**Chapter 1**

**What’s a compiler ?**

A compiler is a **complex** program to translates one language to another, used in many forms of computing(cmd interpreters, interface programs)

**Programs related to compilers:**  
**Interpreters**(A language translator. It executes the source program immediately) **Assemblers**(A translator translates assembly language into object code) **Linkers**(Collects code separately compiled or assembled in different object files into a file. Connects the code for standard library functions. Connects resources supplied by the operating system of the computer) **Loaders**(Loaders resolve all relocatable address relative to the starting addres) **Preprocessors**(delete comments, include other files, perform macro substitutions) **Editors**(Produce a standard file ( structure based editors)) **Debuggers**(Determine execution errors in a compiled program)  **Profilers** (Collect statistics on the behavior of an object program during execution)  **Project managers**( coordinate the files being worked on by different people.

source code control system, revision control system)

**Compiler Phases** (Source code) Scanner (Token) Parser (Syntax Tree) Semantic analyzer (Annotated Tree) Source code optimizer (Intermediate code) Code generator (Target code) Target code optimizer (Target code)

**Data Structure**

TOKENS(scanner collects characters into a token, as a value of an enumerated data type for tokens, also preserve the string of characters or other derived information, such as name of identifier, value of a number token)

SYNTAX TREE(pointer-based structure generated by parser, requires different attributes depending on kind of language structure)

THE SYMBOL TABLE(Keeps information associated with identifiers: function, variable, constants, and data types, hash tables are often used)

THE LITERAL TABLE(Stores constants and strings, reducing size of program)

**Chapter 2 Scanning**

The scanner returns a token of a certain type to the parser whenever it sees a sequence of input characters, a lexeme, that matches the pattern for that type of token.

**Regular Expressions**

\*优先权最高，连结其次，| 最低。

{ }集Φ不包括任何串，而{ε}则包含一个没有任何字符的串。L0 = { ε }

nat = [0-9]+ /// signedNat = (+|-)?nat number = signedNat(“.”nat)?( E sNat ) ?

**Comment:**

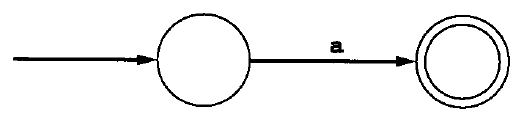
ba(∼**(ab)**)\*ab **×** b\*(a\*~(a|b)b\*)\*a\* **√**

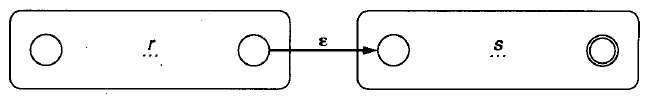
**Deterministic Finite Automata**

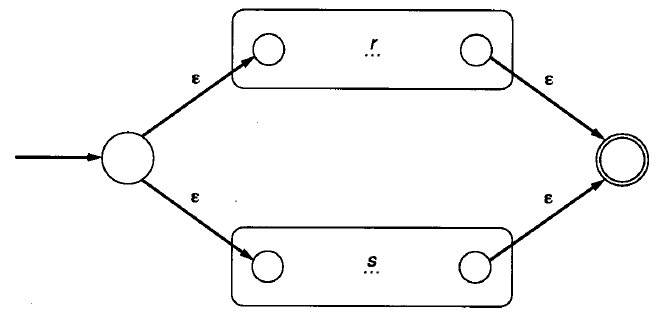
Transition table: Two-dimensional array, indexed by state and input character that expresses the values of the transition function T. Brackets indicate **noninput consume** transition.

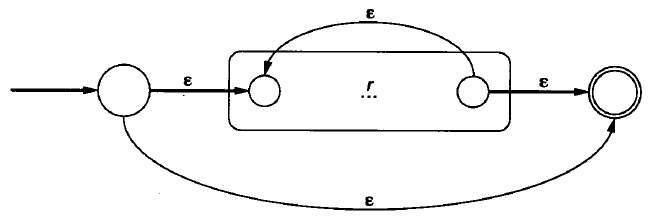
**NFA** does not represent an algorithm, it can be simulated by an algorithm that backtracks through every nondeterministic choice.

**Thompson’s construction:**

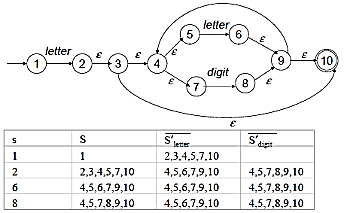




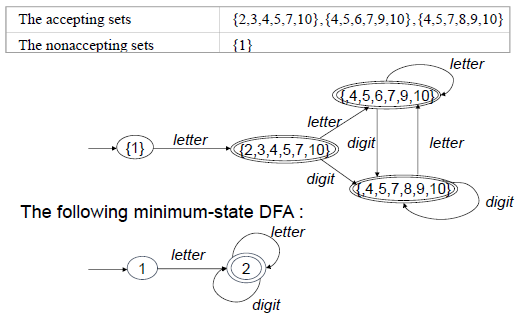




**Subset Construction**



**Minimizing the states in DFA**



**Chapter 3 Parsing & Context Free Grammar**

Structure of the syntax tree depends on the particular syntactic structure of the language, usually defined as a dynamic data structure.

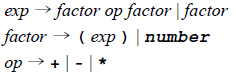
**Derivation**: Grammar rules determine the legal strings of token symbols by means of derivations. At each step in a derivation, a single replacement is made using one choice from a grammar rule.

**Leftmost derivation**: a deri in which the leftmost nonterminal is replaced at each step in the derivation. = preorder =

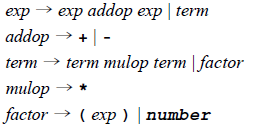
**Rightmost** = reverse post order =

**left recursive:** nonterminal A appears as the first symbol on the right-hand side of the rule defining A.

**Ambiguity**: Grammar generates a string with two distinct parse trees. Change grammar or disambiguating rule.

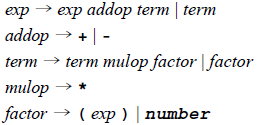


**Precedence**:



**Associativity**:

left recursive rule makes its operators associate on the left.



Many derivations may give rise to the

same parse tree. Each parse tree has a unique leftmost & rightmost derivation that give rise to it.

Chomsky Hierarchy: 0 unrestricted;

1 context sensitive; 2 context free;

3 regular

**Chapter 4 Top-down parsing**

**Recursive-Descent Parsing**: right hand side of the grammar for A specifies the structure of the code for this procedure, requires the use of **EBNF**.

**LL(1) Parsing**: left to right scan / trace out a leftmost derivation for the input string / one token lookahead.

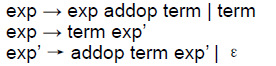
**generate & match**



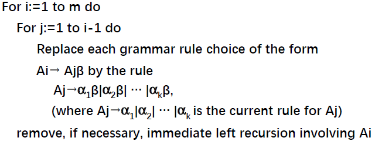
A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most one production in each table entry.

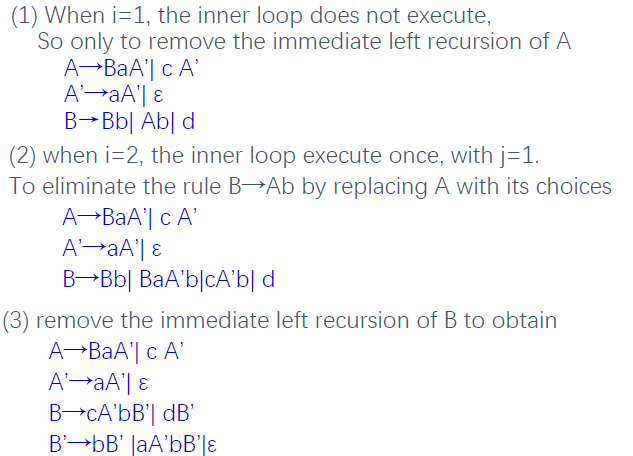
An LL(1) grammar cannot be ambiguous.

**Left Recursion Removal:**

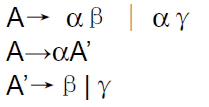








**Left Factoring**:



**first set**: 1. If X is a terminal or ε, then First(X) = {X};

2. If X is a non-terminal, then for each production choice X→X1 X2 … Xn, First(X) contains First(X1)-{ε}.

3. For each i=2,…,n, if for all k=1,..,i-1, First(Xk) contains ε, then First(X) constains First(Xi)-{ε}.

4. If all the set First(X1)..First(Xn) contain ε, the First(X) contains ε.

**follow set**: 1. if A is the start symbol, the $ is in the Follow(A).

2. if there is a production B→ α Aγ, then First(γ)-{ε} is in Follow(A).

3. if there is a production B→α Aγ, such that ε in First(γ), then Follow(A) contains Follow(B).

**Grammar is LL(1) if:**

1. For every production A→α1|α2|…|αn, First(αi) ∩ First(αj) is empty for all i and j, 1≦i,j≦n, i≠j.

2. For every non-terminal A such that First(A) contains ε, First(A) ∩ Follow(A) is empty.

**Construct of the LL(1) parsing table**:

For each non-terminal A and production choice A→ α.

1. For each token a in First(α), add A→α to the entry M[A,a].

2. If ε is in First(α), for each element a of Follow(A) (a token or $), add A→α to M[A,a].

**Chapter 5 Bottom-Up Parsing**

**shift & reduce & accept**



E, E + ,and E+n are all **viable prefixes**

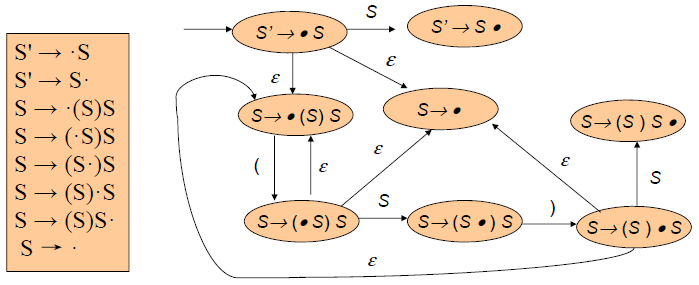
of the right sentential form E+n.

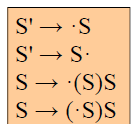
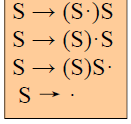
The sequence of symbols on the parsing stack is called a viable prefix of the right sentential form .

The string together with the position in the right sentential form where it occurs, and the production used to reduce it, is called the handle of the right sentential form. n & E->n; E+n & E->E+n

**做题时先写个推导出来再匹配**

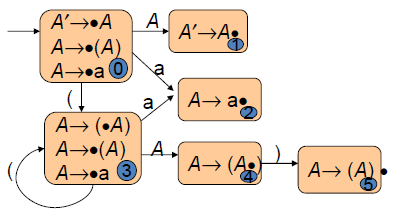
**LR(0):**

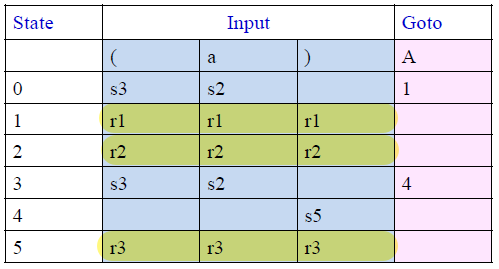


A grammar is LR(0) grammar if the above rules are unambiguous.

A grammar is LR(0) if and only if: each state is a shift state( a state containing only “shift” items); a reduce state containing a single complete item.

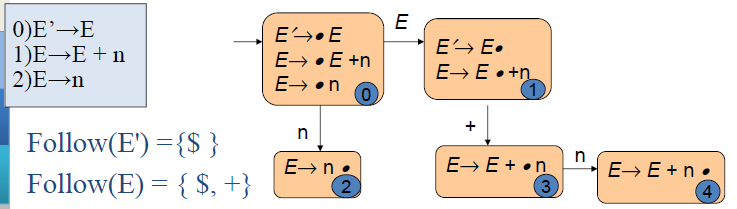




**SLR(1):** 规约时检查是否在follow set

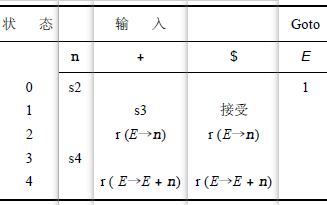
A grammar is SLR(1) grammar if the SLR(1) parsing rules results in no ambiguity.

A grammar is SLR(1) if and only if, for any state s: 1. For any item A → α·Xβ in s with X a terminal, there is no complete item B → γ. in s with X in Follow(B). 2. For any two complete items A → α·and B →β· in s, Follow(A) ∩ Follow(B) is empty.



Follow(E') ={$ }

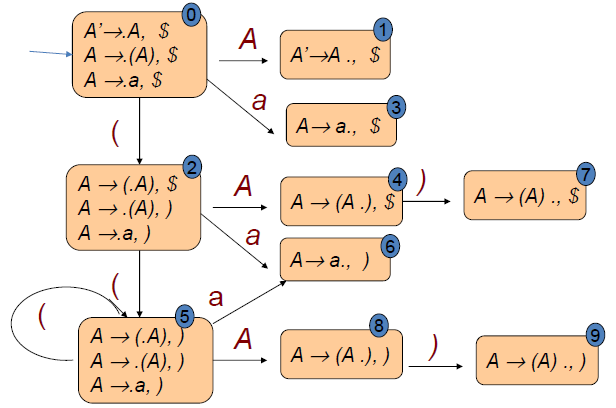
Follow(E) = { $, +}



**SPL(1):** For shift-reduce conflicts, a natural disambiguating rule, still have reduce-reduce conflicts.

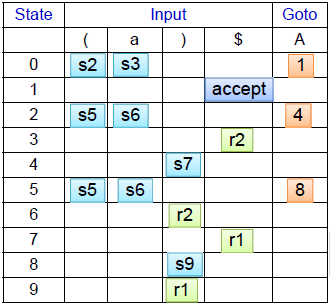
LR(1): [A→α·Bγ, a] , [B→·β,b] for every production token b = First(γa).

[S’ —>·S, $] A→(A) | a

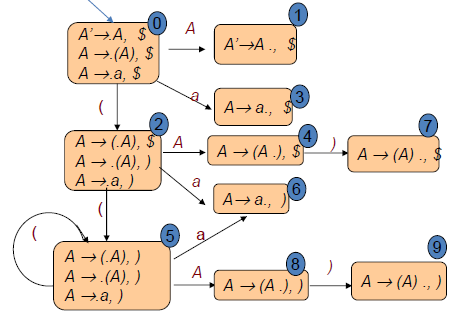


A grammar is an LR(1) grammar if no ambiguity.

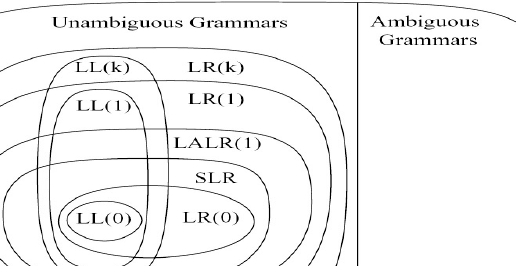
A grammar is LR(1) if and only if, for any state s: 1. For any item [A→α·Xβ,a] in s with X a terminal, there is no item in s of the form [B→γ·,X] (shift-reduce conflict); 2. There are no two items in s of the form [A→α·, a] and [B→β·,a] (reduce-reduce conflict).



**LALR(1):**



* A grammar is an LALR(1) grammar if no parsing conflicts arise in the LALR(1) parsing algorithm.
* If a grammar is LR(1), then the LALR(1) parsing table cannot have any shift-reduce conflicts, there may be reduce-reduce conflicts.
* If a grammar is SLR(1), then it certainly is LALR(1)
* LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(l) parsing.
* Compute the DFA of LALR(1) items directly from the DFA of LR(0) items through a process of propagating lookaheads.
* If the grammar is already LALR( 1 ), the only consequence of using LALR( 1 ) parsing over general LR parsing as following in the presence of errors: some spurious reductions may be made before error is declared



**Chapter 6 Semantic Analysis**

An attribute is **synthesized** if all its dependencies point from child to parent in the parse tree. An attribute that is not synthesized is called an **inherited** attribute.

**S-attributed grammar:** An attribute grammar in which all the attributes are synthesized

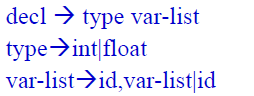
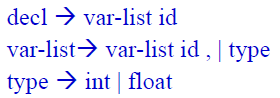
An attribute grammar for attribute a1,…,ak is **L-attributed**: for each inherited attribute aj and each grammar rule: X0 -> X1X2…Xn

The associated equations for aj are all of the form: Xi.aj = fij(X0.a1…X0.ak…Xi-1.a0,Xi-1.ak); S-attributed grammar is L-attributed grammar.

A traversal order of the dependency graph that obeys this restriction is called a **topological sort**, the graph must be **acyclic**.

Inherited attributes be computed in preorder, often be treated as parameters of the call. Synthesized attributes be computed in postorder, often be treated as returned values of the call.

All inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.

Symbol table: Hash

open addressing / separate chaining

Actual size of the bucket array should be chosen to be a prime number.

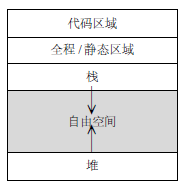
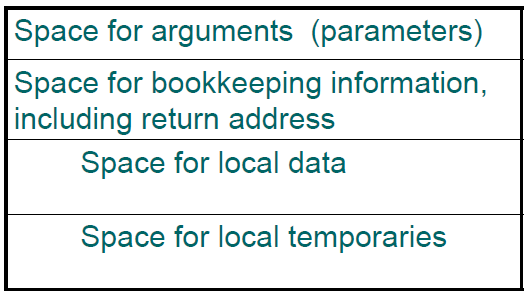
hash func = (αn-1c1+αn-2c2 +…..+ α cn-1+ cn) mod size; A reasonable choice for α is a power of 2, such as 16 or 128.

Scope Rules: declaration before use / the most closely nested rule for block structure

block: any construct that can contain declarations. such as procedure/ function declarations

Type equivalence: Structural equivalence / Name equivalence / Declaration equivalence

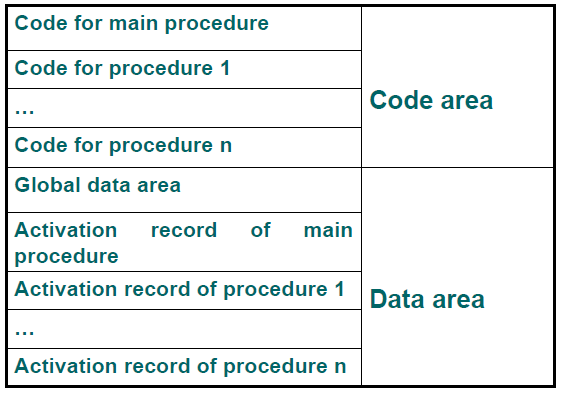
**Chapter 7 Run-Time Environments**

**Fully Static Runtime Environments:**

no pointer or dynamic allocation.

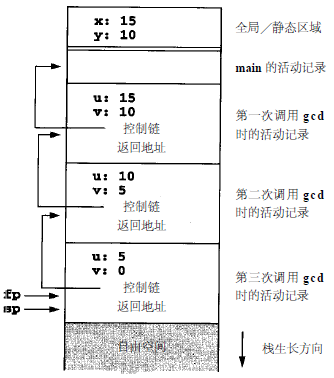
no recursive procedure calling



Procedure has only a single activation record. All variables ( local or global) can be accessed directly via fixed address.

**Stack-Based Runtime Environments:**

Recursive calls are allowed, local v~ are newly allocated at each call. Activation records are allocated in stack.



Control link or dynamic link, a point to a record of the immediately preceding activation.

**Calling sequence:** (1)Compute the arguments and store them in their correct positions in the new activation record of the procedure. (2) Store the fp as the control link in the new activation record; (3) Change the fp so that it points to the beginning of the new activation record; (4) Store the return address in the new activation record; (5) Perform a jump to the code of the procedure to be called.

**Return sequence:** (1) Copy the fp to the sp. (2) Load the control link into the fp. (3) Perform a jump to the return address. (4) Change the sp to pop the arguments.

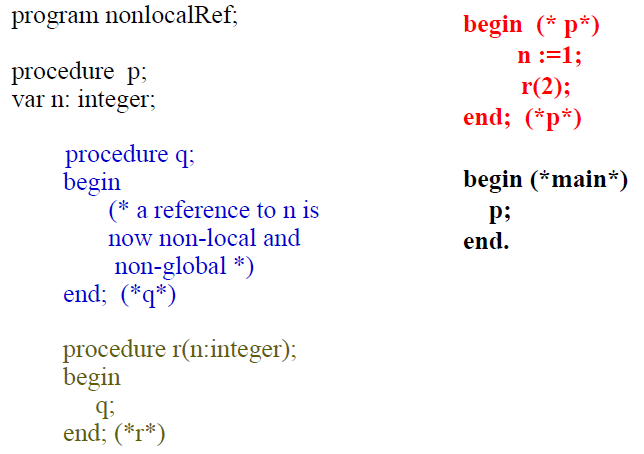
**Variable-Length Data on the Stack**

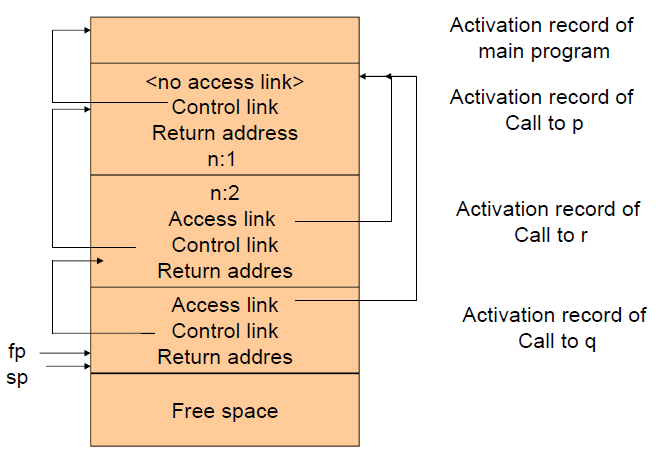
C deal with this by pushing the argu~ to a call in reverse order onto the runtime stack. The first parameter is always located at a fixed offset from the fp in the implementation described above. Another option is to use a processor mechanism such as ap(argument pointer) in VAX arch.

**Stack-Based Environment with local**

**Procedures**

Access link represents the defining environment of the procedure; access link is sometimes also called the static link. Control link represents the calling environment of the procedure.





**How to find procedure access link:**

(1) Using the (compile-time) nesting level information attached to the declaration of the procedure;

(2) Generate an access chain as if to access a variable at the same nesting level;

**Fully Dynamic Runtime Environments**

Allocate: take a size parameter / return a pointer to a block of memory of the correct size, return a null pointer if none exists. || Free: takes a pointer to an allocated block of memory and marks it as being free again.

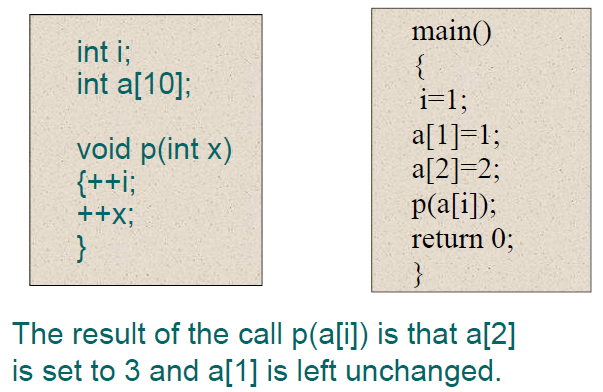
**Heap Management:** 1. a circular linked list of free blocks / memory is taken by malloc / memory is return by free. 2. use a circular linked list data structure that keep track of both allocated and free block.

Automatic: garbage collection:

1. No memory is freed until a call to malloc fails, (1) Follows all pointers recursively, starting with all currently accessible pointer values and marks each block of storage reached.

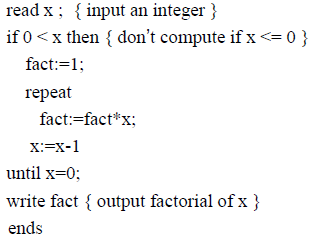
(2) Sweeps linearly through memory, returning unmarked blocks to free memory and perform memory compaction. 2. During the marking pass, all reached blocks are immediately copied to the second half of storage not in use; No extra mark bit is required and only one pass, performs compaction automatically. It does little to improve processing delays during storage reclamation. 3. Generational garbage collection: Allocated objects that survive long enough are simply copied into permanent space and are never deallocated during subsequent storage reclamations.

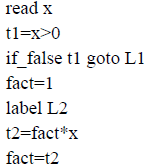
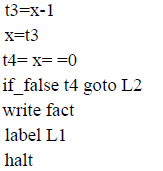
**Parameter Passing**: value / reference / value-result / name



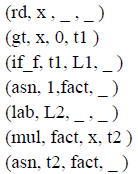
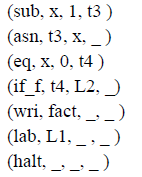
**Chapter 8 Code Generation**

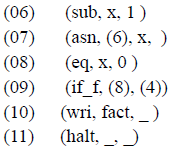
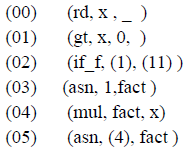
Depends on the characteristics of the source language but also on detailed information about the target arc~, the structure of the runtime environment, and OS running on the target machine.



**Quadruple implementation & triples:**

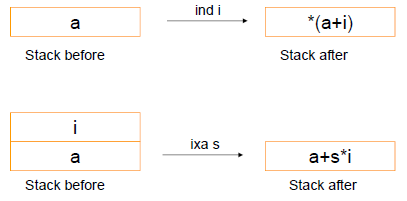
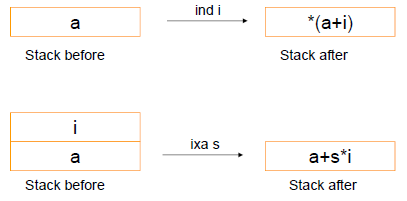
 



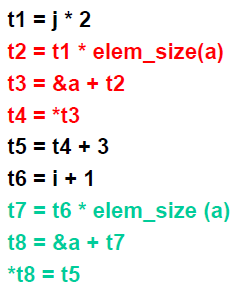
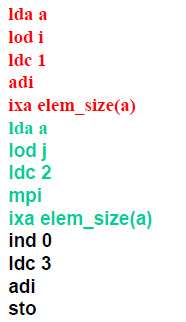
The P-machine consists of a code memory, an unspecified data memory for named variables, and a stack for temporary data, registers are needed

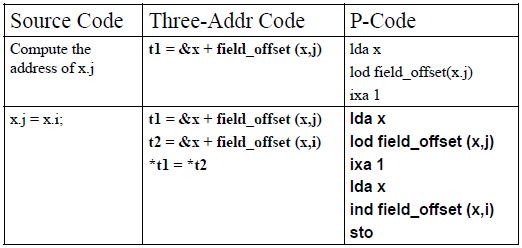
to maintain the stack and support execution.

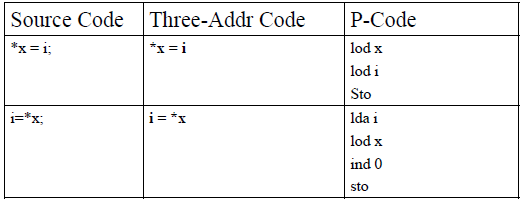
**Generation of Target Code from Intermediate Code:** macro expansion & static simulation

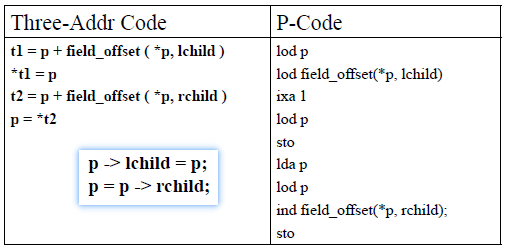


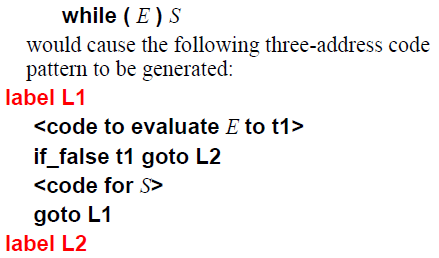
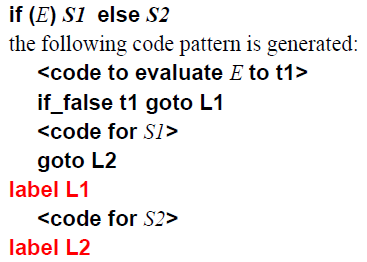
a[i+1] = a[j\*2]+3

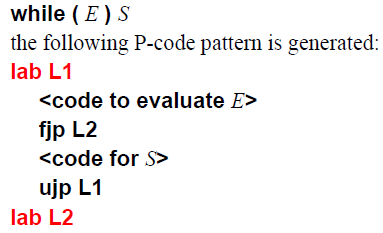
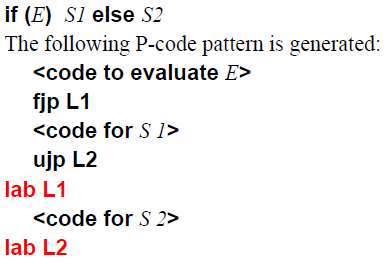
 





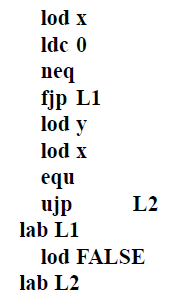


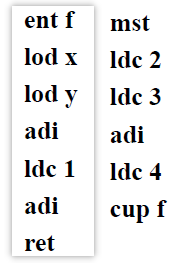
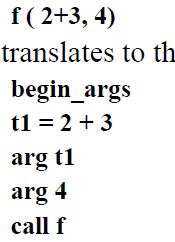
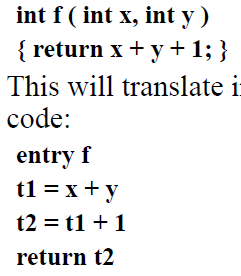




a and b ≡ if a then b else false

a or b ≡ if a then true else b

 (x != 0 )& & ( y == x )



**Code Optimizations: Register Allocation**: increase the number and speed of operations that can be performed directly in memory to avoid register spill; decrease the number of operations can perform directly in memory, but at same time to increase the number of available register to 32/64/128, called RISC approach.

**Unnecessary Operations**: Repeated evaluation of an expression in code, Common sub-expression elimination; Avoid storing the value of a variable or temporary not subsequently used; Elimination of unreachable code

**Costly Operations:** reduced in strength: x3 by x\*x\*x 5\*x by 2\*x+2\*x+x; constant folding; constant propagation: To determine if a variable might have a constant value for part or all of a program; Procedure inlining: to replace the procedure call with the code for the procedure body; Tail recursion removal

**Prediction Program Behavior**

Classification: time(source/target-level) space( local, global, interprocedural )

Using DAG to eliminate local common subexpression, redundant stores

Register descriptors associate with each register a list of the variable names whose value is currently in that register.

Address descriptors associate with each variable name the locations in memory where its value is to be found.

Panic Mode in LL(1) Parsers: Pop: token is $ or is in Follow(A); Scan: token is not $ and is not in First(A)∪Follow(A); Push a new non-terminal

detect errors LR(1)>LALR=SLR>LR(0)