Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.Я. Расторгуев

Учебное пособие

РАЗРАБОТКА ЭЛЕМЕНТОВ УЧЕБНОЙ СИСТЕМЫ ПРОГРАММИРОВАНИЯ. КОМПИЛЯТОР С ЯЗЫКА ВЫСОКОГО УРОВНЯ

Санкт-Петербург
2012

АННОТАЦИЯ

Данное учебное пособие является методическим руководством при подготовке и проведению первой из трех лабораторных работ, направленных на разработку учебной системы программирования в составе:

- компилятора с языка высокого уровня,
- компилятора с АССЕМБЛЕРА,
- абсолютного или непосредственно-связывающего загрузчика, конструктивно объединенного с эмулятором и пошаговым отладчиком объектного (target) микропроцессора, работающего на платформе, состоящей из технологического (host) микропроцессора IBM PC и одной из операционных сред MS Windows или Linux.

Целью проведения этого цикла работ является более глубокое изучение лекционного материала курса и приобретение практических навыков в процессе разработки учебной системы программирования путем адаптации программ-макетов ее элементов. Каждая из работ цикла рассчитана при соответствующей домашней подготовке в среднем на восемь часов работы в лаборатории кафедры.

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Данная лабораторная работа имеет своей целью получение практических навыков построения компилятора с языка высокого уровня (ЯВУ), являющегося одним из элементов системы программирования, образующих в совокупности следующий технологический конвейер:

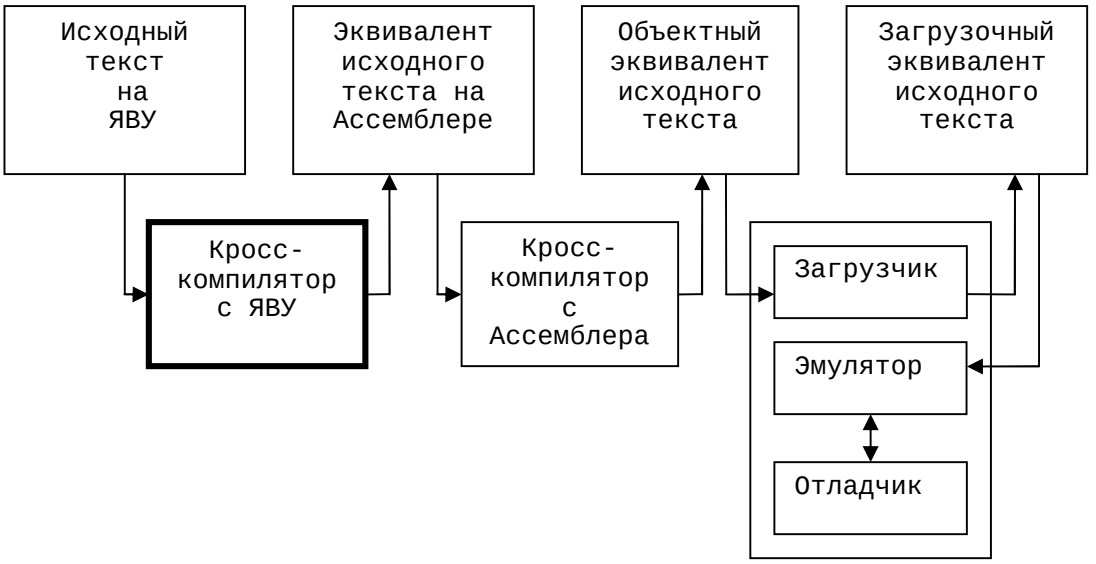


Рис.1. Состав учебной системы программирования

При этом предполагается то, что данная система программирования работает на технологической ЭВМ (IBM PC) и является по существу кросс-системой для объектной ЭВМ (IBM 370). В этой системе:

- в качестве языка высокого уровня (ЯВУ) выбран язык, образованный из подмножества языковых конструкций ПЛ1, а исходная программа готовится в виде текстового файла технологической ЭВМ с расширением *.pli,
- язык АССЕМБЛЕРА сформирован из языковых конструкций АССЕМБЛЕРА IBM 370, а ассемблеровский эквивалент исходной программы формируется в виде текстового файла технологической ЭВМ с расширением *.ass,
- объектный эквивалент исходной программы готовится в формате объектных файлов операционной системы ОС IBM 370 и хранится в виде двоичного файла технологической ЭВМ с расширением *.tex;
- загрузочный эквивалент исходной программы представляет собой машинный код IBM 370, запоминаемый в области ОЗУ технологической ЭВМ, являющейся зоной загрузки для эмулятора объектной ЭВМ.

2. КОМПИЛЯТОР С ЯЗЫКА ВЫСОКОГО УРОВНЯ. ТЕЗИСЫ ЛЕКЦИЙ ПО КУРСУ

2.1. Выбор языка, грамматики и алгоритма компилятора

Рассмотрим особенности конструирования компиляторов с языка высокого уровня на следующем примере, составленном на базе подмножества языка ПЛ/1:

```
EXAMP1: PROC OPTIONS ( MAIN );

 DCL A BIN FIXED ( 31 ) INIT ( 111B );
 DCL B BIN FIXED ( 31 ) INIT ( 100B );
 DCL C BIN FIXED ( 31 ) INIT ( 101B );
 DCL D BIN FIXED ( 31 );

 D = A + B - C;

END EXAMP1;
```

Охарактеризуем подробнее выбранное подмножество ПЛ/1:

а) Нами отобраны для реализации четыре типа операторов подмножества ПЛ/1:

- оператор начала процедурного блока (пролога программы),
- оператор конца процедурного блока (эпилога программы),
- оператор объявления переменной в двух вариантах (с начальной инициализацией и без инициализации),
- оператор присваивания.

б) Далее для упрощения рассмотрения и реализации мы будем игнорировать динамическую природу объявлений переменных, считая их статическими.

в) В операторе присваивания применяются операции одинакового старшинства, что исключает необходимость использования скобок.

Выделенное подмножество ПЛ/1 образует самостоятельный язык со своими синтаксическими и семантическими особенностями.

Известно, что один и тот же язык м.б. описан многими грамматиками. Договоримся далее использовать для описания синтаксиса метавыражения Бэкуса. Заметим, что уже сама эта договоренность определяет то, что описываемый этими выражениями язык не может выходить за рамки класса контекстно-свободных языков (КС-языков) в сторону усложнения.

Один из возможных вариантов описания синтаксиса выбранного подмножества ПЛ/1 выглядит следующим образом:

1. <PRO> ::= <OPR><TEL><OEN>
2. <OPR> ::= <IPR>:PROC_OPTIONS(MAIN);
3. <IPR> ::= <IDE>
4. <IDE> ::= <BUK> | <IDE><BUK> | <IDE><CIF>
5. <BUK> ::= A | B | C | D | E | M | P | X
6. <CIF> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
7. <TEL> ::= <ODC> | <TEL><ODC> | <TEL><OPA>
8. <ODC> ::= DCL_<IPE>_BIN_FIXED(<RZR>); |
   DCL_<IPE>_BIN_FIXED(<RZR>)INIT(<LIT>);
9. <IPE> ::= <IDE>
10. <RZR> ::= <CIF> | <RZR><CIF>
11. <LIT> ::= <MAN>B
12. <MAN> ::= 1 | <MAN>0 | <MAN>1
13. <OPA> ::= <IPE>=<AVI>;
14. <AVI> ::= <LIT> | <IPE> | <AVI><ZNK><LIT> |
    <AVI><ZNK><IPE>
15. <ZNK> ::= + | -
16. <OEN> ::= END_<IPR>

Здесь использованы следующие метасимволы и символы:

"<" и ">" - левый и правый ограничители нетерминального символа,

"::=" - метасимвол со смыслом "равно по определению",

"|" - метасимвол альтернативного определения "или",

"_" - терминальный символ со смыслом "пробел",

"<PRO>" - нетерминал "программа",

"<OPR>" - нетерминал "оператор пролога программы",

"<IPR>" - нетерминал "имя программы",

"<IDE>" - нетерминал "идентификатор",

"<BUK>" - нетерминал "буква",

"<CIF>" - нетерминал "цифра",

"<TEL>" - нетерминал "тело программы",

"<ODC>" - нетерминал "оператор declare",

"<IPE>" - нетерминал "имя переменной",

"<RZR>" - нетерминал "разрядность",

"<LIT>" - нетерминал "литерал",

"<MAN>" - нетерминал "мантисса",

"<OPA>" - нетерминал "оператор присваивания арифметический",

"<AVI>" - нетерминал "арифметическое выражение",

"<ZNK>" - нетерминал "знак",

"<OEN>" - нетерминал "оператор эпилога программы".

З а м е ч а н и я:

1) Формально грамматика описывает выбранное подмножество ПЛ/1 как самостоятельный КС-язык, однако, для точного определения принадлежности языка к КС-типу нужны дополнительные исследования. Так, скажем, можно уточнить не является ли этот язык более простым, например, линейным (автоматным).

2) Грамматика должна быть выбрана так, чтобы по возможности исключить многозначность вывода одинаковых языковых конструкций.

3) Выбранная нами грамматика не подвергалась экспертизе на выполнение критериев, указанных в пп. 1) и 2) и, следовательно, может оказаться неоптимальной, что, как следствие, приводит к усложнению алгоритма компилятора и ухудшению его временных характеристик.

4) Алгоритм компиляции м.б. либо обшим (годящимся для любого языка данного класса, скажем КС-языка), либо специальным, т.е. ориентированным на конкретный язык. Мы выберем для дальнейшего рассмотрения первый способ, т.е. построим компилятор для языков КС-типа, но настроим его на синтаксис и семантику выбранного подмножества ПЛ/1.

Проектирование компилятора с языка высокого уровня следует выполнять по типовому плану, состоящему из следующих этапов:

№ этапа	Содержание этапа
1	Постановка задачи.
2	Определение состава базы данных (БД).
3	Определение форматов элементов БД.
4	Синтез алгоритма.
5	При необходимости выделение из блок-схемы алгоритма элементов (модулей), нуждающихся в дальнейшей детализации.
6	Рекурсивное повторение этапов 1 – 5, для выделенных модулей.

2.2. Постановка задачи

Попробуем проанализировать особенности конструирования компилятора с АССЕМБЛЕРА (см. учебное пособие "Разработка элементов учебной системы программирования. Компилятор с АССЕМБЛЕРА) для заимствования оттуда проектных решений, годящихся для нашего случая.

Успех конструирования компилятора с АССЕМБЛЕРА был обеспечен следующими обстоятельствами:

1. Обрабатываемый исходный текст был по предположению изначально безошибочным, что позволило не включать в состав компилятора с АССЕМБЛЕРА анализаторов лексики и синтаксиса.

2. Текст был структурирован, т.е. при его составлении были выполнены следующие правила:

а) На каждой перфокарте с первой позиции записан ровно один оператор АССЕМБЛЕРА.

б) Внутри перфокарта разбита на поля:
- "метка" (с 1-й позиции);
- "код операции" (с 10-й позиции);
- "операнды" (с 16-й позиции).

Выполнение этих соглашений означает по существу то: что мы на входе АССЕМБЛЕРА использовали не просто линейный, а структурированный исходный текст. Действительно, используя правила из пп.2, мы можем на линейном тексте:

```
EXAMP1    START 0
          USING *,15
          ...
```

построить его структурированный вариант в следующем виде.

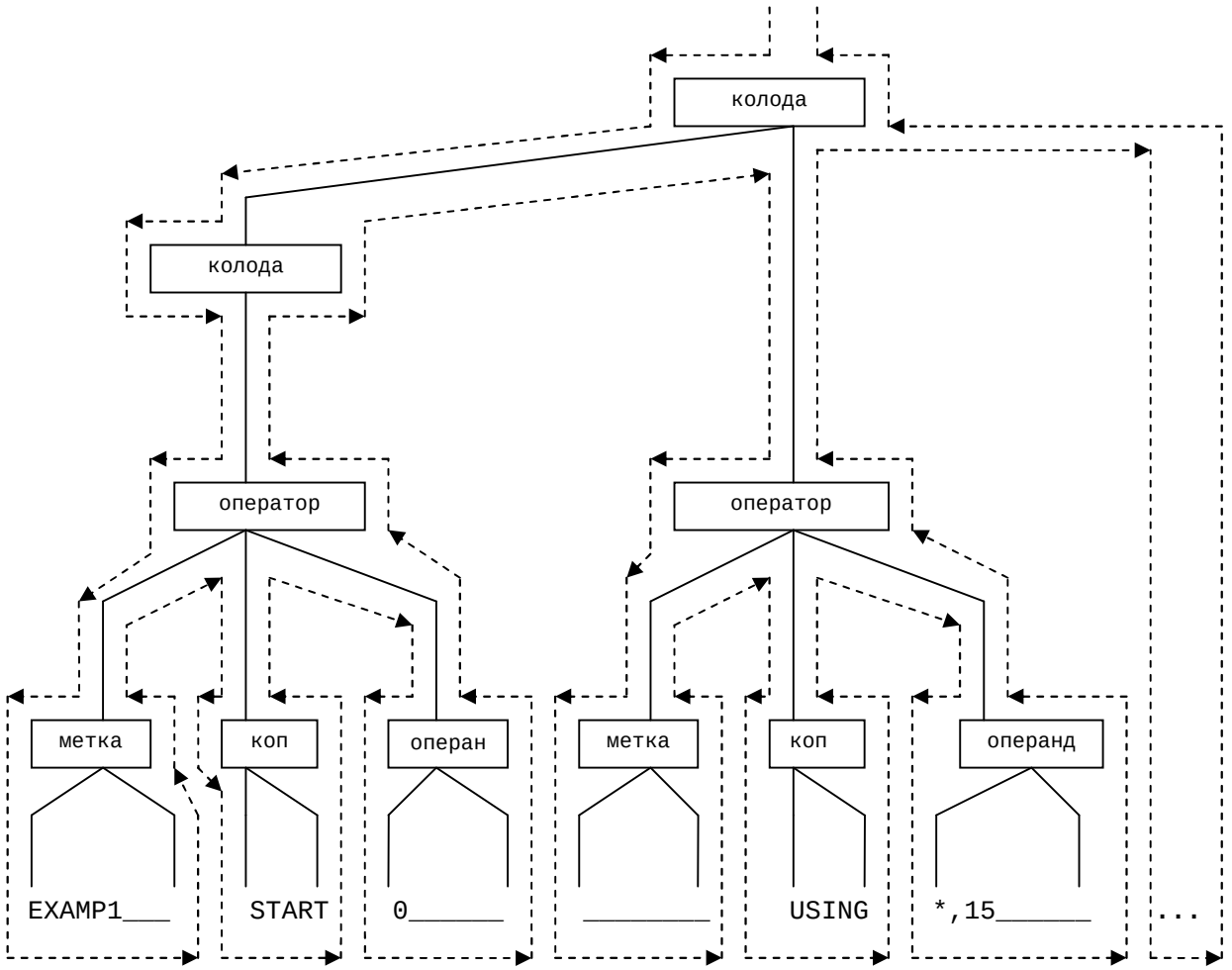


Рис.2. Синтаксический граф для исходного текста на Ассемблере

Очевидно, что построить структуру линейного исходного текста, можно не только вручную с использованием правил из пп. 2, а и автоматически, вводя дополнительный просмотр линейного исходного текста, называемый синтаксическим анализом, в результате которого строится структура линейного текста, известная как дерево синтаксического разбора или, иначе, - синтаксический граф.

Рассмотрим подробнее алгоритм неявного использования синтаксического графа при обработке текста компилятором с АССЕМБЛЕРА. Вспомним то, что мы обрабатывали исходную колоду карт оператор за оператором в следующей последовательности для каждого оператора:

1) Метка.
2) Код операции.
3) Операнды.

Этот порядок соответствует траектории движения по структуре синтаксического графа, нарисованной пунктирной линией.

Если теперь мысленно:

- отождествить узлы синтаксического графа с функциями, названия которых совпадают с названиями узлов,
- принять подчиненные по иерархии узлы в качестве аргументов таких функций,
- договориться о том, что момент вычисления каждой функции наступает в момент завершения кругового обхода на 360 градусов по траектории движения вокруг соответствующего узла,

то становится очевидным, что:

1) Синтаксический граф м.б. интерпретирован как графическая форма записи суперпозиции функций.

2) Траектория обхода задает порядок вычисления суперпозиции функций.

3) Количество разноименных функций, участвующих в суперпозиции, конечно (что следует из конечности синтаксического графа).

4) Среди функций могут быть и рекурсивные.

Просуммируем результаты анализа проектирования компилятора с АССЕМБЛЕРА, полученные на данный момент, сделав для дальнейшего изложения следующие выводы:

1) Успех нашей реализации компилятора с АССЕМБЛЕРА базировался в том числе и на неявном использовании синтаксического графа, который явно компилятором не строился, но строго соблюдался организационным способом (путем введения строгих формализованных правил ручного форматирования исходного текста).

2) Синтаксический граф м.б. получен автоматически путем введения в алгоритм компилятора с АССЕМБЛЕРА соответствующего блока – синтаксического анализатора, так как это делается в компиляторах с любых других КС-языков.

3) Понятие "синтаксический граф" имеет смысл только для КС-языка и поэтому далее предлагаемый подход к конструированию компиляторов, базирующийся на использовании синтаксического графа, справедлив только для множества КС-языков высокого уровня, в который к счастью попадают или почти попадают все известные и потенциально возможные языки программирования.

С учетом сказанного приступим к проектированию компилятора для выбранного нами языка высокого уровня, который участвует в обработке исходной программы согласно следующей схеме:

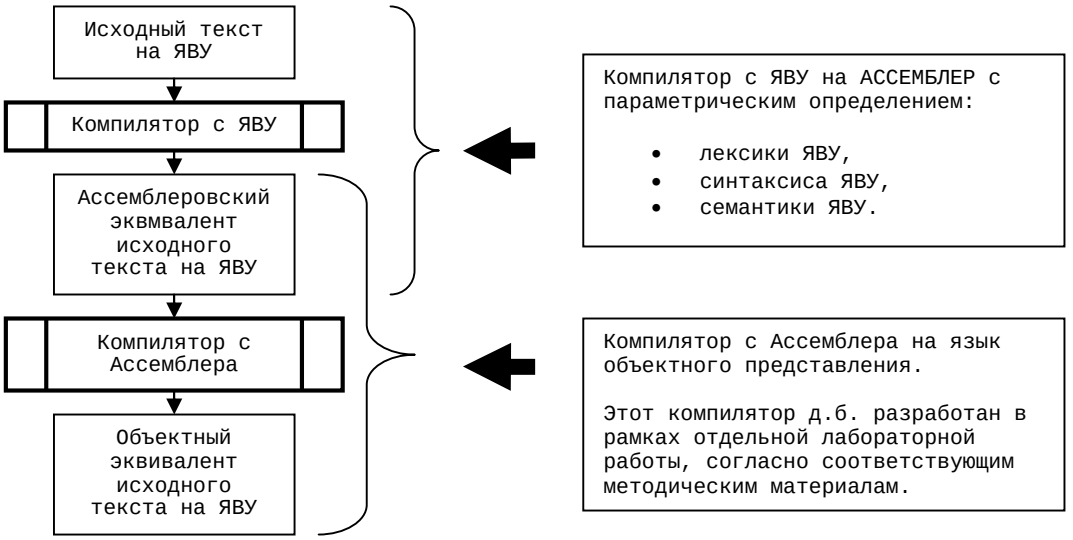


Рис.3. Схема получения объектного представления исходной программы

Из рисунка очевиден состав Базы Данных (БД) компилятора с ЯВУ на язык объектного представления, а именно:

- исходный текст на подмножестве ПЛ/1,
- ассемблеровский эквивалент исходного текста на ЯВУ,
- объектный эквивалент исходного текста на ЯВУ,
- параметрическое определение: лексики, синтаксиса и семантики ЯВУ, т.е. в нашем случае подмножества ПЛ/1.

Исходный текст располагается на 80-ти позиционных картах (записях) в свободном формате, т.е. формате, допускающем размещение элементов исходного текста в любой позиции с разбиением его на термы специальными информативными символами-разделителями или последовательностями, состоящими из нуля или более символов-пробелов.

Способ определения ассемблеровского и объектного эквивалентов исходного текста был рассмотрен при изучении компилятора АССЕМБЛЕРА и повторно здесь обсуждаться не будет.

Внутреннюю структуру определений лексики, синтаксиса и семантики опишем ниже.

Для построения синтаксического графа исходного текста (как далее будет показано) удобно предварительно обработать его, приведя к виду:

терм	разделитель	терм	разделитель	. . .	терм	разделитель

Где терм (лекскма) м.б.:

- зарезервированным словом иероглифом) языка,
- незарезервированным словом, например, идентификатором,
- литералом и т.д.,

а разделитель - элементом множества:

{ ":" , "+" , "-" , ";" , "(" , ")" , "_" }.

Блок компилятора, осуществляющий такое преобразование путем сжатия группы разделителей в одиночный разделитель и, таким образом, уплотняющий исходный текст, является примитивным лексическим анализатором.

На уплотненном исходном тексте далее должен быть построен синтаксический граф, соответствующий грамматике ЯБУ. Это делается следующим блоком компилятора - синтаксическим анализатором.

Наконец, далее следует "вычисление" суперпозиции семантических функций, соответствующей синтаксическому графу, которое завершает процесс компиляции и выполняется последним блоком компилятора – семантическим вычислителем.

Итак, уточненная схема компилятора с ЯВУ на АССЕМБЛЕР выглядит так, как указано ниже:

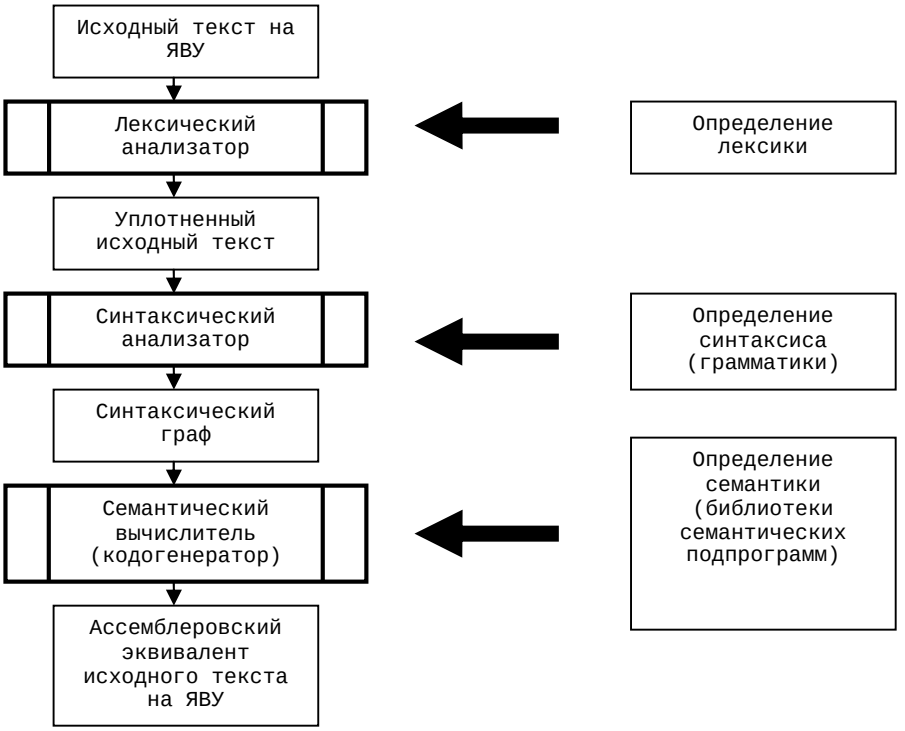


Рис.4. Уточненная функциональная схема компилятора с ЯВУ

2.3. Лексический анализатор

2.3.1. Постановка задачи

В штатных фирменных компиляторах лексический анализатор обычно решает следующие задачи:

- удаление комментариев;
- преобразование исходного текста к виду, удобному для последующей обработки, включающее:

1. Уплотнение исходного текста путем удаления неинформативных фрагментов,
2. Построение таблиц идентификаторов, литералов и т.д.

В нашем случае выберем самый простой вариант, предусматривающий только уплотнение исходного текста путем превращения последовательности смежных пробелов в единственный пробел или удаления второстепенных пробелов, смежных с первостепенными разделителями.

2.3.2. База данных

База данных лексического анализатора включает в себя:

1. Исходный текст ЯВУ на перфокартах в свободном формате.

2. Полный список из первостепенных и второстепенных разделителей SPIS1:

{ ":" , "(" , ")" , ";" , "+" , "-" ,"=" , "_" }.

3. Список первостепенных разделителей, несовместимых с второстепенным пробелом, SPIS2:

{ ":" , "(" , ")" , ";" , "+" , "-" , "=" }.

2.3.3. Алгоритм

Алгоритм лексического анализатора представлен на следующей блок-схеме, приведенной ниже на рис.5.

2.4. Синтаксический анализатор

2.4.1. Постановка задачи

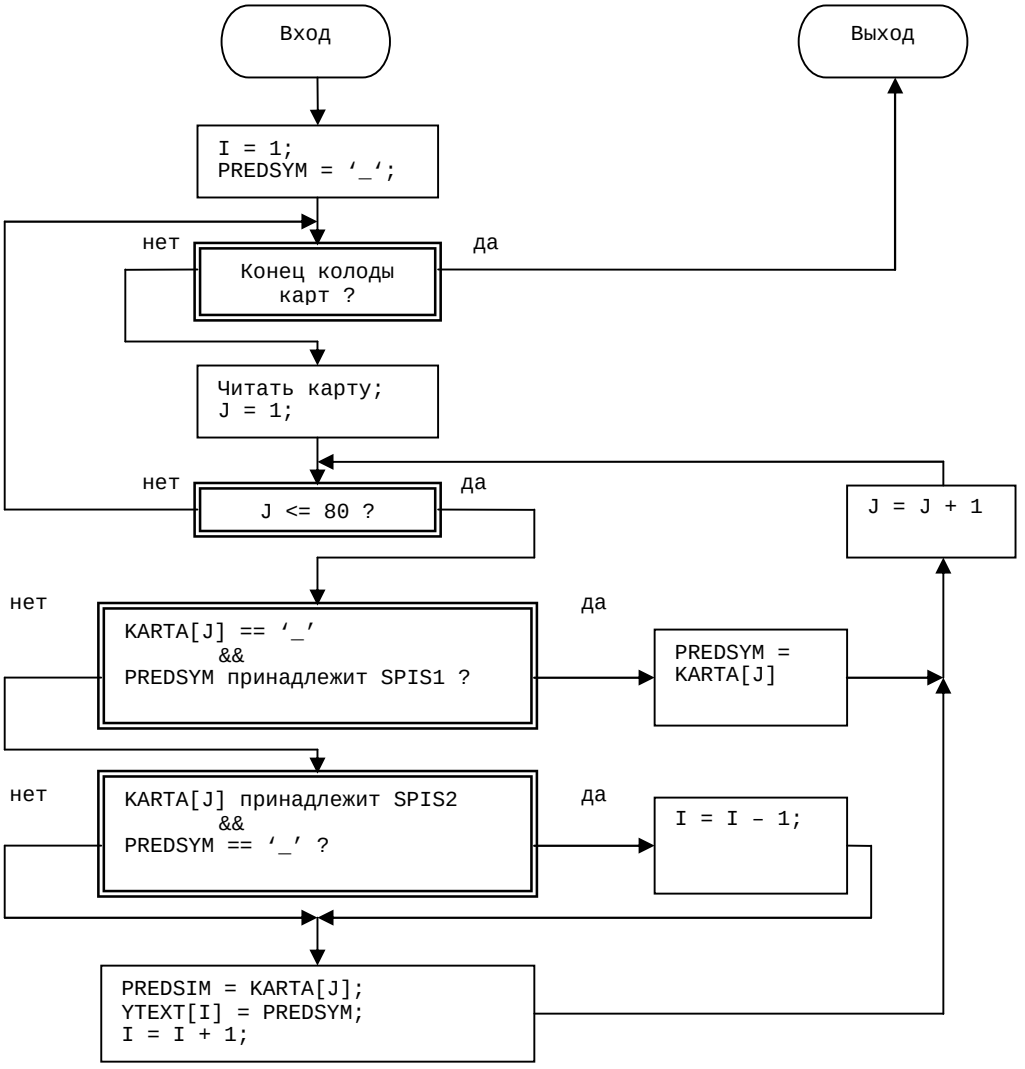
На этапе синтаксического анализа следует решить две задачи:

1) Определить принадлежность последовательности символов, образующей уплотненный исходный текст, выбранному подмножеству языка ПЛ/1.

2) Если принадлежность установлена, то построить синтаксический граф для анализируемой линейной последовательности символов.

Существует большое количество способов построения таких анализаторов, каждый из которых м.б. отнесен к одному из двух типов:

- анализ в направлении "сверху-вниз";
- анализ в направлении "снизу-вверх".



Где:

- YTEXT – результат уплотнения исходного текста,
- I – текущий указатель на первую свободную позицию в YTEXT,
- PREDSYM – локальная переменная,
- KARTA – буфер для чтения очередной карты из колоды,
- J – текущий указатель на символ исходного текста в буфере KARTA,
- SPIS1 и SPIS2 – элементы БД лексического анализатора, описанные выше,
- Символ пробела ' ' для наглядности заменен на символ подчеркивания '_'.

Рис.5. Алгоритм лексического анализатора

Для дальнейшего рассмотрения и использования мы выберем один из возможных алгоритмов синтаксического анализа второго типа, т.е. такого алгоритма, в котором синтаксический анализ и, соответственно, построение синтаксического графа выполняются в направлении "снизу-вверх". При этом обсуждение будем проводить на конкретном демонстрационном примере, который предполагает дальнейшее упрощение выбранного выше подмножества ПЛ/1. Упрощение делаем исключительно для экономии времени и сокращения объема и, соответственно, обозримости начальных, промежуточных и конечных результатов, сопровождающих наше рассмотрение. Все дальнейшее изложение и, соответственно, результаты м.б. распространены и на неупрощенный вариант подмножества ПЛ/1.

Упрощение сводится, к сокращению количества грамматических правил до следующего списка:

1. <PRO> ::= <OPR><OEN>

2. <OPR> ::= <IPR>:PROC_OPTIONS(MAIM);

3. <IPR> ::= <IDE>

4. <IDE> ::= <BUK> | <IDE><BUK> | <IDE><CIF>

5. <BUK> ::= E | X

6. <CIF> ::= 1

7. <OEN> ::= END_<IPR>;

Легко видеть, что эта грамматика описывает "пустую" программу, состоящую из двух операторов:

- пролога программы,
- эпилога программы.

Программа, соответствующая этой грамматике, может выглядеть, например, следующим образом:

```
EX1:PROC_OPTIONS(MAIN);END_EX1;
```

Т.к. грамматика языка демонстрационного примера, приведенная выше и заданная в нотациях метаязыка Бэкуса, непосредственно алгоритмом синтаксического анализа использована быть не может, то ее следует привести к виду, адекватному для машинного использования. Для этого сделаем несколько последовательных преобразований формы представления грамматики, имея конечную цель – получение грамматики в виде некоторого списка, который при программировании на языке “C” может быть представлен как массив структур.

Сначала перепишем каждое правило грамматики, заменяя формат метаязыка Бэкуса на формат продукций (или на форму распознавания).

Для этого в каждой альтернативе каждого правила в формате метаязыка Бэкуса поменяем местами правую и левую части, изменив при этом символ "::=" на "——>". Получим следующий набор продукций:

1. <OPR><OEN> ——> <PRO>

2. <IPR>:PROC_OPTIONS(MAIN); ——> <OPR>

3. <IDE> ——> <IPR>

4. <BUK> ——> <IDE>

5. <IDE><BUK> ——> <IDE>

6. <IDE><CIF> ——> <IDE>

7. E ——> <BUK>

8. X ——> <BUK>

9. 1 ——> <CIF>

10. END_<IPR>; ——> <OEN>

Теперь, просматривая каждую из продукций слева-направо, сгруппируем продукции, имеющие общие части в "кусты", в которых роль "ствола" играют общие части продукций, а роль "ветвей" – различающиеся части продукций. Получим следующий результат:

1. <OPR><OEN> ——> <PRO>

2. <IPR>:PROC_OPTIONS(MAIN); ——> <OPR>

3. <BUK> ——> <IDE>

4. <IDE> ——> <IPR>
```
              └——> <BUK> ——> <IDE>
                        └——> <CIF> ——> <IDE>
```

5. E ——> <BUK>
```
           └——> ND_<IPR>; ——> <OEN>
```

6. X ——> <BUK>

7. 1 ——> <CIF>

Попробуем представить полученную нами систему из 7-ми "кустов", в виде графа, в котором:

- узлы соответствуют терминальным и нетерминальным символам, образующим продукции,
- дуги связывают как смежные символы продукций, так и символы, непосредственно связанные знаками "──>" или "└──>",
- каждый из узлов синтезируемого графа будет либо истоком, либо стоком, либо транзитным,
- каждый из узлов может иметь до трех дуг к смежным узлам (ссылки на левый, правый и альтернативный узел-смежник).

Если же дополнительно мы:

- пронумеруем каждый узел, указывая в верхней части изображения узла символ продукции, которому данный узел соответствует, а в нижней части - номер узла,
- будем далее воспринимать номер каждого узла как координату, однозначно определяющую положение этого узла в графе,
- вынесем координаты анкерных узлов-истоков в отдельную Таблицу Входов (ТВ),
- введем специальный узел-сток, внутри которого будем размещать метасимвол "*" непринадлежащий ни терминальному, ни нетерминальному алфавитам языка. Этот узел в графе будет играть роль маркера-ограничителя, указывающего конец продукции,

то мы получим результат в виде графа, приведенного ниже на рис.6 и рис.7.

Все структурированные в виде графа данные, приведенные на этих рисунках, удобно включить в БД синтаксического анализатора в виде двух таблиц:

- таблицы входов;
- таблицы продукций (синтаксических правил в форме распознавания), сгруппированных в “кусты”. В этой таблице каждая строка м.б. представлена как объект типа struct языка “С”, а вся таблица – как одномерный массив объектов типа struct. Для удобства модификации таблицы в состав объекта struct, соответствующего отдельной строке, будут введены до трех указателей по трем возможным направлениям: “последователь”, “предшесвенник”, “альтернатива”.

Таблица входов (идентификатор ВХ) имеет три графы, в которые для каждого терминального и нетерминального символа заносятся:

- обозначение символа (графа "символ" с идентификатором ВХ1),
- тип символа (графа "тип" с идентификатором ВХ2), имеющий два альтернативных значения: Т - терминальный и N – нетерминальный,
- структурный адрес (координата в графе) начала "куста" продукций, "ствол" которого образуется с данного символа (графа "вход" с идентификатором ВХ3).

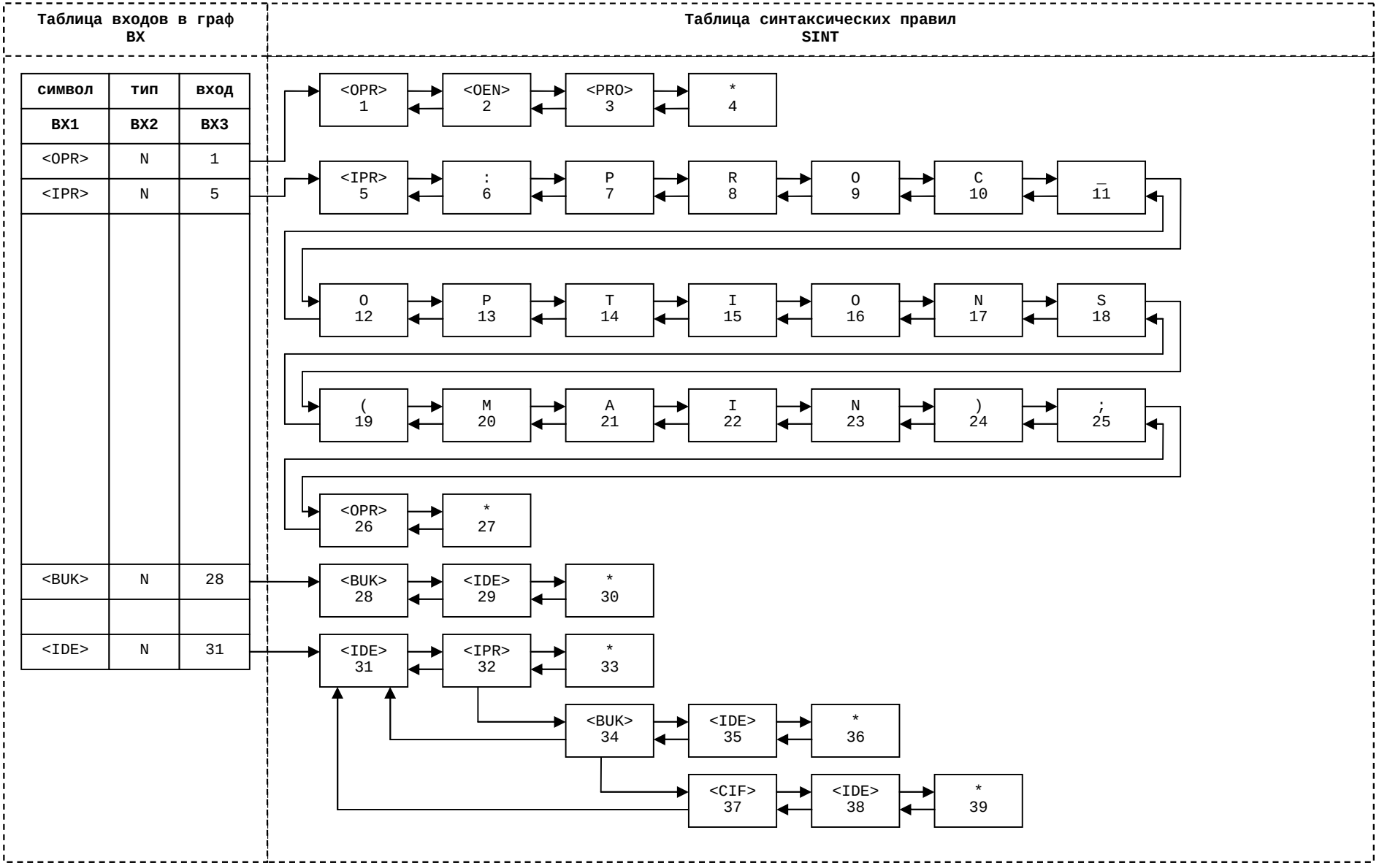


Рис.6. Преобразованные грамматические правила в формате графа (часть 1)

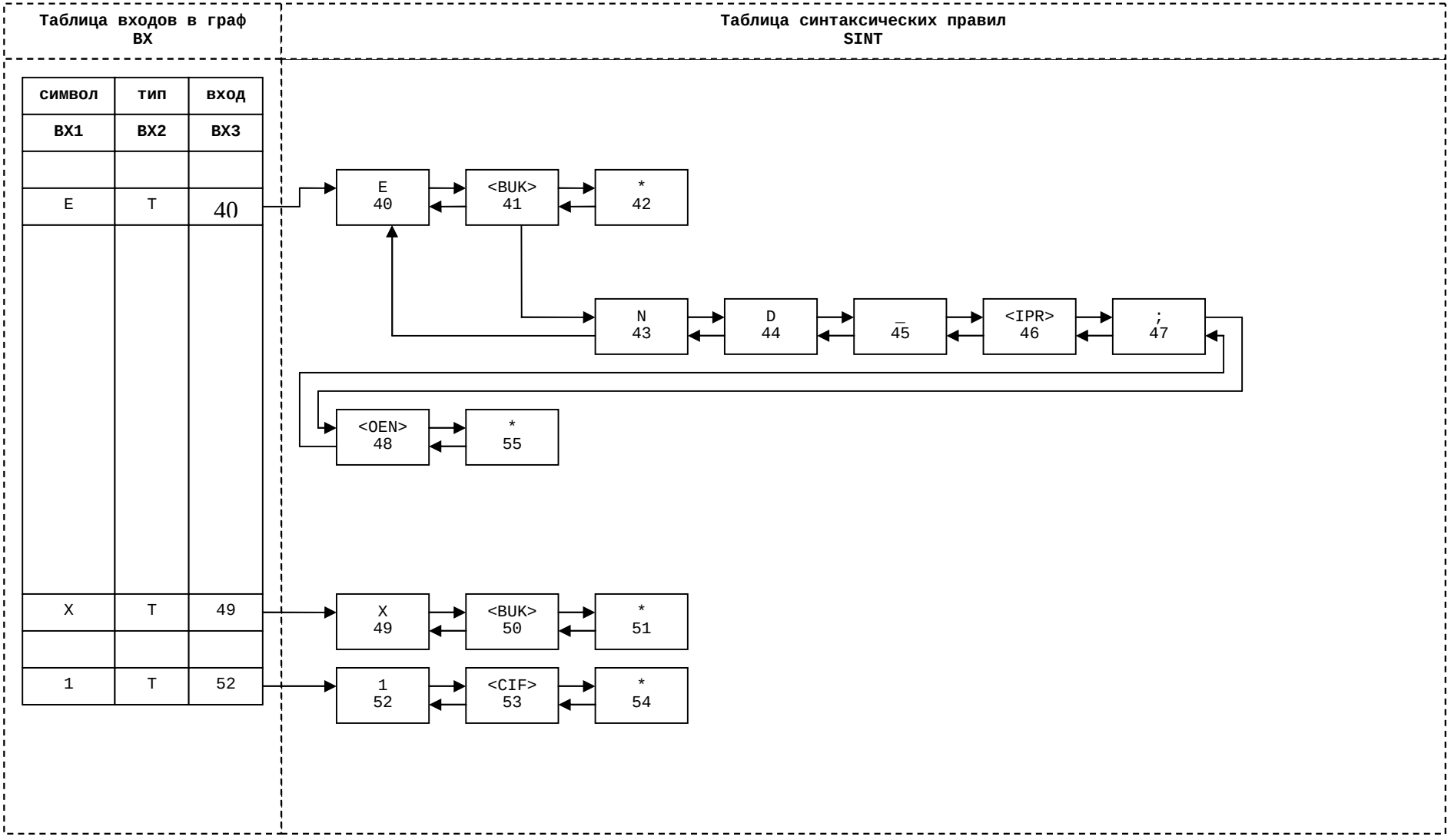


Рис.7. Преобразованные грамматические правила в формате графа (часть 2)

Таблица продукций (идентификатор SINT), которая имеет четыре графы, содержащие, соответственно:

- обозначение символа (столбец "символ" с идентификатором GRAF), совпадающее со значением символа, указанным в верхней части изображения узла графа,
- структурный адрес правого смежного узла графа (столбец "последователь" с идентификатором POSL),
- структурный адрес левого смежного узла графа (столбец "предшественник" с идентификатором PRED),
- структурный адрес альтернативного узла графа (столбец "альтернатива" с идентификатором ALT).

Итак, таблица SINT выглядит следующим образом:

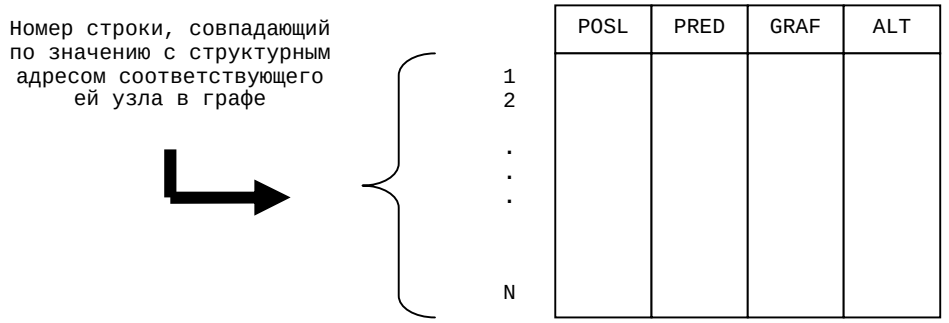


Рис.8. Формат таблицы синтаксических правил - SINT

Теперь приведем несколько предварительных эскизных соображений, положенных в основу рассматриваемого алгоритма синтаксического анализа и, имеющих своей целью перед рассмотрением собственно алгоритма синтаксического анализатора установить дополнительные элементы состава его Базы Данных (БД).

Если представить исходный текст, подвергнувшийся уплотнению лексическим анализатором, рассмотренным выше, в виде линейной последовательности символов, то в каждый конкретный момент времени синтаксического анализа эта последовательность будет делиться на две части: распознанную и нераспознанную.

В начальный момент распознанной части нет или, иначе, она есть, но имеет нулевую длину. По отношению к нераспознанной части исходного текста рекурсивно ставится цель: "Распознать нетерминальный символ XXX", где в качестве символа XXX в начальный момент времени задается аксиома грамматики языка (в нашем случае <PRO>).

При попытке достижения поставленной цели может возникнуть необходимость в постановке промежуточной цели. Такое рекурсивное тиражирование целей в общем случае приводит к возможности построения вектора целей, каждый следующий компонент которого является производным от предыдущего. Для хранения такого вектора будем использовать специальный стек следующей структуры (стек целей с идентификатором CEL).

	Нетерминальный символ-цель	Координаты левой границы цели в исходном тексте	Координаты цели в таблице SINT
	CEL1	CEL2	CEL3
1 . . . N			

Рис.9. Формат таблицы стека поставленных целей - CEL

В вершине стека целей (т.е. в строке таблицы CEL с максимальным номером) хранится самая последняя в хронологическом порядке установленная цель.

Распознанные (т.е. достигнутые) цели должны быть запомнены в другом стеке (стеке достижений с идентификатором DOST), который имеет следующий вид:

	Нетерминальный символ-достижение	Координаты начала цели в исходном тексте	Координаты цели в таблице SINT	Координаты конца цели в исходном тексте	Координаты производной цели в таблице SINT
	DST1	DST2	DST3	DST4	DST5
1 . . .					

Рис.10. Формат таблицы достигнутых целей - DST

В процессе синтаксического анализа рекурсивно придется отвечать на следующий вопрос: “Можно ли **принципиально** на нераспознанной части исходного текста, начинающейся с ее первого

символа, достичь актуальную цель, занесенную в верхушку стека целей?".

Если учесть то, что актуальная цель всегда задается некоторым нетерминальным символом, определенным в грамматике языка, а грамматика языка представлена графом, хранящемся в таблице SINT, то сформулированный выше вопрос в терминах теории графов м.б. переформулирован так: “Можно ли принципиально, начиная с терминального символа, открывающего нераспознанную часть исходного текста и представленного как узел-исток грамматического графа в таблице SINT, достичь нетерминальный символ, являющийся актуальной целью и представленный как узел-сток грамматического графа в таблице SINT?”. Ответ на вопрос в такой формулировке м.б. получен после решения одной из хорошо известной в теории графов транспортно-навигационной задачи о существовании принципиальной возможности перехода из одной точки лабиринта в другую. Одним из возможных методов решения этой задачи является метод, основанный на выполнении трех действий:

- представлении графа в виде матрицы смежности узлов графа,
- построении матрицы связности узлов графа из матрицы смежности помощью алгоритма Варшалла,
- использовании построенной матрицы связности для получения ответа на вопрос, поставленный выше.

Первые два действия следует выполнить до начала синтаксического анализа, а третье действие делается в процессе синтаксического анализа в моменты возникновения вопроса.

Далее наравне с термином матрица связности будем применять синоним – матрица преемников.

Демонстрируя далее способ построения матриц смежности и связности, будем предполагать, что:

- нетерминальный алфавит языка содержит **n** элементов, а полный алфавит, объединяющий терминальный и нетерминальный алфавиты, - **m** элементов,
- матрица связности состоит из **m** строк и **n** столбцов,
- единица на перекрестии строки **X** и столбца **Y** матрицы смежности ставится в случае, если в грамматике существует продукция, начинающаяся с символа **X** из полного алфавита и кончающаяся символом **Y** из нетерминального алфавита. При этом символ **X** определяет положение строки, а **Y** - столбца перекрестия.

Другими словами наличие единицы на некотором перекрестии строки и столбца матрицы смежности, определяемыми соответственно символами **X** и **Y**, означает то, что:

- во-первых, **принципиально** возможно распознать нетерминал **Y** на строке исходного текста, в начале которой распознан символ **X**,
- во-вторых, такое распознавание можно выполнить использованием **единственной** продукции.

Для грамматики рассматриваемого нами демонстрационного примера матрица смежности примет следующий вид.

Нетермин.алф. / Полный алф.	<BUK>	<CIF>	<IDE>	<IPR>	<OEN>	<OPR>	<PRO>
<BUK>			1				
<CIF>							
<IDE>			1	1			
<IPR>						1	
<OEN>							
<OPR>							1
<PRO>							
E	1						
X	1						
1		1					

m

n

Рис.11. Матрица смежности

Из матрицы смежности хотелось бы получить производную от нее матрицу связности (матрицу преемников), в которой наличие единицы на некотором перекрестии строки и столбца, определяемыми соответственно символами X и Y, означает то, что:

- во-первых, **принципиально** возможно распознать нетерминал Y на строке исходного текста, в начале которого распознан символ X,
- во-вторых, такое распознавание можно выполнить с помощью последовательного применения **конечного числа** продукций.

Другими словами единица на перекрестии строки и столбца матрицы связности, определяемыми соответственно символами X и Y, указывает на **принципиальную** возможность с помощью конечного числа продукций непосредственно или транзитивно распознать на исходном тексте, начинающемся с символа X, нетерминал Y.

Построение матрицы связности из матрицы смежности м.б. осуществлено с помощью алгоритма Варшалла, применяемого в транспортных задачах для определения наличия пути между узлами транспортной сети. Этот алгоритм на языке “C” м.б. представлен так:

```
for ( j=1; j < n+1; j++ ) {
      for ( i=1; i < m+1; i++ ) {
            if ( PR [i,j] & (i != j) ) then {
                  for ( k=1; k < n+1; k++ ) {
                        PR [i,k] |= PR [j,k];
                  }
            }
      }
}
```

Где с идентификатором PR сопоставляется некоторая матрица, которая в начале работы алгоритма равна матрице смежности, а к концу работы алгоритма превращается в конечный результат - матрицу связности.

Для рассматриваемого нами примера матрица связности м.б. получена из матрицы смежности с помощью алгоритма Варшалла в следующем виде:

Нетермин.алф. / Полный алф.	<BUK>	<CIF>	<IDE>	<IPR>	<OEN>	<OPR>	<PRO>
<BUK>			**1**	1		1	1
<CIF>							
<IDE>			**1**	**1**		1	1
<IPR>						**1**	1
<OEN>							
<OPR>							**1**
<PRO>							
E	**1**		1	1	1	1	1
X	**1**		1	1		1	1
1		**1**					

m

n

Рис.12. Матрица связности - PR

В построенной матрице связности жирным шрифтом с подчеркиванием выделены единицы, заимствованные из матрицы связности, а остальные получены расчетным путем по алгоритму Варшалла.

2.4.2. База данных

База данных синтаксического анализатора состоит из следующих элементов рассмотренных выше:

- уплотненный исходный текст на подмножестве ПЛ/1 (идентификатор STR),
- таблица входов (идентификатор BX),
- таблица продукций (идентификатор SINT),
- стек целей (идентификатор CEL),
- стек достижений (идентификатор DOST),
- матрица связности (идентификатор PR).

2.4.3. Алгоритм

Алгоритм синтаксического анализатора, состоит из блоков:

- начальных установок (см. рис.14),
- проецирования нетерминала по выбранной гипотезе (см.рис.14),
- обработки терминала (см. рис.14),
- обработки нетерминала (см. рис.15 и рис.16),
- перебора гипотез (см. рис.17).

Далее на рисунках каждый из блоков заключен в пунктирную рамку, с надписью названия блока в левом верхнем углу этой рамки. Блоки соединены друг с другом через поименованные алгоритмические разъемы, которые изображаются следующим образом:

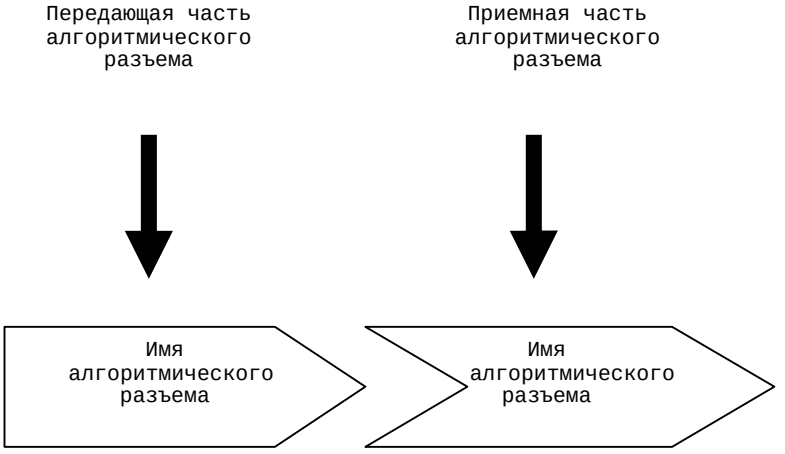


Рис.13. Вид представления межблочных алгоритмических разъемов

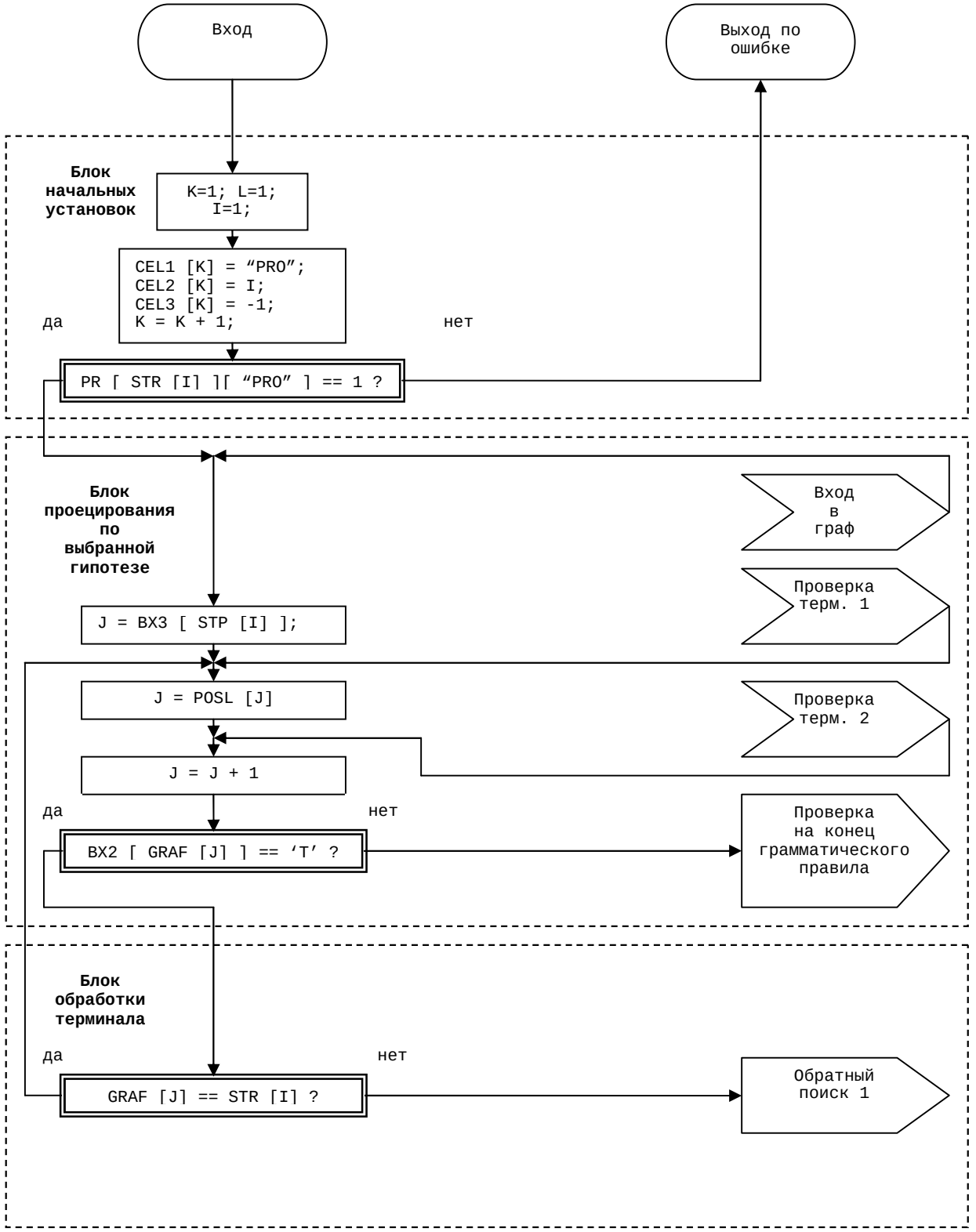


Рис.14. Алгоритм синтаксического анализатора (часть 1)

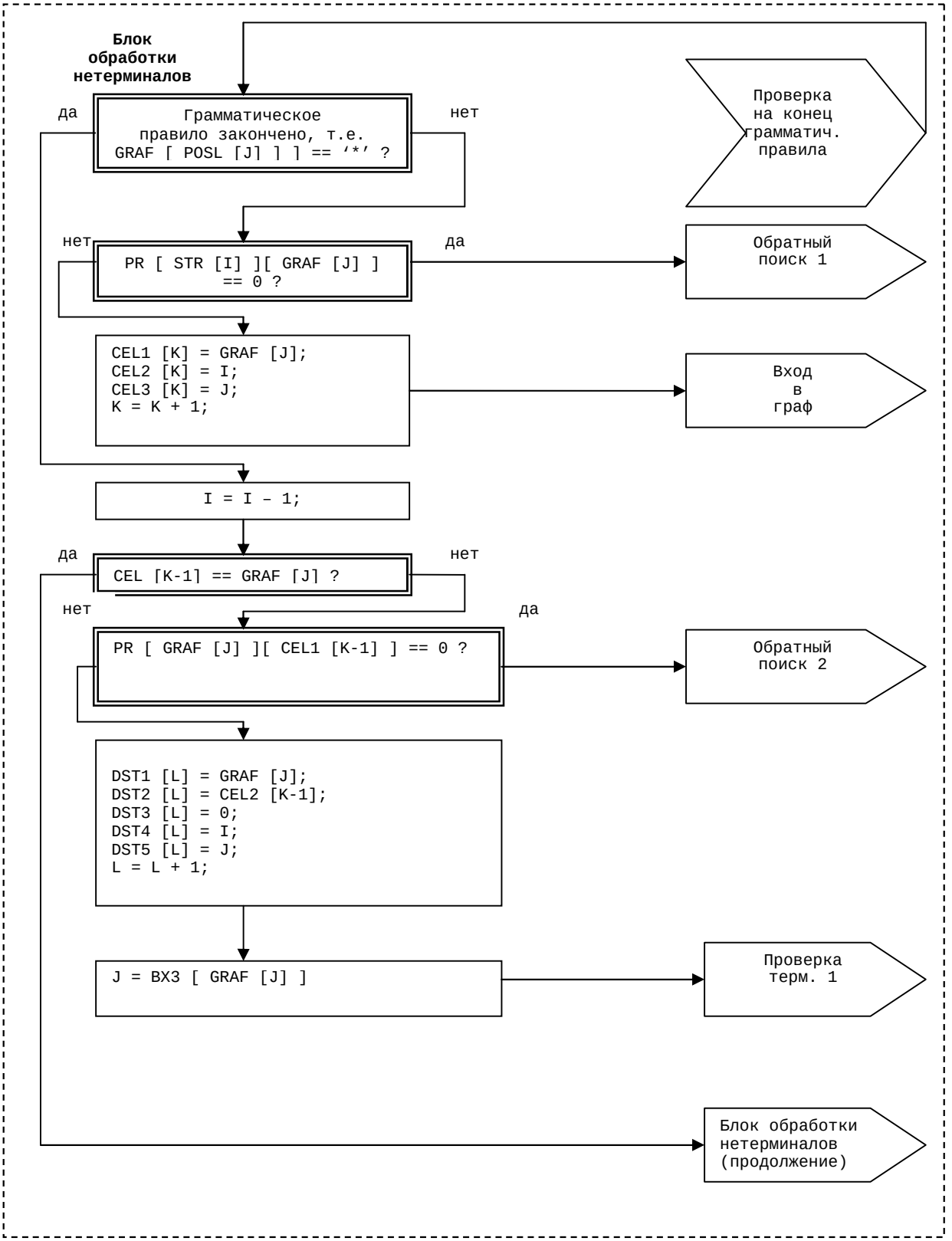


Рис.15. Алгоритм синтаксического анализатора (часть 2)

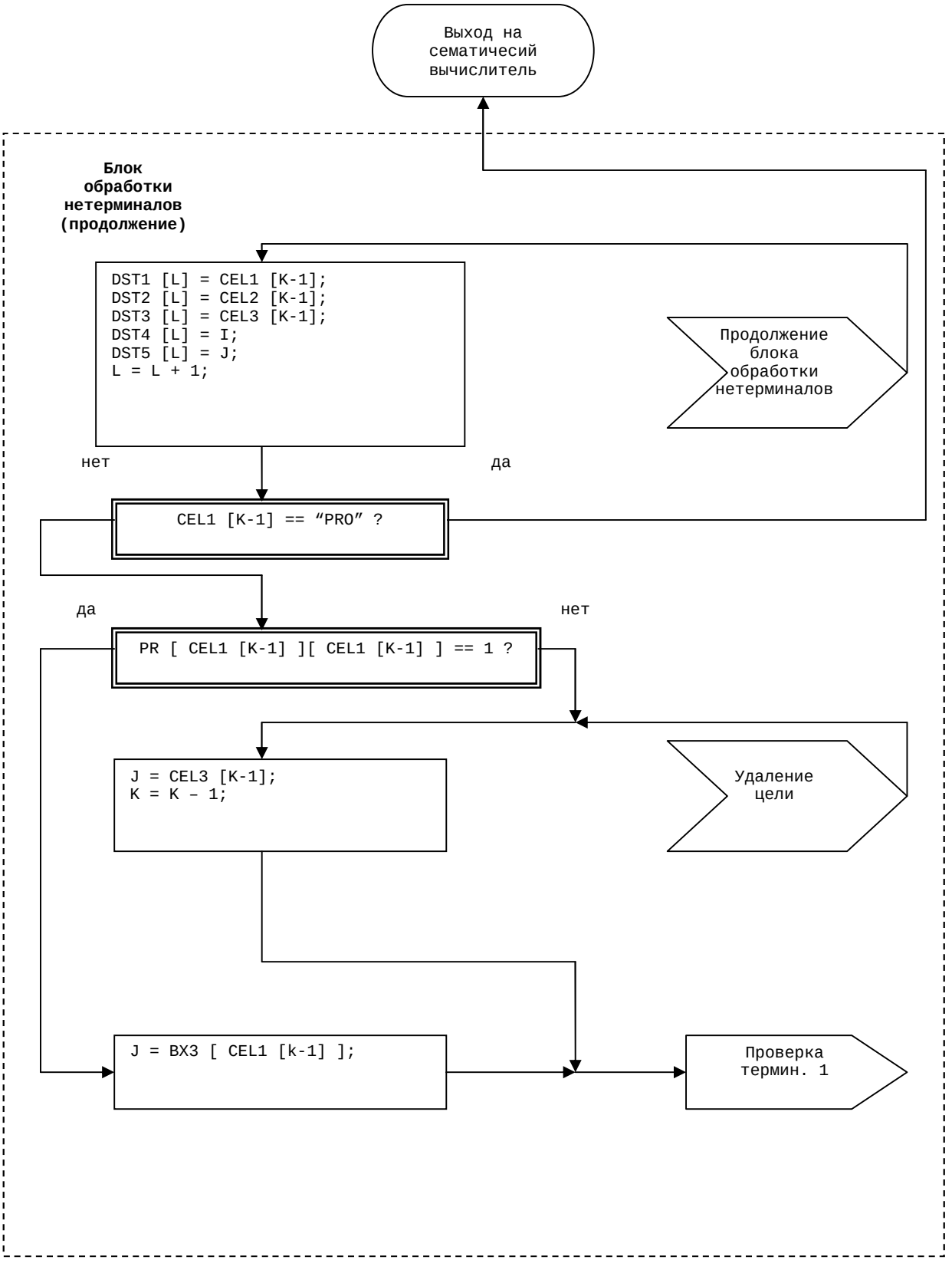


Рис.16. Алгоритм синтаксического анализатора (часть 3)

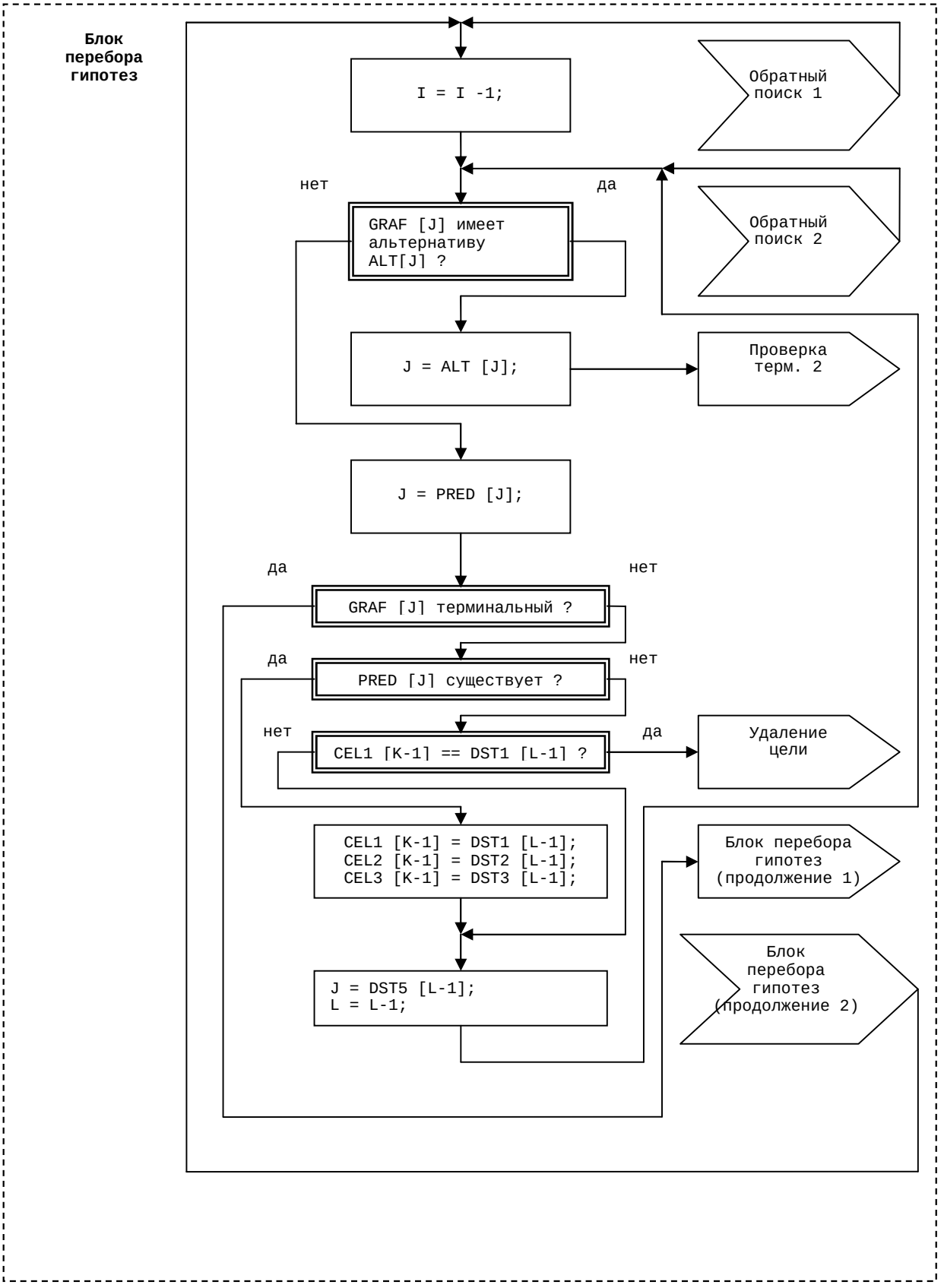


Рис.17. Алгоритм синтаксического анализатора (часть 4)

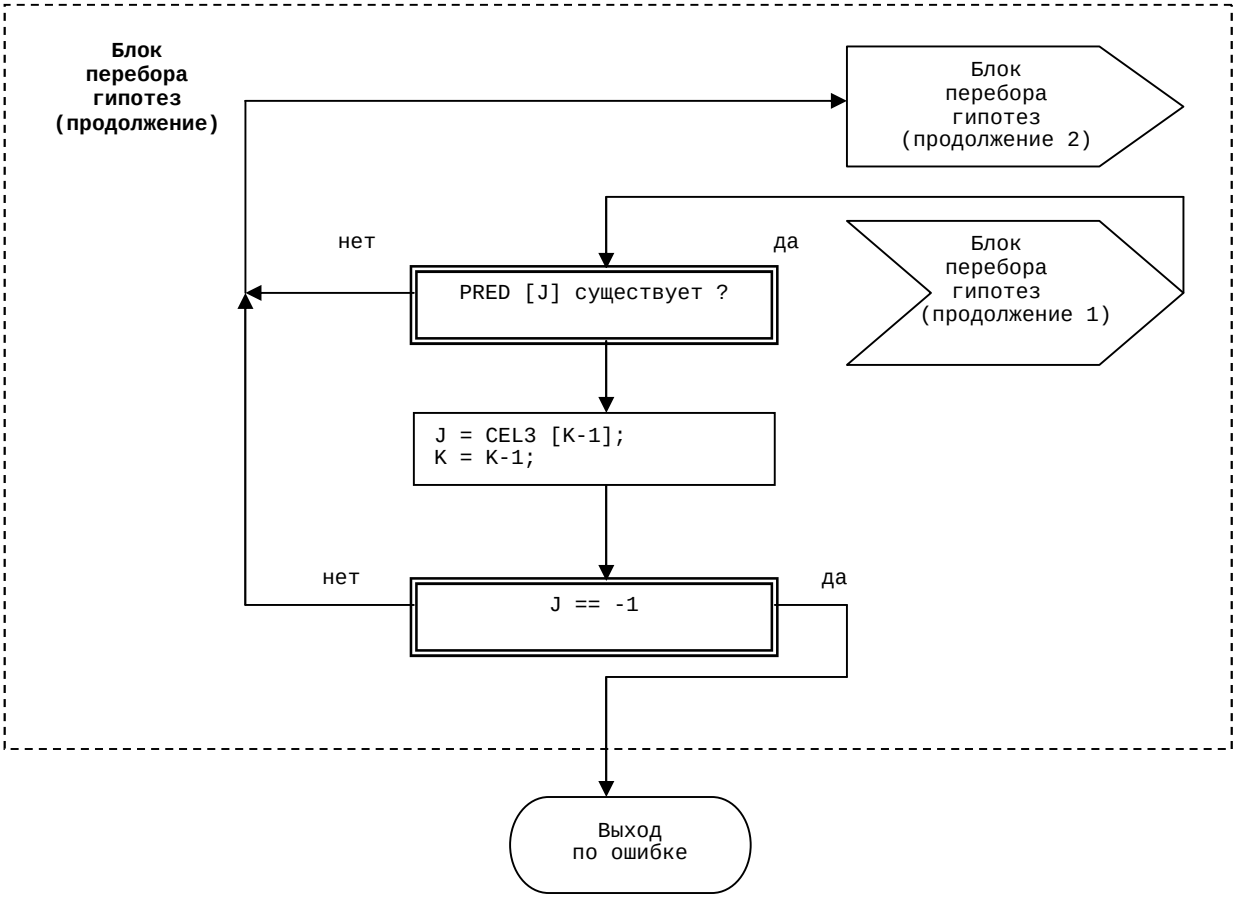


Рис.18 Алгоритм синтаксического анализатора (часть 5)

Р е к о м е н д а ц и я:

Для детального изучения алгоритма целесообразно выполнить вручную в соответствии с ним несколько циклов действий, завершающихся занесением или стиранием данных в стек достижений. Такую проверку алгоритма действием следует выполнять с помощью базы данных, заполненной данными, соответствующими рассматриваемому нами выше примеру, который в посимвольно пронумерованном виде выглядит следующим образом:

E	**X**	**1**	**:**	**P**	**R**	**O**	**C**	**_**	**O**	**P**	**T**	**I**	**O**	**N**	**S**	**(**	**M**	**A**	**I**	**N**	**)**	**;**
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

E	**N**	**D**	**_**	**E**	**X**	**1**	**;**
24	25	26	27	28	29	30	31

Если такая ручная проверка выполнена правильно, то в результате многократного внесения и удаления данных в стеке достижений DST останется следующая информация (см. рис.19).

	Нетерминальный символ-достижение	Координаты начала цели в исходном тексте	Координаты цели в таблице SINT	Координаты конца цели в исходном тексте	Координаты производной цели в таблице SINT
	DST1	DST2	DST3	DST4	DST5
1	BUK	1	0	1	41
2	IDE	1	0	1	29
3	IPR	1	0	1	32
3	BUK	2	34	2	50
4	IDE	1	0	2	35
5	IPR	1	0	2	32
5	CIF	3	37	3	53
6	IDE	1	0	3	38
7	IPR	1	0	3	32
8	OPR	1	0	23	26
9	BUK	28	0	28	41
10	IDE	28	0	28	29
11	IPR	28	46	28	32
11	BUK	29	34	29	50
12	IDE	28	0	29	35
13	IPR	28	46	29	32
13	CIF	30	37	30	53
14	IDE	28	0	30	38
15	IPR	28	46	30	32
16	OEN	24	2	31	48
17	PRO	1	-1	31	31

Рис.19. Результат заполнения стека DST

З а м е ч а н и е:

На рис.19 строки стека достижений с повторяющимися номерами и обведенные пунктирными линиями следует понимать как первично записанные в стек достижений в результате подтверждения гипотезы о

законности нетерминала, приведенного в пунктирной рамке, но далее признанные ошибочными из-за ложности гипотезы, вычеркнутые из стека достижений и замененные на строки с такими же номерами, но обведенные сплошными линиями.

Покажем, что в стеке достижений на рис. 19 записан синтаксический граф исходного текста. Для этого перепишем каждую строку стека в виде отдельного узла и объединим эти узлы в единый синтаксический граф.

Начнем с первой строки стека достижений, обратив внимание на содержимое полей: DST1, DST2, DST4. При этом получим результат, приведенный на рис. 20.

Вторая строка стека достижений вместе с первой строкой м.б. расписаны в виде, представленном на рис. 21.

Последовательно обрабатывая так все строки стека достижений с первой по семнадцатую, получим весь синтаксический граф, который приведен на рис. 22.

2.5. Семантический вычислитель

2.5.1. Постановка задачи

Выше в пп. 2.3 мы установили, что синтаксический граф м.б. интерпретирован как некая программа, состоящая из отдельных действий, соответствующих узлам синтаксического графа и выполняемых в последовательности, устанавливаемой принятым порядком обхода узлов графа.

В случае применения синтаксического анализатора, описанного в пп. 2.3, синтаксический граф строится в стеке достижений, где на каждый узел графа отводится одна строка стека с фиксированным расположением полей, среди которых нас будут интересовать далее (см. рис. 19):

- первое, содержащее название узла синтаксического графа,
- второе, содержащее левую границу фрагмента исходного текста, соответствующего узлу синтаксического графа,
- четвертое, содержащее правую границу фрагмента исходного текста, соответствующего узлу синтаксического графа.

Легко видеть, что принятый ранее порядок обхода узлов графа при семантическом вычислении (описан в пп. 2.2) соответствует последовательному перебору строк стека достижений в направлении от дна к вершине. Это было продемонстрировано при переписи содержимого стека достижений в виде графа в конце пп. 2.4.3.

Если при этом воспринимать каждую строку стека достижений с тремя упомянутыми полями как команду некоторой абстрактной ЭВМ, где:

- название узла (первое поле) задает код команды абстрактной ЭВМ,
- левая граница (второе поле) является первым операндом команды абстрактной ЭВМ,

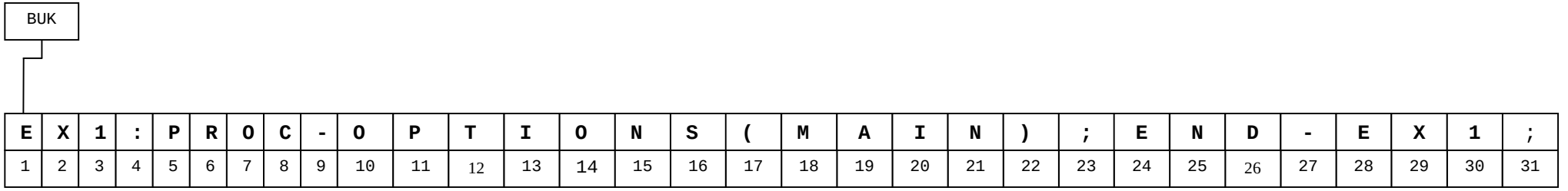


Рис.20. Фрагмент синтаксического графа, согласно первой строке стека достижений

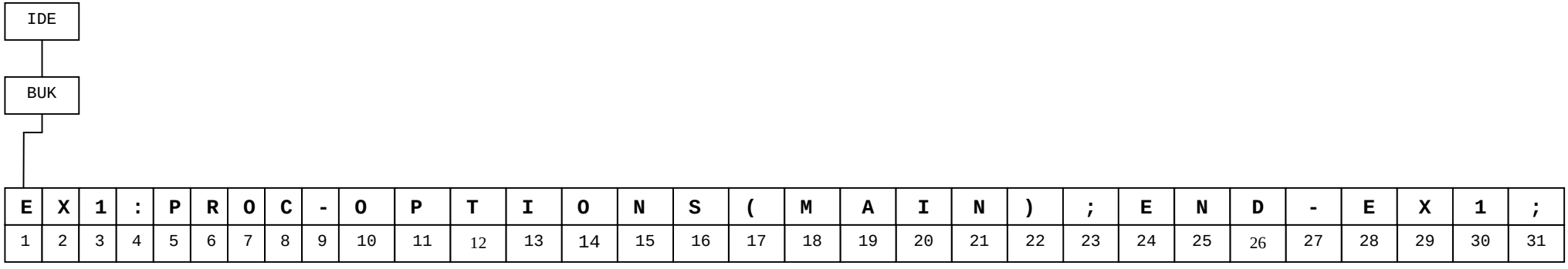


Рис.21. Фрагмент синтаксического графа, согласно первой и второй строкам стека достижений

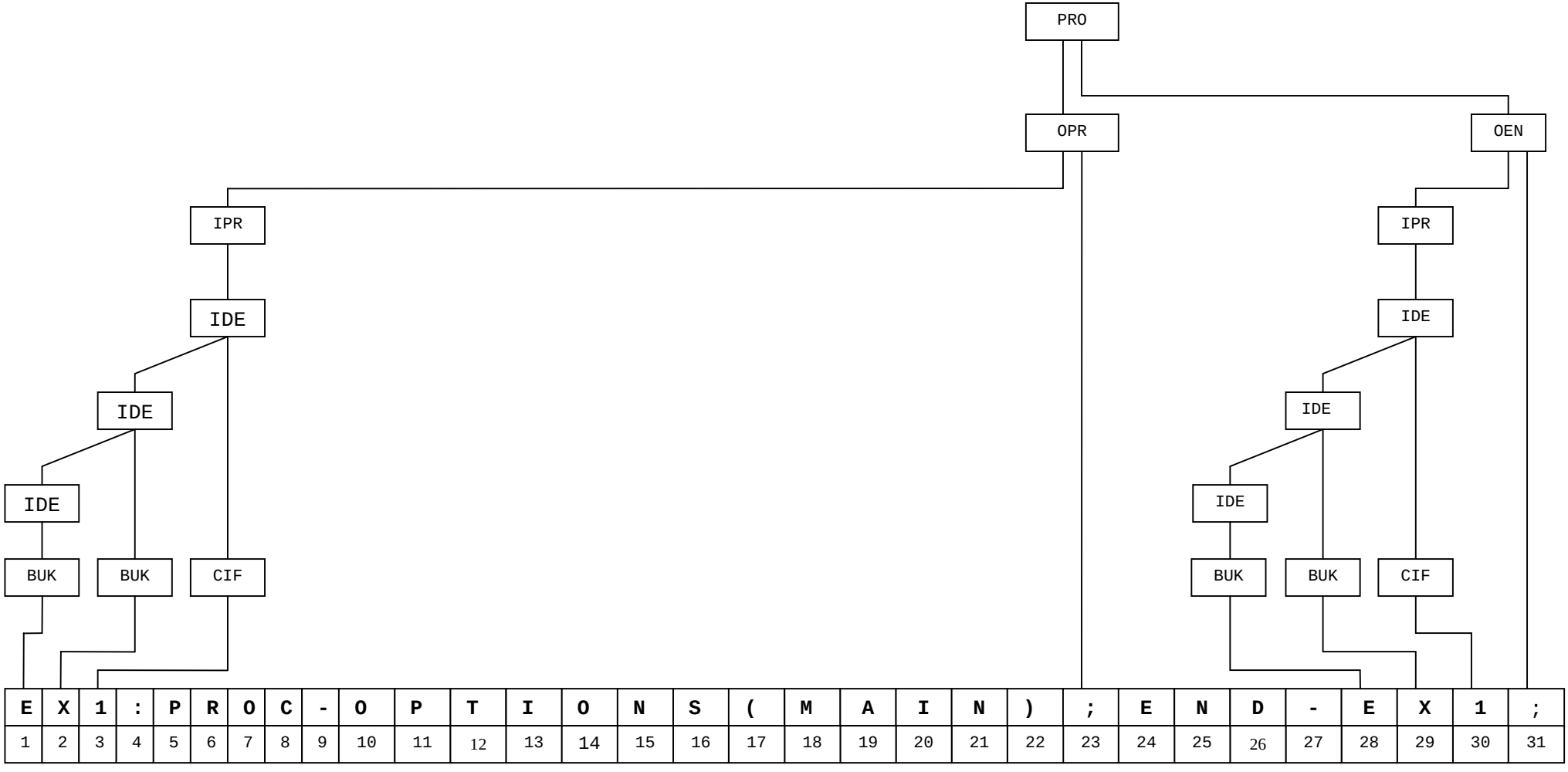


Рис.22. Полный синтаксический граф, согласно содержимому стека достижений

- правая граница (четвертое поле) является вторым операндом команды абстрактной ЭВМ,

то далее построение семантического вычислителя сведется к программной реализации указанной абстрактной ЭВМ.

Заметим то, что из отождествления семантического вычислителя с абстрактной ЭВМ вытекают следующие тесные зависимости свойств абстрактной ЭВМ и грамматики языка:

- формат команд абстрактных ЭВМ, ориентированных на различные грамматики, одинаков (т.е. состоит из трех полей, указанных выше),
- между командой абстрактной ЭВМ и соответствующим нетерминалом грамматики существует взаимно однозначная связь (отсюда следует то, что список команд конечен и содержит столько элементов, сколько нетерминалов содержится в грамматике),
- семантика каждой команды задается программно (с помощью семантической программы) и однозначно определяет семантику соответствующего нетерминала.

2.5.2. База данных

Состав БД абстрактной ЭВМ, которую мы сопоставили семантическому вычислителю, следующий:

- программа для абстрактной ЭВМ, которая образуется в стеке достижений,
- выходной файл - результат семантического вычисления, который в нашей постановке должен содержать эквивалент ПЛ/1-программы на АССЕМБЛЕРЕ IBM 370,
- библиотека семантических определений команд абстрактной ЭВМ.

Семантические определения мы будем представлять в виде подпрограмм, которые в совокупности далее будем называть **библиотекой семантических программ.**

2.5.3. Алгоритм

Для нашего примера (в отличие от макета) достаточно выбрать однопроходный (прошу подумать почему?) алгоритм семантического вычислителя, который м.б. представлен в виде блок-схемы на рис. 23 следующим образом:

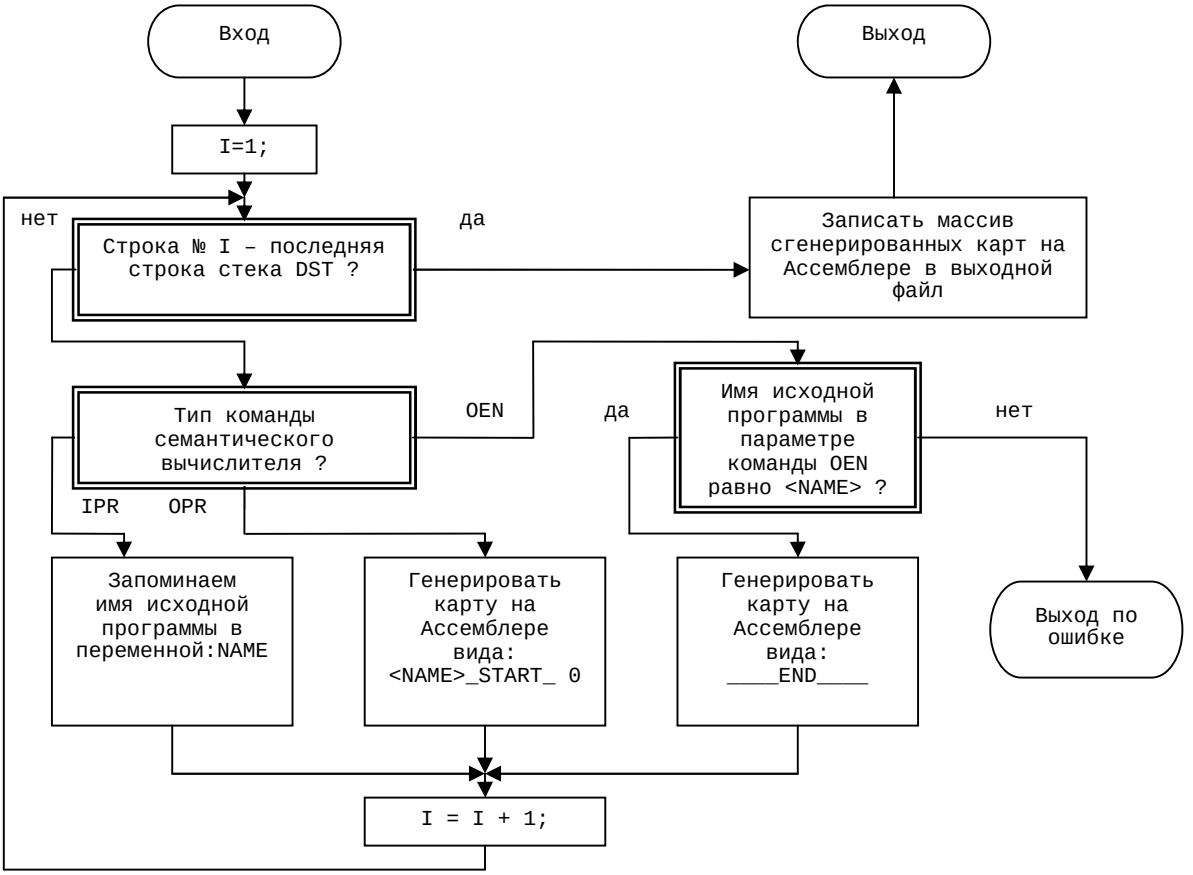


Рис. 23. Алгоритм семантического вычислителя

3. РЕКОМЕНДАЦИИ ПО ПОРЯДКУ ВЫПОЛНЕНИЯ РАБОТЫ

Данная лабораторная работа предполагает использование в качестве начальной основы отлаженной и работающей программы-макета компилятора с языка ПЛ1, с ограниченным набором языковых конструкций, достаточным для написания законченной правильной программы контрольного примера. Программа-макет написана на алгоритмическом языке Си и снабжена подробным контекстным комментарием.

Студенту предлагается расширить первичный набор языковых конструкций, поддерживаемый макетным компилятором с ЯВУ, до уровня, позволяющего оттранслировать пример, содержащийся в его личном задании, и произвести соответствующую доработку макетного компилятора с ЯВУ. При этом необходимо:

- ознакомиться с исходными текстами программы-макета и контрольного примера, на который настроена программа-макет,
- проверить работоспособность программы-макета путем проведения компиляции контрольного примера,
- визуально выделить из исходного текста примера, содержащегося в личном задании, множества языковых конструкций, отсутствующих в контрольном примере,
- спроектировать перечень коррекций элементов БД и подпрограмм программы-макета с целью обеспечения компиляции примера из личного задания,
- ввести коррекции в исходный текст программы-макета, снабдив измененные места исходного текста построчным комментарием, и записать модифицированный вариант в свой личный директорий на сервере,
- выполнить отладку модифицированной программы-макета,
- составить отчет по результатам работы.

4. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ ОТЧЕТА

Отчет должен соответствовать рекомендациям по оформлению студенческих курсовых работ, размещенных на сайте университета, и в общем случае должен содержать следующие разделы:

- описания примера из личного задания (указать формулировку личного задания),
- постановки задачи модификации программы-макета (описание выбранного способа модификации программы-макета),
- содержащие перечень и пояснения модификаций состава и (или) структуры БД компилятора,
- перечень и пояснения модификаций алгоритма программы-макета компилятора (возможные формы описаний в виде: блок-схемы, текста на Си с комментариями, а в очевидных случаях обычным текстом),
- выводы по результатам работы (указать позитивные и негативные моменты, проявившиеся в процессе работы, с общей личной оценкой проделанной работы).

5.ТРЕБОВАНИЯ К КОМПЛЕКТАЦИИ, ОФОРМЛЕНИЮ И ПРЕДЪЯВЛЕНИЮ РЕЗУЛЬТАТОВ КУРСОВОЙ РАБОТЫ

К результатам курсовой работы относятся:

- Отчеты в бумажном исполнении,
- Отчеты в электронном исполнении,
- Исходные тексты компилятора с ЯВУ,
- Исходные тексты компилятора с АССЕМБЛЕРА,
- Исходные тексты загрузчика.

Сводный отчет по курсовой работе “Системы программирования”, выполненной студентом, должен:

- быть укомплектован из отчетов по всем элементам разработанной учебной системы программирования,
- по своей структуре соответствовать рис.24,
- быть подготовлен как в бумажном, так и в электронном вариантах исполнения,
- быть передан преподавателю после завершения курсовой работы в момент защиты результатов курсовой работы.

Работы по отдельным элементам разрабатываемой учебной системы программирования проводятся в течение семестра. Отчеты по каждому элементу:

- составляются по завершению работы над его реализацией,
- предъявляются преподавателю в течении семестра,
- включаются в сводный отчет к концу семестра.

Исходные тексты элементов учебной системы программирования вместе с электронным вариантом сводного отчета оформляются и передаются преподавателю в виде архивного файла, рекомендуемый состав папок и файлов которого приведен на рис.25. В названии архива **Var_XX**, вместо **XX** следует использовать номер варианта своего задания на курсовую работу.

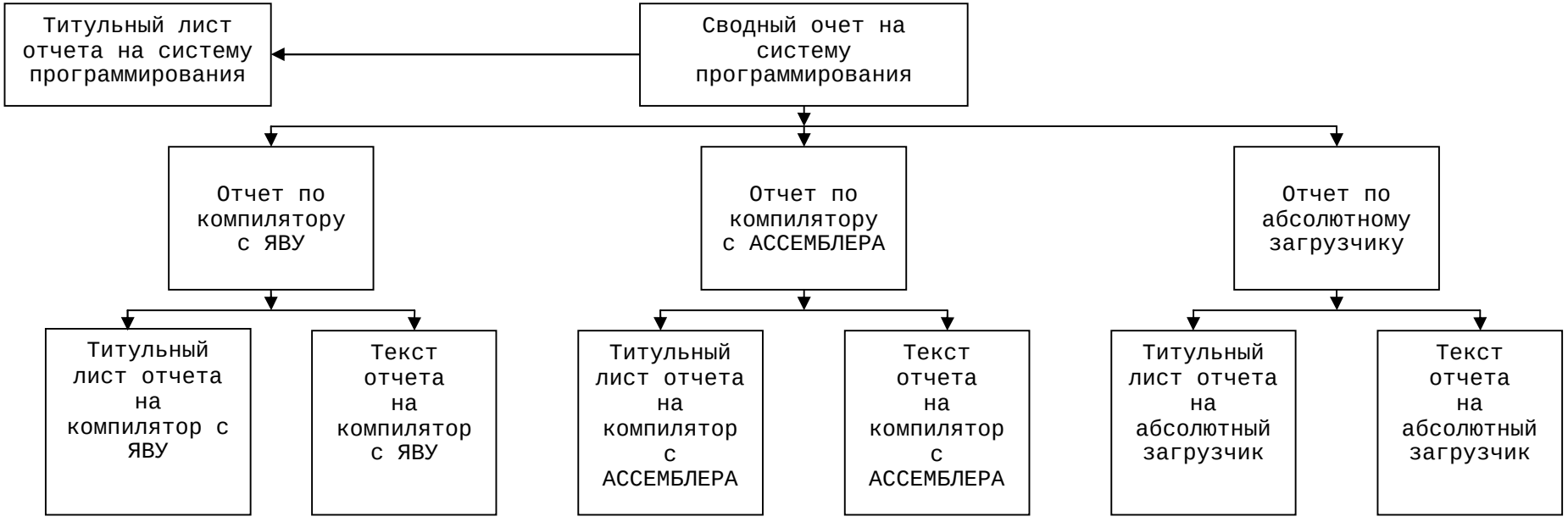


Рис.24. Структура сводного отчета по курсовой работе

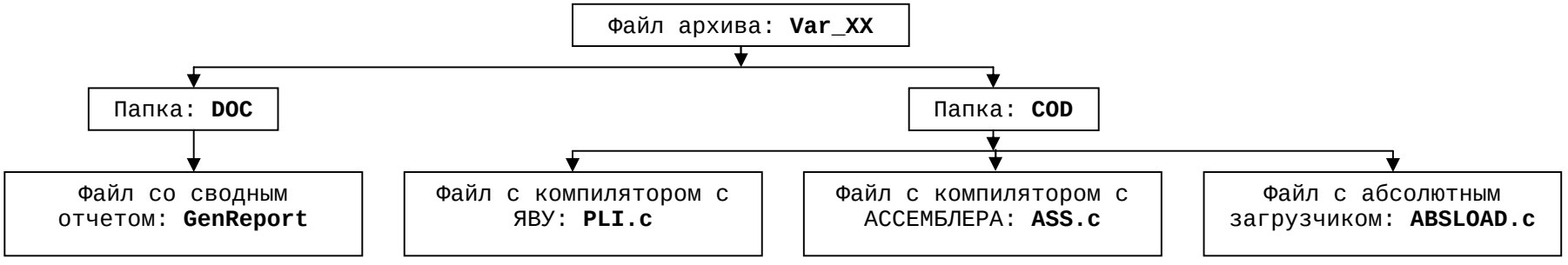


Рис.25. Рекомендуемый состав архива