Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Лабораторный практикум
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

В 2-х частях

Часть 2

А. А. Уваров, В. А. Прытков, Д. И. Самаль

Компиляторы

Минск БГУИР 2009

УДК 004.4′422 (075.8)
ББК 32.973.26-018.2 я73
С40

Р е ц е н з е н т
профессор кафедры информационно-вычислительных систем
учреждения образования «Военная академия Республики Беларусь»,
канд. техн. наук Д. Н. Одинец

Системное программное обеспечение ЭВМ : лаб. практикум для
С40 студ. спец. 1-40 02 01 «Вычислительные машины, системы и сети». В 2 ч.
Ч. 2 : Компиляторы / А. А. Уваров, В. А. Прытков, Д. И. Самаль. – Минск :
БГУИР, 2009. – 40 с.
ISBN 978-985-488-367-0 (ч. 2)

В практикуме представлены лабораторные работы по курсу «Системное программное обеспечение ЭВМ». Вторая часть практикума посвящена теории компиляторов. В нее включены 4 лабораторные работы, каждая из которых позволяет ознакомиться с различными аспектами разработки и создания компиляторов и интерпретаторов на практике. Все лабораторные работы связаны общим заданием. Студенту предлагается разработать собственный язык программирования и создать для него компилятор или интерпретатор.

УДК 004.4′422 (075.8)
ББК 32.973.26-018.2 я73

Часть 1 «Операционные системы» издана БГУИР в 2009 году.

ISBN 978-985-488-367-0 (ч. 2)

ISBN 978-985-488-366-3

Принятые сокращения

(по алфавиту)

ГК – генератор кода

ЛА – лексический анализатор

СА – синтаксический анализатор

ФБН – форма Бэкуса – Наура

ЯП – язык программирования

Содержание

Лабораторная работа №1
РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: изучить процесс разработки формальной спецификации на процедурный язык программирования и основы построения и функционирования лексических анализаторов; разработать формальную спецификацию ЯП, состоящую из трех частей: общего описания, описания токенов, описания грамматики языка; разработать ЛА для данного языка на основе генератора лексических анализаторов flex.

1.1. РАЗРАБОТКА ФОРМАЛЬНОЙ СПЕЦИФИКАЦИИ ЯЗЫКА

1.1.1. Теоретические сведения

Общие понятия

В общем случае *язык* состоит из знаковой системы (множество допустимых последовательностей знаков), множества смыслов этой системы, соответствия между последовательностями знаков и смыслами. В общем случае знаками могут быть буквы алфавита, математические обозначения, звуки и т.д. Символ (буква) – это простой неделимый знак. *Алфавит* – это счетное множество допустимых символов языка. Обозначим алфавит символом A. *Цепочка символов* (фраза) – это произвольная упорядоченная конечная последовательность символов алфавита. Цепочки будем обозначать греческими буквами α, β, γ и т.д. Произвольность означает, что в цепочку может входить любая допустимая последовательность символов. Упорядоченность означает, что цепочка имеет определенный состав входящих в нее символов, их количество и порядок следования.

Язык L над алфавитом A: L(A) – это некоторое счетное подмножество цепочек конечной длины из множества всех цепочек алфавита A. Множество цепочек языка необязательно должно быть конечным. Длина цепочки символов может быть сколь угодно большой. В общем случае язык можно задать тремя способами:

– перечислением всех допустимых цепочек языка;

– указанием способа порождения цепочек языка (задание грамматики);

– определением метода распознавания цепочек языка.

Лексика – это совокупность слов (словарный запас) языка. *Лексема* (слово, лексическая единица) языка – это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Например, для русского языка лексемами являются слова русского языка. Знаки препинания и пробелы – это разделители, не образующие лексем. Для алгебры лексемами являются числа, знаки операций, обозначения функций и неизвестных величин, для ЯП – ключевые слова, идентификаторы, константы, метки, знаки операций. *Синтаксис* – набор правил, определяющий допустимые конструкции языка, т.е. задающий набор цепочек символов, которые принадлежат языку.

Формальное определение грамматики

Для задания языка используется математический аппарат, который носит название *формальной грамматики*. Теория формальных грамматик занимается описанием, распознаванием и переработкой языков. Существует два основных способа описания отдельных классов языков: с помощью порождающей процедуры и с помощью распознающей процедуры. Порождающая процедура задается с помощью конечного множества правил, называемых грамматикой. Распознающая процедура задается с помощью абстрактного распознающего устройства – автомата.

Грамматику можно описать, используя формальное описание, построенное на основе системы правил. Правило (*продукция*) – упорядоченная пара цепочек символов (α, β). Как правило, их записывают в виде $\alpha \to \beta$ (α порождает β).

Формально грамматика определяется как четверка элементов G (T, N, P, S), где T – конечное непустое множество *терминальных* символов языка, N – конечное непустое множество *нетерминальных* символов, P – конечное множество правил грамматики вида $\alpha \to \beta$, где $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$, S – начальный символ (*аксиома*, корневой нетерминал) грамматики, $S \in N$. Начальный символ всегда нетерминальный. Множество терминальных символов включает в себя символы, входящие в алфавит языка, порождаемого грамматикой. Нетерминальные символы определяют слова, понятия, конструкции языка. Нетерминальные символы будем обозначать прописными латинским буквами A, B, C,…, терминальные символы – строчными – a, b, c… , произвольные цепочки – греческими α, β, γ,… .

Цепочка ω’ *непосредственно выводима* из цепочки ω в грамматике G ($\omega \Rightarrow \omega'$), если $\omega = \xi_1 \varphi \xi_2$, $\omega' = \xi_1 \psi \xi_2$ и $\exists (\varphi \to \psi) \in P$. Цепочка ω’ выводима из цепочки ω в грамматике G ($\omega \overset{*}{\Rightarrow} \omega'$), если $\exists$ цепочка последовательностей $\omega = \omega_0, \omega_1, \ldots, \omega_n = \omega'$: $\omega_{i+1} \Rightarrow \omega_i$, i = 0, 1,…, n-1 либо ω = ω’. Последовательность цепочек $\omega = \omega_0, \omega_1, \ldots, \omega_n$ называется *выводом* цепочки ω_n из цепочки ω_0 в грамматике G. Каждая строка, которую можно вывести из аксиомы – *сентенциальная форма*. Сентенциальная форма, состоящая только из терминалов, представляет собой строку языка.

Для описания правил грамматики можно воспользоваться *формой Бэкуса – Наура (ФБН)*. Для обозначения символа порождения в ФБН используется строка ‘::=’. Нетерминал в ФБН обозначается словом или последовательностью слов, заключенных в < >. Терминальный символ обозначается словом, которое не заключается в < >. В левой части правила всегда находится один нетерминал, а в правой – цепочка символов грамматики. Для обозначения альтернативных цепочек используется символ |. Для обозначения пустого символа грамматики используется цепочка символов нулевой длины.

Регулярные выражения

Одним из способов задания строковых шаблонов являются *регулярные выражения*. Они получили очень широкое распространение при решении задач поиска файлов, поиска и замены строки в тексте и т.д. Существует большое ко-

личество нотаций, в которых могут быть записаны регулярные выражения, но все нотации имеют схожую функциональность. В лабораторном практикуме регулярные выражения будут использоваться для задания токенов грамматики и для построения ЛА.

Для изучения регулярных выражений будем использовать Unix-нотацию. Все символы, которые входят в регулярные выражения, можно разделить на *литеральные символы* и *метасимволы*. Литеральные символы – это символы выходной строки. К ним относятся все символы (буквы, цифры, знаки препинания и пр.), кроме метасимволов.

Метасимволы обозначают некоторое действие и не отображаются напрямую в выходную строку. К метасимволам относятся: [ ] ( ) \ | ? . { } + * / ^ $. Для того чтобы использовать метасимвол в качестве литерального символа, перед ним можно поставить символ \ или заключить его в “”.

Описание метасимволов

. – метасимвол выделения произвольного символа. Обозначает вхождение любого символа, кроме символа конца строки \n.

[] – метасимвол обозначения класса символов. Используется для определения вхождения одного любого символа из числа заключенных в квадратные скобки. Метасимвол, заключенный в квадратные скобки, считается литеральным символом. [xy] обозначает класс символов, в который входят символы x и y. [x-z] обозначает класс символов, в который входят символы из лексикографически упорядоченной последовательности от x до z. Символ ^ внутри квадратных скобок обозначает исключение из класса следующих за ним символов. При этом символ ^ обязательно должен стоять первым, в противном случае он будет считаться литеральным символом:

Регулярное выражение	Строка
[A-z]	*Строка, содержащая один любой латинский символ*
[+-0-9]	*Строка, содержащая либо одну цифру, либо знаки + или –*
[^x]	*Строка, содержащая один любой символ, кроме символа x*

***** и **+** – метасимволы, задающие повторение. Используются для задания повторного вхождения символа или класса символов: например, **x*** обозначает вхождение символа x в выходную строку ноль и более раз, **x+** обозначает вхождение символа x в выходную строку один и более раз.

Регулярное выражение	Строка
a+	*Множество строк длиной от 1 до бесконечности, состоящих только из символа a*
*[ab]**	*Множество строк, состоящих из символов a и b, в том числе и пустая строка*

{} – метасимвол для задания повторений с произвольным числом вхождений. Может иметь три различных формата: {n} – обозначает точное количество повторений, равное числу n; {n,} – обозначает число повторений от n до бесконечности. {n,m} – обозначает число повторений от n до m, при этом n < m. Примеры:

Регулярное выражение	Строка
a{4}	*aaaa*
a{3,5}	*Множество строк {aaa,aaaa,aaaaa}*

/ – метасимвол следования символов. Запись x/y обозначает, что символ x учитывается только тогда, когда за ним следует символ y.

| – метасимвол «или». Запись x|y обозначает вхождение символа x или символа y.

? – метасимвол необязательности. Запись x? – обозначает, что в строке символ x является необязательным.

^ и **$** – метасимволы начала и конца строки во входном тексте. Входной текст может состоять из нескольких строк. Окончанием строки являются символы \r\n или \n либо конец текста. Началом строки является начало текста или место, следующее за предыдущим концом строки.

( ) – метасимволы группировки. Позволяют изменить приоритет операций и формировать группы для других метасимволов:

Регулярное выражение	Строка
^x$	*Строка, состоящая из одного символа x*
(ab)?cd	*Множество строк {abcd,cd }*

Разработка грамматики

Разработка грамматики сводится к определению алфавита, выделению сущностей-нетерминалов, формированию правил. Рассмотрим пример:

Пусть задан алфавит (множество терминальных символов) : {a,b} и допустимые строки языка: { aab, aba, ab}. Построить грамматику в ФБН.

<A>::= ab

<B>::=<A>|a<A>|<A>a.

Нетерминал <B> является аксиомой грамматики.

Возможность описания бесконечного множества цепочек языка с помощью конечного набора правил достигается за счет рекурсий. Рекурсия может быть явной, когда символ определяется сам через себя, и неявной, когда то же самое происходит через цепочку правил. Чтобы рекурсия не была бесконечной, должны существовать и другие правила, определяющие тот же символ.

Рекурсивные правила обычно используются для описания списков однотипных элементов. Например, в качестве таких списков могут выступать спи-

ски операторов в теле функции, список описания аргументов в заголовке процедуры и др.

При переходе от абстрактных построений к составлению грамматики реального ЯП мы сталкиваемся с проблемой выбора алфавита. Алфавит определяет минимальную единицу грамматики и от него зависит сложность ЯП. В качестве алфавита можно использовать и обычные символы, но это не всегда оправдано. На практике грамматику любого ЯП можно разбить на два уровня. На первом уровне находится лексическая грамматика. Ее алфавитом являются символы, а фразами – все допустимые лексемы языка. Описывается она обычно с помощью регулярных выражений. На втором уровне находится синтаксическая грамматика. Ее входом являются токены, а фразами – допустимые тексты программ. Описывается этот уровень грамматики обычно с помощью ФБН. Разбиение грамматики на два уровня позволяет:

– упростить каждый из уровней;

– использовать более подходящий способ описания каждого из уровней;

– использовать различные типы распознавателей для каждого из уровней;

– повысить скорость синтаксического анализа.

Хотелось бы обратить внимание на два термина: «лексема» и «токен». Первый используется в лексической грамматике, а второй – в синтаксической грамматике. Лексема обозначает некоторую допустимую последовательность символов. Токен же является неделимой сущностью и формирует алфавит синтаксической грамматики. Каждому токену соответствует лексема.

Рассмотрим пример двухуровневого описания грамматики для задания списка деклараций прототипов функций. Синтаксис деклараций С подобный.

Пример текста:

void func1 (int param1);

int func2 (int param1, string param2);

В примере токенами будут: типы данных (*void, int, string*), идентификатор (*ident* – цепочка символов, которая начинается с буквы, за которой следует цифра или буква), символы *‘,’* и *‘(‘, ‘)’, ’;’*. Запишем имена токенов и соответствующие им регулярные выражения.

Токен	Регулярное выражение	Токен	Регулярное выражение
VOID	*void*	*COMMA*	*,*
INT	*int*	*LEFT_BRACKET*	*\(*
STRING	*string*	*RIGHT_BRACKET*	*\)*
IDENT	*[a-z_][a-z0-9_]**	*SEMICOLON*	*;*

Запишем правила грамматики в ФБН:

<Тип_параметра>::= INT | STRING

<Тип>::= VOID | < Тип_параметра>

<Декларация параметра> ::= < Тип_параметра> IDENT

<Список_деклараций_параметров>::=<Список_деклараций_параметров>

COMMA <Декларация параметра> | <Декларация параметра>

<Прототип_функции>::= <Тип> IDENT LEFT_BRACKET <Список_деклараций_ параметров> RIGHT_BRACKET SEMICOLON
<Список_прототипов_функций>::=<Список_прототипов_функций> <Прототип_ функции> | <Прототип_функции>.

Аксиомой грамматики является *<Список_прототипов_функций>*. Правила *<Список_деклараций_параметров>* и *<Список_прототипов_функций>* являются рекурсивными и обозначают списки однотипных элементов.

Построения любой грамматики можно вести двумя способами: от аксиомы к терминалам или от терминалов к аксиоме. В первом случае мы начинаем раскрывать в правилах все нетерминалы, начиная с аксиомы. Во втором случае мы начинаем строить правила на основе токенов, обозначая нетерминалы, пока не построим аксиому.

1.1.2. Задание

Необходимо по общим требованиям, выданным преподавателем, разработать свой ЯП и построить для него формальное описание в виде двух уровней грамматики. Для задания лексем использовать регулярные выражения. Описание синтаксической грамматики выполнить в ФБН. Результатом выполнения задания является документ с описанием грамматик, который в электронном виде нужно сдать преподавателю.

1.2. РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

1.2.1. Теоретические сведения

Лексический анализатор (ЛА) – часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка, на основе которых происходит генерация токенов. На вход лексическому анализатору приходит текст, последовательность символов. Выходом лексического анализатора является поток токенов, который затем поступает на вход синтаксического анализатора.

Переход от лексемы к токену однозначен, но приводит к потере некоторой информации. Например, все идентификаторы на уровне синтаксического анализатора будут представлены одной сущностью – *ИДЕНТИФИКАТОР*, несмотря на то, что их лексемы будут различаться. Чтобы не потерять эту информацию и при этом сохранить общность сущностей, вводится понятие *атрибута токена*.

Атрибут токена – это связанная с токеном информация, полученная на этапе лексического анализа, которая не важна на этапе синтаксического анализа, но может понадобиться в дальнейшем. Примерами атрибутов являются имена идентификаторов, значения числовых и строковых констант. Для задания лексем используются шаблоны. Шаблон содержит информацию обо всех допустимых лексемах для данного токена. ЛА состоит из множества шаблонов, которые определяют множество выходных токенов. ЛА производит просмотр текста слева направо. К каждой текущей позиции ЛА применяет все множества имеющихся у него шаблонов, которые проверяют, не является ли строка симво-

лов, состоящая из текущего символа и символов, следующих за ним, лексемой одного из токенов. В случае если такое произошло, то выполняется генерация токена.

Для задания шаблонов используются регулярные выражения. Распознавание токенов производится с помощью специальных процедур – *распознавателей*. Распознавателем называется объект, которому на вход приходит строка символов, а он выдает ответ о принадлежности данной строки языку. Распознавателем для регулярной грамматики является конечный автомат. Наличие алгоритмов, которые строят распознаватели на основе регулярных выражений в автоматическом режиме, позволило разработать программные средства для автоматизации процесса построения ЛА. К подобным средствам относится программа flex, которая способна в автоматическом режиме по специальному описанию на языке lex выполнить синтез ЛА.

Входом для программы flex является формальное описание ЛА на языке lex. Результатом работы программы является исходный код на языке C или C++, который реализует заданный ЛА. В дальнейшем этот файл может быть скомпилирован с использованием любого C/C++ компилятора для создания исполняемого файла.

Программа flex является свободной open-source программой, дополнительную документацию, исполняемые и исходные файлы к которой можно получить по адресу *http://flex.sourceforge.net*.

Структура файла грамматики flex

Файл flex-грамматики имеет следующую структуру:

Секция объявлений
%%
Секция правил трансляции
%%
Секция вспомогательных процедур

Секция вспомогательных процедур содержит C/C++ код, который без изменений переносится в выходной файл.

Секция объявлений содержит объявления регулярных выражений и состояний.

Для объявления регулярного выражения используется следующий синтаксис: *имя регулярное_выражение*. Объявление должно начинаться в строке без отступа. Объявленное регулярное выражение может быть использовано для других объявлений как в этой секции, так и в секции правил. Для использования объявленных регулярных выражений используется следующий синтаксис: *{имя}*.

Состояние ЛА – это механизм, который позволяет управлять активностью отдельных правил. Для декларации состояний используется следующий синтаксис:

%s имя, … , имя
%x имя, … , имя

Различают два типа состояния ЛА: *эксклюзивное* и *инклюзивное*. Эксклюзивное декларируется через %x, а инклюзивное – через %s (более подробно о состояниях ЛА см. в описании секции правил).

В секции объявлений можно разместить C/C++ код, который будет размещен в начале генерируемого выходного файла. В коде можно разместить дополнительные подключаемые файлы, декларации глобальных переменных, функций, макросов. Код должен быть заключён в *%{ }%*, при этом *%{* должен идти с начала строки.

Секция правил трансляции

Секция правил трансляции начинается за первым разделителем %% и продолжается до второго разделителя %% или до конца программы. Эта секция содержит список правил трансляции. Каждое правило представляет собой пару «регулярное выражение» – «действие» и имеет следующий синтаксис:

регулярное_выражение действие

Все правила начинаются с начала строки. В качестве регулярного выражения может использоваться ссылка на объявленное регулярное выражение.

Действие – это последовательность из одного или более операторов языка C/C++. Действие отделяется от регулярного выражения минимум одним пробелом или табуляцией. В одну строку за регулярным выражением может быть записано произвольное количество операторов. Если необходимо разбить действие на несколько строк, то операторы нужно заключить в фигурные скобки, при этом открывающая скобка должна быть в той же строке, что и регулярное выражение.

Действия выполняют генерацию токенов и их атрибутов. Вызов действия происходит всякий раз, когда связанное с ним регулярное выражение обнаруживает во входном потоке соответствующую лексему.

Действие может быть *пустым*, т. е. не содержащим ни одной операции. Для обозначения пустого действия используется символ **;** . Для того чтобы задать одно действие нескольким регулярным выражениям, можно воспользоваться специальным действием, которое обозначается |. Пример:

\n |
\t ;

В примере во входном потоке выделяется символ табуляции или перевода каретки без каких-либо иных действий.

Правила бывают двух типов: *помеченные* и *непомеченные*. Помеченное правило всегда начинается со списка состояний, в которых оно активизируется. Синтаксис:

<имя_состояния, …, имя_состояния> регулярное_выражение действие

В любой момент времени в ЛА существует список активных правил. Первоначально в него включаются все непомеченные правила и правила, помеченные состоянием *INITIAL*. Активация инклюзивного состояния приводит к добавлению в список активных всех правил, которые помечены данным со-

стоянием. Деактивация инклюзивного состояния приводит к удалению всех помеченных только им правил из списка. Активация эксклюзивного состояния приводит к удалению из списка активных всех правил и добавления только тех правил, которые помечены этим состоянием.

Для активации состояния в коде действия нужно указать макрос *BEGIN* и имя состояния. Состояние остается активным до повторного вызова макроса *BEGIN* с именем другого состояния. Первоначальное состояние, которое соответствует всем не помеченным правилам, называется *INITIAL*. Поэтому чтобы перевести ЛА в первоначальное состояние, нужно вызвать *BEGIN INITIAL*. В качестве помечаемого состояния можно использовать *. Это значит, что правило будет активно в любом состоянии. Правило, помеченное несколькими состояниями, будет активным, если активно хотя бы одно состояние из списка. Примеры:

```
%s example
%%
<example>foo      do_something(); BEGIN INITIAL;
bar               something_else(); BEGIN example;
```

Как видно из примера, объявлено инклюзивное состояние *example*. Активация этого состояния происходит в правиле с шаблоном *bar*. После этого активными являются два шаблона: *foo, bar*. После выполнения действия для шаблона *foo* анализатор переходит в первоначальное состояние. Аналогичного можно добиться, используя эксклюзивное состояние:

```
%x example
%%
<example>foo      do_something();BEGIN INITIAL;
<INITIAL,example>bar  something_else();BEGIN example;
```

Вместо пары состояний *INITIAL*, *example* для последнего правила можно использовать * :

```
<*>bar   something_else();BEGIN example;
```

В процессе распознавания лексем во входном потоке может оказаться так, что одна цепочка символов удовлетворяет нескольким правилам и, следовательно, возникает проблема: действие какого правила должно выполняться? Для решения этой проблемы flex использует определенный набор правил:

- выбирается действие того правила, которое распознает наиболее длинную последовательность символов из входного потока;
- если несколько правил распознают последовательности символов одной длины, то выполняется действие того правила, которое записано первым в списке правил.

Особенность реализации действий в lex-программе

Все переменные, объявленные внутри действия, являются локальными и доступны только для этого действия. Для объявления переменных, доступных всем действиям, нужно указать их в секции объявлений.

Кроме обычных переменных в тексте действия можно использовать специальные переменные: *yytext, yyleng, yylval, yyin, yyout*. Переменная *yytext* является символьным буфером и содержит текущую распознанную лексему. Переменная *yyleng* содержит длину в символах текущей распознанной лексемы. Переменная *yylval* используется для создания сгенерированного атрибута текущего токена. В отличие от двух предыдущих переменных она инициализируется самим программистом, а не *flex*. Переменные *yyin* и *yyout* являются указателями на входной и выходной потоки соответственно.

Кроме операторов C/C++ в коде действия могут содержаться специальные lex-макросы и функции. К lex-макросам относятся *BEGIN, REJECT, ECHO*. Макрос *ECHO* выводит в стандартный поток вывода текущую распознанную лексему. Макрос *REJECT* используется для отмены текущего выбранного правила. После вызова этого макроса анализатор должен подобрать следующее наиболее подходящее правило, при этом текущая распознанная лексема возвращается во входной поток и участвует в повторном анализе.

```
she    {s++; REJECT;}
he     {h++; REJECT;}
.      |
\n     ;
```

Этот пример определяет количество слов *she* и *he* во входном потоке. Если из первого правила удалить *REJECT*, то подслово *he*, входящее в слово *she*, не будет найдено, так как ЛА просматривает каждый входной символ только один раз.

Специальные lex-функции можно разделить на две группы: функции для работы с входным потоком и функции для управления процессом синтаксического разбора. К первой группе относятся функции *input* () и *unput*(). Функция *input* читает из входного потока один символ и перемещает указатель на следующий. Функция *unput (char c)* возвращает во входной поток символ *c* и перемещает на него указатель. Пример:

```
{ int i;
char *yycopy = strdup( yytext );
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
        unput( yycopy[i] );
unput( '(' );
free( yycopy );}
```

Приведенное в примере действие заключает найденную лексему в круглые скобки и возвращает ее во входной поток.

Ко второй группе функций относятся *yymore*(), *yyless*(). Функция *yymore*() принуждает выполнить следующее действие ко входному потоку, при этом *yytext* не очищается, как результат в нем содержится конкатенация нескольких распознанных лексем. Пример:

```
mega-ECHO; yymore();
kludge      ECHO;
```

Приведенный анализатор, встретив во входном потоке mega-kludge, выведет mega-mega-kludge.

Функция *yyless*(n) оставляет в буфере *yytext* только *n* символов, а остальные возвращает во входной поток. Понятно, что *n* <= *yyleng*. Пример:

```
Foobar      ECHO; yyless(3);
[a-z]+      ECHO;
```

Встретив во входном потоке слово *foobar*, анализатор выводит *foobarbar*.

Функция *yywrap*() стоит несколько особняком. Она определяется через макрос, поэтому может быть перегружена. Функция вызывается, когда анализатор достигает конца входного потока. Если функция возвращает 1, то процесс разбора прекращается, если 0, то процесс разбора возобновляется. Эта функция дает программисту возможность переопределить поток ввода на другой файл и может быть использована для реализации функциональности аналогичной *#include* в языке C.

1.2.2. Задание

Разработать ЛА по составленной в предыдущем задании спецификации с использованием генератора лексических анализаторов flex. Особое внимание следует уделить описанию однострочных и многострочных комментариев, а также строковых констант. ЛА должен быть выполнен в виде консольного приложения. Файл с текстом для лексического разбора должен задаваться через командную строку. В процессе разбора анализатор должен выводить на экран найденные токены и их местоположение во входном файле (начальный номер строки, столбца).

Лабораторная работа №2
РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Цель работы: изучить процесс разработки синтаксического анализатора (СА); разработать СА для языка программирования на основе генератора синтаксических анализаторов Bison.

2.1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Синтаксисом языка называется набор правил, который описывает допустимые конструкции языка. Синтаксическим анализатором является программа – распознаватель языка, которая проверяет, формирует ли заданная последовательность лексем синтаксически верную программу или нет.

Для описания синтаксиса языка используется грамматика. Для описания ЯП обычно используется *контекстно-свободная грамматика*, которая превосходит регулярную грамматику по возможности описания конструкций языка, но вместе с тем позволяет строить синтаксические анализаторы автоматически по описанию. По определению контекстно-свободной грамматикой является грамматика, в которой правила имеют вид $A \to \beta$, где $A \in N$, $\beta \in V^*$.

Для анализа контекстно-свободной грамматики может быть использован *LR(k)-анализ*, где L обозначает сканирование потока входных токенов слева направо, R – построение обращенных правил порождения, k – количество символов, которые могут быть просмотрены для принятия решения (если k опущено, то оно считается равным единице).

Рассмотрим структуру алгоритма LR-анализа.

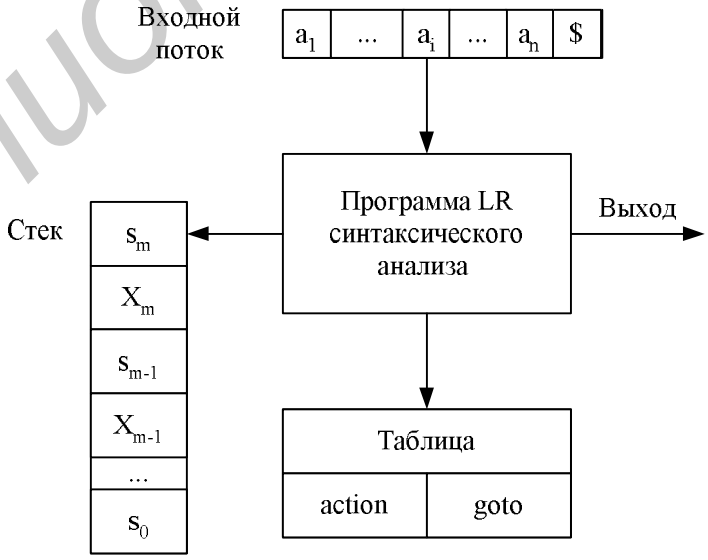


Рис. 1. Схема LR-анализатора

LR-анализатор (рис. 1) состоит из входного потока, выхода, стека, управляющей программы и таблицы из двух частей: действий и переходов. Все анализаторы данного типа имеют одинаковую управляющую программу и отличаются только таблицей, поэтому в процессе построения LR-анализатора необходимо сформировать только один элемент – таблицу. Программа синтаксического анализа считывает символы из входного потока и помещает их в стек, где символы хранятся в следующем виде: $s_0X_1s_1X_2s_2…X_ms_m$, где X_i – это символ грамматики, s_i – символ состояния. Каждый символ состояния обобщает информацию, хранящуюся в стеке ниже него. Символ состояния на вершине стека и текущий входной символ используются в качестве индексов в синтаксической таблице.

Синтаксическая таблица состоит из двух частей – функции действия *action* и функции перехода *goto*. Для того чтобы получить тип выполняемого действия, управляющая программа обращается к ячейке $action[s_i, a_i]$, где s_i – текущее состояние на вершине стека, a_i – текущий входной символ.

Действия бывают четырех типов: 1) перенос (свертка) s, где s – это состояние; 2) свертка в соответствии с продукцией $A \to b$; 3) допуск; 4) ошибка.

Функция состояния *goto* получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние.

Ниже приведены формальные описания действий:

– если $action[s_i, a_i]$ = «перенос s», то СА переносит в стек текущий входной символ a_i и состояние s, определяемое значением $action[s_i, a_i]$, текущим входным символом становится a_{i+1};

– если $action[s_i, a_i]$ = «свертка $A \to b$», то СА вначале снимает с вершины стека $2r$ символов (r символов состояния и r символов грамматики), где r – длина правой части продукции, по которой производится свертка, выводя на вершину стека состояние s_{m-r}. Затем СА вносит в стек левую часть продукции A и s – состояние из ячейки $goto[s_{m-r}, A]$, входной символ при этом не изменяется;

– если $action[s_i, a_i]$ = «допуск», то синтаксический анализ завершается;

– если $action[s_i, a_i]$ = «ошибка», то синтаксический анализатор запускает подпрограмму обработки ошибки.

Из-за высокой сложности алгоритмов построения LR-анализаторов для их создания используются специализированные программные средства, например, генератор анализаторов Bison, позволяющий по формальному описанию построить программу LR-анализатора на языке C или C++.

Программа Bison является свободной open-source программой, дополнительную документацию, исполняемые и исходные файлы к которой можно получить по адресу *http://www.gnu.org/software/bison*.

Синтаксический анализатор всегда работает в паре с лексическим анализатором. Bison генерирует СА, который рассчитан на использование ЛА, сгенерированного с использованием flex. Открытый интерфейс взаимодействия позволяет разработать ЛА и самостоятельно.

Взаимодействие между синтаксическим и лексическим анализатором производится по схеме (рис. 2):

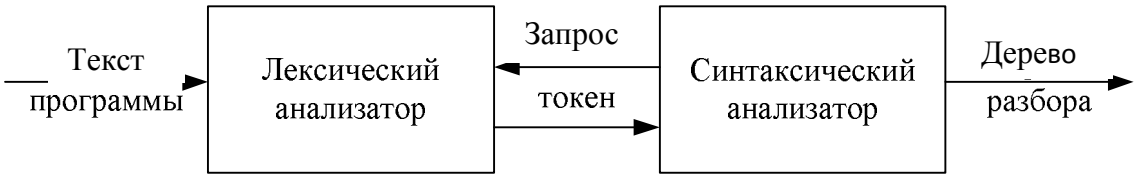


Рис. 2. Схема взаимодействия лексического и синтаксического анализаторов

По запросу от СА лексический анализатор начинает просмотр входного текста до точной идентификации следующего токена, который подается на выход. Обычно в классических системах при организации подобного типа связи ЛА выступает в качестве процедуры, которая вызывается из СА, хотя возможно использование и объектного подхода для построения подобного типа связей.

Структура файла грамматики Bison

Файл грамматики Bison содержит четыре основные секции, показанные здесь с соответствующими разделителями:

%{
Объявления C
%}
Объявления Bison
%%
Правила грамматики
%%
Дополнительный код на C

Комментарии, заключённые в /* ... */, могут появляться в любой секции.

Секция объявлений C содержит макроопределения и объявления функций и переменных, используемых в действиях правил грамматики. Они копируются в начало файла анализатора таким образом, чтобы предшествовать определению *yyparse*. Можно использовать *#include* для получения объявлений из файлов заголовков. Если не нужны какие-либо объявления C, то можно опустить ограничивающие эту секцию *%{* и *%}*.

Секция объявлений Bison содержит объявления, определяющие терминальные и нетерминальные символы, задающие приоритет и т.д. Секция правил грамматики содержит одно или более правил грамматики Bison и ничего более. В данной секции должно быть по меньшей мере одно правило грамматики, при этом первый ограничитель *%%* (предшествующий правилам грамматики) не может быть опущен, даже если это первая строка файла.

Секция дополнительного кода на C в точности копируется в конец файла анализатора, точно так же, как секция объявлений C в начало. Это наиболее удобное место для размещения чего-либо, что необходимо иметь в файле анализатора, но что не нужно помещать перед определением *yyparse*. Если последняя секция пуста, то можно опустить *%%*, отделяющее её от правил грамматики.

Синтаксис правил грамматики

Правило грамматики Bison имеет следующий общий вид:

Нетерминал: строка_символов_грамматики...
;

Пробельные литеры в правилах только разделяют символы. Строка символов грамматики содержит как терминальные символы, так и нетерминальные символы. Терминальные символы могут быть заданы как именами своих токенов, так и лексемами. Лексема задается в виде последовательности одного и более символов, заключенных в одинарные кавычки. Пример:

HEAD: NAME '(' LIST ')'

Для одного нетерминала можно использовать несколько правил, стоящих отдельно или соединённых литерой вертикальной черты |:

Нетерминал: *строка символов грамматики 1...*
| *строка символов грамматики 2...*
;

При необходимости с любым грамматическим правилом можно связать действие – набор операторов языка С, который будет выполняться при свертке нетерминала по данному правилу. Действие не является обязательным элементом правила. Действие заключается в фигурные скобки и помещается вслед за правой частью правила:

Нетерминал: строка_символов_грамматики {операторы С};

В случае если несколько правил объединены символом | , то действие для каждого правила задается отдельно:

Нетерминал: *строка символов грамматики 1* {*действие 1*}
| *строка символов грамматики 2* {*действие 2*};

Если строка символов грамматики не содержит ни одного символа, то это означает, что нетерминал может соответствовать пустой строке:

expseq: */* empty */*
| *expseq1;*

Очень важно также рассмотреть *рекурсивные* правила. Правило называется рекурсивным, если нетерминал его результата появляется также в его правой части. Почти все грамматики Bison должны использовать рекурсию, потому что это единственный способ задать последовательность из произвольного числа элементов. Рассмотрим рекурсивное определение последовательности одного или более выражений, разделённых запятыми:

expseq1: *exp*
| *expseq1 ',' exp* *;*

Последовательность любого вида может быть определена с использованием как левой, так и правой рекурсии, но следует всегда использовать леворекурсивные правила, потому что они могут выполнить разбор последовательности из любого числа элементов, используя ограниченное стековое пространство.

Секция объявлений Bison

Секция объявлений определяет символы, используемые при формулировке грамматики, и типы данных семантических значений.

Все токены (терминальные символы грамматики) должны быть объявлены в секции объявлений. Декларация токенов имеет следующий вид:

%token список_имен_токенов

Благодаря декларации имен токенов Bison отличает имена токенов от имен нетерминалов.

Декларация приоритетов и ассоциативности лексем:

%left список_лексем
%right список_лексем
%nonassoc список_лексем

Лексемы в данных директивах задаются непосредственно или именами. В последнем случае декларация приоритета заменяет также декларацию имени токена, хотя в целях обеспечения наглядности описания желательно наличие в списке директивы *%token* для всего набора токенов.

В процессе работы СА все терминалы заменяются некоторыми целыми числами. Назначение целого числа токену может происходить автоматически или декларативно. Для декларативного задания нужно использовать первое объявление имени токена или его лексемы. Например:

%token IF 1000
%left `z` 800

По умолчанию номера токенам присваиваются следующим образом. Номером токена, обозначаемого лексемой, считается числовое значение данной лексемы, рассматриваемое как однобайтное число. Именованным токенам присваиваются номера в соответствии с очередностью их объявления (начиная с номера 257). Специальный токен *error* получает значение 256.

По умолчанию в качестве аксиомы грамматики выступает нетерминал первого правила в секции правил. Для явного задания аксиомы используется следующий синтаксис: *%start имя_аксиомы*.

Конфликты грамматики и способы их устранения

Перед тем как перейти к рассмотрению конфликтов, нужно ещё раз вспомнить понятия сдвига и свертки. Сдвиг – это перенос входного токена на вершину стека. Свертка – это замена в стеке n элементов, находящихся на вершине стека, на один нетерминал. Выбор действия в каждый момент времени

происходит на основе значения входного терминала и текущего состояния на вершине стека.

Пример первого конфликта сдвига/свертки можно рассмотреть на примере грамматики конструкции *if then else*:

```
if_stmt:    IF expr THEN stmt
          | IF expr THEN stmt ELSE stmt      ;
```

IF, *THEN* и *ELSE* – терминальные символы. Когда токен *ELSE* прочитан и предпросмотрен, содержимое стека как раз подходит для свёртки по первому правилу. Но допустимо также и сдвинуть *ELSE*, потому что это приведёт к последующей свёртке по второму правилу.

Ситуация, когда допустимы как сдвиг, так и свёртка, называется конфликтом «сдвиг/свёртка». Bison разработан так, что он разрешает эти конфликты, выбирая сдвиг, за исключением случаев, когда объявления приоритета операций указывают обратное.

Другая ситуация, когда возникают конфликты «сдвиг/свёртка», – это использование арифметических выражений. Здесь сдвиг – не всегда предпочтительное решение. Объявления приоритета операций позволят вам указать *Bison*у, когда выполняется сдвиг, а когда – свёртка.

Рассмотрим следующий фрагмент неоднозначной грамматики (неоднозначной потому, что входной текст '1 - 2 * 3' может быть разобран двумя разными способами):

```
expr:    expr '-' expr
       | expr '*' expr
       | expr '<' expr
       | '(' expr ')'
```

Предположим, что анализатор рассмотрел лексемы '1', '-' и '2'. Должен ли он выполнить свёртку по правилу для операции вычитания? Это зависит от лексемы, следующей после '2'. Если следующая лексема – '*' или '<', у нас есть выбор: как сдвиг, так и свёртка позволят удачно завершить разбор, но с разными результатами.

Чтобы решить, что должен делать Bison, рассмотрим результаты. Если сдвинуть следующую лексему знака операции *op*, она должна быть свёрнута первой, чтобы дать возможность свернуть разность. Результатом (на самом деле) будет '1 - (2 *op* 3)'. С другой стороны, если вычитание свернуть до сдвига *op*, результатом будет '(1 - 2) *op* 3'. Поэтому ясно, что выбор сдвига или свёртки должен зависеть от относительного приоритета операций.

А что можно сказать о таком входном тексте, как '1 - 2 - 5'? Должен ли он означать '(1 - 2) - 5' или '1 - (2 – 5)'? Для большинства операций мы предпочитаем первый вариант, называемый левой ассоциативностью. Второй вариант – правая ассоциативность – желателен для операций присваивания. Выбор левой или правой ассоциативности – это вопрос о том, будет анализатор выбирать

сдвиг или свёртку, когда стек содержит '1 - 2' и предпросмотренная лексема – '-'. Для выполнения сдвига оператор должен иметь правую ассоциативность.

Bison позволяет указать требуемый выбор с помощью объявлений приоритета операций *%left* и *%right*. Каждое такое объявление содержит список лексем, являющихся операциями, приоритет и ассоциативность которых определяется объявлением. Объявление *%left* делает все эти операции левоассоциативными, а *%right* – правоассоциативными. Возможен и третий вариант – *%nonassoc*, устанавливающий, что появление одной операции два раза подряд будет синтаксической ошибкой.

Относительный приоритет различных операций управляется порядком, в котором они объявляются. Первое в файле объявление *%left* или *%right* задаёт операции самого низкого приоритета, следующее такое объявление задаёт операции, приоритет которых немного выше и т.д.

В данном примере нам нужны были следующие объявления:

```
%left '<'
%left '-'
%left '*'
```

В более развернутом примере, поддерживающем также другие операции, нам следует объявлять их группами равного приоритета. Например, '+' объявляется вместе с '-':

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

Разрешение конфликтов с использованием механизма приоритетов основано на сравнении приоритета рассматриваемого правила с приоритетом предпросмотренной лексемы. Приоритет правила равен приоритету последнего терминального символа среди его компонентов. Если приоритет лексемы выше, то выполняется сдвиг, если ниже – свёртка. Если приоритеты одинаковы, выбор делается исходя из ассоциативности данного уровня приоритета. Не все правила и не все лексемы имеют приоритет. Если у правила или у предпросмотренной лексемы нет приоритета, по умолчанию производится сдвиг.

Часто приоритет операции зависит от контекста. Например, знак "минус" обычно имеет очень высокий приоритет для унарной операции, и несколько меньший (меньший, чем умножение) для бинарной операции.

Объявления приоритета Bison – *%left*, *%right* и *%nonassoc* – для данной лексемы могут использоваться только один раз, поэтому лексема имеет только один приоритет, объявленный таким образом. Чтобы воспользоваться контекстно-зависимым приоритетом, вам нужно задействовать дополнительный механизм – модификатор правил *%prec*.

Модификатор *%prec* объявляет приоритет конкретного правила, указывая терминальный символ, приоритет которого следует использовать для данного

правила. Этот терминальный символ необязательно должен появляться в самом правиле. Синтаксис модификатора таков: *%prec терминальный_символ*. Он ставится после всех компонентов правила. В результате правилу присваивается не тот приоритет, который должен быть присвоен обычным способом, а приоритет символа *терминальный_символ*.

Пример решения проблемы унарного минуса:

```
%left '+' '-'
%left '*'
%left UMINUS
……..……
exp:   ...
       | exp '-' exp
       | '-' exp %prec UMINUS
```

Второй тип конфликта – это «свёртка/свёртка». Конфликт «свёртка/свёртка» возникает, когда есть два (или более) правила, применимых к одной последовательности входного текста. Это обычно свидетельствует о серьёзной ошибке в грамматике. Данный тип конфликта не устраняется ни автоматически, ни через механизм приоритетов.

Bison разрешает конфликты «свёртка/свёртка», выбирая правило, появляющееся в грамматике раньше, но полагаться на такое решение проблемы очень рискованно. Каждый конфликт «свёртка/свёртка» должен быть изучен и по возможности исключён при модификации грамматики.

Восстановление от ошибок

В тексте любой программы могут встречаться ошибки. Ошибкой называется такая последовательность терминалов, которая не может быть выведена из грамматики языка. Синтаксический анализатор должен по возможности обнаружить все ошибки и сообщить о них пользователю. Для каждого состояния, которое может находиться на вершине стека, существует допустимое множество токенов. Если предпросмотренный токен не принадлежит этому множеству, это свидетельствует о наличии ошибки.

После обнаружения первой ошибки анализатору нужно восстановиться для продолжения разбора. Если восстановление не произойдет, то СА завершит свою работу. Для восстановления от ошибки СА должен содержать правила, распознающие специальный токен *error*. Анализатор Bison генерирует лексему *error* каждый раз, когда обнаружена синтаксическая ошибка. Если вы предусмотрели правило для распознавания этой лексемы в текущем контексте, разбор может быть продолжен.

Например:

```
stmnts: /* empty */
    | stmnts '\n'
```

```
| stmnts exp '\n'
| stmnts error '\n'
```

Четвёртое правило в этом примере говорит, что ошибка, за которой следует переход на новую строку, является допустимым дополнением для любого *stmnts*.

Что случится, если синтаксическая ошибка будет обнаружена внутри *exp*? Правило восстановления после ошибки (если его интерпретировать строго) применимо к последовательности, состоящей в точности из *stmnts*, *error* и перехода на новую строку. Если ошибка обнаружена внутри *exp*, вероятно, в стеке после последнего *stmnts* будут находиться некоторые дополнительные лексемы и подвыражения, а до литеры новой строки нужно будет прочитать ещё несколько лексем. Поэтому указанное выше правило обычным способом неприменимо.

Но Bison может принудительно привести ситуацию к правилу, отбрасывая часть семантического контекста и часть входного текста. Вначале он отбрасывает состояния и объекты в стеке до тех пор, пока не вернётся к правилу, в котором приемлема лексема *error* (это означает, что уже разобранные подвыражения будут отброшены, вплоть до последнего завершённого *stmnts*). В этот момент может быть выполнен сдвиг лексемы *error*. Потом, если нельзя выполнить сдвиг старой предпросмотренной лексемы, анализатор читает лексемы и отбрасывает их до тех пор, пока не найдёт подходящую лексему. В данном примере Bison читает и отбрасывает входной текст, пока не обнаружит литеру новой строки, так что можно применить четвёртое (в примере) правило.

Выбор правил грамматики для ошибок – это выбор стратегии восстановления после ошибки. Простая и полезная стратегия – при обнаружении ошибки пропустить остаток текущей входной строки или текущего оператора: stmnt: error ';'.

Также полезно восстанавливать до закрывающего ограничителя, соответствующего уже разобранному открывающему ограничителю. В противном случае закрывающий ограничитель, вероятно, оказался бы без пары, и вызвал новое, ложное сообщение об ошибке:

```
primary: '(' expr ')'
    | '(' error ')'    ;
```

Стратегии восстановления после ошибки неизбежно связаны с предположениями. Когда предположение неверно, одна синтаксическая ошибка часто приводит к появлению других. В вышеприведённом примере правило восстановления после ошибки предполагает, что ошибка вызвана неправильным входным текстом внутри одного *stmnt*. Допустим, что вместо этого внутрь правильного *stmnt* вставлена точка с запятой. После того как правило восстановления после ошибки произведёт восстановление от первой ошибки, сразу же будет обнаружена другая синтаксическая ошибка, поскольку текст, следующий за лишней точкой с запятой, также не является верным *stmnt*.

Чтобы предотвратить поток сообщений об ошибках, анализатор не будет выводить сообщения об ошибках, произошедших вскоре после первой. Вывод сообщений возобновится только после того, как будет успешно произведён сдвиг трёх лексем подряд.

Следует иметь в виду, что правила, принимающие лексему *error*, могут содержать действия, так же как и любые другие правила.

Вы можете возобновить вывод сообщений об ошибках немедленно, используя в действиях макрос *yyerrok*. Но если вы примените макрос в действии правила обработки ошибки, сообщения об ошибках не будут подавляться.

Сразу после обнаружения ошибки предыдущая предпросмотренная лексема анализируется заново. Если это невозможно, используется макрос *yyclearin* для очистки этой лексемы.

Взаимодействие ЛА и СА

Анализатор Bison представляет собой функцию на C, называющуюся *yyparse*. Если разбор завершён успешно, то *yyparse* возвращает значение 0. Значение 1 возвращается, если разбор не удался (возврат вызван синтаксической ошибкой).

В действии можно завершить работу *yyparse*, используя следующие макросы: *YYACCEPT* – немедленный возврат со значением 0 (сообщение об удачном разборе). *YYABORT* – немедленный возврат со значением 1 (сообщение об ошибке).

Функция лексического анализатора – *yylex*. Bison не создаёт эту функцию автоматически. В простых программах *yylex* часто определяется в конце файла грамматики Bison. Если *yylex* определена в отдельном исходном файле, для него необходимо сделать доступными макроопределения типов лексем. Для этого при запуске Bison используйте параметр *'-d'*, чтобы он записал эти макроопределения в отдельный файл '*имя*.tab.h', который можно включить в другие исходные файлы. Значение, возвращаемое *yylex*, должно быть или числовым кодом токена, или 0 – для обозначения конца входного текста.

Если в правилах грамматики объявлены имена токенов, то это имя становится в файле анализатора макросом C, определением которого будет числовой код, соответствующий этому токену. Таким образом, *yylex* может использовать для обозначения токена его имя.

Если в правилах грамматики на токен ссылаются с помощью однолитерной константы, то числовой код этой литеры также является кодом токена. Таким образом, *yylex* может просто вернуть код этой литеры. Этот интерфейс разрабатывался так, чтобы выход утилиты flex мог быть без изменений использован как определение *yylex*. Каждое действие в описании ЛА должно заканчиваться возвращением типа распознанного токена, а именно – оператором *return*.

2.2. ЗАДАНИЕ

На основе имеющейся спецификации и разработанного ЛА выполнить разработку синтаксического анализатора с использованием программы Bison. СА должен представлять консольную программу, которой на вход через командную строку подается файл для анализа. Анализатор должен уметь обнаруживать более одной ошибки, если таковые присутствуют, т. е. уметь восстанавливаться. Одновременно с программой должны быть разработаны два текста для тестирования при сдаче: первый – без ошибок, второй – с двумя и более ошибками. В результате работы программы на экран должна быть выдана информация об обнаруженных синтаксических конструкциях (нетерминалах) и сообщения обо всех обнаруженных ошибках.

Лабораторная работа №3
ПОСТРОЕНИЕ ДЕРЕВА СИНТАКСИЧЕСКОГО РАЗБОРА

Цель работы: изучить процесс построения дерева синтаксического разбора; дополнить разработанные лексический и синтаксический анализаторы сбором семантических атрибутов и построить дерево синтаксического разбора.

3.1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

ЯП нельзя описать только синтаксисом, нужно обязательно описать его семантику. Для СА язык – это просто последовательность токенов, он не несет информации о том, каким образом данная последовательность токенов должна быть превращена в последовательность полезных для пользователя действий, т. е. о работе программы. Под семантикой языка мы понимаем поведение его конструкций, способ его обработки и получаемый результат.

Дерево синтаксического разбора – это дерево, которое содержит информацию о процессе свертки. Элементами дерева обычно являются нетерминалы. Корень дерева – это корневой нетерминал или аксиома. Листьями дерева являются терминалы. Связи между элементами отражают процесс свертки. Символы грамматики, находящиеся в правой части некоторой продукции, связаны с нетерминалом в левой части. Чтобы использовать дерево синтаксического разбора для анализа семантики программы, необходимо с каждым элементом дерева связать атрибут. Атрибут является носителем семантической информации, которая не участвует в процессе синтаксического анализа, но требуется на более поздних этапах.

Атрибуты подразделяют на синтезируемые и наследуемые. Синтезируемые атрибуты в некотором узле дерева зависят только от атрибутов узлов-потомков. Они служат для передачи информации по дереву снизу вверх. Наследуемые атрибуты в некотором узле являются функциями атрибутов его узла-предка и(или) атрибутов узлов-потомков этого предка.

В большинстве случаев дерево синтаксического разбора является избыточным для семантического анализа, поэтому вместо него часто используют семантическое дерево. Семантическое дерево по структуре связей схоже с деревом синтаксического разбора. Из семантического дерева исключена несущественная для семантического анализа информация, которая присутствует в дереве синтаксического разбора. Элементами дерева являются описания некоторых семантических конструкций языка, которые включают все необходимые атрибуты элементов дерева синтаксического разбора.

Рассмотрим пример. Дерево синтаксического разбора оператора if-else языка С будет содержать открывающие и закрывающие скобки условного выражения, операторные скобки и др. Оператор if-else в семантическом дереве будет связан с элементом, который обозначает условие, и двумя элементами, которые обозначают список операторов для двух веток: if и else.

Семантическое дерево является входом для последующих этапов компиляции или интерпретации.

Генерация семантических значений в ЛА

Для сохранения и передачи семантического значения распознанной лексемы в ЛА используется переменная *yylval.*Тип переменной определяется в файле описания синтаксического анализатора через макрос *YYSTYPE*. По умолчанию этот тип задан как *double*. Обычно в качестве типа используется некоторый *union*. Это связано с тем, что семантические значения для разных типов токенов могут различаться. Например, для идентификатора семантическое значение – это его имя, для строковой константы – это строка символов, для числовой константы – это целое или вещественное число. Для хранения такого разнообразного множества типов можно использовать либо структуру, либо объединение. Объединение предпочтительнее, т. к. в один момент времени токен может иметь только один тип семантического значения, а объединение позволяет их хранить в одной области памяти. В качестве альтернативы при использовании в качестве целевого языка C++ для ЛА и СА можно использовать указатель на некоторый базовый класс. Семантические атрибуты всех токенов представляются классами, которые имеют общего предка.

Bison предлагает специальный способ для декларации типа объединения семантического значения. В секции объявления Bison нужно воспользоваться объявлением *%union.* Пример:

```
%union {   double val;   char *str;}
```

Тогда для сохранения значения строковой константы в ЛА нужно написать следующий код:

```
yylval.str = new char [yyleng + 1];
strcpy(yylval.str, yytext);
```

Семантические значения в СА

Каждый символ грамматики в СА может иметь собственное семантическое значение. Для доступа к семантическому значению символа грамматики, расположенной в правой части правила, нужно воспользоваться специальной переменной *$n*, где n – порядковый номер символа грамматики в правой части правила, который начинается с единицы. Тип переменной будет соответствовать типу YYSTYPE, или заданному объединению. Пример:

```
exp:    ...
        | exp '+' exp  { $$ = $1 + $3; }
```

В примере *$1* соответствует первому *exp*, а *$3* – второму *exp*. Семантическое значение терминала, заданного лексемой *'+'*, не используется.

Переменная *$$* обозначает семантическое значение генерируемого нетерминала, который находится в левой части правила. В процессе синтаксического

разбора семантические значения хранятся в стеке СА и удаляются в процессе свертки одновременно с соответствующими символами грамматики.

Если в действии для правила не задаются никакие операции над семантическими переменными, то Bison применяет действие по умолчанию: $$ = $1. Для пустого правила отсутствует действие по умолчанию; каждое пустое правило должно иметь явное действие, за исключением случая, при котором его значение никогда не будет использоваться.

Переменная $*n* с нулевым или отрицательным значениями *n* допустима для ссылки на символы грамматики, находящиеся в стеке перед соответствующими текущему правилу символами грамматики.

Если в качестве типа данных для семантического значения используется объединение, то возникает проблема с обращением к определенному полю объединения. Так как переменная $*n* будет являться объединением, а не скалярным типом, необходимо выбирать требуемое поле при каждом обращении.

Для этого используется следующий синтаксис: *$<имя_поля>n.*

В этом объявлении значение *n* – порядковый номер символа грамматики в правой части правила, а *имя_поля* указывает, в каком поле объединения хранится семантическое значение.

При задании типа семантического значения для нетерминала в левой части правила используется следующий синтаксис: *$<имя_поля>$.*

Теперь рассмотрим пример:

```
%union {  int itype;  double dtype;}
%%
exp:  exp '+' exp  { $$<itype>$ = $<itype>1 + $<itype>3; }
```

В примере объявлено объединение с двумя полями. В действии в качестве семантического значения нетерминала *exp* явно выбирается поле *itype*, которое имеет целочисленное значение.

Кроме явного задания типа семантического значения при обращении, в секции объявления описания СА можно связать тип семантического значения и любой символ грамматики. Для терминалов существует два способа задания семантического значения: первый – в объявлении токена, второй – при объявлении приоритета:

```
%token <имя_поля> имя_токена
%left <имя_поля> имя_токена
```

При задании типа семантического значения для нетерминала используется следующий синтаксис:

```
%type < имя_поля > список нетерминалов
```

Тип семантического значения, заданный при обращении к переменной, имеет более высокий приоритет, чем тип, который задан в секции декларации.

Построение семантического дерева

Семантическое дерево может быть построено на основе дерева синтаксического разбора. Но можно пропустить эту стадию и выполнять построение семантического дерева в процессе синтаксического анализа.

Семантическое дерево служит для проверки отсутствия семантических ошибок и является входом для генератора кода или интерпретатора. Существует несколько типов семантических ошибок.

Первый тип – отсутствие декларации имени. К этому типу ошибок относятся все ошибки, связанные с использованием некоторого имени, для которого должно существовать определение, однако данное определение на самом деле не существует. Например, имя переменной, для которой не существует объявления, имя функции, для которой не объявлен прототип и пр.

Второй тип – повторное использование имени. Возникает при добавлении объекта с уже существующим именем в область видимости. Например, повторная декларация переменной или функции.

Третий тип – неверное приведение типов. К этому типу можно отнести все ошибки, связанные с проверкой и приведением типов. Например, присвоение строковой переменной целочисленного значения, вызов функции с неверным типом и количеством параметров.

Список задач, решаемых в процессе семантического анализа:

- проверка наличия объявления переменной в области видимости в момент ее использования;
- проверка наличия объявления функции/процедуры в области видимости в момент ее вызова;
- проверка повторной декларации переменной и функции/процедуры в моменты их деклараций;
- проверка типов при выполнении оператора присвоения, при вычислении вложенных выражений в условном операторе и операторах цикла;
- проверка соответствия количества и типов формальных и действительных параметров функций/процедур.

Организация дерева семантического разбора должна быть ориентирована на упрощение решения поставленных выше задач. Поэтому в ряде случаев можно отказаться от построения единого дерева, а разбить его на несколько независимых списков и иерархических описаний. Разбиение соответствует естественному разбиению программы. На самом верхнем уровне программу можно представить как список функций. Каждая из функций содержит описание входных параметров: тип и имя, а также один операторный блок.

Операторный блок в зависимости от реализации может содержать просто список операторов или состоять из списка деклараций либо списка операторов. Блок может выступать в качестве оператора для другого родительского блока. Обычно каждый блок определяет некоторую область видимости, поэтому с блоком связывается список декларируемых в нем переменных.

Каждый оператор может содержать выражения и вложенные операторные блоки. Для представления выражений обычно используется дерево синтаксического разбора.

Для решения задачи по проверке наличия объявления переменной в области видимости в момент ее использования необходимо иметь общую область видимости для каждой точки программы. В простом процедурном языке область видимости для конкретного оператора формируется как объединение областей видимости всех родительских блоков, списка параметров функции и глобальных объявлений. Для программной реализации области видимости удобно использовать стек списков. Первым элементом в стек помещается список глобальных объявлений, вторым – список параметров текущей функции, третий элемент содержит список переменных основного блока функции. Далее при входе в каждый следующий блок в стек добавляется еще один элемент, который содержит список деклараций текущего блока. При выходе из операторного блока соответствующий ему элемент из стека удаляется. Для проверки наличия объявления переменной нужно просмотреть все списки стека.

Для решения задачи по проверке наличия объявления функции или процедуры в момент ее вызова достаточно одного списка имен объявленных процедур и функций.

Наличие типизированных переменных и параметров функций требует вычисления типа присваиваемого выражения. Тип выражения вычисляется на основе известных типов переменных и констант, которые в него входят, и выполняемых над ними операций. Вычисление типа выражения происходит снизу вверх на основе специальных таблиц вычисления типов. Таблица вычисления типа содержит связь между двумя входными типами операции и получаемым результатом. Например:

Операторы	Операнд 1	Операнд 2	Результат
+,-,*,/	int	int	int
+,-,*,/	double	double	double
+,-,*,/	int	double	double
+,-,*,/	double	Int	double

В примере приведена таблица для арифметических операций и двух типов данных. Если для данного оператора не удалось найти в таблице требуемого сочетания типов операндов, это говорит об ошибке типов.

Также в большинстве языков существует неявное приведение типов. Например, любой целый тип меньшей разрядности приводится к типу с большей разрядностью: *char, short, int, long.* Любой целый тип может быть приведен к вещественному типу.

Разрабатываемый язык имеет строгую типизацию, которая позволяет вычислить тип любого выражения на этапе семантического анализа. Приведение типов всегда выполняется однозначно, чтобы исключить неопределенность.

При вызове процедуры или функции необходимо вычислить типы всех действительных параметров и сравнить их с типами формальных параметров. Если тип и число параметров не совпадают, это говорит об ошибке.

Содержимое семантического описания оператора зависит от его типа. Оператор присвоения будет содержать имя переменной левой части и выражение правой. Цикл *for* содержит один операторный блок и три выражения: первое выражение используется для инициализации переменной цикла и не имеет типа, второе вычисляет условия окончания цикла и имеет булевский тип, третье определяет изменение переменной цикла и не имеет типа. По аналогии можно описать все остальные операторы языка.

3.2. ЗАДАНИЕ

На основе имеющейся спецификации и разработанных ЛА и СА разработать семантический анализатор для построения семантического дерева. Семантический анализатор должен представлять консольную программу, которой на вход через командную строку подается файл для анализа. Анализатор должен выполнять проверки использования необъявленных функций и переменных, повторного объявления функций и переменных, соответствия типов. Одновременно с программой должны быть разработаны два текста для тестирования: первый – без ошибок, второй – со всеми возможными типами семантических ошибок. В результате работы программы на экран должна быть выдана информация о созданных семантических объектах и сообщения обо всех обнаруженных ошибках.

Лабораторная работа №4
ГЕНЕРАТОР КОДА

Цель работы: изучить процесс построения генератора кода и интерпретатора; в соответствии с заданием разработать генератор кода или интерпретатор, которые в качестве входной информации используют семантическое дерево.

4.1. РАЗРАБОТКА ГЕНЕРАТОРА КОДА

4.1.1. Теоретические сведения

Разработка генератора кода (ГК) является завершающим этапом в создании компилятора. ГК – это программный модуль, который выполняет генерацию исходного кода программы на целевом языке. Генерация кода происходит на основе дерева семантического разбора, построенного на предыдущих этапах. В качестве целевого языка может выступать любой ЯП. Это может быть как язык высокого уровня, так и различные типы ассемблеров.

Построение оптимального целевого кода является сложным процессом. Сложность проистекает из многокритериальности оценки генерируемого кода (объем кода, объем памяти, используемой в процессе работы программы, скорость исполнения кода), а также из-за наличия семантического разрыва между языком описания исходной программы и языком целевой машины. В качестве примера семантического разрыва можно привести различия между высокоуровневыми объектно-ориентированными языками и ассемблером архитектуры x86.

Для рассмотрения в лабораторной работе выделим два типа архитектур: регистровую и стековую. Стековая архитектура в качестве операндов инструкций использует стек. Регистровая архитектура соответственно в качестве операндов инструкций использует регистры. В лабораторной работе в качестве целевых языков предлагается выбрать один из следующих: ассемблер x86 как представитель регистровой архитектуры целевой машины; MSIL (Microsoft Intermediate Language) или Java Byte Code как представители целевых машин со стековой архитектурой. Основная задача разрабатываемого ГК – построить текст программы на целевом ассемблере, который может быть скомпилирован и корректно исполнен на целевой машине.

Основное внимание в теоретической части будет уделено построению ГК для ассемблера x86. Генераторы для других ассемблеров проще и могут быть построены по аналогии.

По-другому задачу ГК можно сформулировать как задачу отображения сущностей и конструкций исходного языка на сущности и конструкции языка ассемблер. В исходном языке можно выделить сущности для хранения информации (глобальные и локальные переменные, параметры процедур), функции, выражения, операторы языка.

Программа на языке ассемблер состоит из секции данных и секции программ. В секции данных объявляются статические переменные. Статическими называются переменные, под которые память выделяется в процессе запуска

программы, и адреса которых в дальнейшем не изменяются. В секции программ находится основной блок программы и объявление процедур. Использование статических переменных в инструкциях осуществляется по именам.

Тривиальным является только отображение глобальных переменных и деклараций функций, которые отображаются соответственно на статические переменные и объявление процедур. Отображение остальных сущностей исходного языка может быть проведено различными способами.

Отображение локальных переменных и параметров функций

Характерной особенностью локальных переменных и параметров функция является ограниченная область видимости. Поэтому в разных частях программы могут встречаться локальные переменные, различные по типам, но одинаковые по именам. При этом в случае вложенных или рекурсивных функций может существовать несколько копий одной и той же локальной переменной, которые имеют разные значения.

Существует два основных способа отображения локальных переменных: отображение на статические переменные или на динамические переменные в стеке.

В первом случае под все локальные переменные отводится некоторая статическая область памяти, называемая пулом. Каждая локальная переменная ставится в соответствие некоторой ячейке пула. В качестве более простого варианта для ГК можно использовать подход, основанный на прямом отображении локальной переменной на статическую. Декларация любой локальной переменной в коде программы приводит к декларации статической переменной в ассемблерном коде. Основной проблемой при прямом отображении является генерация имени статической переменной. Имя статической переменной строится обычно на основе имени переменной, порядкового номера декларации в процедуре и имени процедуры. Основное требование к ним – это уникальность отображаемых имен для различных переменных. Основным недостатком приведенного метода является невозможность реализации рекурсивных вызовов функций, т. к. при отображении переменных второй и последующих функций будут использоваться одни и те же переменные.

Вторым вариантом является отображение локальных переменных на стек архитектуры x86. Для каждой процедуры создается свой стековый фрейм, в котором содержатся локальные переменные. Стековый фрейм выделяется в стеке при входе в процедуру, а при выходе из процедуры стек освобождается. Так как каждая процедура использует свой стековый фрейм, то это дает возможность реализовать рекурсивный вызов функций.

Для адресации локальной переменной в стековом фрейме используется относительный адрес. Относительный адрес в архитектуре x86 используется относительно вершины стекового фрейма. В архитектуре x86 для работы со стеком используются регистры SP и BP. Регистр SP указывает на вершину стека. Регистр BP указывает на младший адрес стекового фрейма. Для адресации локальной переменной к значению регистра BP добавляется смещение.

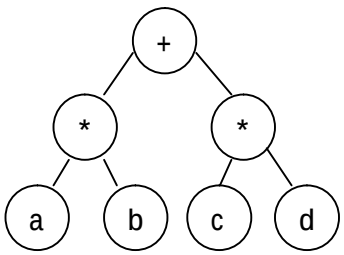
Так как стековый фрейм выделяется в начале процедуры, то необходимо знать его размер. Проблемы могут возникнуть, если исходный ЯП поддерживает объявление переменных в любом месте программы, а не только в начале процедуры. Тогда нужно вначале определить объем, требуемый для всех локальных переменных процедуры, а потом выделить стековый фрейм нужного размера.

Локальные переменные могут быть не только скалярными переменными, но и массивами. Доступ к ячейке массива осуществляется по индексу. Массив отображает в блок памяти, который может быть выделен двумя вышеописанными подходами. Для доступа к элементу массива необходимо вычислить его смещение относительно начала массива. Для этого необходимо знать размер элемента массива. В самом простом случае для одномерного массива, если размер элемента равен 1, 2 или 4 байта, можно обойтись масштабируемым режимом адресации x86 архитектуры: [*bx + 2*si*]. Начало элемента массива помещается в регистр *bx*, индекс – в регистр *si* или *di*. Множитель перед индексом обозначает размер элемента.

Двухмерные массивы обычно размещаются в памяти построчно. По младшему адресу блока памяти располагается первая строка, затем – вторая и т. д. Для доступа к элементам двухмерного массива необходимо также знать размер строки. Тогда вычисление адреса элемента будет идти в два этапа: на первом этапе вычисляется адрес начала требуемой строки, затем – адрес элемента относительно начала строки. При отображении многомерного массива ГК должен знать размер строки для каждого массива и генерировать код вычисления адреса для каждого обращения.

Отображение выражений

Под выражениями в данном случае мы понимаем как арифметические, так и логические выражения. Выражение состоит из операндов и операции. Сложное выражение содержит более двух операндов. Для представления выражений в семантическом дереве используется дерево синтаксического разбора. Каждый внутренний элемент дерева является операцией, которая связана с двумя операндами. Листьями дерева являются константы и локальные переменные. Ниже приведен пример дерева для выражения *a*b+c*d*. В качестве операндов для операций * выступают переменные, а для операции + результаты вычисления произведений.



Вычисление сложного выражения состоит из последовательности вычислений простых выражений, которые содержат не более двух операндов. Для получения правильной последовательности необходимо, чтобы код вычисления операндов операции предшествовал вычислению операции. В приведенном выше примере можно выделить три простых выражения: a*b, c*d, сумма произведений. Для выполнения правила необходимо, чтобы коды двух произведений стояли перед кодом суммы. Для получения такого порядка обычно применяется рекурсивная генерация кода. Каждый элемент в дереве «знает» о своих непосредственных аргументах и отправляет в выходной поток код своих аргументов перед своим кодом.

При генерации кода сложного выражения нужно решить проблему хранения результатов промежуточных вычислений. Архитектура x86 поддерживает операции максимум над двумя операндами. Поэтому для реализации операций над тремя и более операндами нужно использовать промежуточное хранилище. Например, *a*b+c*d* требует хранения промежуточных результатов вычислений двух произведений.

Для хранения промежуточных значений можно использовать либо временные переменные, либо стек.

При хранении промежуточных значений вычислений в локальных переменных каждому простому выражению соответствует некоторая локальная переменная, куда оно сохраняет результат своего вычисления. Любое вышестоящее выражение, которое использует его результат, будет загружать его из этой переменной. Поскольку область видимости для результатов выражения распространяется максимум на один оператор, то можно реализовать пул временных локальных переменных. При переходе к генерации следующего оператора языка пул сбрасывается.

В случае использования стека для хранения промежуточного результата вычисление каждого выражения заканчивается внесением результата в стек. Тогда для использования своих аргументов любое вышестоящее выражение должно извлечь их значения с вершины стека.

Отображение вызова функций и передачи параметров

Отображение вызова функции происходит в команду CALL, которая сохраняет адрес следующей команды в стек и выполняет переход на первую инструкцию вызываемой функции. Передача аргументов функции происходит либо по значению, либо по указателю. В первом случае значение каждого аргумента помещается в стек, во втором случае в стек помещается значение адреса переменной, содержащей этот аргумент.

Еще одним вопросом является то, каким образом будет очищаться стек от помещенных туда параметров: вызывающей функцией или вызываемой функцией. В лабораторной работе рекомендуется использовать передачу параметров по значению и чтобы за удаление параметров отвечала вызывающая функция.

Отображение операторов языка

Операторы языка представляются в семантическом дереве в виде структур, которые содержат всю необходимую информацию для генерации кода. Отображение операторов языка связано с генерацией некоторых шаблонов. В структуре операторов языка можно выделить встроенные выражения и операторные блоки. Рассмотрим условный оператор *if-then-else*. Оператор содержит встроенное выражение для определения условия и два встроенных операторных блока: первый выполняется, если условие истинно, а второй – если условие ложно.

Рассмотрим шаблон этого оператора:

```
[код условного выражения]
pop ax
cmp ax,1
jne ELSE
[код блока then]
jmp END
ELSE:
[код блока else]
END:
```

Первоначально генерируется код условного выражения. Для хранения промежуточных результатов используется стек. Оператор *pop* извлекает вычисленное значение условного выражения из стека. Оператор *cmp* определяет истинность или ложность выражения. В данном примере принято, что истинное значение равно единице, все остальные значения являются ложными. В коде шаблона присутствуют две метки. Первая метка указывает начало блока *else*, вторая – окончание оператора. При генерации кода необходимо предусмотреть механизм обеспечения их уникальности, так как ассемблер не допускает существования нескольких одинаковых меток.

Самым простым механизмом для обеспечения уникальности меток является использование глобального счетчика. При каждом создании новой метки с именем конкатенируется значение счетчика, затем значение счетчика увеличивается на единицу.

Реализация ввода-вывода

Для реализации функций ввода-вывода можно воспользоваться функциями прерывания 21h или вызовом функций стандартной библиотеки C.

4.1.2. Задание

Разработать генератор кода для указанного преподавателем целевого языка. В качестве входа для ГК использовать семантическое дерево, построенное на предыдущих этапах. Полученный с помощью ГК код должен быть скомпилирован с использованием компилятора целевого языка и исполнен на машине. Основным критерием работоспособности ГК является работоспособность

сгенерированных им программ. Логика сгенерированной программы должна соответствовать логике исходной программы.

4.2. РАЗРАБОТКА ИНТЕРПРЕТАТОРА КОДА

4.2.1. Теоретические сведения

Разработка интерпретатора сводится к задаче вычисления семантического дерева. Исполнение оператора языка может изменить значение переменных программы или вывести некоторое значение на экран. Процесс вычисления семантического дерева – это процесс вычисления переменных программы.

Для хранения значения переменных используется стек фреймов. Каждый фрейм содержит переменные для отдельного операторного блока. При входе в операторный блок добавляется новый фрейм, куда помещаются переменные блока. В процессе исполнения инструкций блока они могут обращаться ко всем переменным в области видимости и модифицировать их. По завершении блока фрейм удаляется.

Обработка деклараций переменных сводится к добавлению в текущий фрейм требуемых структур для хранения значения заданного типа.

Значения глобальных переменных хранятся в отдельном списке, чтобы обеспечить доступ к ним из любой функции. При вызове функции для нее создается новый стек фреймов. Первый фрейм в этом стеке содержит значение параметров функции.

Для вычисления выражений используется рекурсивный обход дерева синтаксического разбора, при помощи которого представляются выражения в семантическом дереве. Для того чтобы вычислить выражение, нужно вычислить все аргументы, которые в свою очередь также могут являться выражениями и поэтому им также требуется вычислить свои аргументы и т. д.

Результатом вычисления выражения является некоторое значение, которое может либо быть присвоено переменной либо использоваться внутри оператора.

Вычисление блока операторов сводится к последовательности вычислений каждого оператора.

4.2.2. Задание

Разработать интерпретатор языка. Интерпретатор является консольной программой, которой на вход через командную строку задается интерпретируемый файл. В качестве входа для интерпретатора использовать семантическое дерево, построенное на предыдущих этапах. Консоль программы используется для выдачи сообщений интерпретируемой программы и ввода пользовательских данных. Основным критерием работоспособности интерпретатора является корректность выводимых на консоль данных интерпретируемой программы.

Литература

1. Ахо, А. Компиляторы. Принципы, технологии, инструменты / А. Ахо, Р. Сети, Д. Ульман. – М. : Вильямс, 2003.

2. Молчанов, А. Ю. Системное программное обеспечение : учеб. для вузов / А. Ю. Молчанов. – СПб. : Питер, 2003.

3. Соколов, А. П. Системы программирования: теория, методы, алгоритмы : учеб. пособие / А. П. Соколов. – М. : Финансы и статистика, 2004.

4. Компаниец, Р. И. Системное программирование. Основы построения трансляторов / Р. И. Компаниец, Е. В. Маньков, Н. Е. Филатов. – СПб. : Коронапринт, 2004.

5. Опалева, Э. А. Языки программирования и методы трансляции / Э. А. Опалева, В. П. Самойленко. – СПб. : БХВ-Петербург, 2005.

6. Карпов Ю. Г. Теория автоматов : учеб. для вузов / Ю. Г. Карпов. – СПб. : Питер, 2003.

7. Мозговой, М. В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход / М. В. Мозговой. – СПб. : Наука и техника, 2006.

8. Хантер, Р. Проектирование и конструирование компиляторов / Р. Хантер. – М. : Финансы и статистика, 1984.

Св. план 2009, поз. 64

Учебное издание

Уваров Андрей Александрович
Прытков Валерий Александрович
Самаль Дмитрий Иванович

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Лабораторный практикум
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

В 2-х частях

Часть 2

Компиляторы

Редактор Г. С. Корбут
Корректор Е. Н. Батурчик

Подписано в печать 9.01.2009.	Формат 60×84 1/16.	Бумага офсетная.
Гарнитура «Таймс».	Печать ризографическая.	Усл. печ. л. 2,44
Уч.-изд. л. 2,0.	Тираж 150 экз.	Заказ 549.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6