

PRINCIPLES OF PROGRAMMING LANGUAGES

OBJECTIVES

- To understand and describe syntax and semantics of programming languages
- To understand data, data types, and basic statements
- To understand call-return architecture and ways of implementing them
- To understand object-orientation, concurrency, and event handling in programming languages
- To develop programs in non-procedural programming paradigms

UNIT I

SYNTAX AND SEMANTICS

- **Evolution of programming languages**
- **Describing syntax**
 - **Context-free grammars**
 - **Attribute grammars**
- **Describing semantics**
- **Lexical analysis**
 - **Parsing**
 - **Recursive-decent**
 - **Bottom up parsing**

Improved background for choosing appropriate languages

- C vs. Modula-3 vs. C++ for systems programming
- Fortran vs. APL vs. Ada for numerical computations
- Ada vs. Modula-2 for embedded systems
- Common Lisp vs. Scheme vs. Haskell for symbolic data manipulation
- Java vs. C/CORBA for networked PC programs

Increased ability to learn new languages

www.IntukMaterials.com

- Easy to walk down language family tree
- Concepts are similar across languages
- If you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum
- Analogy to human languages: good grasp of grammar makes it easier to pick up new languages

Increased capacity to express ideas

Figure out how to do things in languages that don't support them:

- lack of suitable control structures in Fortran
- use comments and programmer discipline for control structures
- lack of recursion in Fortran, CSP, etc
- write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)
- lack of named constants and enumerations in Fortran
 - use variables that are initialized once, then never changed
- lack of modules in C and Pascal
 - use comments and programmer discipline
- lack of iterators in just about everything
 - fake them with (member?) functions

What makes a language successful?

- Easy to learn (BASIC, Pascal, LOGO, Scheme)
- Easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
- Easy to implement (BASIC, Forth)
- Possible to compile to very good (fast/small) code (Fortran)
- Backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
- Wide dissemination at minimal cost (Pascal, Turing, Java)

What makes a successful language?

The following key characteristics:

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

www.JntukMaterials.com

Simplicity and Readability

- Small instruction set
 - E.g., Java vs Scheme
- Simple syntax
 - E.g., C/C++/Java vs Python
- Benefits:
 - Ease of learning
 - Ease of programming

www.JntukMaterials.com

Clarity about Binding

A language element is bound to a property at the time that property is defined for it.

So a *binding* is the association between an object and a property of that object

- Examples:
 - a variable and its type
 - a variable and its value
- Early binding *takes place at compile-time*
- Late binding *takes place at run time*

A language is *reliable* if:

- Program behavior is the same on different platforms
 - E.g., early versions of Fortran
- Type errors are detected
 - E.g., C vs Haskell
- Semantic errors are properly trapped
 - E.g., C vs C++
- Memory leaks are prevented
 - E.g., C vs Java

www.JntukMaterials.com
Language Support

- Accessible (public domain) compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)

Abstraction in Programming

- Data
 - Programmer-defined types/classes
 - Class libraries
- Procedural
 - Programmer-defined functions
 - Standard function libraries

A language is *orthogonal* if its features are built upon a small, mutually independent set of primitive operations.

- Fewer exceptional rules = conceptual simplicity
 - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency

Efficient implementation

- Embedded systems
 - Real-time responsiveness (e.g., navigation)
 - Failures of early Ada implementations
- Web applications
 - Responsiveness to users (e.g., Google search)
- Corporate database applications
 - Efficient search and updating
- AI applications
 - Modeling human behaviors

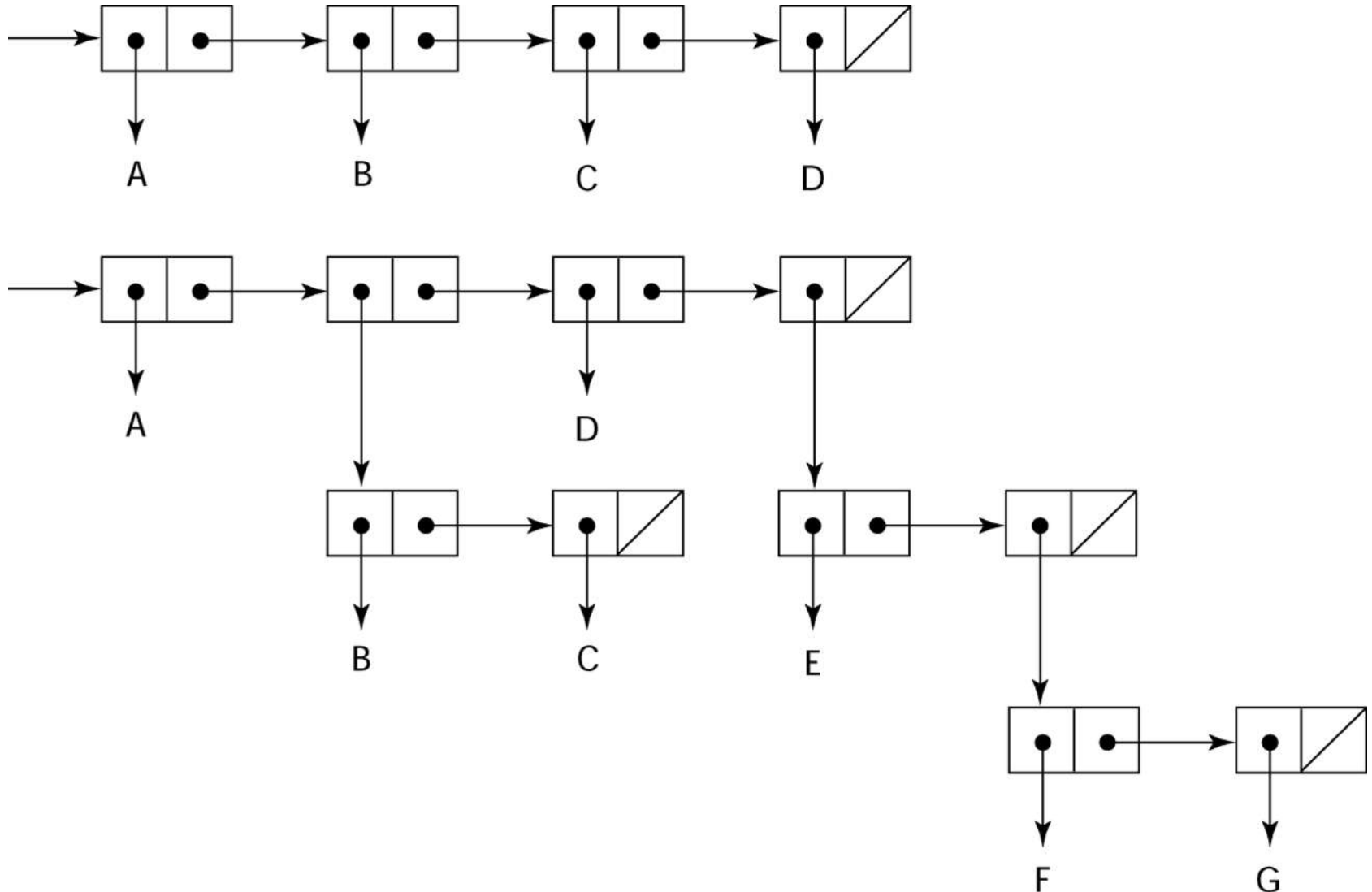
www.JntukMaterials.com What is a language for?

- Why do we have programming languages?
 - way of thinking---way of expressing algorithms
 - languages from the user's point of view
 - abstraction of virtual machine---way of specifying what you want the hardware to do without getting down into the bits
 - languages from the implementor's point of view

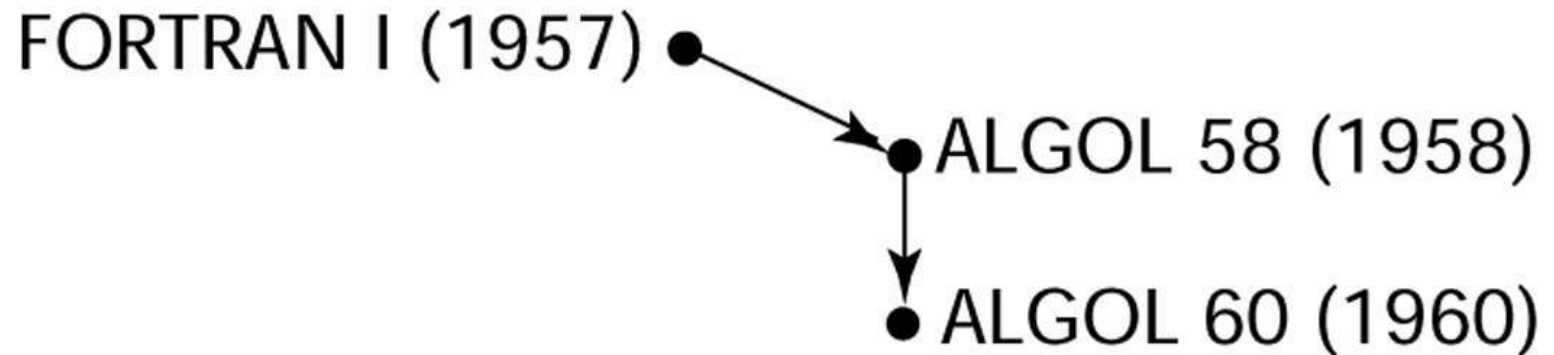
www.JntukMaterials.com



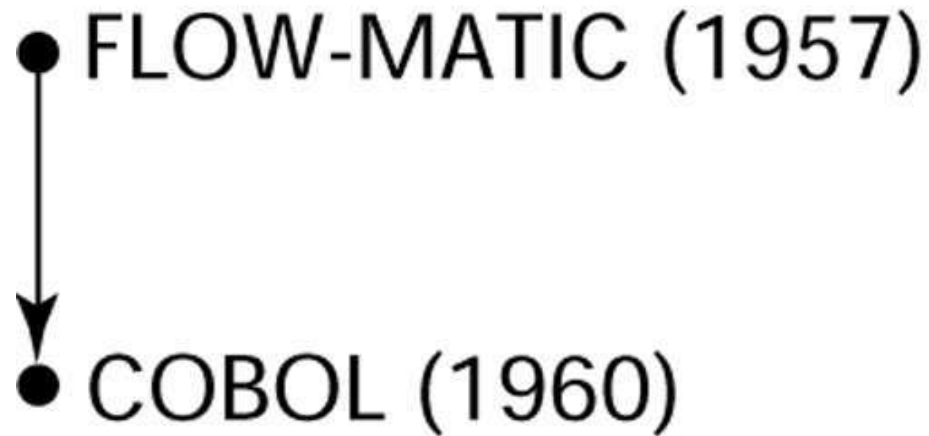
Internal representation of two LISP lists



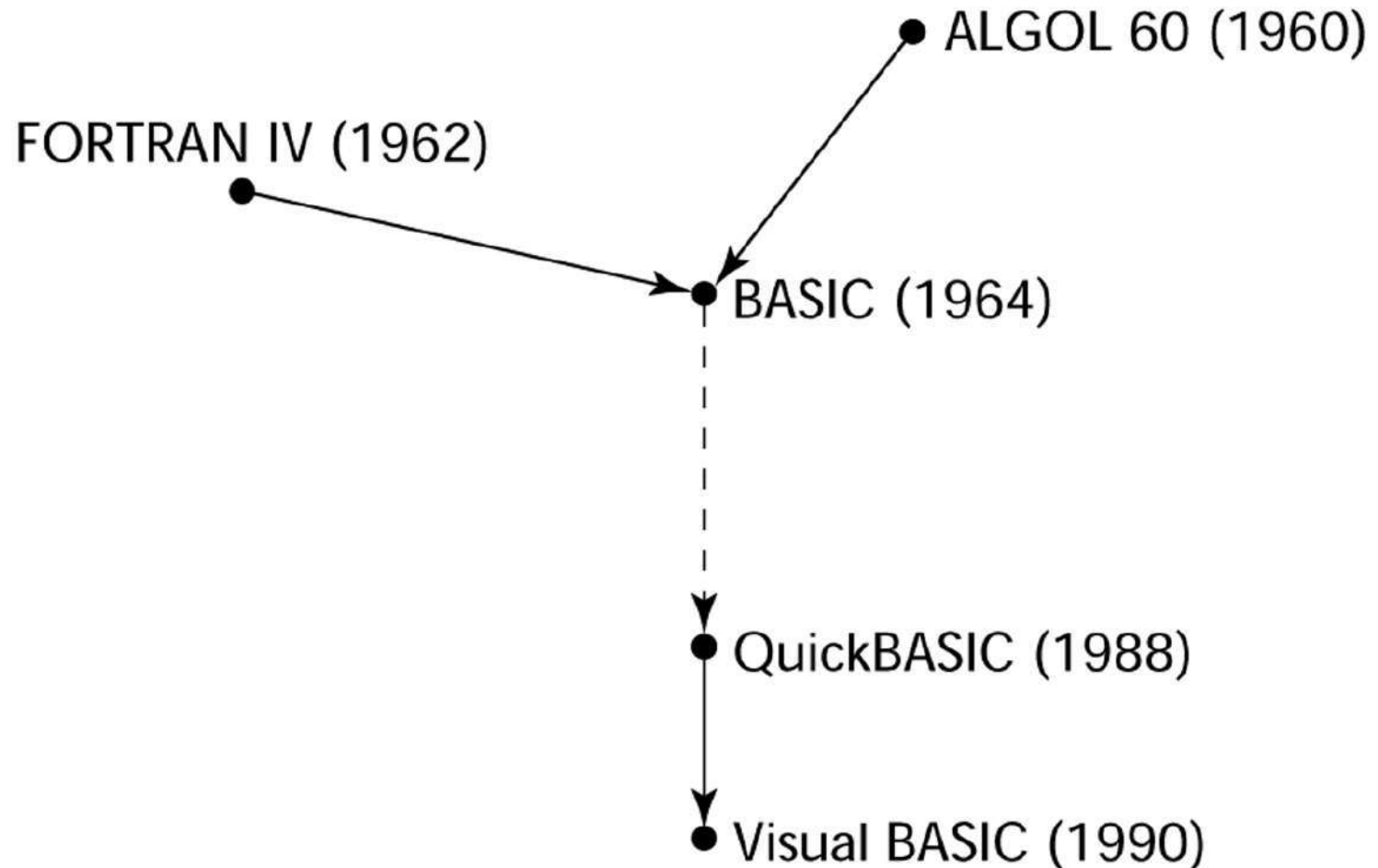
Genealogy of ALGOL 60



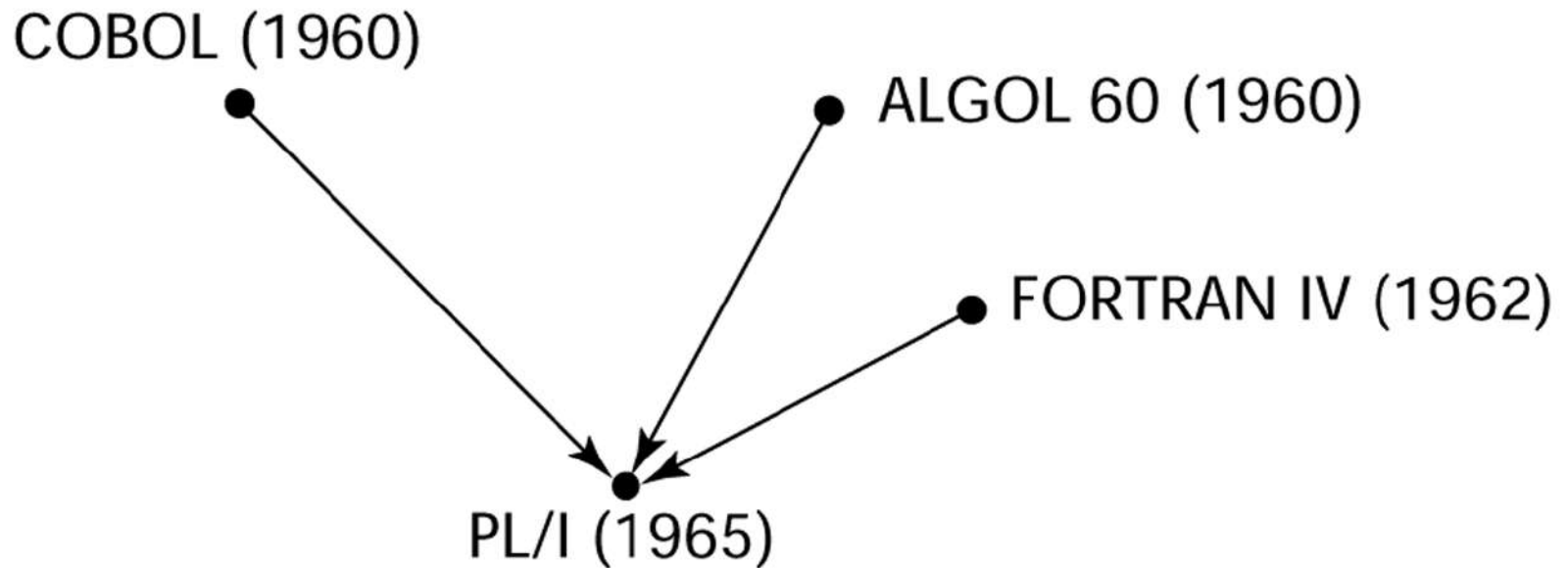
Genealogy of COBOL



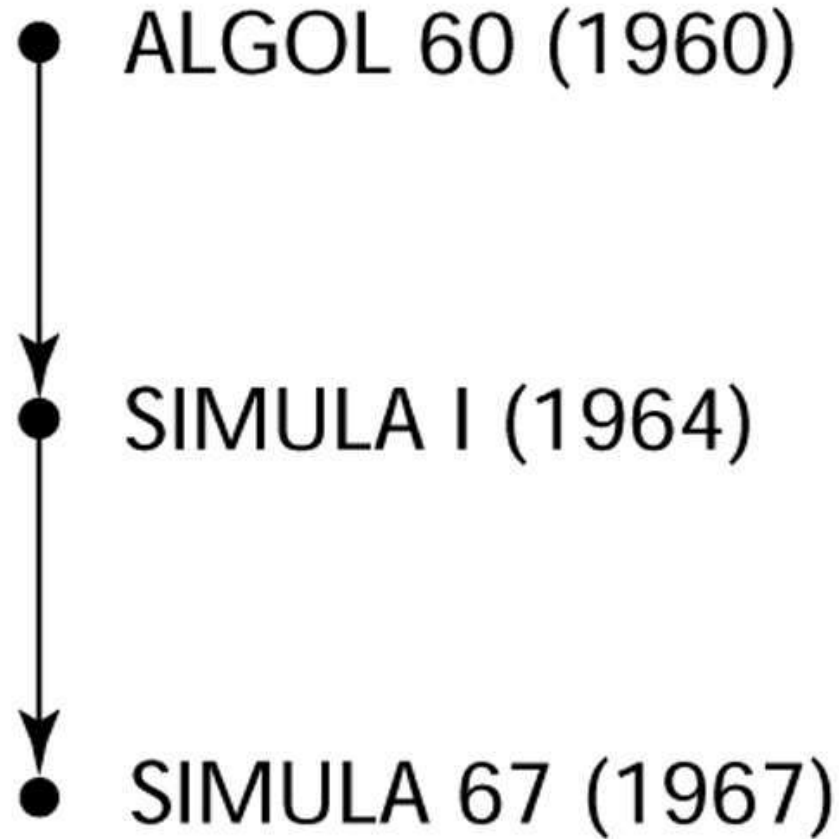
Genealogy of BASIC



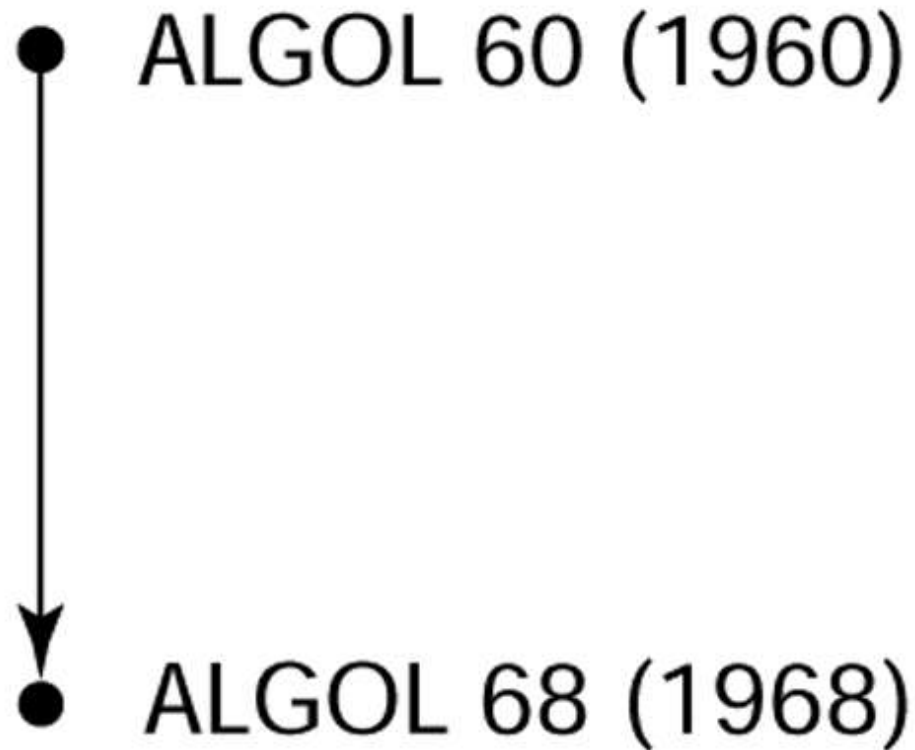
Genealogy of PL/I



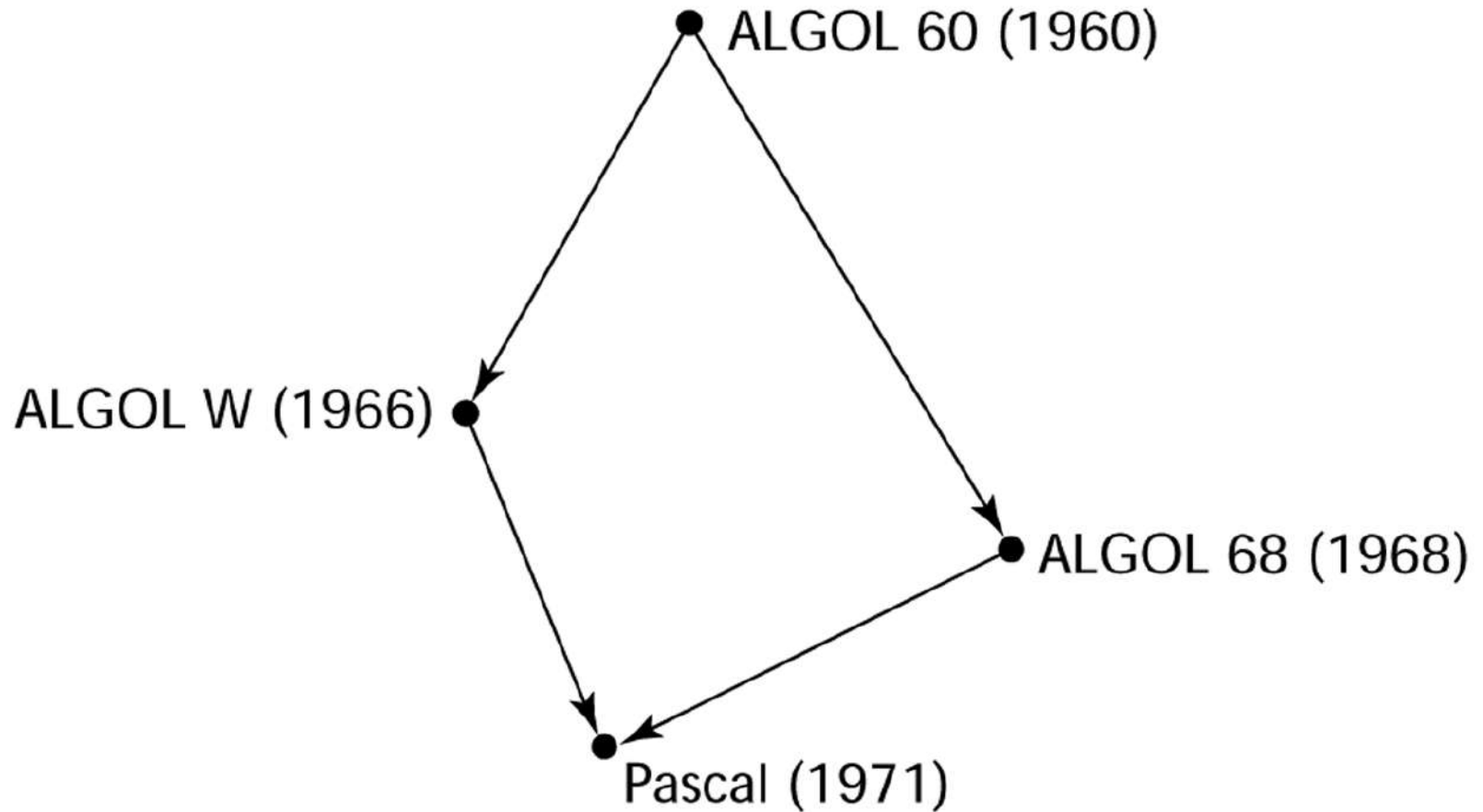
Genealogy of SIMULA 67

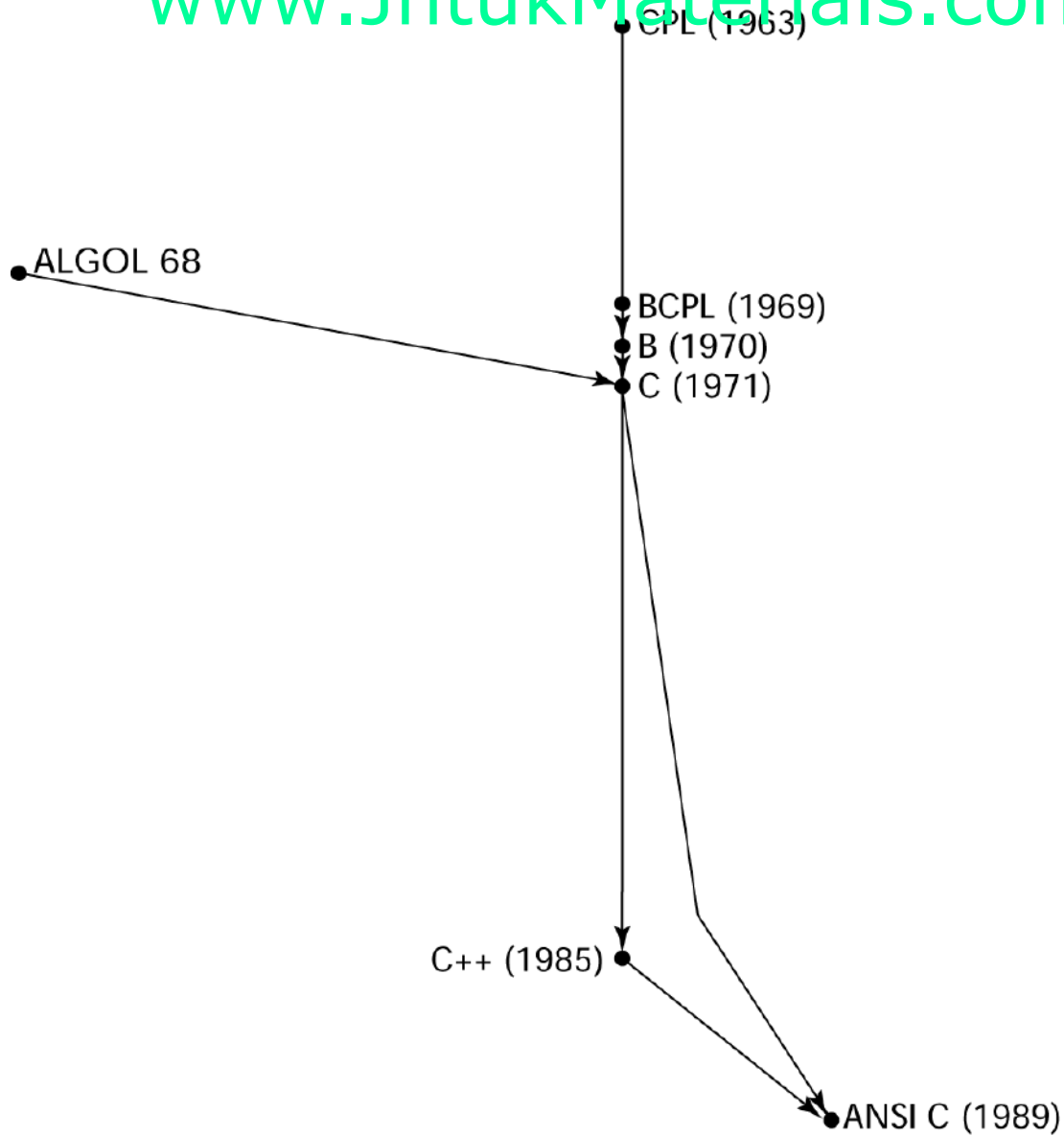


Genealogy of ALGOL 68

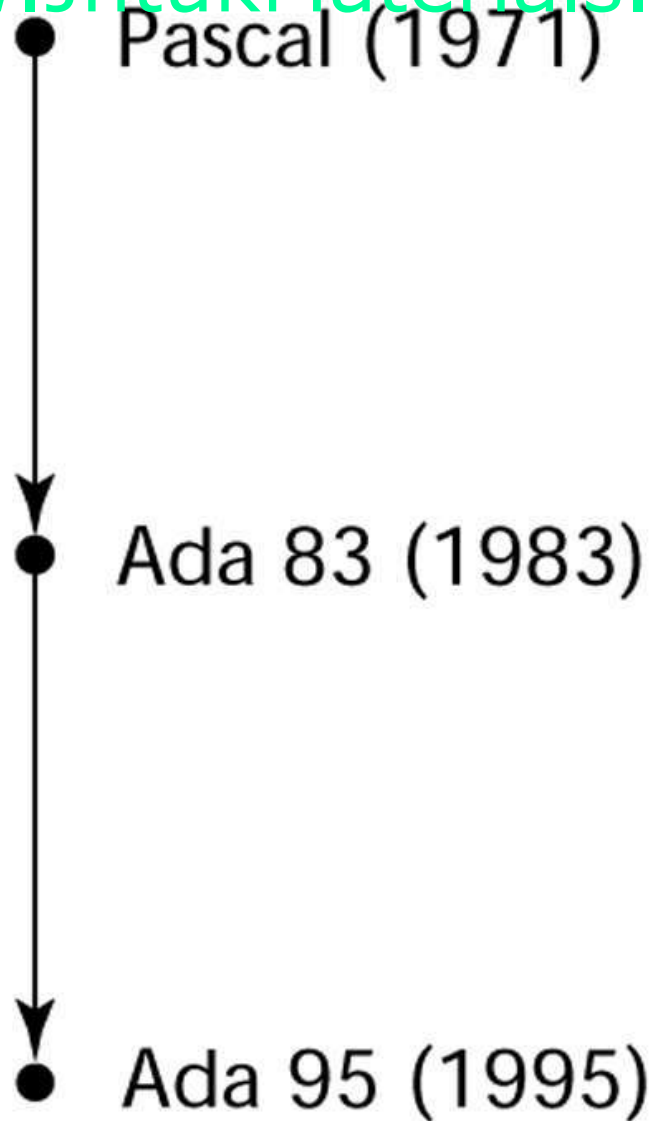


Genealogy of Pascal

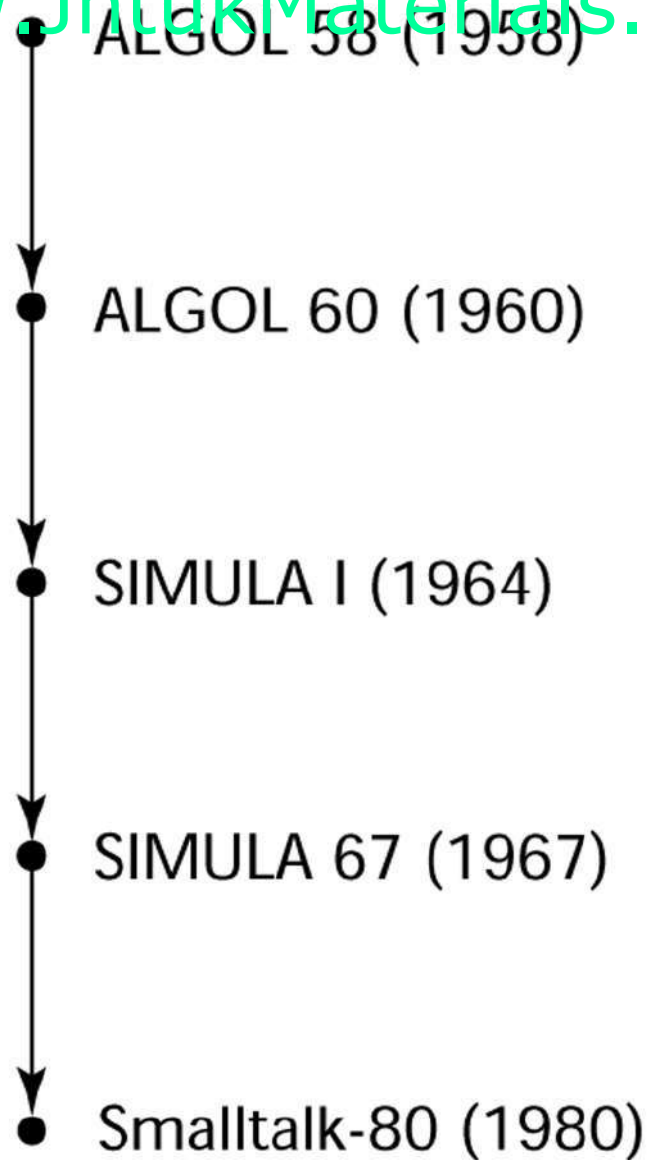




Genealogy of Ada



Genealogy of Smalltalk



www.JntukMaterials.com

History

- Early History : The first programmers
- The 1940s: Von Neumann and Zuse
- The 1950s: The First Programming Language
- The 1960s: An Explosion in Programming languages
- The 1970s: Simplicity, Abstraction, Study
- The 1980s: Consolidation and New Directions
- The 1990s: Internet and the Web
- The 2000s: tbd

www.JntukMaterials.com Early History: The First Programmer

- Jacquard loom of early 1800s
 - Translated card patterns into cloth designs
- Charles Babbage's analytical engine (1830s & 40s)
 - Programs were cards with data and operations
- Ada Lovelace – first programmer
 - “The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; And in fact might bring out its results in algebraic notation, were provision made.”*

The 1940s: Von Neumann and Zuse

- Konrad Zuse (Plankalkul)
 - in Germany - in isolation because of the war
 - defined Plankalkul (program calculus) circa 1945 but never implemented it.
 - Wrote algorithms in the language, including a program to play chess.
 - His work finally published in 1972.
 - Included some advanced data type features such as
 - Floating point, used twos complement and hidden bits
 - Arrays
 - records (that could be nested)

www.JntukMaterials.com

Plankaikul notation

A(7) := 5 * B(6)

		5	*	B	=>	A	
V				6		7	(subscripts)
S				1.n		1.n	(data types)

Machine Code (1940's)

- Initial computers were programmed in raw machine code.
- These were entirely numeric.
- What was wrong with using machine code?
Everything!
 - Poor readability
 - Poor modifiability
 - Expression coding was tedious
 - Inherit deficiencies of hardware, e.g., no indexing or floating point numbers

www.JntukMaterials.com

Pseudocodes (1949)

- Short Code or SHORTCODE - John Mauchly, 1949.
- Pseudocode interpreter for math problems, on Eckert and Mauchly's BINAC and later on UNIVAC I and II.
- Possibly the first attempt at a higher level language.
- Expressions were coded, left to right, e.g.:

`X0 = sqrt(abs(Y0))`

`00 X0 03 20 06 Y0`

- Some operations:

01 –	06 abs	1n (n+2)nd power
02)	07 +	2n (n+2)nd root
03 =	08 pause	4n if <= n
04 /	09 (	58 print & tab

www.JntukMaterials.com More Pseudocodes

Speedcoding; 1953-4

- A pseudocode interpreter for math on IBM 701, IBM 650.
- Developed by John Backus
- Pseudo ops for arithmetic and math functions
- Conditional and unconditional branching
- Autoincrement registers for array access
- Slow but still dominated by slowness of s/w math
- Interpreter left only 700 words left for user program

Laning and Zierler System - 1953

- Implemented on the MIT Whirlwind computer
- First "algebraic" compiler system
- Subscripted variables, function calls, expression translation
- Never ported to any other machine

www.JntukMaterials.com The 1950s: The First Programming Language

- Pseudocodes: interpreters for assembly language like
- Fortran: the first higher level programming language
- COBOL: the first business oriented language
- Algol: one of the most influential programming languages ever designed
- LISP: the first language to depart from the procedural paradigm
- APL:

www.JntukMaterials.com

Fortran (1954-57)

- FORMula TRANslator
- Developed at IBM under the guidance of John Backus primarily for scientific programming
- Dramatically changed forever the way computers used
- Has continued to evolve, adding new features & concepts.
 - FORTRAN II, FORTRAN IV, FORTRAN 66, FORTRAN 77, FORTRAN 90
- Always among the most efficient compilers, producing fast code
- Still popular, e.g. for supercomputers

Fortran 0 and 1

FORTTRAN 0 – 1954 (not implemented)

FORTTRAN I - 1957

Designed for the new IBM 704, which had index registers and floating point hardware

Environment of development:

Computers were small and unreliable

Applications were scientific

No programming methodology or tools

Machine efficiency was most important

Impact of environment on design

- No need for dynamic storage
- Need good array handling and counting loops
- No string handling, decimal arithmetic, or powerful input/output (commercial stuff)

www.JntukMaterials.com

Fortran I Features

- Names could have up to six characters
- Post-test counting loop (DO)
- Formatted I/O
- User-defined subprograms
- Three-way selection statement (arithmetic IF)
IF (ICOUNT-1) 100, 200, 300
- No data typing statements
variables beginning with i, j, k, l, m or n were integers, all else floating point
- No separate compilation
- Programs larger than 400 lines rarely compiled correctly, mainly due to IBM 704's poor reliability
- Code was very fast
- Quickly became widely used

www.JntukMaterials.com Fortran II, IV and 77

FORTRAN II - 1958

- Independent compilation
- Fix the bugs

FORTRAN IV - 1960-62

- Explicit type declarations
- Logical selection (IF) statement
- Subprogram names could be parameters
- ANSI standard in 1966

FORTRAN 77 - 1978

- Character string handling
- Logical loop control (WHILE) statement
- IF-THEN-ELSE statement

Fortran 90 (1990)

Added many features of more modern programming languages, including

- Pointers
- Recursion
- CASE statement
- Parameter type checking
- A collection of array operations, DOTPRODUCT, MATMUL, TRANSPOSE, etc
- dynamic allocations and deallocation of arrays
- a form of records (called derived types)
- Module facility (similar Ada's package)

- COmmon Business Oriented Language
- Principal mentor: (Rear Admiral Dr.) Grace Murray Hopper (1906-1992)
- *Based on FLOW-MATIC* which had such features as:
 - Names up to 12 characters, with embedded hyphens
 - English names for arithmetic operators
 - Data and code were completely separate
 - Verbs were first word in every statement
- CODASYL committee (Conference on Data Systems Languages) developed a programming language by the name of COBOL

First CODASYL Design Meeting - May 1959

Design goals:

- Must look like simple English
- Must be easy to use, even if that means it will be less powerful
- Must broaden the base of computer users
- Must not be biased by current compiler problems

Design committee were all from computer manufacturers and DoD branches

Design Problems: arithmetic expressions? subscripts?
Fights among manufacturers

Contributions:

- First macro facility in a high-level language
- Hierarchical data structures (records)
- Nested selection statements
- Long names (up to 30 characters), with hyphens
- Data Division

Comments:

- First language required by DoD; would have failed without DoD
- Still the most widely used business applications language

www.IntukMaterials.com BASIC (1964)

- Beginner's All purpose Symbolic Instruction Code
- Designed by Kemeny & Kurtz at Dartmouth for the GE 225 with the goals:
 - Easy to learn and use for non-science students and as a path to Fortran and Algol
 - Must be "pleasant and friendly"
 - Fast turnaround for homework
 - Free and private access
 - User time is more important than computer time
- Well-suited for implementation on first PCs, e.g., Gates and Allen's 4K Basic interpreter for the MITS Altair personal computer (circa 1975)
- Current popular dialects: Visual BASIC

- LISt Processing language (Designed at MIT by McCarthy)
- *AI research needed a language that:*
 - Process data in lists (rather than arrays)
 - Handles symbolic computation (rather than numeric)
- One universal, recursive data type: the s-expression
 - An s-expression is either an atom or a list of zero or more s-expressions
- Syntax is based on the lambda calculus
- *Pioneered functional programming*
 - No need for variables or assignment
 - Control via recursion and conditional expressions
- Status
 - Still the dominant language for AI
 - COMMON LISP and Scheme are contemporary dialects
 - ML, Miranda, and Haskell are related languages

Environment of development:

1. FORTRAN had (barely) arrived for IBM 70x
2. Many other languages were being developed, all for specific machines
3. No portable language; all were machine-dependent
4. No universal language for communicating algorithms

ACM and GAMM met for four days for design

- *Goals of the language:*

1. Close to mathematical notation
2. Good for describing algorithms
3. Must be translatable to machine code

www.JntukMaterials.com Algol 58 Features

- Concept of type was formalized
- Names could have any length
- Arrays could have any number of subscripts
- Parameters were separated by mode (in & out)
- Subscripts were placed in brackets
- Compound statements (begin ... end)
- Semicolon as a statement separator
- Assignment operator was :=
- if had an else-if clause

Comments:

- Not meant to be implemented, but variations of it were (MAD, JOVIAL)
- Although IBM was initially enthusiastic, all support was dropped by mid-1959

Modified ALGOL 58 at 6-day meeting in Paris adding such new features as:

- Block structure (local scope)
- Two parameter passing methods
- Subprogram recursion
- Stack-dynamic arrays
- Still no I/O and no string handling

Successes:

- The standard way to publish algorithms for over 20 years
- All subsequent imperative languages are based on it
- First machine-independent language
- First language whose syntax was formally defined (BNF)

www.JntukMaterials.com

Algol 60 (1960)

Failure: Never widely used, especially in U.S., mostly because

1. No I/O and the character set made programs nonportable
2. Too flexible--hard to implement
3. Entrenchment of FORTRAN
4. Formal syntax description
5. Lack of support by IBM

- A Programming Language
- Designed by K.Iverson at Harvard in late 1950's
- A language for programming mathematical computations
 - especially those using matrices
- Functional style and many whole array operations
- Drawback is requirement of special keyboard

The 1960s: An Explosion in Programming Languages

- The development of hundreds of programming languages
- PL/I designed in 1963-4
 - supposed to be all purpose
 - combined features of FORTRAN, COBOL and Algol 60 and more!
 - translators were slow, huge and unreliable
 - some say it was ahead of its time.....
- Algol 68
- SNOBOL
- Simula
- BASIC

- Computing situation in 1964 (IBM's point of view)
 - Scientific computing
 - IBM 1620 and 7090 computers
 - FORTRAN
 - SHARE user group
 - Business computing
 - IBM 1401, 7080 computers
 - COBOL
 - GUIDE user group
- IBM's goal: develop a single computer (IBM 360) and a single programming language (PL/I) that would be good for scientific and business applications.
- Eventually grew to include virtually every idea in current practical programming languages.

PL/I contributions:

1. First unit-level concurrency
2. First exception handling
3. Switch-selectable recursion
4. First pointer data type
5. First array cross sections

Comments:

- Many new features were poorly designed
- Too large and too complex
- Was (and still is) actually used for both scientific and business applications
- Subsets (e.g. PL/C) developed which were more manageable

- Designed and built by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Centre (NCC) in Oslo between 1962 and 1967
- Originally designed and implemented as a language for discrete event simulation
- Based on ALGOL 60

Primary Contributions:

- Coroutines - a kind of subprogram
 - Classes (data plus methods) and objects
 - Inheritance
 - Dynamic binding
- => Introduced the basic ideas that developed into object-oriented programming.

Algol 68

From the continued development of ALGOL 60, but it is not a superset of that language

- Design is based on the concept of orthogonality
- *Contributions:*
 - User-defined data structures
 - Reference types
 - Dynamic arrays (called flex arrays)
- *Comments:*
 - Had even less usage than ALGOL 60
 - Had strong influence on subsequent languages, especially Pascal, C, and Ada

The 1970s: Simplicity, Abstraction, Study

- Algol-W - Nicklaus Wirth and C.A.R.Hoare
 - reaction against 1960s
 - simplicity
- Pascal
 - small, simple, efficient structures
 - for teaching program
- C - 1972 - Dennis Ritchie
 - aims for simplicity by reducing restrictions of the type system
 - allows access to underlying system
 - interface with O/S - UNIX

www.JntukMaterials.com Pascal (1971)

- Designed by Wirth, who quit the ALGOL 68 committee (didn't like the direction of that work)
- Designed for teaching structured programming
- Small, simple
- Introduces some modest improvements, such as the case statement
- Was widely used for teaching programming ~ 1980-1995.

- Designed for systems programming at Bell Labs by Dennis Ritchie and colleagues.
- Evolved primarily from B, but also ALGOL 68
- Powerful set of operators, but poor type checking
- Initially spread through UNIX and the availability of high quality, free compilers, especially gcc.

Other descendants of ALGOL

- **Modula-2** (mid-1970s by Niklaus Wirth at ETH)
 - Pascal plus modules and some low-level features designed for systems programming
- **Modula-3** (late 1980s at Digital & Olivetti)
 - Modula-2 plus classes, exception handling, garbage collection, and concurrency
- **Oberon** (late 1980s by Wirth at ETH)
 - Adds support for OOP to Modula-2
 - Many Modula-2 features were deleted (e.g., for statement, enumeration types, with statement, non-integer array indices)

The 1980s: Consolidation and New Paradigms

- Ada
 - US Department of Defence
 - European team lead by Jean Ichbiah. (Sam Lomonaco was also on the ADA team)
- Functional programming
 - Scheme, ML, Haskell
- Logic programming
 - Prolog
- Object-oriented programming
 - Smalltalk, C++, Eiffel

- In study done in 73-74 it was determined that the DoD was spending \$3B annually on software, over half on embedded computer systems.
- The Higher Order Language Working Group was formed and initial language requirements compiled and refined in 75-76 and existing languages evaluated.
- In 1997, it was concluded that none were suitable, though Pascal, ALGOL 68 or PL/I would be a good starting point.
- Language DoD-1 was developed through a series of competitive contracts.

- Renamed Ada in May 1979.
- Reference manual, Mil. Std. 1815 approved 10 December 1980. (Ada Bryon was born 10/12/1815)
- “mandated” for use in DoD work during late 80’s and early 90’s.
- Ada95, a joint ISO and ANSI standard, accepted in February 1995 and included many new features.
- The Ada Joint Program Office (AJPO) closed 1 October 1998 (Same day as ISO/IEC 14882:1998 (C++) published!)

Contributions:

1. Packages - support for data abstraction
2. Exception handling - elaborate
3. Generic program units
4. Concurrency - through the tasking model

Comments:

- Competitive design
- Included all that was then known about software engineering and language design
- First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed
- Very difficult to mandate programming technology

www.IntukMaterials.com

Logic Programming: Prolog

- Developed at the University of Aix Marseille, by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries

Functional Programming

- **Common Lisp:** consolidation of LISP dialects spurred practical use, as did the development of Lisp Machines.
- **Scheme:** a simple and pure LISP like language used for teaching programming.
- **Logo:** Used for teaching young children how to program.
- **ML:** (MetaLanguage) a strongly-typed functional language first developed by Robin Milner in the 70's
- **Haskell:** polymorphically typed, lazy, purely functional language.

www.IntukMaterials.com Smalltalk (1972-80)

- Developed at Xerox PARC by Alan Kay and colleagues (esp. Adele Goldberg) inspired by Simula 67
- First compilation in 1972 was written on a bet to come up with "the most powerful language in the world" in "a single page of code".
- In 1980, Smalltalk 80, a uniformly object-oriented programming environment became available as the first commercial release of the Smalltalk language
- Pioneered the graphical user interface everyone now uses
- Industrial use continues to the present day

www.JntukMaterials.com C++ (1985)

- Developed at Bell Labs by Stroustrup
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67, added to C
- Also has exception handling
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November, 1997

Eiffel

- Eiffel - a related language that supports OOP
 - (Designed by Bertrand Meyer - 1992)
 - Not directly derived from any other language
 - Smaller and simpler than C++, but still has most of the power

1990's: the Internet and Web

During the 90's, Object-oriented languages (mostly C++) became widely used in practical applications

The Internet and Web drove several phenomena:

- Adding concurrency and threads to existing languages
- Increased use of scripting languages such as Perl and Tcl/Tk
- Java as a new programming language

- Developed at Sun in the early 1990s with original goal of a language for embedded computers
- Principals: Bill Joy, James Gosling, Mike Sheradin, Patrick Naughton
- Original name, Oak, changed for copyright reasons
- Based on C++ but significantly simplified
- Supports *only* OOP
- Has references, but not pointers
- Includes support for applets and a form of concurrency (i.e. threads)

- In the 60's, the dream was a single all-purpose language (e.g., PL/I, Algol)
- The 70s and 80s dream expressed by Winograd (1979)

“Just as high-level languages allow the programmer to escape the intricacies of the machine, higher level programming systems can provide for manipulating complex systems. We need to shift away from algorithms and towards the description of the properties of the packages that we build. Programming systems will be declarative not imperative”
- Will that dream be realised?
- Programming is not yet obsolete

- ROLE OF THE LEXICAL ANALYZER
 - The main function is to read the input and produce the output as a sequence of tokens that the parser uses for syntax analysis
 - The command namely “get next token” is used by the lexical analyzer to read the input characters until it can identify the next token
 - It also performs the user interface task’s
 - It also correlate error messages from compiler. The two phases of LA are
 - Scanning (simple task)
 - Lexical Analysis (complex task)

Tokens Patterns and Lexemes

- Token represents a logically cohesive sequence of characters
- The set of string is described by a rule called pattern associated with the token
- The character sequence forming a token is called lexeme for the token
- Tokens are keywords, operators, identifiers, constants and punctuations
- Pattern is a rule describing the set of lexeme that can represent a particular token in the program
- Lexeme matched by the pattern for the token represents strings of characters

TOKEN	LEXEME	PATTERN
const	const	const
Relation	<, <=, =, >, >=, <>	< or <= or = or > or >= or <>
Num	3.14, 6.2	Any constant
Id	Pi, count	Letter followed by letters and digits

Specification of Patterns for Tokens: Regular Definitions

- Example:

letter $\rightarrow$ **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**

digit $\rightarrow$ **0** | **1** | ... | **9**

id $\rightarrow$ **letter** (**letter** | **digit**)^{*}

- We frequently use the following shorthands:

$$r^+ = rr^*$$

$$r? = r \mid \varepsilon$$

$$[\mathbf{a-z}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots \mid \mathbf{z}$$

- For example:

$$\text{digit} \rightarrow [0-9]$$

$$\text{num} \rightarrow \text{digit}^+ (. \text{digit}^+)? (\text{E} (+|-)? \text{digit}^+)?$$

Regular Definitions and Grammars

www.JntukMaterials.com

Grammar

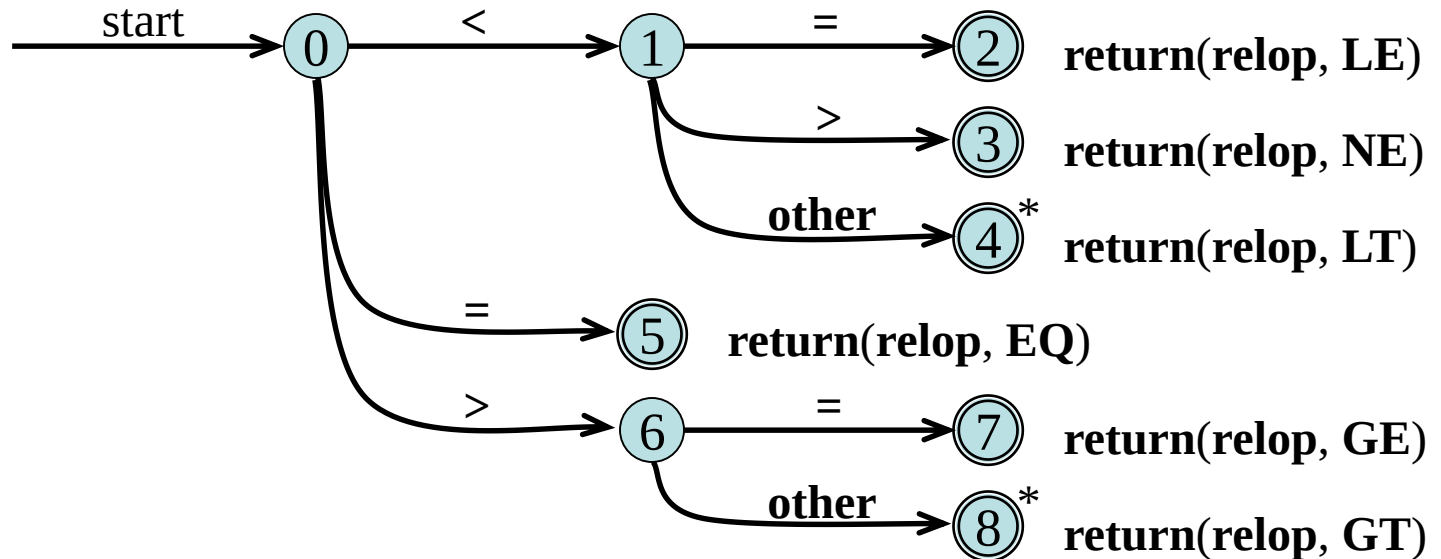
$stmt \rightarrow \mathbf{if} \ expr \ \mathbf{then} \ stmt$
 $\quad | \ \mathbf{if} \ expr \ \mathbf{then} \ stmt \ \mathbf{else} \ stmt$
 $\quad | \ \epsilon$
 $expr \rightarrow term \ \mathbf{relop} \ term$
 $\quad | \ term$
 $term \rightarrow \mathbf{id}$
 $\quad | \ \mathbf{num}$

Regular definitions

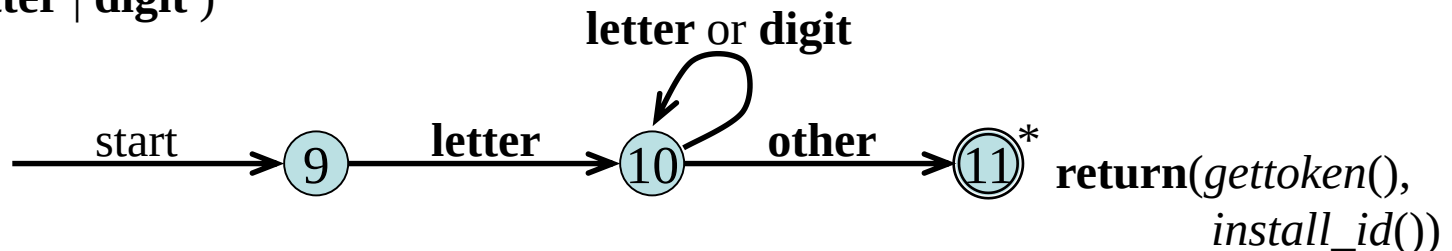
$\mathbf{if} \rightarrow \mathbf{if}$
 $\mathbf{then} \rightarrow \mathbf{then}$
 $\mathbf{else} \rightarrow \mathbf{else}$
 $\mathbf{relop} \rightarrow < \mid <= \mid <> \mid > \mid >= \mid =$
 $\mathbf{id} \rightarrow \mathbf{letter} \ (\ \mathbf{letter} \mid \mathbf{digit} \)^*$
 $\mathbf{num} \rightarrow \mathbf{digit}^+ \ (\ . \ \mathbf{digit}^+)? \ (\ \mathbf{E} \ (+ \mid -)? \ \mathbf{digit}^+)?$

Implementing a Scanner Using Transition Diagrams

relop $\rightarrow$ < | <= | <> | > | >= | =



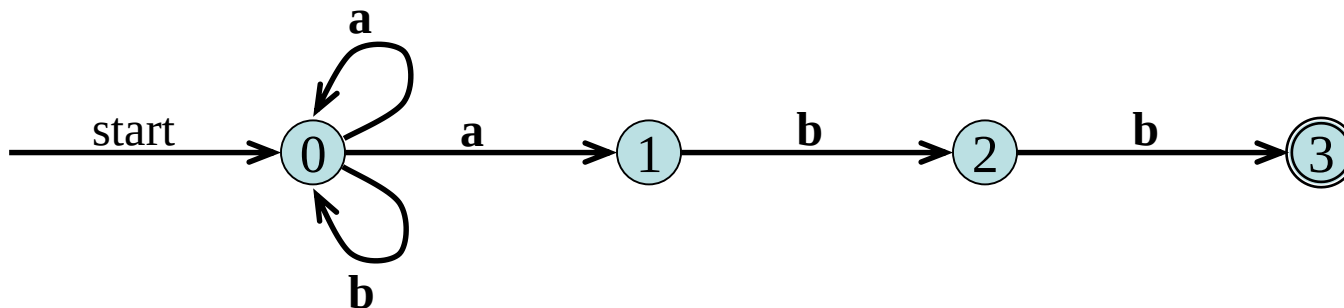
id $\rightarrow$ letter (letter | digit)*



www.JntukMaterials.com

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$S = \{0,1,2,3\}$

$\Sigma = \{a,b\}$

$s_0 = 0$

$F = \{3\}$

Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$$\delta(0, \mathbf{a}) = \{0, 1\}$$

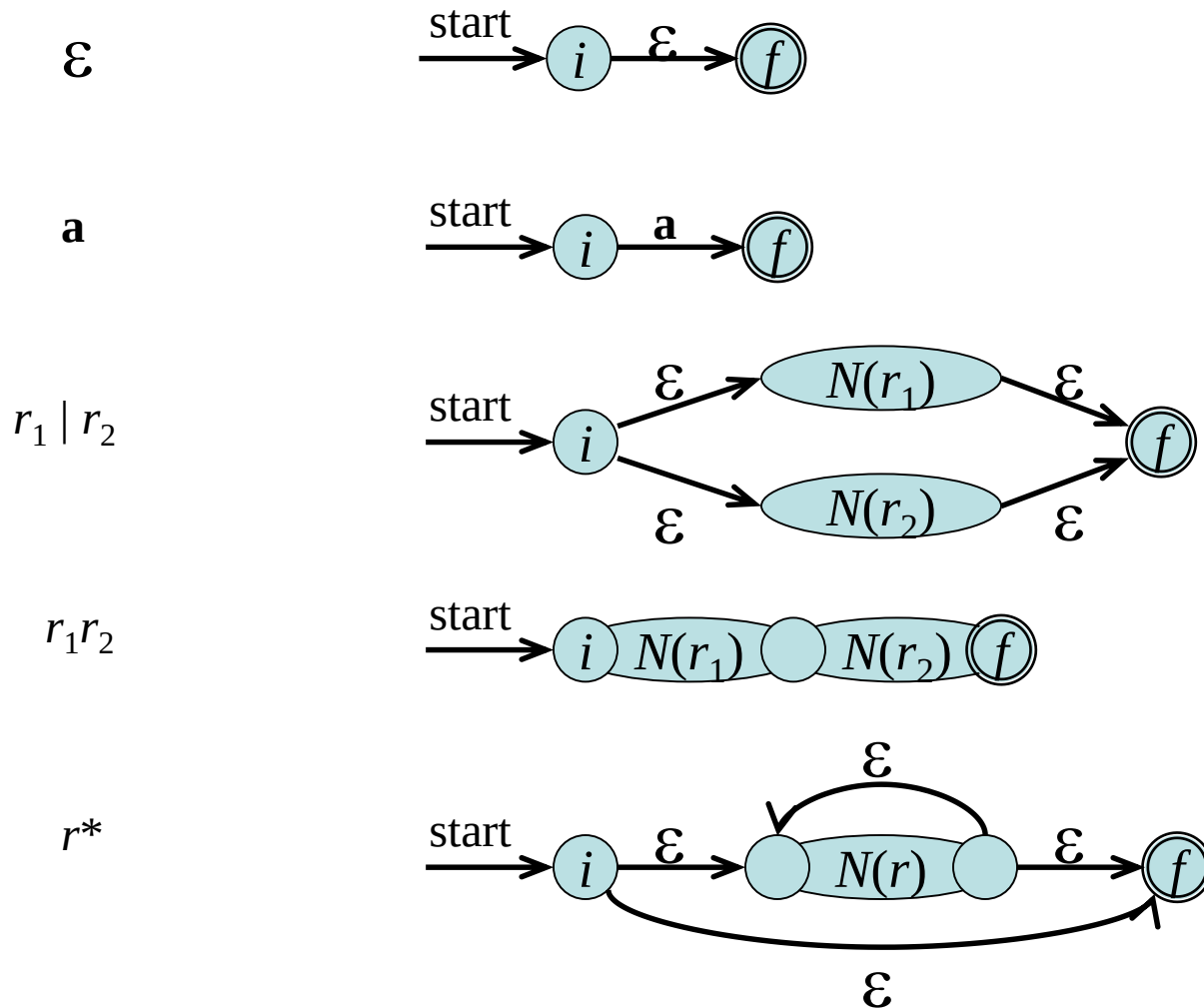
$$\delta(0, \mathbf{b}) = \{0\}$$

$$\delta(1, \mathbf{b}) = \{2\}$$

$$\delta(2, \mathbf{b}) = \{3\}$$

<i>State</i>	<i>Input a</i>	<i>Input b</i>
0	{0, 1}	{0}
1		{2}
2		{3}

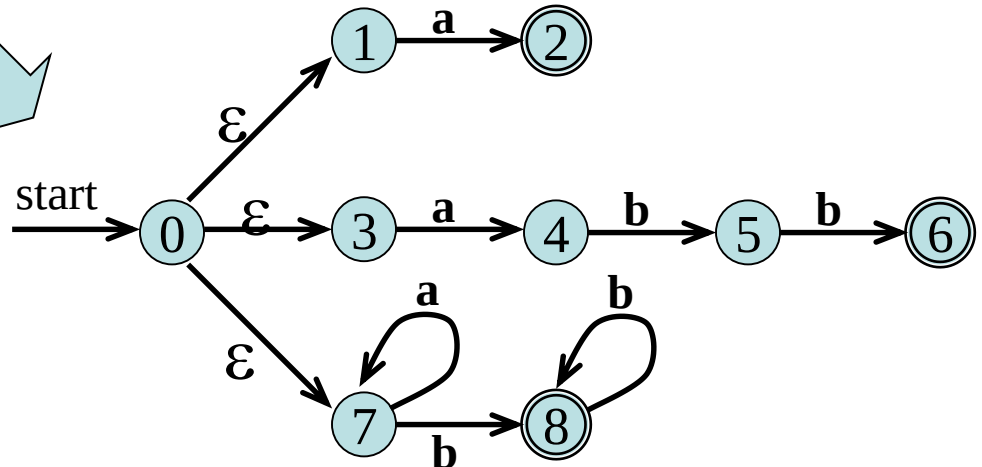
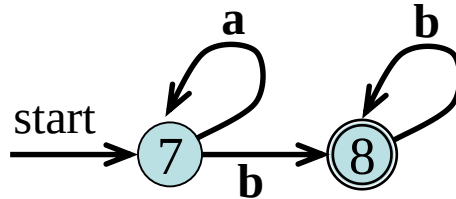
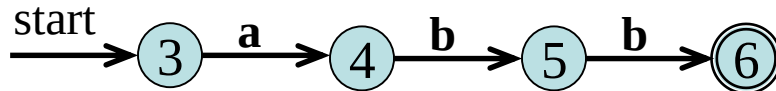
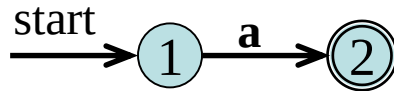
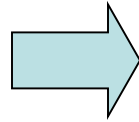
From Regular Expression to NFA (Thompson's Construction)



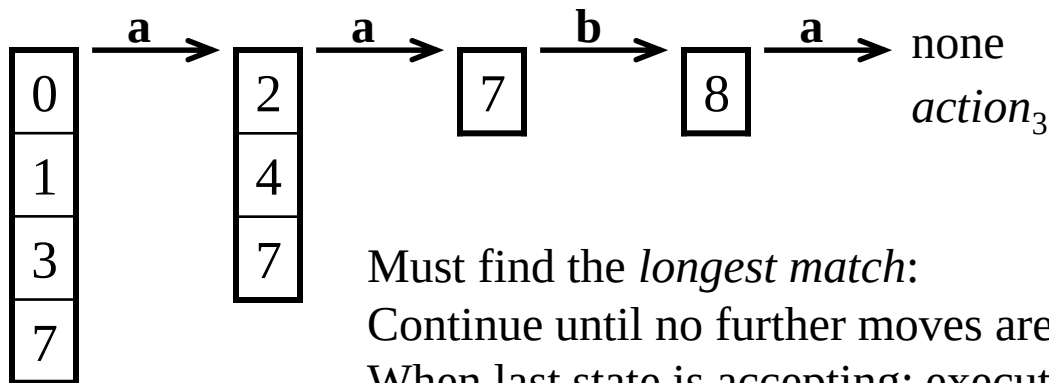
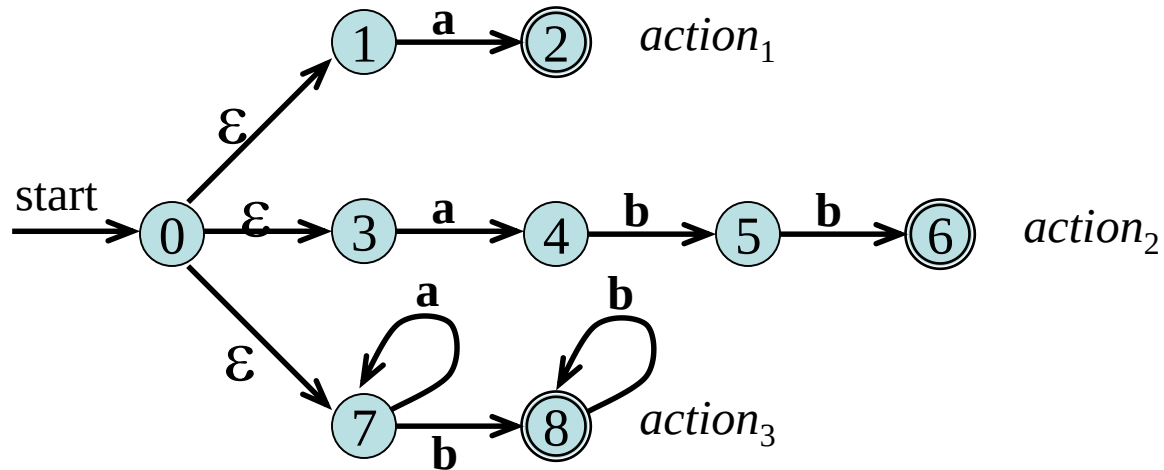
Combining the NFAs of a Set of Regular Expressions

www.JntukMaterials.com

a $\{ action_1 \}$
abb $\{ action_2 \}$
a*b+ $\{ action_3 \}$



Simulating the Combined NFA Example 1

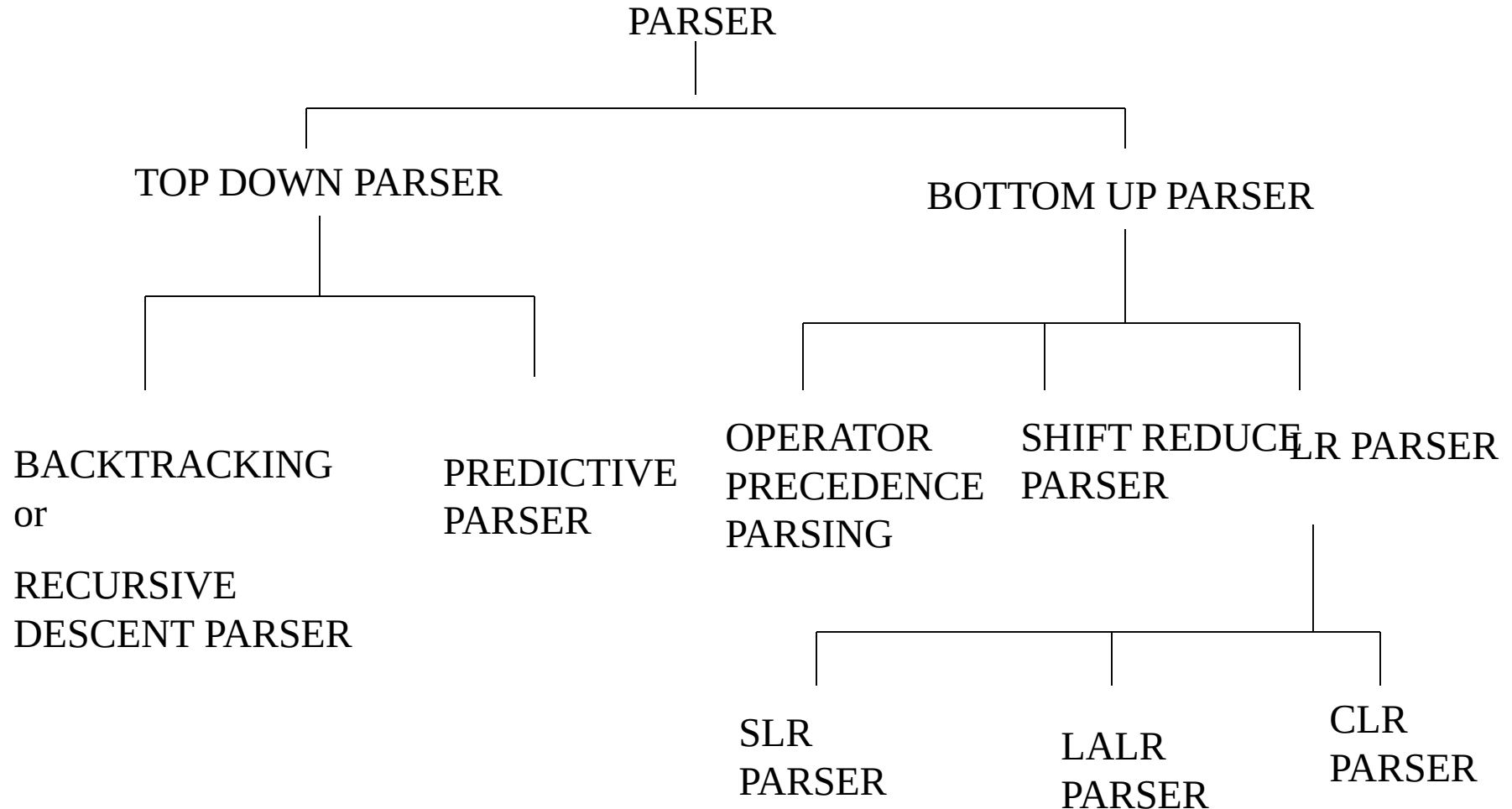


Must find the *longest match*:

Continue until no further moves are possible

When last state is accepting: execute action

PARSING TECHNIQUES



TOP DOWN VS BOTTOM UP

SNo	TOP DOWN PARSER	BOTTOM UP PARSER
1	Parse tree can be built from root to leaves	Parse tree can be built from leaves to root
2	This is simple to implement	This is complex
3	Less efficient. Various problems that occurs during top down techniques are ambiguity, left recursion, left factoring	When the bottom up parser handles ambiguous grammar conflicts occur in parse table
4	It is applicable to small class of languages	It is applicable to a broad class of languages
5	Parsing techniques i. Recursive descent parser ii. Predictive parser	Parsing techniques. i. shift reduce, ii. Operator precedence, iii. LR parser

RECURSIVE DESCENT PARSER

- A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent parser
- The CFG is used to build the recursive routines
- The RHS of the production rule is directly converted to a program.
- For each NT a separate procedure is written and body of the procedure is RHS of the corresponding NT.

Basic steps of construction of RD Parser

- The RHS of the rule is directly converted into program code symbol by symbol
 1. If the input symbol is NT then a call to the procedure corresponding the non-terminal is made.
 2. If the input is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol
 3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
 4. The parser should be activated by a procedure corresponding to the start symbol.

Example www.JntukMaterials.com

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

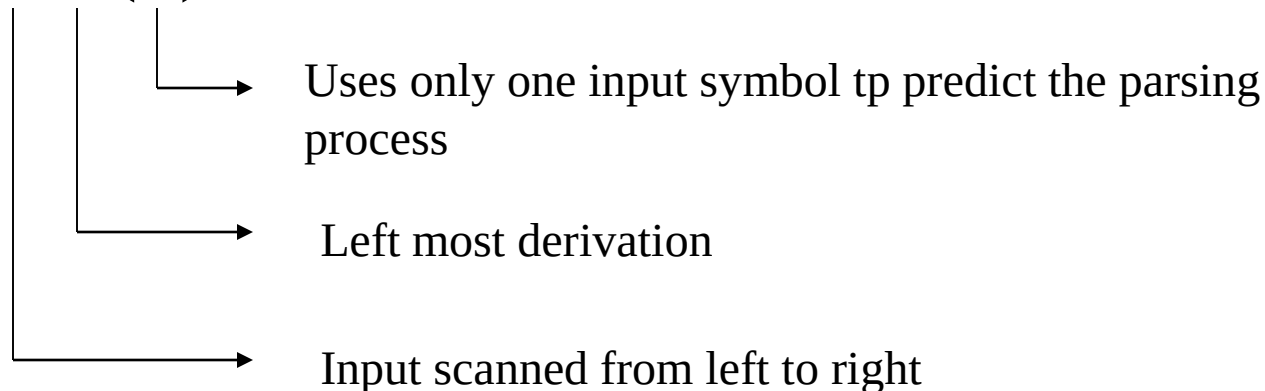
```
proc A {  
  case of the current token {  
    a: - match the current token with a,  
        and move to the next token;  
        - call B;  
        - match the current token with e,  
        and move to the next token;  
    c: - match the current token with c,  
        and move to the next token;  
        - call B;  
        - match the current token with d,  
        and move to the next token;  
    f: - call C  
  }  
}
```

```
proc C {  match the current token with f,  
          and move to the next token; }
```

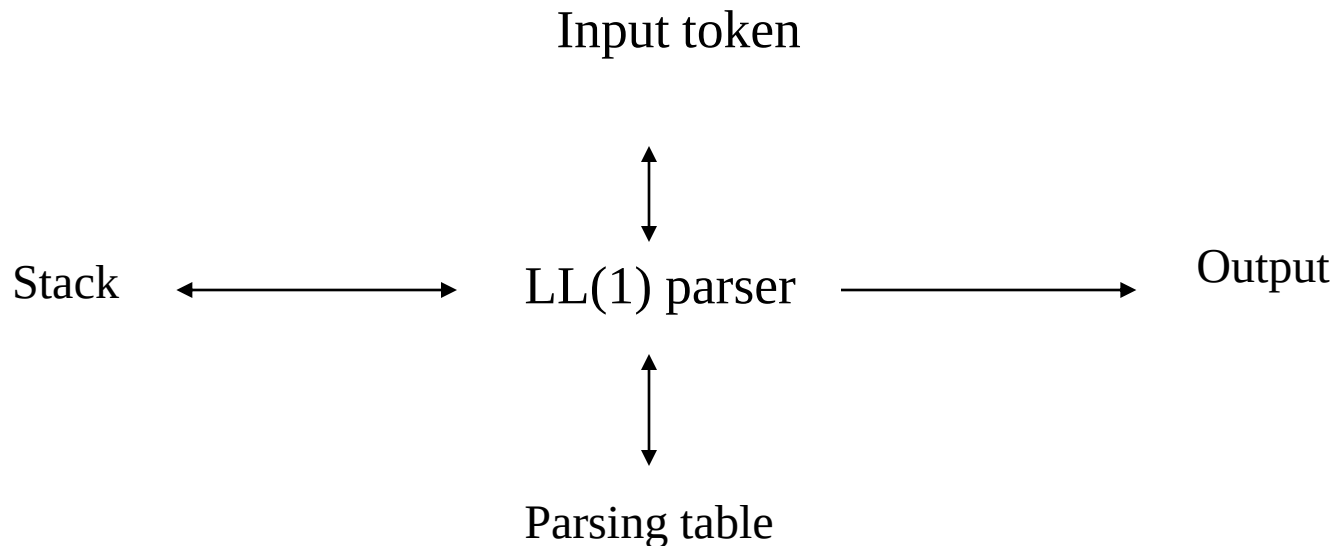
```
proc B {  
  case of the current token {  
    b: - match the current token with b,  
        and move to the next token;  
        - call B  
     $\varepsilon$  : do nothing  
  }  
}
```

Predictive Parsing - LL(1) Parser

- This top-down parsing algorithm is of non-recursive type.
- In this type parsing table is built
- For LL(1)



- The data structures used by LL(1) are
 - Input buffer (store the input tokens)
 - Stack (hold left sentential form)
 - Parsing table (row of NT, column of T)



input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol \$.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol \$.
- initially the stack contains only the symbol \$ and the starting symbol S.
\$S ← initial stack
- when the stack is emptied (ie. only \$ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array $M[A,a]$
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol \$
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.
 1. If X and a are $\$$ $\rightarrow$ parser halts (successful completion)
 2. If X and a are the same terminal symbol (different from $\$$)
 $\rightarrow$ parser pops X from the stack, and moves the next symbol in the input buffer.
 3. If X is a non-terminal
 $\rightarrow$ parser looks at the parsing table entry $M[X,a]$. If $M[X,a]$ holds a production rule $X \rightarrow Y_1 Y_2 \dots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \dots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \dots Y_k$ to represent a step of the derivation.
 4. none of the above $\rightarrow$ error
 - all empty entries in the parsing table are errors.
 - If X is a terminal symbol different from a , this is also an error case.

- The construction of predictive LL(1) parser is based on two very important functions and those are FIRST and FOLLOW.
- For the construction
 1. Computation of FIRST and FOLLOW function
 2. Construction the predictive parsing table using FIRST and FOLLOW functions
 3. Parse the input string with the help of predictive parsing table

FIRST function

- $\text{FIRST}(\alpha)$ is a set of terminal symbols that are first symbols appearing at RHS in derivation of α .
- Following are the rules used to compute the FIRST functions
 1. if the terminal symbol a then $\text{FIRST}(a) = \{a\}$
 2. If there is a rule $X \rightarrow \epsilon$ then $\text{FIRST}(X) = \{\epsilon\}$
 3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$, then place a in $\text{First}(X)$ if for some i a is in $\text{First}(Y_i)$ and ϵ is in all of $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ that is $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$. if ϵ is in $\text{First}(Y_j)$ for $j=1, \dots, k$ then add ϵ to $\text{First}(X)$.

- FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A.
- $\text{FOLLOW}(A) = \{ a \mid S \rightarrow \alpha A a \beta \text{ where } \alpha \text{ and } \beta \text{ are some grammar symbols may be terminal or non-terminal} \}$
- The rules for computing FOLLOW function are as given below –
 1. For the start symbol S place \$ in FOLLOW(S)
 2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without ϵ is to be placed in FOLLOW(B)
 3. If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and FIRST(β) = $\{\epsilon\}$ then FOLLOW(A) = FOLLOW(B) or FOLLOW(B) = FOLLOW(A). That means everything in FOLLOW(A) is in FOLLOW(B)

FIRST AND FOLLOW EXAMPLE

$E \rightarrow TE'$; $E' \rightarrow +TE' | \epsilon$; $T \rightarrow FT'$;
 $T' \rightarrow *FT' | \epsilon$; $F \rightarrow (E) | id$.

- $E \rightarrow TE'$; $T \rightarrow FT'$; $F \rightarrow (E) | id$.
- $FIRST(E) = FIRST(T) = FIRST(F)$
- Here, $F \rightarrow (E)$ and $F \rightarrow id$
- So, $FIRST(F) = \{ (, id \}$
- $FIRST(E') = \{ +, \epsilon \}$ since, $E' \rightarrow +TE' | \epsilon$;
- $FIRST(T') = \{ *, \epsilon \}$ since, $T' \rightarrow *FT' | \epsilon$;

- FOLLOW(E)
 - For $F \rightarrow (E)$
 - As there is $F \rightarrow (E)$, symbol $)$ appears immediately after E . so $)$ will be in FOLLOW(E)
 - By rule $A \rightarrow \alpha B \beta$, we can map this with $F \rightarrow (E)$ then, FOLLOW(E)=FIRST() $= \{ \}$
 - Since E is a start symbol, $\$$ will be in FOLLOW(E)
 - Hence, FOLLOW(E) = $\{ , \$ \}$
- FOLLOW(E')
- For $E \rightarrow TE'$ By rule $A \rightarrow \alpha B \beta$, we can map this with $E \rightarrow TE'$ then FOLLOW(E) is in FOLLOW(E')
 - FOLLOW(E')= $\{ , \$ \}$
- For $E' \rightarrow +TE'$ FOLLOW(E') is in FOLLOW(E')
 - FOLLOW(E')= $\{ , \$ \}$
- FOLLOW(T)
 - For $E \rightarrow TE'$
 - By rule $A \rightarrow \alpha B \beta$, FOLLOW(B) = $\{ \text{FIRST}(\beta) - \epsilon \}$, so FOLLOW(T) = $\{ \text{FIRST}(E') - \epsilon \} = \{ + \}$
 - For $E' \rightarrow +TE'$
 - By rule $A \rightarrow \alpha B \beta$, FOLLOW(T)=FOLLOW(E'). so, FOLLOW(T)= $\{ , \$ \}$
 - Hence FOLLOW(T) = $\{ + , , \$ \}$

- FOLLOW(T')
 - For $T \rightarrow FT'$
 - By $A \rightarrow \alpha B \beta$, then $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{+,), \$\}$
 - For $T \rightarrow *FT'$
 - By $A \rightarrow \alpha B \beta$, then $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{+,), \$\}$
 - Hence $\text{FOLLOW}(T') = \{+,), \$\}$
- FOLLOW(F)
 - For $T \rightarrow FT'$
 - By $A \rightarrow \alpha B \beta$, then $\text{FOLLOW}(F) = \{\text{FIRST}(T') - \epsilon\}$
 - $\text{FOLLOW}(F) = \{*\}$
 - For $T \rightarrow *FT'$
 - By $A \rightarrow \alpha B \beta$, then $\text{FOLLOW}(F) = \text{FOLLOW}(T') = \{+,), \$\}$
 - Hence, $\text{FOLLOW}(F) = \{+, *,), \$\}$

Predictive parsing table construction

- For the rule $A \rightarrow \alpha$ of grammar G
 1. For each a in $\text{FIRST}(\alpha)$ create $M[A, a] = A \rightarrow \alpha$ where a is a terminal symbol
 2. For ϵ in $\text{FIRST}(\alpha)$ create entry in $M[A, b] = A \rightarrow \alpha$ where b is the symbols from $\text{FOLLOW}(A)$
 3. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$ then create entry in the table $M[A, \$] = A \rightarrow \alpha$
 4. All the remaining entries in the table M are marked as ERROR

www.JntukMaterials.com

PARSING TABLE

	Id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Lets parse the input string $id+id*id$ using the above table. At initial configuration stack will contain start symbol E, in the input buffer the input string is placed and ended with \$

Stack	Input	Action
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$E'T'F	id+id*id\$	$T \rightarrow FT'$
\$E'T'id	id+id*id\$	$F \rightarrow id$
\$E'T'	+id*id\$	
\$E'	+id*id\$	$T' \rightarrow \epsilon$
\$E'T+	+id*id\$	$E' \rightarrow +TE'$
\$E'T	Id*id\$	
\$E'T'F	Id*id\$	$T \rightarrow FT'$
\$E'T'id	Id*id\$	$F \rightarrow id$
\$E'T'	*id\$	
\$E'T'F*	*id\$	$T' \rightarrow FT'$
\$E'T'F	Id\$	
\$E'T'id	Id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

BOTTOM UP PARSING

- The input string is taken first, and we try to reduce this string with the help of grammar and try to obtain the start symbol
- The process of parsing halts successfully as soon as we reach the start symbol
- **Handle – pruning**
 - find the substring that could be reduces by appropriate non-terminal is called handle
 - Handle is the string of substring that matches the right side of the production and we can reduce
 - In other words, a process of detecting handles and using them in reduction

www.JntukMaterials.com

HANDLE PRUNING

- Consider the grammar $E \rightarrow E + E$; $E \rightarrow \text{id}$
- RMD for the string $\text{id} + \text{id} + \text{id}$
 - $E \Rightarrow \mathbf{E + E}$
 - $E \Rightarrow E + \mathbf{E + E}$
 - $E \Rightarrow E + E + \mathbf{id}$
 - $E \Rightarrow E + \mathbf{id} + \text{id}$
 - $E \Rightarrow \mathbf{id} + \text{id} + \text{id}$

The bold strings are called handles

www.IntukMaterials.com

SHIFT REDUCE PARSER

- It attempts to construct parse tree from leaves to root.
- It requires the following data structures
 - The input buffer storing the input string
 - A stack for storing and accessing the LHS and RHS of rules

\$S

Stack

W\$

Input buffer

PARSING OPERATIONS

- SHIFT
 - Moving of the symbols from input buffer onto the stack
- REDUCE
 - If the handles present in the top of the stack then reduction of it by appropriate rule. RHS is popped and LHS is pushed
- ACCEPT
 - If the stack contains start symbol only and input buffer is empty at the same time that action is called accept
- ERROR
 - A situation in which parser cannot either shift or reduce the symbols

- Two rules followed
 - If the incoming operator has more priority than in stack operator then perform SHIFT
 - If in stack operator has same or less priority than the priority of incoming operators then perform REDUCE

Viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of shift/reduce parser are called viable prefixes. It is always possible to add terminals to the end of a viable prefix to obtain a right sentential form

Consider the grammar $E \rightarrow E-E$; $E \rightarrow E * E$; $E \rightarrow \text{id}$. Perform shift-reduce parsing of the input string $\text{id-id} * \text{id}$

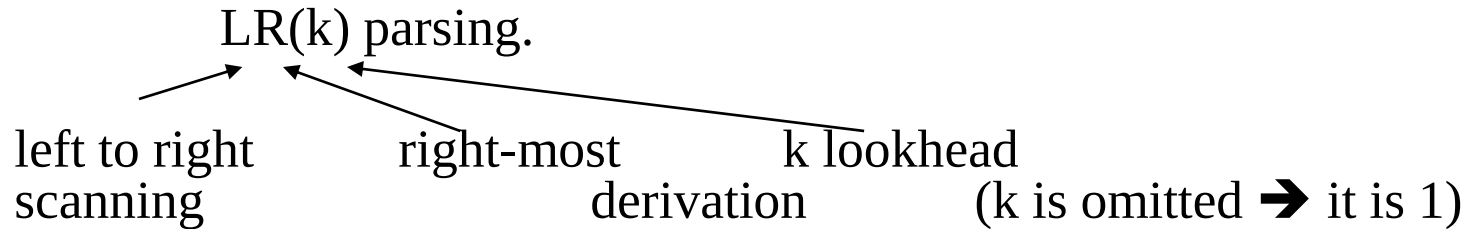
STACK	INPUT BUFFER	PARSING ACTION
\$	id-id*id\$	Shift
\$id	-id*id\$	Reduce by $E \rightarrow id$
\$E	-id*id\$	Shift
\$E-	id*id\$	Shift
\$E-id	*id\$	Reduce by $E \rightarrow id$
\$E-E	*id\$	Shift
\$E-E*	id\$	Shift
\$E-E*id	\$	Reduce $E \rightarrow id$
\$E-E*E	\$	Reduce $E \rightarrow E*E$
\$E-E	\$	Reduce $E \rightarrow E-E$
\$E	\$	Accept

OPERATOR PRECEDENCE PARSER

- A grammar G is said to be operator precedence if it poses following properties
 - No production rule on the right side is ϵ
 - There should not be any production rule possessing two adjacent non-terminals at the right hand side
- Parsing method
 - Construct OPP relations(table)
 - Identify the handles
 - Implementation using stack

- Advantage of OPP
 - Simple to implement
- Disadvantages of OPP
 - Operator minus has two different precedence(unary and binary). Hence, it is hard to handle tokens like minus sign
 - This can be applicable to only small class of grammars
- Application
 - The operator precedence parsing is done in a language having operators.

- The most powerful shift-reduce parsing (yet efficient) is:



- LR parsing is attractive because:
 - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
 - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

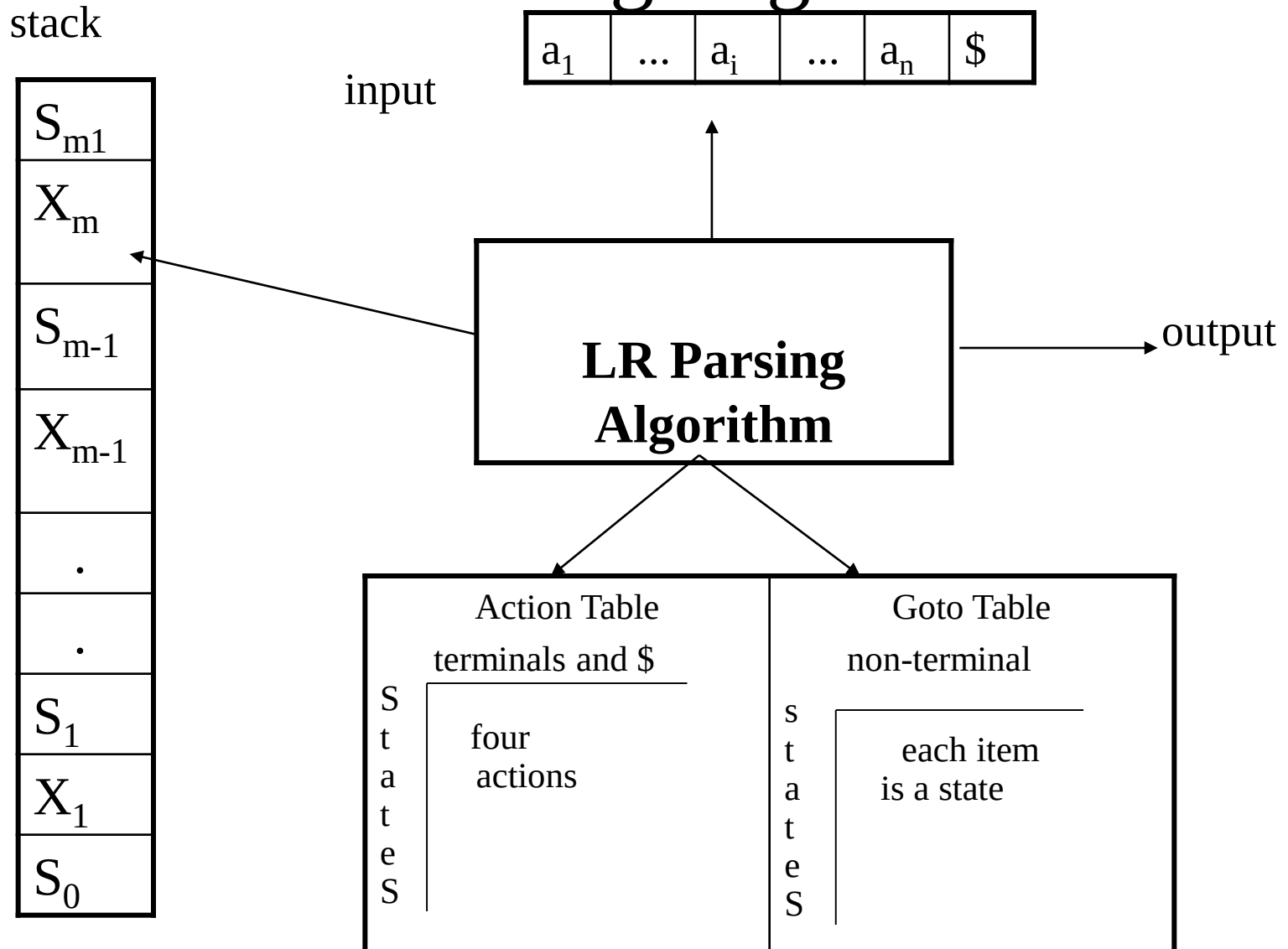
$LL(1)\text{-Grammars} \subset LR(1)\text{-Grammars}$

- An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

- **LR-Parsers**
 - covers wide range of grammars.
 - SLR – simple LR parser
 - LR – most general LR parser
 - LALR – intermediate LR parser (look-head LR parser)
 - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

www.JntukMaterials.com

LR Parsing Algorithm



- Initialize the stack with start symbol and invokes scanner to get next token
- It determines S_j the state currently on the top of the stack and a_i the current input symbol
- It consults the parsing table for the action $[S_j, a_i]$ which can have one of the four values
 - S_i means shift state i
 - r_j means reduce by rule j
 - Accept means successful parsing is done
 - Error indicates syntactical error

Simple LR parsing (SLR) definitions

- LR(0) items
 - The LR(0) item for grammar G is production rule in which symbol . Is inserted at some position in RHS of the rule.
 - Example
$$S \rightarrow .ABC$$
$$S \rightarrow A.BC$$
$$S \rightarrow AB.C$$
$$S \rightarrow ABC.$$
- Augmented grammar
 - If a grammar G is having start symbol S then augmented grammar is a new grammar G' in which S' is a new start symbol such that $S' \rightarrow S$
 - The purpose this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

- Kernel items
 - It is a collection of items $S' \rightarrow \cdot S$ and all the items whose dots are not at the leftmost end of RHS of the rule
 - Non-kernel items
 - The collection of all the items in which $\cdot$ Are at the left end of RHS of the rule
- Functions
 - Closure
 - Goto
 - These are two important functions required to create collection of canonical set of items
- Viable prefix-
 - set of prefixes in the right sentential form of production $A \rightarrow \alpha$. This set can appear on the stack during shift/reduce action

- For a CFG G , if I is the set of items then the function $\text{closure}(I)$ can be constructed using following rules
 - Consider I is a set of canonical items and initially every item I is added to $\text{closure}(I)$
 - If rule $A \rightarrow \alpha.B\beta$ is a rule in $\text{closure}(I)$ and there is another rule for B such as $B \rightarrow \gamma$ then,
 - **Closure(I) :**
 - $A \rightarrow \alpha.B\beta$
 - $B \rightarrow \gamma$

- This rule is applied until no more new items can be added to $\text{closure}(I)$.
- The meaning of rule $A \rightarrow \alpha.B\beta$ is that during derivation of the input string at some point we may require strings derivable from $B\beta$ as input.
- A non-terminal immediately to the right of $.$ Indicates that it has to be expanded shortly

Goto operation

- If there is a production $A \rightarrow \alpha.B\beta$ then $\text{goto}(A \rightarrow \alpha.B\beta, B) = A \rightarrow \alpha B.\beta$
- this means simply shifting of $.$ One position ahead over the grammar symbol (T or NT)
- The rule $A \rightarrow \alpha.B\beta$ is in I then the same goto function can be written as $\text{goto}(I, B)$

- **Construct the SLR(1) parsing table for**

1 $E \rightarrow E + T$

2 $E \rightarrow T$

3 $T \rightarrow T * F$

4 $T \rightarrow F$

5 $F \rightarrow (E)$

6 $F \rightarrow id$

I0:
 $E' \rightarrow .E$
 $E \rightarrow .E+T$
 $E \rightarrow .T$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .(E)$
 $F \rightarrow .id$

Goto(I0, E)
 I1: $E' \rightarrow E.$
 $E \rightarrow E.+T$

 Goto(I0, T)
 I2: $E \rightarrow T.$
 $T \rightarrow T.*F$

Goto(I1, +)
 I6: $E \rightarrow E+.T$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .(E)$
 $F \rightarrow .id$

Goto(I0, F)
 I3: $T \rightarrow F.$

Goto(I2, *)
 I7: $T \rightarrow T*.F$
 $F \rightarrow .(E)$
 $F \rightarrow .id$

Goto(I0, ()
 I4: $T \rightarrow (.E)$
 $E \rightarrow .E+T$
 $E \rightarrow .T$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .(E)$
 $F \rightarrow .id$

Goto(I4, E)
 I8: $F \rightarrow (E.)$
 $E \rightarrow E.+T$

Goto(I6, T)
 I9: $E \rightarrow E+T.$
 $T \rightarrow T.*F$

Goto(I7, F)
 I10: $T \rightarrow T*F.$

Goto(I0, id)
 I5: $F \rightarrow id.$

Goto(I8,))
 I11: $F \rightarrow (E).$

- $\text{FOLLOW}(E') = \{\$\}$
- $\text{FOLLOW}(E) = \{+,), \$\}$
- $\text{FOLLOW}(T) = \{+, *,), \$\}$
- $\text{FOLLOW}(F) = \{+, *,), \$\}$

state	id	+	*	(	)	\$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

STACK	INPUT BUFFER	ACTION TABLE	GO TO TABLE	PARSING ACTION
\$0	Id*id*id\$	[0,id]=s5		Shift
\$0id5	*id+id\$	[5,*]=r6	[0,f]=3	Reduce F→id
\$0F3	*id*id\$	[3,*]=r4	[0,T]=2	Reduce T→F
\$0T2	*id+id\$	[2,*]=s7		Shift
\$0T2*7	Id+id\$	[7,id]=s5		Shift
\$0T2*7id5	+id\$	[5,+]=r6	[7,F]=10	reduce
\$0T2*7F10	+id\$	[10,+]=r3	[0,T]=2	Reduce
\$0T2	+id\$	[2,+]=r2	[0,E]=1	Reduce
\$0E1	+id\$	[1,=]=s6		Shift
\$0E1+6	+id\$	[6,id]=s5		Shift
\$0E1+6ID5	\$	[5,\$]=r6	[6,F]=3	Reduce
\$0E1+6F3	\$	[3,\$]=r4	[6,T]=9	Reduce
\$0E1+6T9	\$	[9,\$]=r1	[0,E]=1	Reduce
\$0E1	\$	Accept		accept

CLR PARSING OF LR(1) PARSING

- Construction of canonical set of items along with lookahead
- For the grammar G initially add $S' \rightarrow .S$ in the set of item C
- For each set of items I_i in C and for each grammar symbol X (T or NT) add $\text{closure}(I_i, X)$. This process is repeated by applying $\text{goto}(I_i, X)$ for each X in I_i such that $\text{goto}(I_i, X)$ is not empty and not in C . The set of items has to be constructed until no more set of items can be added to C
- The closure function can be computed as : for each item $[A \rightarrow \alpha.X\beta, a]$ is in I and rule $[A \rightarrow \alpha X.\beta, a]$ is not in I then add $[A \rightarrow \alpha X.\beta, a]$ to I
- This process is repeated until no more set of items can be added to the collection C

CONSTRUCTION OF CLR PARSING TABLE

- Construct set of items $C = \{I_0, I_1, I_2, \dots, I_n\}$ where C is a collection of set of LR(1) items for the input grammar G' .
- The parsing actions are based on each items I_i .
 - If $[A \rightarrow \alpha B \beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$ then create a entry in the action table $\text{action}[I_i, a] = \text{shift } j$.
 - If there is a production $A \rightarrow \alpha.$, $a]$ in I_i then in action table $\text{action}[I_i, a] = \text{reduce by } A \rightarrow \alpha$. Here A should not be S' .
 - If there is a production $S' \rightarrow S., \$$ in I_i then $\text{action}[i, \$] = \text{accept}$.
- The goto part of LR table can be filled as: the goto transition for state i is considered for NT only. If $\text{goto}(I_i, A) = I_j$, then $\text{goto}(I_i, A) = j$
- All other entries are defined as ERROR

- Construct CLR for the grammar

$$E \rightarrow E + E / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / \text{id}.$$

- $\text{FOLLOW}(E) = \{+,), \$\}$
 $\text{FIRST}(E) = \{ (, \text{id} \}$
- $\text{FOLLOW}(T) = \{+, *,), \$\}$
 $\text{FIRST}(T) = \{ (, \text{id} \}$
- $\text{FOLLOW}(F) = \{+, *,), \$\}$
 $\text{FIRST}(F) = \{ (, \text{id} \}$

- Augmented grammar

$$E' \rightarrow E$$

$$E \rightarrow E+T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

- LR(1) items

- LR(0) items

$$E' \rightarrow \cdot E$$

$$E \rightarrow \cdot E+T$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot id$$

- LR(1) items

$$E' \rightarrow \cdot E, \$$$

$$E \rightarrow \cdot E+T, \$/+$$

$$E \rightarrow \cdot T, \$/+$$

$$T \rightarrow \cdot T * F, \$/+/*$$

$$T \rightarrow \cdot F, \$/+/*$$

$$F \rightarrow \cdot (E), \$/+/*$$

$$F \rightarrow \cdot id, \$/+/*$$

Goto(I0, E)

I1 : $E' \rightarrow E. , \$$

$E \rightarrow E.+T, \$/+$

Goto(I0,T)

I2: $E \rightarrow T., \$/+$

$T \rightarrow T.*F, \$/+/*$

Goto(I0,F)

I3: $T \rightarrow F., \$/+/*$

Goto(I0,())

I4: $F \rightarrow (.E), \$/+,*$

$E \rightarrow .E+T,)/+$

$E \rightarrow .T,)/+$

$T \rightarrow .T*F,)/+/*$

$T \rightarrow .F,)/+,*$

$F \rightarrow .(E),)/+/*$

$F \rightarrow .id,),+,*$

Goto(I0, id)

I5: $F \rightarrow id. , \$/+/*$

Goto(I1,+)

I6: $E \rightarrow E+.T, \$/+$

$T \rightarrow .T*F, \$/+/*$

$T \rightarrow .F, \$/+/*$

$F \rightarrow .(E), \$/+/*$

$F \rightarrow .id, \$/+/*$

STAT ES	+	*	(	)	ad	\$	E	T	F
0			S4		S5		1	2	3
1	S6					ACC			
2	R2	S7				R2			
3	R4	R4				R4			
4			S11		S12		8	9	10
5	R6	R6				R6			
6			S4		S5			13	3
7			S4		S5				14
8	S16			S15					
9	R2	S17		R2					
10	R4	R4		R4					130

11			S11		S12		18	9	10
12	R6	R6		R6					
13	R1	S7				R1			
14	R3	R3				R3			
15	R5	R5				R5			
16			S11		S12			19	10
17			S11		S12				20
18	S16			S21					
19	R1	S17		R1					
20	R3	R3		R3					
21	R3	R5		R5					131

STACK	INPUT BUFFER	ACTION
\$0	id+id*id\$	Shift s5
\$0id5	+id*id\$	R6
\$0F3	+id*id\$	R4
\$0T2	+id*id\$	R2
\$0E1	+id*id\$	S6
\$0E1+6	id*id\$	S5
\$0E1+6 id 5	*id\$	R6
\$0E1+6 F 3	*id\$	R4
\$0E1+6 T13	*id\$	S7
\$0E1+6T13*7	Id\$	S5
\$0E1+6T13*7id 5	\$	R6
\$0E1+6T13*7F14	\$	R3
\$0E1+6T13	\$	R1
\$0E1	\$	ACC

- Construction of LALR parsing table
- Construct LR(1) items
- Merge two states I_i and I_j if the first component are matching and create a new state replacing one of the older states such as $I_{ij} = I_i \cup I_j$
- The parsing actions are based on each item I_i .
 - If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$ then create an entry in the action table $\text{action}[I_i, a] = \text{shift } j$
 - If there is a production $[A \rightarrow \alpha., a]$ in I_i then in the action table $\text{action}[I_i, a] = \text{reduce by } A \rightarrow \alpha.$ Here A should not be S' .
 - If there is a production $S' \rightarrow S, \$$ in I_i then $\text{action}[i, \$] = \text{accept}$
- The goto part : the goto transitions for state i is considered for NTonly. If $\text{goto}(I_i, A) = I_j$, then $\text{goto}[I_i, A] = j$
- If the parsing action conflicts, then the grammar is not LALR(1). All other entries are ERROR

LALR STATES FROM CLR

www.JntukMaterials.com

I2,9: $E \rightarrow T, \$ / + /)$

$T \rightarrow T * F, \$ / + /) / *$

I7,17: $T \rightarrow T * . F, \$ / + /) / *$

$F \rightarrow . (E), \$ / + /) / *$

$F \rightarrow . id, \$ / + / * /)$

I3,10: $T \rightarrow F, \$ / + /) / *$

I4,11: $F \rightarrow (. E), \$ / + /) / *$

$E \rightarrow . E + T,) / +$

$E \rightarrow . T,) / +$

$T \rightarrow . T * F,) / + / *$

$T \rightarrow . F,) / + / *$

$F \rightarrow . (E),) / + / *$

$F \rightarrow . id,) / + / *$

I8,18: $F \rightarrow (E .), \$ / + /) / *$

$E \rightarrow E . + T,) / +$

I13,19: $E \rightarrow E + T, \$ /) / +$

$T \rightarrow T * F, \$ /) / + / *$

I5,12: $F \rightarrow id, \$ / + /) / *$

I14, 40: $T \rightarrow T * F, \$ / + /) / *$

I6,16: $E \rightarrow E + . T, \$ /) . +$

$T \rightarrow . T * F, \$ /) / + / *$

$T \rightarrow . F, \$ / + /) / *$

$F \rightarrow . (E), \$ /) / + / *$

$F \rightarrow . id, \$ /) / + / *$

I15, 21: $F \rightarrow (E) . , \$ / + /) / *$

STATE	+	*	(	)	Id	\$	E	T	F
0			S4,11	S5,12			1	2,9	3,10
1	S6,16					ACC			
2,9	R2	S7,17		R2		R2			
3,10	R4	R4		R4		R4			
4,11			S4,11		S5,12		8,18	2,9	3,10
5,12	R6	R6		R6		R6			
6,16			S4,11		S5,12			13,9	3,10
7,17			S4,11		S5,12				14,20
8,18	S6,16			S15,21					
13,19	R1	S7,17		R1					
14,20	R3	R3		R3		R3			
15,21	R5	R5		R5		R5			