Mobile Computing

Mobile computing and related networking issues have received considerable attention in the academic and industrial research community. So, it's logical that computer science departments in various universities have started offering a graduate-level course on this topic.

Several departments offer the course as a research seminar where the instructor and students study a set of research papers and work on some projects. Other places structure it as a formal course. Both instruction approaches face similar challenges. There is no comprehensive, widely accepted textbook on mobile computing. Recently, Charles E. Perkins published Ad Hoc Networking (see the "Further Reading" sidebar for more details), a collection of articles on mobile ad hoc networks (MANETs). However, this book addresses only a specific area of mobile computing. Similarly, while selecting a set of papers for a mobile computing course, how do you determine that the selection truly represents the area?

In my four years of teaching a graduate-level course on mobile computing systems at the University of Texas at Dallas, I have seen the course evolve from a seminar to a formally structured course. Based on this experience, this article describes what a semester-long mobile computing course should cover. My choices reflect my bias; I'm sure others have differing views, and I hope this article generates discussion.

What is mobile computing?

Students must be able to distinguish between mobile computing and wireless networking. These two terms are not synonymous. You could do computing on the move with a wired connection to the network wherever you go: office, home, conference, hotel, and so on. However, wireless networking is, and will continue to be, an important ingredient of mobile computing because it offers users greater flexibility of movement.
Students must also be sensitized to the scarcity of resources in a mobile computing system, especially if the communication medium is wireless. Today's mobile computers run on batteries with a limited energy supply and might have low communication bandwidth. Technological advancements will improve battery performance and increase wireless-network bandwidth, but the resource scarcity is relative. Mobile computing environments will continue to be resource-poor compared to their fixed counterparts, at least in the foreseeable future. Some early papers survey mobile computing's challenges (see the "Further Reading" sidebar).

A discussion of resource constraints challenges the students to think of new solutions and new ways of porting existing network applications to mobile computing. At the same time, consideration of the offered flexibility and increased information availability should motivate them to think of new applications. After all, everybody is waiting for that killer application!

**What do mobile networks look like?**

To initiate meaningful class discussion on mobile computing, the course must first cover various network models. Students need to know the most common models:

- Nomadic users with wired network connectivity at access points. A nomadic user might disconnect from an access point before a move and might later reconnect with an access point. In the interim period, the user operates in disconnected mode, relying solely on information resident on the mobile computer.
- A cellular-like network with a wired infrastructure and wireless connection between a user terminal and the network infrastructure. A wireless LAN with a connection to a larger infrastructure network belongs to this category. The wireless LAN resembles a cell in a cellular network.
- A MANET with no wired infrastructure. In this model, all the nodes are mobile, and communication is over wireless links. Because paths between nodes might comprise multiple wireless links, each node should be capable of data forwarding. A MANET
might have a gateway that connects it to other networks. In the simplest form, all MANET nodes are identical. However, other possibilities exist—for example, a hierarchical mobile network where some nodes are more powerful, have an abundant energy supply, and collectively support the relatively resource-poor nodes.

**Mobile computing issues**

Next, the course needs to cover design issues. The approach I prefer is to progressively ascend the network protocol stack and consider the relevant issues. This mirrors most networking courses' organization. Because most students in a mobile computing course have already taken a networking course, they will probably feel comfortable with this sequence.

The data link layer

It is relatively easy to convince students that, with the increasing popularity of mobile computation and communication systems, the demand for the limited number of wireless-communication channels is also increasing. So, efficient channel utilization is of paramount importance. To illustrate the importance that service providers attach to bandwidth availability, I talk about the high price they pay for it in FCC bandwidth auctions. Even though the course will cover mobile computing applications later, it is important to state early on that applications' needs for channels differ significantly. Most non-real-time packet-based applications do not require a dedicated channel. However, real-time applications that expect a certain quality of service (QoS) would be best served by having some dedicated bandwidth. This topic provides an easy transition to discuss reservation-based solutions such as

- channel allocation for cellular networks and
- contention-based approaches for wireless LANs.

Channel allocation. This solution is appropriate when connections between the mobile node and the base stations will be long-lived.
Proposed channel allocation techniques take one of two approaches: centralized or distributed. The course should emphasize the relative strengths of these approaches. If centralized solutions are simpler and always result in channel utilization that is at least as high as that for distributed solutions, why use distributed solutions? This question lets me initiate discussions on reliability and scalability: centralized solutions have a single point of failure and might not scale well. To a student who has already taken a distributed computing course, the advantages of distributed solutions are obvious. Others start appreciating this point early in the course, which helps later when we cover MANETs.

Similarly, the course needs to discuss the relative merits of fixed channel allocation and dynamic channel allocation algorithms. FCA assigns a fixed number of channels to each cell in a cellular network. It can result in poor channel utilization, especially if the spatial distribution of demand is nonuniform. DCA protocols, while resulting in better channel utilization, tend to be more complex and require more communication between base stations.

A third alternative is hybrid channel allocation, which divides the set of channels into a fixed and a dynamic set. This solution divides channels in the fixed set into subsets of nominal channels associated with each cell and allocates these channels using FCA algorithms. Channels in the dynamic set are available to all cells and are allocated using DCA algorithms when a cell's nominal channels are all in use.

By the end of the channel allocation discussion, students notice that they have sacrificed design simplicity for better utilization, scalability, and reliability. However, the course needs to emphasize that simplicity of design can help prove the correctness of the design and implementation. Otherwise, some students will develop a complicated solution that is only marginally better than a much simpler, obvious solution.

See the "Further Reading" sidebar for papers on channel allocation.
Contention-based approaches. Early on, the students need to understand the hidden terminal problem. Those who have taken networking courses have studied carrier sense multiple access with collision detection (CSMA/CD) protocols, which rely on the transmitter for collision detection. However, in wireless LANs, an interferer can be in the intended receiver's communication range but the transmitter doesn't sense the interferer's transmissions. In this situation, the transmitter won't sense a collision even though the receiver experiences interference, and CSMA/CD will not yield desirable performance. Therefore, protocols such as IEEE 802.11 employ collision avoidance. Wireless LANs can also operate in contention-free mode when the base station regulates access to the channel. So, unlike Ethernet, where control is distributed, protocols such as IEEE 802.11 support both distributed and centralized operation.

**Location management and Mobile IP**

Location management in mobile networks doesn't fit neatly into a layer of the protocol stack. However, I cover it before Mobile IP and routing because

- call establishment in cellular networks and packet routing in mobile networks must deal with a similar problem, and

- the solutions are similar in both contexts.

The course first examines the following issues, which results in interesting discussions:

- A change in a mobile node's location means a change in the route to that node. Should the mobile node be responsible for informing all its correspondent nodes about changes in its location?

- Should the correspondent nodes be responsible for tracking the mobile node? What if a correspondent node is old and doesn't have mobility support? Does this mean that the node can't communicate with mobile nodes?
A mobile node can't predict which nodes might want to communicate with it. So, if it is responsible for location updates, should it send its location updates (and the resultant route updates) to the entire network?

A mobile node might want to communicate with other nodes without revealing its location. How do you support such a facility?

Location management in cellular telephony employs home location registers and visitor location registers. When a mobile phone is in a foreign region, it registers with the VLR serving the foreign region. The VLR conveys this registration information to the mobile phone's HLR. Subsequently, when a call for the mobile phone is routed to its home region, the HLR forwards the call to the VLR. The VLR determines which cell the phone is in and completes the call.

Mobile IP, the protocol to deliver IP packets to mobile nodes, has a similar approach. Home agents and foreign agents correspond to the HLRs and VLRs. Unlike mobile-telephone networks, Mobile IP doesn't support real-time handoff of an ongoing call between cells. For details of Mobile IP, the Internet Engineering Task Force's (IETF) Web page on the Mobile IP working group (http://www.ietf.org/html.charters/mip6-charter.html) is a good starting point, along with Perkins' book Mobile IP Design Principles and Practices (see the "Further Reading" sidebar).

**Routing**

After studying the data link layer issue of delivering frames across a wireless link—that is, between adjacent nodes—the course moves on to data communication between nonadjacent nodes. Routing in mobile networks is a very interesting problem. We could spend an entire semester on it.

The routing solution depends on whether the mobile computing model is nomadic, cellular, or a MANET.

Nomadic computing. As I mentioned before, this model assumes no network connectivity during a move. So, each time a node connects to
the network, the node could register with appropriate name servers and locate the servers that provide the desired services by probing the known directory services. Subsequently, packets could be routed to the mobile node. If the node's mobility will be hidden, the network could employ Mobile IP. All the communicating nodes could route packets to the mobile node's home address. There, the home agent would intercept the packets and tunnel them to the mobile node.

Cellular computing. Cellular networks that provide continuous connectivity and data communication for mobile nodes take a different approach. As a mobile node moves out of one cell into a neighboring cell (or from one wireless LAN to a neighboring wireless LAN), seamless handoff between the cells is needed. A mobile computing course must discuss this problem for three reasons:

- Unlike with handoff of voice calls in cellular networks, occasional loss of a few data packets can severely degrade the performance of several data communication applications.

- Applications that rely on data communication in a cellular setting will become prevalent when third-generation packet radio networks become a reality. The cellular networking domain currently doesn't have many such applications, owing to the low available bandwidth.

- Depending on the nature of the application, data packets might or might not have delivery deadlines and reliability requirements. So, QoS issues are important in routing decisions. The added dimension of mobility makes the problem more challenging. QoS research and solutions related to backbone networks do not necessarily address the issue of source and destination mobility.

MANETs. Owing to the absence of any infrastructure and the need for all nodes to collaborate in routing and packet forwarding, MANET routing is fertile ground for the application of distributed computing ideas. While no deployed MANETs exist, such networks have interesting potential civilian and military applications. Also, this area has several interesting proposed solutions and many unsolved problems. Hence, I
can't overemphasize this topic's importance in a graduate-level mobile computing course. While the IETF MANET working group is relatively new, similar problems of routing and packet forwarding were investigated in the context of DARPA's packet radio networks in the '70s and '80s. So, students must have a good historical perspective lest they reinvent the wheel.

Owing to the dynamic nature of MANETs, any routing protocol for them should undergo these tests:

- What volume of control traffic do you need to propagate routing information?
- How much time do the routing tables at all the nodes take to converge after a topology change?
- Are there situations when the routing tables might not stabilize?
- Does the protocol determine routes that are optimal with respect to the metric that is being optimized? Some protocols try to minimize the number of links on the path between the source and destination. Others might have different goals, such as load-balancing among all nodes, minimizing energy consumption, or minimizing latency.
- Are the routes stable?
- Wireless links might be unidirectional. Can the routing protocol utilize such links? If so, do you need any modifications at the data link layer?
- What fraction of packets is dropped owing to nonavailability of routes even though the source and destination are part of the same connected component?
- What fraction of packets is misrouted?
The answers to these questions will determine if the routing protocol is scalable. Such scalability will be a requirement for the wide deployment of MANETs.

Discussing the abundance of proposed routing protocols for MANETs in minute detail is neither possible nor useful, especially because several have similar design principles. (For a brief description of several MANET routing protocols, see the entry for Charles Perkins' Ad Hoc Networking in the "Further Reading" sidebar.) So, I find it useful to discuss the criteria for classification of MANET routing protocols and the relative strengths and suitability of each category of protocols. Graduate students are mature enough that, once they understand the principles, they will read the Internet drafts and absorb the details.

You can classify MANET routing protocols on the basis of when routing information is gathered or propagated or on the basis of the network's logical organization. In the first classification, routing protocols can be proactive or reactive. Proactive protocols rely on the periodic exchange of routing information between nodes, and are triggered by network topology changes. Subsequently, when a node has to forward data packets, it performs a table lookup. Proactive protocols have been criticized for incurring high route update overheads even when no data communication takes place between nodes. This criticism has led to the development of reactive protocols that perform route discovery only when there is data to forward and then cache the discovered routes. Subsequently, if a source sends additional packets to a destination, it can use the cached route information.

Proactive protocols have been discarded in haste. Students must realize that they should compare protocols for a range of communication-to-mobility ratios. Proactive protocols will definitely lose when communication is infrequent and mobility is high (requiring frequent routing updates). However, in the reverse situation, reactive protocols might lose their advantage over the proactive protocols. Furthermore, don't reactive protocols suffer from higher latency due to on-demand route discovery?
In terms of the network's logical organization, MANET routing protocols can be cluster-based and hierarchical, or flat, or a hybrid of the two. Cluster-based protocols logically divide the network into clusters, each having a head. The heads collectively maintain topology information and provide information for packet forwarding to their cluster members. Flat protocols maintain no such node hierarchy; all nodes maintain routing tables. So, when would a hierarchical protocol be better than a flat protocol, and vice versa? There does not seem to be significant literature on this question, and I find that some students are intrigued by this question.

Some researchers suggest that nodes should track the location of other nodes for packet-forwarding purposes. A packet could be forwarded to a neighbor in the destination's general direction, and topology information wouldn't need to be disseminated throughout the network. Students must understand that such solutions work only under some circumstances. Wireless signals do not propagate outward in nice circles. Instead, the terrain, the vegetation, the weather, and activities by other transmitters influence a received signal's quality. At times, the route that a packet should take between two nodes might have little correlation to the straight line connecting those nodes.

I use the abundance of competing MANET routing protocols and the sometimes conflicting claims made by the related papers to instill a measure of skepticism among the students. A few students have, unfortunately, a tendency to unhesitatingly accept anything that is published in a reputed technical journal or conference. It's interesting to propel the students to thoroughly investigate the validity of claims made for various protocols.

In addition to unicast communication, the course should cover multicasting and broadcasting. No paper on multicasting protocols conclusively proves that every node in the multicast group gets the packet.
The transport layer

Teaching issues pertaining to the transport layer, especially TCP, can get tricky in the context of mobile networks. At this layer, only end-to-end considerations matter. However, in a mobile network, the data link layer characteristics of wireless links, such as high bit error rate and low bandwidth, can significantly affect the performance of TCP connections. Consider a TCP connection in a cellular system where only one link (the last or first) is wireless. Packets dropped across a wireless link will result in TCP time-outs and retransmissions by the source. Retransmitted packets will travel over the entire path, including the reliable wired links.

Students need to determine the best possible solutions that do not violate the end-to-end TCP semantics or the layered protocol stack approach. For example, the Snoop protocol relies on an agent at a base station to monitor the reliable delivery of link layer frames carrying TCP traffic. Does this solution stay true to the layered approach of solving networking problems? Should transport layer information be available to the data link layer?

Indirect TCP, another solution, relies on establishing two connections for a TCP connection. One connection spans the relatively reliable and bandwidth-rich wired part; the other spans only the wireless link and is optimized for this link. Some people object to I-TCP because it violates the end-to-end semantics of TCP connections. (For articles describing Snoop and I-TCP, see the "Further Reading" sidebar.)

These topics should lead to further discussion about the layered approach to solving networking problems. Ideally, intermediate nodes should treat all the packets the same, whether they carry TCP or UDP (user datagram protocol) traffic. So, how widely deployable is a solution that expects intermediate nodes to do special processing for TCP traffic? Will such a solution work for TCP connections that span several diverse networks, all of which might not be under one administrative control?

For MANETs, the network’s dynamism and the resultant changes in
path length and propagation time can impact TCP's mechanisms for congestion control in ways that are difficult to predict. So, are TCP's simple time-out and self-clocking mechanisms good enough to handle such dynamism? Or do we need new solutions for end-to-end traffic management in a MANET?

**Mobility support for applications**

At this point, students should understand the issues relevant to end-to-end communication between nodes in a mobile network. So, this is an opportune moment to apply this know-how to the design of new applications and to porting existing applications to mobile networks.

An important issue in designing applications for mobile networks is the disconnected operation mode. We can realistically assume that the mobile node, to save energy, will not always be connected to the network. Even when the node is connected, the available bandwidth and the bit error rate might fluctuate. Applications must be aware of such conditions and adapt to them. The class should discuss how applications can adapt to intermittent network connectivity. A solution that several students usually offer and that appears in the scientific literature is increased reliance on a locally stored copy of the files or database records.

However, studying this solution should trigger a thorough reexamination of existing cache-coherence and replica-consistency solutions in the context of mobile applications. Reads performed on the local copy are not guaranteed to return the result of the latest write. Because writes might be performed concurrently on mutually unreachable replicas, the replicas' state might diverge. So, when two previously unreachable nodes with mutually divergent replicas become reachable, how do you synchronize the replicas? It's interesting to nudge the class toward the realization that the insistence on strict replica consistency or cache coherence they learned in database and computer architecture courses, respectively, might be bad for the disconnected mode: data availability will be adversely affected. So, we
need a weaker notion of replica consistency. However, what is the meaning of weak data consistency? We do not yet fully understand the related issues of synchronization. The exploration of such issues is especially interesting for a graduate-level course.

A related issue is network-aware computing. As the mobile device moves between heterogeneous networks (from connection to a wired access point, to a wireless connection through a wireless LAN, to a MANET) and as the speed of motion changes, the quality of the network link and of other available resources might change significantly. So, the application servers and clients should be able to adjust accordingly. For example, a user walks along a street carrying a mobile node with a high-bandwidth wireless link but with a limited energy supply and a very basic I/O interface. Later, the user enters a car, slips the mobile node into its docking port, and starts driving along a highway. Now, the node has a relatively abundant energy supply from the car's battery and a much better I/O interface, but a much lower wireless bandwidth owing to the car's high speed. What do the server and client need to do to hide such environmental fluctuations from the user? Should we design applications to adapt to changing conditions? Should we store data in a variety of formations, each suited for a specific condition? We need to discuss these questions and more with students.

Students should also investigate new applications that provide information to a user on the basis on the user's location. Important issues in this regard are location-based filtering, security, and privacy.

A variety of tools exist for students to conduct lab experiments and try various options (see the sidebar, "Resources for Course Projects").

Because mobile computing is a fast-changing research field with tremendous commercial potential, we must properly train students in this field. Toward that end, I've tried to provide a set of issues that a mobile computing course should cover. However, due to the field's continuous evolution, this set is bound to become obsolete. So, the course will need to supplement the basic materials with research papers from various conferences (such as MobiCom, SIGCOMM,
INFOCOM, the International Communications Conference, Milcom, and MobiHoc) and journals (such as IEEE/ACM Transactions on Networking, the IEEE Journal on Selected Areas of Communications, IEEE Transactions on Vehicular Technology, ACM/Baltzer Mobile Networks, and ACM/Baltzer Wireless Networks).

**Introduction to Parallel Computing**

**Abstract**

This presentation covers the basics of parallel computing. Beginning with a brief overview and some concepts and terminology associated with parallel computing, the topics of parallel memory architectures and programming models are then explored. These topics are followed by a discussion on a number of issues related to designing parallel programs. The last portion of the presentation is spent examining how to parallelize several different types of serial programs.

Level/Prerequisites: None

**Overview**

**What is Parallel Computing?**

- Traditionally, software has been written for *serial* computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.
In the simplest sense, **parallel computing** is the simultaneous use of multiple compute resources to solve a computational problem.

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

The compute resources can include:

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.
The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources than with a single compute resource.

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence. Some examples:
  - Planetary and galactic orbits
  - Weather and ocean patterns
  - Tectonic plate drift
  - Rush hour traffic in LA
  - Automobile assembly line
  - Daily operations within a business
  - Building a shopping mall
  - Ordering a hamburger at the drive through.

Traditionally, parallel computing has been considered to be "the high end of computing" and has been motivated by numerical simulations of complex systems and "Grand Challenge Problems" such as:
  - weather and climate
  - chemical and nuclear reactions
  - biological, human genome
  - geological, seismic activity
  - mechanical devices - from prosthetics to spacecraft
  - electronic circuits
  - manufacturing processes

Today, commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Example applications include:
  - parallel databases, data mining
  - oil exploration
  - web search engines, web based business services
  - computer-aided diagnosis in medicine
  - management of national and multi-national corporations
  - advanced graphics and virtual reality, particularly in the entertainment industry
  - networked video and multi-media technologies
  - collaborative work environments

Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.
Overview

Why Use Parallel Computing?

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever faster serial computers:
  - Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
  - Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
  - Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

- The future: during the past 10 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that **parallelism is the future of computing**.

Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of **Instruction** and **Data**. Each of these dimensions can have only one of two possible states: **Single** or **Multiple**.
The matrix below defines the 4 possible classifications according to Flynn.

Windows Programming is what Delphi is about, and to use it efficiently, you must know a little bit of Windows' "Base System Architecture". Flip through and absorb - You'll find Windows Programming a pretty interesting topic...

**What is Windows?**

Going back into history, you could say that Windows was just simply the result of envious PC owners who wished that their computer would be 'easier to use'. When Apple Computer released the Macintosh, (which some still hail as the greatest invention of all time) developers started working on a similar operating system for the PC. Through some marketing and lots of tension, Windows became the forerunner of the whole lot. Today, Windows has become a sort of standard for all PCs. Almost all new computers are pre-installed with Windows, and almost all Windows programs today are comparable, or even better, than their DOS counterparts. How about that for a success story?

**How do you program in Windows?**

Now that you know more about the history of Windows, we shall now talk about how Windows programming works. This chapter will make it easier for you to understand Delphi. (You are getting confused, right? Right?) You'll find out that Windows programming isn't that difficult - after you understand it of course.

First thing about Windows - it is a *multi-tasking* environment. That means that it can actually run more than one program at one time, giving you higher efficiency. This gave the creators of Windows a great headache. How do you implement an operating system where multiple programs can run at the same time? You'll need to process code for each program simultaneously, which is no easy feat. So, the makers of Windows created a system where each program was like a procedure - code was run only when the system requested for it. This meant that they needed to create a whole new system for Windows, containing the API (Application Program Interface). Code was in the form of 'events'. Only when an event occurs will code be run. Now you know why Delphi makes you type code that way!
But that's not all. Remember I told about something called the API? The API is actually just a whole lot of commands built into Windows to do what programmers need for Windows Programming. You may not know it, but Delphi gives you access to the API too! But don't worry if you look up the API to find millions of commands - Most are them are not used very often.

Another thing. Have you ever wondered how come people like to start type definitions with the letter "T"? Well, we call this Hungarian Notation and it's just a naming method to help programmers identify the data type of a variable quickly. We mention this because some of you may be getting the misconception that the 'T' is compulsory, while in actual fact the naming method comes from tradition.

**How Delphi implements Windows**

As you've already seen, Delphi implements Windows very closely to how Windows programming is supposed to work. Code is attached to events. API calls are allowed. This flexibility allows for an efficient Windows programming environment. But of course, all this code is nothing if there isn't anything to attach it to - So go on to the next course, where you'll learn how to use the Visual Component Library (VCL), the building blocks of a Delphi program.

- Combining these two types of problem decomposition is common and natural.

**Designing Parallel Programs**

**Communications**
Who Needs Communications?

The need for communications between tasks depends upon your problem:

- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - These types of problems are often called *embarrassingly parallel* because they are so straight-forward. Very little inter-task communication is required.

- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task’s data.

Factors to Consider:

There are a number of important factors to consider when designing your program’s inter-task communications:

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

- **Latency vs. Bandwidth**
- **Latency** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- **Bandwidth** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

**Visibility of Communications**
- With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
- With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

**Synchronous vs. Asynchronous Communications**
- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
- Synchronous communications are often referred to as **blocking** communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communications are often referred to as **non-blocking** communications since other work can be done while the communications are taking place.
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications.
Welcome to E Technologies

We are a full service IT solutions firm based in Palm Harbor, Florida. Put over 11 years of experience to work for you on your next project.

E Technologies is a website design, search engine optimisation, website marketing, graphic design and website hosting company, that specialises in helping businesses to utilise Internet based technologies to grow their businesses.

Incorporating a team of highly proficient website designers, search engine optimisation consultants, online marketing consultants, innovative graphic designers, website design architects and highly experienced business consultants help ensure all projects undertaken exceed client expectations and deliver that all important return on investment.

We find our clients have different sized enterprises at different stages of life. Some are new and emerging, others well established. Some have large budgets, whilst others operate with more of a 'shoe-string' approach. We're flexible - and we'll help you work out what needs to be incorporated to build your online business.

E Technologies offers a free initial consultation and quotation service so contact us now (email sales@etechusa.com), who knows, it could be the smartest thing you do today, and it will most certainly help you to realize the potential your business has online.

e-commerce

Often referred to as simply e-commerce, business that is conducted over the Internet using any of the applications that rely on the Internet, such as e-mail, instant messaging, shopping carts, Web services, UDDI, FTP, and EDI, among others. Electronic commerce can be between two businesses transmitting funds, goods, services and/or data or between a business and a customer.
A **program** or group of programs designed for **end users**. **Software** can be divided into two general classes: **systems software** and **applications software**. Systems software consists of low-level programs that interact with the **computer** at a very basic level. This includes operating systems, compilers, and utilities for managing computer resources.

In contrast, applications software (also called **end-user programs**) includes **database** programs, **word processors**, and **spreadsheets**. Figuratively speaking, applications software sits on top of systems software because it is unable to **run** without the operating system and system utilities.

Also see a diagram of **n-tier application architecture** in the **quick reference** section of Webopedia.

**EDI**
Short for **Electronic Data Interchange**, the transfer of data between different companies using **networks**, such as **VANs** or the **Internet**. As more and more companies get connected to the Internet, EDI is becoming increasingly important as an easy mechanism for companies to buy, sell, and trade information. **ANSI** has approved a set of EDI standards known as the **X12 standards**

**SOFTWARE AGENTS**

**INTRODUCTION**

The main goal of this paper is to overview the rapidly evolving area of software agents. The overuse of the word "agent" has tended to mask the fact that, in reality, there is a truly heterogeneous body of research being carried out under this banner. This paper places agents in context, defines them and then goes on, *inter alia*, to overview critically the rationales, hypotheses, goals, challenges and state-of-the-art demonstrators/prototypes of the various agent types currently under investigation. It also proceeds to overview some other general issues which pertain to all the classes of agents identified. Finally, it speculates as to the future of the agents research in the short, medium and long terms. This paper largely reviews software agents. Since we are overviewing a broad range of agent types in this paper, we do not provide a definition of agenthood at this juncture. We defer such issues until Section 4 where we present our typology of agents.

**Who are Investigating Software Agents for What and Why?**

We eschew answering this question in a futuristic sense in favour of providing a flavour of the scope of the research and development underway in universities and industrial organisations. The range of firms and universities actively pursuing agent technology is quite broad and the list is ever growing. It includes small non-household names (e.g. Icon, Edify and Verity), medium-size organisations (e.g. Carnegie Mellon University (CMU), General Magic, Massachusetts Institute of Technology (MIT), the University of London) and the real big multinationals (e.g. Alcatel, Apple, AT&T, BT, Daimler-Benz, DEC, HP, IBM, Lotus, Microsoft, Oracle, Sharp). Clearly, these companies are by no means completely homogeneous, particularly if others such as Reuters and Dow Jones are appended to this list.

The scope of the applications being investigated and/or developed is arguably more impressive: it really does range from the mundane (strictly speaking, not agent applications) to the moderately ūsmartí. Lotus, for example, will be providing a scripting language in their forthcoming version of Notes which would allow users to
write their own individual scripts in order to manage their e-mails, calendars, and set up meetings, etc. This is based on the view that most people do not really need smart agents. Towards the smart end of the spectrum are the likes of Sycarais (1995) visitor hosting system at CMU. In this system, "task-specific" and information-specifici agents cooperate in order to create and manage a visitoris schedule to CMU. To achieve this, first, the agents access other on-line information resources in order to determine the visitoris areas of interest, name and organisation and resolve the inevitable inconsistencies and ambiguities. More information is later garnered including the visitoris status in her organisation and projects she is working on. Second, using the information gathered on the visitor, they retrieve information (e.g. rank, telephone number and e-mail address) from personnel databases in order to determine appropriate attendees (i.e. faculty). Third, the visitor hosting agent selects an initial list of faculty to be contacted, composes messages which it dispatches to the calendar agents of these faculties, asking whether they are willing to meet this visitor and at what time. If the faculty does not have a calendar agent, an e-mail is composed and despatched. Fourth, the responses are collated. Fifth, the visitor hosting agent creates the schedule for the visitor which involves booking rooms for the various appointments with faculty members. Naturally, the system interacts with the human organiser and seeks her confirmation, refutations, suggestions and advice.

Most would agree that this demonstrator is pretty smart, but its smartness derives from the fact that the value gained from individual stand-alone agents coordinating their actions by working in cooperation, is greater than that gained from any individual agent. This is where agents really come into their element.

More examples of applications are described later but application domains in which agent solutions are being applied to or investigated include workflow management, network management, air-traffic control, business process re-engineering, data mining, information retrieval/management, electronic commerce, education, personal digital assistants (PDAs), e-mail, digital libraries, command and control, smart databases, scheduling/diary management, etc. Indeed, as Guilfoyle (1995) notes

"in 10 years time most new IT development will be affected, and many consumer products will contain embedded agent-based systems".

The potential of agent technology has been much hailed, e.g. a 1994 report of Ovumis, a UK-based market research company, is titled "Intelligent agents: the new revolution in software" (Ovum, 1994). The same firm has apparently predicted that the market sector totals for agent software and products for USA and Europe will be worth at least $3.9 billion by the year 2000 in contrast to an estimated 1995 figure of $476 million (computed from figures quoted in Guilfoyle, 1995). Such predictions are perhaps overly optimistic.

Moreover, as King (1995) notes telecommunications companies like BT and AT&T are working towards incorporating smart agents into their vast networks; entertainment, e.g. television, and retail firms would like to exploit agents to
capture our program viewing and buying patterns respectively; computer firms are building the software and hardware tools and interfaces which would harbour numerous agents; Reinhardt (1994) reports that IBM plans (or may have already done) to launch a system, the IBM Communications Systems (ICS), which would use agents to deliver messages to mobile users in the form they want it, be it fax, speech or text, depending on the equipment the user is carrying at the time, e.g. a PDA, a portable PC or a mobile phone. At BT Laboratories, we have also carried out some agent-related research on a similar idea where the message could be routed to the nearest local device, which may or may not belong to the intended recipient of the message. In this case, the recipientís agent negotiates with other agents for permission to use their facilities, and takes into consideration issues such as costs and bandwidth in such negotiations (Titmuss et al., 1996). At MIT, Pattie Maesí group is investigating agents that can match buyers to sellers or which can build coalitions of people with similar interests. They are also drawing from biological evolution theory to implement demonstrators in which some user only possesses the ëfittestí agents: agents would ëreproduceí and only the fittest of them will survive to serve their masters; the weaker ones would be purged.

It is important to note that most of these are still demonstrators only: converting them into real usable applications would provide even greater challenges, some of which have been anticipated but, currently, many are unforeseen. The essential message of this section is that agents are here to stay, not least because of their diversity, their wide range of applicability and the broad spectrum of companies investing in them. As we move further and further into the information age, any information-based organisation which does not invest in agent technology may be committing commercial hara-kiri.

### 4.1 A Typology of Agents

This section attempts to place existing agents into different agent classes, i.e. its goal is to investigate a typology of agents. A typology refers to the study of types of entities. There are several dimensions to classify existing software agents.

Firstly, agents may be classified by their mobility, i.e. by their ability to move around some network. This yields the classes of static or mobile agents.

Secondly, they may be classed as either deliberative or reactive. Deliberative agents derive from the deliberative thinking paradigm: the agents possess an internal symbolic, reasoning model and they engage in planning and negotiation in order to achieve coordination with other agents. Work on reactive agents originate from research carried out by Brooks (1986) and Agre & Chapman (1987). These agents on the contrary do not have any internal, symbolic models of their environment, and they act using a stimulus/response type of behaviour by responding to the present state of the environment in which they are embedded (Ferber, 1994). Indeed, Brooks has argued that intelligent behaviour can be realised without the sort of explicit, symbolic representations of traditional AI (Brooks, 1991b).
Thirdly, agents may be classified along several ideal and primary attributes which agents should exhibit. At BT Labs, we have identified a minimal list of three: autonomy, learning and cooperation. We appreciate that any such list is contentious, but it is no more or no less so than any other proposal. Hence, we are not claiming that this is a necessary or sufficient set. Autonomy refers to the principle that agents can operate on their own without the need for human guidance, even though this would sometimes be invaluable. Hence agents have individual internal states and goals, and they act in such a manner as to meet its goals on behalf of its user. A key element of their autonomy is their proactiveness, i.e. their ability to take the initiative rather than acting simply in response to their environment (Wooldridge & Jennings, 1995a). Cooperation with other agents is paramount: it is the raison d'être for having multiple agents in the first place in contrast to having just one. In order to cooperate, agents need to possess a social ability, i.e. the ability to interact with other agents and possibly humans via some communication language (Wooldridge & Jennings, 1995a). Having said this, it is possible for agents to coordinate their actions without cooperation (Nwana et al., 1996). Lastly, for agent systems to be truly 'smart', they would have to learn as they react and/or interact with their external environment. In our view, agents are (or should be) disembodied bits of 'intelligence'. Though, we will not attempt to define what intelligence is, we maintain that a key attribute of any intelligent being is its ability to learn. The learning may also take the form of increased performance over time. We use these three minimal characteristics in Figure 1 to derive four types of agents to include in our typology: collaborative agents, collaborative learning agents, interface agents and truly smart agents.

We emphasise that these distinctions are not definitive. For example, with collaborative agents, there is more emphasis on cooperation and autonomy than on learning; hence, we do not imply that collaborative agents never learn. Likewise, for interface agents, there is more emphasis on autonomy and learning than on cooperation. We do not consider anything else which lie outside the intersecting areas to be agents. For example, most expert systems are largely 'autonomous' but, typically, they do not cooperate or learn. Ideally, in our view, agents should do all three equally well, but this is the aspiration rather than the reality. Truly smart agents do not yet exist: indeed, as Maes (1995a) notes "current commercially available agents barely justify the name", yet alone the adjective 'intelligent'. Foner
(1993) is even more incandescent; though he wrote this in 1993, it still applies today:

"... I find little justification for most of the commercial offerings that call themselves agents. Most of them tend to excessively anthromorphize the software, and then conclude that it must be an agent because of that very anthromomorphization, while simultaneously failing to provide any sort of discourse or "social contract" between the user and the agent. Most are barely autonomous, unless a regularly-scheduled batch job counts. Many do not degrade gracefully, and therefore do not inspire enough trust to justify more than trivial delegation and its concomitant risks" (Foner, 1993, 39/40).

In effect, like Foner, we assert that the arguments for most commercial offerings being agents suffer from the logical fallacy of *petitio principii* - they assume what they are trying to prove - or they are circular arguments. Indeed, this applies to other "agents" in the literature.

In principle, by combining the two constructs so far (i.e. static/mobile and reactive/deliberative) in conjunction with the agent types identified (i.e. collaborative agents, interface agents, etc.), we could have *static deliberative collaborative agents*, *mobile reactive collaborative agents*, *static deliberative interface agents*, *mobile reactive interface agents*, etc. But these categories, though quite a mouthful, may also be necessary to further classify existing agents. For example, Lashkari et al. (1994) presented a paper at AAAI on "Collaborative interface agents" which, in our classification, translates to *static collaborative interface agents*.

Fourthly, agents may sometimes be classified by their roles (preferably, if the roles are major ones), e.g. world wide web (WWW) information agents. This category of agents usually exploits internet search engines such as WebCrawlers, Lycos and Spiders. Essentially, they help manage the vast amount of information in wide area networks like the internet. We refer to these class of agents in this paper as *information* or *internet agents*. Again, information agents may be static, mobile or deliberative. Clearly, it is also pointless making classes of other minor roles as in report agents, presentation agents, analysis and design agents, testing agents, packaging agents and help agents - or else, the list of classes will be large.

Fifthly, we have also included the category of *hybrid* agents which combine of two or more agent philosophies in a single agent.

There are other attributes of agents which we consider *secondary* to those already mentioned. For example, is an agent versatile (i.e. does it have many goals or does it engage in a variety of tasks)? Is an agent benevolent or non-helpful, antagonistic or altruistic? Does an agent lie knowingly or is it always truthful (this attribute is termed veracity)? Can you trust the agent enough to (risk) delegate tasks to it? Is it temporally continuous? Does it degrade gracefully in contrast to failing drastically at the boundaries? Perhaps unbelievably, some researchers are also attributing emotional attitudes to agents - do they get fed up being asked to do the same thing time and time again? What role does emotion have in constructing believable agents (Bates, 1994)? Some agents are also imbued with *mentalistic* attitudes or notions such as beliefs, desires and intentions - referred to typically as BDI agents.
(Rao & Georgeff, 1995). Such attributes as these provide for a stronger definition of agenthood.

In essence, agents exist in a truly multi-dimensional space, which is why we have not used a two or three-dimensional matrix to classify them - this would be incomplete and inaccurate. However, for the sake of clarity of understanding, we have collapsed this multi-dimensional space into a single list. In order to carry out such an audacious move, we have made use of our knowledge of the agents we know are currently out there and what we wish to aspire to. Therefore, the ensuing list is to some degree arbitrary, but we believe these types cover most of the agent types being investigated currently. We have left out collaborative learning agents, see Figure 1, on the grounds that we do not know of the existence of any such agents which collaborate and learn, but are not autonomous. Hence, we identify seven types of agents:

- Collaborative agents
- Interface agents
- Mobile agents
- Information/Internet agents
- Reactive agents
- Hybrid agents
- Smart Agents

There are some applications which combine agents from two or more of these categories, and we refer to these as heterogeneous agent systems. Such applications already exist even though they are relatively few. However, we also overview briefly such systems in the next section.

Another issue of note (for completeness sake) is that agents need not be benevolent to one another. It is quite possible that agents may be in competition with one another, or perhaps quite antagonistic towards each other. However, we view competitive agents as potential subclasses of all these types. That is, it is possible to have competitive collaborative-type agents, competitive interface agents, competitive information agents, etc.

The breakdown of the paper is as follows. Section 2 notes the situation of smart agents research in the broad field of Distributed Artificial Intelligence (DAI) and provides a brief history. Section 3 identifies the scope of applicability of agents research and notes that there is a diverse range of interested parties. Before the core critical overview of the agent typology of Section 5, Section 4 provides our view of what smart agents are; it also identifies the different types of agents which fall under the agents banner and warns that truly smart or intelligent agents do not yet exist! They are still very much the aspiration of agent researchers. Section 6 overviews some more general issues on agents and and speculates briefly towards
the future of agents in the short, medium and long terms. Section 7 concludes the paper.

1

BROADBAND TELECOMMUNICATIONS ARCHITECTURE
The Two-Way System Architecture
by Fred Slowik
Director of System Marketing
Transmissions Network Systems
General Instrument Corporation

Recent advancements in fiber optics technology have caused radical changes in cable Communications system network architectures. And with telcos entering the market, the cable industry is challenged to support a sophisticated interactive environment that includes video, internet, information, telephony options, and a host of interactive services. Planning system upgrades, rebuilds or new builds based solely upon forward bandwidth expansion is no longer enough. To meet the challenge of increasing competition, broadband network operators must ensure quality and reliability while expanding their services.

Operators must ask themselves if their systems are “ready” to deliver the exciting and eagerly anticipated new services that will be evolving over the next several years. The word “ready” can hold various meanings to different operators. Having new services is one thing. Having the capacity to deliver the new services reliably is quite another.

To build a network capable of reliably delivering expanded services requires operators to make an investment and a commitment to the future of cable. It requires operators to build an acceptable business model to justify the cost. A phased in approach over time is quite likely to manage costs.

As fiber optics usage for cable communications network applications became widespread, node sizes have decreased steadily. Since the inception of fiber optic deployment eight years ago, downward opto-electronics pricing trends have enabled cable operators to reduce node sizes from 10,000 homes passed or more to the current norm of 500 homes per node. Today, the most commonly accepted node size is 500 homes per node. This number is driven by a number of factors including economics, performance, and capacity.

What is the “right” node size for planning a network? Since service models and the products being deployed vary according to their bandwidth efficiency, there is not one specific answer to this question,. However, an architecture exists that permits migration to the smallest needed node size, solving the capacity problem.

Broadband Telecommunications Architecture (BTA), developed by General Instrument, is a system design platform that enables operators to cost-effectively migrate to smaller node sizes over time, as services and demands on the network evolve. But what forces drive this migration? Numerous services are emerging such
as network management, set-top polling, cable modems for internet access, telephony and interactive multimedia (IMTV). Operators must keep up-to-date with these services particularly in capacity planning, system integrity, and reliability. When planning node sizes, operators should consider available network bandwidth, homes per node, subscriber and service penetration rates, and bandwidth per service.

However, these elements may not produce completely accurate estimates, because various upstream service components and their different modulation schemes may alter the results. Also, levels of upstream ingress may affect the amount of useable network bandwidth from system to system or within the geographical areas of the system itself.

Timing of service implementation is another essential factor. Why invest more capital during initial system construction than the return on investment will justify? Conversely, investing too little could quickly render a system obsolete.

Chart 1 presents a cost per mile comparison with varying node sizes and densities. Included are opto-electronic and RF components, fiber, coaxial cable, and power supplies. Excluded are headend electronics, set-top units, and installation costs.

The initial node size may be as large or small as needed and utilize higher power lasers (optically split) to feed several nodes or lower power lasers for a point-to-point configuration. Note that the 500-home initial node is segmented, ideally into four smaller cells containing 125 homes each. This segmentation often depends upon system density and geography. Each segment then uses express (untapped) cable to the first active device in cascade and beyond if necessary.

This first active location has the potential to become a future node if necessary down the road. The operator installs the proper fiber counts to the initial 500-home node locations to provide for future migration. The usual count for initial and future nodes is a minimum of one downstream, one upstream, and one spare fiber. The operator has the option of installing three additional fibers between the existing node and the node locations either during the initial construction or deferring until the need arises.

Active cascades range from one to five devices beyond the future node site depending on system density. Service requirements and the operator’s business plan dictate a need to migrate further.

A system experiencing explosive subscriber growth most likely will need further migration.

**Data warehousing**

Data warehousing is combining data from multiple and usually varied sources into one comprehensive and easily manipulated database. Common accessing systems

31
of data warehousing include queries, analysis and reporting. Because data warehousing creates one database in the end, the number of sources can be anything you want it to be, provided that the system can handle the volume, of course. The final result, however, is homogeneous data, which can be more easily manipulated.

Data warehousing is commonly used by companies to analyze trends over time. In other words, companies may very well use data warehousing to view day-to-day operations, but its primary function is facilitating strategic planning resulting from long-term data overviews. From such overviews, business models, forecasts, and other reports and projections can be made. Routinely, because the data stored in data warehouses is intended to provide more overview-like reporting, the data is read-only. If you want to update the data stored via data warehousing, you'll need to build a new query when you're done.

This is not to say that data warehousing involves data that is never updated. On the contrary, the data stored in data warehouses is updated all the time. It's the reporting and the analysis that take more of a long-term view.

Data warehousing is not the be-all and end-all for storing all of a company's data. Rather, data warehousing is used to house the necessary data for specific analysis. More comprehensive data storage requires different capacities that are more static and less easily manipulated than those used for data warehousing.

Data warehousing is typically used by larger companies analyzing larger sets of data for enterprise purposes. Smaller companies wishing to analyze just one subject, for example, usually access data marts, which are much more specific and targeted in their storage and reporting. Data warehousing often includes smaller amounts of data grouped into data marts. In this way, a larger company might have at its disposal both data warehousing and data marts, allowing users to choose the source and functionality depending on current needs.

**Data Mining: What is Data Mining?**

**Overview**

Generally, data mining (sometimes called data or knowledge discovery) is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cut costs, or both. Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases.

**Continuous Innovation**
Although data mining is a relatively new term, the technology is not. Companies have used powerful computers to sift through volumes of supermarket scanner data and analyze market research reports for years. However, continuous innovations in computer processing power, disk storage, and statistical software are dramatically increasing the accuracy of analysis while driving down the cost.

Example

For example, one Midwest grocery chain used the data mining capacity of Oracle software to analyze local buying patterns. They discovered that when men bought diapers on Thursdays and Saturdays, they also tended to buy beer. Further analysis showed that these shoppers typically did their weekly grocery shopping on Saturdays. On Thursdays, however, they only bought a few items. The retailer concluded that they purchased the beer to have it available for the upcoming weekend. The grocery chain could use this newly discovered information in various ways to increase revenue. For example, they could move the beer display closer to the diaper display. And, they could make sure beer and diapers were sold at full price on Thursdays.

Data, Information, and Knowledge

Data

Data are any facts, numbers, or text that can be processed by a computer. Today, organizations are accumulating vast and growing amounts of data in different formats and different databases. This includes:

- operational or transactional data such as, sales, cost, inventory, payroll, and accounting
- nonoperational data, such as industry sales, forecast data, and macro economic data
- meta data - data about the data itself, such as logical database design or data dictionary definitions

Information

The patterns, associations, or relationships among all this data can provide information. For example, analysis of retail point of sale transaction data can yield information on which products are selling and when.

Knowledge

Information can be converted into knowledge about historical patterns and future trends. For example, summary information on retail supermarket sales can be analyzed in light of promotional efforts to provide knowledge of consumer buying
behavior. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts.

**Data Warehouses**

Dramatic advances in data capture, processing power, data transmission, and storage capabilities are enabling organizations to integrate their various databases into *data warehouses*. Data warehousing is defined as a process of centralized data management and retrieval. Data warehousing, like data mining, is a relatively new term although the concept itself has been around for years. Data warehousing represents an ideal vision of maintaining a central repository of all organizational data. Centralization of data is needed to maximize user access and analysis. Dramatic technological advances are making this vision a reality for many companies. And, equally dramatic advances in data analysis software are allowing users to access this data freely. The data analysis software is what supports data mining.

**What can data mining do?**

Data mining is primarily used today by companies with a strong consumer focus - retail, financial, communication, and marketing organizations. It enables these companies to determine relationships among "internal" factors such as price, product positioning, or staff skills, and "external" factors such as economic indicators, competition, and customer demographics. And, it enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to "drill down" into summary information to view detail transactional data.

With data mining, a retailer could use point-of-sale records of customer purchases to send targeted promotions based on an individual's purchase history. By mining demographic data from comment or warranty cards, the retailer could develop products and promotions to appeal to specific customer segments.

For example, Blockbuster Entertainment mines its video rental history database to recommend rentals to individual customers. American Express can suggest products to its cardholders based on analysis of their monthly expenditures.

WalMart is pioneering massive data mining to transform its supplier relationships. WalMart captures point-of-sale transactions from over 2,900 stores in 6 countries and continuously transmits this data to its massive 7.5 terabyte *Teradata* data warehouse. WalMart allows more than 3,500 suppliers, to access data on their products and perform data analyses. These suppliers use this data to identify customer buying patterns at the store display level. They use this information to manage local store inventory and identify new merchandising opportunities. In 1995, WalMart computers processed over 1 million complex data queries.
The National Basketball Association (NBA) is exploring a data mining application that can be used in conjunction with image recordings of basketball games. The Advanced Scout software analyzes the movements of players to help coaches orchestrate plays and strategies. For example, an analysis of the play-by-play sheet of the game played between the New York Knicks and the Cleveland Cavaliers on January 6, 1995 reveals that when Mark Price played the Guard position, John Williams attempted four jump shots and made each one! Advanced Scout not only finds this pattern, but explains that it is interesting because it differs considerably from the average shooting percentage of 49.30% for the Cavaliers during that game.

By using the NBA universal clock, a coach can automatically bring up the video clips showing each of the jump shots attempted by Williams with Price on the floor, without needing to comb through hours of video footage. Those clips show a very successful pick-and-roll play in which Price draws the Knick’s defense and then finds Williams for an open jump shot.

**How does data mining work?**

While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, and neural networks. Generally, any of four types of relationships are sought:

- **Classes**: Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.

- **Clusters**: Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.

- **Associations**: Data can be mined to identify associations. The beer-diaper example is an example of associative mining.

- **Sequential patterns**: Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer’s purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

- Extract, transform, and load transaction data onto the data warehouse system.
- Store and manage the data in a multidimensional database system.
- Provide data access to business analysts and information technology professionals.
- Analyze the data by application software.
- Present the data in a useful format, such as a graph or table.

Different levels of analysis are available:

- **Artificial neural networks**: Non-linear predictive models that learn through training and resemble biological neural networks in structure.
- **Genetic algorithms**: Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.
- **Decision trees**: Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID). CART and CHAID are decision tree techniques used for classification of a dataset. They provide a set of rules that you can apply to a new (unclassified) dataset to predict which records will have a given outcome. CART segments a dataset by creating 2-way splits while CHAID segments using chi square tests to create multi-way splits. CART typically requires less data preparation than CHAID.
- **Nearest neighbor method**: A technique that classifies each record in a dataset based on a combination of the classes of the \( k \) record(s) most similar to it in a historical dataset (where \( k \geq 1 \)). Sometimes called the \( k \)-nearest neighbor technique.
- **Rule induction**: The extraction of useful if-then rules from data based on statistical significance.
- **Data visualization**: The visual interpretation of complex relationships in multidimensional data. Graphics tools are used to illustrate data relationships.

**What technological infrastructure is required?**

Today, data mining applications are available on all size systems for mainframe, client/server, and PC platforms. System prices range from several thousand dollars for the smallest applications up to $1 million a terabyte for the largest. Enterprise-
wide applications generally range in size from 10 gigabytes to over 11 terabytes. **NCR** has the capacity to deliver applications exceeding 100 terabytes. There are two critical technological drivers:

- **Size of the database**: the more data being processed and maintained, the more powerful the system required.

- **Query complexity**: the more complex the queries and the greater the number of queries being processed, the more powerful the system required.

Relational database storage and management technology is adequate for many data mining applications less than 50 gigabytes. However, this infrastructure needs to be significantly enhanced to support larger applications. Some vendors have added extensive indexing capabilities to improve query performance. Others use new hardware architectures such as Massively Parallel Processors (MPP) to achieve order-of-magnitude improvements in query time. For example, MPP systems from NCR link hundreds of high-speed Pentium processors to achieve performance levels exceeding those of the largest supercomputers.

**Introduction To Windows Programming**

**Introduction**

Believe it or not, many of the coders that I’ve talked to over the years have avoided Windows programming -- not because they think dos is superior or even because they dislike Windows; but simply because they’re afraid of Windows code and the learning curve. They take a few peeks at sample code and see lots of scary words and constants that lead them to believe they’ll have to dedicate enormous amounts of time to switch from dos to windows -- which they don’t really want to do. So what’s the story? Windows programming is not that difficult once you understand a few key concepts. Sure the code can be intimidating when you first take a look at it, but it makes sense (in a twisted way :) when you understand what’s going on and it can actually be quite elegant. This document sets aside things like MFC, OWL, and other such facilitators. We'll be looking at bare Win32 code and examining how it works and why. Be forewarned: This is part of a series that eventually specializes in graphic/game programming techniques -- **not** general application software. At first the difference may not be visible, but throughout the series I’ll purposefully omit certain concepts and practices in the name of speed or neccessity for what we’re working on. Also, the sample code included with this tutorial was tested and compiled using Microsoft’s excellent Visual C++ 6.0 compiler. Just a personal preference and one of the most common. You shouldn’t have too many problems compiling the code with other compilers (hopefully). The code provided for this tutorial is in C for clarity, but the documents to come will most likely be in C++.
Event-Driven Programming

Most likely you have heard the term 'event-driven' before, but what does that have to do with Windows programming? Much. If you're used to dos coding (or similar system code), you're probably accustomed to having programs execute lines of code, one by one, in order from start to finish via function calls and such until the program is done or accomplishes some task. This is not how Windows programming works. Windows programming is typically based on 'events'. What kind of events? An event can be anything such as a mouse click or a button click on a window. So if a user clicks a button, how does our program know to execute the code for a button click? For our purposes, imagine a sort of omnipresent sentinel that sits around and waits for these events then converts them into 'messages' that our program can deal with. Each window is sent messages based on events and such, which are then accordingly processed. So if a button is clicked, the parent window is sent a message (via a function call) that tells it what has happened. How does a window receive a message? Each window has a 'window procedure' that handles all of the messages sent to it. We'll see more on that and how to react to messages in a bit. First we need to create a window.

Application Initialization

The sample application for this source code is a very simple program that basically just pops up a window and provides feedback as to what messages were processed when certain events occur. You should download and run the program to see how it works and what this tutorial is working towards showing you. Before we can actually create a window, there are a few steps that we must take, but even before that -- let's take a quick look at WinMain.

```c
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrev, LPSTR lpCmdLine, int Cmd) {

    MSG msg;

    ... // other initialization code;

    while(GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage (&msg);
    }

    return msg.wParam;
}
```
WinMain is called by the system when our program is first run. It's the starting point of our application. The first parameter is the handle to the instance of our application. The second parameter will always be NULL for our purposes. The third parameter is a string pointer to the command line. The last parameter specifies a show state (there are several values that start with "SW_" that you can look up in the help files). We will look at the other initialization code in a moment, but take a look at the while block. **GetMessage** gets a message from our thread's message queue and puts it into our **MSG** variable. The only time GetMessage returns 0 is when it receives a WM_QUIT message. You can use **GetLastError** to get error information if GetMessage returns an error (-1). Otherwise its return will be non-zero. The GetMessage parameters provided say that we want to retrieve any type of message available. The message is received into the **msg** variable which is then used by **TranslateMessage** to translate special keyboard messages into messages that are sent to the thread's message queue (you don't really need to worry about what this means right now). **DispatchMessage** then sends the message to the proper window procedure. Our window procedure is a special function that will process all of our messages. Note: there is a function called **PeekMessage** you can look up on your own time. Its quite useful because it returns (boolean) immediately instead of waiting around for a message like GetMessage does. It has its advantages, but I won't discuss them here. Now lets see how we go about actually creating our window.

First we need to register the window class. This would happen somewhere in the "other initialization code" section I left out of the WinMain above. Our window class can be registered via code such as:

```c
// Setup and register the window class;

WNDCLASS wClass;
    wClass.style         =CS_HREDRAW|CS_VREDRAW;
    wClass.lpfnWndProc   =WindowProcedure;        // callback function;
    wClass.cbClsExtra    =0;
    wClass.cbWndExtra    =0;
    wClass.hInstance     =hInstance;
    wClass.hIcon         =LoadIcon(hInstance,IDI_APPLICATION);
    wClass.hCursor       =LoadCursor(NULL,IDC_ARROW);
    wClass.hbrBackground =(HBRUSH)(COLOR_WINDOW+3);
    wClass.lpszMenuName  =NULL;
    wClass.lpszClassName =WINDOWNAME;

    RegisterClass(&wClass);
```

The **WNDCLASS** structure contains window class attributes for the window we plan on creating. For a complete description and listing of all the available constants,
read the help files that came with your compiler. (Word to the wise: Throughout your days of Windows coding, you will most likely be dealing with MANY constants, api calls, and things of this nature. Be sure to familiarize yourself with the help files. They're extremely useful when you're learning the way things work and as a general reference.) I'll briefly try to explain what each member means:

**style**
Specifies the class style. CS_HREDRAW and CS_VREDRAW mean that our window will redraw whenever our window is moved or resized horizontally or vertically.

**lpfnWndProc**
Specifies our window's 'window procedure' which handles the messages that are sent to it. We'll see more on that later.

**cbClsExtra**
Amount of extra space (in bytes) to allocate for the class structure.

**cbWndExtra**
Amount of extra space (in bytes) to allocate for the window structure.

**hInstance**
Instance handle.

**hIcon**
Handle to an icon (use a resource handle)

**hCursor**
Handle to a cursor (use a resource handle)

**hbrBackground**
Handle to a brush to use as the class background. There are several "COLOR_" HBRUSH values you can look up in the help files.

**lpszMenuName**
Pointer to a string that holds the menu name (from a resource). We aren't using a menu for this example program.

**lpszClassName**
Pointer to a string that specifies our class name. WINDOWNAME is #defined elsewhere in the program as "Simple Sample Application".

After setting up our WNDCLASS structure, we register our class by using **RegisterClass** which registers our window class so we can use **CreateWindow** to actually create our window such as the following code does.

```c
// Create the window and store the handle;

hWnd = CreateWindow(WINDOWNAME,                    // class name;
                    WINDOWNAME,                    // window name;
                    WS_OVERLAPPEDWINDOW,           // window style;
                    CW_USEDEFAULT, CW_USEDEFAULT,  // starting position (x,y);
                    320, 240,                      // width and height;
                    NULL,                          // parent handle;
                    NULL,                          // menu handle;
                    hInstance,                     // instance handle;
                    NULL);                         // other parameters;

// Check if window creation failed; otherwise show and update;

if(hWnd==NULL) return FALSE;
else {
```

40
ShowWindow (hWnd, Cmd);
UpdateWindow(hWnd);
}

The parameters should be fairly self explanatory (again, all the values for things such as 'window style' can be located in the help files), but I'll quickly explain them. The first parameter is the class name that we registered earlier. The second parameter is the window name -- which will show up in our window caption. The next parameter is the window style, which in our case is WS_OVERLAPPEDWINDOW. There are many combinations of styles you can use, but you'll have to look up all the values in the help file because there are too many for me to list here. The next two parameters specify the starting position of the window (horizontally and vertically). CW_USEDEFAULT just means to use the default position. The next two parameters specify the window's width and height respectively. The next parameter is the handle to a parent window. If our window was a child or owned window, we'd set this parameter to the parent window handle. Next up is a menu handle, but we're not using a menu so it's NULL. The next-to-last parameter is our instance handle and the last parameter is a pointer to other data to be sent to the window. After creating the window, you should usually check if the returned handle is valid then call ShowWindow and UpdateWindow which will set our window's show state and then tell it to repaint itself.

Now we would have a functioning window on our screen as shown below:

But don’t think we’re done just yet. We still need to write the window procedure which is how we make our seemingly useless and pointless window respond to some events.
Processing Messages

As I mentioned before, we need to look at how the window procedure for our function works. Here is an example:

```c
LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch(msg) {
        case WM_MOVE:
            MessageBox(hWnd, "WM_MOVE: The Window Moved", WINDOWNAME, MB_OK);
            return 0;
        case WM_DESTROY:
            MessageBox(hWnd, "WM_DESTROY: Exiting Application", "Goodbye!", MB_OK);
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

The code above is a little strange and doesn't do very much, but hopefully it will illustrate my point. The convention is to use a large switch/case setup to handle messages that we want to process explicitly -- then return `DefWindowProc` to take care of any messages that we don't. The "WM_" values above are 'window messages' that are simply constants that refer to events. For example as I mentioned earlier, a "WM_QUIT" message would return 0 to GetMessage (thus ending our message loop and exiting). If we wanted to process the WM_QUIT message ourselves, we'd add to the `switch` above, "case WM_QUIT:" and the corresponding code. In the example program I only added two messages to process, the "WM_MOVE" message which occurs after the window has been moved, and the "WM_DESTROY" which occurs when a window is being destroyed. You can add as many other window messages as you'd like such as "WM_CHAR" and "WM_SIZE" that all occur based on different events. Hopefully you should be seeing how things are coming together now. Events (like moving our window with the mouse) occur which are then translated (in the message loop) into messages. The messages are sent to our window and we do specific things based on what message it is. Any messages we don't necessarily
Closing

Well, I hope you learned something today, but obviously don’t stop here. The whole point of this tutorial is to get you started so you can figure things out for yourself and gain experience on your own. One of the most important things I’d like to emphasize is that I’ve repeated "look at the help files" numerous times throughout this document. If you’re new to Windows programming, without them you’ll probably have a very difficult time with many of the Windows-specific API calls, parameters, messages, and constants that you have to deal with. People buy tons of books, download terabytes of source code, flood newsgroups with questions, and all sorts of other things to try to find answers that are probably sitting right there in the help files or manuals that came with the compiler, sdk, library, etc. Try to figure things out for yourself and actually understand what’s going on. We’ve all heard RTFM, so do it.

ADVANCED WINDOWS PROGRAMMING CONCEPTS

Windows Programming is what Delphi is about, and to use it efficiently, you must know a little bit of Windows' "Base System Architecture". Flip through and absorb - You’ll find Windows Programming a pretty interesting topic...

What is Windows?

Going back into history, you could say that Windows was just simply the result of envious PC owners who wished that their computer would be ‘easier to use’. When Apple Computer released the Macintosh, (which some still hail as the greatest invention of all time) developers started working on a similar operating system for the PC. Through some marketing and lots of tension, Windows became the forerunner of the whole lot. Today, Windows has become a sort of standard for all PCs. Almost all new computers are pre-installed with Windows, and almost all Windows programs today are comparable, or even better, than their DOS counterparts. How about that for a success story?

How do you program in Windows?

Now that you know more about the history of Windows, we shall now talk about how Windows programming works. This chapter will make it easier for you to understand Delphi. (You are getting confused, right? Right?) You’ll find out that Windows programming isn’t that difficult - after you understand it of course.

First thing about Windows - it is a multi-tasking environment. That means that it can actually run more than one program at one time, giving you higher efficiency. This gave the creators of Windows a great headache. How do you implement an operating system where multiple programs can run at the same time? You’ll need to
process code for each program simultaneously, which is no easy feat. So, the makers of Windows created a system where each program was like a procedure - code was run only when the system requested for it. This meant that they needed to create a whole new system for Windows, containing the API (Application Program Interface). Code was in the form of ‘events’. Only when an event occurs will code be run. Now you know why Delphi makes you type code that way!

But that’s not all. Remember I told about something called the API? The API is actually just a whole lot of commands built into Windows to do what programmers need for Windows Programming. You may not know it, but Delphi gives you access to the API too! But don’t worry if you look up the API to find millions of commands - Most are them are not used very often.

Another thing. Have you ever wondered how come people like to start type definitions with the letter 'T'? Well, we call this Hungarian Notation and it’s just a naming method to help programmers identify the data type of a variable quickly. We mention this because some of you may be getting the misconception that the 'T' is compulsory, while in actual fact the naming method comes from tradition.

**How Delphi implements Windows**

As you’ve already seen, Delphi implements Windows very closely to how Windows programming is supposed to work. Code is attached to events. API calls are allowed. This flexibility allows for an efficient Windows programming environment. But of course, all this code is nothing if there isn’t anything to attach it to - So go on to the next course, where you'll learn how to use the Visual Component Library (VCL), the building blocks of a Delphi program.

**THEORY OF COMPUTATION**

What is formal language?

**Definition**

Formal language is language use characterized by

- speech before a passive audience
- the assumption of a role by the speaker
- the use of artificial means of communication such as writing or electronics, and
- the use of a “high” dialect or language in preference to a “low” one.

**Examples (English)**

Here are some examples of domains where formal language takes place:

- A sermon
NEED FOR FORMAL COMPUTATIONAL MODELS

ABSTRACT

This research examines the use of formal computational models to design intelligent device interfaces able to predict the function or application a user will use and automatically invoke this function. Computational models considered are Markov Chains, Markov Decision Process, Hidden Markov Model, Fuzzy Logic, and Bayesian Networks. Descriptive statistics obtained examine patterns of use on a mobile device. Usage profiles are modeled using the various computational models and verified with historical nonlearning data. This computational model framework is then deployed in field/laboratory trials where predictions are made on what the next user operation will be. Probabilistic predictions made by Markov Chains and Markov Decision Process are compared with context classification predictions made by Hidden Markov Model, Fuzzy Logic and Bayesian Networks. During the field/laboratory trials, each instance of use is a learning trial and the probabilities are recalculated for the models. In the next experimental phase, model predictions result in the automated delivery of certain functions or user operations. The benefits of automation are assessed in terms of task performance data (ease, accuracy, and speed of task completion) and user perceived usability using a questionnaire. The costs of automation are also assessed in terms of the costs of choosing a different operation and reversing the automated operation. In the last experimental phase, the ability to "undo" an automated operation is given and delivery of the next most highly probably operation is given as well as the capacity to "undo" each time or to come out of the automated prompting altogether. The benefits of this type of automation are assessed in terms of user performance and perceived usability. Future directions for research discuss how formal models can be used to design intelligent, highly-automated device interfaces and how best to design the automation to work in the users best interests.

EQUIVALENCE OF DFA AND NFA

Conversion of NFA to DFA

\[ L = (0 + 1)^*01 \]
Nondeterministic Finite Automata (NFA)

- play central role in language theory
- for some models, deterministic & nondeterministic versions are equivalent in power. For others, they are not. For a few, this question is deep (eg. P \(\neq\) NP).
Objectives
· To check the equivalence of DFA & NFA

Introduction

Two acceptors are equivalent if they accept the same language. A DFA is just a special case of an NFA that happens not to have any null transitions or multiple transitions on the same symbol. So DFAs are not more powerful than NFAs.

For any NFA, we can construct an equivalent DFA (see below).

So NFAs are not more powerful than DFAs. DFAs and NFAs define the same class of languages - the regular languages. To translate an NFA into a DFA, the trick is to label each state in the DFA with a set of states from the NFA. Each state in the DFA summarizes all the states that the NFA might be in. If the NFA contains \(|Q|\) states, the resultant DFA could contain as many as \(|2^{|Q|}|\) states. (Usually far fewer states will be needed.)

**From NFA to DFA**

Consider the following NFA:

What states can we be in (in the NFA) before reading any input? Obviously, the start state, A. But there is a transition from A to B, so we could also be in state B. For the DFA, we construct the composite state \(\{A, B\}\).

State \(\{A, B\}\) lacks a transition for x. From A, x takes us to A (in the NFA), and the null transition might take us to B; from B, x takes us to B. So in the DFA, x takes us from \(\{A, B\}\) to \(\{A, B\}\).

State \(\{A, B\}\) also needs a transition for y. In the NFA, \(d(A,y)=C\) and \(d(B,y)=C\), so we need to add a state \(\{C\}\) and an arc y from \(\{A, B\}\) to \(\{C\}\). In the NFA, \(d(C,x)=A\), but then a null transition might or might not take us to B, so we need to add an arc x from \(\{C\}\) to \(\{A, B\}\).

Also, there are two arcs from C labeled y, going to states B and C. So in the DFA we need to add the state \(\{B, C\}\) and the arc y from \(\{C\}\) to this new state.

**EQUIVALENCE OF DFA & NFA**

In the NFA, \(d(B,x)=B\) and \(d(C,x)=A\) (and by a null transition we might get back to B), so we need an x arc from \(\{B, C\}\) to \(\{A, B\}\). \(d(B,y)=C\), while \(d(C,y)\) is either B or C, so we have an arc labeled y from \(\{B, C\}\) to \(\{B, C\}\).
We now have a transition from every state for every symbol in S. The only remaining chore is to mark all the final states. In the original NFA, B was a final state, so in the DFA, every state containing B is a final state.

In order to try and understand how lex builds a scanner, we will construct a formal model of the scanning process. This model should take characters one-by-one from the input, and identify them with particular patterns which match up with a regular expression.

The standard model used is the **finite-state automaton**. We can convert any definition involving regular expressions into an implementable finite automaton in two steps:

Regular expression NFA DFA

A **non-deterministic finite-state automaton** (NFA) consists of:

**THEORY OF COMPUTATION**

· A set of states, of which we distinguish:
· A unique start state, usually numbered “0”.

A set of final states

· A transition relation which, for any given state and symbol gives the (possibly empty) set of next states

We can represent a NFA diagrammatically using a (labelled, directed) graph, where states are represented by nodes (circles) and transitions are represented by edges (arrows).

The purpose of an NFA is to model the process of reading in characters until we have formed one of the words that we are looking for.

**How an NFA Operates**

· We begin in the start state (usually labelled 0) and read the first character on the input.

· Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state.

· There may be more than one possible transition, in which case we choose one at random.

· If at any stage there is an output transition labelled with the empty string, \( \epsilon \), we may take it without consuming any input.
· We keep going like this until we have no more input, or until we have reached one of the final states.

· If we are in a final state, with no input left, then we have succeeded in recognising a pattern.

· Otherwise we must backtrack to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

The key concept here is that: every NFA corresponds to a regular expression

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm.

**Exercises**

1. Construct DFAs equivalent to the NFAs in the fig
2. Convert the NFA to an equivalent DFA (Answer the "?")

| 0 | 1 F = { q2, q4 } |
|---+-----------------|
| q0 | {q0,q3} | {q0,q1} |
| q1 | phi | {q2} |
| q2 | {q2} | {q2} |
| q3 | {q4} | phi |
| q4 | {q4} | {q4} |

M = (Q, sigma, delta, q0, F)  
Q = { } You may simplify to 16 states, but no fewer.  
F = { }  
q0 = ?

**What is a PDA?**

A PDA (Personal Digital Assistant) is a computer that fits in your hand. These small computers are sometimes called palmtops and are a great way to store telephone numbers, email addresses, access the internet, make calculations, keep a digital calendar and play games.

Today's PDA is very small in size, fits comfortably in a pocket and generally has good battery life which allows you to recharge the device at night or when at the
office. A PDA usually includes a small screen usually bigger than a digital phone, however smaller than the smallest laptop, a small QWERTY keyboard that is made for thumb typing and a stylus which is a metal or plastic pen to input data or communicate with the device by a touch pad screen.

While components and specifications change throughout the years, today you can find PDA's with lots of RAM memory, storage in either miniature hard drives or compact flash cards or sticks and some can be expanded by a PC card that fits into a PCMCIA slot. Newer palmtop computers have USB ports to plug in a variety of peripherals to use with your PDA.

Today, you can find great features on many palmtop computers including internet access, the ability to send and receive email and files, mp3 player with onboard storage and game playing functions. Some palmtop computers have even morphed into an all in one device that includes a wireless phone.

A PDA is a great tool to stay organized. They are very popular with road warriors, corporate professionals, college students, etc. There are many different types of palmtop computers that range in price and features. For instance, you can purchase a palmtop computer that fulfills basic requirements such as telephone and address book, calendar, calculator and other travel functions for less than $100. Most of these low end palmtop computers also include infrared synching which means that you can send and receive data from your palmtop computer to a host of other computers.

A moderately priced PDA includes extra functions such as a color screen, large amounts of memory and storage capabilities, USB port and can play video games. If you are looking for a high end palmtop computer, you will find almost every available option loaded onto these super high tech devices including a wireless phone, mp3 player, the ability to handle large files, lots of memory, Bluetooth capability, etc. The top of the line palm top computers are usually priced at about $450 to $800.

**Deterministic pushdown automaton**

From Wikipedia, the free encyclopedia

Jump to: navigation, search

In automata theory, a **deterministic pushdown automaton** is a deterministic finite automaton with an additional stack of symbols; its transitions can take the top
symbol on the stack and depend on its value, and they can add a new top symbol to the stack.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafetaria, since the operations never work on elements other than the top element. A stack automaton, by contrast, does allow operations on other elements, and stack automata can recognize a strictly larger set of languages than pushdown automata.

A deterministic pushdown is effectively a particular type of pushdown automaton, namely ones that have at most one transition for the same combination of input symbol, state, and top stack symbol.

A deterministic context-free language is a language recognized by some deterministic pushdown automaton. Not all context-free languages are deterministic.\cite{1} This is unlike the situation for deterministic finite automata, which are also a subset of the nondeterministic finite automata but can recognize the same class of languages (as demonstrated by the subset construction).

- is a finite set of states
- is a finite set of input symbols
- is a finite set of stack symbols
- is the start state
- is the starting stack symbol
- , where \( A \) is the set of accepting states
- is a transition function, where

\( M \) is deterministic if it satisfies both the following conditions:

- For any \( s \), the set has at most one element.
- For any \( s' \) if \( s' \), then for every

There are two possible acceptance criteria: acceptance by empty stack and acceptance by final state. The two are not equivalent for the deterministic pushdown automaton (although they are for the non-deterministic pushdown automaton). The languages accepted by empty stack are the languages that are accepted by final state, as well as have no word in the language that is the prefix of another word in the language.