
  store %"Node"* %".126", %"Node"** %".53"
  br label %"entry.loop_end.loop_update"
entry.loop_end.loop_end:
  ret i32 0
entry.loop_end.loop_update:
  br label %"entry.loop_end.loop_cond"
}
```

Листинг 2.1. Ввод и вывод

```
int scanf(const char * restrict format,...);
int printf(const char *format,...);
int main()
{
    printf("Please input number a (int values): ");
    int a;
    int c = 2;
    scanf("%d", &a);
    printf("a + c = %d\n", a+c);
    return 0;
}
```

Листинг 2.2. Файл промежуточного представления для листинга 2.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""
declare i32 @"scanf"(i8* %".1", ...)
declare i32 @"printf"(i8* %".1", ...)
define i32 @"main"()
{
entry:
  %".2" = alloca [37 x i8]
  store [37 x i8] c"Please input number a (int values): \00", [37 x i8]* %".2"
  %".4" = getelementptr [37 x i8], [37 x i8]* %".2", i32 0, i32 0
  %".5" = call i32 (i8*, ...) @"printf"(i8* %".4")
  %".6" = alloca i32
  %".7" = alloca i32
  store i32 2, i32* %".7"
  %".9" = load i32, i32* %".6"
  %".10" = alloca [3 x i8]
  store [3 x i8] c"%d\00", [3 x i8]* %".10"
  %".12" = getelementptr [3 x i8], [3 x i8]* %".10", i32 0, i32 0
  %".13" = call i32 (i8*, ...) @"scanf"(i8* %".12", i32* %".6")
  %".14" = load i32, i32* %".7"
  %".15" = load i32, i32* %".6"
  %".16" = add i32 %".15", %".14"
  %".17" = alloca [12 x i8]
  store [12 x i8] c"a + c = %d\0a\00", [12 x i8]* %".17"
  %".19" = getelementptr [12 x i8], [12 x i8]* %".17", i32 0, i32 0
  %".20" = call i32 (i8*, ...) @"printf"(i8* %".19", i32 %".16")
  ret i32 0
}
```

Листинг 3.1. Арифметические операции

```
int printf(const char *format,...);
int main() {
    int a;
    printf("\n=====Start test assignment=====\n");
    a = 1;
    printf("a = %d\n", a);
    a = a + 1;
    printf("a = a + 1, a = %d\n", a);
    a = a - 1;
    printf("a = a - 1, a = %d\n", a);
    a = a * 2;
    printf("a = a * 2, a = %d\n", a);
    a = a / 2;
    printf("a = a / 2, a = %d\n", a);
    a = 7;
    a = a % 4;
    printf("a = 7, a = a %% 4, a = %d\n", a);
```

```
    int b;
    printf("\n=====Start test operator=====\n");
    b = 1;
    printf("b = %d\n", b);
    b += 1;
    printf("b += 1, b = %d\n", b);
    b -= 1;
    printf("b -= 1, b = %d\n", b);
    b *= 2;
    printf("b *= 2, b = %d\n", b);
    b /= 2;
    printf("b /= 2, b = %d\n", b);
    b = 7;
    b %= 4;
    printf("b = 7, b %%= 4, b = %d\n", b);

    printf("\n=====Start arithmetic=====\n");
    printf("23+5 = %d (wait 28)\n", 23+5);
    printf("3*4 = %d (wait 12)\n", 3*4);
    printf("22-(6-4) = %d (wait 20)\n", 22-(6-4));
    printf("22/(6/3) = %d (wait 11)\n", 22/(6/3));
    printf("1+(5-2)*4/(2+1) = %d (wait 5)\n", 1+(5-2)*4/(2+1));
    printf("0+(1+23)/4*5*67-8+9 = %d (wait 2011)\n", 0+(1+23)/4*5*67-8+9);
    printf("192/(3*8+4*(33/5))-5 = %d (wait -1)\n", 192/(3*8+4*(33/5))-5);
    return 0;
}
```

Листинг 3.2. Файл промежуточного представления для листинга 3.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define i32 @"main"()
{
entry:
  %".2" = alloca i32
  %".3" = alloca [34 x i8]
  store [34 x i8] c"\0a=====Start test assignment=====\0a\00", [34 x i8]* %".3"
  %".5" = getelementptr [34 x i8], [34 x i8]* %".3", i32 0, i32 0
  %".6" = call i32 (i8*, ...) @"printf"(i8* %".5")
  %".7" = load i32, i32* %".2"
  store i32 1, i32* %".2"
  %".9" = load i32, i32* %".2"
  %".10" = alloca [8 x i8]
  store [8 x i8] c"a = %d\0a\00", [8 x i8]* %".10"
  %".12" = getelementptr [8 x i8], [8 x i8]* %".10", i32 0, i32 0
  %".13" = call i32 (i8*, ...) @"printf"(i8* %".12", i32 %".9")
  %".14" = load i32, i32* %".2"
  %".15" = load i32, i32* %".2"
  %".16" = add i32 %".15", 1
  store i32 %".16", i32* %".2"
  %".18" = load i32, i32* %".2"
  %".19" = alloca [19 x i8]
  store [19 x i8] c"a = a + 1, a = %d\0a\00", [19 x i8]* %".19"
  %".21" = getelementptr [19 x i8], [19 x i8]* %".19", i32 0, i32 0
  %".22" = call i32 (i8*, ...) @"printf"(i8* %".21", i32 %".18")
  %".23" = load i32, i32* %".2"
  %".24" = load i32, i32* %".2"
  %".25" = sub i32 %".24", 1
  store i32 %".25", i32* %".2"
  %".27" = load i32, i32* %".2"
  %".28" = alloca [19 x i8]
  store [19 x i8] c"a = a - 1, a = %d\0a\00", [19 x i8]* %".28"
  %".30" = getelementptr [19 x i8], [19 x i8]* %".28", i32 0, i32 0
```

```
  %".31" = call i32 (i8*, ...) @"printf"(i8* %".30", i32 %".27")
  %".32" = load i32, i32* %".2"
  %".33" = load i32, i32* %".2"
  %".34" = mul i32 %".33", 2
  store i32 %".34", i32* %".2"
  %".36" = load i32, i32* %".2"
  %".37" = alloca [19 x i8]
  store [19 x i8] c"a = a * 2, a = %d\0a\00", [19 x i8]* %".37"
  %".39" = getelementptr [19 x i8], [19 x i8]* %".37", i32 0, i32 0
  %".40" = call i32 (i8*, ...) @"printf"(i8* %".39", i32 %".36")
  %".41" = load i32, i32* %".2"
  %".42" = load i32, i32* %".2"
  %".43" = sdiv i32 %".42", 2
  store i32 %".43", i32* %".2"
  %".45" = load i32, i32* %".2"
  %".46" = alloca [19 x i8]
  store [19 x i8] c"a = a / 2, a = %d\0a\00", [19 x i8]* %".46"
  %".48" = getelementptr [19 x i8], [19 x i8]* %".46", i32 0, i32 0
  %".49" = call i32 (i8*, ...) @"printf"(i8* %".48", i32 %".45")
  %".50" = load i32, i32* %".2"
  store i32 7, i32* %".2"
  %".52" = load i32, i32* %".2"
  %".53" = load i32, i32* %".2"
  %".54" = srem i32 %".53", 4
  store i32 %".54", i32* %".2"
  %".56" = load i32, i32* %".2"
  %".57" = alloca [27 x i8]
  store [27 x i8] c"a = 7, a = a %% 4, a = %d\0a\00", [27 x i8]* %".57"
  %".59" = getelementptr [27 x i8], [27 x i8]* %".57", i32 0, i32 0
  %".60" = call i32 (i8*, ...) @"printf"(i8* %".59", i32 %".56")
  %".61" = alloca i32
  %".62" = alloca [32 x i8]
  store [32 x i8] c"\0a=====Start test operator=====\0a\00", [32 x i8]* %".62"
  %".64" = getelementptr [32 x i8], [32 x i8]* %".62", i32 0, i32 0
  %".65" = call i32 (i8*, ...) @"printf"(i8* %".64")
  %".66" = load i32, i32* %".61"
  store i32 1, i32* %".61"
  %".68" = load i32, i32* %".61"
  %".69" = alloca [8 x i8]
  store [8 x i8] c"b = %d\0a\00", [8 x i8]* %".69"
  %".71" = getelementptr [8 x i8], [8 x i8]* %".69", i32 0, i32 0
  %".72" = call i32 (i8*, ...) @"printf"(i8* %".71", i32 %".68")
  %".73" = load i32, i32* %".61"
  %".74" = add i32 %".73", 1
  store i32 %".74", i32* %".61"
  %".76" = load i32, i32* %".61"
  %".77" = alloca [16 x i8]
  store [16 x i8] c"b += 1, b = %d\0a\00", [16 x i8]* %".77"
  %".79" = getelementptr [16 x i8], [16 x i8]* %".77", i32 0, i32 0
  %".80" = call i32 (i8*, ...) @"printf"(i8* %".79", i32 %".76")
  %".81" = load i32, i32* %".61"
  %".82" = sub i32 %".81", 1
  store i32 %".82", i32* %".61"
  %".84" = load i32, i32* %".61"
  %".85" = alloca [16 x i8]
  store [16 x i8] c"b -= 1, b = %d\0a\00", [16 x i8]* %".85"
  %".87" = getelementptr [16 x i8], [16 x i8]* %".85", i32 0, i32 0
  %".88" = call i32 (i8*, ...) @"printf"(i8* %".87", i32 %".84")
  %".89" = load i32, i32* %".61"
  %".90" = mul i32 %".89", 2
  store i32 %".90", i32* %".61"
  %".92" = load i32, i32* %".61"
  %".93" = alloca [16 x i8]
  store [16 x i8] c"b *= 2, b = %d\0a\00", [16 x i8]* %".93"
  %".95" = getelementptr [16 x i8], [16 x i8]* %".93", i32 0, i32 0
  %".96" = call i32 (i8*, ...) @"printf"(i8* %".95", i32 %".92")
  %".97" = load i32, i32* %".61"
  %".98" = sdiv i32 %".97", 2
  store i32 %".98", i32* %".61"
```

```
  %".100" = load i32, i32* %".61"
  %".101" = alloca [16 x i8]
  store [16 x i8] c"b /= 2, b = %d\0a\00", [16 x i8]* %".101"
  %".103" = getelementptr [16 x i8], [16 x i8]* %".101", i32 0, i32 0
  %".104" = call i32 (i8*, ...) @"printf"(i8* %".103", i32 %".100")
  %".105" = load i32, i32* %".61"
  store i32 7, i32* %".61"
  %".107" = load i32, i32* %".61"
  %".108" = srem i32 %".107", 4
  store i32 %".108", i32* %".61"
  %".110" = load i32, i32* %".61"
  %".111" = alloca [24 x i8]
  store [24 x i8] c"b = 7, b %%= 4, b = %d\0a\00", [24 x i8]* %".111"
  %".113" = getelementptr [24 x i8], [24 x i8]* %".111", i32 0, i32 0
  %".114" = call i32 (i8*, ...) @"printf"(i8* %".113", i32 %".110")
  %".115" = alloca [29 x i8]
  store [29 x i8] c"\0a=====Start arithmetic=====\0a\00", [29 x i8]* %".115"
  %".117" = getelementptr [29 x i8], [29 x i8]* %".115", i32 0, i32 0
  %".118" = call i32 (i8*, ...) @"printf"(i8* %".117")
  %".119" = add i32 23, 5
  %".120" = alloca [21 x i8]
  store [21 x i8] c"23+5 = %d (wait 28)\0a\00", [21 x i8]* %".120"
  %".122" = getelementptr [21 x i8], [21 x i8]* %".120", i32 0, i32 0
  %".123" = call i32 (i8*, ...) @"printf"(i8* %".122", i32 %".119")
  %".124" = mul i32 3, 4
  %".125" = alloca [20 x i8]
  store [20 x i8] c"3*4 = %d (wait 12)\0a\00", [20 x i8]* %".125"
  %".127" = getelementptr [20 x i8], [20 x i8]* %".125", i32 0, i32 0
  %".128" = call i32 (i8*, ...) @"printf"(i8* %".127", i32 %".124")
  %".129" = sub i32 6, 4
  %".130" = sub i32 22, %".129"
  %".131" = alloca [25 x i8]
  store [25 x i8] c"22-(6-4) = %d (wait 20)\0a\00", [25 x i8]* %".131"
  %".133" = getelementptr [25 x i8], [25 x i8]* %".131", i32 0, i32 0
  %".134" = call i32 (i8*, ...) @"printf"(i8* %".133", i32 %".130")
  %".135" = sdiv i32 6, 3
  %".136" = sdiv i32 22, %".135"
  %".137" = alloca [25 x i8]
  store [25 x i8] c"22/(6/3) = %d (wait 11)\0a\00", [25 x i8]* %".137"
  %".139" = getelementptr [25 x i8], [25 x i8]* %".137", i32 0, i32 0
  %".140" = call i32 (i8*, ...) @"printf"(i8* %".139", i32 %".136")
  %".141" = add i32 2, 1
  %".142" = sub i32 5, 2
  %".143" = mul i32 %".142", 4
  %".144" = sdiv i32 %".143", %".141"
  %".145" = add i32 1, %".144"
  %".146" = alloca [31 x i8]
  store [31 x i8] c"1+(5-2)*4/(2+1) = %d (wait 5)\0a\00", [31 x i8]* %".146"
  %".148" = getelementptr [31 x i8], [31 x i8]* %".146", i32 0, i32 0
  %".149" = call i32 (i8*, ...) @"printf"(i8* %".148", i32 %".145")
  %".150" = add i32 1, 23
  %".151" = sdiv i32 %".150", 4
  %".152" = mul i32 %".151", 5
  %".153" = mul i32 %".152", 67
  %".154" = add i32 0, %".153"
  %".155" = sub i32 %".154", 8
  %".156" = add i32 %".155", 9
  %".157" = alloca [38 x i8]
  store [38 x i8] c"0+(1+23)/4*5*67-8+9 = %d (wait 2011)\0a\00", [38 x i8]* %".157"
  %".159" = getelementptr [38 x i8], [38 x i8]* %".157", i32 0, i32 0
  %".160" = call i32 (i8*, ...) @"printf"(i8* %".159", i32 %".156")
  %".161" = sdiv i32 33, 5
  %".162" = mul i32 4, %".161"
  %".163" = mul i32 3, 8
  %".164" = add i32 %".163", %".162"
  %".165" = sdiv i32 192, %".164"
  %".166" = sub i32 %".165", 5
  %".167" = alloca [37 x i8]
```

```
  store [37 x i8] c"192/(3*8+4*(33/5))-5 = %d (wait -1)\0a\00", [37 x i8]* %".167"
  %".169" = getelementptr [37 x i8], [37 x i8]* %".167", i32 0, i32 0
  %".170" = call i32 (i8*, ...) @"printf"(i8* %".169", i32 %".166")
  ret i32 0
}
```

Листинг 4.1. Логические операции и условный оператор

```
int printf(const char *format,...);
int main() {
    int x = -5;
    int y = 17;
    if (x > 0 && y > 0) {
        printf("Both x and y are greater than 0\n");
    } else if (x > 0) {
        printf("x is greater than 0\n");
    } else if (y > 0) {
        printf("y is greater than 0\n");
    } else {
        printf("Both x and y are smaller than 0\n");
    }
    if (x < 10 || y < 10) {
        printf("At least one of x or y is less than 10\n");
    }
    return 0;
}
```

Листинг 4.2. Файл промежуточного представления для листинга 4.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""
declare i32 @"printf"(i8* %".1", ...)
define i32 @"main"()
{
entry:
  %".2" = sub i32 0, 5
  %".3" = alloca i32
  store i32 %".2", i32* %".3"
  %".5" = alloca i32
  store i32 17, i32* %".5"
  %".7" = load i32, i32* %".3"
  %".8" = icmp sgt i32 %".7", 0
  %".9" = alloca i1
  br i1 %".8", label %"entry.if", label %"entry.else"
entry.if:
  %".11" = load i32, i32* %".5"
  %".12" = icmp sgt i32 %".11", 0
  store i1 %".12", i1* %".9"
  br label %"entry.endif"
entry.else:
  store i1 0, i1* %".9"
  br label %"entry.endif"
entry.endif:
  %".17" = load i1, i1* %".9"
  br i1 %".17", label %"entry.endif.if", label %"entry.endif.else"
entry.endif.if:
  %".19" = alloca [33 x i8]
  store [33 x i8] c"Both x and y are greater than 0\0a\00", [33 x i8]* %".19"
  %".21" = getelementptr [33 x i8], [33 x i8]* %".19", i32 0, i32 0
  %".22" = call i32 (i8*, ...) @"printf"(i8* %".21")
  br label %"entry.endif.endif"
entry.endif.else:
  %".24" = load i32, i32* %".3"
  %".25" = icmp sgt i32 %".24", 0
  br i1 %".25", label %"entry.endif.else.if", label %"entry.endif.else.else"
```

```
entry.endif.endif:
  %".47" = load i32, i32* %".3"
  %".48" = icmp slt i32 %".47", 10
  %".49" = alloca i1
  br i1 %".48", label %"entry.endif.endif.if", label %"entry.endif.endif.else"
entry.endif.else.if:
  %".27" = alloca [21 x i8]
  store [21 x i8] c"x is greater than 0\0a\00", [21 x i8]* %".27"
  %".29" = getelementptr [21 x i8], [21 x i8]* %".27", i32 0, i32 0
  %".30" = call i32 (i8*, ...) @"printf"(i8* %".29")
  br label %"entry.endif.else.endif"
entry.endif.else.else:
  %".32" = load i32, i32* %".5"
  %".33" = icmp sgt i32 %".32", 0
  br i1 %".33", label %"entry.endif.else.else.if", label %"entry.endif.else.else.else"
entry.endif.else.endif:
  br label %"entry.endif.endif"
entry.endif.else.else.if:
  %".35" = alloca [21 x i8]
  store [21 x i8] c"y is greater than 0\0a\00", [21 x i8]* %".35"
  %".37" = getelementptr [21 x i8], [21 x i8]* %".35", i32 0, i32 0
  %".38" = call i32 (i8*, ...) @"printf"(i8* %".37")
  br label %"entry.endif.else.else.endif"
entry.endif.else.else.else:
  %".40" = alloca [33 x i8]
  store [33 x i8] c"Both x and y are smaller than 0\0a\00", [33 x i8]* %".40"
  %".42" = getelementptr [33 x i8], [33 x i8]* %".40", i32 0, i32 0
  %".43" = call i32 (i8*, ...) @"printf"(i8* %".42")
  br label %"entry.endif.else.else.endif"
entry.endif.else.else.endif:
  br label %"entry.endif.else.endif"
entry.endif.endif.if:
  store i1 1, i1* %".49"
  br label %"entry.endif.endif.endif"
entry.endif.endif.else:
  %".53" = load i32, i32* %".5"
  %".54" = icmp slt i32 %".53", 10
  store i1 %".54", i1* %".49"
  br label %"entry.endif.endif.endif"
entry.endif.endif.endif:
  %".57" = load i1, i1* %".49"
  br i1 %".57", label %"entry.endif.endif.endif.if", label %"entry.endif.endif.endif.endif"
entry.endif.endif.endif.if:
  %".59" = alloca [40 x i8]
  store [40 x i8] c"At least one of x or y is less than 10\0a\00", [40 x i8]* %".59"
  %".61" = getelementptr [40 x i8], [40 x i8]* %".59", i32 0, i32 0
  %".62" = call i32 (i8*, ...) @"printf"(i8* %".61")
  br label %"entry.endif.endif.endif.endif"
entry.endif.endif.endif.endif:
  ret i32 0
}
```

Листинг 5.1. Типы данных

```
int printf(const char *format,...);
int main()
{
    float i=5.5666;
    printf("float type: i=%f\n", i);

    double i2 = 4.44435;
    printf("double type (with 3 nums after dot): i2=%.3f\n",i2);

    char j = 49;
    printf("49 as char: j=%c\n", j);
```

```
    int k = 5;
    printf("int type: k=%d\n", k);

    char s[5] = "5\t6\n";
    printf("Massive of chars s: %s", s);

    s[2] = 33;
    printf("Remove s[2] with 33 as char: %s", s);

    return 0;
}
```

Листинг 5.2. Файл промежуточного представления для листинга 5.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define i32 @"main"()
{
entry:
  %".2" = alloca double
  store double 0x40164432ca57a787, double* %".2"
  %".4" = load double, double* %".2"
  %".5" = alloca [18 x i8]
  store [18 x i8] c"float type: i=%f\0a\00", [18 x i8]* %".5"
  %".7" = getelementptr [18 x i8], [18 x i8]* %".5", i32 0, i32 0
  %".8" = call i32 (i8*, ...) @"printf"(i8* %".7", double %".4")
  %".9" = alloca double
  store double 0x4011c703afb7e910, double* %".9"
  %".11" = load double, double* %".9"
  %".12" = alloca [46 x i8]
  store [46 x i8] c"double type (with 3 nums after dot): i2=%.3f\0a\00", [46 x i8]* %".12"
  %".14" = getelementptr [46 x i8], [46 x i8]* %".12", i32 0, i32 0
  %".15" = call i32 (i8*, ...) @"printf"(i8* %".14", double %".11")
  %".16" = trunc i32 49 to i8
  %".17" = alloca i8
  store i8 %".16", i8* %".17"
  %".19" = load i8, i8* %".17"
  %".20" = alloca [18 x i8]
  store [18 x i8] c"49 as char: j=%c\0a\00", [18 x i8]* %".20"
  %".22" = getelementptr [18 x i8], [18 x i8]* %".20", i32 0, i32 0
  %".23" = call i32 (i8*, ...) @"printf"(i8* %".22", i8 %".19")
  %".24" = alloca i32
  store i32 5, i32* %".24"
  %".26" = load i32, i32* %".24"
  %".27" = alloca [16 x i8]
  store [16 x i8] c"int type: k=%d\0a\00", [16 x i8]* %".27"
  %".29" = getelementptr [16 x i8], [16 x i8]* %".27", i32 0, i32 0
  %".30" = call i32 (i8*, ...) @"printf"(i8* %".29", i32 %".26")
  %".31" = alloca [5 x i8]
  store [5 x i8] c"5\096\0a\00", [5 x i8]* %".31"
  %".33" = getelementptr [5 x i8], [5 x i8]* %".31", i32 0, i32 0
  %".34" = alloca [23 x i8]
  store [23 x i8] c"Massive of chars s: %s\00", [23 x i8]* %".34"
  %".36" = getelementptr [23 x i8], [23 x i8]* %".34", i32 0, i32 0
  %".37" = call i32 (i8*, ...) @"printf"(i8* %".36", i8* %".33")
  %".38" = getelementptr [5 x i8], [5 x i8]* %".31", i32 0, i32 0
  %".39" = getelementptr [5 x i8], [5 x i8]* %".31", i32 0, i32 2
  %".40" = load i8, i8* %".39"
  %".41" = trunc i32 33 to i8
  store i8 %".41", i8* %".39"
  %".43" = getelementptr [5 x i8], [5 x i8]* %".31", i32 0, i32 0
  %".44" = alloca [32 x i8]
  store [32 x i8] c"Remove s[2] with 33 as char: %s\00", [32 x i8]* %".44"
```

```
  %".46" = getelementptr [32 x i8], [32 x i8]* %".44", i32 0, i32 0
  %".47" = call i32 (i8*, ...) @"printf"(i8* %".46", i8* %".43")
  ret i32 0
}
```

Листинг 6.1. Функция, одномерный массив и цикл for

```
int printf(const char *format,...);
void array_print(int array[], int n)
{
    printf("array=[");
    for(int i=0;i<n;i++){
        printf(" %d ", array[i]);
    }
    printf("] and n=%d (number of elements)\n",n);
}

int main()
{
    int array[7] = {3,4,5,1,0,8,9};
    array_print(array, 7);
    return 0;
}
```

Листинг 6.2. Файл промежуточного представления для листинга 6.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define void @"array_print"(i32* %".1", i32 %".2")
{
entry:
  %".4" = alloca [8 x i8]
  store [8 x i8] c"array=[\00", [8 x i8]* %".4"
  %".6" = getelementptr [8 x i8], [8 x i8]* %".4", i32 0, i32 0
  %".7" = call i32 (i8*, ...) @"printf"(i8* %".6")
  %".8" = alloca i32
  store i32 0, i32* %".8"
  br label %"entryloop_do"
entryloop_do:
  br label %"entry.loop_cond"
entry.loop_cond:
  %".12" = load i32, i32* %".8"
  %".13" = icmp slt i32 %".12", %".2"
  br i1 %".13", label %"entry.loop_body", label %"entry.loop_end"
entry.loop_body:
  %".15" = load i32, i32* %".8"
  %".16" = getelementptr i32, i32* %".1", i32 %".15"
  %".17" = load i32, i32* %".16"
  %".18" = alloca [5 x i8]
  store [5 x i8] c" %d \00", [5 x i8]* %".18"
  %".20" = getelementptr [5 x i8], [5 x i8]* %".18", i32 0, i32 0
  %".21" = call i32 (i8*, ...) @"printf"(i8* %".20", i32 %".17")
  br label %"entry.loop_update"
entry.loop_end:
  %".27" = alloca [33 x i8]
  store [33 x i8] c"] and n=%d (number of elements)\0a\00", [33 x i8]* %".27"
  %".29" = getelementptr [33 x i8], [33 x i8]* %".27", i32 0, i32 0
  %".30" = call i32 (i8*, ...) @"printf"(i8* %".29", i32 %".2")
  ret void
entry.loop_update:
  %".23" = load i32, i32* %".8"
  %".24" = add i32 %".23", 1
```

```
  store i32 %".24", i32* %".8"
  br label %"entry.loop_cond"
}

define i32 @"main"()
{

entry:
  %".2" = alloca [7 x i32]
  store [7 x i32] [i32 3, i32 4, i32 5, i32 1, i32 0, i32 8, i32 9], [7 x i32]* %".2"
  %".4" = getelementptr [7 x i32], [7 x i32]* %".2", i32 0, i32 0
  call void @"array_print"(i32* %".4", i32 7)
  ret i32 0
}
```

Листинг 7.1. Двумерный массив, символ, строка и цикл for

```
int printf(const char *format,...);
void char_array_test()
{
    char c=33;
    printf("33 is %c\n", c);
    char* s = "Hello world";
    printf("%s\n", s);
}
int main()
{
    int array[3][4] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,11,12,13}
    };
    printf("array=\n");
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<4; j++){
            printf("%d\t", array[i][j]);
        }
        printf("\n");
    }
    char_array_test();
    return 0;
}
```

Листинг 7.2. Файл промежуточного представления для листинга 7.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define void @"char_array_test"()
{
entry:
  %".2" = trunc i32 33 to i8
  %".3" = alloca i8
  store i8 %".2", i8* %".3"
  %".5" = load i8, i8* %".3"
  %".6" = alloca [10 x i8]
  store [10 x i8] c"33 is %c\0a\00", [10 x i8]* %".6"
  %".8" = getelementptr [10 x i8], [10 x i8]* %".6", i32 0, i32 0
  %".9" = call i32 (i8*, ...) @"printf"(i8* %".8", i8 %".5")
  %".10" = alloca [12 x i8]
  store [12 x i8] c"Hello world\00", [12 x i8]* %".10"
  %".12" = getelementptr [12 x i8], [12 x i8]* %".10", i32 0, i32 0
```

```
  %".13" = alloca [4 x i8]
  store [4 x i8] c"%s\0a\00", [4 x i8]* %".13"
  %".15" = getelementptr [4 x i8], [4 x i8]* %".13", i32 0, i32 0
  %".16" = call i32 (i8*, ...) @"printf"(i8* %".15", i8* %".12")
  ret void
}
define i32 @"main"()
{
entry:
  %".2" = alloca [3 x [4 x i32]]
  store [3 x [4 x i32]] [[4 x i32] [i32 1, i32 2, i32 3, i32 4], [4 x i32] [i32 5, i32 6, i32
7, i32 8], [4 x i32] [i32 9, i32 11, i32 12, i32 13]], [3 x [4 x i32]]* %".2"
  %".4" = alloca [8 x i8]
  store [8 x i8] c"array=\0a\00", [8 x i8]* %".4"
  %".6" = getelementptr [8 x i8], [8 x i8]* %".4", i32 0, i32 0
  %".7" = call i32 (i8*, ...) @"printf"(i8* %".6")
  %".8" = alloca i32
  store i32 0, i32* %".8"
  br label %"entryloop_do"
entryloop_do:
  br label %"entry.loop_cond"
entry.loop_cond:
  %".12" = load i32, i32* %".8"
  %".13" = icmp slt i32 %".12", 3
  br i1 %".13", label %"entry.loop_body", label %"entry.loop_end"
entry.loop_body:
  %".15" = alloca i32
  store i32 0, i32* %".15"
  br label %"entry.loop_bodyloop_do"
entry.loop_end:
  call void @"char_array_test"()
  ret i32 0
entry.loop_update:
  %".43" = load i32, i32* %".8"
  %".44" = add i32 %".43", 1
  store i32 %".44", i32* %".8"
  br label %"entry.loop_cond"
entry.loop_bodyloop_do:
  br label %"entry.loop_body.loop_cond"
entry.loop_body.loop_cond:
  %".19" = load i32, i32* %".15"
  %".20" = icmp slt i32 %".19", 4
  br i1 %".20", label %"entry.loop_body.loop_body", label %"entry.loop_body.loop_end"
entry.loop_body.loop_body:
  %".22" = getelementptr [3 x [4 x i32]], [3 x [4 x i32]]* %".2", i32 0, i32 0
  %".23" = load i32, i32* %".8"
  %".24" = getelementptr [3 x [4 x i32]], [3 x [4 x i32]]* %".2", i32 0, i32 %".23"
  %".25" = load [4 x i32], [4 x i32]* %".24"
  %".26" = load i32, i32* %".15"
  %".27" = getelementptr [4 x i32], [4 x i32]* %".24", i32 0, i32 %".26"
  %".28" = load i32, i32* %".27"
  %".29" = alloca [4 x i8]
  store [4 x i8] c"%d\09\00", [4 x i8]* %".29"
  %".31" = getelementptr [4 x i8], [4 x i8]* %".29", i32 0, i32 0
  %".32" = call i32 (i8*, ...) @"printf"(i8* %".31", i32 %".28")
  br label %"entry.loop_body.loop_update"
entry.loop_body.loop_end:
  %".38" = alloca [2 x i8]
  store [2 x i8] c"\0a\00", [2 x i8]* %".38"
  %".40" = getelementptr [2 x i8], [2 x i8]* %".38", i32 0, i32 0
  %".41" = call i32 (i8*, ...) @"printf"(i8* %".40")
  br label %"entry.loop_update"
entry.loop_body.loop_update:
  %".34" = load i32, i32* %".15"
  %".35" = add i32 %".34", 1
  store i32 %".35", i32* %".15"
  br label %"entry.loop_body.loop_cond"
}
```

Листинг 8.1. Структура

```
int printf(const char *format,...);
struct Student
{
    int age;
    float gpa;
};
int main()
{
    struct Student student1;
    student1.age = 20;
    student1.gpa = 4.5;
    printf("Student age: %d\n", student1.age);
    printf("Student GPA: %.2f", student1.gpa);
    return 0;
}
```

Листинг 8.2. Файл промежуточного представления для листинга 8.1

```
; ModuleID = ""
target triple = "unknown-unknown-unknown"
target datalayout = ""

%"Student" = type {i32, double}

declare i32 @"printf"(i8* %".1", ...)

define i32 @"main"()
{
entry:
  %".2" = alloca %"Student"
  %".3" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".4" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".5" = load i32, i32* %".4"
  store i32 20, i32* %".4"
  %".7" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".8" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 1
  %".9" = load double, double* %".8"
  store double 0x4012000000000000, double* %".8"
  %".11" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".12" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".13" = load i32, i32* %".12"
  %".14" = alloca [17 x i8]
  store [17 x i8] c"Student age: %d\0a\00", [17 x i8]* %".14"
  %".16" = getelementptr [17 x i8], [17 x i8]* %".14", i32 0, i32 0
  %".17" = call i32 (i8*, ...) @"printf"(i8* %".16", i32 %".13")
  %".18" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 0
  %".19" = getelementptr %"Student", %"Student"* %".2", i32 0, i32 1
  %".20" = load double, double* %".19"
  %".21" = alloca [18 x i8]
  store [18 x i8] c"Student GPA: %.2f\00", [18 x i8]* %".21"
  %".23" = getelementptr [18 x i8], [18 x i8]* %".21", i32 0, i32 0
  %".24" = call i32 (i8*, ...) @"printf"(i8* %".23", double %".20")
  ret i32 0
}
```