Министерство образования и науки 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 компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:
«Разработка компилятора FALSE для платформы DOS (процессор 8086)»

Студент _____ _____ _____
(Группа) (Подпись, дата) (И.О.Фамилия)

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

2017 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Большинство современных решений в IT-индустрии оперируют довольно высоким уровнем абстракции, однако зачастую возникает необходимость реализации низкоуровневых решений, например, компиляторов. Показать возможные подходы к задачам такого типа мы постараемся в рамках данного курсового проекта.

Объектом исследования являются вопросы теории трансляции и генерации кода. *Предмет исследования* – компилятор стекового языка программирования FALSE *Практическое значение* выбранной темы определяется возможностью использования результатов для написания программ с помощью языка FALSE.

Цель работы — разработка компилятора стекового языка программирования FALSE для платформы MS-DOS наименьшего размера. В ходе работы выполняются следующие задачи: изучение аналогичных решений, проектирование архитектуры и подходов к реализации компилятора; реализация компилятора языка FALSE с помощью языка ассемблера, тестирование и отладка готовой версии, оптимизация.

1. ОБЗОР ЯЗЫКА FALSE ИСХОДНОГО КОМПИЛЯТОРА ДЛЯ АРХИТЕКТУРЫ 8086

1.1 История создания и краткое описание языка FALSE

FALSE – эзотерический язык программирования, придуманный Ваутером ван Оортмерсеном в 1993 году. Сам автор утверждал, что преследовал две цели, создавая FALSE – разработать язык, синтаксис которого выглядел бы шифровкой и в то же время, компилятор которого имел бы минимально возможный размер. [1] Так Ваутер ван Оортмерсен создал первый компилятор языка FALSE, написанный на языке ассемблера для процессора 68000, исходный код которого занимал всего лишь 1024 байта. В том же году была выпущена первая спецификация языка. [2] Опираясь на спецификацию, опишем основные функции языка FALSE.

FALSE унаследовал способ вычисления выражения от стекового языка программирования Forth. Таким образом, все элементы языка определяются тем, как они взаимодействуют со стеком. По факту, программу на языке FALSE можно рассматривать как поток символов, которые каким-то образом манипулируют стеком. Для описания команд языка мы будем пользоваться нотацией (<pop> - <push>). В первых угловых скобках будут записаны операнды, удаляемые со стека, а в правых – добавляемые.

Итак, язык FALSE предоставляет некоторые элементарные математические функции.

- «+» – сумма двух верхних операндов стека (<a, b> - <b+a>);
- «-» – разность двух верхних операндов стека (<a, b> - <b-a>);
- «*» – произведение двух верхних операндов стека (<a, b> - <b*a>);
- «/» – целая часть от частного двух верхних операндов стека (<a, b> - <[b/a]>);
- «_» – унарный минус (<a> - < -a >).

Помимо этого, спецификация описывает также операции сравнения.

◆ «=» – проверка двух верхних операндов стека на равенство (<a, b> - <1>), если операнды равны и (<a, b> - <0>) – иначе;

◆ «>» – проверка двух верхних операндов стека на неравенство (<a, b> - <1>), если b > a и (<a, b> - <0>) – иначе.

Также определены логические операции над операндами стека.

◆ «&» – побитовое «и», примененное к двум верхним операндам стека (<a, b> - <a & b>);

◆ «|» – побитовое «или», примененное к двум верхним операндам стека (<a, b> - <a | b>);

◆ «~» – логическое «не», примененное к верхнему операнду стека (<a> - <~a>).

Язык FALSE также, очевидно, предоставляет возможности добавления операндов на стек непосредственно по значению.

◆ «integer» – добавляет целое беззнаковое число integer на стек (< - > - <integer>)

◆ « ’ symbol » – добавляет на стек ascii-код, соответствующий symbol (< - > - <integer>)

Помимо этого, язык предоставляет поддержку глобальных переменных. Хотя, в силу особенностей языка, они не особо нужны для хранения числовых значений, но переменные активно используются для хранения функций. Имена переменных строго фиксированы – это только символы от “a” до “z” включительно. Для работы с переменными FALSE также предоставляет операции.

◆ «:» – функция присваивания, присваивает переменной значение или функцию. Например, “1a:” присвоит переменной “a” значение “1”.

◆ «;» – функция, обратная функции присваивания. Эта операция кладет значение переменной на стек. (< - > - <a>)

Естественно, язык FALSE обеспечивает возможность объявления функций.

◆ «[]» – оператор объявления функции. Все команды языка FALSE, находящиеся внутри таких скобок определяют функцию. Функции языка FALSE не могут принимать аргументы, но могут оперировать со стеком. Например, функция [1+] добавляет единицу к числу, находящемуся на вершине стека. Таким образом, необходимые модификации стека непосредственно до вызова функции позволяют обходиться без аргументов.

◆ «!» – оператор вызова функции. Этот оператор позволяет применить функцию, объявленную ранее с помощью оператора «[]» или функцию, присвоенную переменной. Например, функция инкремента может быть объявлена как:

```
[ 1+ ] i:
```

и вызвана как:

```
2 i; !
```

Также язык FALSE позволяет оперировать непосредственно со стеком программы.

◆ «$» – оператор, дублирующий верхний элемент стека (<a> - <a, a>);

◆ «%» – оператор, удаляющий верхний элемент стека (<a> - < - >);

◆ «\» – оператор, меняющий местами два верхних элемента стека (<a, b> - <b, a>);

◆ «@» – оператор, выполняющий циклическую перестановку трех верхних элементов стека (<a, b, c> - <b, c, a>);

◆ «Ø» – оператор, копирующий n-й элемент стека на вершину (<n> - <a_n>);

Очевидно, в языке должны быть структуры, позволяющие управлять выполнением кода. FALSE предоставляет нам такие операторы.

◆ «?» – оператор, аналогичный оператору «if» в C-подобных языках. (<bool, fun> - < - >). Рассмотрим этот оператор подробнее.

```
1 2 = [ a; 1 + ] ?
```

Оператор «?» берет со стека два значения. Второе значение интерпретируется как истинное или ложное. Если второе значение истинное, то первое – интерпретируется как адрес функции и затем исполняется. В примере выше, на стек сначала будут положены числа 1 и 2, потом будет выполнена операция сравнения, на стек будет положено число 0, затем оператор условного ветвления снимет со стека адрес функции и число 0, а после прекратит выполнение.

◆ «#» – оператор, аналогичный оператору “while” в С-подобных языках. Приведем пример.

```
[ 1 1 = ] [ a; 1 + ] #
```

Оператор «#» также снимает со стека два значения. Оба интерпретируются как адреса функций. Сначала исполняется функция, чей адрес лежал на стеке вторым. Если эта функция кладет на стек истинное значение, то исполняется функция, чей адрес был первым. Затем снова выполняется функция, чей адрес был вторым и т.д., до тех пор, пока функция-условие не положит на стек ложное значение. В примере выше представлена реализация вечного цикла на языке FALSE. Действительно, сначала выполнится функция, расположенная в коде раньше, положит на стек значение «1», что будет интерпретировано как «истина», затем выполнится код функции, объявленной позже, затем снова выполнится код первой функции и т. д.

Также язык обеспечивает возможности работы с потоками ввода/вывода.

◆ «“string”» – оператор, печатающий строку в стандартный поток вывода. Не поддерживает escape-последовательности.

◆ «.» – оператор, печатающий в стандартный поток вывода верхний элемент стека.

◆ «,» – оператор, интерпретирующий верхний элемент стека как ascii-символ и выводящий его в стандартный поток вывода.

◆ «^» – оператор, считывающий со стандартного поток ввода символ и помещающий его на верхушку стека.

◆ «ß» – оператор, выводящий все содержимое буфера в стандартный поток вывода. Аналогичен функции flush() в языке C.

Также реализация компилятора на языке ассемблера для архитектуры 68000 позволяла расширять возможности FALSE вставками машинного кода.

◆ «<integer>`» – оператор, позволяющий записать команду процессору 68000 непосредственно в генерируемый файл.

1.2 Компилятор языка FALSE для архитектуры 68000

Рассмотрим реализацию компилятора языка FALSE для архитектуры 68000, представленную Вутером ван Оортмерсеном в 1993 году. Исходный код компилятора размещен на официальном сайте языка FALSE [1]. Остановимся на основных подходах, примененных при разработке компилятора.

Рассмотрим подробнее алгоритм, предложенный автором. Блок-схему данного алгоритма можно увидеть на рисунке ниже.

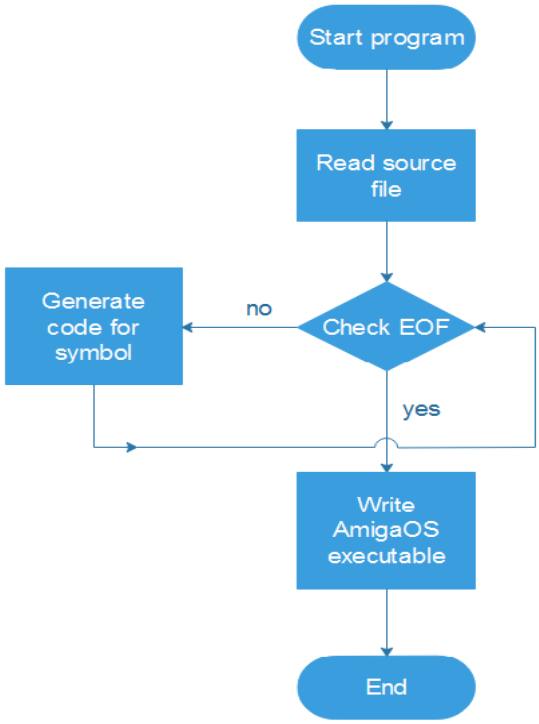

Рисунок №1. Блок-схема алгоритма, предложенного Вутером ван Оортмерсеном

Алгоритм довольно простой – изначально весь файл считывается в память устройство, затем за один проход по считанному файлу генерируется исполняемый код для каждого символа. Если встречается символ конца текста, то программа завершает работу, предварительно записав на диск сгенерированный код.

Вутер ван Оортмерсен пользуется здесь таким свойством языка FALSE как конкатенативность – конкатенация любых двух фрагментов кода порождает их композицию. [3] Рассмотрим понятие конкатенативности подробнее.

Язык программирования называется конкатенативным если он удовлетворяет следующим требованиям:

- Элементарное правильно построенное выражение языка – это унарная функция с одним аргументом и одним возвращаемым значением.
- Если X и Y – правильно построенные выражения языка, то их конкатенация тоже правильно построена.
- Если Z – конкатенация X и Y, то значение Z – это композиция значений функций X и Y.

Нетрудно убедиться в том, что любой стековый язык программирования является конкатенативным. Функции в таком случае рассматриваются как унарные, принимающие один аргумент – стек, и возвращающие его измененным.

Конкатенативные языки имеют множество достоинств, однако главное, с точки зрения разработчиков компиляторов, состоит в том, что любую программу можно рассматривать как набор несвязных между собой символов. Именно это позволяет реализовать отдельную генерацию кода для каждого символа в исходном файле.

Мы не будем рассматривать детали реализации алгоритма, описанного выше. Реализация Вутера ван Оортмерсена довольно активно использует библиотеку dos.library, разработанную для AmigaOS, включающую в себя

основные аспекты работы с динамической памятью, стандартными потоками ввода-вывода, файловой системой. Эта библиотека специфична для платформы AmigaOS, что затрудняет реализацию вышеприведенного алгоритма на других операционных системах.

2. РАЗРАБОТКА АРХИТЕКТУРЫ И ПОДХОДОВ К РЕАЛИЗАЦИИ КОМПИЛЯТОРА ДЛЯ ПЛАТФОРМЫ DOS 8086

2.1 Разработка архитектуры компилятора

Было решено воспользоваться алгоритмом, описанным в разделе 1.2, однако внести в него некоторые модификации, связанные со спецификой платформы DOS 8086. Блок-схему разработанного алгоритма можно увидеть на рисунке 2.

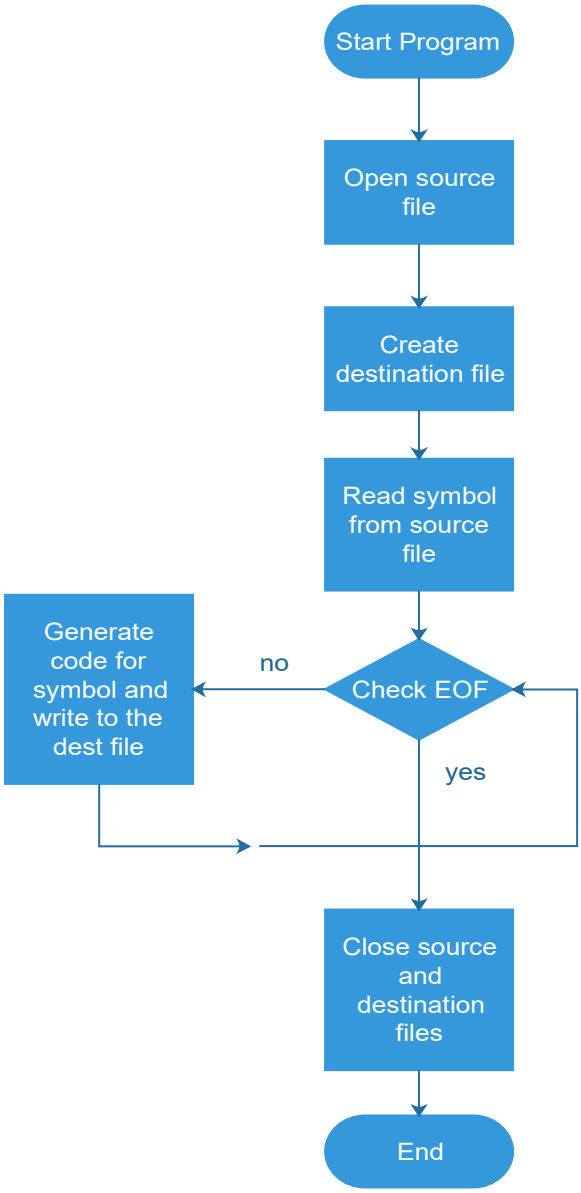

Рисунок №2. Блок-схема разработанного алгоритма

Можно заметить, что концепция алгоритма Вутера ван Оортмерсена осталась неизменной – мы все так же генерируем машинный код для процессора 8086 по отдельности для каждого символа. Однако реализация, представленная выше, не считывает файл в память, а оперирует с ним непосредственно.

Плюсы такого подхода очевидны. Во-первых, существенно упрощается реализация – нет нужды работать с динамической памятью, разнообразными режимами адресации и т. д. Во-вторых, учитывая особенности работы с платформой DOS 8086, загрузка файла в память обошлась бы «дорого» в плане размера исполняемого файла компилятора.

Минусы этого решения не столь значительны в данном контексте – скорость получения данных с диска будет существенно ниже, чем скорость получения их же из оперативной памяти.

По аналогичным соображениям, генерируемый код не хранится непосредственно в памяти, а сразу записывается в файл.

Рассмотрим теперь подробнее вопросы генерации кода для каждого символа.

Так как FALSE – стековый язык программирования, было решено представить всю программу в виде набора инструкций, изменяющих два стека – стек возвратов и стек данных, растущие навстречу друг другу. Упрощенное внутреннее представление программы изображено на рисунке 3.

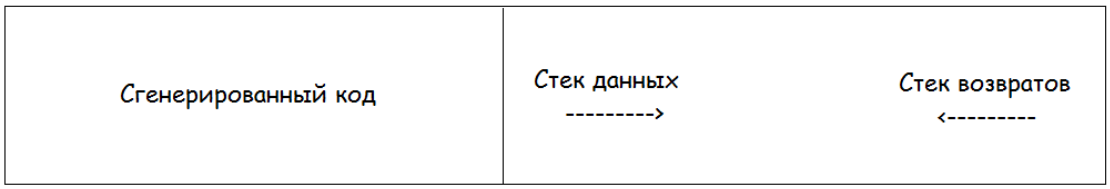

Рисунок 3. Упрощенное внутреннее представление программы на языке FALSE

Довольно просто в таком контексте обрабатываются числовые значения – необходимо просто сгенерировать код, добавляющий число на вершину стека данных.

Также несложно реализовать функции непосредственной работы со стеком, описанные в разделе 1.1. Каждая из операций должна соответствующим образом манипулировать со стеком данных.

Аналогично наивная реализация была предложена и для математических функций. Сперва необходимо снять соответствующее количество операндов со стека данных, затем произвести над ними определенную операцию и положить результат обратно на вершину стека данных.

Чуть сложнее устроена работа с переменными. Под переменные резервируется массив из 26 элементов, каждый из которых представляет собой одну переменную-букву от «a» до «z». Операции «:» и «;», описанные в разделе 1.1, обращаются по индексу к соответствующим элементам массива и оперируют с ячейками массива или стеком данных. Таким образом, измененная структура программы выглядит так, как показано на рисунке 4.

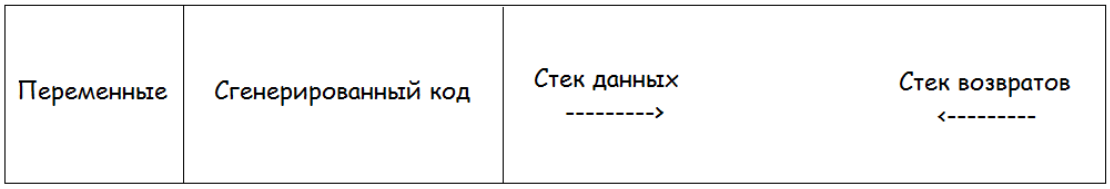

Рисунок 4. Внутренне представление программы с учетом переменных

Основные сложности состояли в работе с функциями. В итоге, было решено обрабатывать функции следующим образом.

Первым делом, при встрече оператора «[», обозначающего начало функции, в файл с генерируемым кодом записывается команда перехода на n байт вперед, где n – размер функции в байтах. Затем код генерируется обычным

образом, посимвольно, для каждого следующего элемента программы, до тех пор, пока не будет встречен оператор «]», сигнализирующий о конце функции. В этом случае, необходимо корректно обработать выход из функции, то есть, вернуть управление коду, который вызывал функцию. Для этого и необходим стек возвратов. При выходе из функции, необходимо снять значение со стека возвратов, интерпретировать его как адрес и передать управление коду, находящемуся по этому адресу. Однако, при такой реализации теряется адрес функции, то есть, функцию нельзя ни присвоить переменной, ни передать в качестве аргумента другой функции, ни вызвать сразу же после объявления. Впрочем, этот недочет легко исправить – нужно просто положить адрес начала функции на стек данных.

Очевидно, для корректный работы оператора «[]» необходимо корректно обрабатывать оператор вызова функции – «!». Код, сгенерированный для этого оператора, должен выполнять следующую последовательность действий. Сначала необходимо снять со стека данных адрес вызываемой функции, затем положить на вершину стека возвратов текущий адрес, а после передать управление по адресу, снятому со стека данных.

Стоит обратить внимание на существенную деталь, которая мешает такой «наивной» реализации, приведенной выше. Проблема состоит в том, что количество команд в функции заранее неизвестно, то есть, сгенерировать переход на n байт вперед невозможно, так как число n нам все еще неизвестно. Существует два основных подхода к решению данной задачи [4] – многопроходная компиляция и механизм обратных поправок.

Метод многопроходной компиляции требует неоднократного прохода по файлу, сохранения какой-либо вспомогательной информации (например, длины функций), а затем генерацию кода. Этот метод не подходит для нашей задачи в виду неоптимального потребления памяти.

Механизм обратных поправок заключается в том, что для каждой метки синтезируется список атрибутов, хранящих в себе адреса переходов. В

частности, при генерации перехода целевая метка остается временно неизвестной. Каждый такой переход помещается в список переходов, метки которых будут указаны после того, как эти метки смогут быть определены. Все переходы в одном списке имеют одну и ту же целевую метку.

Однако и этот подход видится для нашей задачи избыточным. Дело в том, что при генерации кода для функций, метка перехода в нашем случае всего одна для каждой функции. Значит, для корректной обработки меток, включая вложенные функции, достаточно всего лишь стека.

Опишем идею подробнее. При запуске программы инициализируется стек обратных поправок. При генерации кода для оператора «[» в стек обратных поправок кладется текущий адрес, после код генерируется посимвольно для каждого элемента программы, а по достижении оператора «]» из стека обратных поправок выбирается верхний элемент, вычисляется смещение и записывается адрес перехода.

Этот же метод можно применить и для генерации кода оператора «“”». Встретив «“», необходимо положить в стек текущий адрес, затем сгенерировать ascii-коды последующих символов, а встретив «”», снять со стека адрес поправки и вычислить смещение, после сгенерировать вызов прерывания для печати строки на экран, указав правильный адрес.

Довольно просто обрабатываются в этом случае и управляющие конструкции – «?» и «#». Достаточно только снять со стека адреса соответствующих функций и выполнить их, в зависимости от логических значений, помещенных ранее на стек.

2.2 Разработка архитектуры библиотеки времени исполнения

Также было принято решение вынести часть функционала языка FALSE в библиотеку времени исполнения.

Таким образом, несколько изменяется алгоритм работы компилятора (см. рисунок №5).

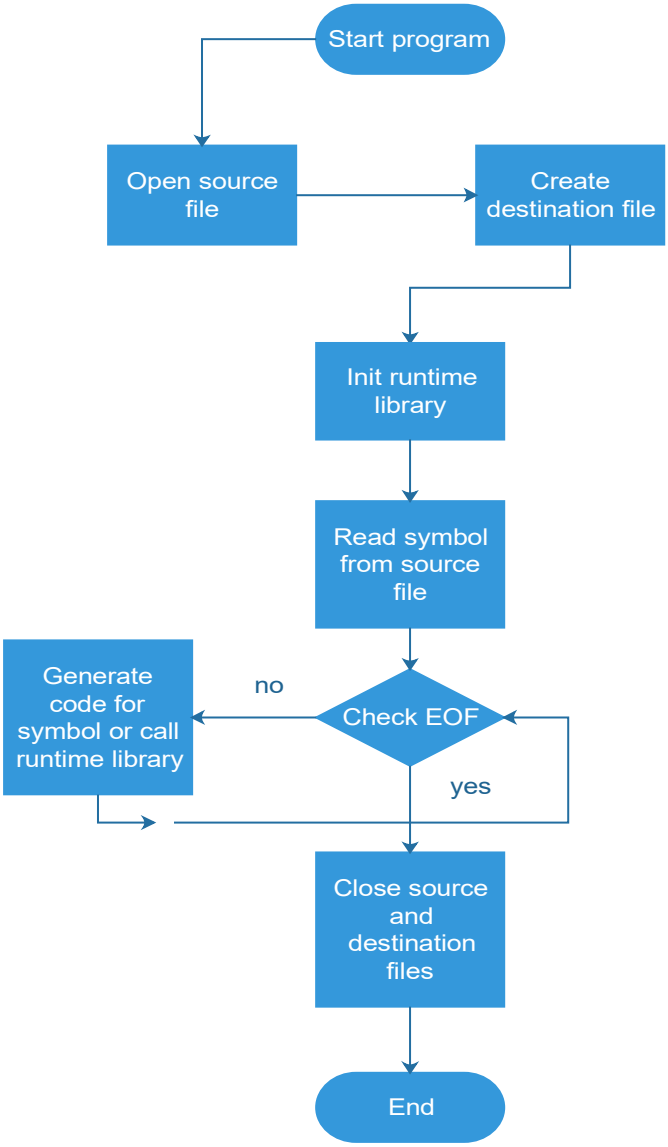

Рисунок №5. Блок-схема модифицированного алгоритма работы компилятора

Как можно заметить, алгоритм усложнился несущественно. Необходимо лишь корректно инициализировать библиотеку времени выполнения и на каждой итерации генерировать не только код для считанного символа, но и вызовы библиотеки времени выполнения.

Таким образом, несколько меняется и внутренняя структура программы на языке FALSE. (см. рисунок №6).

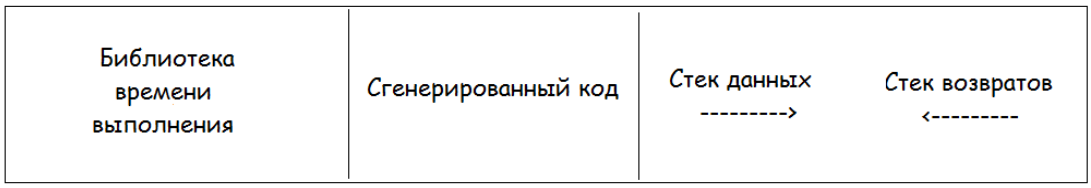

Рисунок №6. Внутренне представление программы на языке FALSE с учетом библиотеки времени выполнения

Мы видим, что вопросы инициализации переменных теперь являются сферой ответственности библиотеки времени выполнения. Также было решено вынести в библиотеку времени выполнения большую часть функций и операторов языка FALSE: математические функции, функции работы со стеком, управляющие конструкции, пользовательский ввод-вывод.

Для начальной инициализации библиотеки времени выполнения, её код должен будет располагаться или в начале целевого файла, или где-то в другом месте, но тогда первой инструкцией должен быть вызов процедуры инициализации.

Был выбран первый вариант – в начало целевого файла класть кусок кода как стандартный заголовок, передавая управление на инструкцию, следующую за его последним байтом. После чего генерировать целевой код, просто дописывая в конец. Этот подход с успехом применяется в BeRo Tiny Pascal. [5]

Также стоит обратить внимание на вопрос генерации кода библиотеки. К примеру, в компиляторе BeRo Tiny Pascal применен такой подход – сначала создается отдельный файл с ассемблерным кодом, компилируется, затем вставляется в компилятор просто как последовательность байт. Можно также, например, обойтись одним файлом с исходным кодом — начать писать в каком-то месте, предварив код директивой .org 100h.

Однако был выбран вариант проще — положить библиотеку времени выполнения в начало самого компилятора. В этом случае библиотека делает бесполезную инициализацию и передаёт управление на инструкцию за своим концом — там уже начинается код компилятора. При формировании целевого файла просто берется начало кода компилятора и пишем его в файл. Смещения всех процедур библиотеки одни и те же, поэтому никаких дополнительных вычислений делать не надо, что тоже упрощает задачу.

Такой подход оправдан тем, что существенно упрощает реализацию компилятора и оптимизирует потребление памяти за счет устранения дублирования кода.

3. РЕАЛИЗАЦИЯ КОМПИЛЯТОРА ЯЗЫКА FALSE ДЛЯ ПЛАТФОРМЫ DOS 8086

3.1 Реализация компилятора языка FALSE

Для реализации компилятора языка FALSE был выбран язык ассемблера для процессора 8086, в качестве компилятора языка ассемблера использовался TASM.

Рассмотрим реализацию алгоритма, описанного в разделе 2.2. Опустим довольно тривиальные процедуры работы с файлами, перейдем непосредственно к генерации библиотеки времени выполнения. Код процедуры можно увидеть в листинге №1.

```
init_runtime proc near
    mov cx, end_runtime - begin_runtime
    lea dx, begin_runtime
    call write

    mov cx, 3
    lea dx, OPCODE_INIT_SE
    call write

    mov dx, address_pointer
    sub dx, 100h
    call set_fp

    ret

init_runtime endp
```

Листинг №1. Процедура генерации библиотеки времени выполнения

Рассмотрим подробнее, что происходит в этом листинге.

Первые три строки осуществляют запись кода библиотеки времени выполнения в файл. Так как библиотека времени выполнения размещается в

начале исполняемого файла, что в коде компилятора, что в генерируемом коде, то смещения, используемые для вызова процедур будут определены корректно компилятором TASM.

Вторые три строки инициализируют стек данных. Константная строка «OPCODE_INIT_SE» определена ранее в секции данных кода компилятора и включает в себя инициализацию регистра si, который будет служить указателем на вершину стека данных.

Остановимся подробнее на последних строках функции инициализации библиотеки времени выполнения. Переменная address_pointer должна содержать адрес последней команды, записанной в генерируемой файл. Эта переменная нужна для корректной обработки обратных поправок. Изначально переменная address_pointer содержит значение 100h. Такое значение было определено на основании специфики работы платформы MS-DOS. Система, по умолчанию, загружает программу пользователя по адресу 100h и начинает выполнение с этого адреса. Таким образом, чтобы установить указатель записи файла для дальнейшей корректной обработки исходного кода на языке FALSE, необходимо вычесть этот адрес из address_pointer.

Далее рассмотрим процедуру, осуществляющую чтение файла с исходным кодом на языке FALSE. (см. листинг №2).

```
readfile proc near
         mov  ah,3FH
         mov  bx,handle
         lea  dx,fbuff
         mov  cx,1
         int  21H
         cmp  ax,0
         jz   eoff
         mov  dl,fbuff
         cmp  dl,1ah
         jz   eoff
         call proc_symbol
         jmp  readfile
eoff:
```

```
ret
readfile endp
```

Листинг №2. Процедура посимвольного чтения файла

Процедура посимвольного чтения файла тоже довольно простая. После установки всех необходимых параметров, вызывается прерывание DOS, которое считывает один символ из файла. После этого считанный символ сохраняется в переменной fbuff, которая далее используется для его дальнейшей обработки. Если был считан символ конца файла, то функция завершает свою работу и передает управление далее, иначе вызывается функция-обработчик считанного символа.

В следующих листингах рассмотрим подробнее код функции-обработчика считанного символа. (см. листинги №3 - №8)

```
proc_symbol proc near
        cmp fbuff, 7Bh
        jnz _end_comm
        mov flags, 02h
        ret

        _end_comm:
        cmp fbuff, 7Dh
        jnz _comm
        mov flags, 00h
        ret

        _comm:
        cmp flags, 02h
        jnz _0
        ret

        _0:
...
```

Листинг №3. Функция-обработчик считанного символа

Данный отрывок кода обрабатывает комментарии. Хочется отметить типичный для этой функции-обработчика прием. Сначала символ в переменной fbuff сравнивается с ascii-кодом символа, соответствующим началу комментария, затем, если сравнение не устанавливает zero flag, то совершается переход на метку следующую за кодом-обработчиком символа. В противном случае вызывается обработчик считанного символа и функция-обработчик завершает свою работу после выполнения этого кода.

Обратим также внимание на одну деталь, характерную не только для обработки комментариев – переменную flags. Эта переменная отвечает за текущее состояние программы-компилятора и меняет свое значение в зависимости от обработки комментария, строки или числа.

```
...
_0:
     cmp fbuff, 22h ;22H -- "
     jnz _1
     jmp call_proc_quotation

 _1:
     cmp flags, 01h;
     jnz _cmp_num
     mov cx, 1
     lea dx, fbuff
     call write
     ret

...
```

Листинг №4. Генерация кода для строки

Остановимся чуть подробнее на генерации кода для строк. Основные подходы к реализации уже были описаны выше, обратим внимание только на вызов функции-обработчика кавычек, которая сама занимается установкой

флагов. Затем проверяется значение флага и при соответствующем значении генерируется код для символов строки.

Рассмотрим подробнее обработчик чисел.

```
_cmp_num:
    cmp fbuff, 2Fh; 30 -- 0
    jna _not_num
    jmp check_is_num
_num:
    mov flags, 03H
    mov al, fbuff
    sub al, 30H
    mov bytes[di], al
    inc di
    ret
_not_num:
    cmp flags, 03H
    jnz _2
    mov flags, 00H ; clear flags
    mov cx, 1
    lea dx, OPCODE_MOV_AX
    call write
    xor ax, ax
    mov dx, 10
    mov cx, di
    xor di, di
proc_byte:
    cmp di, cx
    jz write_num
    mov bl, bytes[di]
    mul dx
    mov dx, 10
    add ax, bx
    inc di
    jmp proc_byte
write_num:
    mov number, ax
    mov cx, 2
    lea dx, number
    call write

    call _push
```

```
    xor di, di

...

check_is_num:

    cmp fbuff, 3Ah
    jnb _fail
    jmp _num
    _fail:
    jmp _not_num

...
```

Листинг №5. Код обработчика чисел

Рассмотрим код обработчика чисел подробнее. Здесь мы можем наблюдать тот же подход к определению лексемы, что и ранее.

В том случае, если сравнение успешно, считанная из файла цифра записывается в массив bytes, инкрементируется счетчик массива,процедура-обработчик завершает свою работу, также устанавливается соответствующим образом переменная flags.

В противном случае, проверяется значение переменной flags. Это позволяет запустить парсер числа после того, как число было считано из файла целиком. Парсер числа довольно прост, реализует примитивную версию функции atoi() языка C. После завершения работы парсера, обработанное число записывается в генерируемый файл. Стоит обратить внимание на то, что процедура-обработчик не завершает свою работу после записи. Это необходимо для того, чтобы корректно обработать следующий за последней цифрой символ. После этого обработанное число кладется на вершину стека.

Рассмотрим теперь подход, использованный для вызова функций библиотеки времени выполнения, представленный в листинге №6.

```
...
cmp fbuff, 3Ah
```

```
jnz _3
lea dx, runtime_assign
mov proc_addr, dx
call write_exec
ret
...
write_exec proc near
mov cx, 1
lea dx, OPCODE_MOV_DX
call write

mov cx, 2
lea dx, proc_addr
call write

mov cx, 4
lea dx, OPCODE_RUNTIME_CALL
call write

mov cx, 2
lea dx, OPCODE_CALL_ABS
call write

ret

write_exec endp
```

Листинг №6. Вызов функций библиотеки времени выполнения

Подробнее опишем код, представленный в листинге № 6. Такой подход применяется для генерации кода вызова большинства функций библиотеки времени выполнения. После описанной уже процедуры сравнения, устанавливаются необходимые значения регистров и вызывается процедура write_exec, чей код также представлен в листинге.

Рассмотрим код этой процедуры. Сперва в генерируемый файл записывается код, который кладет в регистр dx смещение, по которому находится код функции библиотеки времени выполнения. Затем в файл записываются константы, представляющие машинные коды, которые позволяют

процессору вызывать функцию, находящуюся по адресу, который содержится в регистре dx.

Также следует обратить внимание на код, отвечающий за корректную обработку функций языка FALSE, представленный в листингах №7 и №8.

```
_5:
    cmp fbuff, 5Bh
    jnz _6

    mov cx, 2
    lea dx, OPCODE_JMP
    call write

    mov si, jmp_labels_pointer
    mov bx, address_pointer
    dec bx
    mov jmp_labels[si], bx
    add jmp_labels_pointer, 2
    ret

_6:
    cmp fbuff, 5Dh
    jnz _7
    jmp call_proc_end
```

Листинг №7. Вызов процедуры-обработчика функций языка FALSE

```
proc_end proc near

    mov cx, 1
    lea dx, OPCODE_RET
    call write

    sub jmp_labels_pointer, 2
    mov si, jmp_labels_pointer
    lea bx, jmp_labels[si]
    mov dx, [bx]
    sub dx, 100h

    call set_fp
```

```
    mov si, jmp_labels_pointer;
    lea bx, jmp_labels[si]
    mov ax, address_pointer
    sub ax, [bx];
    dec ax
    mov jmp_offset, ax;
    lea dx, jmp_offset
    mov bx, exechandle
    mov ax, 4000h
    mov cx, 1
    int 21h

    call ret_fp

    mov ax, jmp_labels[si]
    inc ax
    mov number, ax

    mov cx, 1
    lea dx, OPCODE_MOV_AX
    call write

    mov cx, 2
    lea dx, number
    call write

    call _push
    ret

proc_end endp
```

Листинг №8. Код процедуры-обработчика функций языка FALSE

Поясним код, приведенный в листингах выше. Код, вызывающий обработчики операторов «[» и «]» довольно стандартный. Сами же обработчики реализуют алгоритм обратных поправок, представленный в разделе 2.2.

3.2 Реализация библиотеки времени выполнения

Рассмотрим реализацию некоторых функций библиотеки времени выполнения.

Весь код библиотеки времени выполнения расположен в самом начале файла что в коде компилятора, что в сгенерированном машинном коде для программы на языке FALSE. Это необходимо для корректной обработки смещений при вызовах процедур. Общая структура кода библиотеки времени выполнения приведена в листинге №9.

```
begin_runtime:
        jmp end_runtime
        VARS: dw 26 dup(0)

runtime_if proc near
    ...
runtime_if endp

runtime_while proc near
    ...
runtime_while endp

...

end_runtime:

;здесь начинается код компилятора
```

Листинг №9. Общая структура кода библиотеки времени выполнения

Как можно заметить, весь код библиотеки времени выполнения обрамлен метками «begin_runtime» и «end_runtime». Эти метки используются для корректной записи кода библиотеки в генерируемый файл. (см. листинг №1). В самом начале кода библиотеки времени выполнения расположена команда «jmp end_runtime», которая пропускает весь код библиотеки, позволяя компилятору и программам не выполнять ошибочные действия.

Рассмотрим некоторые детали реализации функций библиотеки времени выполнения. В листинге №10 обратим внимание на работу со стеком.

```
runtime_push proc near
        add si, 2;
        mov bx, ax
        mov [si], ax;
        ret
runtime_push endp

runtime_pop proc near
        sub si, 2
        mov bx, [si]
        ret
runtime_pop endp
```

Листинг №10. Работа со стеком

Было принято решение хранить вершину стека по адресу, расположенному в регистре si, для удобства некоторых операций вершина стека дублировалась в регистр bx. Каждый элемент стека занимает одно машинное слово. Процедуры, приведенные в листинге №10 обеспечивают добавление и удаление элементов со стека.

Не будем подробно останавливаться на коде всех процедур библиотеки времени выполнения, их общий принцип не отличается от работы функций, представленных выше, в листинге №10.

4. ТЕСТИРОВАНИЕ И ОПТИМИЗАЦИЯ РАЗМЕРА ИСПОЛНЯЕМОГО ФАЙЛА

4.1 Оптимизация размера исполняемого файла

Вутер ван Оортмерсен разрабатывал язык FALSE с намерением написать для него компилятор размером менее, чем 1кб. К сожалению, из-за разных особенностей платформ MS-DOS и AmigaOS повторить этот результат не удалось. Размер исполняемого файла компилятора был равен 1792 байтам. В целях улучшения результата были проведены некоторые оптимизации в этом направлении были.

В частности, была несколько изменена работа библиотеки времени выполнения. Дело в том, что зачастую «ненужный» код использовался для сохранения одинаковых значений в [si] и bx. Это разительно изменило объем некоторых функций библиотеки времени исполнения. Сравнение процедур «swap» приведено в листинге №11.

```
;До оптимизации
runtime_swap proc near
        mov ax, bx
        call runtime_pop
        mov dx, bx
        call runtime_pop
        call runtime_push
        mov ax, dx
        call runtime_push
        ret
runtime_swap endp

;После оптимизации
runtime_swap proc near

        xchg bx, [si]
        ret
```

```
runtime_swap endp
```

Листинг №11. Сравнение процедур «swap» до оптимизации и после

Как можно увидеть, до оптимизации процедура «swap» вынуждена была использовать дополнительный регистр и вызывать процедуры, оперирующие со стеком, после оптимизации все это сократилось до одной команды xchg.

Также оптимизации подвергся цикл обработки считанных символов. Проблема состояла в том, что код вызова функций библиотеки времени исполнения отображал одну и ту же логику, но при этом не мог быть заменен одной функцией вследствие активного использования констант. Оптимизация этого момента представлена в листинге №12.

```
simple_commands_table DW (?)
dw 3A00h, offset runtime_assign
dw 3B00h, offset runtime_get_value ...
end_commands_table DW (?)

_20:
        mov si, offset simple_commands_table
        mov bx, offset end_commands_table
        sub bx, si
        mov len, bx
        _iter_table:
        xor bx, bx
        mov bh, fbuff
        cmp [si], bx
        jz _print_function
        add si, 2
        mov cx, si
        sub cx, offset simple_commands_table
        cmp cx, len
        ja _21
        jmp _iter_table
        _print_function:
        add si, 2
        mov dx, [si]
        mov proc_addr, dx
        call write_exec
```

```
        ret

...
```

Листинг №12. Оптимизация вызова функций времени исполнения

Как мы видим, была составлена таблица, содержащая ascii-коды символов и соответствующие им смещения функций библиотеки времени исполнения. Затем в цикле обработки считанных символов запускался еще один цикл, отвечающий за итерацию по символам таблицы. После этого смещение соответственно выбранной функции помещается в регистр dx, после чего вызывается процедура write_exec, подробно описанная в разделе 2.1.

После всех оптимизаций размер исполняемого файла компилятора получился 1426 байт.

4.2 Тестирование

Для первичного тестирования был составлена простая программа, написанная на языке FALSE, и отображающая его основные функции. Код программы приведен в листинге №13.

```
1 2 + . "
should be 3
"
12341 123 + . "
should be 12464
"
3 4 * . "
should be 12
"
5 5 * 5 * . "
should be 125
"
7 10 - . "
should be 3
"
14 1233 - ."
```

```
should be 1219
"
3 39 / . "
should be 13
"
111 777 / . "
should be 7
"
"if test
"
1 2 = ["hello
"]? "
should print nothing
"
1 1 = ["hello"]? "
should print hello
"
10a: [2 a; >][1 a; - a: "hello"]#
```

Листинг №13. Код тестовой программы на языке FALSE

Также использовался дистрибутив языка FALSE, содержащий интерпретатор, написанный на языке C и некоторые примеры программ [6]. Например, разнообразные утилиты для обработки текстовых файлов, приведенные в листингах №14 и №15.

```
[^$$1_=~][10=[13,]?,]#
```

Листинг № 14. Пример программы на языке FALSE

Эта программа считывает ввод пользователя и заменяет символы перевода строки на возврат каретки.

```
[^$$1_=~][13=~[$,]?%]#
```

Листинг № 15. Пример программы на языке FALSE

Этот скрипт позволяет удалить все возвраты каретки из стандартного потока ввода, оставив все переводы строки.

ЗАКЛЮЧЕНИЕ

При выполнении данного курсового проекта были изучены различные архитектуры процессоров, языки ассемблера, подходы к реализации компиляторов стековых языков программирования. Также был разработан компилятор стекового языка программирования FALSE.

Однако разработанный компонент, к сожалению, далек от идеала. Например, пока что отсутствует проверка ошибок в работе компилятора. Также недостает стадии семантического анализа, которая позволила бы проверять многие действия пользователя на этапе компиляции. Впрочем, все эти детали не вписываются в рамки курсового проекта и не отражают поставленную задачу – написание компилятора языка FALSE как можно меньшего размера. В этом плане также существует место для оптимизаций, тщательной переработкой кода можно добиться сокращения размера исполняемого файла компилятора еще на несколько десятков байт.

Несмотря на указанные недостатки, разработанный компилятор может успешно использоваться для работы с программами на языке FALSE.

СПИСОК ЛИТЕРАТУРЫ

[1] Официальный сайт языка программирования FALSE.
URL: http://strlen.com/false-language/ (дата последнего обращения 08.03.2018).

[2] Документация языка FALSE. URL: http://strlen.com/files/lang/false/false.txt (дата последнего обращения 09.03.2018).

[3] J. Purdy «Why Concatenative Programming Matters». URL: http://evincarofautumn.blogspot.ru/2012/02/why-concatenative-programming-matters.html (дата последнего обращения 09.03.2018).

[4] Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. — 2 изд. — М.: Вильямс, 2008.

[5] Исходный код и документация компилятора Bero Tiny Pascal
URL: https://github.com/BeRo1985/berotinypascal (дата последнего обращения: 22.03.2018)

[6] Дистрибутив языка FALSE, содержащий материалы для тестирования.
URL: http://strlen.com/files/lang/false/False12b.zip (дата последнего обращения 09.03.2018).