ТЕОРІЯ КОМПІЛЯТОРІВ

Лекції 1-2. Введення в теорію компіляторів. Логічна структура компілятора

2019

ТЕРМИНОЛОГИЯ

Начнем с того, что дадим классические определения терминам, которые будут использоваться нами далее.

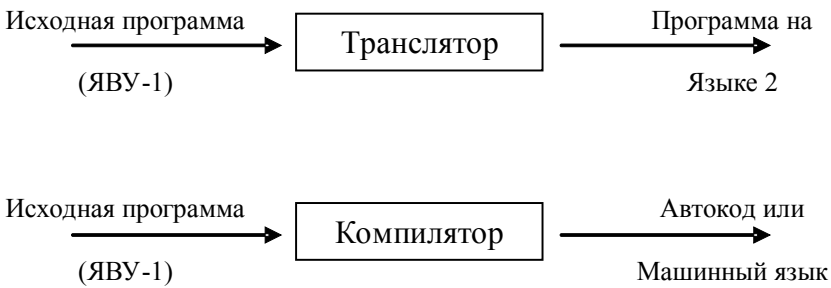
Транслятор – это программа, которая переводит текст исходной программы в эквивалентную объектную программу. Если объектный язык представляет собой автокод или некоторый машинный язык, то транслятор называется *компилятором*.

Автокод очень близок к машинному языку; большинство команд автокода – точное символическое представление команд машины.

Ассемблер – это программа, которая переводит исходную программу, написанную на автокоде или на языке ассемблера (что, суть, одно и то же), в объектный (исполняемый) код.

Интерпретатор принимает исходную программу как входную информацию и выполняет ее. Интерпретатор не порождает объектный код. Обычно интерпретатор сначала анализирует исходную программу (как компилятор) и транслирует ее в некоторое внутреннее представление. Далее интерпретируется (выполняется) это внутреннее представление.

Условно это можно изобразить следующим образом:

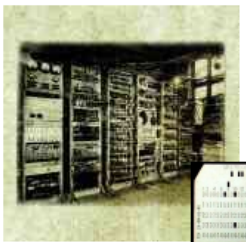


Компиляторы пишутся как на автокоде, так и на языках высокого уровня. Кроме того, существуют и специальные языки конструирования компиляторов – *компиляторы компиляторов*.

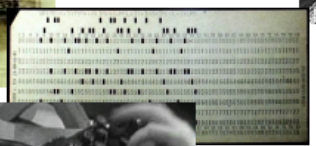
Компилятор компиляторов (КК) – система, позволяющая генерировать компиляторы; на входе системы – множество грамматик, а на выходе, в идеальном случае, – программа. Иногда под КК понимают язык программирования, в котором исходная программа – это описание компилятора некоторого языка, а объектная программа – сам компилятор для этого языка. Исходная программа КК – это просто формализм, служащий для описания компиляторов, содержащий, явно или неявно, описание лексического и синтаксического анализаторов, генератора кодов и других частей создаваемого компилятора. Обычно в КК используется реализация схемы т.н. синтаксически управляемого перевода. Кроме того, некоторые из них представляют собой специальные языки высокого уровня, на которых удобно описывать алгоритмы, используемые при создании компиляторов.

Языковые парадигмы. «Снизу-вверх». Шаг 1.

Последовательная автоматизация программирования

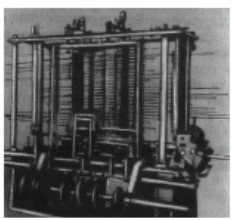


“Марк-1”



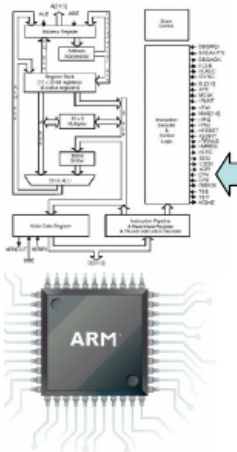
“ЭНИАК”

Аналитическая машина Чарльза Бебеджа



Языковые парадигмы. «Снизу-вверх». Шаг 2.

- Считывание команд, как исполнение программы.
- Ассемблер. Схемотехнический уровень.
- Библиотеки
- «Обертка» машинных инструкций.



Программатор

```
:0A0000002EC0FECFFDCFFCCFFBCFDA
:10000A00FACFF9CFF8CFF7CFF6CFF5CFF4CFF3CFBA
:10001A00F2CFF1CFF0CFEFCFEECFEDCFECCFEBCFEA
:10002A00EACFE9CFE8CFE7CFE6CF1027E8036400AD
:10003A000A000100001000011000010071776572CA
:10004A00747900000010020D4E414E000001000002AB
:10005A004E000000F894EE27EFBBF1E0F5BFE5BFD4
:10006A00F8E1AB95A4B7A77FA4BFF0936000E09336
:10007A0060008DE0A2E0BB27ED938A95E9F780E066
:10008A0094E0A0E0B1E0ED930197E9F7E6E5F0E04E
:10009A00859159191009761F0A591B5910590159 07C
:1000AA00BF01F00105900D920197E1F7FB01F0CF36
```

HEX

```
CPI  R26,LOW(@0)
LDI  R30,HIGH(@0)
CPC  R27,R30
LDI  R30,BYTE3(@0)
CPC  R24,R30
LDI  R30,BYTE4(@0)
CPC  R25,R30
```

ASM

```
CPI  R26,LOW(@0)
LDI  R30,HIGH(@0)
CPC  R27,R30
LDI  R30,BYTE3(@0)
CPC  R24,R30
LDI  R30,BYTE4(@0)
CPC  R25,R30
```

MACRO

Библиотеки

```
void main(void)
{ int i;
for(i=0;i<10;i++)
{
  printf("qwerty");
  i=i+1;
  i++;
}}
```

C

Транслятор

Языковые парадигмы. «Сверху-вниз»

Сверхвысокоуровневые языки программирования (VHLL — very high-level programming language)

Языки программирования с высоким уровнем абстракции.

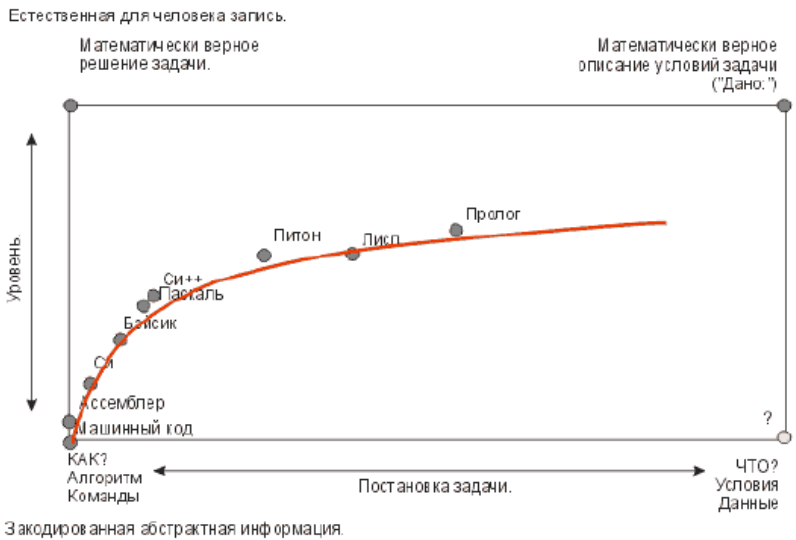
Основной принцип – **декларативность**, т.е. описание не того, «как нужно сделать», а «что нужно сделать».

- Современные скриптовые языки
- Декларативные
- Функциональные
- Предметно-ориентированные языки (для специфических приложений и задач) со своими особенностями синтаксиса.

Уровни языков программирования

Уровень языка программирования показывает, насколько язык близок к естественной для человека записи.

- **ООП.** Основа - процедурная модель программирования.
- **Функциональное программирование.** Программа состоит из совокупности функций. Лаконизм. Сложность реализации. Малое быстродействие.
- **Логическое программирование.** Основа – формальная логика. Декларативность.

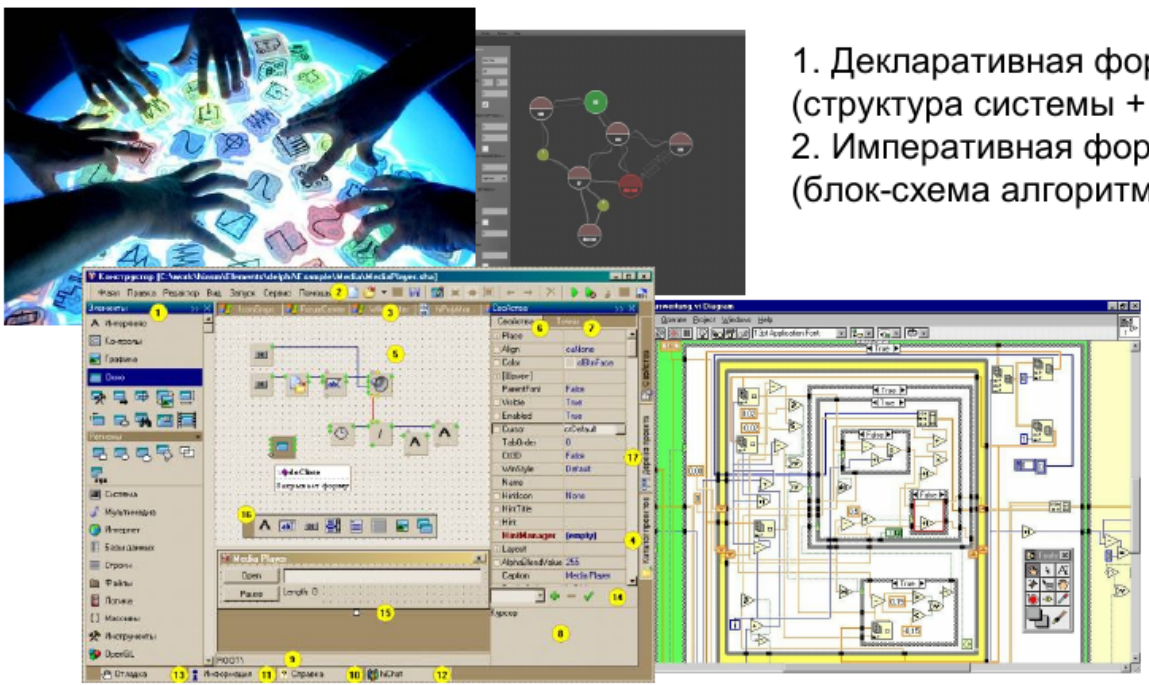


Основная парадигма развития языков – **максимальная автоматизация** процесса решения задачи.

Идеал – избавиться от программиста вообще.

«Визуальное» программирование

Еще одна форма автоматизации
Визуальная форма выражения способа решения задачи



1. Декларативная форма (структура системы + ...)
2. Императивная форма (блок-схема алгоритма + ...)

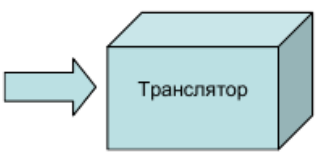
ПРОЦЕСС КОМПИЛЯЦИИ

ЛОГИЧЕСКАЯ СТРУКТУРА КОМПИЛЯТОРА

Постановка задачи

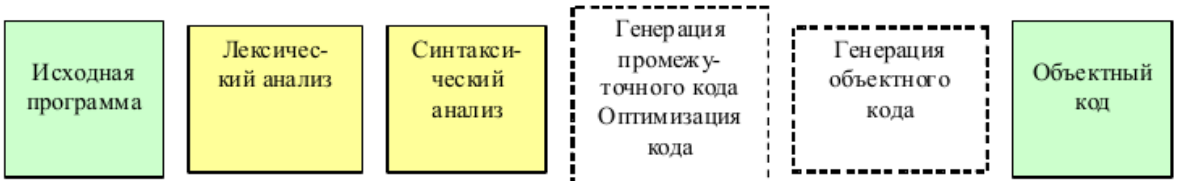
- Вход: исходный текст
- Выход: объектный код

```
// ...
void main(void)
{ int i;
for(i=0;i<10;i++)
{
  printf("qwerty");
  i=i+1;
  i++;
}}
```




```
:0A0000002EC0FECFFDCFFCCFFBCFDA
:10000A00FACFF9CFF8CFF7CFF6CFF5CFF4CFF3CFBA
:10001A00F2CFF1CFF0CFEFCFEECFEDCFECCFEBCFEA
:10002A00EACFE9CFE8CFE7CFE6CF1027E8036400AD
:10003A000A000100001000011000010071776572CA
:10004A007479000001002D4E414E000001000002AB
:10005A004E000000F894EE27EFBBF1E0F5BFE5BFD4
:10006A00F8E1A895A4B7A77FA4BFF0936000E09336
:10007A0060008DE0A2E0BB27ED938A95E9F780E066
:10008A0094E0A0E0B1E0ED930197E9F7E6E5F0E04E
:10009A008591959100976 1F0A591B591059015907C
:1000AA00BF01F00105900D920197E1F7FB01F0CF36
```

```
CPI  R26,LOW(@0)
LDI  R30,HIGH(@0)
CPC  R27,R30
LDI  R30,BYTE3(@0)
CPC  R24,R30
LDI  R30,BYTE4(@0)
CPC  R25,R30
```

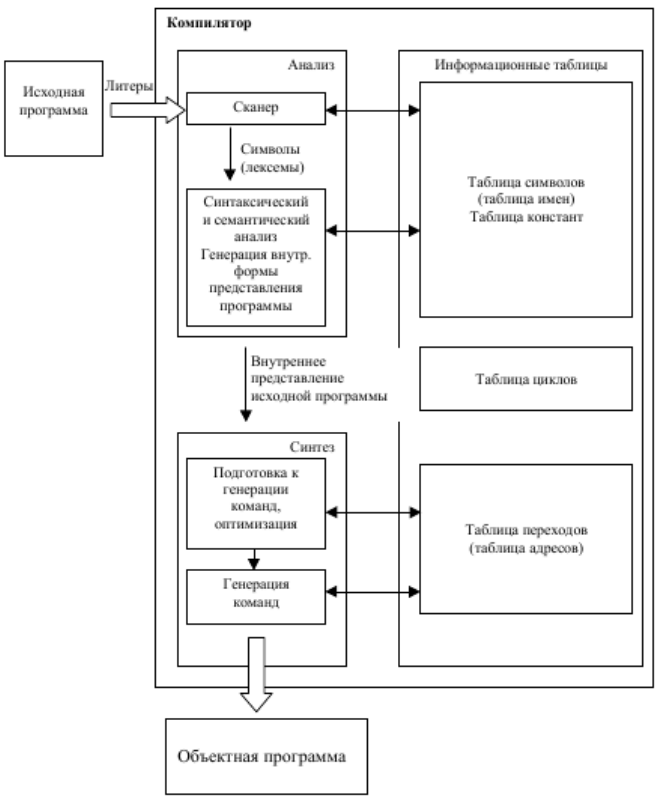


Ниже приведена классическая структура компилятора. Это – достаточно общая схем. Реальный компилятор чаще всего представляет собой целый программный комплекс.

Логическая структура компилятора

```
// ...
void main(void)
{ int i;
for(i=0;i<10;i++)
{
  printf("qwerty");
  i=i+1;
  i++;
}}
```

```
1.          ; 0000 0023 // Начало цикла
2.          ; 0000 0024 for(i=0;i<10;i++)
3.          ;       i -> R16,R17
4.                  __GETWRN 16,17,0
5.          _0x4:
6.                  __CPWRN 16,17,10
7.                  BRGE _0x5
8.          ; 0000 0025 {
9.          ; 0000 0026   printf("qwerty");
10.                 __POINTW1FN _0x0,0
11.                 RCALL SUBOPT_0x0
12.                 LDI  R24,0
13.                 RCALL _printf
14.                 ADIW R28,2
15.         ; 0000 0027   i=i+1;
16.                 __ADDWRN 16,17,1
17.         ; 0000 0028   i++;
18.                 __ADDWRN 16,17,1
19.         ; 0000 0029 }
20.                 __ADDWRN 16,17,1
21.                 RJMP _0x4
22.         _0x5:
23.         ; 0000 002A
24.         ; 0000 002B //Конец цикла
25.         ; 0000 002C
```



Рассмотрим далее некоторые ее составляющие.

Исходная программа – текст программы на языке высокого уровня, который должен быть переведен в машинный код.

Информационные таблицы – самостоятельные структуры, заполняющиеся в ходе лексического анализа и дополняющиеся во время работы.

Лексический анализатор (или сканер), имеющий на выходе поток лексем, нужен для того, чтобы убрать все лишнее (комментарии, разделители), выделить лексемы, т.е. лексические единицы, из которых строится машинный язык, и преобразовать их к внутренним или промежуточным формам представления. На этом этапе идет активная работа с таблицами, в которые заносится информация о распознанных лексемах, их типах, значениях и т.д. Результатом является поток лексем, эквивалентный исходному тексту.

Синтаксический анализатор необходим для того, чтобы выяснить, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики этого языка. Наряду с проверкой синтаксиса параллельно происходит генерация *внутренней формы* представления программы.

ОСНОВНЫЕ ЧАСТИ КОМПИЛЯТОРА

Итак, можно выделить следующие этапы компиляции:

1) Лексический анализ. Замена лексем их внутренним представлением (например, замена операторов, разделителей и идентификаторов числами).
2) Синтаксический анализ. Иногда на этом этапе также вводятся дополнительные разделители и заменяются существующие для облегчения обработки.
3) Генерация промежуточного кода (трансляция). На этом этапе осуществляется контроль типа и вида всех идентификаторов и других операндов. При этом обычно преобразование исходной программы в промежуточную (например, польскую) форму записи осуществляется одновременно с синтаксическим анализом.

4) Оптимизация кода.
5) Распределение памяти для переменных в готовой программе.
6) Генерация объектного кода и компоновка программных сегментов.

На всех этих этапах происходит работа с различного рода таблицами. В частности, для каждого блока (если таковые существуют в языке) идентификаторы, описанные внутри, запоминаются вместе со своими атрибутами. Условно все эти этапы можно изобразить следующим образом:


Исходная программа
Сканер
Синтакси-ческий анализ
Генерация промежу-точного кода
Оптимизация кода
Генерация объектного кода
Работа с таблицами

Очевидна зависимость структуры компилятора от структуры компьютера, точнее, от имеющейся производительности системы. Например, при малой памяти увеличивается количество проходов компиляции (т.н. *многопроходные компиляторы*), а при наличии памяти большого объема можно все этапы компиляции произвести за один проход (и тогда мы имеем дело с *однопроходным компилятором*). Тем не менее, независимо от вычислительных ресурсов, всю вышеперечисленную работу приходится так или иначе делать.

Далее мы рассмотрим вкратце некоторые из этих составных частей процесса компиляции.

Лексический анализ (сканер)

На входе сканера – цепочка символов некоторого алфавита (именно так выглядит для сканера наша исходная программа). При этом некоторые комбинации символов рассматриваются сканером как единые объекты. Например:

- один или более пробелов заменяются одним пробелом;
- ключевые слова (вроде BEGIN, END, INTEGER и др.);
- цепочка символов, представляющая константу;
- цепочка символов, представляющая идентификатор (имя);

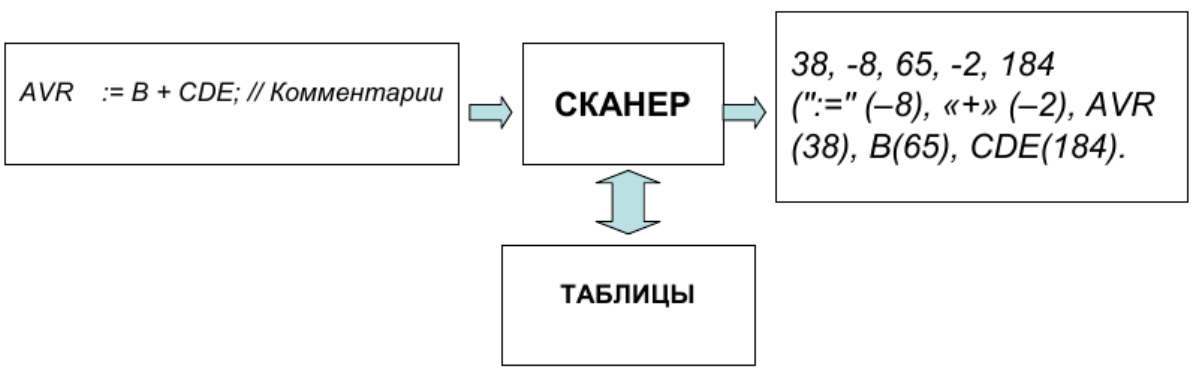
Лексический анализатор (сканер)

Вход: цепочка символов некоторого алфавита. При этом некоторые комбинации символов рассматриваются сканером как единые объекты. Например:

- один или более пробелов заменяются одним пробелом;
- ключевые слова (вроде BEGIN, END, INTEGER и др.);
- цепочка символов, представляющая константу;
- цепочка символов, представляющая идентификатор (имя);

ЛА группирует определенные терминальные символы (т.е. входные символы) в единые синтаксические объекты – *лексемы*.

Лексема: <тип_лексемы, значение>.



Таким образом, лексический анализатор (ЛА) группирует определенные терминальные символы (т.е. входные символы) в единые синтаксические объекты – *лексемы*. В простейшем случае лексема – это пара вида <тип_лексемы, значение>.

Задача выделения лексем из входного потока зачастую оказывается весьма нетривиальной и зависящей от структуры конкретного языка. Как будет интерпретироваться такая входная последовательность "567AB"? Это может быть одна лексема (идентификатор), а может – и пара лексем – константа "567" и имя "AB". Во втором случае либо между лексемами необходимо указание разделителя (например, пробела), либо надо заранее знать, что будет следовать дальше.

Существует два основных типа лексических анализаторов – *прямые* (ПЛА) и *непрямые* (НЛА).

Прямой ЛА определяет лексему, расположенную непосредственно справа от текущего указателя, и сдвигает указатель вправо от части текста, образующей лексему (ПЛА определяет тип лексемы, которая образована символами справа от указателя).

Непрямой ЛА определяет, образуют ли знаки, расположенные непосредственно справа от указателя, лексему этого типа. Если да, то указатель передвигается вправо от части текста, образующей лексему. Иными словами, для него *заранее* задается тип лексемы, и он распознает символы справа от указателя и проверяет, удовлетворяют ли они заданному типу.

У ПЛА более сложная структура – он должен выполнять больше операций, нежели НЛА. Тем не менее в большинстве современных языков программирования используется синтаксис прямых лексических анализаторов (это может быть видно по внешнему виду фраз языка).

Фортран – это классический пример языка, использующего непрямой лексический анализатор. Все дело в том, что в этом языке игнорируются пробелы.

Рассмотрим, например, конструкцию

DO5I=1,10 …

Для разбора этого предложения необходим непрямой лексический анализатор, который и определит, что означает цепочка "DO5I" – идентификатор "DO5I", или же ключевое слово "DO", за которым следует метка 5, а далее – имя переменной "I". Впрочем, аналогичные неприятности ожидают и разработчиков компиляторов языков типа C или C++, в которых существуют строковые комментарии "/*…*/" и "//…". При проведении лексического анализа, встретив символ "/", изначально неясно, является ли он оператором или началом строкового комментария. И вообще, многосимвольные лексемы – штука малоприятная для анализа.

Итак, на выходе сканера – внутреннее представление имен, разделителей и т.п.

Например:

Вход сканера:*AVR := B + CDE; // Комментарии*

Выход сканера: *38, -8, 65, -2, 184*

(Если мы условимся обозначать оператор присваивания ":=" числом с кодом –8, операцию сложения – числом –2, а имена переменных числами 38, 65 и 184).

Кроме того, сканер занимается формированием различного рода таблиц. И прежде всего – таблицы имен, в которую он будет заносить распознанные имена – идентификаторы, константы, метки и т.п.

Сканер. Таблица имен

Механизм работы с таблицами должен обеспечивать:

- быстрое добавление новых идентификаторов и сведений о них;
- быстрый поиск информации (*хеш-функции*).

№ элемента	Идентификатор	Дополнительная информация (тип, размер и т.п.)
1	ABC	идент., тип = "строка"
…	…	…
N	103	константа, число

```
/*   ...   */
int A;
A = B + 3.1415;
...
```

№ элемента	Имя	Дополнительная информация (тип, размер и т.п.)
1	A	идент., тип = ?; значение = ?
2	B	идент., тип = ?; значение = ?
3	3.1415	константа; тип = float; число; значение = 3.1415
…		

Синтаксический и семантический анализ

Синтаксический анализ – это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка. Это – самая сложная часть компилятора.

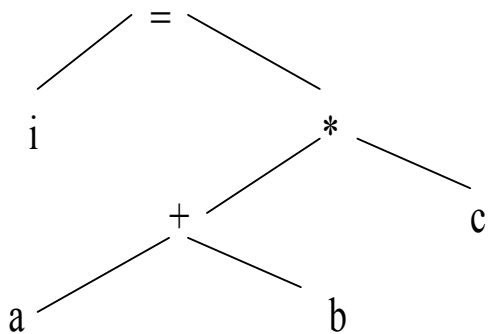
Синтаксический анализатор расчленяет исходную программу на составные части, формирует ее внутреннее представление, заносит информацию в таблицу символов и другие таблицы. При этом производится полный синтаксический и, по возможности, семантический контроль программы. Фактически, это – синтаксически управляемая программа.

При этом обычно стремятся отделить синтаксис от семантики насколько это возможно – когда синтаксический анализатор распознает конструкцию исходного языка, он вызывает *семантическую* процедуру, которая контролирует эту конструкцию, заносит информацию куда надо, проверяет на дублирование описания переменных, проверяет соответствие типов и т.п.

Предложения исходной программы обычно записываются в *инфиксной* форме. Однако эта привычная форма, в которой оператор записывается между операндами, является совершенно не пригодной для автоматического вычисления. Дело в том, что необходимо постоянно помнить о приоритетах операторов, "забегая" при анализе выражения "вперед". К тому же очень усложняют жизнь применяемые скобки, определяющие очередность вычислений.

Альтернативой инфиксной форме является *польская форма* записи (в честь польского математика Лукасевича): *постфиксная* и *префиксная*. Обычно под польской формой понимают именно постфиксную форму записи. Кроме того, используются и такие внутренние формы представления исходной программы, как *дерево* (синтаксическое) и *тетрады*.

Дерево. Пусть имеется входная цепочка i=(a+b)*c. Тогда дерево будет выглядеть так:



У каждого элемента дерева может быть только один “предок”. Дерево “читается” снизу вверх и слева направо. Дерево – это прежде всего удобная математическая абстракция. На практике дерево можно реализовать в виде списковой структуры.

Польская форма записи. Существуют три вида записи выражений:

- инфиксная форма, в которой оператор расположен между операндами (например, "a+b");
- постфиксная форма, в которой оператор расположен после операндов (то же выражение выглядит как "a b + ");
- префиксная форма, в которой оператор расположен перед операндами ("+ a b").

Постфиксная и префиксная формы образуют т.н. *польскую* форму записи. Польская форма удобна прежде всего тем, что в ней отсутствуют скобки. На практике наиболее часто используется постфиксная форма. Поэтому под польской обычно понимают именно постфиксную форму записи.

В этой форме записи выражение i=(a+b)*c выглядит так: " iab+c*=". Это выражение удобно расписывается по дереву: с нижней строки записываются “a” и “b”, далее “+” и “c”, выше – “i” и “*” и в вершине дерева “=”.

Тетрада- это четверка, состоящая из кода операции, приемника и двух операндов. Если требуется не два, а менее операторов, то в этом случае тетрада называется *вырожденной*. Например:

Исходное выражение	Код	Приемник	Операнд1	Операнд2
a+b→T1	+	T1	a	b
T1+c→T2	*	T2	T1	c
i=T2	=	I	T2	(вырожденная тетрада)

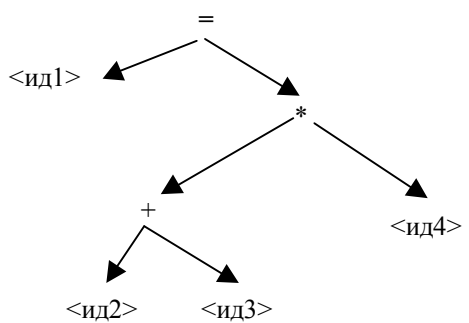
Польская форма записи и тетрады удобны своей однозначностью. Фактически в обеих этих формах мы раскладываем исходное выражение на элементарные составляющие.

Пусть на вход синтаксического анализатора подаются выражения

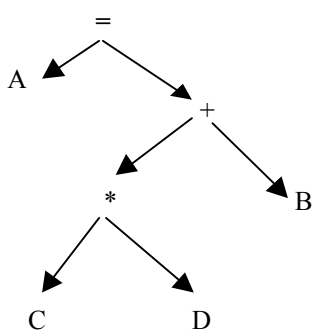
"<ид1>=(<ид2>+<ид3>)*<ид4>" и "A = B+C*D"

На выходе будем иметь:

1) Дерево для выражения "<ид1>=(<ид2>+<ид3>)*<ид4>"



Дерево для выражения "A = B+C*D"



2) Тетрады для "<ид1> = (<ид2>+<ид3>)*<ид4>"

+, <ид2>, <ид3>, T1

*, T1, <ид4>, T2

=, T2, <ид1>

Тетрады для "A = B+C*D"

*, C, D, T1

+, B, T1, T2

=, T2, A

(T1, T2 – временные переменные, созданные компилятором)

3) Польская форма для "<ид1> = (<ид2>+<ид3>)*<ид4>":
<ид1> <ид2> <ид3> + <ид4> * =

Польская форма для "A=B+C*D" будет выглядеть как "ABCD*+=". Обратите внимание на то, что *порядок следования операндов* в польской форме записи *такой же, как и в исходном инфиксном выражении* (записи "*abc*=*"* и "*bc*a="* – это вовсе не одно и то же).

Алгоритм вычисления польской формы записи очень прост:

Просматриваем последовательно символы входной цепочки. Если очередной символ является операндом (идентификатором или константой), то читаем дальше. Если символ является бинарным оператором, то извлекаем из цепочки два предыдущих операнда вместе с оператором, производим операцию и помещаем результат обратно в цепочку символов.

Вычисление польской формы

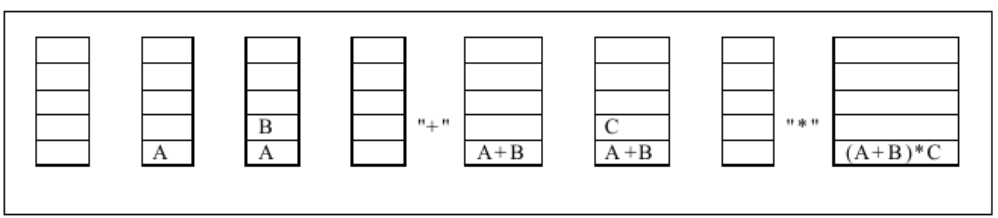
Просматриваем последовательно символы входной цепочки.
Если символ - операнд, то
помещаем его в стек и читаем дальше.
Кесли
Если символ является оператором, то
извлекаем из стека нужное кол-во операндов,
производим операцию
помещаем результат обратно в стек.
Кесли

```
while TRUE do
begin
case gettype(P[n]) of
        operand:
                Push(P[n]);
        operator:
                arg1 := Pop();
                arg2 := Pop();
                Push(arg1 @ arg2);
        else: error();
    endcase
    n := n+1
end
```

Стек

$E_{инф}$: (A+B)*C

$E_{пост}$: AB+C*



Более подробно этот алгоритм будет рассмотрен ниже. Оставшиеся стадии компиляции, связанные с подготовкой к генерации команд, распределением памяти, оптимизацией программы и собственно с генерацией команд и генерацией объектного кода, безусловно важны, однако сейчас на них останавливаться мы не будем.