Лекция 1

1. Програмни езици

Програмното осигуряване в голяма степен отразява специфичните особености на архитектурата на компютърните системи. Поради тази причина, усъвършенстването на хардуерната част на КС е съпроводено и с много бързо развитие на програмното осигуряване.

Хардуерните елементи на КС работят с инструкции записани в двоичен код. Това означава, че програмите които се изпълняват, трябва да съдържат само двоични инструкции (машинен код на програмите). Програмирането с машинни (двоични) команди е изключително трудоемка дейност. Това се смята за програмиране с програмни езици от първо поколение.

В началото на развитие на компютърните технологии, програмирането се е извършвало в машинен код. Първите опити за създаване на език за програмиране, различен от машинния датират от 1949 година. Тогава е разработена система за кодиране на програми наречена Short Code.

Още в началото на развитие на електронните изчислителни машини започват опитите да се облекчи програмирането, като се замени използването на двоични цифри за запис на командите и операндите към тях. За тази цел се въвежда мнемонически запис на различните команди, вместо техният шестнадесетичен код. Например, вместо цифровият код на командата за зареждане на регистър (запис на данни в регистъра), програмистът може да запише **LD** (от Load - натовари), вместо копиране на стойността на даден регистър в паметта – **ST** (от Store - запази), а вместо добавяне на стойност към съдържанието на даден регистър – **Add** (Add – добави). За записване на адресите в паметта, където се намират данните подлежащи на обработка се въвеждат правила с които на дадена област от паметта може да се присвои някакво описателно име (променлива, идентификатор). Първоначално, програмистите съставяли програмите с мнемонически означения на хартия и после ги превеждали на машинен език. Скоро станало ясно, че това може да извършва самата машина посредством специална програма.

През 1950 година Майкъл Уилкс от Кембриджският университет разработва така наречената **Assembly System** (транслираща система за език **Асемблер**), предназначена за компютър EDSAC. Целта на тази система е била да обединява (събира, асемблира) в една програма, отделни програмни части написани в двоичен код и съхранявани в специална библиотека за подпрограми. Тази система се явява начало на развитието на така наречените асемблерски езици за програмиране. Тези езици се използват и до днес.

В асемблерските езици, командите се записват с мнемонически кодове (кодови съкращения), които специална транслираща програма преобразува в машинни команди (двоичен код). При тези езици, за първи път величините се представят чрез буквени означения (идентификатори), както в математиката.

За времето си, разработването на асемблерските езици се е смятало за огромен скок в развитието на софтуерните технологии. За много програмисти от това време, тези езици представлявали ново ниво, което дава много по-големи възможности от машинните езици.

Машинният език и Асемблер са машинно зависими езици. Тяхната структура и налични команди зависят от конкретната компютърна система. Въпреки че се използват мнемонически означения на командите, структурата на записването им е близка до структурата на машинните команди. Те се отнасят към програмните езици от второ поколение.

При разработването на алгоритми за решаване на задачи, обикновено се използват прийоми и структури от по-високо логическо ниво, отколкото машинните инструкции на компютърните системи. Следвайки този подход специалистите занимаващи се разработване на програмно осигуряване стигат до извода, много по подходящо ще бъде програмите да се записват в термините на това по-високо логическо ниво, а специални програми да

преобразуват тези записи в инструкции от по-ниско ниво (машинни инструкции). През втората половина на 50-те години на 20-ти век се появяват първите програмни езици от високо ниво (трето поколение). При тях се разработва специален набор от езикови конструкции и правила за запис на инструкции от алгоритъма на дадена задача. Тъй като те се разработват като средство за описание на алгоритми, тези езици се наричат още алгоритмични или програмни езици Те се отличават от машинно зависимите езици, по по-високото структурно ниво на инструкциите и машинната независимост. Специални програми (компилатори, транслатори или интерпретатори) се използват за преобразуване на текста на програмите записани с алгоритмичните езици в двоичен код.

Машинната независимост на езиците от високо ниво се осигурява от програмата, която преобразува текста на програмата от високо ниво в двоични команди (компилатор, интерпретатор). Тъй, като машинните инструкции на различните компютърни системи са различни, то и компилаторите трябва да съответстват на даденият тип машина. Затова, в миналото имаше различни компилатори за един и същ език, които се наричаха ***диалекти*** на езика.

През 1954 година започва работата по създаването на езика **FORTRAN** (FORmula TRANslation). През 1957 година той е завършен и по идея този език е реализиран като средство за създаване на програми за решаване на научно-технически задачи. Има различни версии: FORTRAN II (1958), FORTRAN III (1961), FORTRAN IV (1962), FORTRAN 77 (1966). FORTRAN получава много голямо разпространение и се използва и до днес.

Езикът **Алгол** се явява европейски вариант на алгоритмичен език от високо ниво. Той се разработва по поръчка на международен комитет, който се формира през 1958 година в Цюрих. През 1960 година този комитет се събира за втори път, който дава наименованието на този език **Алгол 60**.

Езикът Алгол внася съществено нови идеи в езиците за програмиране. В него се залага така наречената блокова структура в програмите. Той има голямо значение за разработваните след това програмни езици. За него професорът по информатика Тони Хоар казва, че това е език, който толкова е изпреварил времето си, че се явява подобрение не само на своите предшественици, но и на почти всички свои наследници. Алгол не е получил голямо разпространение, но той има сериозно теоретично значение. Една голяма група програмни езици се обединяват в група с наименование алголоподобни езици.

Езикът FORTRAN получи широко разпространение, но той не може да удовлетвори многостранните интереси в областа на програмирането. За приложения в областта на деловата сфера (икономика, финанси) е разработен специален програмен език **Кобол**. Той е поръчан от Министерството на Отбраната на САЩ. Кобол е универсален език за обработка на информация в големи бази от данни (икономически и статистически изчисления).

Стремежът за създаване на универсален програмен език се въплъщава в разработката на езика **PL\1.** Това е универсален програмен език, който обединява идеи от FORTRAN, Алгол и Кобол. Въпреки безспорните си достойнства той не можа да намери голямо приложение, но също изигра важна роля в по-нататъшното развитие на програмните езици.

През 1967 е създаден програмният език **Бейсик,** който е близък по идеи с езика FORTRAN, но с по-опростена структура. Той е замислен като език за лесно програмиране, дори и от хора, които нямат специални знания по програмиране. С него се работи преди всичко в диалогов режим. Програмите написани в Бейсик се изпълняват посредством интерпретатор – програма, която последователно преобразува командите в машинен код и ги изпълнява. Бейсик става основа за развитието на персоналните компютри в началната им фаза на развитие. Тогава този език се използваше като операционна система за компютъра.

Интересно е да се отбележи, че версията М-Бейсик е разработена от Уилям Гейтс и Пол Алън и с него е поставено началото на най-мощната в момента софтуерна компания Microsoft.

В краят на 60 – те години на миналия век, вече съществували доста програмни езици, които имали очевидни недостатъци. По тази причина били сформирани две изследователски групи, които трябвало да разработят език без недостатъци (универсален програмен език). Този език е трябвало да включва всички положителни идеи от Фортран, Алгол, Кобол, Бейсик и др. Като резултат били разработени езиците PL/1 и Алгол-68 (1967 и 1969 год. съответно). Групата разработваща PL/1 била финансирана от IBM. Нейните усилия не пропаднали, и сега много програмисти ползват този език. Но въпреки всичко той не можа да измести основните езици, Фортран, Кобол, Бейсик.

Алгол-68 бил разработен от група европейски компютърни специалисти. Някои го оценяват много високо, предимно сред академическите среди. Но той се оказа много език, изискващ много машинни ресурси. Освен това изучаването му е доста трудоемка работа.

Един от проектите, които били отклонени от групата разработваща Алгол-68, бил проекта на специалиста от Института по информатика в Цюрих, Никлаус Вирт. Разработването на програмни езици било хоби на Вирт и той не прекратил разработването на език, който да замени, ползващият се със сериозна репутация език Алгол-60. През 1971 година той публикува описанието на своя език и го нарича на името на изобретателя на първото изчислително устройство – Паскал. През 1973 е разработен първият транслатор за този език, а малко по-късно е разработена програмната среда Turbo, получила много голяма популярност сред програмистите в целият свят.

И така през 1971 година е разработен езикът **Pascal**. Той получава голямо разпространение. Близък е по структура до езика Алгол, но има много нови идеи от структурното програмиране. Използва се за разработване предимно на научно-технически приложения, но има и големи възможности за бизнес приложения.

Главната новост на този език е, че удовлетворява изискванията на обикновените потребители и специалистите по изчислителна техника. Това е първият език, който може лесно да се научи и е логически правилно построен. Граматическите правила на Pascal могат да бъдат написани на 4 – 5 страници.

C (C++) е програмен език създаден през 1972 година. Това е универсален език с големи възможности. Близък е по структура до Pascal, но е с по-големи възможности за разработване на системно програмно осигуряване. Големи части от операционната система Windows са написани на езика C и C++.

Програмният език **АДА** е създаден през 1979 година. Той е универсален език за програмиране с развити възможности за структуриране на данни. За него е характерна модулната организация в процеса на програмиране и намира широко приложение при програмиране на вградени микрокомпютърни устройства в различни машини и оборудване.

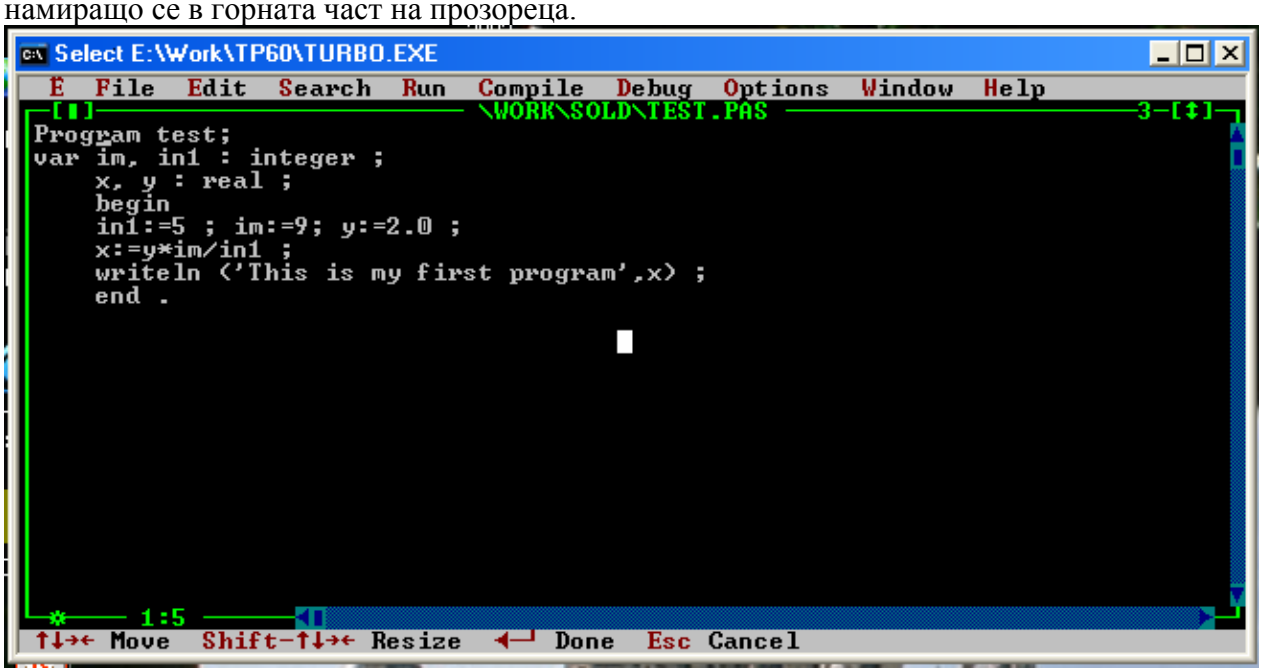
2. Интегрирана програмна среда

Програмните средства, които осигуряват написването, редактирането, изчистването от грешки, преобразуването в двоичен код и стартирането на една програма, написана с помощта на определен програмен език се нарича интегрирана програмна среда. Най-общо, интегрираната програмна среда съдържа следните основни елементи: текстов редактор, компилатор, свързващ редактор, дебъгер, система библиотечни функции, помощна система.

2.1. Текстов редактор.

Това е програма която подържа написването и редактирането на текста на програмите и съхраняването им в подходящ вид върху постоянен външен носител (диск). Текста на програмата може да се напише на какъв да е редактор, например Notepad, но редакторът на интегрираната програмна среда предоставя някои улеснения: записва файла с подходящо разширение (‘.pas’- за Pascal,’.cpp’- за C++,’.bas’- за Basic и др.), отбелязва ключовите думи в даденият език, предоставя помощна информация за програмният език и др. На фиг. 3.1 е показан общият вид на прозореца на програмната среда TURBO Pascal, основната част на

която е текстовия редактор, а другите елементи на средата се активират с основното меню намиращо се в горната част на прозореца.



Фиг. 3.1 Програмна среда Turbo Pascal

2.2. Компилатор (транслатор).

Компилаторът е програмна система, която преобразува текста на програмата в машинни (двоични) команди. Процесорът на КС може да изпълнява само двоични команди. Затова, програмите написани на даден програмен език трябва да се преобразуват в двоични команди. Процесът на превеждане на програмата на друг език (в случая в машинен език) се нарича транслация (компилация). Оригиналният вид на програмата се нарича изходен код (**source**), а преобразуваната версия – обектен код (**object code**). Процесът на транслация се състои от три етапа – логически анализ, синтактически разбор и генериране на обектен код.

Логическият анализ е процес на формиране на отделните символни групи от текста на изходната програма. Например, когато се срещне група символи '**if**' тя трябва да се интерпретират от транслатора като съвкупност, която е ключова дума в програмният език и има точно определена функция при записа на някои оператори. Или ако се срещнат символите '154', те трябва да се интерпретират не като отделни цифри 1, 5 и 4, а като единно числово значение – числото 154. Тази дейност изглежда доста елементарна, но за компютърната система, това е важна и отговорна дейност, защото машината не притежава интелигентност и за нея всички символи са само номера от таблицата за ASCII код.

Целта на логическият анализ е, чрез последователно прочитане на символите от изходната програма, да се образуват самостоятелни логически единици. Тези самостоятелни текстови единици се класифицират, като се определя вида им – числа, думи, аритметически операции и т.н. Генерираните текстови единици се предоставят на синтактическият анализатор за по-нататъшна обработка.

При синтактическият разбор се игнорират всички коментари и се анализира програмата в термините на логически единици формирани от логическият анализатор. Задачата на синтактическият анализатор е да обедини логическите единици в оператори и да идентифицира граматическата структура на програмните елементи и разпознае ролята на всеки компонент. За опростяване на синтактическият разбор, ранните езици за програмиране изискваха всеки оператор от програмата да се разполага в определено място на бланката за изходния текст. Тези езици се наричат езици с фиксиран формат. Сега се използват предимно езици със свободен формат.

Процесът на синтактическият анализ се основава на съвкупност от правила, определящи синтаксиса на програмният език. Един от начините за представяне на синтактическите правила се състои в използването на така наречените синтактични диаграми. Отделните редове на програмата се сравняват за съответствие с някаква съвкупност от синтактически диаграми. Съставя така нареченото дърво на синтактическия анализ за изходната програма – граматическа структура на програмата.
При синтактическият анализ на операторите за декларации, съдържащата се в тях информация се записва в специална таблица на символите. Тази таблица определя какви променливи са описани в програмата и какви типове данни или структури са свързани с тях.
Компилаторът прави проверка за правилността на текста на програмите и формира подходящи съобщения за допуснатите грешки при записа на отделните оператори в програмите. Компилаторът открива и съобщава само за така наречените синтактически грешки. Това са грешки от формално естество – неправилно записани оператори, пропуснати части от команди, използване на недекларирани променливи и др.
След коригиране на всички синтактически грешки, компилаторът преобразува текста на програмата в съвокупност от двоични команди, които записва в междинен файл (**obj -** обектен файл). Този процес включва различни аспекти, един от които е повишаването на ефективността на генерираният код. Например, нека трябва да се транслира последователност от следните два оператора

x = y + z ;

w = x + z ;.

Тези оператори могат да се транслират поотелно. Това обаче ще бъде неефективно. Генераторът на двоичният код трябва да отчете, че когато се изпълни първият оператор, променливите ***x*** и ***z*** вече се намират в регистрите с общо предназначение на микропроцесора и не е необходимо отново да се зареждат в тях от паметта. Използването на такива подходи при генерацията на двоичния код се нарича **оптимизация** на кода.

2.3. Свързващ редактор.

Обектният файл, който създава компилаторът макър и записан в двоичен код все още не е готов за изпълнение от микропроцесора. Една от причините за това е, че програмните езици позволяват програмите да се изготвят от отделни модули, компилирани в различно време. Затова обектният код получен при транслацията често представлява само някаква съставна част от цялостната програма, която трябва да се свърже с другите части за да се получи едни цяло. Даже програмата да се състои само от един модул, обектният код от транслацията не е готов за изпълнение, тъй като към него не са включени стандартните библиотечни функции, които осигурява програмната среда.
Свързващият редактор е програмна система, която свързва файловете създадени от компилатора в единнен изпълним файл (приложение). Той добавя всички необходими библиотечни подпрограми към създадените от компилатора обектни файла и ги свързва, като създава изпълним файл (разширение ‘.exe’) или библиотечен файл (разширение ‘.dll’). Свързващият редактор проверява правилността на всички връзки между отделните обектни модули и генерира съобщения, ако има грешки.

2.4. Дебъгер.

Дебъгерът е програмна система, която подпомага програмистите за отстраняване на грешки от логическо естество, допуснати в програмите. Тези грешки не са програмни, а грешки или пропуски в алгоритъма на задачата. Тези грешки могат да доведат до опит за изпълнение на недопустими операции от микропроцесора (аритметично деление на нула, препълване на полето за дадена променлива, адресиране извън адресното пространство на паметта и др.). Тогава се генерира съобщение за вида на грешката и адрес в паметта, където се намира двоичната команда предизвикала грешката. В други случаи, грешката няма да предизвика

спиране на изпълнението на програмата, но получаваните резултати нама да са верни заради грешният алгоритъм. Тези грешки са по-опасни и в редица случаи се не се откриват или се откриват след като програмата вече е била използвана.
Исторически за тези грешки е останало наименованието ‘бъг’ – ове (‘bug’ – буболечка) и поради тази причина самата програма се нарича дебъгер (премахване на ‘бъг’ – ове.
Основният проблем, при отстраняването на логическите грешки е че генерираните съобщения се отнасят до адреси в паметта, където се намират машинните команди, а програмистът разработва логиката на програмата на основата на първичният текст написан на определен програмен език. За да се установи съответствие между машинните команди и текста на програмата, програмната система **дебъгер,** изготвя специален файл – карта на паметта на програмата (‘.map’ – файл). В тази карта се записва относителният адрес на машинната команда в паметта, с която започва всеки ред от текста на програмата. При наличието на тази карта на паметта, всеки адрес съпровождащ дадено съобщение за грешка, може да бъде отнесен към определен ред в изходният текст на програмата.
Дебъгерът предоставя, някои специални средства за проверка и отстраняване на логически грешки. Така например, могат да се маркират определени оператори в текста на програмата (точки на прекъсване), в които да се спира изпълнението на програмата и да се проверяват стойностите на някои променливи с цел да се проследи развитието на даден процес на обработка на информацията. Освен това, програмата може да бъде стартирана в така нареченият режим ‘трасиране’. При него програмата се изпълнява стъпка по стъпка (оператор след оператор) и може да се следи изменението на стойностите на някои величини.

2.5. Подсистема с библиотечни функции.
Това е система. Която осигурява използването на множество стандартни функции, които облекчават програмирането на дадения алгоритмичен език. Тези библиотечни функции се групират в различни раздели – функции за входно-изходни операции, математически функции, функции за работа със текстови низове и т.н. Колкото повече са библиотечните функциивключени в дадена програмна среда, толкова по-големи възможности предоставя тя на програмистите.

2.6. Подсистема с помощна информация.
Тази система осигурява обяснителни текстове за голяма част от операциите и правилата използвани в даденият програмен език.