Федеральное агентство связи

Государственное образовательное учреждение высшего профессионального образования

«Сибирский государственный университет телекоммуникаций и информатики»

Кафедра вычислительных систем

**Расчётно-графическое задание**

по предмету

«Системное программное обеспечение»

Выполнили:

студенты гр. ВМ-87

Соловьева Н.А.

Гусаров А.А.

Хорошев А.Е.

Проверила:

преподаватель

Молдованова О.В.

Новосибирск 2011

**Содержание**

|  |  |
| --- | --- |
| Введение   1. Справочник по разработанному языку 2. Описание грамматики входного языка в форме Бэкуса-Наура 3. Структура разработанного транслятора 4. Описание средств разработки 5. Примеры тестовых программ на входном языке и результирующих программ, сгенерированных транслятором 6. Текст программы транслятора (листинг программы) | 3  4  7  9  11  12  14 |

**Введение**

Необходимо придумать язык программирования со своей грамматикой в качестве входного языка. Дальше необходимо реализовать транслятор с помощью программных средств разработки лексического анализатора flex и синтаксического анализатора bison, который будет транслировать наш входной язык в язык ассемблера процессоров типа Intel 80х86.

**1.** **Справочник по разработанному языку**

**Идентификаторы**

Идентификатор – это последовательность букв нижнего и верхнего регистра, цифр и знака подчеркивания ”\_”. Первым символом должна быть буква или знак подчеркивания.

**Ключевые слова**

Следующие идентификаторы зарезервированы в качестве ключевых слов и в качестве идентификаторов использоваться не могут:

- int

- string

- write

- read

- if

- else

- while

**Комментарии**

Символ $ открывают комментарий, а символ $ закрывают его.

**Спецификаторы типа**

Спецификаторы типа определяются ключевыми словами:

* int
* string

**Целые числа и строки**

Целые числа определяются типом int, а строка (- последовательность любых букв и символов кроме ”) – string. Преобразования невозможны.

**Аддитивные операторы**

Аддитивные операторы (+) и (-) выполняются слева направо. Результат выполнения оператора (+) есть сумма его операндов. Результат выполнения оператора (-) есть разность операндов.

**Мультипликативные операторы**

Мультипликативныеоператоры (\*) и (/) выполняются слева направо и имеют выше приоритет, чем аддитивные операторы. Результат выполнения оператора (\*) есть произведение его операндов. Результат выполнения оператора (/) есть частное операндов.

**Операторы сравнения**

Операторы сравнения выполняются слева направо и это:

- < (меньше)

- > (больше)

- = (равно)

- != (неравно)

Все операторы выдают 0, если специфицируемое сравнение ложно, и 1, если оно истинно. Тип результата - int.

**Оператор присваивания**

Оператор присваивания (:=) придает значения идентификаторам.

**Оператор логического *И***

Оператор (&) выполняется слева направо. Оператор (&) выдает 1, если оба операнда не равны нулю, и 0 в противном случае. Тип результата - int.

**Оператор логического *ИЛИ***

Оператор (|) выполняется слева направо. Оператор (|) выдает 1, если, по крайней мере, один из операндов не равен нулю, и 0 в противном случае. Тип результата - int.

**Инструкции выбора**

Инструкции выбора осуществляют подборку одной из нескольких альтернатив, определяющих порядок выполнения инструкций:

*- if (выражение) { инструкции }*

*- if (выражение) { инструкции } else { инструкции }*

Оба вида if содержат выражения, которые должны иметь арифметический тип. Сначала вычисляется выражение со всеми его побочными эффектами, результат сравнивается с 0. В случае несовпадения с 0 выполняется первая инструкция. В случае совпадения с 0 для второго типа if выполняется вторая инструкция.

**Циклическая инструкция**

Циклическая инструкция:

*while (выражение) { инструкции }*

В инструкциях while выполнение инструкций повторяется до тех пор, пока значение выражения не станет равным нулю. Выражение должно иметь арифметический тип.

**2. Описание грамматики входного языка в форме Бэкуса-Наура**

<program> -> <declar> <block>

<declar> -> <declar> ‘int’ <id> ‘;’

| <declar> ‘string’ <id> ‘;’

| ɛ

<funct> -> <funct> <state>

| ɛ

<state> -> <id> ‘:=‘ <expr> ‘;’

| ‘write’ ‘(‘ <id> ‘)’ ‘;’

| ‘write’ ‘(‘ <str> ‘)’ ‘;’

| ‘while’ ‘(‘ <bool> ‘)’ <block>

<block> -> ‘{‘ <funct> ‘}’

<expr> -> <expr> ‘+’ <expr>

| <expr> ‘-’ <expr>

| <expr> ‘\*’ <expr>

| <expr> ‘/’ <expr>

| <id>

| <str>

| <digit>

| ‘(‘ <expr> ‘)’

<bool> -> <expr> ‘=’ <expr>

| <expr> ‘<’ <expr>

| <expr> ‘>’ <expr>

| <expr> ‘!=’ <expr>

| <bool> ‘&’ <bool>

| <bool> ‘|’ <bool>

<id> -> ‘\_’ <idl>

| ‘\_’ <digit> <idl>

| <idl>

<idl> -> <letter> <id>

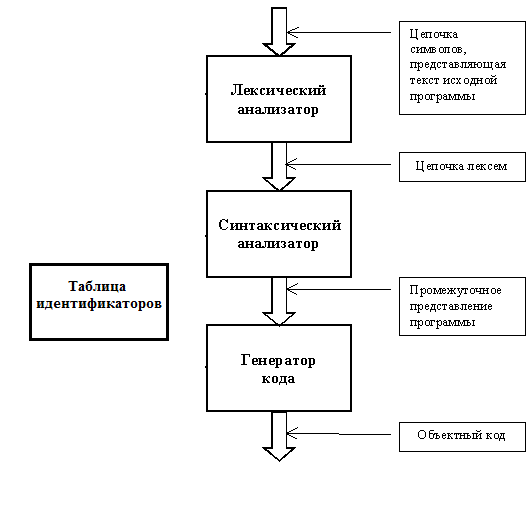
| <letter>

| <letter> <digit>

<letter> -> ‘a’ | ‘A’ | ‘b’ | ‘B’ | ‘c’ | ‘C’ | ‘d’ | ‘D’ | ‘e’ | ‘E’ | ‘f’ | ‘F’ | ‘g’ | ‘G’ | ‘h’ | ‘H’ | ‘i’ | ‘I’ | ‘j’ | ‘J’ | ‘k’ | ‘K’ | ‘l’ | ‘L’ | ‘m’ | ‘M’ | ‘n’ | ‘N’ | ‘o’ | ‘O’ | ‘p’ | ‘P’ | ‘q’ | ‘Q’ | ‘r’ | ‘R’ | ‘s’ | ‘S’ | ‘t’ | ‘T’ | ‘u’ | ‘U’ | ‘v’ | ‘V’ | ‘w’ | ‘W’ | ‘x’ | ‘X’ | ‘y’ | ‘Y’ | ‘z’ | ‘Z’ | ‘\_’

<digit> -> ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

**3.** **Структура разработанного транслятора**



На вход транслятора подаётся текст исходной программы на входном языке. На выходе получается ассемблерный листинг программы, эквивалентной входной.

Транслятор состоит из трёх частей: лексический анализатор, синтаксический анализатор и генератор кода. Во время разбора программы на входном языке лексический анализатор выделяет из текста программы лексемы, определяет ее тип и передаёт её синтаксическому анализатору для следующего разбора. Синтаксический анализатор, в свою очередь, имеет стек, куда попадают терминалы (полученные от лексического анализатора) и нетерминалы (полученные в процессе работы синтаксического анализатора). Если на верхушке стека находится последовательность символов, в точности соответствующая какому-либо из правил грамматики, синтаксический анализатор выполняет свёртку – удаляет из стека все символы, совпавшие с правилом, и заменяет их на нетерминал, порождающий цепочку по найденному правилу. При свёртке синтаксический анализатор выполняет построение узла дерева разбора. Каждый узел имеет тип (константа, переменная, операция) и связанные с этим типом данные (переменная имеет имя, константа – значение, операция – операнды). При этом существуют специальные граничные правила, при свёртках по которым происходит не построение дерева, а генерация кода по построенному дереву. Для объявлений дерево не строится, код генерируется по мере поступления объявлений. При нахождении объявления переменной синтаксический анализатор помещает эту переменную в таблицу идентификаторов. При этом отслеживается попытка повторного объявления идентификатора.

**4. Описание средств разработки**

При разработке транслятора были использованы языки:

- flex (Fast Lexical Analyzer) - это генератор программ, предназначенный для лексической обработки символьных входных данных;

- bison - программный инструментарий, предназначенный для генерации синтаксических анализаторов на основе LALR(1)-грамматик;

- C (написание генератора кода, а также вспомогательных инструментов, таких как функции работы с деревом синтаксического разбора, таблицей идентификаторов и прочее).

**5. Примеры тестовых программ на входном языке и результирующих программ, сгенерированных транслятором**

*Тестовая программа номер 1:*

В качестве примера написана программа вычисления факториала введённого пользователем числа:

int x;

int f;

{

f:=1;

write(“Введите x, чтобы посчитать факториал:”);

read(x);

if (x=0) {f:=1;}

while (x>1)

{

f:=f\*x;

x:=x-1;

}

write(“x! равен”);

write(f);

}

Данная программа будет оттранслирована в следующий код:

*Тестовая программа номер 2:*

В качестве примера написана программа простого присвоения переменной типа string строки любой:

string a;

string b;

{

b:=”def”;

a:=b;

write(a);

}

Данная программа будет оттранслирована в следующий код:

*Тестовая программа номер 3:*

В качестве примера написана программа “Угадай число =)”:

int secret1;

int x;

{

secret1:=42;

write(“Угадай число”);

while(x!=secret1)

{

if (x<secret1) { write(“Бери больше!”); }

else { write(“Переборщил!”); } ;

}

write(“Угадал!Урааааа!=)))”);

}

Данная программа будет оттранслирована в следующий код:

В результате компиляций программы сначала командой *nasm –felf tst?.t* получается объектный файл. Скомпилировав его командой gcc tst?.o

, получим исполняемый файл tst?.out.

1. **Текст программы транслятора (листинг программы)**