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 Информатика и системы управления

КАФЕДРА Теоретическая информатика и технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

ПО КУРСУ КОНСТРУИРОВАНИЕ КОМПИЛЯТОРОВ

НА ТЕМУ:

«Оптимизация интерпретатора P-кода компилятора P5»

Студент _____ _____

подпись, дата *фамилия, и.о.*

Научный руководитель _____ _____

подпись, дата *фамилия, и.о.*

2022 г

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Существует два основных подхода к трансляции и выполнению программ: компиляция и интерпретация. При компиляции осуществляется перевод программы из языка S в язык T, которая сама написана на языке H и должна иметь возможность запускаться на некоторой машине, поддерживающей H. У компиляции существует ряд преимуществ:

1. Быстродействие – компилятор запускается один раз, генерируя код на языке, который машина может выполнять произвольное количество раз.
2. Зачастую проведение статического анализа программы, который может выявить ошибки на раннем этапе.
3. Возможность на этапе компиляции проводить оптимизации, в том числе под конкретную архитектуру машины.

Последнее преимущество связано и с самым главным недостатком компилятора: ориентированность программы на конкретную архитектуру процессора, а также операционную систему. Данный недостаток можно решить следующим образом: откомпилировать программу в какой-либо платформонезависимый промежуточный код с возможными оптимизациями и анализом программы, а затем выполнять с помощью различных виртуальных машин, которые могут выполняться на конкретной архитектуре процессора и типе ОС. Такого подхода придерживается и образованный в середине 90-х годов язык Java, использующий в своем составе компилятор и так называемую виртуальную машину JVM. Таким образом, скомпилированная один раз программа становится универсальной для каждой системы, которую поддерживает JVM. Однако данный подход не был первым в своем роде: аналогичная система использовалась в компиляторе Паскаля P5, образованного в 80-х годах. Ключевой особенностью является то, что языком реализации и интерпретатора, и компилятора является Паскаль. Поэтому можно самоприменять компилятор и выполнять раскрутку, добавляя тем самым новые возможности в компилятор. То есть можно применить бинарную версию компилятора к его исходному тексту, получив тем самым промежуточный P-

code, и выполнить его бинарной версией интерпретатора, передав ей на вход измененный исходный текст. При этом получится раскрученный на один шаг компилятор в промежуточном представлении. Однако последний шаг выполняется довольно долго: в среднем 5 минут на разных машинах.

Целью данной курсовой работы является оптимизация и реализация на более современном языке интерпретатора P-кода компилятора P5.

1. Обзор P5 машины

Архитектура оригинального интерпретатора похожа на архитектуру компьютера: есть оперативная память и процессор [1]. Интерпретатор имеет определенный набор команд (P-code), которые управляют работой со стеком, с адресами в оперативной памяти, с вызовом пользовательских и стандартных процедур, с различными операциями над переменными и т.д. Все эти инструкции исполняются на так называемой P5 машине. У нее имеется набор регистров:

1. PC (program counter) – указатель на текущую инструкцию
2. SP (stack pointer) – указатель на вершину стека
3. MP (mark stack pointer) – указатель на базовый адрес фрейма стека
4. NP (new pointer) – указатель конца кучи
5. EP (extreme pointer) – указатель конца фрейма стека

Смысл каждого станет понятен по ходу изложения.

1.1. Раскладка памяти

Рисунок 1 отображает раскладку оперативной памяти в интерпретаторе.

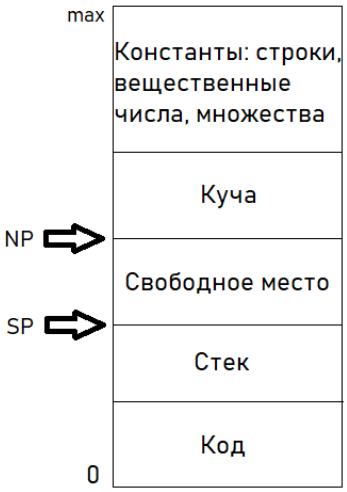

Рисунок 1 - Раскладка оперативной памяти

Все данные программы, кроме регистров, хранятся в оперативной памяти. Секция констант заполняется на предварительном этапе чтения

псевдокода и ассемблирования строкового представления команд в их коды. Примером такой команды может служить lca 3 ‘str’, которая загружает строку ‘str’ в секцию констант и генерирует соответствующий код инструкции 56 с параметром, являющимся адресом, куда была записана строка.

Куча заполняется по мере выполнения программы и управляется инструкциями new и dsp – выделение памяти с нужным размером и освобождение памяти по адресу соответственно. Если новую ячейку памяти нельзя выделить в текущих границах кучи, то нижняя граница смещается на необходимое значение и обновляется регистр NP.

Стек включает в себя локальные переменные и фрейм стека, который создается при вызове процедуре или функции.

Вся оперативная память ограничена определенным числом max, поэтому секция констант и куча растет вниз по мере заполнения, а секция с кодом и стеком – вверх. При этом проверку на переполнение памяти можно осуществлять с помощью двух регистров – NP должен быть строго больше SP.

1.2. Фреймы стека

Фрейм стека процедуры или функции подразделяется на две части: служебная (mark) и локальная (local), которые следуют друг за другом. В служебной части хранится различная метаинформация о контексте вызова и необходимая для возврата из функции. Данная часть изображена на рисунке 2.

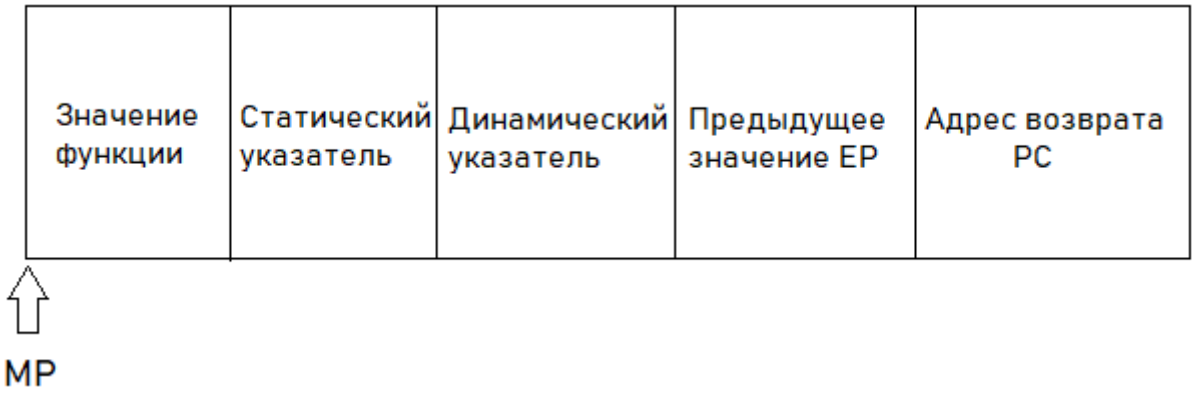

Рисунок 2 – Служебная часть фрейма

Ранее объявленный регистр MP носит следующую функцию: является базовым адресом текущего стека фрейма, прибавив к которому смещение можно получить адрес нужной переменной. Также при возвращении из функции он используется для того, чтобы освободить память. При этом новое значение SP будет равняться данному MP.

Значение функции – использующаяся только в функциях область, в которой будет храниться возвращаемое значение для вызывающего контекста. Эта область должна занимать максимальное значение, которое может быть возвращено из функции, так как контроля за размером этой части памяти никакая инструкция не предоставляет, хотя данная информация должна быть известна на этапе компиляции. Поскольку в P-коде функция может возвращать только значения атомарных типов: bool, real, integer, char, address, - берется размер самого большого, то есть размер типа real – 8 байт.

Динамический указатель – указатель на предыдущий фрейм. Используется для того, чтобы обновить значение регистра MP при возврате в вызывающий контекст.

Статический указатель – указатель на базовый адрес фрейма (MP) функции, которая синтаксически включает в себя вызываемую функцию. Используется как базовый адрес для обращения к переменным во внешних областях. Можно заметить, что такой фрейм всегда будет в памяти, поскольку функция может вызываться либо своим непосредственным родителем (тогда динамический указатель совпадает со статическим), либо своим дочерним элементом любого уровня вложенности, в цепочке вызовов которых обязательно будет она сама, потому что такое возможно только при рекурсии. Эти два случая доказывают утверждение. Стоит отметить, что если подходящих под определение фреймов в цепочке вызовов несколько, выберется последний. Это необходимо, так как при вызове функции ее локальные переменные должны принимать значение по умолчанию, иными словами сбрасываться, что должно отражаться в дочерних функциях, имеющих доступ к данным переменным.

Предыдущее значение EP – аналогично динамическому указателю, используется для обновления значения соответствующего регистра при выходе из процедуры.

Адрес возврата – обновление указателя на инструкцию (PC) при возврате из процедуры.

Структура локальной части фрейма изображена на рисунке 3.

Рисунок 3 – Локальная часть фрейма

Она состоит из трех секций:

1. Параметры – заполняется вызывающей функцией и хранит в себе аргументы вызываемой функции. Может содержать ссылки на значения или сами значения.
2. Локальные переменные – переменные определенные в секции var после названия и параметров и перед телом функции. К ним также могут обращаться все дочерние функции и процедуры.
3. Непосредственно стек – используется для хранения и вычисления промежуточных значений.

После различных операций на стеке регистр SP обновляется – либо уменьшатся, либо увеличивается. Например, инструкция ‘ldci 10’ загружает на стек целочисленную константу 10 и увеличивает SP на 4 – размер Integer.

Поскольку во время компиляции известен нужный размер для локального стека функции, логично было бы где-то эту информацию хранить. Для этого используется регистр EP. Он позволяет не делать при каждом увеличении SP проверку на переполнение памяти, когда SP >= NP (указатель на

кучу). Другая полезная возможность заключается в том, что он указывает на место, где можно разместить следующий фрейм.

1.3. Вызов процедур и функций

Кратко рассмотрим инструкции, связанные с вызовом процедур и функций. Их четыре: MST p, CUP p L i, ENTS L I, ENTE L i.

Инструкция MST аллоцирует память под служебную часть фрейма и устанавливает значения следующих секций: статический и динамический указатели и предыдущее значение EP. Параметр p размером 1 байт определяет уровень вложенности вызываемой процедуры. Он используется для вычисления статического указателя. По сути это то, сколько раз нужно перейти по цепочке вызовов назад для нахождения значения MP, который и определяет нужный фрейм. То есть если аргумент 0, то значение равно значению MP вызывающего контекста, если 1 – то значению статического указателя вызывающего контекста.

Так как после выполнения инструкции MST была полностью выделена память под служебный стек, можно загружать параметры для процедуры обычными инструкциями для заполнения стека, например, ранее упомянутой ldc.

После того как были загружены параметры (при наличии), вызывается инструкция CUP. Она заполняет адрес возврата в уже выделенную служебную часть стека, поскольку это последняя инструкция перед вызовом новой функции, а, значит, номер следующей инструкции и есть адрес возврата. Также функция принимает параметр p, который задает количество байт, занимаемых аргументами функции. Этот параметр нужен, чтобы найти адрес MP. Параметр L i, где i – положительное целое число, номер метки в программе, куда нужно перенести управление. Таким образом, обновляется регистр PC.

Далее идут две инструкции уже в новой процедуре:

1. ENTS с параметром метки, по адресу которой хранится размер секции локальных переменных в байтах плюс размер служебного стека – 32 байта.

Соответственно, данная область инициализируется нулями. Также получается актуальное значение указателя на вершину стека.

2. ENTE с параметром метки, по адресу которой хранится максимальный размер локального стека данного фрейма. Обновляется регистр EP.

На листинге 1 изображен фрагмент программы с вызовом функции, представленной меткой 3 и в которую передается целочисленный аргумент 10 и вещественный – 3.14. Параметр инструкция CUP говорит о том, что размер секции локальных переменных 12, что равно размеру переменной целочисленного типа – 4 байт, и размеру вещественной – 8 байт. Тоже самое подтверждает инструкция ENTS с параметром 44 = 32 (размер служебного стека) + 12.

```
l 3
ents l   4
ente l   5
.
.
.
l   4= 44
l   5= 34
.
.
.
mst  0
ldci 10
ldcr 3.14
cup  12 l 3
```

Листинг 1 – Пример вызова функции

1.4. Формат инструкций

Инструкции имеют следующий общий формат:

1. Название инструкции
2. Параметр P – 1 байт (при наличии)
3. Параметр Q – 4 байта (при наличии)

Большинство инструкций состоят из 3 букв, но многие из них рассчитаны на разные типы данных, поэтому к ним прибавляется суффикс, обозначающий тип данных:

1. Address – a
2. Boolean – b
3. Character – c
4. Integer – i
5. Real – r
6. Set - s

Параметр P обычно отвечает за уровень вложенности процедуры и используется вместе со статическим указателем в инструкциях, где нужно вычисление базового адреса, например, ‘lod p q’, которая загружает на стек значение по адресу base_addr(p) + q, то есть вычисляя базовый адрес на основе параметра p и прибавляя к нему смещение, представленное параметром q. Инструкция lod может обрабатывать разные типы данных, например, для загрузки на стек вещественного числа будет использоваться процедура lodr.

Многие инструкции также и управляют стеком, например, бинарные операции берут два значения на вершине стека и перезаписывают одним – результатом операции. Примером может служить инструкция ADI, которая складывает два целочисленных значения на вершине стека и записывает результат туда же.

2. Разработка интерпретатора и оптимизаций

Различают два подхода к реализации и проектированию виртуальной машины:

1. Динамическая генерация кода
2. Непосредственная интерпретация

Вкратце рассмотрим каждый из них.

2.1. Динамическая генерация кода

Динамическая генерация кода – это прием программирования, заключающийся в том, что фрагменты кода порождаются и запускаются непосредственно во время выполнения программы [2]. Целью динамической генерации кода является использование информации, доступной только во время выполнения программы, а именно – тип процессора. JIT-компилятор транслирует некоторое промежуточное представление в инструкции процессора и дает ему команду на их выполнение. Наличие информации о типе процессора также позволяет делать ряд платформозависимых оптимизаций, например, векторизация. Задачей же разрабатываемого интерпретатора является генерация данного промежуточного представления. Для реализации данного решения можно использовать, например, технологию JIT компилятора LLVM.

Также есть другой подход – генерация кода на каком-либо более высокоуровневом языке, нежели чем промежуточное представление. Данный подход является более простым с точки зрения реализации, так как не нужно работать с низкоуровневым промежуточным представлением. При этом время выполнения самой сгенерированной программы сравнимо с временем выполнения программы с использованием JIT-компилятора, поскольку в обоих случаях инструкции интерпретатора в конечном итоге транслируются в команды, которые выполняются на процессоре, а не непосредственно с помощью виртуальной машины реализуемого интерпретатора. Однако присутствуют дополнительные расходы на компиляцию программы, зависящие от используемого языка и его компилятора. Тем не менее, большим плюсом

является возможность повторного использования программы в виде исполняемого файла.

При большом количестве плюсов динамической генерации кода и ее соответствии цели курсовой работы реализация такого подхода является нетривиальной. Если рассматривать компилятор JIT LLVM как инструмент для реализации динамической генерации кода, то стоит задача преобразования инструкций P-кода в промежуточное представление LLVM. Проблема заключается в том, что P-код не имеет определения переменных, которые есть в LLVM. Можно связать адреса переменных и их тип с переменной LLVM. И это просто реализуется для константных адресов, например, в инструкции lca, которая записывает в константую область памяти строку-аргумент, адрес которой в течение времени выполнения программы не меняется.

Однако становится неочевидно, как преобразовывать инструкции по работе со стеком, который изменяется динамически, в инструкции LLVM. Для этого понадобилось бы иметь какую-либо возможность контролировать стек внутри сгенерированного IR LLVM, то есть необходимо было бы реализовать какую-либо библиотеку времени выполнения.

По этим причинам было решено создать собственную виртуальную машину для выполнения P-кода в соответствии с его спецификацией, то есть реализовать непосредственную интерпретацию.

2.2. Непосредственная интерпретация

Под непосредственной интерпретацией понимается чтение каждой строчки программы и ее выполнение в соответствии с кодом инструкции без генерации дополнительного кода. Для реализации данной задачи необходимо спроектировать интерпретатор. Глобально он будет состоять из двух взаимосвязанных частей:

1. Ассемблер
2. Виртуальная машина

Задачей ассемблера будет являться преобразование текстового представления программы на входе в набор бинарных инструкций и констант.

Таким образом, во время работы ассемблера будет заполняться секция кода и констант, изображенные на рисунке 1. Стоит отметить, что работа интерпретатора в режиме сопрограмм, с переменным запуском прохода ассемблера и виртуальной машины, невозможна, так как в коде существуют переходы по меткам, другими словами, процедурам, определенным в дальнейших частях программы. Задачей ассемблера будет связать все эти метки с указателем на инструкцию их непосредственного начала (похожим образом работает линкер в компиляторе gcc). Таким образом, вызовы ассемблера и виртуальной машины будут реализованы в строгом порядке друг за другом.

Секция кода будет состоять из инструкций и их параметров в бинарном формате. Максимум, который может занимать инструкция, 6 байт. В них включается 1 байт на код инструкции (всего их 175, поэтому 1 байта будет достаточно), 1 байт на параметр P (обычно он отвечает за уровень вложенности процедуры для поиска нужного фрейма стека) и 4 байта на параметр Q (обычно адрес). Возможна ситуация, когда параметра P и/или Q в инструкции нет. В таком случае для, например, вывода отладочной информации о содержимом области памяти с кодом необходимо хранить об этом информацию. Параметр Q также может использоваться для хранения различных констант с размером меньшим или равным 4 байтам. По сути это все атомарные типы данных, кроме типа real, так как он занимает 8 байт. Для сложных типов (строк и множеств) и типа real в качестве аргумента необходимо хранить адрес (4 байта), указывающий на расположение значения в секции констант. Например, инструкция ‘lca 2 ‘ab’’ при расположении строки по адресу 160000 преобразуется в последовательность байт “0x38 0x00 0x71 0x02 0x00”, если в архитектуре процессора, на котором запущена программа, используется порядок байт от младшего к большему - little endian. А, например, инструкция ‘ldci 10` преобразуется в последовательность “0x7B 0x0A 0x00 0x00 0x00”.

Каждая инструкция хранится друг за другом для экономии места. Это не делает возможным быстрое обращение по индексу, однако это и не нужно, так как все инструкции выполняются последовательно.

Входном файл с промежуточным представлением имеет простой формат: на каждой строке указана ровно одна инструкция или отладочная информация. Также возможно оставлять комментарии. Тип строки определяется по ее начальному символу:

1. i – комментарий, игнорируем оставшуюся часть.
2. l – метка, имеет два типа в зависимости от следующего символа.
    a. Символ = означает, что метка равна целочисленному значению, следующему за ней.
    b. Символ пробел означает, что метка указывает на текущую инструкцию, как свое начало.
3. q – конец программы.
4. : - отладочная информация, содержащая номер строки в исходном коде программы на Паскале, соответствующей следующей далее инструкции.
5. Пробел – означает, что далее в строке содержится какая-либо инструкция из заданного набора.

Таким образом, формат позволяет использовать простой лексический анализ. Следует уточнить, что P-код спроектирован создателями так, что есть ровно 2 строки с обозначением конца программы (символ q). Первая относится непосредственно к окончанию пользовательской программы. После нее идет три инструкции, которые есть в каждой программе: mst, cup, stp. Как было сказано ранее, они отвечают за вызов функции, а именно пользовательской программы – тела секции program, подготавливая самый первый по счету служебный фрейм стека.

Также в наборе инструкций есть специальная инструкция csp, после которой идет строка, обозначающая стандартную процедуру. В основном они отвечают за работу с файлами, математическими функциями и кучей. Сами

авторы компилятора признают, что разграничение между стандартными процедурами и просто инструкциями довольно размыто. Например, существует инструкция eof, обозначающая проверку на конец файла, хотя логичнее бы было сделать ее стандартной процедурой.

Таким образом, ассемблер передает бинарные данные с инструкциями и константами на вход виртуальной машине. Задача виртуальной машины заключается в непосредственном выполнении подготовленного на предыдущем шаге кода. Код выполняется последовательно, начиная с нулевого индекса, то есть PC = 0 изначально. При встрече условных и безусловных переходов (меток) происходит изменение регистра PC на соответствующее параметру инструкции значение адреса в массиве кода.

2.3. Возможные оптимизации

Проанализировав исходный код интерпретатора P5 можно заметить, что поиск кода инструкции при ассемблировании происходит линейно после ее прочтения из входного файла. Фрагмент исходного кода представлен на листинге 2.

```
procedure getname;
      var i: alfainx;
      begin
        if eof(prd) then errorl('unexpected eof on input  ');
        for i := 1 to maxalfa do word[i] := ' ';
        i := 1; { set 1st character of word }
        while ch in ['a'..'z'] do begin
          if i = maxalfa then errorl('Opcode label is too long ');
          word[i] := ch;
          i := i+1; ch := ' ';
          if not eoln(prd) then read(prd,ch); { next character }
        end;
        pack(word,1,name)
      end; (*getname*)

getname;
while (instr[op]<>name) and (op < maxins) do op := op+1;
```

Листинг 2 – Фрагмент с линейным поиском кода инструкции

Поскольку инструкций довольно много и они представлены в виде строк, линейный поиск является неоптимальным решением. Вместо него можно использовать отображение, снизив алгоритмическую сложность с O(n) до O(log n) при использовании бинарного дерева или и вовсе до O(1) в среднем в случае хеш-отображения.

Помимо этого можно рассмотреть способ хранения множеств. В оригинальном интерпретаторе множество сериализуется из внутреннего представления в программе в массив байт каждый раз, когда нужно записать значение на стек или в другое место в памяти. И, аналогично, десериализуется при чтении. Фрагмент исходного кода сериализации представлен на листинге 3.

```
procedure putset(a: address; s: settype);
var r: record case boolean of
          true:  (s: settype);
          false: (b: packed array [1..setsize] of byte);
       end;
    i: 1..setsize;
begin
   r.s := s;
   for i := 1 to setsize do store[a+i-1] := r.b[i]
end;
```

Листинг 3 – Фрагмент с сериализацией множества

Данная процедура записывает множество по адресу аргументу. Вне зависимости от размера она всегда записывает 32 байта в память, что бывает излишне. Для оптимизации можно использовать следующий подход: хранить все множества во внутреннем представлении в программе, а при необходимости записи/чтения в память записывать лишь их идентификатор в едином хранилище – целочисленное значение. Это позволит эффективно обращаться к множества и выполнять операции над ними. Также можно предусмотреть простой сборщик мусора с помощью подсчета ссылок. Можно легко контролировать записи на стек всех идентификаторов и инкрементировать счетчик при очередной записи, а когда при очередной операции pop счетчик равен нулю, удалять из единого хранилища множество, соответствующее данному идентификатору.

3. Реализация

Языком реализации был выбран C++ в силу своего быстродействия, наличия удобных контейнеров и возможности использовать функции-шаблоны, которые помогут не писать повторяющийся код. В качестве системы сборки используется CMake.

Прежде всего, необходимо определиться с типами данных, которые используются в промежуточном представлении и должны использоваться во всех частях программы во избежание ошибок чтения/записи данных не того размера. Для этого удобно определить заголовочный файл, в котором будут заданы псевдонимы, имитирующие типы данных в Паскале. Фрагмент данного файла представлен на листинге 4.

```
namespace P5 {
    typedef int addr_t;  //q
    typedef unsigned char ins_t; //opcode
    typedef unsigned char lvl_t; //p
    typedef unsigned char set_el_t;
    typedef short set_t;
    typedef std::vector<unsigned char> store_t;
    typedef double real_t;
    typedef bool bool_t;
    typedef char char_t;
    typedef int int_t;

    static int addr_size = sizeof(P5::addr_t);
    static int int_size = sizeof(P5::int_t);
    static int char_size = sizeof(P5::char_t);
    static int bool_size = sizeof(P5::bool_t);
    static int set_size = 32;
    static int max_store = 16777216;
}
```

Листинг 4 – Определение псевдонимов для типов Паскля

Таким образом, получится абстрагироваться от конкретных типов данных языка C++ и использовать лишь псевдонимы при работе с типами Паскаля, что будет поддерживать согласованность.

Для чтения инструкций из исходного файла был реализован класс Lexer. Главным методом в нем является шаблонная функция get с единственным аргументом – типом, который необходимо прочитать и вернуть, например, real.

Данные читаются с помощью объекта класса ifstream, который поддерживает чтение разных типов данных с помощью оператора <<. Поскольку метод get имеет одинаковую структуру для разных типов данных, в паре с ifstream было логично использовать шаблон. Однако пришлось использовать специализацию для нескольких типов данных: строки и символа. По умолчанию оператор << ifstream пропускает все пробельные символы, что не подходит для char. Реализация шаблонного метода и его специализации для char изображена на листинге 5.

```
template<typename T> T Lexer::get() {
	T val;
	strm >> std::skipws >> val;
	return val;
}
template<>
char Lexer::get<char>() {
	char val = ' ';
	if (strm.peek() == '\n') {
		return val;
	}
	strm >> std::noskipws >> val;
	return val;
}
```

Листинг 5 – Реализация метода get

Для ассемблирования входного кода был реализован класс Assembler. Как было сказано в предыдущей главе, для поиска кодов инструкций должно использоваться отображение. В стандартной библиотеке языка C++ можно использовать контейнер unordered_map, реализующий хеш-отображение из имени инструкции в код инструкции. Для вывода отладочной информации также хранится длина параметра Q и bool значение, обозначающее факт наличия параметра P. Для удобства заполнения отображения был реализован макрос, принимающий 4 аргумента, название инструкции, код и 2 параметра для отладочной информации. Аналогичный макрос реализован для стандартных процедур. Они вместе с примером использования представлены на листинге 6.

```
#define ins_op(name_p, opcode, has_p_p, q_len_p) instr[name_p] = opcode;
op_codes[opcode] = {.name = (name_p), .has_p = (has_p_p), .q_len = (q_len_p)};
```

```
#define sp_table_op(name, opcode) sp_table[name] = opcode;

ins_op("lodi", 0, true, P5::int_size)
sp_table_op("sin", 14)
```

Листинг 6 – Макросы для заполнения хеш-таблиц

После инициализации таблиц происходит непосредственный запуск ассемблирования кода методом generate. Стоит отметить, что данный метод вызывается два раза, потому что, как было сказано в прошлой главе, сначала в файле находится основная программа, а дальше преамбула, состоящая из 3 инструкций. Для первого вызова регистр PC равен 9, это как раз смещение, перед которым должны расположиться 3 инструкции преамбулы: mst, cup, stp размером 2, 6 и 1 байт соответственно. Для второго вызова PC уже будет равен 0. Далее в соответствии с описанием в прошлой главе происходит преобразование текстового представления инструкции в бинарное.

Для реализации чтения/записи по адресам был реализован ряд функций шаблонов, представляющих собой код, умеющий обрабатывать разные типы данных. Реализация функций представлена на листинге 7.

```
void put_val_to_addr_by_ptr(P5::store_t &store, P5::addr_t addr, unsigned char
*bytes, int size) {
     for (int i = 0; i < size; i++) {
           store[addr+i] = bytes[i];
     }
}

void get_val_at_addr_by_ptr(P5::store_t &store, P5::addr_t addr, unsigned char
*bytes, int size) {
     for (int i = 0; i < size; i++) {
           bytes[i] = store[addr+i];
     }
}

template<typename T>
void put_val_to_addr(P5::store_t &store, P5::addr_t addr, T v) {
     auto *bytes = (unsigned char*)&v;
     int size = sizeof(T);
     put_val_to_addr_by_ptr(store, addr, bytes, size);
}

template<typename T>
T get_val_at_addr(P5::store_t &store, P5::addr_t addr) {
```

```
    int size = sizeof(T);
    unsigned char bytes[size];
    for (int i = 0; i < size; i++) {
          bytes[i] = store[addr+i];
    }
    return *(T*)bytes;
}
```

Листинг 7 – Функции для записи/чтения по адресам

Шаблонные функции используют в своей реализации функции без шаблона, которые также используются в некоторых частях кода.

Для реализации связки меток с указателями на их первую инструкцию был реализован класс LabelTable. Он имеет две операции Lookup и Update. Первая используется для получения адреса по метке, а вторая для обновления адреса метки. Основным сценарием использования этого класса является обращение к еще неинициализированной метке. С каждой меткой связано две переменные bool defined и addr pc. Первая хранит информацию о том, определена ли метка, вторая указывает на начальную инструкцию, если метка определена. В ином случае указывает на тот адрес, куда нужно записать значение pc. Понятно, что если существует несколько обращений к неопределенной метке, одного поля pc недостаточно. Для разрешения этой ситуации в каждой ячейке, куда нужно записать значение pc, хранится адрес следующей такой ячейки, образуя некий список, который заканчивается числом -1. При обновлении адреса метки выполняется вышеописанная последовательность действий. Рисунок 4 демонстрирует пример, как метка 1 используется в качестве объявленной дальше в коде. Когда ассемблеру встретится определение метки, вызывается метод Update, состояние ‘not defined’ заменится на ‘defined’ и обновленное значение pc будет распространяться по 3 ячейкам памяти с адресами: 680, 90 и 53.

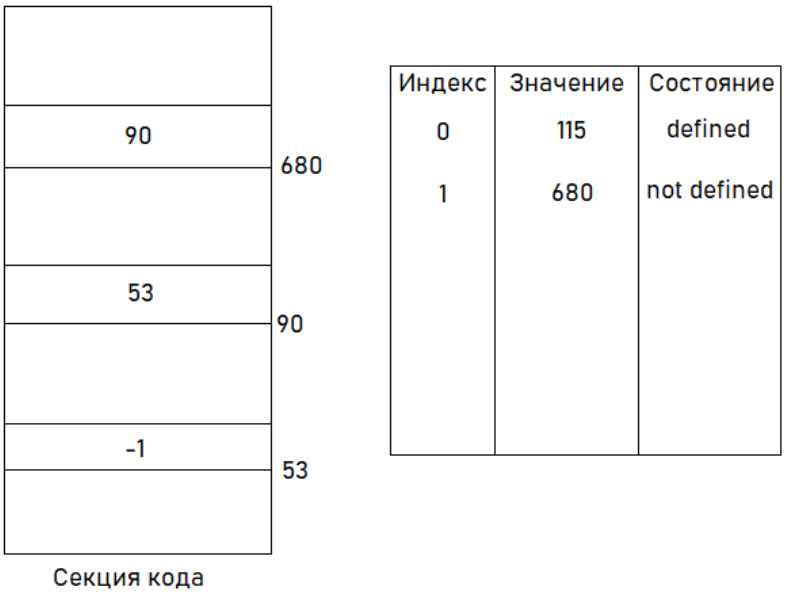

Рисунок 4 – Пример с таблицей меток

После ассемблирования создается объект класса Interpreter, в конструктор которого передаются все нужные данные из ассемблера: массив оперативной памяти, адрес окончания секции констант и секции кода. После этого вызывается метод run, который начинает непосредственную интерпретацию команд. В классе интерпретатора также много шаблонных методов, например, по взаимодействию со стеком: push_stack и pop_stack. Данные методы имеют 3 специализации: для bool_t, char_t и set_t. Дело в том, что типы char_t и bool_t хотя и занимают один байт памяти, но хранятся на стеке как целочисленные значения типа int_t. Данное требование накладывается самим P-кодом, который ожидает на стеке тип именно integer. Именно из-за этого, например, инструкцию ord, переводящую char в int, можно игнорировать при выполнении команд.

Как было сказано в предыдущей главе, в памяти можно хранить индексы множеств вместо их сериализованного представления. Из-за этого необходима специлизация шаблонных методов push_stack и pop_stack, так как необходимо проинкрементировать указатель на вершину стека SP не на размер идентификатора, а на размер который ожидает P-код: 32 байта. Помимо этого, был реализован класс SetStorage, с помощью которого можно создать, получить, увеличить или уменьшить счетчик ссылок. Метод создания возвращает целочисленный идентификатор, который генерируется простым

инкрементом предыдущего идентификатора. Данное значение потом нужно записать в ячейку памяти и проинкрементировать счетчик ссылок.

Выделение памяти в куче реализовано следующим образом: память в занятом кучей пространстве поделена на части, каждая из которых описывается метаинформацией, за которой непосредственно следует непрерывный блок памяти. Метаинформация представляет собой целое число и определяет, занят ли данный блок памяти, и его размер. Размер равен модулю числа, а критерий свободной памяти определяется знаком числа. Поэтому для выделения новой памяти нужно линейно пройти по всем блокам и найти такой свободный блок, чей размер больше запрашиваемого. Если такой не нашелся, то уменьшаем регистр NP на запрашиваемый размер и создаем блок памяти.

4. Тестирование

После реализации приложение стоит задача тестирования. Она включает в себя проверку корректности выполнения программ на реализованном интерпретаторе с помощью сравнения с оригинальной версией и проведение замеров времени работы программы со сравнением результатов.

Программы запускались на Windows в среде WSL. Корректность выполнения программ выполнялась на 4 программах разной степени сложности: от простого вывода в консоль до запуска ассемблированной версии компилятора PCOM. Были протестированы стандартные процедуры, циклы, пользовательские функции, чтение из файла, работоспособность реализованных множеств. Были проанализированы и сравнены результаты работы программ, выполняемых на созданном и оригинальном интерпретаторе. Результаты совпали. Примеры тестовых программ изображены на листингах в приложении А.

Производительность замерялась с помощью утилиты time. Запускался интерпретатор, на вход которому подавался компилятор PCOM в промежуточном представлении, который компилировал себя же в представлении языка Паскаль. Имели место следующие результаты:

1. Оригинальный интерпретатор: 3 минуты 2 секунды
2. Реализованный интерпретатор, скомпилированный в режиме Debug: 6 минут 14 секунд
3. Реализованный интерпретатор, скомпилированный в режиме Release: 0 минут 47 секунд

Как видно, режим Debug не показал удовлетворительных результатов. Это произошло, потому что в таком режиме компиляции сохраняется множество информации для отладки, что увеличивает время работы. При этом режим Release без отладочной информации и с оптимизациями показал хороший результат, уменьшив время работы программы более чем в 3 раза по сравнению с оригинальной реализацией.

ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы был реализован интерпретатор P5 с оптимизациями на языке C++. Получившаяся реализации ускорила самоприменение компилятора P5 более чем в 3 раза по сравнению с оригинальным интерпретатором.

Также были приобретены и улучшены навыки в области лексических анализаторов, компиляторов и интерпретаторов. Были изучены способы интерпретации программ, вызова подпрограмм и построения фреймов стека.

Дальнейшая разработка может быть направлена на:

1. добавление более тщательной обработки ошибок на этапе лексического анализа
2. добавление оставшихся нереализованными стандартных процедур
3. рассмотрение возможности использования JIT LLVM

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Steven Pemberton, Martin Daniels. Pascal Implementation [Электронный ресурс] - URL: https://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html (дата обращения: 2.Ноябрь.2022)
2. Лекция по динамической генерации кода [Электронный ресурс] - URL: https://intuit.ru/studies/courses/1206/89/lecture/28315 (дата обращения: 30.Ноябрь.2022)

ПРИЛОЖЕНИЕ А

На листинге А.1 изображена программа на языке Паскаль, тестирующая основные операции над множествами.

```
program hello(input, output);
var
  i, l: integer;
  s, s2, s3, s4: set of 0..255;
  elem: 0..255;
  arr: array[1..3] of 0..255;
begin
  writeln('Hello', ord(true));
  read(l);
  for i := 1 to l do begin
    s := s + [i];
  end;
  writeln;

  writeln('difference');
  if 4 in s then writeln('4 in set');
  s := s - [4];
  if not (4 in s) then writeln('4 not in set');

  writeln;
  writeln('intersection');
  arr[1] := 1;
  arr[2] := 8;
  arr[3] := 35;
  for i := 1 to 3 do s2 := s2 + [arr[i]];
  s := s * s2;
  for i := 1 to 3 do begin
    if arr[i] in s then begin
      write(arr[i]:1);
      write(' ');
    end;
  end;
  writeln;
  writeln;
  writeln('union');

  s2 := [1, 2, 4, 123];
  s3 := [1, 2, 5, 123];
  s4 := s2 + s3;
  if 1 in s4 then writeln('1');
  if 4 in s4 then writeln('4');
  if 2 in s4 then writeln('2');
  if 5 in s4 then writeln('5');
```

```
  if 23 in s4 then writeln('23');
  if 123 in s4 then writeln('123');

end.
```

Листинг А.1 – Тестовая программа 1

На листинге А.2 изображена программа, выводящая содержимое файла prd в консоль.

```
program readfile(input, output, prd);
var
  c: char;
begin
  reset(prd);
  while not eof(prd) do
  begin
    if eoln(prd) then
    begin
      read(prd, c);
      writeln;
    end
    else
    begin
      read(prd, c);
      write(c);
    end;
  end;

end.
```

Листинг А.2 – Тестовая программа 2