**CST8152 – Lecture Notes**

**Lecture 1**

- A **compiler** is a program that runs on a computer architecture under an OS and transforms (**translates**) an input program (**source program**) written in some programming language into an output program (**target program**) expressed in a different programming language.

- Programming language compilers are part of a more general category of language manipulation tools or programs, **language processors**.

* Language processors include natural language translators and interpreters, text editors, TTS and STT coverers, spell and grammar checkers and other language tools.

- Programming language interpreters are a subset of programming language compilers.

- A programming language is a notational system for describing computations in machine-readable and human-readable form.

- Computation in general is any process that can be carried out by a computer.

* Programming languages must provide two types of abstractions: **data abstractions** and **control abstractions**.

**Why Study Compilers**

- Compiler construction is a very specialized field of computer science and system programming.

**Compilers are Used by all Programmers**

- Knowledge about how compilers work can be used to write more efficient and error free code in a high-level language.

**Compiler Elements and Techniques are Used in Almost All Applications**

- Many programmers use compiler components for programming other applications.

- There’s a good chance that programmers will need to write a compiler or interpreter for a domain-specific language.

* Writing a parser for XML, HTML, or some other structured data file is a common task.

- Scanning and parsing a command or user input line is a very common task.

**Compilers Are Excellent Milestone Programming Project**

- Writing a compiler requires an understanding of almost all of the basic computer science subfields.

* These include regular expressions, grammars, finite automata theory, programming paradigms, operating systems, computer architecture, a large range of data structures and algorithms, some programming languages, and a good and sound software engineering principles.

**Important Concepts**

**Compiler Input**

- Source program

- Configuration parameters or pragmatics (#pragma directives)

- Source and Target language definitions

**Compiler Output**

- Target program

- Error messages

- Information accompanying the target program – external symbol tables, cross-reference tables.

**Target Program**

- High-level language

- Low-level code

**Target Low-Level Code Type**

- Pure machine code

- Augmented machine code

- Virtual machine code

**Target Low-Level Code Format**

- Assembly

- Pseudo-assembly

- Relocatable Binary Format

- Memory-image format (load and go)

**Run-Time Environment**

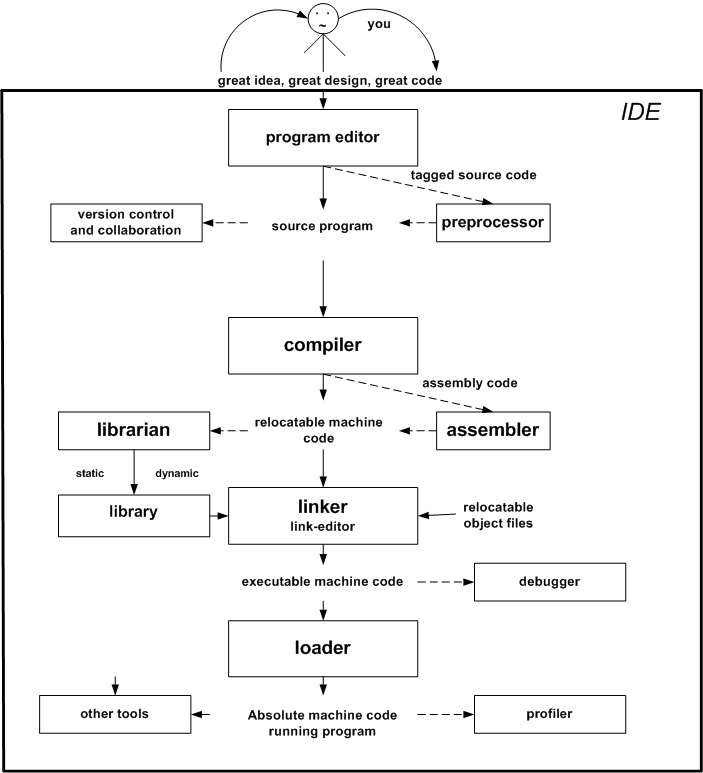
- Fully static environment

- Fully dynamic environment

- Mixed environment

- Stack-based environment

**Lecture 2 – The Context of a Compiler**



**Program Editor**: Allows the user to enter the text (code) of the program and to save it in a text form (ASCII or Unicode).

* Any text editor can be used as a program editor, but modern specialized program editors provide many additional features specific to the programming language.
* Modern program editors also incorporate some compiler elements like static syntax checking.

**Compiler**: Translates the text of the program into another language, usually assembler or some form of machine code.

**Assembler**: Assemblers are simple compilers which translate assembly language into machine code.

**Linker**: Combines all necessary components of a program into some executable form.

* Not all programming languages require linkers.

**Loader**: Loads an executable program and passes the control to the program.

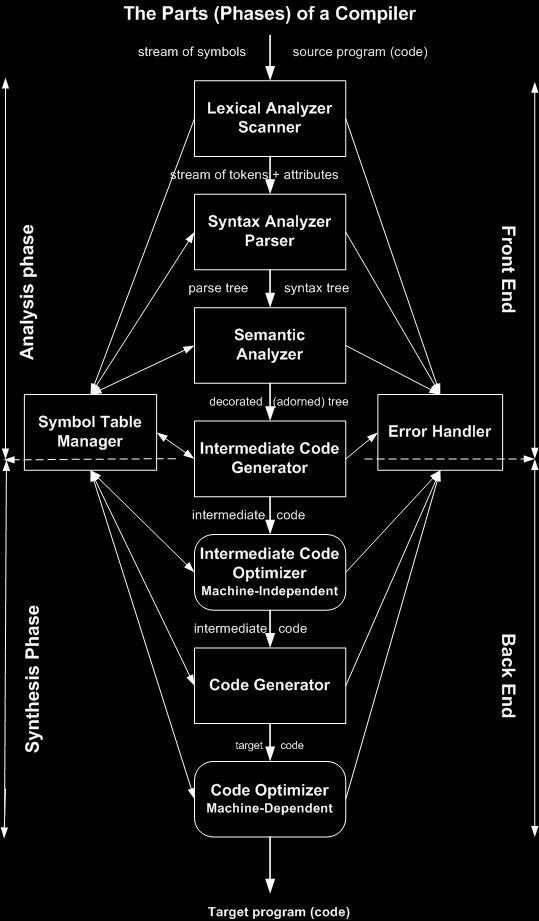
**Librarian**: Allows the creation and maintenance of libraries of pre-compiled components which can be used later without the need to be compiled again.

**Debugger**: Allows the user to trace the execution of a program statement by statement and inspect the content of different parts of the program memory.

**Other tools**

* Automatic or unit testers (e.g. JUnit)
* Version control and collaboration tools (e.g. Git)
* Code inspectors and analyzers.
* Profilers
* Run-time inspectors
* Error loggers
* Make and build scripting tools
* Refactoring tools
* Task managers (e.g. Mylen)
* Project managers (e.g. Maven)

**Lecture 4 – Components of a Compiler**



**Analysis**

- The **analysis phase** breaks up the source program into constituent pieces and imposes a grammatical structure on them.

- Uses this structure to create an intermediate representation of the source program.

- Often describes what is referred to as the **front end** of the compiler.

**Synthesis**

- The **synthesis phase** constructs the desired target program from the intermediate representation and the information in the symbol table.

- Often describes what is referred to as the **back end** of the compiler.

**Scanning**

- The first phase of a compiler is called **lexical analysis** or **scanning**.

- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**.

* For each lexeme, the lexical analyzer produces output tokens of the form (token-name (code), attribute-value).

**Parsing**

- The second phase of the compiler is **syntax analysis** or **parsing**.

- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation called a **parse tree**, which depicts the grammatical structure of the token stream.

* Typically, the parse tree is reduced to another form of representation called a **syntax tree** in which each interior node represents an operation and the children of the node represent the arguments making up the operation.

- The **semantic analyzer** uses the syntax tree and the information in the **symbol table** to check the source program for semantic consistency with the language definition.

* It also gathers type information and saves it in either the syntax tree or the symbol table, used for subsequent intermediate code generation.

**Intermediate Representation**

- In the process of translating a source program into target code, a compiler constructs one or more intermediate representations, which can have a variety of forms.

* e.g. Syntax trees are a form of intermediate representation.
  + They are consistently used during syntax and semantic analysis.

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level **machine-like intermediate representation.**

* One can think of this as a program for an abstract machine.
* This intermediate representation should have two important properties.
  + Should be easy to produce.
  + Should be easy to translate into the target code.

- This is typically a dividing point between the front end and back end responsibilities of a compiler.

**Code Optimization**

- The **machine-independent code optimization** phase attempts to improve the intermediate code to eventually produce better target code.

- There is a great variation in the amount of **code optimization** that different compilers perform.

* **Optimizing compilers** spend a significant amount of time on this task.
* Most compilers, however, can perform simple optimizations that will significantly improve the running time of the target program, without slowing down the compilation process by too much.

**Code Generation**

- The **code generator** takes an intermediate representation as input and maps it into the target language

- An essential compilation function is to record the variable names used in the source program and collect information about various attributes of each name.

* These attributes may provide information about the storage allocated for a name, its type, its scope, and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument, and the type returned.

- The **symbol table** is a data structure containing a record for each variable name, with fields for the attributes of the name.

**Error Handler**

- The **error handler** is responsible to report to the programmer the lexical, syntactical, and semantic errors discovered during the course of compilation.

- It is also responsible to prevent the compiler from producing a target code if an error has been detected, halting the whole process.

///

- The discussion of **phases** deals with the logical organization of a compiler.

- In an implementation, activities from several phases may be grouped together into a **pass** that reads an input file and writes an output file.

* e.g. The front end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
* Code optimization might be an optional pass.
* Could also be a back end pass consisting of code generation for a particular target machine.

**Lecture 5 – Tombstone Diagrams**

- **Tombstone diagrams** are a convenient graphical representation of software, hardware (machines), processing of software by machines, and manipulation of programs by other programs (compilers).

**Machines**

- Programs run on machines executing some machine code.

- A pentagon shaped tombstone represents a machine executing machine code.

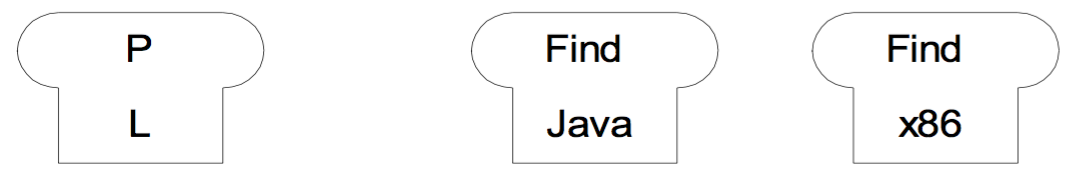
* e.g. x86 is a machine executing Intel machine code.



**Programs**

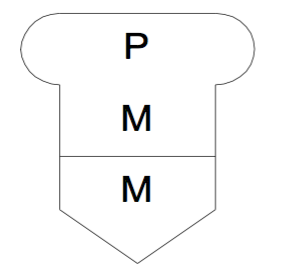
- Round-top (mushroom) tombstones represent a program **P** expressed in language **L**.

* e.g. A program called *Find*, written in Java and **expressed** in machine language.



- Programs always run on some machine.

* To express this, a program is put on top of a machine.
* **The language the program is expressed in and the machine language must be the same.**

****

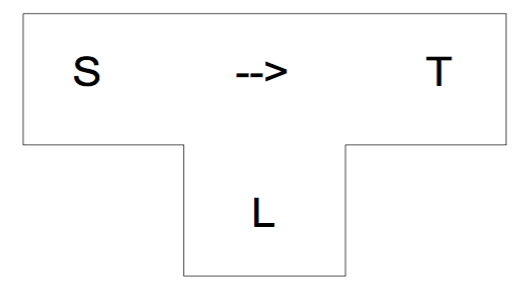
**Translators**

- T-shaped tombstones represent **translators** (compilers).

- The head of the tombstone shows the translator’s source language **S** and the target language **T**.

* The arrow represents the direction of the translation.

- The base of the tombstone shows the translator’s implementation language.

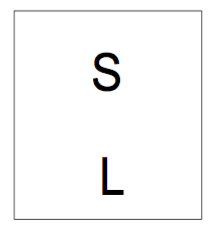


**Interpreters**

- An interpreter is a program that accepts a program written in some source language, and in most cases, runs it immediately without translating the entire source program into a target program.

- An interpreter is represented by a rectangular tombstone.

* The head of the stone indicates the interpreter’s source language **S**.
* The base of the tombstone shows the implementation language **L**.



**Compilation and Execution**

- When one develops programs with a compiled language, the compilation of the program and the execution of the compiled program are two separate and distinctive steps.

- First one needs a compiler, which translates language **S** into language **T** and it is expressed in language **M**.

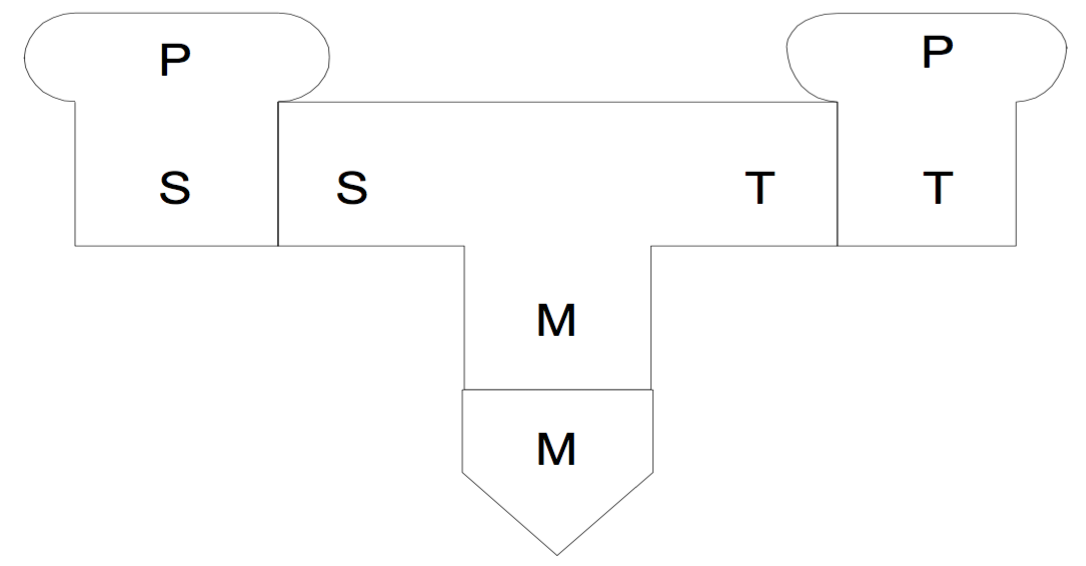
* Since the compiler is a program itself, it must run on some machine that understands language **M**.

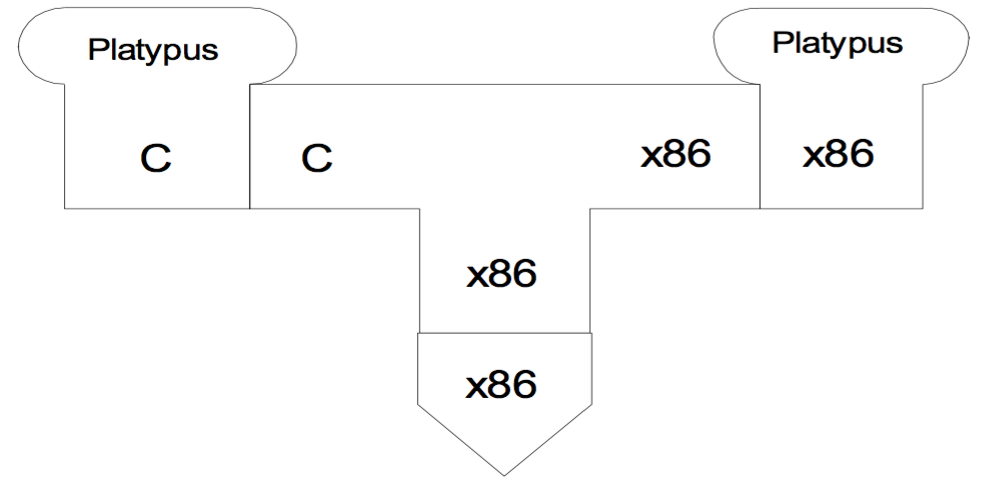
- The compiler translates the program **P** written in language **S** into the same program **P** but expressed in language **T**.

- The compiler runs on a machine that understands the language **M**.

- If **T** and **M** are different languages the compiler is called a **cross compiler**.

* A cross compiler is a compiler that runs on one machine (the host machine) but generates code for a different machine (target machine).
* e.g. A compiler that runs on a SPARC machine but generates code for an Intel machine.





- Once the program is compiled, it can be run on a machine that understands the same language that the program was expressed in.

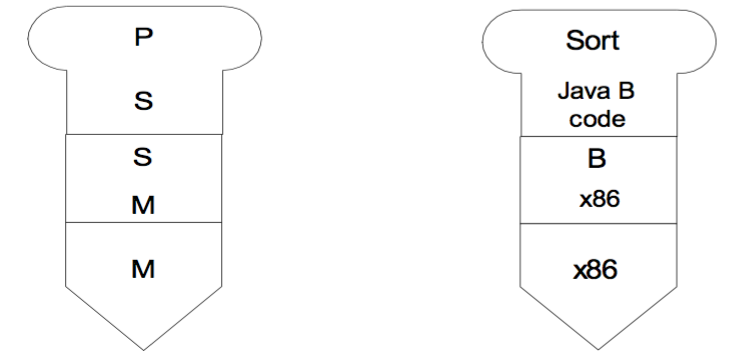
**Interpretation**

- Interpretation is a one-step process. The program is analyzed and executed one or several statements at a time.

- Interpreters do not produce target code.

- An **interpretive compiler** is a combination of compiler and interpreter.

* It translates the source program into an intermediate language used for some **virtual** **machine**.
* After that this code is run by an interpreter of the intermediate language.
* A prominent example of this is **Java**.



**Bootstrapping**

- One might ask themselves if it’s possible to write a compiler using the same language in which the compiler is supposed to compile from?

* It **is** possible, and the process of writing a compiler in this way is called **bootstrapping**.

- First, the compiler is written in in assembler or some other language that can compile a subset of the language.

* Then the compiler and the subset are used to write a compiler for the full language.

- This method is called a **full bootstrap** because the whole compiler is to be written from scratch.

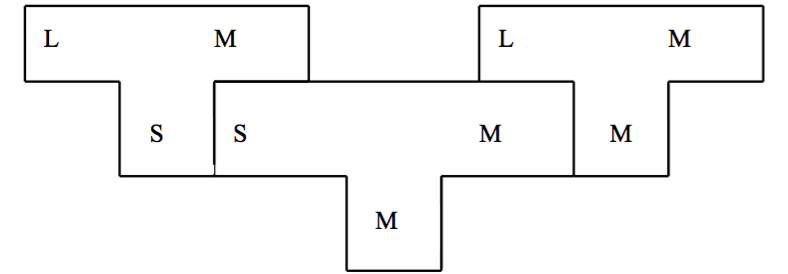
* This method can be applied one or several times for richer and richer subset of the language until a compiler for the full language is implemented.

- If a compiler for the language already exists but one wants to write a compiler for a different machine using the same language the compiler compiles, the process is named a **half bootstrap** since roughly have the compiler must be modified.

**Full Bootstrapping**

1. Write a compiler in machine code which compiles a subset S of a language L in machine code M to run on the same machine: SMM
2. Write a compiler for a fuller version L of the language using the subset S: -> LSM
3. Use SM­­M to translate LSM: -> LMM

LSM + SMM = [LMM]0



1. With the fuller version of the language, write a better compiler and use LMM to compile it. LLM + [LMM]0 = [LMM]1
2. And then do it again LLM + [LMM]1 = [LMM]2.

- The compiler must consistently compile itself.

**Lecture 6 – Grammars and Languages**

- Computer languages have much in common with other human languages.

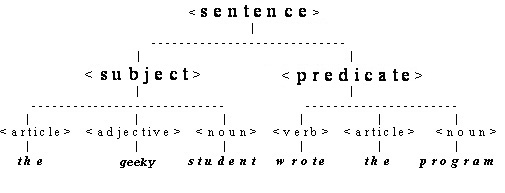
* Understanding their structure is the key to translating code written in them into an unambiguous set of instructions that can be accurately executed by a computing machine.

- The structure of a language is defined by its **grammar**.

**Grammars**

- One requires some well-defined notation allowing them to describe the syntactical rules of a language without any ambiguities.

- In terms of the grammar of the English language, this can be broken down into a **hierarchical parse tree**.



- The use of **meta-symbols** like <subject> enclosed in angle brackets denote the **syntactic entities**.

* These constitute a special language that describes a language.

- The parse tree suggests a form for the metalanguage as a set of rules for the correct ways the syntactic entities may be formed.

- The following metalanguage is that of Noam Chomsky in a notation proposed by John Backus.

* The rules for this grammar are:
  + <sentence> ::= <subject> <predicate>
  + <subject> ::= <article> <adjective> <noun>
  + <predicate> ::= <verb> <direct object>
  + <direct object> ::= <article> <noun>

- The metasymbol ::= means *hast the form of* or *may be composed of*.

- A single rule is called a **production** since what is on the LHS can produce a more detailed string on the RHS.

- The metasymbols <> are called **non-terminals** since they will be replaced by applying the production.

- The symbols in bold are called **terminals** since they terminate the syntax (they are the **leaf nodes** of the parse tree).

- The set of terminal symbols form the **sentence** that we finally recognize as being in the form of the target language.

**Languages**

- A **language** is the set of sentences that are generated from the grammar.

- Sentences can be syntactically correct but **semantically** incorrect.

**Scanning and Parsing (Lexical and Syntactic Analysis)**

- Given a string of characters, it is important to be able to determine whether it is a valid sentence according to the grammar. Two issues are at hand here:

1. **Lexical analysis** (the **scanner**) involves constructing the string of tokens (terminal symbols) from the string of characters.
2. **Syntactic analysis** (the **parser**) involves validating (recognizing) the token string against the grammar.

- This division is not completely precise, since parsing in its general meaning is also part of the scanning process.

**Lexical Analysis**

- In lexical analysis one attempts to collect the incoming characters into tokens, or terminal symbols of the grammar.

* “The geeky student wrote the program” generates the tokens ARTICLE, ADJECTIVE, VERB, NOUN (represented in the program code as symbolic constants, possibly with the values 0,1,2,3), with their lexical values the, geeky, wrote, student, program.

- When the input stream has been scanned one ends up with the symbol table containing the entries:

|  |  |
| --- | --- |
| **Token** | **Attribute – Lexeme** |
| ARTICLE | The |
| ADJECTIVE | geeky |
| NOUN | student |
| VERB | wrote |
| ARTICLE | the |
| NOUN | program |

- However, the task of the scanner is also to reject improper input strings such as “Thr gyeke stuent wnote te pgram” since no tokens can be identified.

- The task of the scanner in lexical analysis is to parse the input stream of characters, group them into lexemes and recognize them as tokens matching the patterns associated with the tokens.

- In doing this, one will use a simpler grammar for describing the lexemes called a **regular grammar** for which is equivalent to **regular expressions** notation for describing strings.

**Syntactic Analysis**

- The role of the parse is to confirm (recognize) that the sentence of tokens satisfies the language grammar.

* This is **parsing the token stream**.

- The syntactic analyzer (parser) must then confirm that it is valid grammatically by using the productions of the grammar to construct (usually implicitly) a syntax tree.

* If the tree fails, then the string of tokens is not a sentence of the language.

- This can be done by a **top-down** or **bottom-up** parsing technique.

**Top Down Parsing**

- In this one begins with the <sentence> nonterminal, called the **start symbol**, and by successive applications of the production rules, attempts to arrive at the sentence.

* At each stage, one token is consumed.

<sentence> -> <subject> <predicate>

<subject> -> <article> <adjective> <noun>

<predicate> -> <verb> <direct object>

<direct object> -> <article> <noun>

<article> -> **the**

<adjective> -> **geeky**

<verb> -> **wrote**

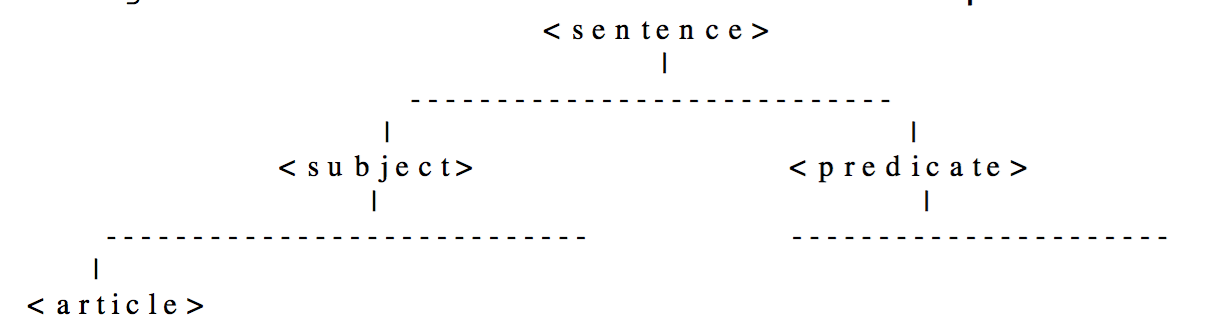
<noun> -> **student | program**

- Starting from the left of the token string (a **leftmost derivation**) one looks at the first token ARTICLE.

ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN

- One begins at the start nonterminal <sentence>. They find using rule 1 that a sentence starts with a <subject> that starts with an <article>.

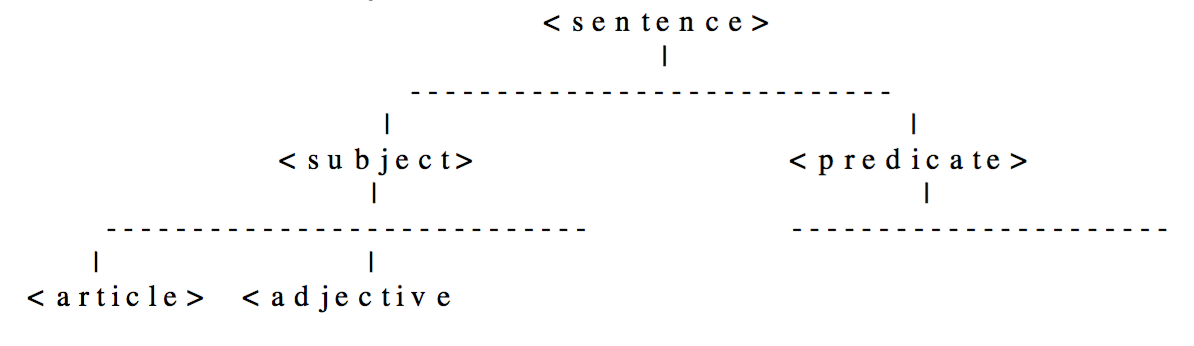
* The parse tree at this point looks like.



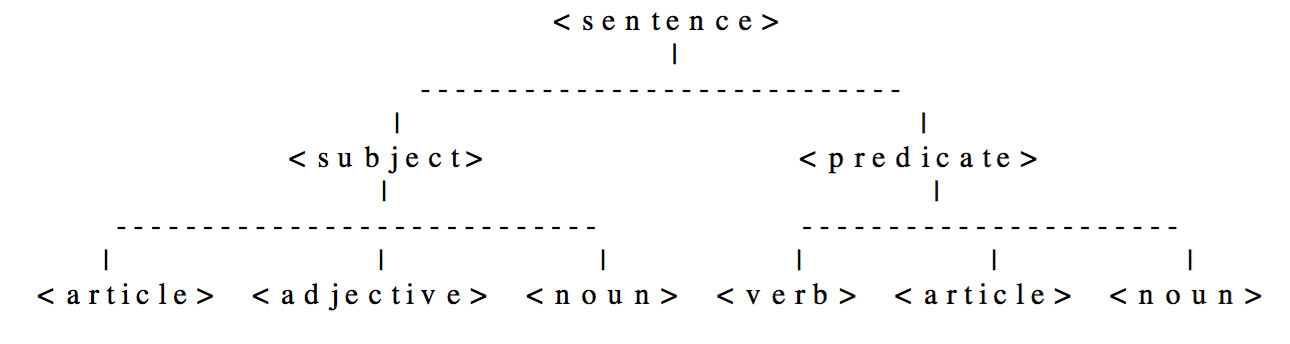
- Now that ARTICLE has been recognized, it is consumed, the token string is shorter and the scanner now attempts to recognize ADJECTIVE.

- From rule 2, a <subject> is an <article> followed by an <adjective>.

* So ADJECTIVE is consumed and the parse tree is as follows and <adjective> is consumed.



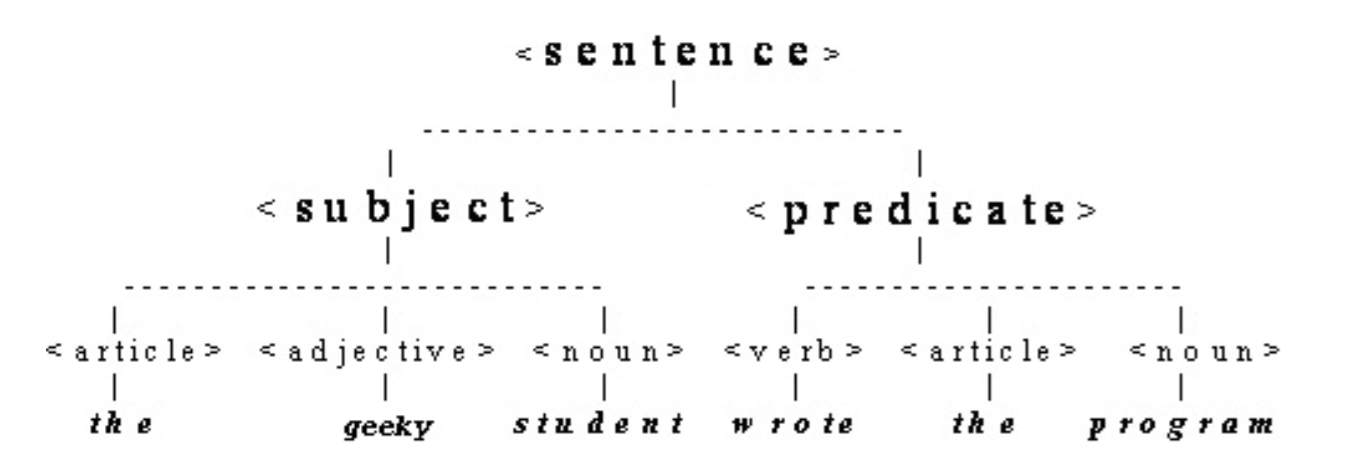
- The scanner continues in this way until all the input tokens are consumed and the parse tree builds the token string in order:



- The input token string has then been successfully parsed and is grammatically valid.

- Because the scanner is used to do prior lexical analysis, the actual lexemes do not appear in this tree.

* If the scanner was included in the complete process, the complete tree would have been



- Once one has defined the grammar for a sentence, they can define the grammar for a novel (program).

<novel> -> <sentences>

<sentences> -> <sentence> <sentences> | <sentence>

- Since the nonterminal <sentences> appears on both sides of the production, it means that a <sentence> is **self-describing**.

- The <sentences> nonterminal is described in terms of itself and is therefore **recursive**.

**Lecture 7 – Language Specification**

**Grammars**

- The Noam Chomsky grammar hierarchy consists of

1. Regular Grammars
2. Context Free Grammars (CFG or BNF)
3. Context Sensitive Grammars
4. General

- A **context free grammar** (**CFG**) is defined by the following four components

1. A finite set of terminal symbols (**terminals**) or a final terminal vocabulary **Vt**.
   1. For the lexical grammar the terminals are the alphabet.
   2. For the syntactic grammar the terminals are the token set produced by the scanner and defined by the lexical grammar.
2. A finite set of **nonterminals** or a nonterminal vocabulary **V­n**.
   1. Nonterminals are not part of the language. They are intermediate symbols used to define the grammar for the language.
3. A finite set of **productions** (rewriting or replacement or substitution or derivation rules) **P**.
   1. Productions have the form A -> X1X2X3­…Xm where A is an element of VnXi which is an element of Vn U Vt, 1 ≤ I ≤ m, m > 0 and A -> .
4. A **start** symbol **S**. The start symbol is an element of Vn­, which is always the root of the parse tree.

- Following the above definition, a CFG is made up from the 4-tuple **G = (Vt, Vn, P, S)**.

- L(G) is the language defined or generated by the grammar, G.

**Kleene’s Theorem**

- Stephen Kleene formulated and proved the following theorem.

- A language that can be defined by either regular expressions (or a regular grammar), transition graphs (transition or state diagrams), or a finite automaton (finite state machine) can be defined interchangeably by all three methods.

**Regular Expressions**

- Regular expressions are a convenient notation for specifying certain simple sets of strings over some alphabet.

- They can be used for describing lexical symbols or for specifying the structure of the token used in a programming language.

- A regular expression can be used to construct a **deterministic finite automaton** which therefore can recognize strings of the grammar, which is the purpose of the Scanner.

- A regular expression is a shorthand equivalent to a regular grammar.

* It is a pattern that strings must match or conform to, to be valid words (or sentences).

- The sets of strings defined by regular expressions are termed regular sets.

- As defined above, the grammar is the following 4-tuple: G = (Vt, Vn, P, S).

* If one denotes **r** as a regular expression, then since **G** and **r** are equivalent, they must produce the same languages, so L(G) = L(r).

- To define the strings, regular expressions (as any expression notation) use operands and operations.

* The operands are alphabet symbols or strings defined by regular expressions (regular definitions).
* The standard operations are concatenation, union or alteration (|), and Kleen closure (\*).

- Regular expressions use the metasymbols |, (, ), {, }, [, ], \*, + (along others like ?, ^) to define its operations.

**Alphabet**

- An alphabet is a finite, non-empty set of symbols.

- For the scanner, the alphabet may be characters in the ASCII character set.

- For the parser, the alphabet is the set of tokens produced by the scanner.

- The set of keywords is also an alphabet (even though by itself it is insufficient to form a language).

**String**

- A string is a finite set of symbols from an alphabet (not necessarily in a grammar).

- The order that characters appear to make up a string matters.

- The length of a string **s** is **| s |** and is equal to the number of symbols in the string.

**Symbol Character Regular Expressions**

**A Single Character**

- A regular expression can be a pattern for a single character from the alphabet.

- Consider an alphabet ∑ = {a,b,c,d}. If one writes the regular expression **b**, then L(b) = {b}.

* This means that the regular expression **b** means the set of characters consisting only of b, which means that the regular expression **b** is the character b.
* On the same train of thought, a string s is the regular expression denoting a set containing only s.
  + If it contains metcharacters, s can be quoted to avoid ambiguity (“s”).

**The Empty String**

- A regular expression can be the empty string.

- This regular expression simply means the set containing only the empty string.

**The Regular Expression That Matches Nothing**

- This is not the same as the empty string defined by L(Empty) = {}, the pattern for nothing. It generates the set containing nothing.

**Concatenation of Strings**

- Concatenation joins the strings together.

- Concatenation with the empty string leaves a string unchanged.

**Operations with Regular Expressions**

**The | Operator**

- If **x** and **y** are regular expressions, then **x | y** is the regular expression with alternatives.

* It means a string that matches either the regular expression **x** or the regular expression **y**.
* L(x|y) = L(x)L(y).
* If ∑ = {a,b,c,d} then L(a|b) = {a,b} and L(c|d) = {c,d} so a|b is either a or b and c|d is either c or d.

**Concatenation (Writing Adjacent Letters or Strings, or . or ,)**

- The concatenation of two regular expressions x and y is xy. It means take one of x’s string followed by one of y’s strings.

* L(xy) = L(x)L(y)

e.g.

* (a|b)c
  + The parenthesis enforces alternation before concatenation.
  + L((a|b)c) = L(a|b)L(c) = {a,b}{c} = {ac,bc}
* (a|b)(a|b)
  + The parenthesis enforces alternation before concatenation.
  + L((a|b)(a|b)) = L(a|b)L(a|b) = {a,b}{a,b} = {aa, bb, ab, ba}
* aa | ba | ab | bb
  + This is another way of writing the example from above.

**Kleene Closure**

- For a regular expression r, Kleene closure is defined as L(r\*) = L(r)\*, which means concatenation with all powers of L.

* e.g. L((a|bb)\*) = L(a|bb)\* = L(a|bb)0 + L(a|bb)1 + L(a|bb)2 + L(a|bb)3 + …
  + with L(a|bb)0 = {empty string}
  + L(a|bb)1 = {a, bb}
  + L(a|bb)2 = {a,bb}{a,bb} = {aa, abb, bba, bbbb}
  + L(a|bb)3 = {aa, abb, bba, bbbb}{a,bb} = {aaa, abba, bbaa, bbbba, aabb, abbbb, bbabb, bbbbbb}
  + Therefore L((a|bb)\*) = {empty, a, bb, aa, abb, bba, bbbb, aaa, abba, bbaa, bbbba, aabb, abbbb, bbabb, bbbbbb,…}

**Nonstandard Regular Expressions Operations**

**Positive Closure**

- a+ = aa\* (also a\* = a+|empty)

**Exponentiation or Power Operation**

- ak – aaa..a (k times)

**Optional Inclusion (Zero or one ?)**

- a? = empty | a

**Character Classes**

- Specify a range of characters or numbers that follow a sequence.

* [a-z] means any character in the range a to z.
* [A-Z] means any character in the range A to Z.

- A regular expression for the pattern for an identifier that begins with a letter or an underscore and is followed by any number of numbers and letters is

[a-zA-Z\_][A-Za-z0-9]\*

- A **complement character class** is specified using the ^ or ~ symbols, or **not** operator.

* [^a-z] matches any character except a to z.

**Operator Precedence**

- The parentheses, () allow one to override the precedence of operations.

- In order of decreasing precedence:

* () – grouping
* [] – character classes
* \*, +, k, ?, ‘.’, ‘,’ – Kleene star, positive closure, power, optional inclusion, concatenation
* | Alternation operator

**Formal Definition of Regular Expressions**

- Each regular expression denotes (defines) a set of strings (regular sets).

- Formally, regular expressions are defined as follows:

* denotes the empty set
* is a regular expression denoting the set that contains only the empty string.
* A string **s** is a regular expression denoting a set containing only **s**.
* If A and B are regular expressions, then A|B, AB, and A\* are also regular expressions. In this case, A and B are sometimes called **regular definitions**.

- Regular expressions define sets of strings. Regular expression operations are operations on sets of strings.

**Limitations of Regular Expressions**

- Regular expressions are simpler than grammars, but are less powerful.

* Some patterns can only be derived from grammars.

- Every pattern that can be described by a regular expression can also be described by a grammar.

**Operations on Sets of Strings**

**Concatenation of Sets**

- The concatenation of two sets A and B is defined by AB = {xy | x in A and y in B}, which reads as the set of strings xy such that x is in A and y is in B.

* e.g. if A = {a,b} and B = {c,d}, then AB = {ac, ad, bc, bd}

**Powers of Sets**

- The power of a set A: A4 = {x | four-symbol string} reads as the set of strings with four symbols.

- This is just repeated concatenation: A0 = {empty}, A1 = A, A2 = AA, A3 = AAA, A4 = AAAA

**Union of Sets**

- The union of two sets A and B is defined by A U B = {x | x in A or x in B} which reads as the set of strings x such that x is in A or x is in B.

* e.g. If A = {a,b} and B = {c,d}, then AUB = {a,b,c,d}

**Kleene Closure**

- The Kleene closure of a set A is the \* operator defined as the set of all strings including the empty string A\* = and it is the union of all powers of A.

- A\* = A0 + A1 + A2 + A3 + …

**Positive Closure**

- The positive closure is the + operator defined as the set of all strings excluding the empty string.

**Lecture 8 – Finite Automata**

- A **finite automaton** (**FA**) or **finite state machine** (**FSM**) consists of the following components:

1. A finite set of states, one of which is designated as the initial state, called the **start state**, and some states which are designated as **final state** AKA **halting states**, **terminal states**, or **accepting states**.
2. An alphabet ∑ of possible input symbols (including the empty symbol) from which are formed the strings (words) of the language recognized by the FA.
3. A finite set of transitions that determine for each state and for each symbol of the input alphabet which state to go to next.
   1. The FA must have a transition function **move** or **next\_state** that maps each state-symbol pair to a single state or set of states.
   2. If the function maps each state-symbol pair to a set containing only one state, the FA is defined as a **Deterministic Finite Automaton** (**DFA**).
   3. If the function maps each state-symbol pair to a set containing more than one state, the FA is defined as a **Nondeterministic Finite Automaton** (**NFA**)
      1. NFA can be easily constructed from regular expressions and then converted to DFA. Any NFA can be converted to a DFA.
   4. The set of transitions can be represented in different ways, but the most common and the easiest for implementation in computers is the **transition table** representation.
      1. The transition table has one row for each state and one column for each input symbol.
      2. The entry for row **I** and column **S** is the state (for DFA) that can be reached by a transition from state **I** on input symbol **S**.
      3. The transition table representation has the advantage that it can be built directly from the corresponding transition diagram and it provides very fast access.
      4. Its disadvantage is that it takes a lot of space.
      5. A compromise to this is to compress the table to save space and make access a bit slower.

- Since the FA accept or recognize strings of a language they are also called **finite acceptors** or **language recognizers**.

- A **recognizer** is able to take a string ‘x’ and determine whether the string belongs to the language defined by the FA.

- A regular expression can be turned into a recognizer by making a transition diagram or transition table which represents a finite automaton.

- An automata can be considered either deterministic or non-deterministic.

- Both types of automata can be used in developing a lexical analyzer.

- Algorithms involving automata exist which can help improve the design of scanners.

**Deterministic Finite Automata (DFA)**

- A DFA is a special case of an NFA, except for the following restrictions.

* No empty symbol transitions exist within it.
* Every transition from a state is unique.

- An NFA can always be converted into a DFA.

* As a result of this, its number of states will increase.

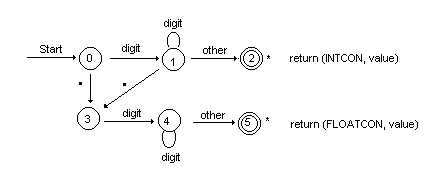
- A DFA will often be faster than an NFA because decisions regarding transitions don’t have to be made.

**Implementing Transition Diagrams – Manually**

- A transition diagram can be implemented manually, with states being represented by variables, and transitions being decided on by if statements.

- Consider a procedure which recognizes floating point and integer constants.

* Transitions are indicated by arrows with an input symbol label, states are indicated by circles with unique labels and final states are indicated by two concentric circles with unique labels.
* A \* symbol indicates that the input must be retracted – the last symbol read must be returned back to the input stream of symbols.



- Cases would be developed which handle all states, and all transitions from one state to another.

**Example of Transition Diagram Implementation**

Token next\_token() {

while (1) {

switch(state)

case 0:

c = nextChar();

if (c == ' ') {

lexeme\_start++;

state = 0;

}

else if (isdigit(c)) state = 1;

else if (c == '.') state = 3;

break;

case 1:

c = nextChar();

if (isdigit(c)) state = 1;

else if (c == '.') state = 3;

else state = 2;

break;

case 2:

retract(1); //Backtrack in input

store\_the\_lexeme();

return(INTCON);

// CASES 3,4, and 5 would also

// be implemented here.

}

}

- Note that since functions can only return one value (and in this case, it is returning just the token), one is storing the actual lexeme somewhere else in memory by calling a function.

**Implementing Transition Tables – Arrays**

- A two-dimensional array can be used to implement a transition table.

- A table consists of rows representing states and columns representing character classes.

- Examples of character classes include:

* Digits
* Letters
* Punctuation
* Special separator charactors

- The classes for the previous example, and an assigned value would be as follows.

|  |  |
| --- | --- |
| **Class** | **Value** |
| Digit | 0 |
| . | 1 |
| Other | 2 |

- A table representing the transition diagram is as follows:

int Table[6][3] = {

/\* State 0 \*/ { 1 , 3 , ES },

/\* State 1 \*/ { 1 , 3 , 2 },

/\* State 2 \*/ { IS , IS , IS }};

- Every row in this transition table represents one state, while every column represents a character class.

- Error State and Illegal state are pre-defined constants indicating either an invalid transition (IS), or a final state (ES) reporting a semantic (spelling) error.

- Note that the rows and the columns can be swapped in the table.

**Implementation of Table-driven Transitions**

- Given the state table developed previously, the following functions can be used to find the transition from one state to another.

int next\_ntate(int currentState, char ch)

{

int column = get\_column(ch)

return Table[currentState][column];

}

// Given a character, find what class it

// belongs to, to be used for table indexing

int get\_column(char ch) {

if isdigit(ch) return(0);

else if (ch = '.') return 1;

else return 2;

}

- The NFA implementation is a loop in which a character is taken from the input, the *next\_state()* function is called and the type of the returned state is tested.

* If the state is not-accepting, the loop continues.
* If the state is accepting, the loop is terminated and the recognized string (lexeme) is processed by the corresponding accepting function.
  + Usually an array of pointers to functions is used to call an accepting function using the accepting state number as an index.

**Lecture 9 – From Regular Expressions To NFA To DFA**

- Kleene’s theorem proves that RE and FA are equivalent language definition methods.

- Based on this theoretical result, practical algorithms have been developed enabling one to actually construct FA’s from RE’s and to simulate the FA with a computer program using Transition Tables.

* Following this progression, an NFA is constructed first from a regular expression, and then the NFA is reconstructed to a DFA.
* Finally, from here, a transition table is built.

- The **Thompson’s Construction Algorithm** is one of the algorithms that can be used to build a Nondeterministic Finite Automaton (NFA) from RE, and the **Subset Construction Algorithm** can be applied to convert the NFA into a Deterministic Finite Automaton (**DFA**).

* The last step is to build a transition table.

- One needs a finite state machine that is a deterministic finite automaton (DFA) so that each state has one unique edge for an input alphabet element.

* This allows for one to cast away all ambiguity during the code generation process.

- A NFA with more than one edge for an input alphabet element is easier to construct using a general algorithm – Thompson’s construction.

* Then following a standard procedure, the NFA can be converted to a DFA for coding.

**Regular Expression**

- Consider the regular expression r = (a|b)\*abb that matches {abb, aabb, babb, aaabb, bbabb, ababb, aababb}.

- To construct a NFA from this, use Thompson’s construction.

* This method constructs a regular expression from its components using empty symbol transition. The transitions act as a glue for the subcomponent NFAs.
* An -transition table adds nothing since concatenation with the empty string leaves a regular expression unchanged (concatenation with is the identity operation).

Parse the regular expression into its subexpressions involving alphabet symbols a, b, and : , a, b, a|b, ()\*, ab, abb

- These describe:

* A regular expression for single characters , a, and b.
* Alternation between a and b representing the union of the sets: L(a) U L(b)
* The Kleene star ()\*
* Concatenation of a and b: ab, and also abb.

- Subexpressions of these kinds have their own NFA from which the overall NFA is constructed.

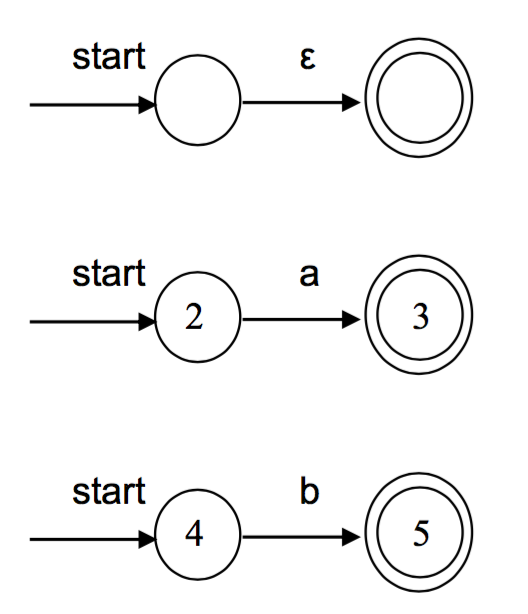
* Each component NFA has its own start and end accepting states.

- A NFA has a transition diagram with possibly more than one edge for a symbol (character of the alphabet) that has a start state and an accepting state.

* The NFA definitely provides an accepting state for the symbol.

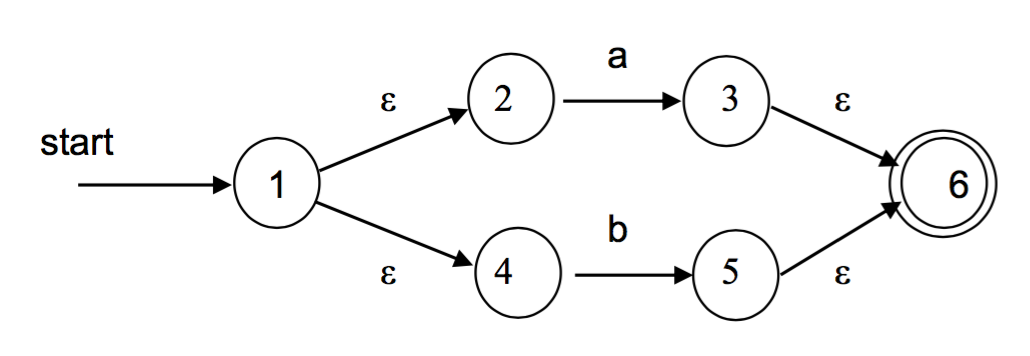
- One takes these NFAs in turn:

* The NFAs for single character regular expressions , a, and b.



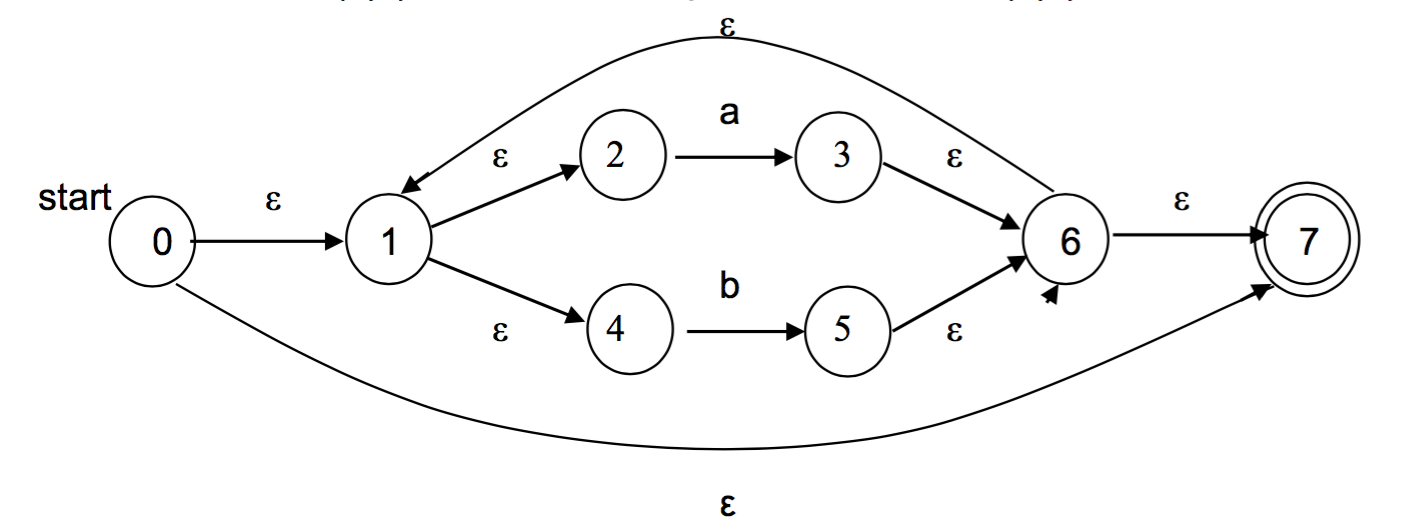
- The NFA for the union of a and b: a|b is constructed from the individual NFAs using the NFA as “glue”.

* Remove the individual accepting states and replace with the overall accepting state.

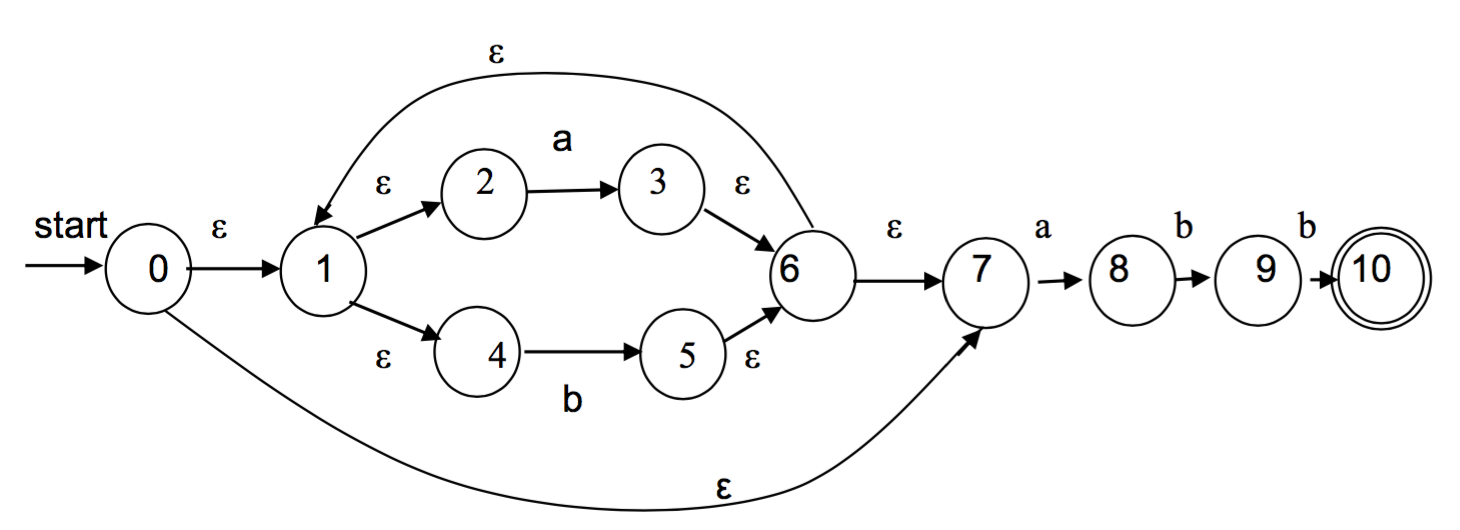


- Apply a Kleene star on (a|b)\*.

* The NFA accepts in addition to (a|b)\*



- Concatenate with abb



- This is the complete NFA and it describes the regular expression (a|b)\*abb.

- The problem is that it is not suitable as the basis of a DFA transition table since there are multiple edges leaving many states (0,1,6).

**Converting an NFA to a DFA**

- A DFA has at most one edge from each state for a given symbol and is a suitable basis for a transition table.

- One needs to eliminate the transitions by subset construction.

**Definitions**

- Consider a single state S and a set of states T.

|  |  |
| --- | --- |
| **Operation** | **Description** |
| -closure (S) | Set of NFA states reachable from NFA state **s** on -transitions alone |
| -closure (T) | Set of NFA states reachable from set of states T on -transitions alone |
| move(T,a) | Set of states to which there is a transition on input symbol **a** from some NFA state in T |

- One has as input the set of N states.

* They can then generate as output a set of D states in a DFA.
* An NFA with n states can generate a DFA with 2n states.

**NFA with n states -> DFA with 2n states**

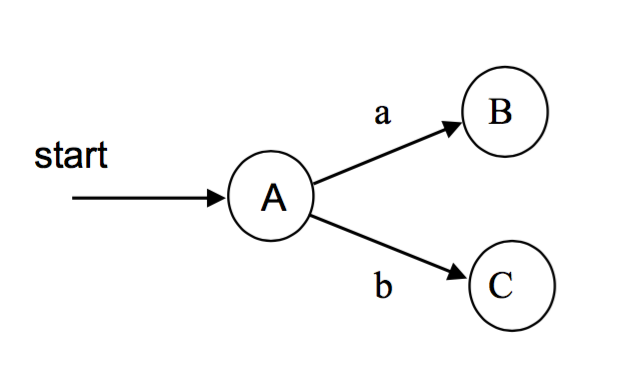
**Start the Conversion**

1. Begin with the start state 0 and calculate -closure(0).

* The set of states reachable by -transitions which includes 0 itself is {0,1,2,4,7}. This defines a new state A in the DFA

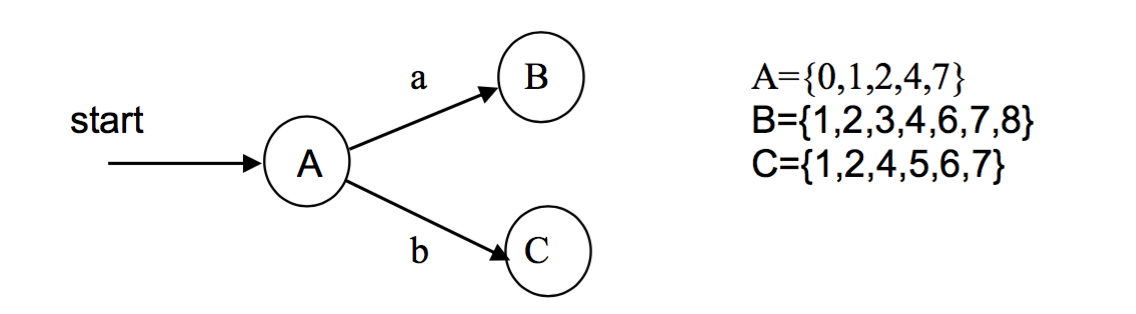
1. One must then find the states that A connects to.

* There are two symbols in the language (a, and b) so in the DFA one expects only two edges: from A on a and from A on b.
* Call these states B and C.



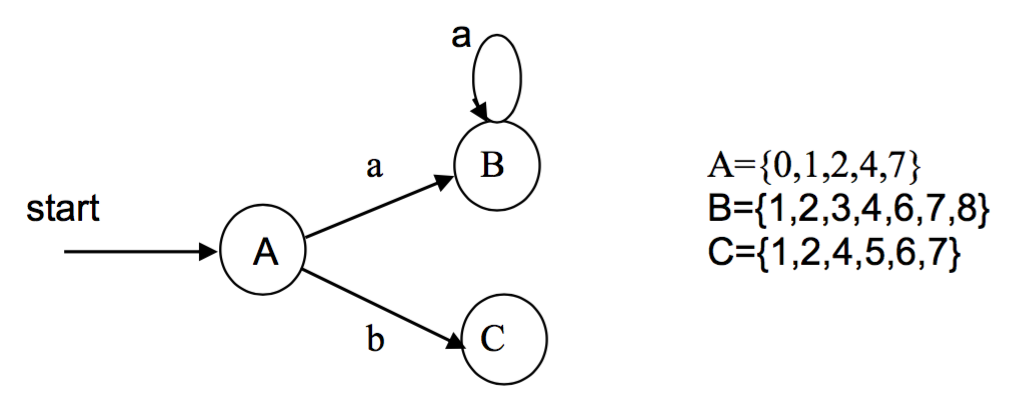
- One finds B and C in the following way:

* **Find the state B that has an edge on a from the set A**.
  + Start with A{0,1,2,4,7}. Find which states in A have states reachable by **a** transitions.
  + This set is called move(A,a) the set is {3,8}.
  + Do an -closure on move(A,a), finding all the states in move(A,a) which are reachable with -transitions.
  + There are states 3 and 8 to consider.
    - Starting with 3, one can get to 3 and 6 and from 6 and from 6 to 1 and 7, and from 1 to 2 and 4.
    - Starting with 8, one can get to 8 only.
  + So the complete set is {1,2,3,4,5,6,7,8}, which defines the new state B that has an edge on a from A.
* **Find the state C that has an edge on b from A**
  + Start with A{0,1,2,4,7}. Find which states in A have states reachable by b transitions.
  + This set is called move(A,b) and it is {5}: move(A,b) = {5}
  + Do an -closure on move(A,b), finding all the states in move(A,b) which are reachable with -transitions.
    - With state 5, one can get to 5,6,7,1,2,4, so the complete set is {1,2,4,5,6,7}
  + So -closure(move(A,b)) = C = {1,2,4,5,6,7,}

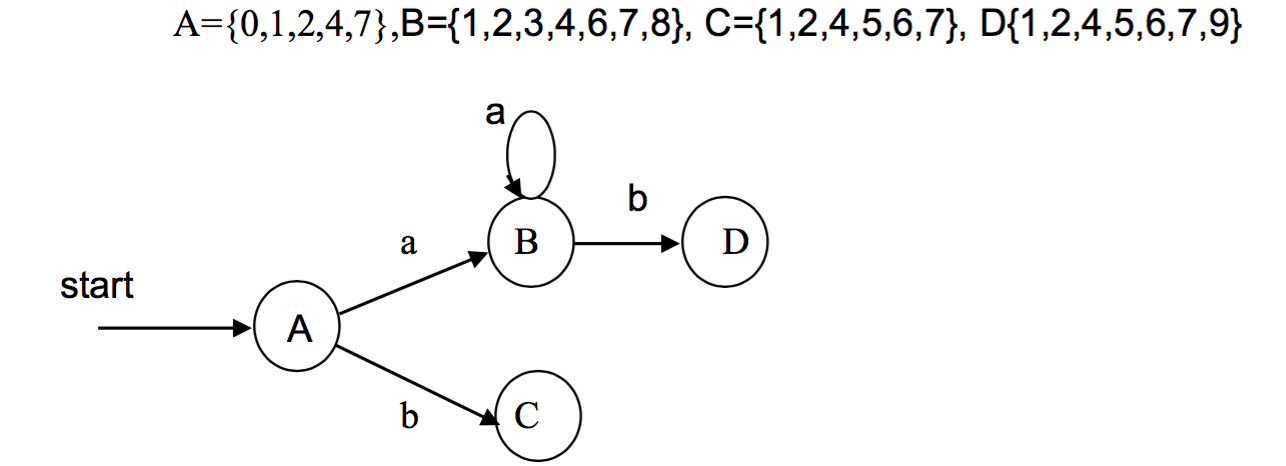


- Now that B and C are found, one can move on to find the states that have a and b transition from B and C.

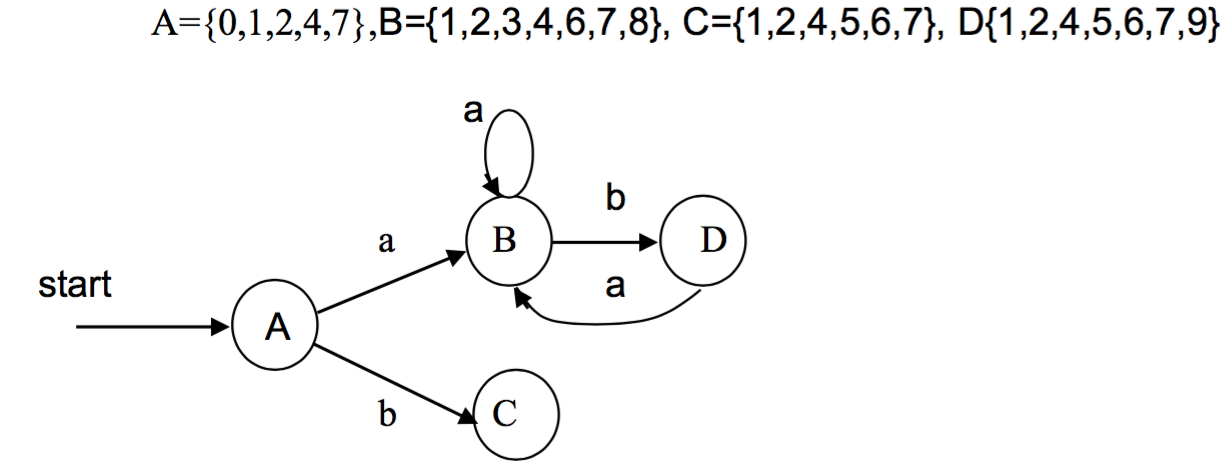
* **Find the state that has an edge on a from B**
  + Start with B{1,2,3,4,6,7,8} and find which states in B have states reachable by **a** transitions.
  + This set is called move (B,A) and the set is {3,8}: move(B,a) = {3,8}
  + Do an -closure on move(B,a), finding all the states within it which are reachable with -transitions.
  + There are states 3 and 8 to consider.
    - With state 3, one can get to 7, and from 1 to 2 and get to 3 and 6 and from 6 to 1 and 4.
    - With state 8, one can get to 8 only.
  + So -closure(move(B,a)) = {1,2,3,4,6,7,8,} **which is the same as the state B itself**.
    - There is a repeating edge to B.



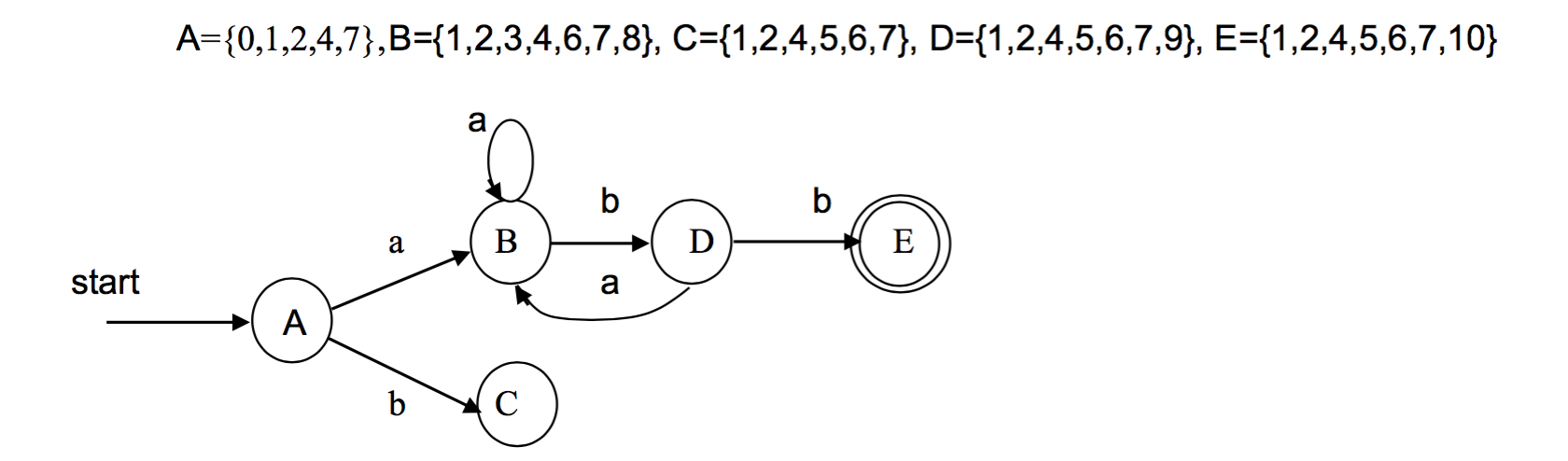
* **Find the state D that has an edge on b from B**
  + Start with B{1,2,3,4,6,7,8,} and find which states are reachable by **b** transitions from states in B.
  + Move(B,b) = {5,9}
  + -closure(move(B,b)) = D = {1,2,4,5,6,7,9}
  + This defines the new state D that has an edge on **b** from B.



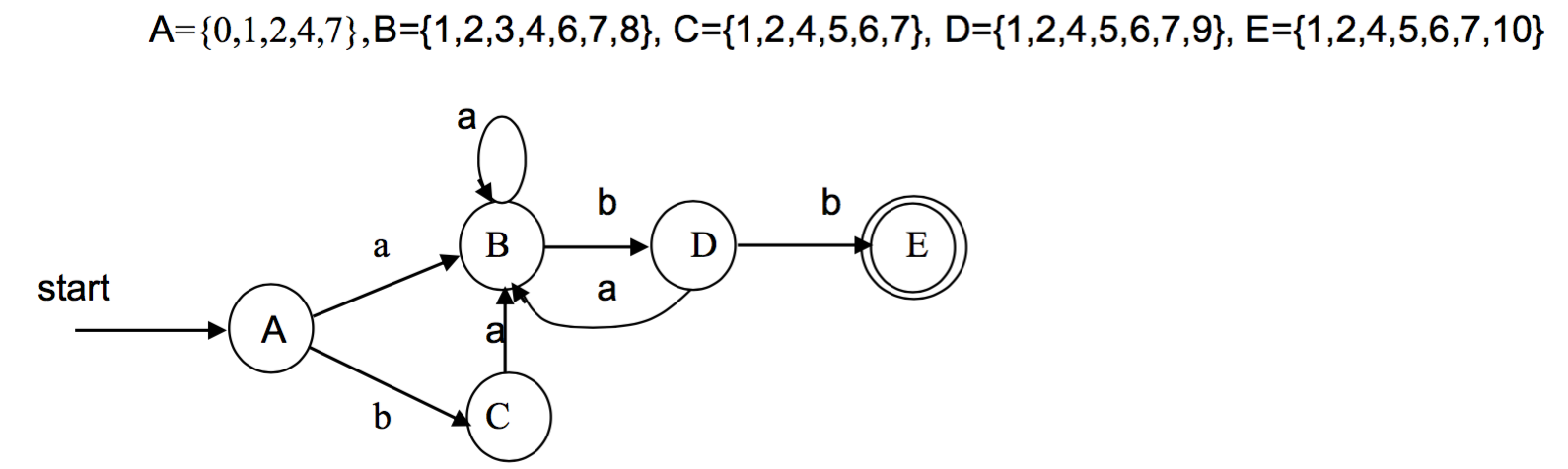
* **Find the state that has an edge on a from D**
  + Start with D{1,2,4,5,6,7,9} and find which states in D have states reachable by **a** transitions.
  + move(D,a) = {3,8}
  + -closure(move(D,a) = {1,2,3,4,6,7,8} = B
  + This is a return edge to B.



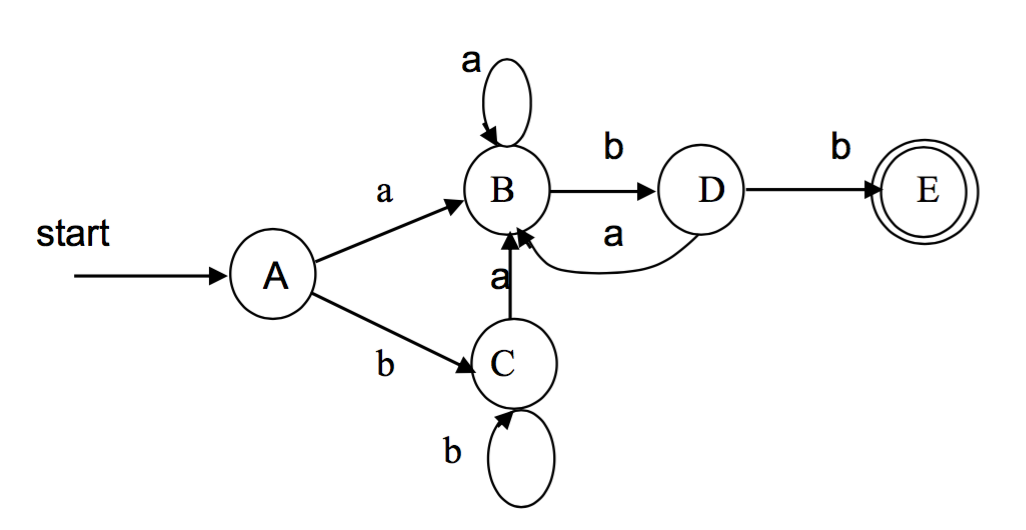
* **Find the state E that has an edge on b from D**
  + Start with D{1,2,4,5,6,7,9} and find which states in D have states reachable by **b** transitions.
  + Move(D,b) = {5,10}
  + -closure(move(D,b)) = E = {1,2,4,5,6,7,10}
  + This defines the new state E that has an edge on **b** from D.
  + **Since it contains an accepting state, it is also an accepting state**.



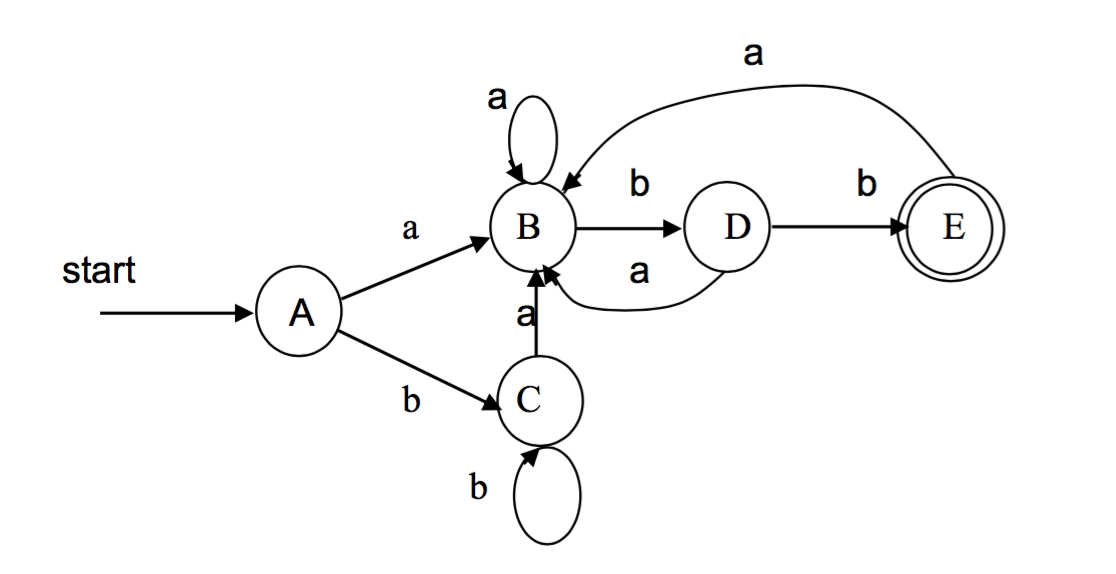
* **Find the state that has an edge on a from C**
  + Start with C{1,2,4,5,6,7} and find which states in C have states reachable by **a** transitions.
  + move(C,a) = {3,8}
  + -closure(move(C,a)) = B



* **Find the state that has an edge on b from C**
  + Start with C{1,2,4,5,6,7} and find which states in C have states reachable by **b** transitions.
  + move(C,b) = {5}
  + -closure(move(C,b)) = {1,2,4,5,6,7} = C

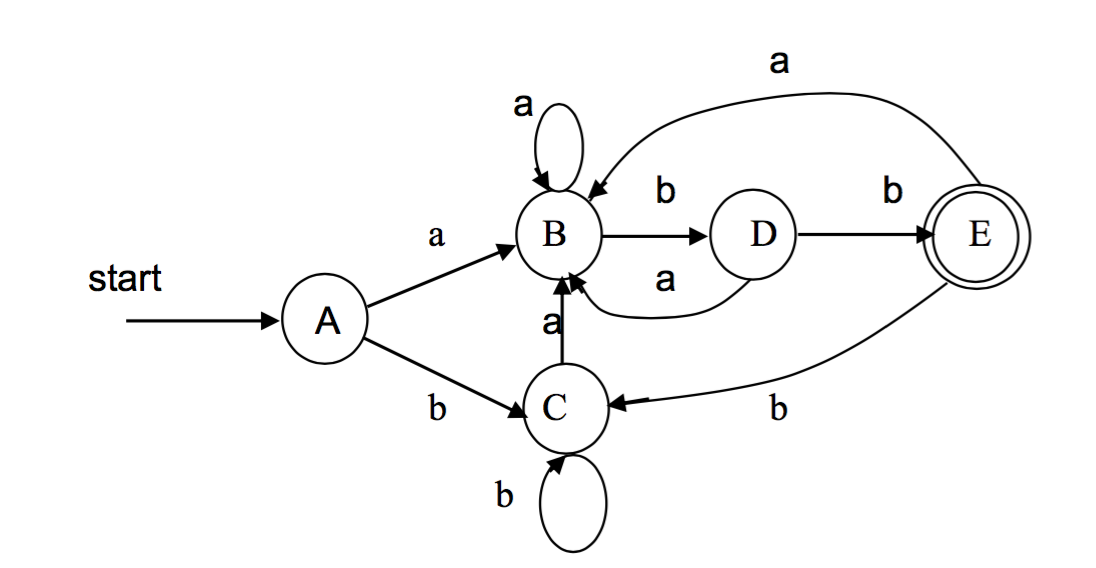


* Looking at E now, one can see that although it is an accepting state, the regex allows one to repeat adding in more as and bs as long as they return to the accepting state E in the end.
* **Find the state that has an edge on a from E**
  + Start with E{1,2,4,5,6,7,10} and find which states in E have states reachable by **a** transitions.
  + move(E,a) = {3,8}
  + -closure(move(E,a)) = B

****

**Find the state that has an edge on b from E**

* Start with E{1,2,4,5,6,7,10} and find which states in E have states reachable by **b** transitions.
* move(E,b) = {5}
* -closure(move(E,b)) = C

****

**-** That does it on the NFA -> DFA transition.

* There is only one edge from each state for a given input character, as expected.
* It now requires **other** as an edge beyond E leading to the ultimate accepting state.
* The DFA can also be optimized to include less states.

- The transition table so far looks like

|  |  |  |
| --- | --- | --- |
| **State** | **a** | **b** |
| A | B | C |
| B | B | D |
| C | B | C |
| D | B | E |
| E | B | C |

**NFAs vs. DFAs (Space-Time Tradeoffs)**

- The table below summarizes the worst-case for determining whether an input string **x** belongs to the language denoted by a regular expression **r** using recognizers constructed from an NFA and DFA.

|  |  |  |
| --- | --- | --- |
| **Automaton** | **Space** | **Time** |
| NFA | O(|r|) | O(|r| \* |x|) |
| DFA | O(2r) | O(|x|) |

* |r| is the length of the regular expression **r** and |x| is the length of the string **x**.

**Lecture 12 – Symbol Tables**

- A **symbol table** is a mechanism that associates **attributes** with **name**.

* Because these attributes are a representation of the meaning (or semantics) of the names with which they are associated, a symbol table is sometimes called a **dictionary**.

- A symbol table is a necessary component of a compiler because the introduction of a name appears in only one place in a program, its declaration or definition, whereas the name may be used in any number of places within the program text.

* Each time the name is encountered, the symbol table provides access to the information collected about the name during the compilation process.

- From an implementation point of view, a symbol table is a specialized database containing records for names and associated attribute, usually one record per name.

* This database consists of two parts: The **Database Manager** (**Symbol Table Manager**) along with the **Database Record Structure**.

- The STM provides services to the client and separates and hides the particular implementation of the database record structure from the client.

- Typical symbol table manager services include:

* Creating a symbol table record structure
* Installing or inserting a name record
* Finding or looking up a name record
* Updating a record
* Deleting a record
* Destroying an entire symbol table
* Other assorted auxiliary services like sorting, printing, and packing

- A typical implementation of the symbol table record structure is:

* An unordered or ordered liner structure: arrays and linked lists.
* Hierarchical structures: binary trees or hash tables
* Combinations of linear and hierarchical structures

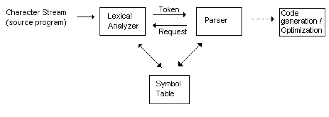
- The design choice of a specific implementation depends mainly on the characteristics of the source language the compiler has to translate and to some extent on the compiler implementation language.

* In all cases, the two important leading criteria determining the design of the symbol table are **storage space** and **access speed**.
* The speed is usually determined by the time required for two frequently used operations, adding and finding a record.

**Lecture 13 – Parsing Overview**

**Overview**

- The parser analyzes streams of tokens retrieved from the lexical analyzer to see if they produce a valid string in the language.



- The parser also provides information to be used by later stages in the compiler.

- The symbol table may be used and updated (e.g. attributes may be set in the symbol table, such as whether an identifier is a function name or variable).

**LL vs. LR Grammars**

- Grammars are divided up into several broad classes.

- Any particular programming language can typically be described using grammars of either type.

* There are also algorithms which can change one grammar to another.

- Two grammars are equivalent if they can generate the same language (although with different parse trees).

- With **LL Grammars**, input to the parser is accepted starting from the left, and the leftmost derivative is taken.

- With **LR Grammars**, input to the parser is accepted starting from the left and the rightmost derivative is taken.

- Often, a number is attached to the grammar, used to indicate the minimum number of tokens that the parser needs to look ahead in the input stream.

* An LL(1) grammar looks ahead by one token.
* If no number is specified, then it is assumed to be 1.

**Specifying Derivations**

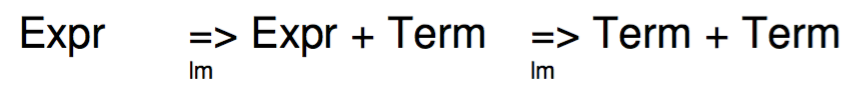
- There are several ways to view the process by which a grammar defines sentences of a language.

* Up to now, this has been viewed as a process of building parse trees.
* It can however also be defined by a linear process of nonterminal replacement called derivations.

- The central idea behind derivations is that a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production.

- When a sequence of derivations are made, the application of a production can be specified using a => symbol.

- If the derivation is leftmost, it will often have an ‘lm’ below the => symbol.



- One can also use an \* above the => symbol to indicate that a string can be derived (perhaps in multiple steps) from the non-terminal on the right hand side.

- If left derivations are used to parse the input string, the non-terminal is said to be in left-sentinel form.

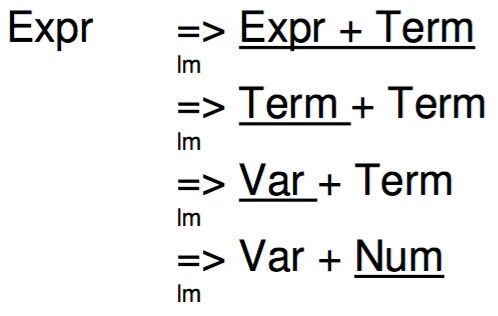
- If right derivations are used to parse the token stream, they are sometimes called canonical derivations.

**Top Down Parsing Example**

- Given a grammar for the following expressions:

* Expr -> Expr + Term | Expr – Term | Term
* Term -> Num | Var

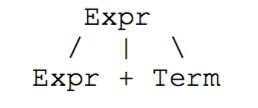
- And the input stream (x + 1), the following steps result.



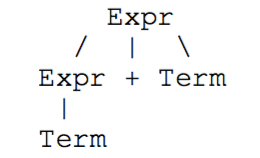
**Parse Tree for Top Down Parsing**

- The parse tree in the previous example would begin with just the start symbol *Expr*.

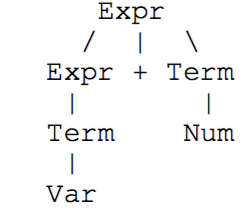
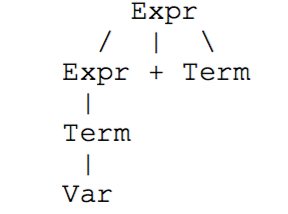
- The expression would then be broken down to:



- The non-terminal expr would be broken down next:



- In the last two trees, the last two terminals are resolved:



**A Further Example**

- Given the following grammar:

* S -> cAd
* A -> aXb | X
* X -> e | f

- If one is given the string *cafbd*, the derivations to build it are as follows:

* S => cAd => caXbd => cafbd

**Bottom Up Parsing**

- The parser tries to match the low level productions first, gradually working up towards building higher level ones.

- This parsing style is best suited for LR grammars.

* It’s also suited for automated implementation (e.g. YYACC, Bison, Javacc).

**Bottom Up Parsing Example**

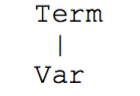
- Given the same grammar as used previously:

* Expr -> Expr + Term | Expr – Term | Term
* Term -> Num | Var

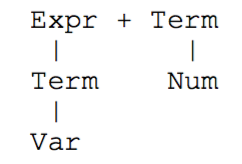
- And the same input stream *x + 1*.

- Parsing begins with the first token *Var*.

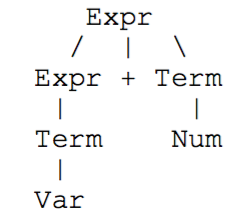
- A production is applied to obtain the following.



- After several more tokens are analyzed, and additional productions are applied, the following is produced:



- Finally, the completed tree is produced:

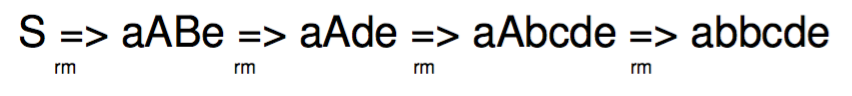


**Another Example**

- Given the following grammar:

* S -> aABe
* A -> Abc | b
* B -> d

- If given the string *abbcde*, the right-most derivation to build this string are as follows:



- The sentence *abbcde*can be reduced to S by the following steps:

* Abbcde, aAbcde, aAde, AABe, S

- These reductions trace out the right-most derivation in reverse.

- Some bottom up parsers are LR(0), SLR(1), LR(1), LR(k), LALR, and operator-precedence parsers.

* They all use parsing tables to perform the shift-reduce parsing operations.

**Error-Handling**

- The parser can detect a much wider range of errors than the scanner, such as:

* Incorrectly structured statements (e.g. if statements with no condition expression)
* Problems with arithmetic expressions.

- These errors are largely syntax errors.

- Any error handling routines should have the following features:

* Errors should be reported accurately.
* It should recover from errors quickly so that later errors can be detected.
* It should not slow down processing of correct programs by a significant margin.

**Error Correction Strategies**

- **Panic Mode** – When an error is detected, tokens are discarded until some type of synchronizing token is found (such as a ; in C).

* Usually the easiest to implement.

- **Phrase-Level Recovery** – Replace tokens with ones that would make the program syntactically correct and allow the parser to continue (e.g. substitute a ; for a comma if it is where the end of a statement should be).

- Build error productions into the grammar (assuming common errors are known beforehand).

- **Global corrections** – Look at the input and find the least number of changes necessary to produce a valid string.

**Lecture 14 – Grammars and Parsing**

- What follows is a look at a description of a grammar as a way of **deriving** a language and how to design a grammar suitable for a recursive descent parser:

* The objective is a recursive descent parser where the parse tree is not actually generated but is implicit in the derivations.
* A recursive descent parser can only work for certain types of grammars with the right kind of productions.

- **The derivation view is that the sentences of the language are generated by repeated application of the productions**.

- In general, a string Ω can consist of a mixture of terminals and nonterminals.

* A production that derives the string Ω from the nonterminal A has the form A -> Ω

- Since strings are derived from other strings, a series of productions that generates from is written as => => …… => .

- The symbol => means “generates in one step”.

- One can use the symbol =\*>, meaning “generates in 0 or more steps”.

* Thus, =\*> .

- In general, a string Ω is in a **sentinel** form if it can be generated by the grammar

S =\*> Ω.

- If a string µ contains only terminals, then it must have been derived from the start symbol by at least one production.

* This is written as S =+> µ
* This equation **defines the grammar**. Such a string is a **sentence** in the language.

- The metasymbol to show that something **derives in one step** is =>.

* One says that a string *v* derives a string *w* if a production can be used to produce *w* from *v* (v => w).

**-** One can write for some strings v and w (where x and z are other strings of terminals and nonterminals) v = xYz and w = xyz and Y-> y is a production in the grammar that produces the string y from the nonterminal Y.

- If the strings x and v are empty, then one has Y => y

- The metasymbol to signify that something can be **derived in one or more steps** is =>+.

- The metasymbol to signify that something can be **derived in zero or more steps** is =\*>.

///

- Let G[Vt, Vn, P, S] be a grammar.

* A string v is called a **sentential form** if v is derivable from S: S=\*> v.
* A string is a **sentence** if v consists only of terminals which are elements of Vt.
* The language L(G[Vt, Vn, P, S]) is therefore the set of sentences produced by the grammar.
* L(G[Vt, Vn, P, S]) = { v | S =\*> v and v in T }

- A language is the set of terminal strings that is produced by the grammar.

* The sentences are a subset of all possible terminal strings.

**Leftmost and Rightmost Derivations**

- Consider the following grammar:

* expr -> expr op expr | ( expr ) | - expr | **id**
* op -> + | - | \* | /

- The string –(x + y) is a sentence of the grammar because:

**Leftmost derivation**: expr => -expr => -(expr) => -(expr op expr) => -(id op expr)

=> -(x op expr) => -(x + expr) => -(x + id) => -(x + y) **or** **just** expr =\*> -(x + y).

- This is the leftmost derivation because the leftmost terminal is replaced at each stage.

**Rightmost derivation**: expr => -expr => -(expr) => -(expr op expr) => -(expr op id)

=> -(expr op y) => -(expr + y) => -(id + y) => -(x + y)

- Leftmost and rightmost derivations generate the same parse tree and every parse tree has associated with it a unique leftmost and unique rightmost derivation.

**Left and Right Associativity and Right Recursion**

- Consider the following strings: 9-5-2 and a = b = c

**9 – 5 – 2**

- The – operator is left associative since the 5 associates with the – on its left, meaning that it is interpreted as (9-5)-2 = 2, not 9-(5-2) = 6.

**a = b = c**

- The = operator is right associative since the b associates with the = on its right, meaning that a = (b = c), not (a = b) = c

**Left Associativity For Strings**

* list -> **list** + digit | **list** – digit | digit
* digit -> [0-9]

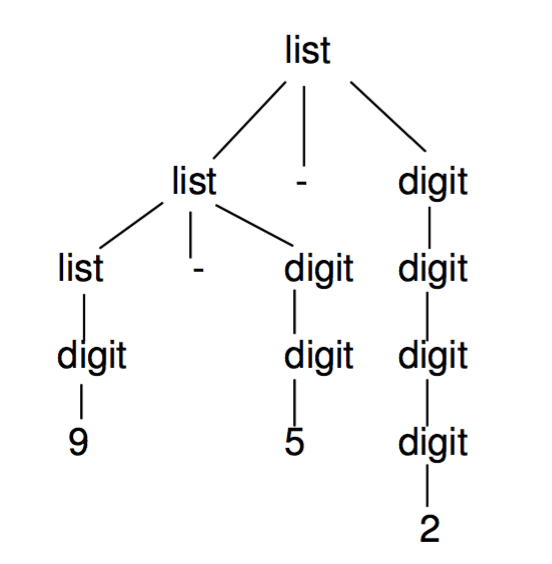
- Left associativity is enforced by evaluating left associating expressions lower down in the parse tree where precedence is higher.

**Right Associativity for Strings of Characters With Assignment**

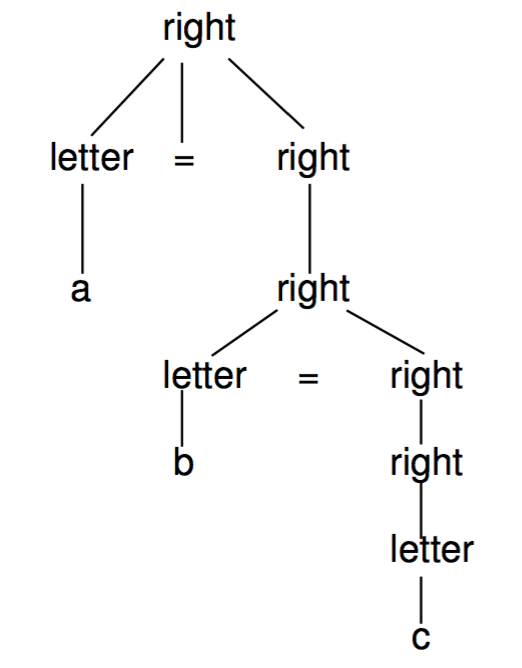
* right -> letter = **right** | letter
* letter -> [a-z]

- Right associativity is enforced by evaluating right associating expressions lower down in the parse tree where precedence is higher.

- Parse trees generated for these expressions are (using leftmost derivation and one derivation at each stage look like



- It is important to note that a **left associative tree expands to the left**.



- It is important to note that a **right associative tree expand to the right**.

**Left and Right Recursion**

- Left associative grammars naturally support the associativity of some operators, but there is a serious problem with recursive-descent parsers.

* Recursive descent parsers implement the productions as function calls.
* So if the nonterminal on the LHS is also the first symbol on the RHS, the function will enter an infinite recursion.

- In general productions look like:

* Left recursion: A -> A Ω | µ
* Right recursion: A -> Ω A | µ

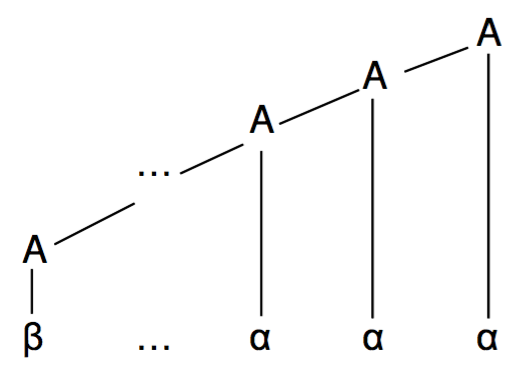
- Here, Ω and µ are sequences of terminals and nonterminals that do not begin with A.

- In general a grammar is left-recursive if it has a nonterminal such that there is a derivation: A =+> A Ω

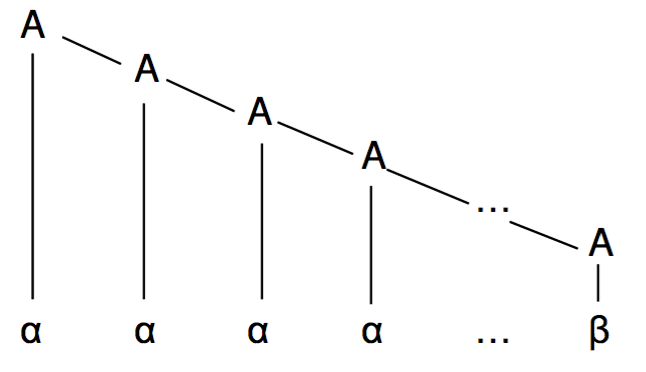
- The strings generated by these productions are of the form:

1. A => A Ω => A Ω Ω => A Ω Ω Ω => ……… µ Ω Ω … Ω

- The parse tree for this looks like:



2. A => Ω A => Ω Ω A => Ω Ω Ω A => ………Ω Ω … Ω µ



- Note that for 1. a recursive-descent parser that attempts to apply the production as a function call would result in the evaluation of the first symbol on RHS which is the same function call, and the beginning of an infinite recursion.

* Hence, in such a parser, one needs to **remove left recursion but implement the code so as to preserve left associativity**.

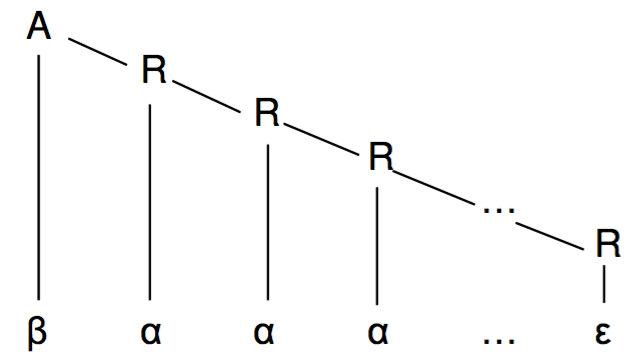
**Eliminating Left Recursion**

- One can convert the left recursion (that follows naturally from the grammar) into a right recursion with the following algorithm that can be applied to any production and does not change the set of derivable strings.

* Original: A -> A Ω | µ
* Converted: A -> µ R

R -> Ω R |

- Note now, that there is right-recursion and an -production, but the strings generated are the same as the left recursive case:



**Example**

Take the left recursive part of the grammar above:

* list -> **list** + digit | **list** – digit | digit

as a set of separate productions:

* list -> list + digit | digit
* list -> list – digit | digit

**For a single alternation**

* list -> list + digit | digit

- With left recursion removed

* list -> digit list’
* list’ -> + digit list’ |

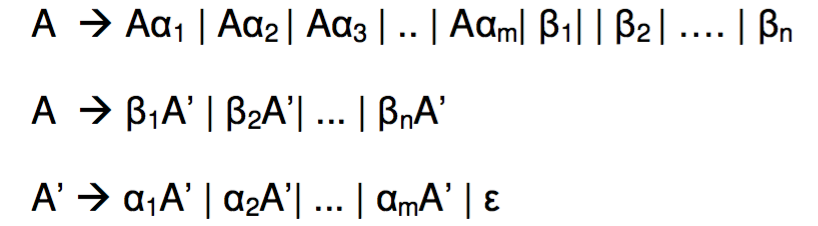
**For both alternations**

* list -> list + digit | digit
* list -> list – digit | digit

- With left recursion removed

* list -> digit list’
* list’ -> + digit list’ | - digit list’ |

- The general rule is:



- These are examples of **immediate** left recursion.

- The recursion could be several steps later:

* S -> A a | b
* A -> A c | S d |

**Left Factoring a Grammar**

- The following grammar is not suitable for predictive parsing.

relational expression -> primary expression > primary expression

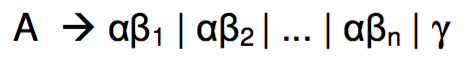
| primary expression < primary expression

- It is not possible to decide which of the alternatives to choose having one token look ahead because all production alternatives have the same left-most terminal.

- It is possible to rework the grammar so that the decision can be deferred until the parser can make the right choice.

* The corresponding grammar transformation is called **left factoring**.

- The general algorithm for left factoring a grammar is that given a grammar



if is not the left factored grammar is

* A -> A’ |
* A’ -> ß1 | ß2 | ß3 | … | ßn

- This transformation must be applied repeatedly until no alternatives for a nonterminal have a common prefix.

- In many cases the grammar may be transformed without applying the general rule.

* e.g. The grammar at the start of the section may be rewritten as

relational expression -> primary expression relational operator primary

expression

relational operator -> < | >

**Building the FIRST and FOLLOW Sets for a Grammar**

- The construction of a predictive parser requires two sets of tokens to be built.

- The FIRST set is a set of tokens that appear at the left-most position after zero or more derivations are applied to a grammar production.

- The formal definition is FIRST(A) = {A | A => A}

- The following rules for computing the FIRST set can be derived from the formal definition.

**BFTS1**. If X is a terminal, then FIRST(X) = {X}

**BFTS2**. If X is a nonterminal and X -> is a production, then add to FIRST(X).

**BFTS3.** If X is a nonterminal and X -> Y1Y2…Y­­k is a production then

* FIRST(X) = {FIRST(Y1)} if Y1 does not derive
* FIRST(X) = {FIRST(Y1), FIRST(Y2)} if Y1 derives
* FIRST(X) = {FIRST(Y1), FIRST(Y­­2), FIRST(Y­3)} if Y1 and Y2 derives .
* FIRST(X) = {FIRST(Y1), FIRST(Y2), FIRST(Y3) … FIRST(Yi)} if is in all of FIRST(Y1), FIRST(Y2), FIRST(Y3), … FIRST(YI-1)
* FIRST(X) = {FIRST(Y1), FIRST(Y2), FIRST(Y3), … FIRST(YK),} if is in all of

FIRST(Y1), FIRST(Y2), FIRST(Y3), … FIRST(Yk)

**BFTS4**. If X is a nonterminal and X -> 1 | 2 | … | m is a production then

FIRST(X) = {FIRST(1),FIRST(2), … , FIRST(k)}

- The FOLLOW set is a set of tokens that appear on the right side of a nonterminal after zero or more derivations in a grammar production.

- The formal definition is FOLLOW(B) = {a | A => Ba}

-The following rules for computing the FOLLOW set can be derived from the formal definition.

**BFWS0.** The FOLLOW set cannot contain .

**BFWS1**. If **S** is the start symbol (nonterminal) for a grammar, place $ in the FOLLOW(S), where $ is the input end-marker (for example, end of file).

**BFWS2**. If there is production A -> | Bb, then FOLLOW(B) = {a,b}

**BFWS3**. If there is a production A -> ß and ß does not derive , then add FIRST(ß) to the FOLLOW(B) set.

**BFWS4.** If there is a production A -> , then add FOLLOW(A) to the FOLLOW(B) set.

**BFWS5**. If there is a production A -> ß and ß derives , then add FOLLOW(A) to the FOLLOW(B) set.

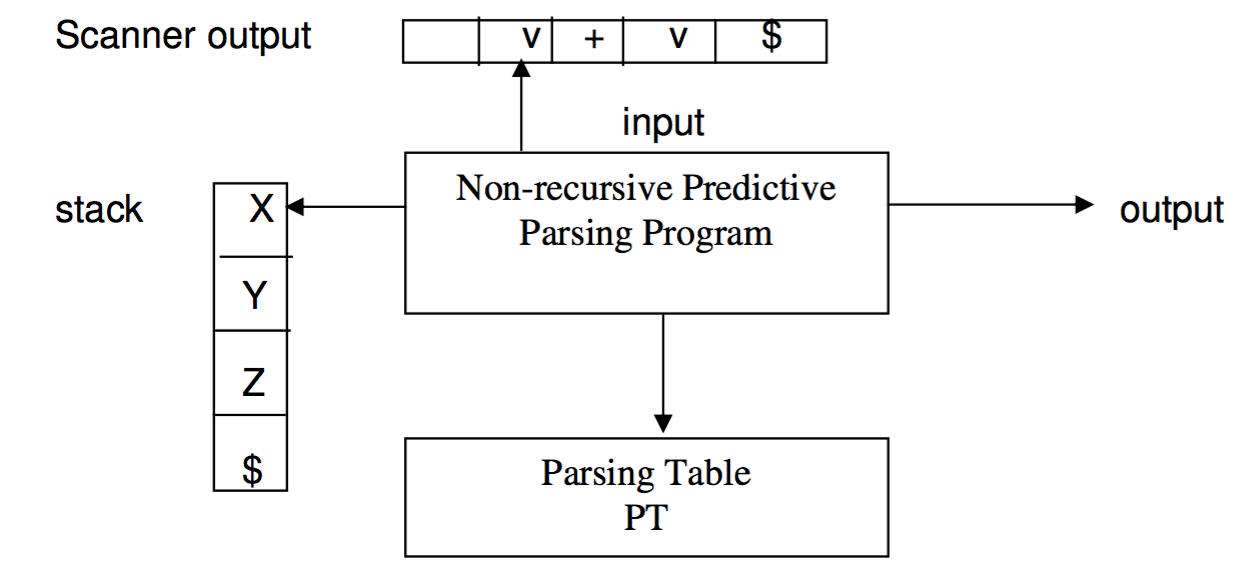
**Lecture 15 – Non-Recursive Predictive Parsing**

- In a recursive descent parser, a nonterminal is implemented as a function call and the RHS of its production as the body of the function.

* Since the body may consist of other nonterminals, then it will call other functions.
* The system stack is used to pass parameters to and from the functions.

- Instead of using the system stack, it is possible for the parser to maintain its own stack and integrate it more closely with the parsing process.

* In addition, since the main task of the function calls is to match the look ahead token, matching can now be done explicitly using grammar symbols (representing productions) on the stack.
* Instead of recursive function calls, a table is now used (analogous to the lexical transition table) to guide the parser to a matching token.



- The components to this table are:

* Input token stream terminated with $
* Stack of grammar tokens with the $ at the bottom
* Parsing table (called frequently like the transition table) showing the next production
* The parsing table is a two-dimensional array PT[A,a], where A is a nonterminal and a is a terminal defined in the language grammar.

- Initially the stack contains the start symbol of the grammar on top of $.

**Operation**

- The program considers X, the symbol at the top of the stack, and a, the current input symbol.

- The subsequent action can be one of:

* If X = a = $ the parsing is complete
* If X = a != $ the parser pops X off the stack and advances the input pointer to the next input symbol.
* If X is a nonterminal, the program consults the table M[A,a] to look up the production with which to replace X on the stack or else give an error.
  + If X -> UVW is the selected production, then X on the stack is replaced by UVW with U at the top of the stack.

- One just needs a parsing table for the grammar.

- The table entries are determined by the FIRST and FOLLOW sets of the grammar.

* The FIRST set determine a production if the input token can be matched to the FIRST set.
* The FOLLOW set determines a production if the production derives and therefore will disappear but can match to a following production that contains the token.

**Building Predictive Parsing Tables**

- The following algorithm can be used to construct a predictive parser table for a language L(G) defined by a grammar G.

**BPPT1**. For each production A -> of the grammar G, do steps 2 and 3.

**BPPT2**. For each terminal a in the FIRST(), add to PT[A,a].

**BPPT3.** If is in FIRST(), add to PT[A,b] for each terminal b in FOLLOW(A).