**Chapter 1**

**Programs related to compilers:**  
**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)

**Data Structure**

TOKENS

SYNTAX TREE

THE SYMBOL TABLE(Keeps information associated with identifiers:

THE LITERAL TABLE

**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 ) ?

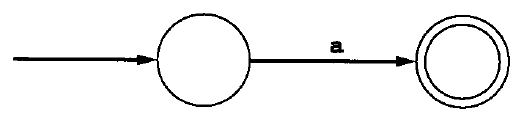
**Comment:**

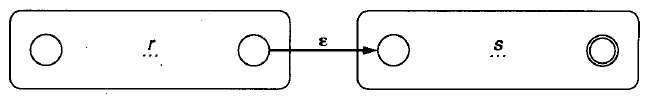
ba(∼**(ab)**)\*ab **×** b\*(a\*~(a|b)b\*)\*a\* **√**

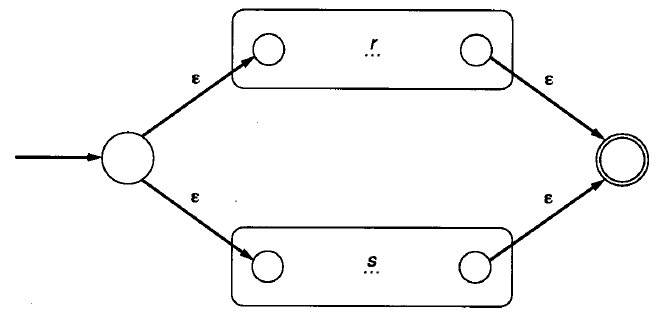
**Deterministic Finite Automata**

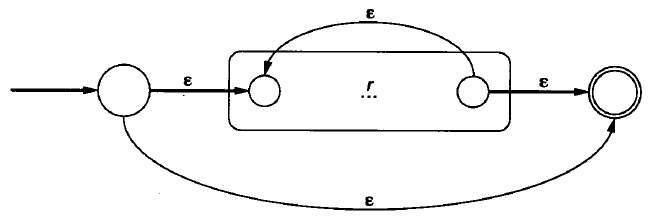
Transition table: Two-dimensional array, indexed by state and input character that expresses the values of the transition function T. Brackets indicate **noninput consume** transition.

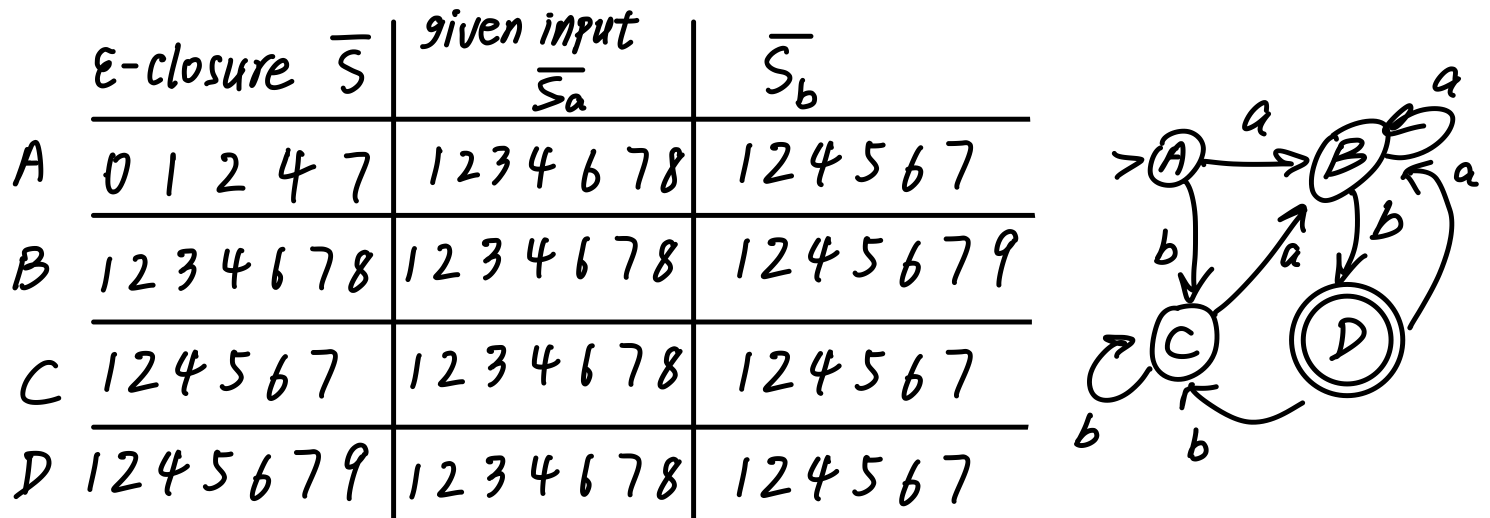
**Thompson’s construction:**

a

rs

r|s

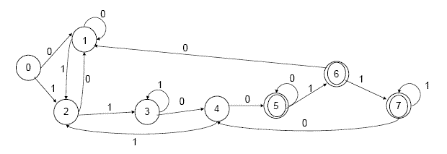
r\*

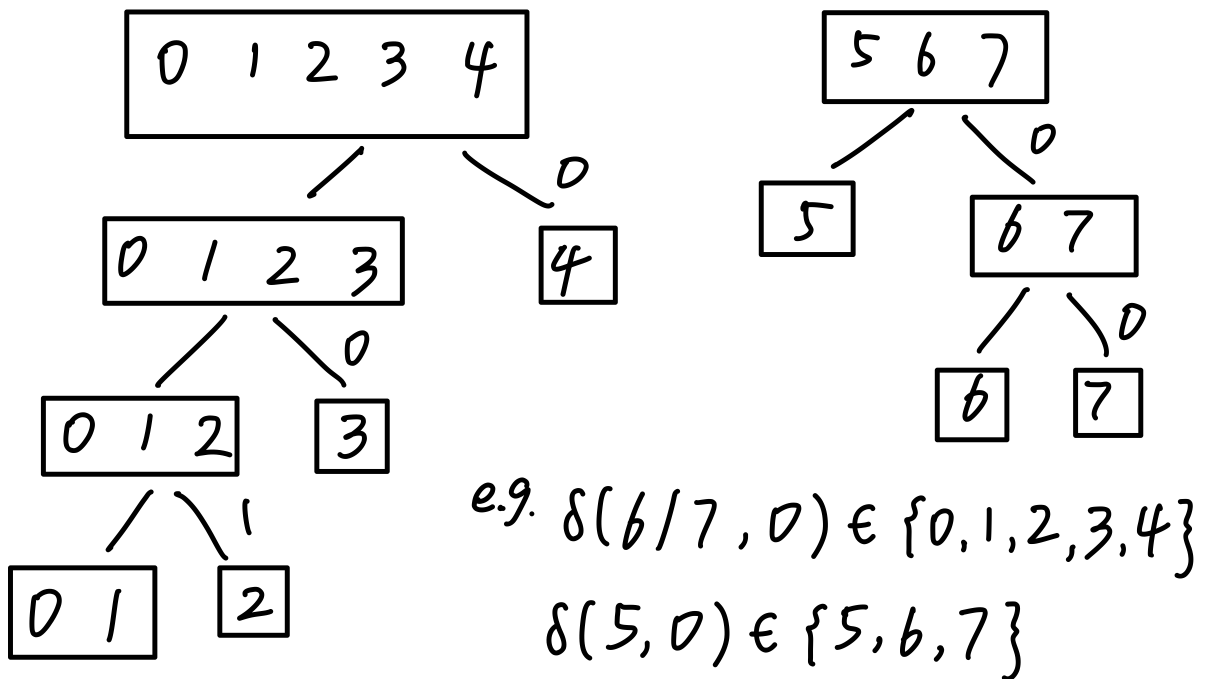


**Subset Construction**

求：状态的ε-闭包；在某字符下转移的目标状态的ε-闭包

**Minimizing the states in DFA**

****



core:两个状态对所有字符的表现是否一样

on *a*, transitions that land in different sets

transitions v.s. no transitions

* *a* distinguishes the states

**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.

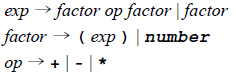
**Leftmost derivation**: = preorder = in which the leftmost nonterminal is replaced at each step.

**Rightmost** = reverse post order =

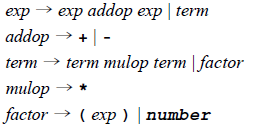
**left recursive:** nonterminal A appears as the first symbol on the right-hand side of the rule defining A.

**Ambiguity**: Grammar generates a string with two distinct parse trees / two leftmost derivations.

Change grammar or disambiguating rule.

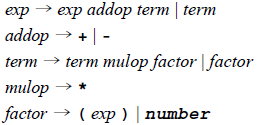


**Precedence**:



**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;

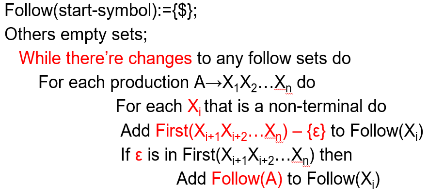
3 regular

**EBNF**: or | ; repetition { .. }

**Chapter 4 Top-down parsing**

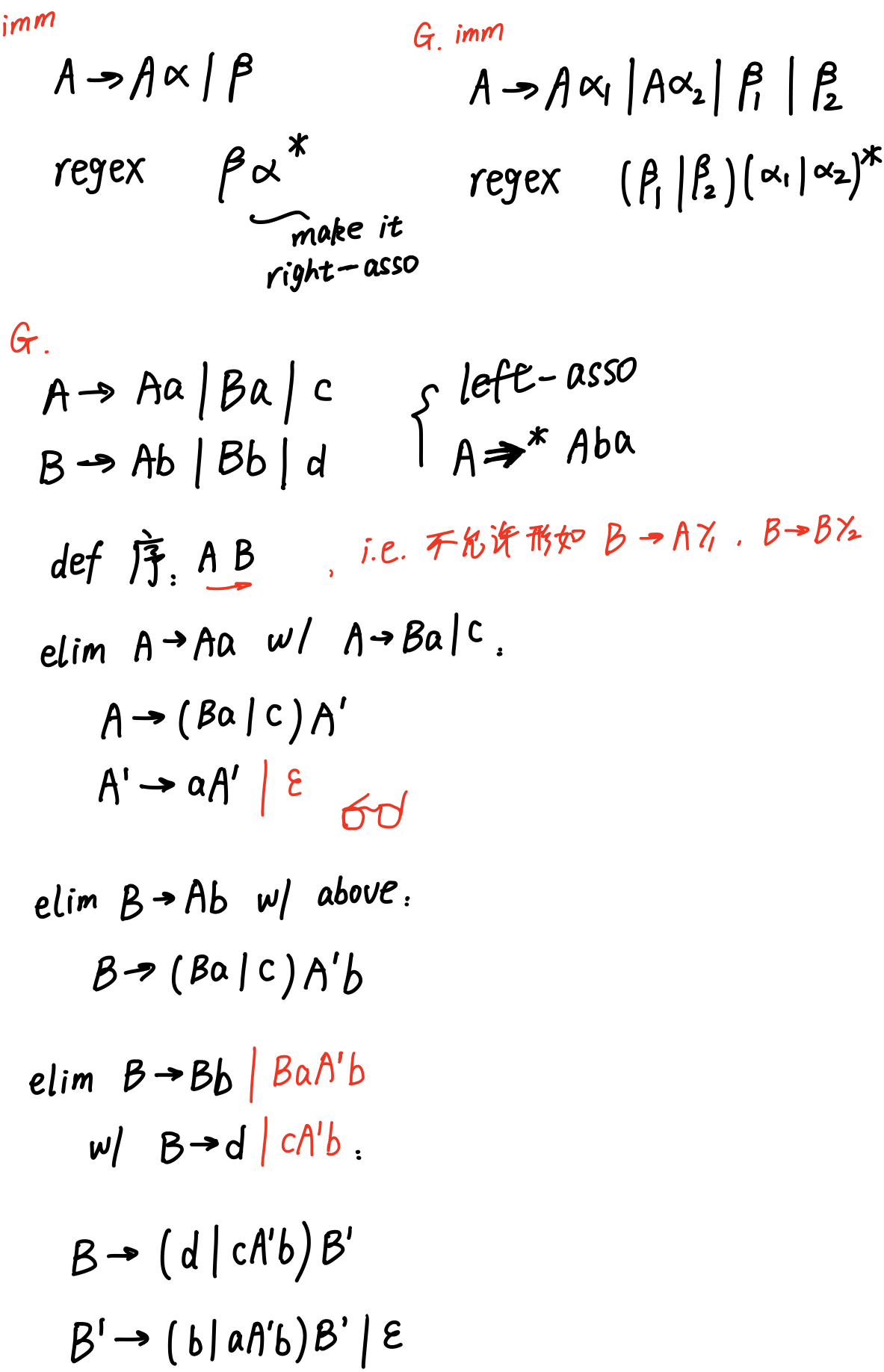
**Recursive-Descent Parsing**: right hand side of the grammar for A specifies the structure of the code for this procedure; requires the use of EBNF.

**LL(1) Parsing**: left to right, leftmost derivation, one token lookahead.

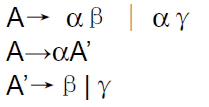
**generate & match**



**Left Recursion Removal:**



**Left Factoring**: kill common prefix string



**first set**: *gen*

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. If for some i<n, all the sets First(X1).. First(Xi) contains ε, then First(X) also contains First(Xi+1)-{ε}.

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

**follow set**: *follow: right next to*

Given a non-terminal A,

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

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

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

*context <-> possibility*

*Talks about what if A=>\*ε ,*

*哪些终结符可能跟在后边*

*img w the cur\_nt is A:*

*possibly derived from B=>\*αA*

*w the cur\_nt is B:*

*context where B is derived*

Construct LL(1) parsing table w/ acquired First(α) and Follow(A) sets:

for each 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 token *a* of Follow(A), add A→α to M[A,*a*].

**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.

*1. For token a in the intersection, M[A,a] has 2 rules A→αi|αj*

*2. For token a in the intersection, M[A,a] has 2 rules A→ε | A→α s.t. α=>\*aβ*

**Chapter 5 Bottom-Up Parsing**

**shift & reduce & accept**



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

of the right sentential form E+n.

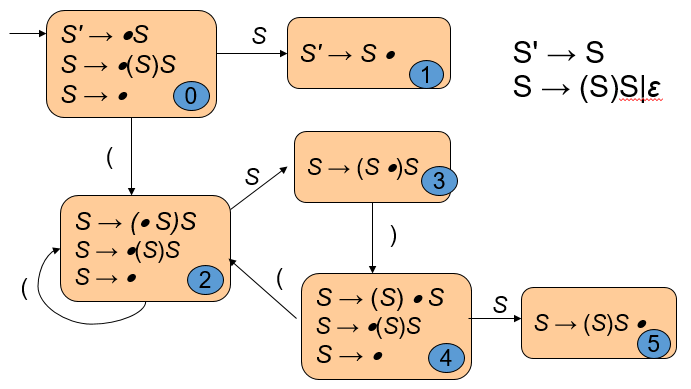
The sequence of symbols on the parsing stack is called a viable prefix of the right sentential form .

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

grammars are always augmented with a ***new start symbol*: *S’***

**做题时先写个推导出来再匹配**

**LR(0):**



*Core: merge ε-closure for each state*

***Kernel item****: those that originate a state as targets of non-ε-transitions*

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.

State trans on an input token or non-terminal (“Goto” col)

**SLR(1):** diff规约时检查是否在follow set

If state s contains the complete item A → γ·, and *the next token in the input string is in Follow(A),* then the action is to reduce by the rule A → γ.

A grammar is SLR(1) iff, for any state s:

*No shift-reduce conflict*: 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).

*No reduce-reduce conflict*: For any two complete items A→α· and B→β· in s, Follow(A) ∩ Follow(B) is empty.

**LR(1):**

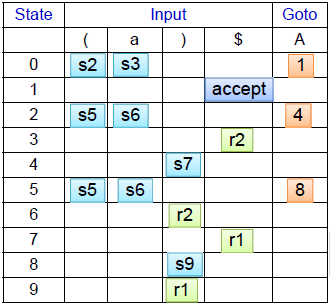
*Definition of LR(1) transitions (part 1: shift)*: Given an LR(1) item [A→α·Xγ, a], where X is any symbol (terminal or nonterminal), there is a transition on X to the item [A→ αX·γ, a]

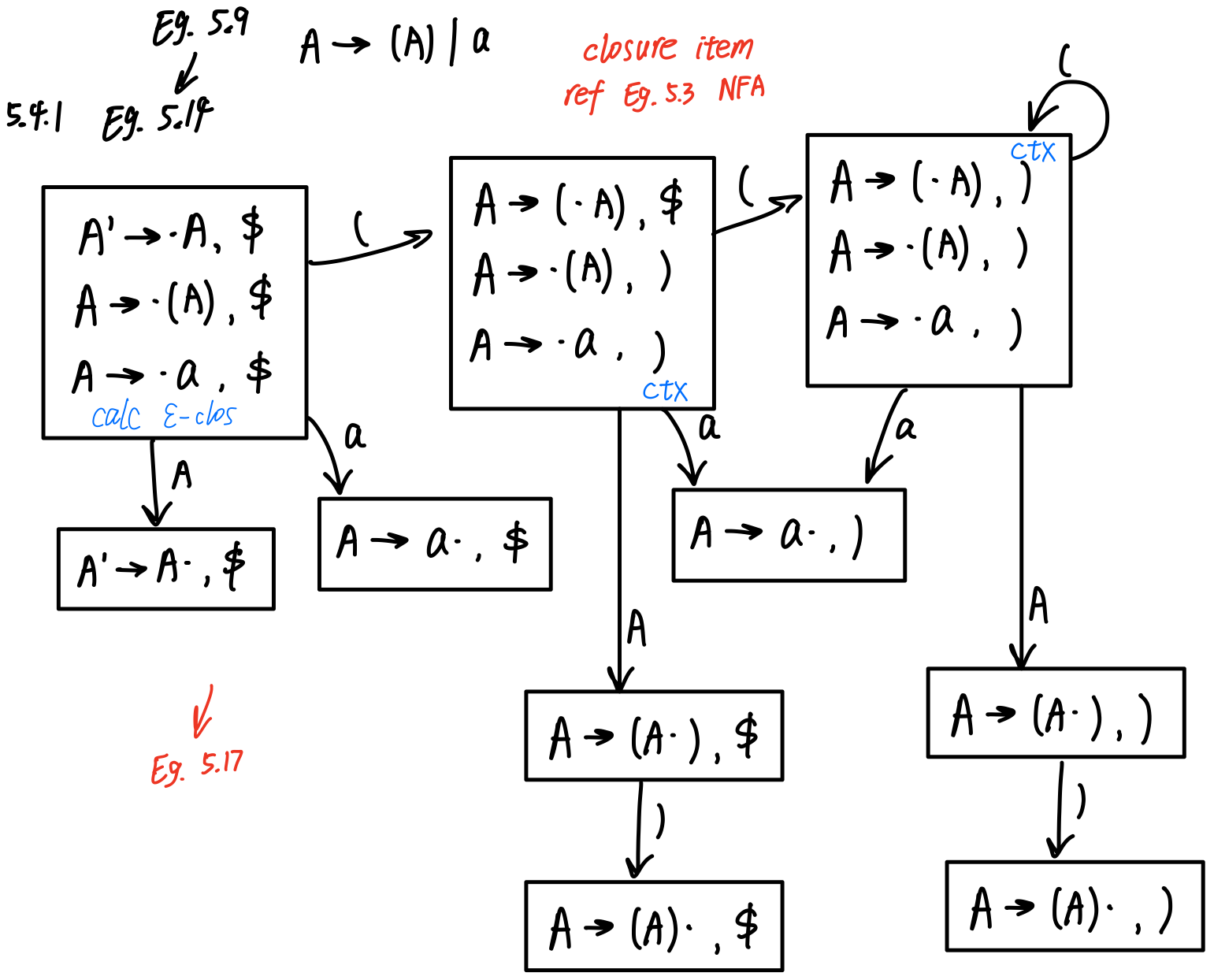
*Definition of LR(1) transitions (part 2)*: Given an LR(1) item [A→α·Bγ, a], where B is a nonterminal, there are ε-transitions to items [B→·β,b] for every production B →βand for every token b in First(γa).

A grammar is LR(1) iff, for any state s:

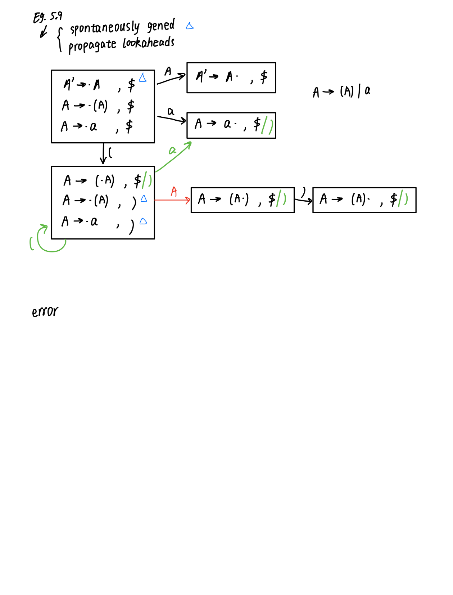
1. For any item [A→α·Xβ,a] in s with X a terminal, there is no item in s of the form [B→γ·,X] (shift-reduce conflict);

2. There are no two items in s of the form [A→α·, a] and [B→β·,a] (reduce-reduce conflict).

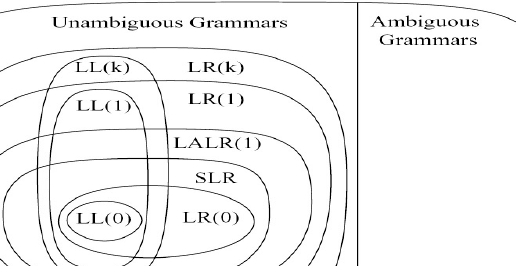




**LALR(1):**



* If a grammar is LR(1), then the LALR(1) parsing table cannot have any shift-reduce conflicts, there may be reduce-reduce conflicts.
* LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(l) parsing.
* LALR( 1 ) 文法, LALR( 1 ) 较 general LR parsing 劣势：in the presence of errors, some spurious reductions may be made before error is declared



**Chapter 6 Semantic Analysis**

**属性文法的分类**

An attribute is **synthesized** if 所有属性都是从子节点计算而来的

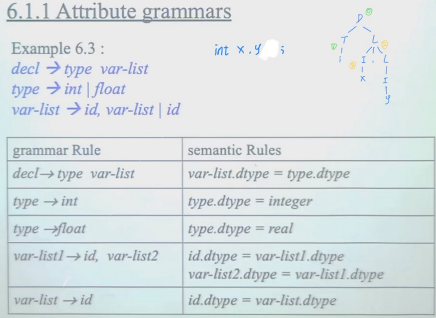
Else an **inherited** attribute.

**S-attributed grammar:** 一个属性文法中所有的属性都是合成的

**L-attributed**:（从左向右）对于继承属性，在分析树中没有从右指向左的依赖

S-attributed grammar is L-attributed ~.

**计算**

****

Inherited attributes: be computed in preorder, often be treated as parameters of the call.

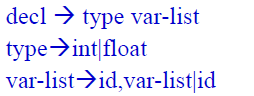
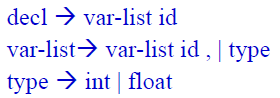
Synthesized attributes: be computed in postorder, often be treated as returned values of the call.

**Value stack**: Computing synthesized attributes during LR parsing

(Inheriting a previously computed synthesized attributes) dealing with inherited attributes in LR parsing: use external data structures to hold inherited attribute values.

**改写文法 便利属性计算**

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

非递归的依赖由相应父节点的操作实现

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.

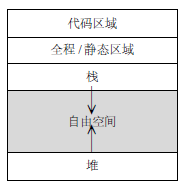
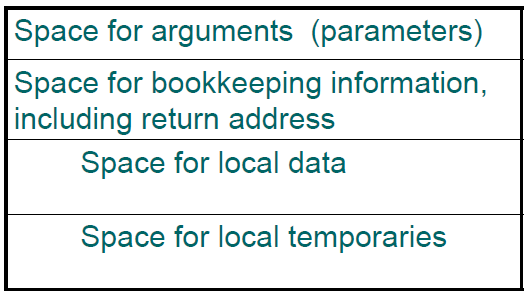
Scope Rules: declaration before use / the most closely nested rule for block structure

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

Type equivalence: Structural equivalence / Name equivalence /

Declaration equivalence: weaker version of name equivalence; t2 = t1; are interpreted as establishing type aliases

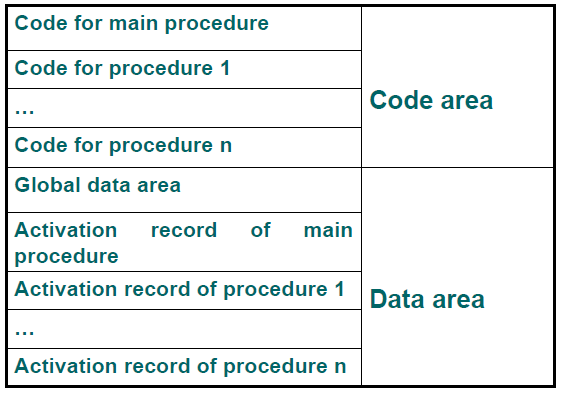
**Chapter 7 Run-Time Environments**

**Fully Static Runtime Environments:**

no pointer or dynamic allocation.

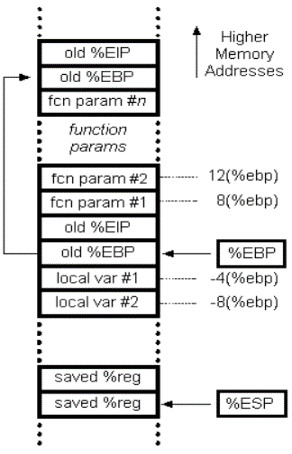
no recursive procedure calling



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 pointer to the fp of the immediately preceding AR

**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**

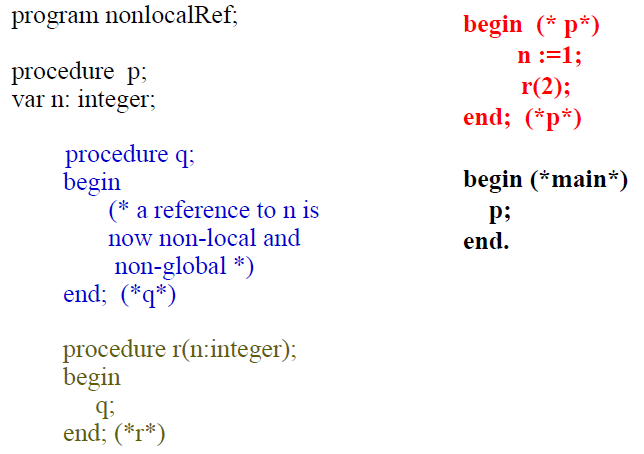
Another option is to use a processor mechanism such as ap(argument pointer) in VAX arch.

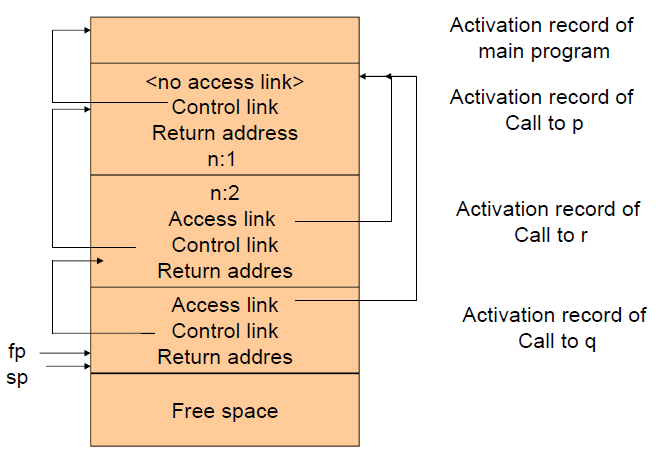
**Stack-Based Environment with local**

**Procedures**

**Access link** represents the defining environment of the procedure; aka **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.

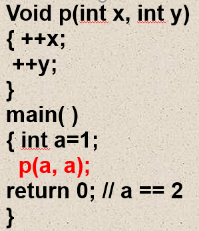
Automatic: garbage collection:

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**:

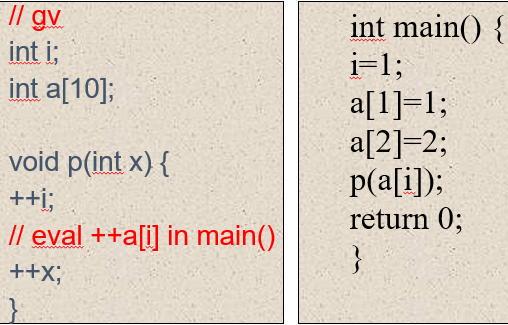
value / reference /

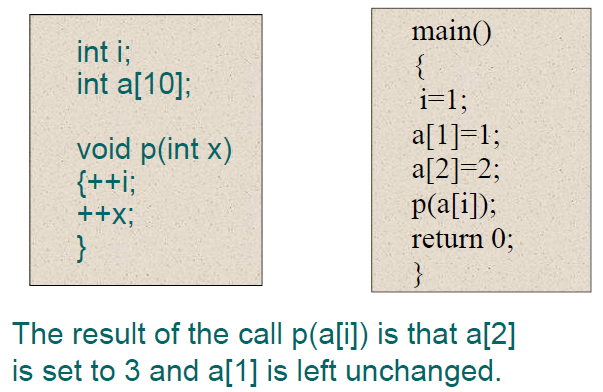
value-result: copy-in, copy-out

≈reference, except that一个 arg 对两个 param: 呕

/ name: delayed evaluation

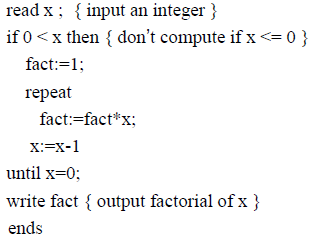
(1)  The text of an argument at point of call is viewed as a function in its own right. (2)  The arguments are evaluated every time the parameter name is ref-ed in the procedure. (3)  The argument will always be evaluated in the caller’s environment.





**Chapter 8 Code Generation**

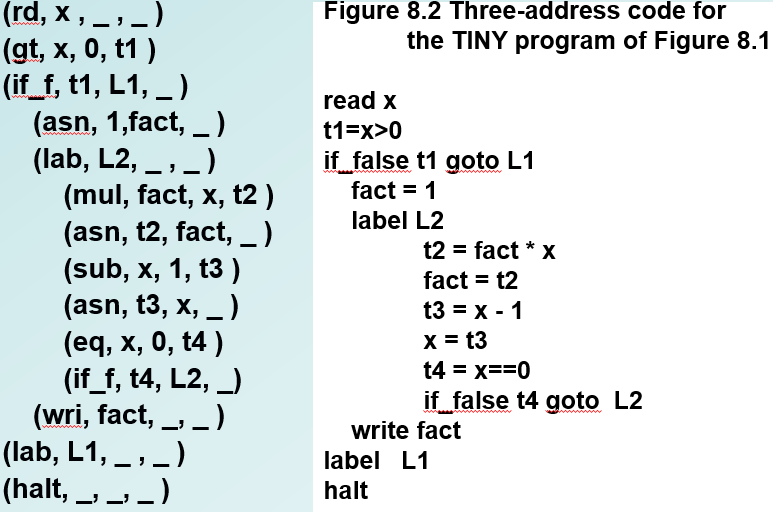
Depends on the characteristics of the source language, info about the target arc~, the structure of the runtime env, and OS running on the target machine.

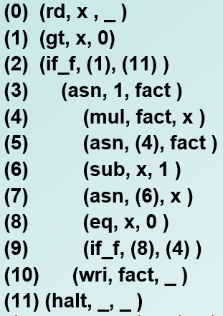


**Quadruple implementation:**

(op, src1, src2, dst)

Typedef enum { rd, gt, if\_f, asn, lab, mul, sub, eq, wri, halt, …} OpKind;



**Triples**

**Triples** have one major drawback: any movement of their positions becomes difficult**.**

**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.

ldc 2 ; load constant 2 *(pushes 2 onto the stack)*

lod a ; load value of variable a *(pushes a onto the stack)*

mpi ; integer multiplication *(pops these two values from the stack, multiplies them (in reverse order), and pushes the result onto the stack.)*

sbi ; integer subtraction *(subtracts the first from the second)*

adi ; integer addition

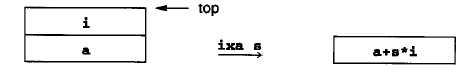
lda x ; load address of x

sto ; store top to address below top & pop both

stn : stores the value to the address but leaves the value at the top of the stack, while discarding the address.



ind i (“ indirect load”)



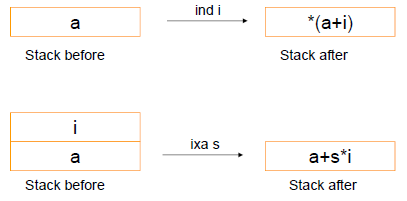
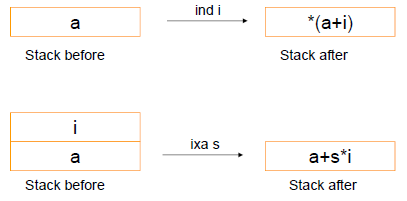
ixa s (“indexed address”)

Mst mark stack

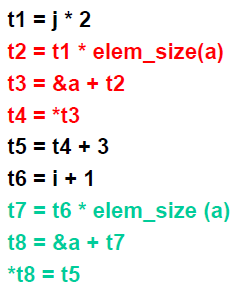
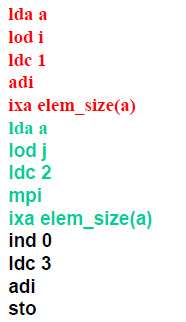
Cup call user procedure

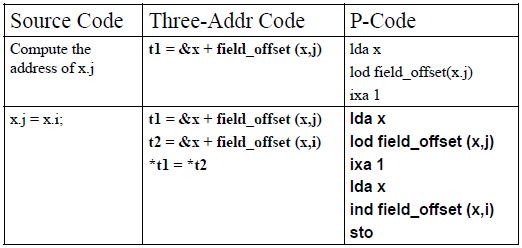
参数入栈顺序与 C 相反

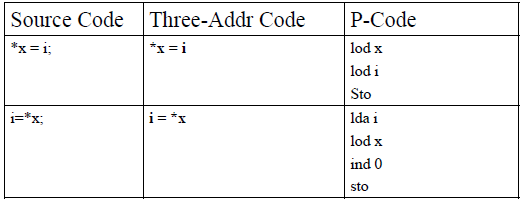
**Generation of Target Code from Intermediate Code:** macro expansion & static simulation

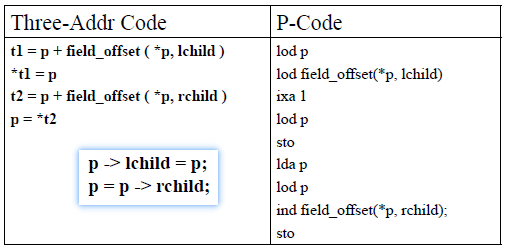


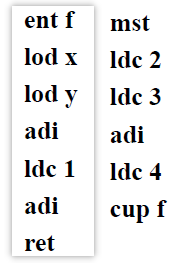
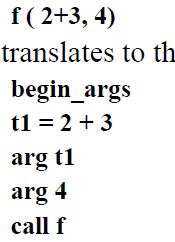
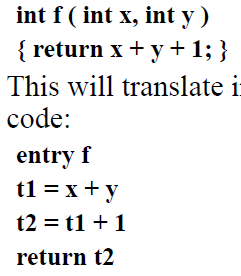
a[i+1] = a[j\*2]+3









**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