1. Introduction. *Why Study Programming Languages?* (1)To increase ability to express ideas via and increased vocabulary of useful programming structures. (2) Improved background for choosing appropriate languages. (3) To make it easier to learn new languages. (4) To Improve your understanding of the significance of implementing issues. (5) To improve your use of programming languages you already know. (6) Overall advancement of computing. (7) Understanding that all languages will all do the same thing but in different ways. *Low-level versus high-level programming languages.* (1) No programming language can prevent us from finding certain solutions to a problem, but a given language can influence the types of solutions we are likely to pursue. (2) *Progression from low level – to high level.* Machine language -> Assembly language -> Assembly language with symbolic addresses -> High level languages -> Object oriented approaches, reusable components. *Programming Language Paradigms.* Imperative Languages – You are telling what the logic, flow, and control of the program is. Command-driven or statement driven. (1) **Procedural** (C, Basic, Statement Oriented). Command-driven or statement based. Like machine states. A program is a sequence of states that change with each command that is executed. (2) **Object-Oriented** (C#, Java, Smalltalk). Referentially transparent. Objects are identified and constructed with a limited set of methods that can invoke on instances of those objects. Declarative Languages – Expresses the logic of a computation without describing its control. Very High level. (1) **Functional** (LISP, Scheme, ML, Haskell). Start with considerations of what functions must be applied to the initial state of the computation in order to perform the desired tasks (applicative or functional). Seek to embody the ideas of mathematical functions. Referential transparency – Every time you call a function with a given piece of data, you will get the same result. (2) **Logic Based** (Prolog). Executed by attempting to match enabling conditions. If a match can be made, appropriate actions are carried out and the computation attempts to match subsequent enabling conditions. *Attributes of a Good Programming Language* (also view Sebastia's Criteria) (1) **Abstraction** – Avoid requiring something ti be stated more than once; factor out the recurring pattern. (2) **Automation** – Automate tedious, error-prone activities. (3) **Defense in Depth** – Have a series of defenses so that if an error isn't caught by one, it will be caught by another. (4) **Information Hiding** – The language should permit modules designed so that the user has all the information needed to use the module correctly, and nothing more; the implementor has all the information needed to implement the module correctly, and nothing more. (5) **Manifest Interface** – All interfaces should be apparent(manifest) in the syntax. (6) **Orthogonality** – Independent functions should be controlled by independent mechanisms. (7) **Portability** – Avoid features or facilities that are dependent on a particular machine or small class of machines. (8) **Preservation of Information** – The language should allow the representation of information that the user might know, and that the compiler might need. (9) **Regularity** – Regular rules, without exceptions are easier to learn, use, describe, and implement. (10) **Security** – No program that violates the definition of the language, or its own intended structure, should escape detection. (11) **Simplicity** – A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination. (12) **Structure** – Static structure of the program should correspond in a way with the dynamic structure of the corresponding computations. (13) **Syntactic Consistency** – Similar things should look similar; different things different. (14) **Zero-One-Infinity** – The only reasonable numbers are zero, one, and infinity.

Sebesta's Criteria

Table 1.1 Language evaluation criteria and the characteristics that affect them.

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity/orthogonality	•	•	•
Control structures	•	•	•
Data types and structures	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

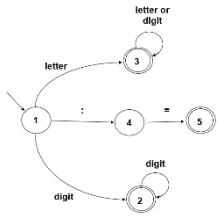
2. Syntax and Semantics. *Chomsky Hierarchy and the classes of grammars*. **4 Tiers** – Type 0: Unrestricted Grammars, Type 1: Context Sensative Grammars. Type 2: Context Free Grammars, Type 3: Regular Grammars. **Automata** – Theoretical Machines. *Finite and Pushdown Automata.* **Finite State Automata** – Represent regular grammars. Used for lexical analysis. Can recognize the symbol patterns which comprise the regular grammars. Lexical Analysis -does a given string in a computer program constitute a legal variable name? **Regular Grammars** (Example left) - Left hand side must contain a single nonterminal. Right hand side must be a terminal, or a terminal followed by a single non terminal or an empty string. State Transition Diagrams (example left) – One circle is designated with a pointer as the start position. One or more have double circles that indicate valid string has been recognized (Accepted states). A string is accepted if the symbols appearing in the string correspond to a sequence of labeled arcs that lead from a start state to the accepted state.

Identifiers:

```
<I> ::= a<L>|b<L>|c<L>|...|z<L>| a |...| z
<L> ::= 1<L>|| 2<L>||...| 0<L>||
        1 | 2 |...| 0 |
        a<L>|| b<L>|| c<L>||...| z<L>||
        a | b |…| z
```

becomes:

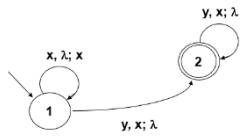
```
<I> ::= <letter> | <letter><Idtail>

<letter> ::= a | b | … | z
<digit> ::= 0 | 1 | … | 9
<Idtail> ::= <letter> | <digit> |
             <letter> <Idtail> |
             <digit><Idtail>
```



Pushdown Automata – Represent context free grammars. Used in syntactical analysis. **Context Free Grammars** – The LHS of a production has one and only one symbol which is a nonterminal. The RHS is unconstrained. Any context free grammar containing lambda rules can be converted to a context free language with no lambda rules. Implemented vi stacks and can be used to build parse trees. Finite automata cannot recognize strings of the form: $\{ x^n y^n : n \in N \}$ because they have no way of remembering how many "x" characters they have encountered to compare with the number of "y" characters they encountered. Has a stack upon which they can store information for later recall. Stack symbols, which constitute some or all the machine's alphabet. Transitions executed are by pushdown automata are variations on this sequence. Read a symbol -> Pop a symbol from the stack -> Push a symbol on the stack -> Enter a new state (example on the right). The lookahead principle – Peeking at future symbols without actually reading them in order to resolve nondeterminism in a state change (The 'K' in LL(K) Parsers). BNF – Backus-Naur Form.

(x, y: z) where:

x = input symbol
y = item that is popped
z = item that is pushed



BNF/EBNF, What they represent, Differences. **BNF** – Provides a formal way of describing the legal constructs of any language that is sufficiently regular. Components of BNF. (1) Terminals (the actual symbols). (2) Non-terminals (that can be replaced by other non-terminals).
Ex(s) <Integer> ::= +<unsigned integer> | -<unsigned integer> | <unsigned integer>
<unsigned integer> ::= <digit> | <unsigned integer><digit>
EBNF (Extended Backus-Naur Form) – Provides extensions to BNF to improving readability. Permits zero or more (*). Permits one or more (+). Optionally include []. Grouping – use an alternative {}.
Ex(s) <integer>::= [+|-]<unsigned integer>
<unsigned integer>:==<digit>*
Writing Grammar Rules. Grammars can be written to generate the exact same language but have different meanings (Semantics). *Left vs Right-Recursive Rules.* Recursive descent needs right recursive rules. Left recursive rules applied to a recursive descent produce an infinite loop. **Recursive decent Parsing** – Top-down algorithm. Can parse context-free grammars. Implemented from the BNF representation of a grammar. Non-terminals of the grammar become procedure. Assume (1)Variable lookahead always contains the type of the next token being read from input. (2) Function getchar reads in a character. (3) At points in the program we might need to ungetch() which puts back a character at which we peeked.
Parse trees, Ambiguous grammars. Ambiguous grammar is a context free grammar for which there exists a string that can have more than one leftmost derivation or parse tree.
Attribute grammars for semantics (can be used to make semantic information available where needed in the parse tree). **Inherited** – Relate non-terminal values with non-terminals that are higher up in the tree. **Synthesized** – relate LHS non-terminal to values of the RHS non-terminal, which are passed up the tree. *Operational, Denotational, Axiomatic Semantics.* **Operational Semantics** – Describe the meaning of a program by specifying the effects of running it on a virtual machine. Machine state changes as the program executes. The virtual machine may be a combination of the actual computer, operating system, and a translator for the language. Representation of a for loop. programming language by translation each construct in the language. programming language is given by a constructs of the grammar.

```
        expr1
loop: if !expr2 goto out
        …
        expr3;
        goto loop
Out: . . .
```

Denotational Semantics – Express semantics of a schema that associate a meaning (denotation) with Meaning Functions – A denotational description of a set of meaning functions associated with the Denotational rule for binary numbers.

$$M_{bin}('0') = 0$$
$$M_{bin}('1') = 1$$
$$M_{bin}(<bin\_num>\ '0') = 2 * M_{bin}\ (<bin\_num>)$$
$$M_{bin}(<bin\_num>\ '1') = 2 * (M_{bin}\ (<bin\_num>) + 1)$$

Consider the synthesized attributes for a grammar for arithmetic expressions:

Production	Attribute
E → E+T	value(E1)= value(E2)+ value(T)
E → T	value(E) = value(T)
T → T*P	value(T1)= value(T2)* value(P)
T → P	value(T) = value(P)
P → I	value(P) = value(I)
P → (E)	value(P) = value((E))

Consider the attribute tree that gives the value of the expression:

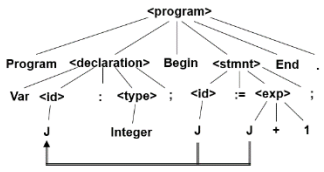
2 + 4 * (1 + 2)

Axiomatic Semantics – Views the definition of programming languages as a theory of the programs written in that language. More abstract than a denotational approach (doesn't try to determine what program constructs mean, only what can be proven about the program's execution). More practical that denotational – not interested in a formal modal language, but interested in facts about the program – type of values they compute, will they halt, etc.

An inference rule for a "while" loop in an algorithm:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S \text{ end } \{I \text{ and not } B\}}$$

3. Translation. *Analysis/Synthesis Model.* Analyzes a source code for syntax and context, then synthesized a target program (executable form). **One pass** (Pascal) linear order of elaboration. **Multiple Passes**. (1) Analyze source program – Understanding the relationship among the tokens in the program. (2) Optimization – Rewritting some representations in the translation process to improve efficiency in execution. (3) Generate Code – Convert the parsed program into an executable format. *Lexical Analysis* (Scanning). Breacking a program into primative components, called tokens (identifiers, numbers, keywords, etc.). Regular grammars and finite state automata are formal modals of this. *Syntactic Analysis* (Parsing). Createa syntax tree of a program. Context free or BNF grammars and pushdown automata are formal models of this. Symbol table creation stores information about declared objects (identifiers, method names, etc.). Takes a falt unstructured sequence of tokens ad produces a structured intermediate form that can be more effectively understood and manipulated (a parse tree). Identifies larger program structures (Statements, declarations, expressions, etc.) using the lexical items (tokens) produced by the tokenizer. Checks and verifies that a program is syntactically correct. Syntac checks (Are all commas, periods, and keywords in the correct place and order. This is handled by a context free grammar). Syntactical analysis alternates with Semantic analysis (identifies a syntactic unit. Analyze the semantics of the unit). Parse Tree -> *LL and LR Parsers – Top-down vs Bottom-up Parsers*. **LL Parsers** (Top-down parsers). Start by putting the start symbol on the stack and growing the parse tree from root node to leaf nodes. Left to right scanning and a leftmost derivation of the sentence. **Difficulties**. (1) Wil result in infinite recursions in recursive descent parsers when the program executes a subprogram based upon them. (2) The parser will be able to predict the rule to apply next based on the first symbol in the RHS. No ambiguity can exist for this prediction to be successful. **LR Parsers** (Bottom-up parsers). Transfer symbols to the stack until the upper-most symbols (called a handle) correspond to the RHS of a single non-terminal found on the LHS of a rule. The single non-terminal is substituted. Subsequent symbols are pushed and substituted until the start symbol only remains on the stack. Shift and Reduce operations – Shift symbols onto the stack, reduce by substituting a LHS symbol. Parse the left- to – right scanning, getting a rightmost derivation. Avoid many of the problems of predictive parsers. Replaces the RHS symbols with the single non-terminal symbol found on the LHS of the rewrite rule before additional symbols are transferred from the input to the stack. On a successful parse, the entire contents of the stack collapse to the grammar's start symbol, indicating that the symbol read to that point from a string that can be derived by the grammar. *Semantic Analysis*. Checks the source program for semantic errors and gathers type information. The compiler checks for data types, scope, types coersions, reals used as subscripts, etc. *Running a program: compile, link, load.* (1) Compile takes source code files and creates corresponding object files (.o files for example). (2) Linking takes all object files, combines them with library files included in order to create a load module. (3) Load takes a .exe and loads it into memory to prepare it to run. *Context Free vs Context Sensative issues and the symbol table.* Symbol table would determine type mismatches. *Intermadiate representations and virtual machines.* **process and system virtual machines.** Process VM – can be stack based or register based. System VM – Emulate other operating systems on top of other host operating systems. **register-based vs stack-based virtual machines.** (1) Register based - Load and store from registers. Performs operations on registers. (2) Stack based - Push operands to a stack, Pop operands , perform operations, and push results. **4. Data names, types, scope, lifetime.** *Implicit vs explicit type declarations*. (1) Implicit takes on a datatype at runtime (Late type binding) based on conventions. (2) Explicit takes on the datatype that is assigned to is as it is declared before the program ever runs. *Strong vs weak typing, typing inference*. Any language that detects all type violations, whether statically before or dynamically during runtime, is strongly typed. Var mystring = 5 (Integer). If there is a type mismatch that is caught later, it is still strongly types. Type inference – assignment variable is assigned to a type required by the expression being evaluated.Determining the types of items without the programmer explicitly specifying them. *Bindings – when can bindings take place?* An association between a named entity and some characteristic of the entity. **When?** (1) Execution or runtime – Variables can arbitrarily bind to storage locations or values. (2) Load-time – Binds chosen by the loader. Globals, static variables, disjoint activation records(Fortran). (3) Translation or Compile time – Binds chosen by programmer (Variable names and types) and binds chosen by translator (relative location of data objects in storage). (4) Program Composition Time – Machine language with absolute addresses. *Static vs dynamic scope*. **Static scope** - definition of a variable is defined in the context of its basic definition. **Dynamic scope** - defined by what is calling it (Exception handlers are all dynamically scoped). *The scope of a variable in a block-strutured language*. Example of a block – For Loop. A variable is declared in and is only used within the span of the code block. *Difference between lifetime and scope of a variable*. Lifetime – the period during program execution when storage for a declared variable exists in some activation record or heap element. Before it is destroyed. Lifetime of a variable is temporal. Lifetime is as long as that program is active. Scope – The portion of the source program where the data item can be accessed. The area of the program where the name of the item is a legal reference. Scope is special. How long in a program is a variable visible (global vs local for example). **5. Procedural Programming**. *Differences between procedural and object-oriented programming languages.* **Procedural** – (1) Can instantiate classes. (2) Uses Procedures instead of methods. (3) User Records instead of data-only class. (4) User Modules instead of classes. (5) Uses Procedure Call instead of message. **Object Oriented** – (1) Uses Methods instead of Procedures. (2) Uses data-only class. (3) Uses classes. (4) Uses Messages. *C as a (not terribly typical) example of a procedural language*. **Irregularities –** (1) Scalar Variables – Using a pointer to a value in memory to set the variable rather than returning a value from a function. (2) Arrays. *When and why was C created?* Created in 1970 in order to create an Operating System. *Pointers (vs references).* (1) Set a pointer and reference using new or malloc. (2) You can set a pointer and reference to null. (3) You can set a pointer to another pointer, same for reference. (4) Cannot set a reference to an absolute memory address, pointers can. *Design tradeoffs (eg: Flexibility versus security)*. C is not considered type safe (it is possible to corrupt a heap using a piece of memory as an incorrect type). This allows for explicit control over low-level issues like the exact layout of objects in memory. Allows for extremely hight performance (A high level language might have required expensive computations). Implement your own garbage collector. Interfaces with code that can only interoperate conveniently and effectively with low level languages. **Why is C unsafe?** C heap values are created in a type-unsafe way. C casts, unchecked array accesses, and unsafe deallocation can corrupt memory during its lifetime. C deallocation is unsafe, can lead to dangling pointers. *Undefined Features, examples and why they are undefined*. Integer Overflow (wraparound or not?). Attempting to modify a string literal (cant modify a string character individually like p[0] = 'W'). Integer divide by 0. Some pointer operations (Attempting to paint to the address just after an array's last index). Reaching the end of a value returning function with no return statement. What to do when arrays go out of bounds as (Can be defined any way the programmer wishes)



1. LL Parser
   a.

Production	Input	Action
<S>	int % int	<S> ::= <T> % <T>
<T> % <T>	int % int	<T> ::= <L>
<L> % <T>	int % int	<L> ::= z
z % <T>	int % int	z ::= int
z % <T>	% int	Match %
z % <T>	int	<T>::= <L>
z % <L>	int	<L>::= z
z % z	Int	z ::= int

Showing <S> ::= z % z

"the man sees the apple"

<sent>	**(the; p1)**
<pred><sub>	**(the; p2)**
<pred><np>	**(the; p3)**
<pred><noun><art>	**(the; p6)**
<pred><noun>the	**(match the)**
<pred><noun>	**(man p7)**
<pred>man	**(match man)**
<pred>	**(sees; p4)**
<dirobj><verb>	**(sees; p10)**
<dirobj>sees	**(match sees)**
<dirobj>	**(the; p5)**
<noun><art>	**(the; p6)**
<noun>the	**(match the)**
<noun>	**(apple; p8)**
apple	**(match apple)**
	(# success)

2. LR Parser
   a.

Workspace	Input	Action
	int % int	Shift
int	% int	int ::= z
z	% int	Shift
z %	int	Shift
z % int		int ::= z
z % z		z ::= <L>
z % <L>		z ::= <L>
<L> % <L>		<L> ::= <T>
<L> % <T>		<L> ::= <T>
<T> % <T>		<T> % <T> ::= <S>
<S>		

Showing z % z ::= <S>

LR Parse

"the man sees the apple"

shift "the"	**| the**
reduce r6	**|<art>**
shift "man"	**|<art>man**
reduce r7	**|<art><noun>**
reduce r3	**|<np>**
reduce r2	**|<sub>**
shift "sees"	**|<sub>sees**
reduce 10	**|<sub><verb>**
shift "the"	**|<sub ><verb>the**
reduce r6	**|<sub><verb><art>**
shift "apple"	**|<sub><verb><art>apple**
reduce r8	**|<sub><verb><art><noun>**
reduce r5	**|<sub><verb><dirobj>**
reduce r3	**|<sub><pred>**
reduce r1	**|<sentence>**