|  |  |
| --- | --- |
| *voenmeh* | МИНОБРНАУКИ РОССИИ  федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-01 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Факультет |  | И |  | Информационные и управляющие системы |
|  |  | шифр |  | наименование |
| Кафедра |  | И9 |  | Систем управления и компьютерных технологий |
|  |  | шифр |  | наименование |
| Дисциплина |  | Программирование на языке высокого уровня | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| "Создание транслятора» |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Выполнила студентка группы | | | |  | И944 |
| Юринова О.М. | | | | | |
| Фамилия И.О. | | | | | |
| **РУКОВОДИТЕЛЬ** | | | | | |
| Афанасьев М.П. | |  |  | | |
| Фамилия И.О. Подпись | | | | | |
| Оценка |  | | | |  |
| «\_\_\_\_\_» |  | | | | 2016 г. |

САНКТ-ПЕТЕРБУРГ

2016г.

**Содержание**

[Определения, обозначения и сокращения 2](#_Toc453187566)

[Введение 4](#_Toc453187567)

[Постановка задачи 5](#_Toc453187568)

[Описание языка 6](#_Toc453187569)

[Терминалы грамматики модифицированного языка Pascal 6](#_Toc453187570)

[Грамматика входного языка в форме Бэкуса-Наура: 7](#_Toc453187571)

[Тип грамматики 9](#_Toc453187572)

[Описание транслятора 10](#_Toc453187573)

[Лексический анализатор 11](#_Toc453187574)

[Синтаксический анализатор 15](#_Toc453187575)

[Генератор кода 17](#_Toc453187576)

[Тестирование 20](#_Toc453187577)

[Пример 1 20](#_Toc453187578)

[Пример 2. «Программа с ошибкой» 22](#_Toc453187579)

[Пример 3. 22](#_Toc453187580)

[Пример 4 24](#_Toc453187581)

[Заключение 28](#_Toc453187582)

[Список использованных источников 29](#_Toc453187583)

Определения, обозначения и сокращения

**Транслятор** **—** это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на результирующем (выходном) языке.

**Ассемблер** (от англ. *assembler* — сборщик) — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

**Компиляция —**трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера)

**Форма Бэкуса — Наура**(БНФ) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик.

**Формальной грамматикой** называется четверка вида:

,

где ***VN***- конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы);

***VT*** - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.), *VT* ∩*VN =∅*;

***Р*** – множество правил вывода грамматики, являющееся конечным подмножеством множества (*VT∪ VN*)*+ ×* (*VT∪ VN*)*\**; элемент (*α, β*) множества *Р* называется правилом вывода и записывается в виде *α*→*β* (читается: «из цепочки *α* выводится цепочка *β*»);

***S*** – начальный символ грамматики, *S* ∈*VN*.

Грамматика G = (N, T, S, P) называется **контекстно-свободной** (или КС-грамматикой), если любая ее продукция имеет вид А → α, где А ∈ N, α ∈ (N ∪ T)\*.

**Терминал** (терминальный символ) - объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»).

**Нетерминал** (нетерминальный символ) - объект, обозначающий какую-либо сущность языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

**ПО** – программное обеспечение.

**ОС** – операционная система.

**ЛА** – лексический анализатор.

**СА –** синтаксический анализатор.

**ДКА-** детерминированный конечный автомат.

Введение

Основы классической теории компиляторов – одной из важнейших составных частей системного программного обеспечения.

Несмотря на более чем полувековую историю вычислительной техники, формально годом рождения теории компиляторов можно считать 1957 год, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом. Лишь появление теории формальных языков и строгих математических моделей позволило перейти от «творчества» к «науке». Именно благодаря этому, стало возможным появление сотен новых языков программирования.

Распространенные в наше время языки высокого уровня, к которым относится большинство языков программирования, представляют собой некоторое промежуточное звено между чисто формальными языками и языками естественного общения людей. От первых им досталась строгая формализация синтаксических структур предложений языка, от вторых – значительная часть словарного запаса, семантика основных конструкций и выражений (с элементами математических операций, пришедшими из алгебры).

Появление языков высокого уровня существенно упростило процесс программирования. Компиляторы создавались и продолжают создаваться не только для новых, но и для давно известных языков. Ныне они являются неотъемлемой частью любой вычислительной системы, ведь без их существования программирование любой прикладной задачи было бы затруднено, а то и просто невозможно. Да и программирование специализированных системных задач, как правило, ведется если не на языке высокого уровня, то на языке ассемблера, следовательно, применяется соответствующий компилятор. Программирование непосредственно на языках машинных кодов происходит исключительно редко и только для решения очень узких вопросов.

Постановка задачи

Задача курсовой работы - написание транслятора для модифицированного языка программирования Pascal на язык ассемблера.

Основные требования программы:

* использование С в качестве языка программирования;
* реализация простых выражений, процедур и циклов как пример работы транслятора.

Процесс написания транслятора состоит из следующих этапов:

1. Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.
2. Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.
3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
4. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.
5. Генерация кода. Из промежуточного представления порождается код на целевом языке.

В конкретных реализациях компиляторов эти этапы могут быть разделены или, наоборот, совмещены в том или ином виде.

Описание языка

Pascal (назван в честь Блеза Паскаля) — это чисто процедурный язык программирования, часто использующийся для обучения структурному программированию. Был разработан Никлаусом Виртом в 1970; вопреки расхожему мнению, он не был исключительно учебным языком, а предназначался для практического применения.

Особенностями языка являются строгая типизация и наличие средств структурного (процедурного) программирования. Pascal был одним из первых таких языков. По мнению Н. Вирта, язык должен был способствовать дисциплинированию программирования, поэтому, наряду со строгой типизацией, в Pascal сведены к минимуму возможные синтаксические неоднозначности, а сам синтаксис интуитивно понятен даже при первом знакомстве с языком. Это упрощает написание компиляторов язык

## Терминалы грамматики модифицированного языка Pascal

|  |  |
| --- | --- |
| **Терминалы** | **Обозначения** |
| Ключевые слова | BEGIN, END, IF, THEN, ELSE, WHILE, DO, WRITE, WRITELN, READ, CALL, NIL, PROCEDURE, ODD, CONST, VAR |
| Знаки операций и отношений | +, -, \*, /, =, >, <, >=, <=, <> |
| Латинские буквы | A, B, …,Z  a, b, ..., z |
| Цифры | 0, 1, ..., 9 |
| Знак присваивания | := |
| Разделители | ‘;’, ‘,’, ‘.’,‘(’,’)’ |
| Комментарий | (\* … \*) |

## Грамматика входного языка в форме Бэкуса-Наура:

*Замечание.* Следующие символы представляют собой метасимволы, присущие формализму БНФ, а не языку Pascal:

::= |{ }

Фигурные скобки обозначают возможное повторение заключенных в них объектов (конструкций или символов) нуль или более раз.

Формула вида:

А {В}

есть просто сокращенная запись рекурсивного, по существу, правила:

А :: = <пусто> | АВ

*<программа> :: = <блок>.*

*<блок> :: = <раздел констант> < раздел переменных > <раздел процедур> <раздел операторов>*

*<раздел констант> ::= <пусто> | const< определение константы>{; <определение константы>};*

*<определение константы> :: = <имя> = <константа>*

*<имя>:: = <буква>{<буква > | < цифра >}*

*<цифра> :: = 0|1|2|3|4|5|6|7|8|9*

*<буква> :: = a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K*

*|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z*

*<константа> ::= <число без знака> | <знак> <число без знака> | <имя константы> | <знак> <имя константы>*

*<число без знака> ::= < цифра >{< цифра >}*

*<знак> :: = + | -*

*<имя константы> :: = <имя> | nil*

*<раздел переменных> :: = <пусто>| var <описание переменной >;*

*<описание переменной> :: = <имя>{, <имя>}*

*<раздел процедур> :: = {<описание процедуры> ;}*

*<описание процедуры > ::= procedure <имя> (<группа параметров>)*

*<группа параметров> :: = <имя> {, <имя>}*

*<раздел операторов> :: = <составной оператор >*

*<оператор> ::= <пустой оператор> | <оператор присваивания>| <составной опратор>*

*<пустой оператор>:: =<пусто>;*

*<пусто> ::=*

*<оператор присваивания > :: = <переменная> := <выражение>*

*<составной оператор> ::= begin <оператор> {; <оператор>} end*

*<выражение> ::= <простое выражение> | <простое выражение> <операция отношения> <простое выражение>*

*<операция отношения > :: = = | < > | < | < = | > = | >*

*<простое выражение> :: = <слагаемое> | <знак> <слагаемое> | <простое выражение> <аддитивная операция> <слагаемое>*

*<аддитивная операция> :: = + | -*

*<слагаемое> :: = <множитель> | <слагаемое> <мультипликативная операция> <множитель>*

*<мультипликативная операция > :: = \* | /*

*<множитель> ::= <переменная> | <константа без знака> | (<выражение> )*

*<условный оператор> :: = <выражение> then <оператор> | if <выражение> then <оператор> else <оператор>*

*<оператор цикла> ::= while <выражение> do <оператор>*

Нетерминалы в уточненной грамматике обозначают следующие конструкции модифицированного языка Pascal: *<программа>; <блок>; <раздел констант>; <определение константы>; <имя>; <цифра>; <буква>; <константа>; <число без знака>; <знак> ; <имя константы>; <раздел переменных>; <описание переменной>; <раздел процедур>; <описание процедуры >; <группа параметров>; <раздел операторов>; <оператор>; <пустой оператор>; <пусто>; <оператор присваивания >; <составной оператор>; <выражение>; <операция отношения >; <простое выражение>; <аддитивная операция>; <слагаемое>;*

*<мультипликативная операция >; <множитель>; <условный оператор>; <оператор цикла>.*

## Тип грамматики

Модифицированная версия языка Pascal относится к языкам с контекстно-свободной грамматикой. Так как каждое правило грамматики G = (N, T, P, S) удовлетворяет A http://citforum.ru/programming/theory/serebryakov/cmsy10-21.gif , где N – алфавит терминальных символов T – алфавит нетерминальных символов, P – конечное множество правил, S – начальный символ (начальное состояние), A – терминальный символ, - цепочка сиволов

Описание транслятора

Транслятор состоит из трех составляющих:

1. Лексического анализатора;
2. Синтаксического анализатора;
3. Генератора кода.

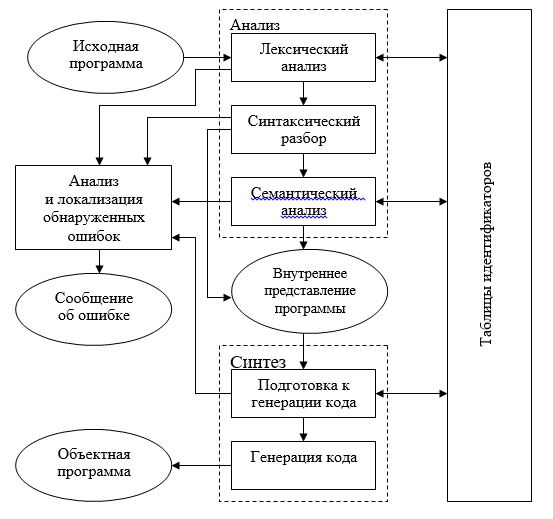


Рисунок 1. Фазы компиляции.

Транслятор переводит текст исходной программы в текст программы на языке Assembler, используя синтаксически управляемую трансляцию. После этого, текст программы на языке Assembler компилируется при помощи компилятора ТASM.

## Лексический анализатор

Лексический анализатор (ЛА) – считывает и обрабатывает текст программы построчно и формирует из них лексемы, которые передаются синтаксическому анализатору. Лексема(token) – минимальная неделимая лексическая единица языка. Также лексический анализатор обрабатывает пробелы и удаляет комментарии.

В данном трансляторе используется ЛА на базе ДКА, с таблицей переходов. Ключевые (зарезервированные) слова занесены в специальную таблицу: ( “ключевое слово” | “тип токена” ). При считывании идентификаторы проверяются по этой таблице, и, если такое слово есть, токену присваивается определенное значение из таблицы.

Основные функции ЛА:

* initsym()

{

int i;

for (i=0; i<256; i++)

ssym[i] = 0;

ssym['+'] = plus;

ssym['-'] = minus;

ssym['\*'] = times;

ssym['/'] = slash;

ssym['('] = lparen;

ssym[')'] = rparen;

ssym['.'] = period;

ssym['='] = eql;

ssym[','] = comma;

ssym['<'] = lss;

ssym['>'] = gtr;

ssym[';'] = semicolon;

ssym[':'] = colon;

return;

}

Функция записывает в таблицу токенов номер, соответствующий ключевому слову, или значение, соответствующее идентефикатору.

* getsym();

{

int digit, endstring, i, j, k;

go:

if (eofflag) return 0;

/\* pass over leading blanks, tabs, etc. \*/

while (isspace(ch)) ch = readchar();

/\* next symbol is identifier or reserved word \*/

if (isalpha(ch) && islower(ch))

{

strcpy(id, "");

k = 0;

do {

if (k < cmax-1)

id[k++] = ch;

ch = readchar();

} while (isalpha(ch) || isdigit(ch));

id[k] = 0;

findresw();

}

/\* next symbol is a number \*/

else if (isdigit(ch))

{

num = 0;

sym = number;

do {

digit = ch - '0';

if ((num < imax/10) ||

((num == imax/10) && (digit <= imax % 10)))

num = num\*10 + digit;

else {

num = 0;

fprintf(stderr, "Wrong number in input file\n");

}

ch = readchar();

} while (isdigit(ch));

}

/\* next symbol is ':' or ':=' \*/

else if (ch == ':')

{

ch = readchar();

if (ch == '=') {

sym = becomes;

ch = readchar();

}

else

sym = colon;

}

/\* next symbol is '<' or '<=' \*/

else if (ch == '<') /\* next symbol is '<' or '<=' \*/

{

ch = readchar();

if (ch == '=') {

sym = leq;

ch = readchar();

}

else

sym = lss;

}

/\* next symbol is '>' or '>=' \*/

else if (ch == '>')

{

ch = readchar();

if (ch == '=') {

sym = geq;

ch = readchar();

}

else

sym = gtr;

}

/\* next symbol is string \*/

else if (ch == '\'')

{

num = 0;

ch = readchar();

sym = stringsym;

endstring = 0;

do {

if (ch == '\'') {

ch = readchar();

endstring = (ch != '\'');

}

if (!endstring) {

str[num++] = ch;

ch = readchar();

if (eofflag) error();

}

} while (!endstring);

if (!endstring) num = 0;

str[num++] = 0;

}

/\* next symbol is '(' or possibly comment \*/

else if (ch == '(')

{

ch = readchar();

if (ch != '\*')

sym = lparen;

else /\* comment \*/

{

ch = readchar();

do {

while (ch != '\*') {

if (eofflag) error();

ch = readchar();

}

ch = readchar();

} while (ch != ')');

ch = readchar();

if (eofflag) error();

goto go;

}

}

else if (ch == EOF)

return 0;

/\* special symbol \*/

else

{

sym = ssym[ch];

if (sym == 0) {

fprintf(stderr,

"Wrong character: '%c' found in source.\n", ch);

ch = readchar();

goto go;

}

else

ch = readchar();

}

return 1;

}

Функция getsym проверяет следующий символ из исходного кода программы. Результат выводится в следующих переменных:

    sym: содержит тип или внутреннее представление следующего символа;

    num: содержит значение числа;

 id: содержит строку символов, определяющий идентификатор или зарезервированное слово;

    str: содержит рассматриваемую строку символов из исходного кода;

## Синтаксический анализатор

Синтаксический анализатор (СА) получает на вход результат работы ЛА и разбирает его в соответствии с грамматикой языка. Основная задача синтаксического анализатора – определить принадлежит ли исходная цепочка лексем входному языку, т.е. является правильным «предложением» языка.

Правила для построения СА берутся из синтаксиса в формате БНФ.

Основные функции СА:

* Получение первого символа и начало рекурсивного спуска:

getsym()(текст программы представлен в описании ЛА.)

program()

{

block(0, 0);

if (sym != period) error(9);

}

void block(int tx, int lev)

{

int ppos; /\* procedure position in

symbol table \*/

int dindex; /\* data allocation index \*/

int txfirst; /\* index of first declaration

in block \*/

int oldcdi; /\* saves old code index \*/

int currcdi; /\* current code index \*/

txfirst = tx + 1;

ppos = tx;

dindex = 3;

oldcdi = cdindex();

if (lev == 0) emit(CAL, 0, 0);

if (sym == constsym)

constpart(&tx, &txfirst, &dindex);

if (sym == varsym)

varpart(&tx, &txfirst, &dindex, lev);

while (sym == procsym)

procpart(&tx, &txfirst, &dindex, lev);

currcdi = cdindex();

if (lev == 0) fixup(oldcdi, currcdi);

entersymt(ppos, currcdi, dindex);

emit(INT, 0, dindex-3);

txcurr = tx;

levcurr = lev;

statement();

emit(RET, 0, 0);

printsymt(stlist, ppos, tx);

}

* Запись кода в файл:

writecode()

{

int i;

for (i=0; i<codeindex; i++)

fprintf(intcode, "%03d %s %2d %5d\n", i,

codemnemo[code[i].o], code[i].l,

code[i].a);

}

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

## Генератор кода

Генерация объектного кода – это перевод компилятором внутреннего представления исходной программы в результирующую объектную программу на языке ассемблера или непосредственно на машинном языке (машинных кодах). Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное пространство под функции и переменные, проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т.д. В данной курсовой работе объектный код является кодом на языке Assembler.

Основные функции генератора кода:

* readcode()

Функция считывает код файла в массив кода;

* checklabels()

Функция проверяет, какие инструкции должны иметь метки;

* prolog1()-assembler prolog 1;
* genstrings()

Первый проход: генерирует строки в сегменте данных;

* prolog2()- аssembler prolog 2;
* Второй проход: генерируется ассемблерный код, промежуточные инструкции кода.

Часть кода:

…

for (pc=0; pc<=count; pc++) {

/\* output pcode instruction as comment,

if required by user \*/

if (comment) gencomm(pc);

/\* generate label, if necessary \*/

if (code[pc].lab) genlab(pc);

switch (code[pc].o) {

case ADD : g\_addsub(ADD);

break;

…

case LOD : g\_lod(code[pc].l, code[pc].a);

break;

case STO : g\_sto(code[pc].l, code[pc].a);

break;

case INT : g\_int(code[pc].a, pc);

break;

case JPU : g\_jpu(code[pc].a);

break;

case JPC : g\_jpc(code[pc].a);

break;

case INP : g\_inp();

break;

case PRN : g\_prn();

break;

case PRS : g\_prs(code[pc].a, &pc);

break;

case PRL : g\_prl();

break;

default : fprintf(stderr,

"Unknown opcode %d\n",

code[pc].o);

exit(1);

}

* epilog()-assembler epilog;

Как и сам язык, ассемблеры, как правило, специфичны для конкретной архитектуры, операционной системы и варианта синтаксиса языка. Вместе с тем существуют мультиплатформенные или вовсе универсальные ассемблеры, которые могут работать на разных платформах и операционных системах. Среди последних можно также выделить группу *кросс-ассемблеров*, способных собирать машинный код и исполняемые модули (файлы) для других архитектур и операционных систем.

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер. Также результатом ассемблирования может быть не исполняемый, а объектный модуль, содержащий разрозненные блоки машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью редактора связей может быть получен исполнимый файл.

Наиболее известными ассемблерами для операционной системы DOS являлись Borland Turbo Assembler (TASM), Microsoft Macro Assembler (MASM) и Watcom Assembler (WASM). Также в своё время был популярен простой ассемблер A86. В данной курсовой работе используется ассемблер ТASM.

Тестирование

## Пример 1

Текст программы:

(\* Testprogramm \*)

const n=10000;

var s, i, j;

begin

write('Test');

writeln;

i:= 0;

while i < n do

begin

i:= i + 1;

write(i);

writeln;

s:= 0;

j:= 0;

while j < n do

begin

j:= j + 1;

s:= s + j;

end;

end;

(\* Kommentar \*)

read(i);

end.

Текст программы на ассемблере

dosseg

.model small

;

.stack 10240

;

.data

base dw 0

buffer db 80 dup (?)

blen db 80

blank db ' $'

minus db '-$'

crlf db 13, 10, '$'

bytecnt dw ?

pflag db ?

str01 db 'Test$'

;

.code

start:

mov ax, @data

mov ds, ax

;

mov ax, 0

push ax

mov [base], sp

mov ax, sp

push ax

;

mov cx, 0

call p\_sbs

push dx

mov bp, sp

mov ax, [base]

push ax

mov [base], bp

call pr001

pop dx

mov [base], dx

pop dx

;

call endprocess

pr001 proc

sub sp, 6

mov bx, offset str01

call outstring

mov bx, offset crlf

call outstring

mov ax, 0

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*4], ax

l010:

mov bp, [base]

push word ptr [bp-2\*4]

mov ax, 10000

push ax

pop ax

pop bx

cmp bx, ax

jl m001

sub ax, ax

jmp m002

m001: mov ax, 1

m002: push ax

pop ax

cmp ax, 0

je m003

jmp m004

m003: jmp far ptr l039

m004:

mov bp, [base]

push word ptr [bp-2\*4]

mov ax, 1

push ax

pop ax

mov bp,sp

add [bp],ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*4], ax

mov bp, [base]

push word ptr [bp-2\*4]

pop ax

mov cx, 5

call printint

mov bx, offset crlf

call outstring

mov ax, 0

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*3], ax

mov ax, 0

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*5], ax

…

end start

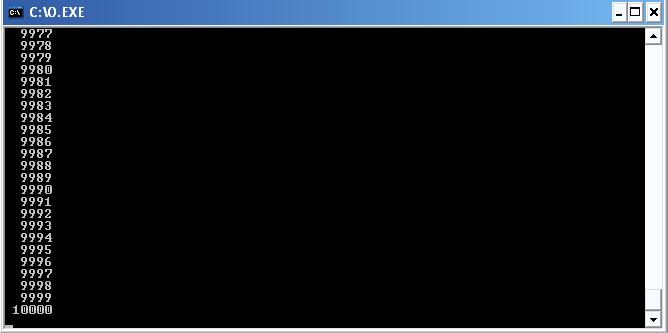


Рисунок 2. Тестирование программы №1.

## Пример 2. «Программа с ошибкой»

Строка программы с ошибкой:

begin writeln('procedure a2'); end;

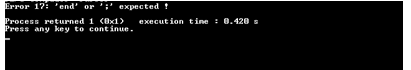


Рисунок 3. Программа с ошибкой

## Пример 3.

Текст программы:

var a,i;

procedure a1;

procedure a2;

begin end;

begin

call a2;

end;

procedure b;

begin

write('procedure b')

end;

procedure c;

begin call b end;

begin call a1; call b; call c;

read(i)

end.

Часть текста программы на ассемблере:

dosseg

.model small

;

.stack 10240

;

.data

base dw 0

buffer db 80 dup (?)

blen db 80

blank db ' $'

minus db '-$'

crlf db 13, 10, '$'

bytecnt dw ?

pflag db ?

str01 db 'procedure b$'

;

.code

start:

mov ax, @data

mov ds, ax

;

mov ax, 0

push ax

mov [base], sp

mov ax, sp

push ax

;

mov cx, 0

call p\_sbs

push dx

mov bp, sp

mov ax, [base]

push ax

mov [base], bp

call pr023

pop dx

mov [base], dx

pop dx

;

call endprocess

pr001 proc

sub sp, 0

add sp, 0

ret

pr001 endp

;

pr003 proc

sub sp, 0

mov cx, 0

call p\_sbs

push dx

mov bp, sp

mov ax, [base]

push ax

mov [base], bp

call pr001

pop dx

mov [base], dx

pop dx

add sp, 0

ret

pr003 endp

;

pr006 proc

sub sp, 0

mov bx, offset str01

call outstring

add sp, 0

ret

pr006 endp

;

pr020 proc

sub sp, 0

mov cx, 1

call p\_sbs

push dx

mov bp, sp

mov ax, [base]

push ax

mov [base], bp

call pr006

pop dx

mov [base], dx

pop dx

add sp, 0

ret

pr020 endp

;

pr023 proc

sub sp, 4

mov cx, 0

call p\_sbs

push dx

mov bp, sp

mov ax, [base]

push ax

mov [base], bp

call pr003

pop dx

mov [base], dx

pop dx

mov cx, 0

…

end start



Рисунок 4. Тестирование программы №3.

## Пример 4

Текст программы:

var h;

procedure io;

var i, j;

begin

write('I/O');

writeln;

write('enter i: ');

read(i);

write('enter j: ');

read(j);

write('i,j:');

write(i, j);

writeln;

end; (\* io \*)

procedure arith;

var i, j, k, l, m, n;

begin

write('Arithmetic');

writeln;

write('enter i: ');

read(i);

write('enter j: ');

read(j);

k:= i+j;

l:= i-j;

m:= i\*j;

n:= i/j;

write('Result d');

writeln;

write('+: ', k);

writeln;

write('-: ', l);

writeln;

write('\*: ', m);

writeln;

write('/: ', n);

writeln;

end; (\* arith \*)

procedure od;

var a;

begin

write('odd-Operator');

writeln;

write('enter a: ');

read(a);

if odd a then

write('odd')

else

write('even');

writeln;

end; (\* od \*)

procedure cond;

var i, j;

begin

write('condition');

writeln;

write('enter i: ');

read(i);

write('enter j: ');

read(j);

if i < j then write('< ');

if i <= j then write('<= ');

if i = j then write('= ');

if i >= j then write('>= ');

if i > j then write('> ');

writeln;

end; (\* cond \*)

procedure whil;

var i, m, n;

begin

write('while');

writeln;

write('enter m: ');

read(m);

write('enter n: ');

read(n);

i:= m-1;

while i < n do

begin

i:= i+1;

write(i);

writeln

end;

end; (\* whil \*)

procedure ifanw;

var i;

begin

write('if');

writeln;

write('enter i: ');

read(i);

if i > 0 then

write('positiv')

else if i = 0 then

write('null')

else write('negativ');

writeln

end; (\* ifanw \*)

begin

call io;

call arith;

call cond;

call od;

call ifanw;

call whil;

read(h);

end.

Часть текста программы на ассемблере: рассмотрим пример процедуры “Arifmetic”.

pr042 proc

sub sp, 12

mov bx, offset str05

call outstring

mov bx, offset crlf

call outstring

mov bx, offset str06

call outstring

mov bx, offset buffer

mov al, [blen]

call instring

mov bx, offset buffer

sub al, 2

mov ch, 0

mov cl, al

call string2number

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*3], ax

mov bx, offset str07

call outstring

mov bx, offset buffer

mov al, [blen]

call instring

mov bx, offset buffer

sub al, 2

mov ch, 0

mov cl, al

call string2number

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*4], ax

mov bp, [base]

push word ptr [bp-2\*3]

mov bp, [base]

push word ptr [bp-2\*4]

pop ax

mov bp,sp

add [bp],ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*5], ax

mov bp, [base]

push word ptr [bp-2\*3]

mov bp, [base]

push word ptr [bp-2\*4]

pop ax

mov bp,sp

sub [bp],ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*6], ax

mov bp, [base]

push word ptr [bp-2\*3]

mov bp, [base]

push word ptr [bp-2\*4]

pop ax

mov bp,sp

imul word ptr [bp]

mov [bp],ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*7], ax

mov bp, [base]

push word ptr [bp-2\*3]

mov bp, [base]

push word ptr [bp-2\*4]

mov bp,sp

pop ax

pop ax

cwd

idiv word ptr [bp]

push ax

pop ax

mov bp, [base]

mov word ptr [bp-2\*8], ax

mov bx, offset str08

call outstring

mov bx, offset crlf

call outstring

mov bx, offset str09

call outstring

mov bp, [base]

push word ptr [bp-2\*5]

pop ax

mov cx, 5

call printint

mov bx, offset crlf

call outstring

mov bx, offset str10

call outstring

mov bp, [base]

push word ptr [bp-2\*6]

pop ax

mov cx, 5

call printint

mov bx, offset crlf

call outstring

mov bx, offset str11

call outstring

mov bp, [base]

push word ptr [bp-2\*7]

pop ax

mov cx, 5

call printint

mov bx, offset crlf

call outstring

mov bx, offset str12

call outstring

mov bp, [base]

push word ptr [bp-2\*8]

pop ax

mov cx, 5

call printint

mov bx, offset crlf

call outstring

add sp, 12

ret

pr042 endp

;

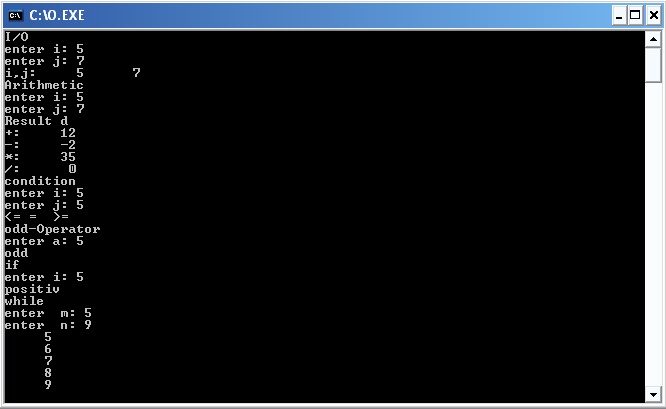


Рисунок 5. Тестирование программы №4.

Заключение

Архитектура компьютеров предлагает массу различных вычислительных ресурсов, которые для получения наивысших результатов должен использовать разработчик компилятора. Пожалуй, наиболее интересно то, что методы оптимизации нашли применение не только в компиляторах. В настоящее время они используются в инструментарии для поиска ошибок в программном обеспечение.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их компиляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью существующих компиляторов. Поэтому, основы теории языков и формальных грамматик, а также практические методы разработки компиляторов лежат в фундаменте инженерного образования по информатике и вычислительной технике.

В данной работе перед нами была поставлена задача создать транслятор для модифицированного языка высокого уровня Pascal. Исходный код проходит три наиболее важных этапа обработки: лексический анализ, синтаксический анализ и, наконец, генерация кода. Результатом этих этапов выступает целевой код, переведенный транслятором на язык Ассемблера версии TASM. Запуск готовой программы через командную строку, рассмотренный в разделе тестирования, продемонстрировал правильность трансляции, что означает, что поставленная задача была корректно выполнена.

Список использованных источников

1. Ахо А., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты, 2-е изд. – М.: Вильямс, 2008. – 1184 с.
2. Зубков С. В. Assembler для DOS, Windows и UNIX, 3-е изд. – М. : ДМК Пресс ; СПб. : Питер, 2004. – 608 с.
3. Иринчеев А.А., Мангадаев А.М. Паскаль в примерах. Учебное пособие. ВСГТУ – Улан-Удэ, 2005. – 76 с.
4. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и написание языка. /Пер, с англ., предисл. и послесл. Д. Б. Подшивалова.— М.: Финансы и статистика, 1982. — 151 с.
5. Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход.  – СПб.: Наука и Техника, 2006. ­­– 320 с.
6. Смирнова Н.Н., Тарасов С.Д. Основы построения компиляторов: учебное-практ. пособие. Балт. Гос. Техн. Ун-т. – СПб., 2007. – 63 c.