Compiler course

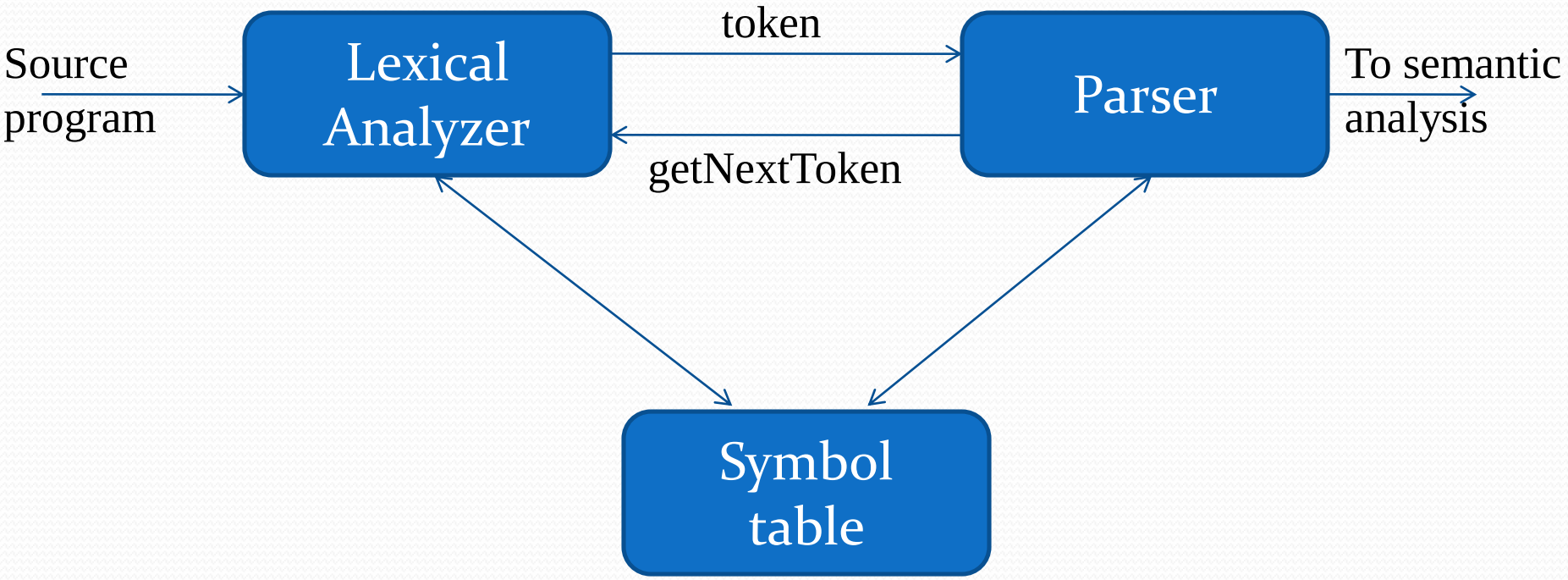
Chapter 1

Lexical Analysis

Outline

- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens
- Lexical analyzer generator
- Finite automata
- Design of lexical analyzer generator

The role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

printf("total = %d\n", score);

Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

Sentinels


```
Switch (*forward++) {
    case eof:
            if (forward is at end of first buffer) {
                    reload second buffer;
                    forward = beginning of second buffer;
            }
            else if {forward is at end of second buffer) {
                    reload first buffer;\
                    forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                    terminate lexical analysis;
            break;
    cases for the other characters;
}
```

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

- Example:

  letter_ -> A | B | … | Z | a | b | … | Z | _

  digit -> 0 | 1 | … | 9

  id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε

expr -> term **relop** term
| term

term -> **id**
| **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

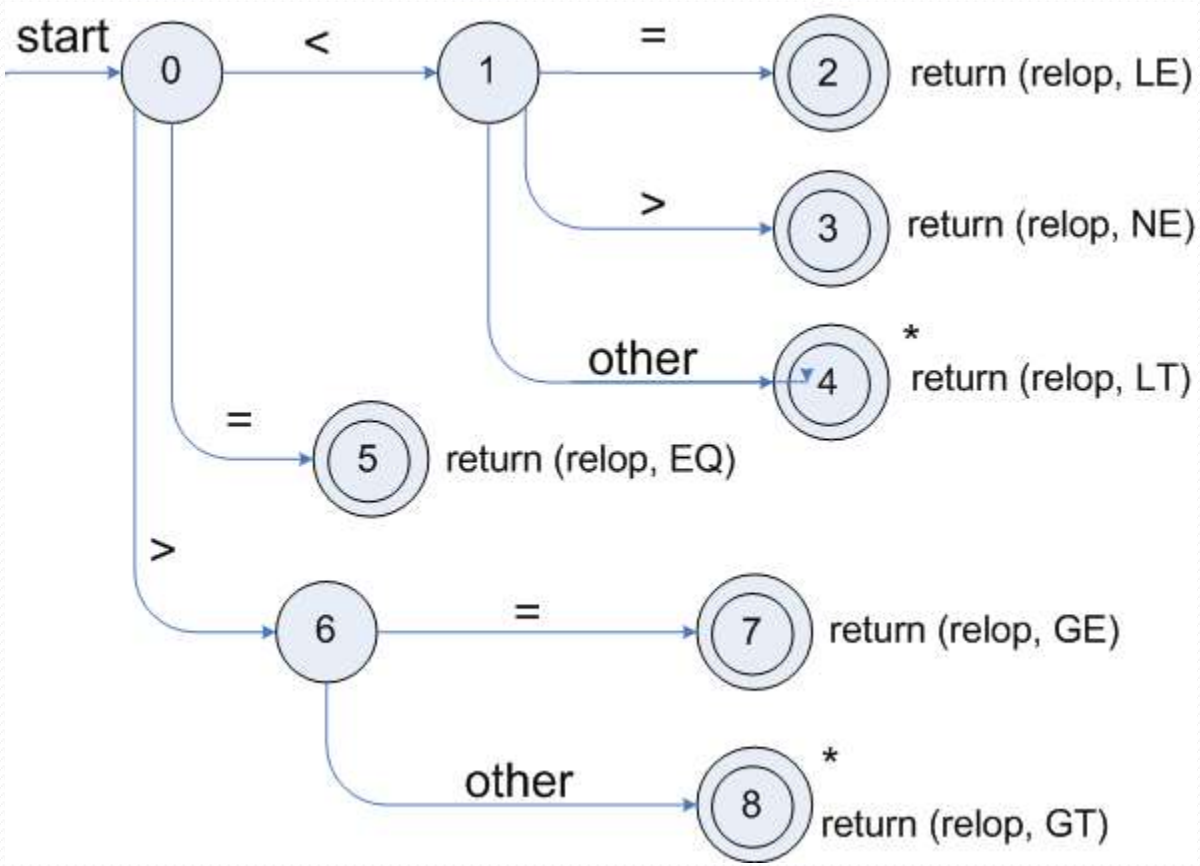
  digit -> [0-9]
  Digits -> digit+
  number -> digit(.digits)? (E[+-]? Digit)?
  letter -> [A-Za-z_]
  id -> letter (letter|digit)*
  If -> if
  Then -> then
  Else -> else
  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

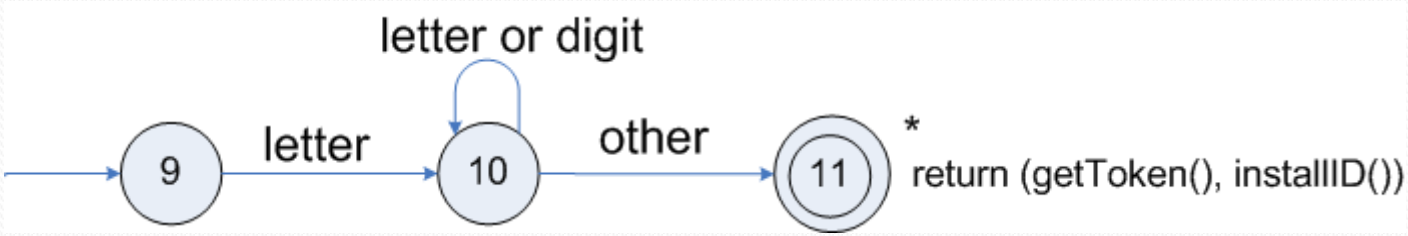
Transition diagrams

- Transition diagram for relop



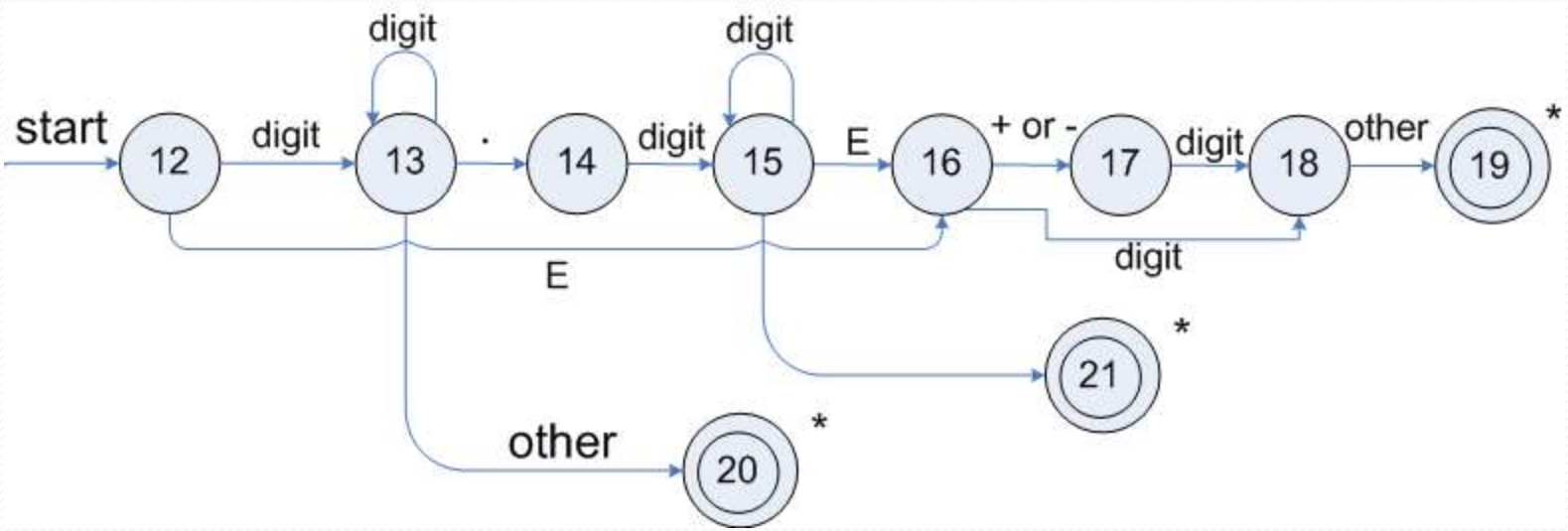
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



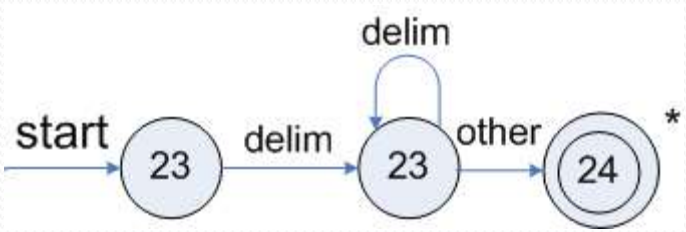
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

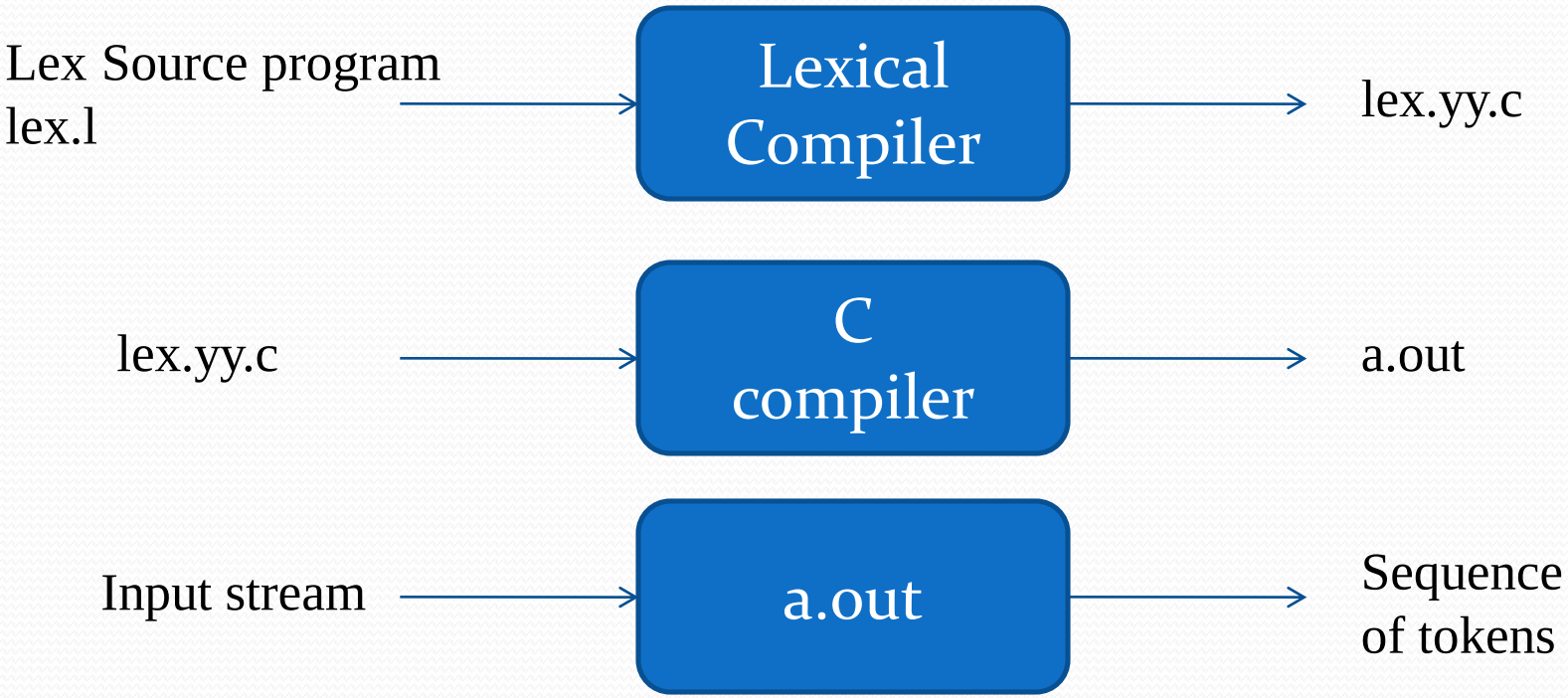
- Transition diagram for whitespace



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {                /* repeat character processing until a
                                        return or failure occurs  */
    switch(state) {
            case 0: c= nextchar();
                        if (c == '<') state = 1;
                        else if (c == '=') state = 5;
                        else if (c == '>') state = 6;
                        else fail();  /* lexeme is not a relop */
                        break;
            case 1: ...
            ...
            case 8: retract();
                        retToken.attribute = GT;
                        return(retToken);
    }
```

Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules → Pattern {Action}
%%
auxiliary functions

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
...
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

- Is read

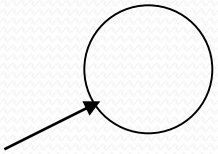
In state s_1 on input "a" go to state s_2

- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

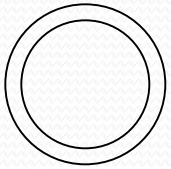
Finite Automata State Graphs

- A state

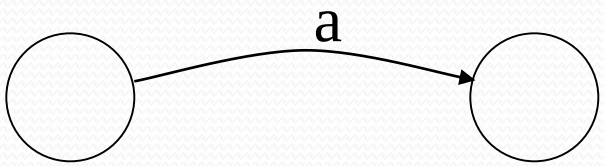
- The start state



- An accepting state

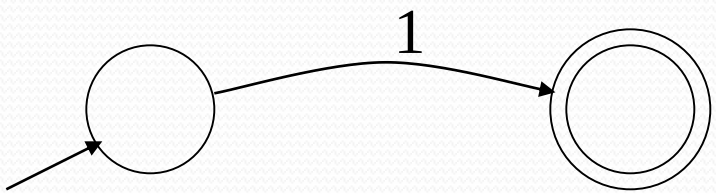


- A transition



A Simple Example

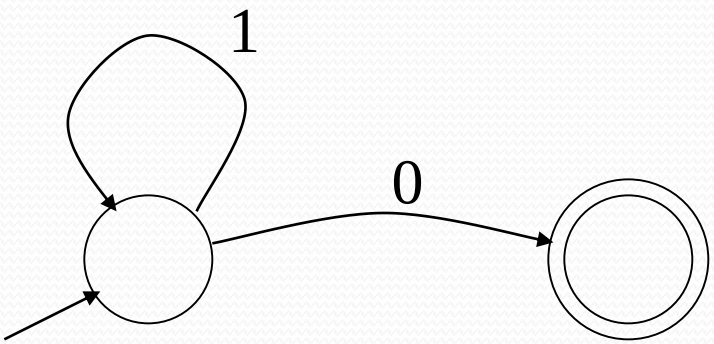
- A finite automaton that accepts only "1"



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

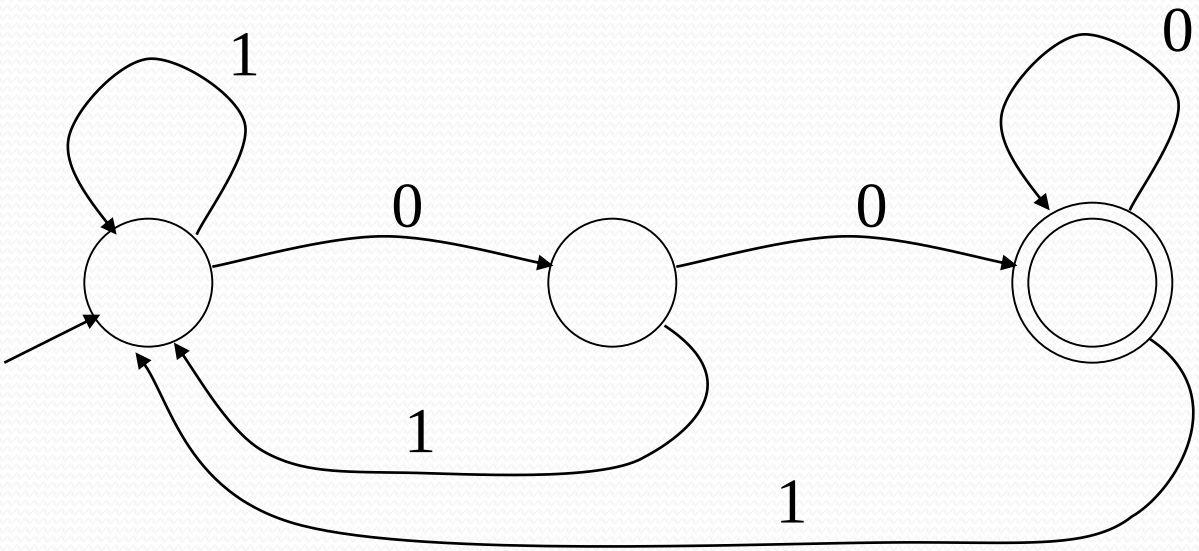
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

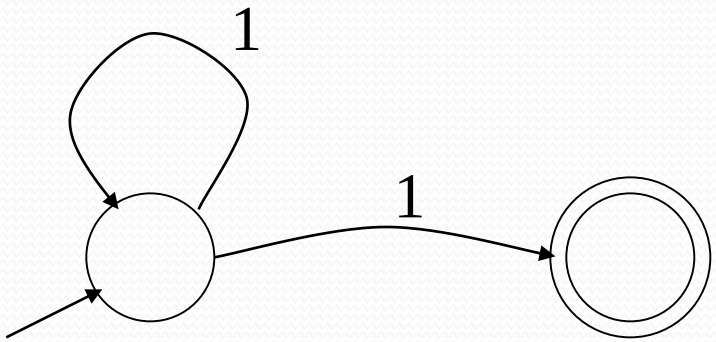
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

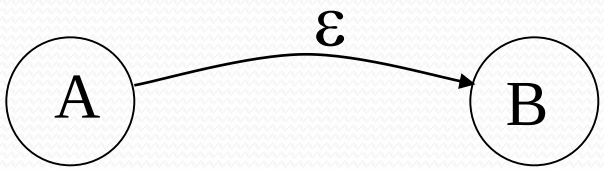
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

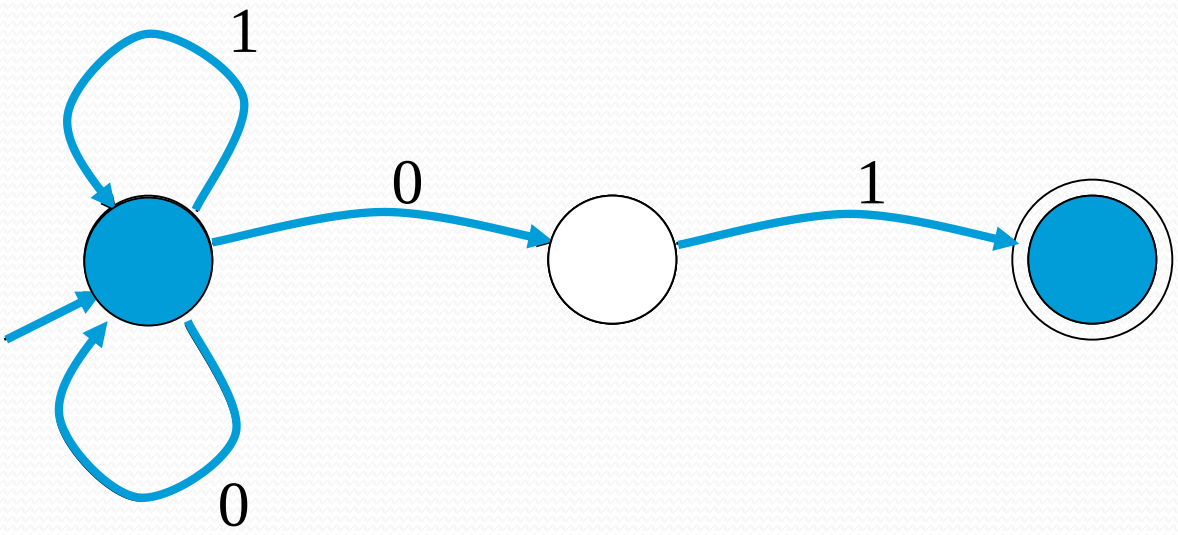
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

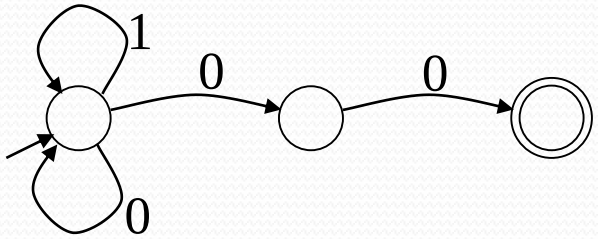
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

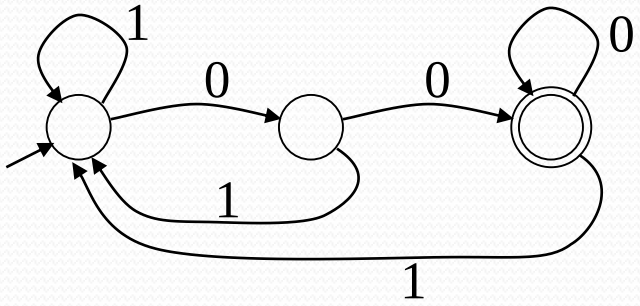
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



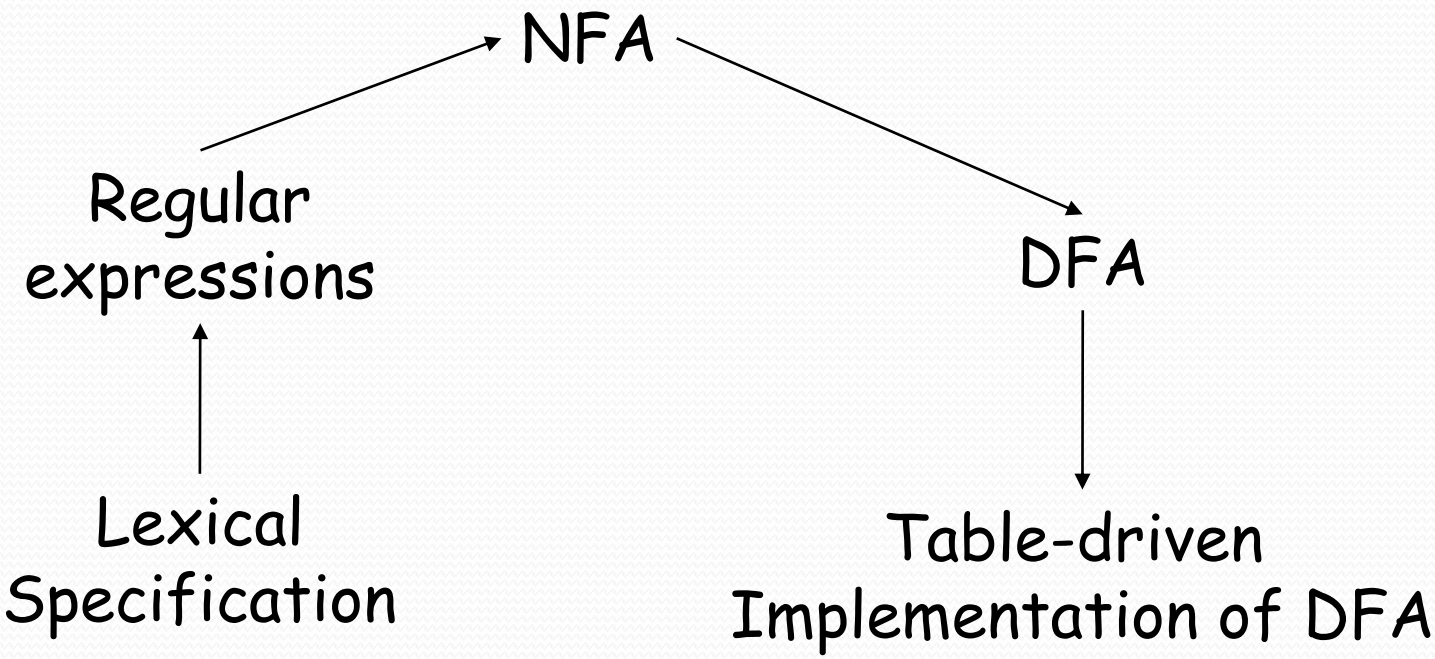
DFA



- DFA can be exponentially larger than NFA

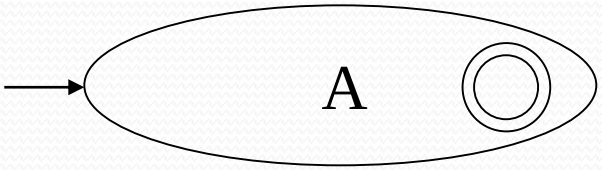
Regular Expressions to Finite Automata

- High-level sketch

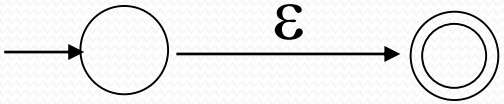


Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



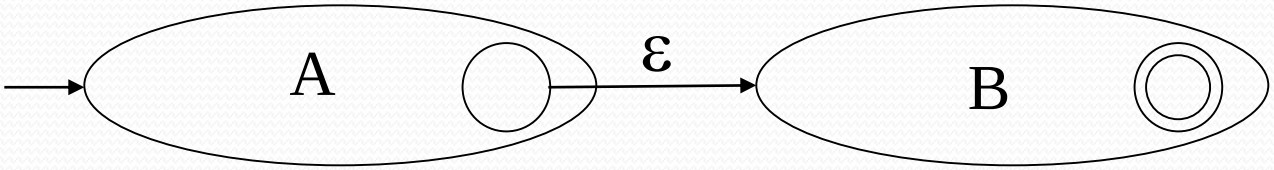
- For ε



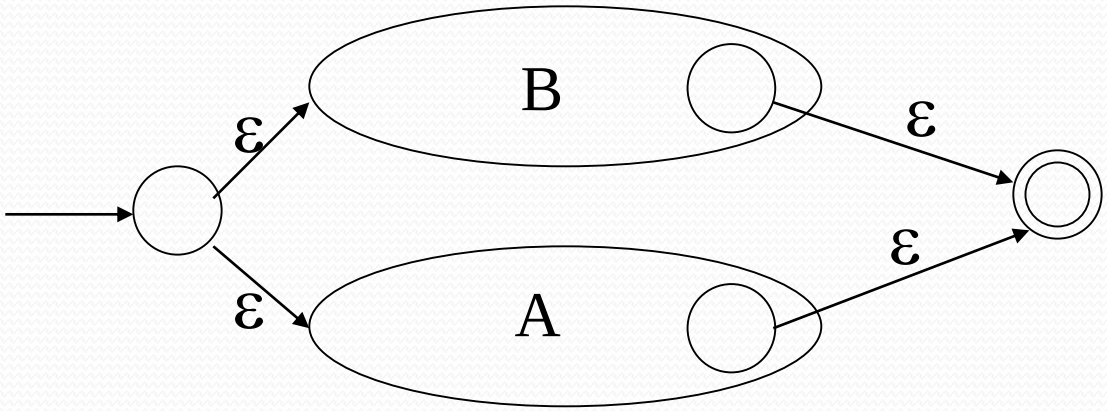
- For input a

Regular Expressions to NFA (2)

- For AB

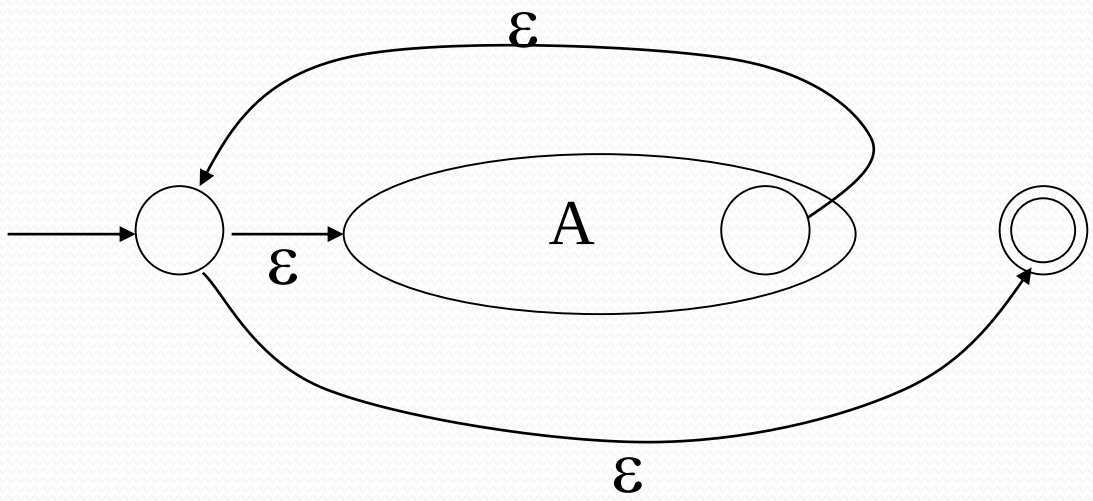


- For A | B



Regular Expressions to NFA (3)

- For A*

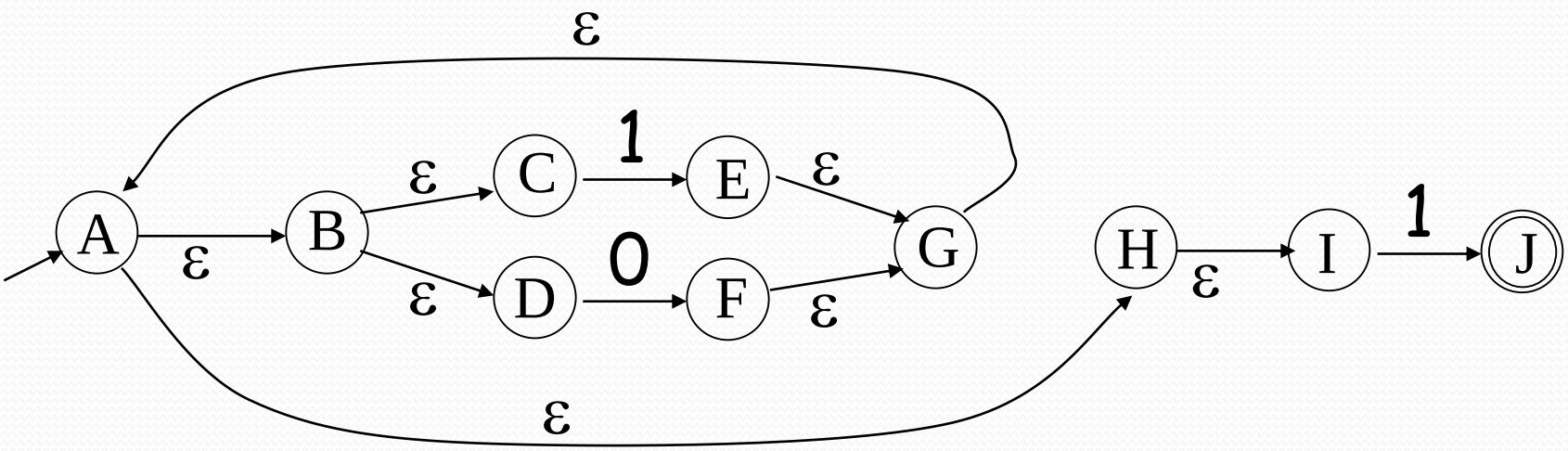


Example of RegExp -> NFA conversion

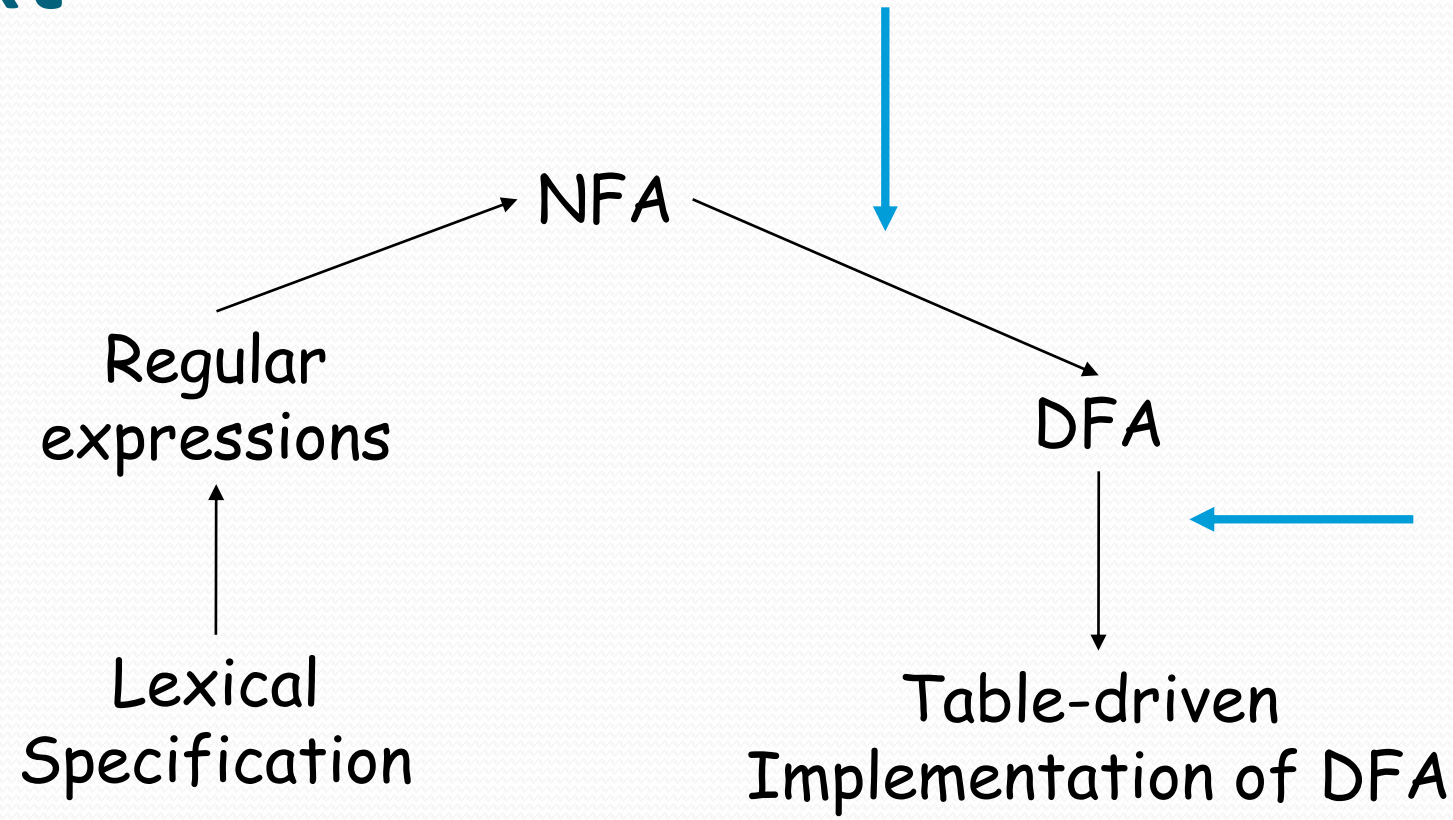
- Consider the regular expression

$$(1 \mid 0)^*1$$

- The NFA is



Next


NFA
Regular expressions
DFA
Lexical Specification
Table-driven Implementation of DFA

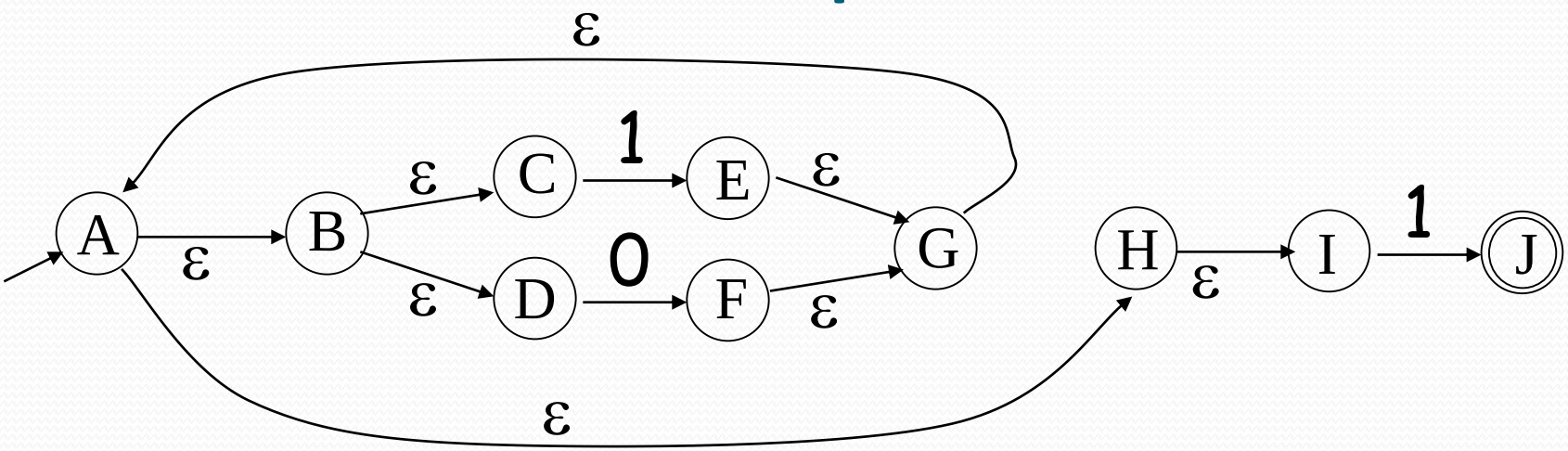
NFA to DFA. The Trick

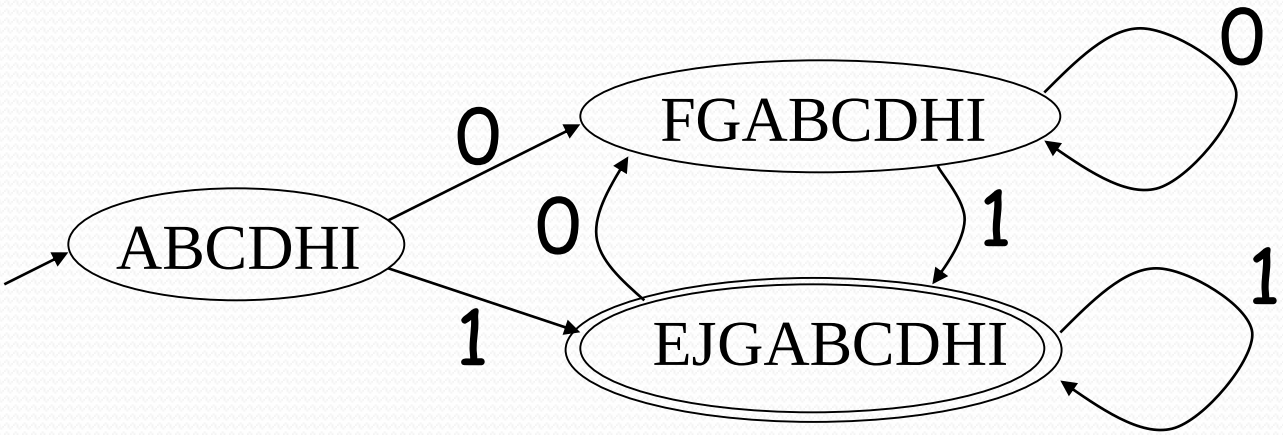
- Simulate the NFA
- Each state of resulting DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example





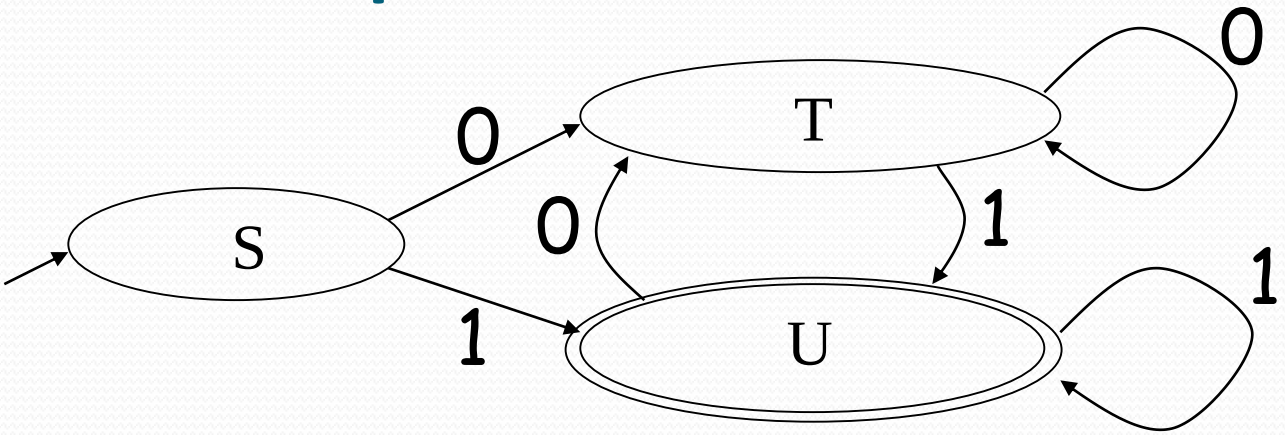
NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jflex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

Readings

- Chapter 3 of the book

Compiler course

Chapter 8

Code Generation

Outline

- Code Generation Issues
- Target language Issues
- Addresses in Target Code
- Basic Blocks and Flow Graphs
- Optimizations of Basic Blocks
- A Simple Code Generator
- Peephole optimization
- Register allocation and assignment
- Instruction selection by tree rewriting

Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Insrtuction ordering

Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
  - IR + Symbol table
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected
- The target program
  - Common target architectures are: RISC, CISC and Stack based machines
  - In this chapter we use a very simple RISC-like computer with addition of some CISC-like addressing modes

complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD      R0, y
ADD     R0, R0, z
ST      x, R0
```

a=b+c
d=a+e

```
LD      R0, b
ADD     R0, R0, c
ST      a, R0
LD      R0, a
ADD     R0, R0, e
ST      d, R0
```

Register allocation

- Two subproblems
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Resister assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

L       R1, a
A       R1, b
M       R0, c
D       R0, d
ST      R1, t
```

```
t=a+b
t=t+c
T=t/d

L       R0, a
A       R0, b
M       R0, c
SRDA    R0, 32
D       R0, d
ST      R1, t
```

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means R1=contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: *r and *100(r)
- immediate constant addressing mode: like LD R1, #100

b = a [i]

```
LD R1, i                    //R1 = i
MUL R1, R1, 8               //R1 = Rl * 8
LD R2, a(R1)
        //R2=contents(a+contents(R1))
ST b, R2                    //b = R2
```

a[j] = c

```
LD R1, c                    //R1 = c
LD R2, j                    // R2 = j
MUL R2, R2, 8               //R2 = R2 * 8
ST  a(R2), R1
        //contents(a+contents(R2))=R1
```

x=*p

LD R1, p **//R1 = p**

LD R2, 0(R1) **// R2 =**

contents(0+contents(R1))

ST x, R2 **// x=R2**

conditional-jump three-address instruction

If x<y goto L

```
LD R1, x          // R1 = x
LD R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M        // i f R1 < 0 jump t o M
```

costs associated with the addressing modes

- LD R0, R1 cost = 1
- LD R0, M cost = 2
- LD R1, *100(R2) cost = 3

Addresses in the Target Code

- A statically determined area Code
- A statically determined data area Static
- A dynamically managed area Heap
- A dynamically managed area Stack

three-address statements for procedure calls and returns

- call callee
- Return
- Halt
- action

Target program for a sample call and return

```
                    // code for c
action1
call p
action2
halt
                    // code for p
action3
return
```

```
                        // code for c
100:  ACTION1           // code for action1
120:  ST 364, #140      // save return address 140 in location 364
132:  BR 200            // call p
140:  ACTION2
160:  HALT              // return to operating system
      ...
                         // code for p
200:  ACTION3
220:  BR *364           // return to address saved in location 364
      ...
                        // 300-363 hold activation record for c
300:                    // return address
304:                    // local data for c
      ...
                        // 364-451 hold activation record for p
364:                    // return address
368:                    // local data for p
```

Stack Allocation

```
LD    SP, #stackStart                    // initialize the stack
code for the first procedure
HALT                                     // terminate execution

ADD   SP, SP, #caller.recordSize         // increment stack pointer
ST    *SP, #here + 16                    // save return address
BR    callee.codeArea                    // return to caller
```

Branch to called procedure

Return to caller

in Callee: BR *0(SP)

in caller: SUB SP, SP, #caller.recordsize

Target code for stack allocation

```
                    // code for m
action1
call q
action2
halt

                    // code for p
action3
return

                    // code for q
action4
call p
action5
call q
action6
call q
return
```

```
                                    // code for m
100:  LD SP, #600                   // initialize the stack
108:  ACTION1                       // code for action1
128:  ADD SP, SP, #msize            // call sequence begins
136:  ST *SP, #152                  // push return address
144:  BR 300                        // call q
152:  SUB SP, SP, #msize            // restore SP
160:  ACTION1 2
180:  HALT
      ...
                                    // code for p
200:  ACTION3
220:  BR *0(SP)                     // return
      ...
                                    // code for q
300:  ACTION4                       // contains a conditional jump to 456
320:  ADD SP, SP, #qsize
328:  ST *SP, #344                  // push return address
336:  BR 200                        // call p
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  BR *SP, #396                  // push return address
388:  BR 300                        // call q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST *SP, #440                  // push return address
440:  BR 300                        // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                     // return
      ...
600:                                // stack starts here
```

Basic blocks and flow graphs

- Partition the intermediate code into basic blocks
  - The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.
- The basic blocks become the nodes of a flow graph

rules for finding leaders

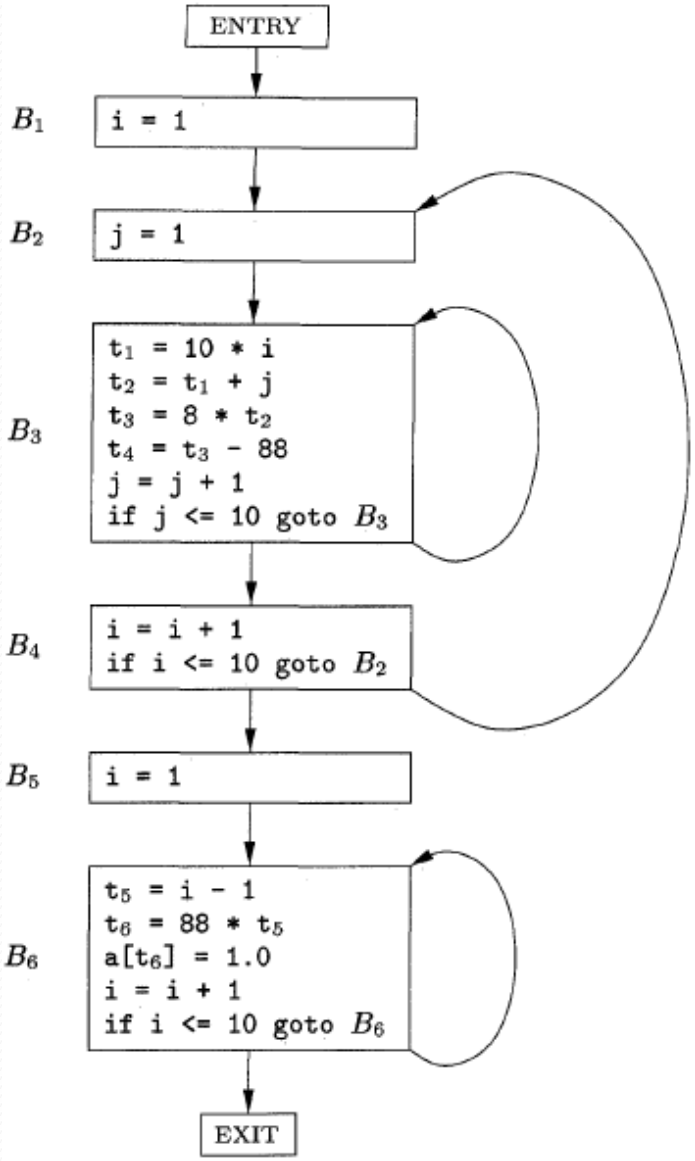
- The first three-address instruction in the intermediate code is a leader.
- Any instruction that is the target of a conditional or unconditional jump is a leader.
- Any instruction that immediately follows a conditional or unconditional jump is a leader.

Intermediate code to set a 10*10 matrix to an identity matrix

```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i, j] = 0.0;
for i from 1 to 10 do
        a[i, i] = 1.0;
```

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Flow graph based on Basic Blocks



liveness and next-use information

- We wish to determine for each three address statement x=y+z what the next uses of x, y and z are.
- Algorithm:
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
  - In the symbol table, set x to "not live" and "no next use.“
  - In the symbol table, set y and *z* to "live" and the next uses of y and *z* to i.

DAG representation of basic blocks

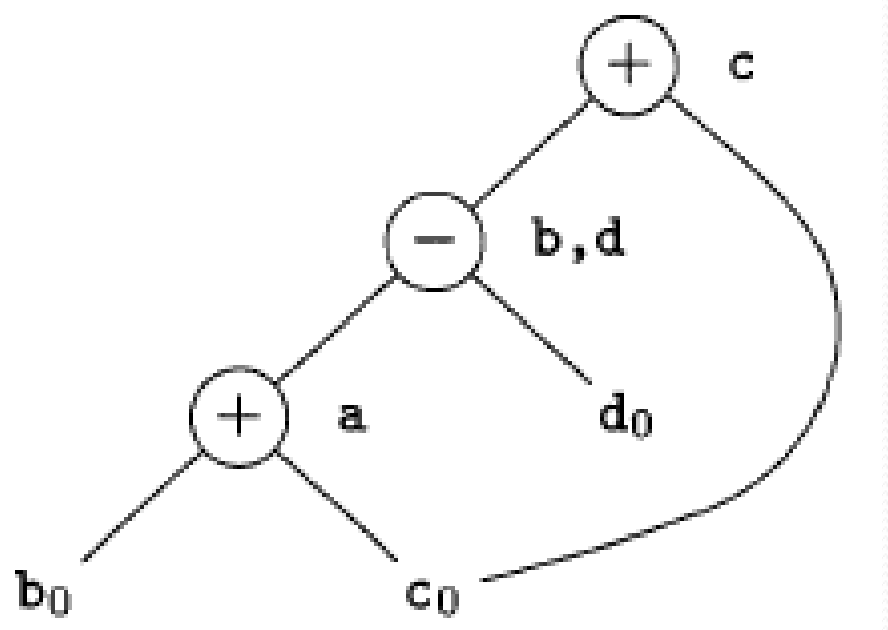
- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
- Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block.

Code improving transformations

- We can eliminate *local common subexpressions,* that is, instructions that compute a value that has already been computed.
- We can eliminate *dead code,* that is, instructions that compute a value that is never used.
- We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- We can apply algebraic laws to reorder operands of three-address instructions, and sometimes t hereby simplify t he computation.

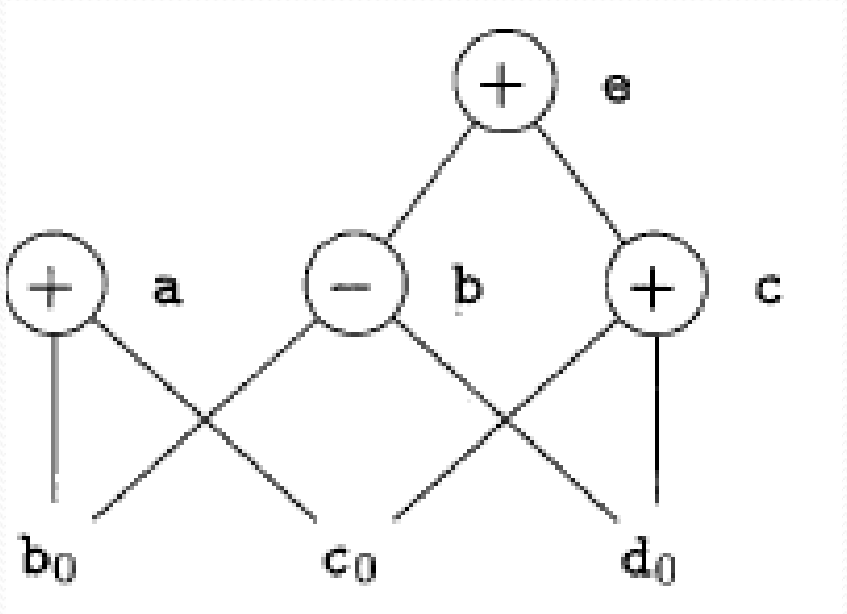
DAG for basic block

```
a = b + c
b = a - d
c = b + c
d = a - d
```



DAG for basic block

```
a = b + c;
b = b - d
c = c + d
e = b + c
```

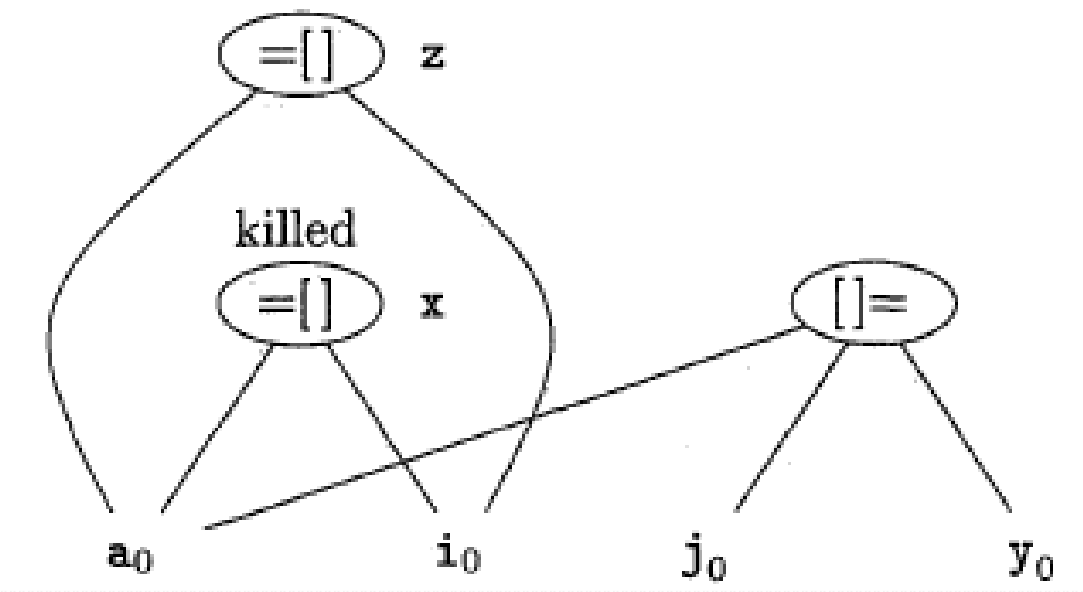


array accesses in a DAG

- An assignment from an array, like x = a [i], is represented by creating a node with operator =[] and two children representing the initial value of the array, ao in this case, and the index i. Variable x becomes a label of this new node.
- An assignment to an array, like a [j] = y, is represented by a new node with operator []= and three children representing ao, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on ao**. A** node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

DAG for a sequence of array assignments

```
x = a[i]
a[j] = y
z = a[i]
```



Rules for reconstructing the basic block from a DAG

- The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.
- Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.
- Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.
- Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.
- Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

principal uses of registers

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries - places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.
- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

Descriptors for data structure

- For each available register, a register descriptor keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

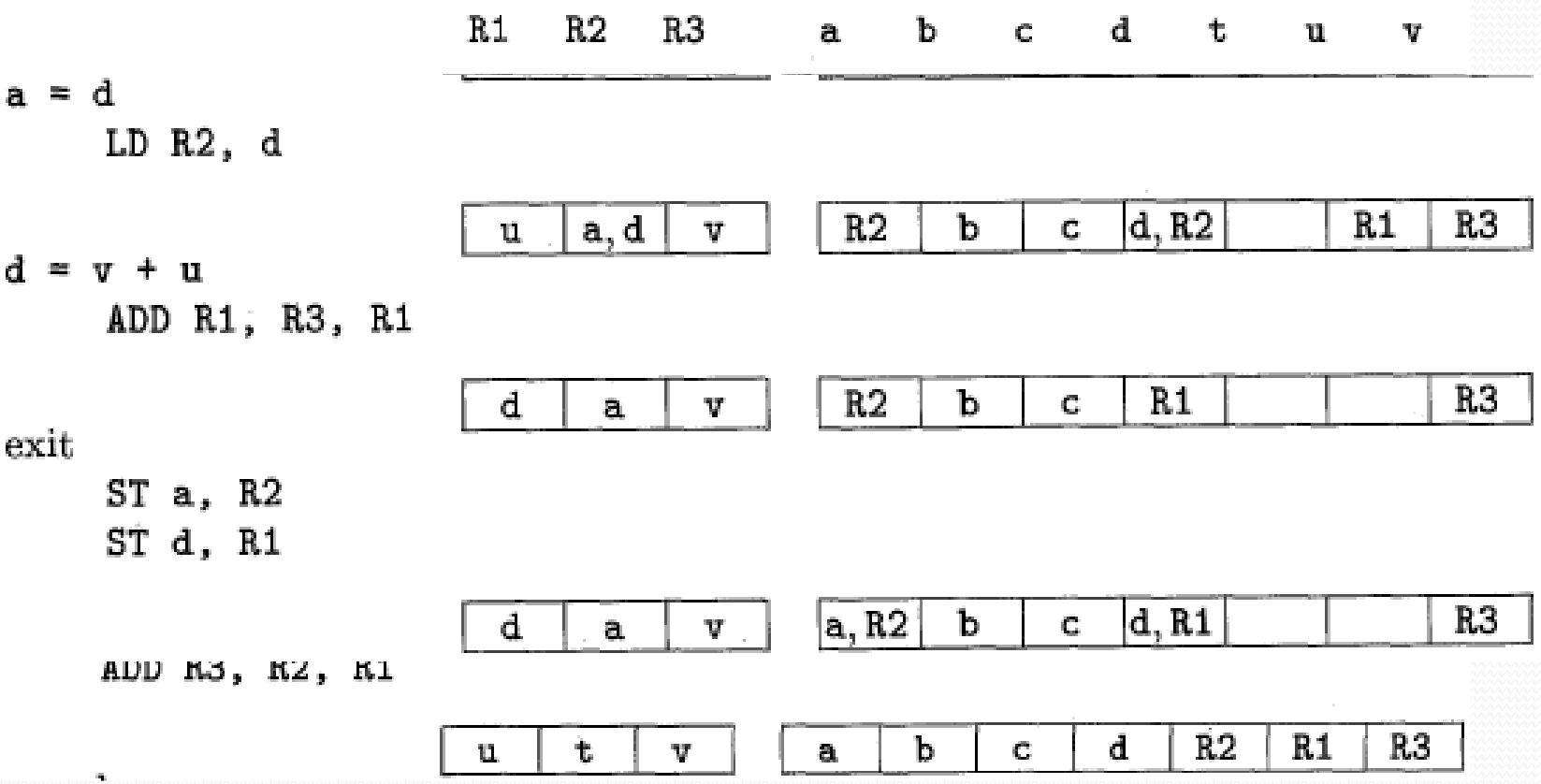
Machine Instructions for Operations

- Use getReg(x = y + z) to select registers for x, y, and z. Call these R_x, R_y and R_z.
- If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).
- Similarly, if z is not in R_z, issue and instruction LD R_z, z', where z' is a location for *x* .
- Issue the instruction ADD R_x , R_y, R_z.

Rules for updating the register and address descriptors

- For the instruction LD R, x
  - Change the register descriptor for register R so it holds only x.
  - Change the address descriptor for x by adding register R as an additional location.
- For the instruction ST x, R, change the address descriptor for x to include its own memory location.
- For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction x = y + *x*
  - Change the register descriptor for R_x so that it holds only *x*.
  - Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
  - Remove R_x from the address descriptor of any variable other than x.
- When we process a copy statement x = y, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule I):
  - Add x to the register descriptor for R_y.
  - Change the address descriptor for x so that its only location is R_y .

Instructions generated and the changes in the register and address descriptors



Rules for picking register R_y for y

- If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.
- If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.
- The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Selection of the register Rx

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for Rx.
2. If y is not used after instruction I, and Ry holds only y after being loaded, Ry can also be used as Rx. A similar option holds regarding z and Rx.

Possibilities for value of R

- If the address descriptor for v says that v is somewhere besides R, then we are OK.
- If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- If we are not OK by one of the first two cases, then we need to generate the store instruction **ST** v, *R* to place a copy of v in its own memory location. This operation is called a spill.

Characteristic of peephole optimizations

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Redundant-instruction elimination

- LD a, Ro

  ST Ro, a
- if debug == 1 goto **L1**

  goto L2

  L I : print debugging information

  L2:

Flow-of-control optimizations

```
goto L1
...
Ll: goto L2
```

Can be replaced by:

```
goto L2
...
Ll: goto L2
```

```
if a<b goto L1
...
Ll: goto L2
```

Can be replaced by:

```
if a<b goto L2
...
Ll: goto L2
```

Algebraic simplifications

- $x = x + 0$
- $x = x * 1$

Register Allocation and Assignment

- Global Register Allocation
- Usage Counts
- Register Assignment for Outer Loops
- Register Allocation by Graph Coloring

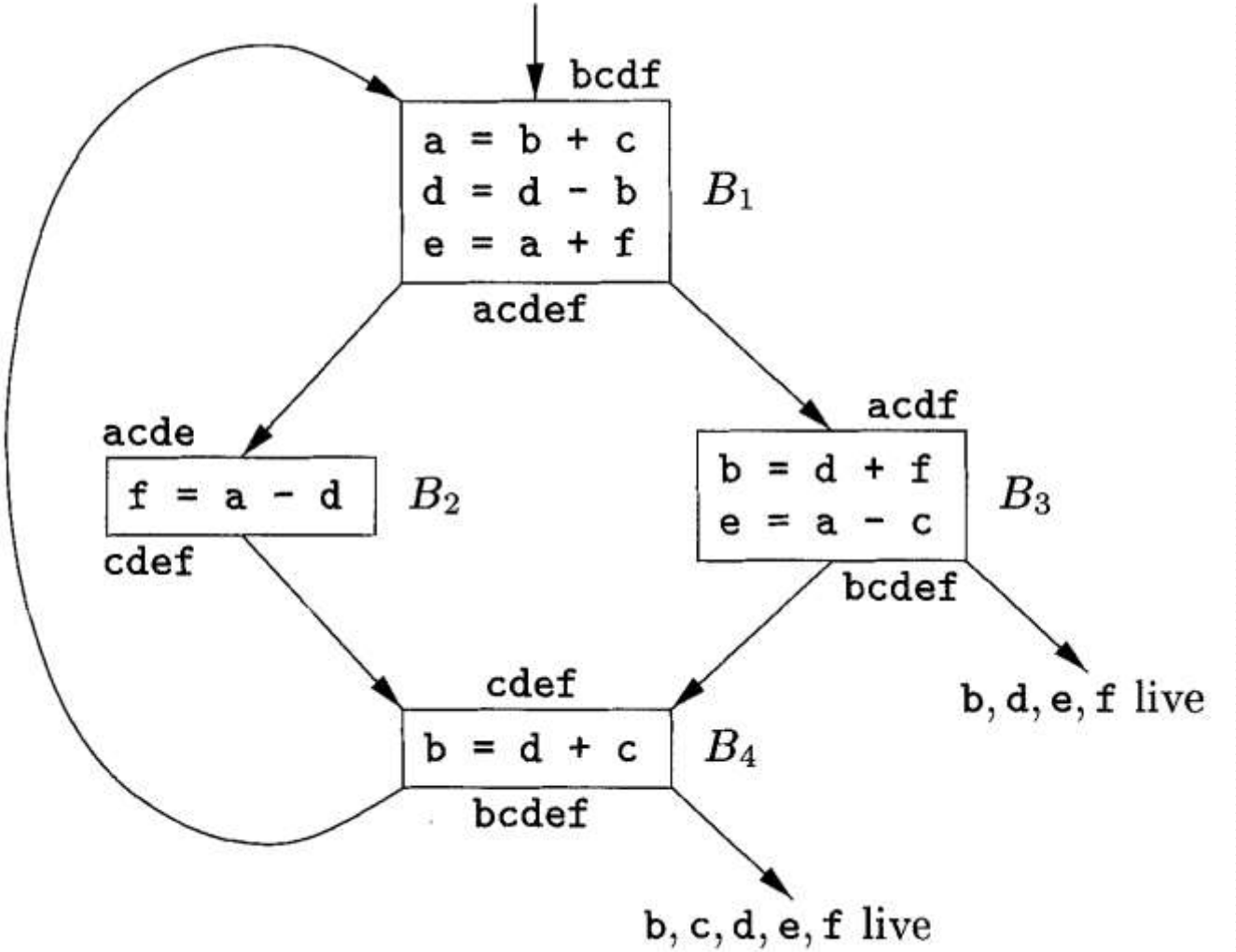
Global register allocation

- Previously explained algorithm does local (block based) register allocation
- This resulted that all live variables be stored at the end of block
- To save some of these stores and their corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally)
- Some options are:
  - Keep values of variables used in loops inside registers
  - Use graph coloring approach for more globally allocation

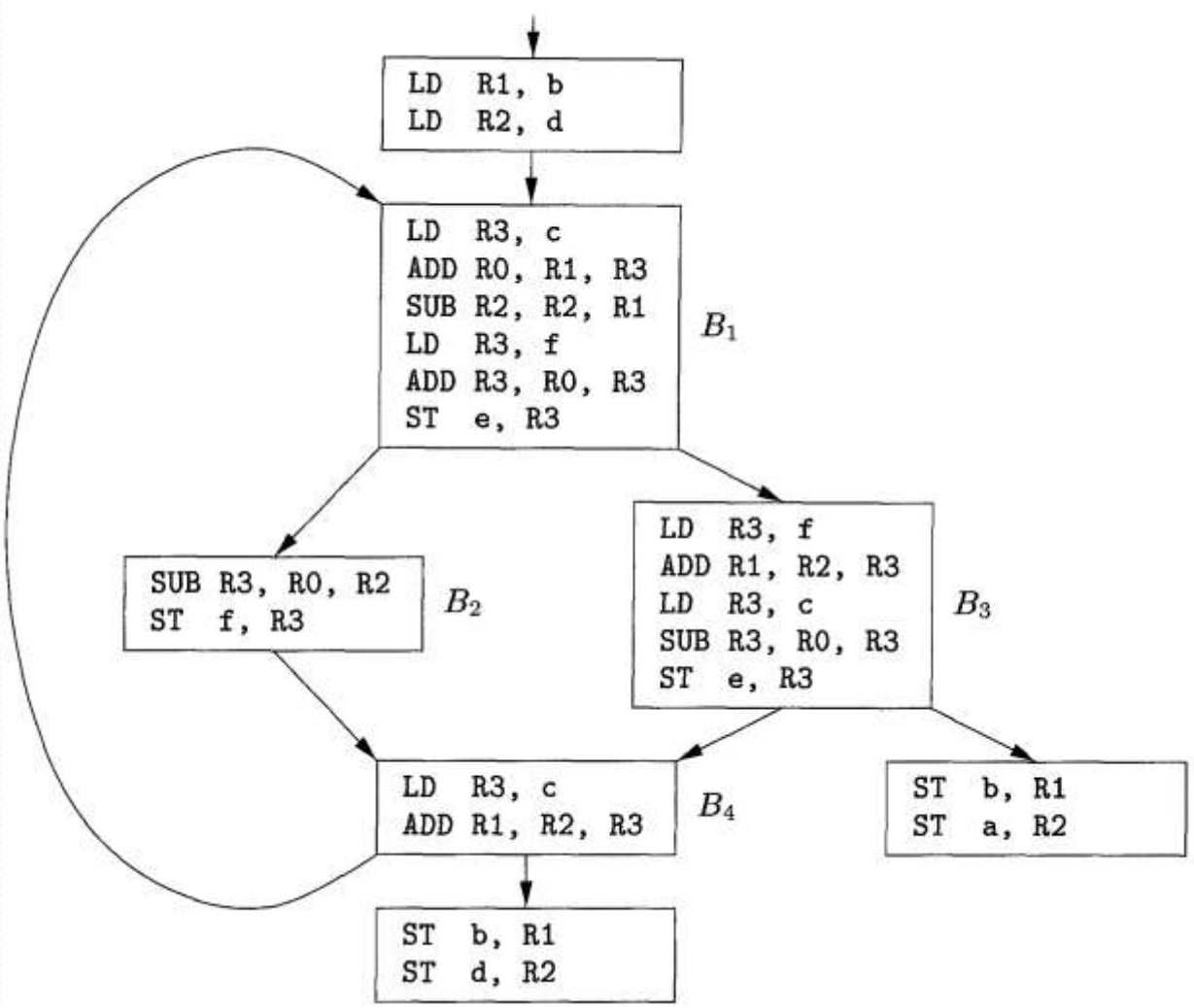
Usage counts

- For the loops we can approximate the saving by register allocation as:
  - Sum over all blocks (B) in a loop (L)
  - For each uses of x before any definition in the block we add one unit of saving
  - If x is live on exit from B and is assigned a value in B, then we ass 2 units of saving

Flow graph of an inner loop



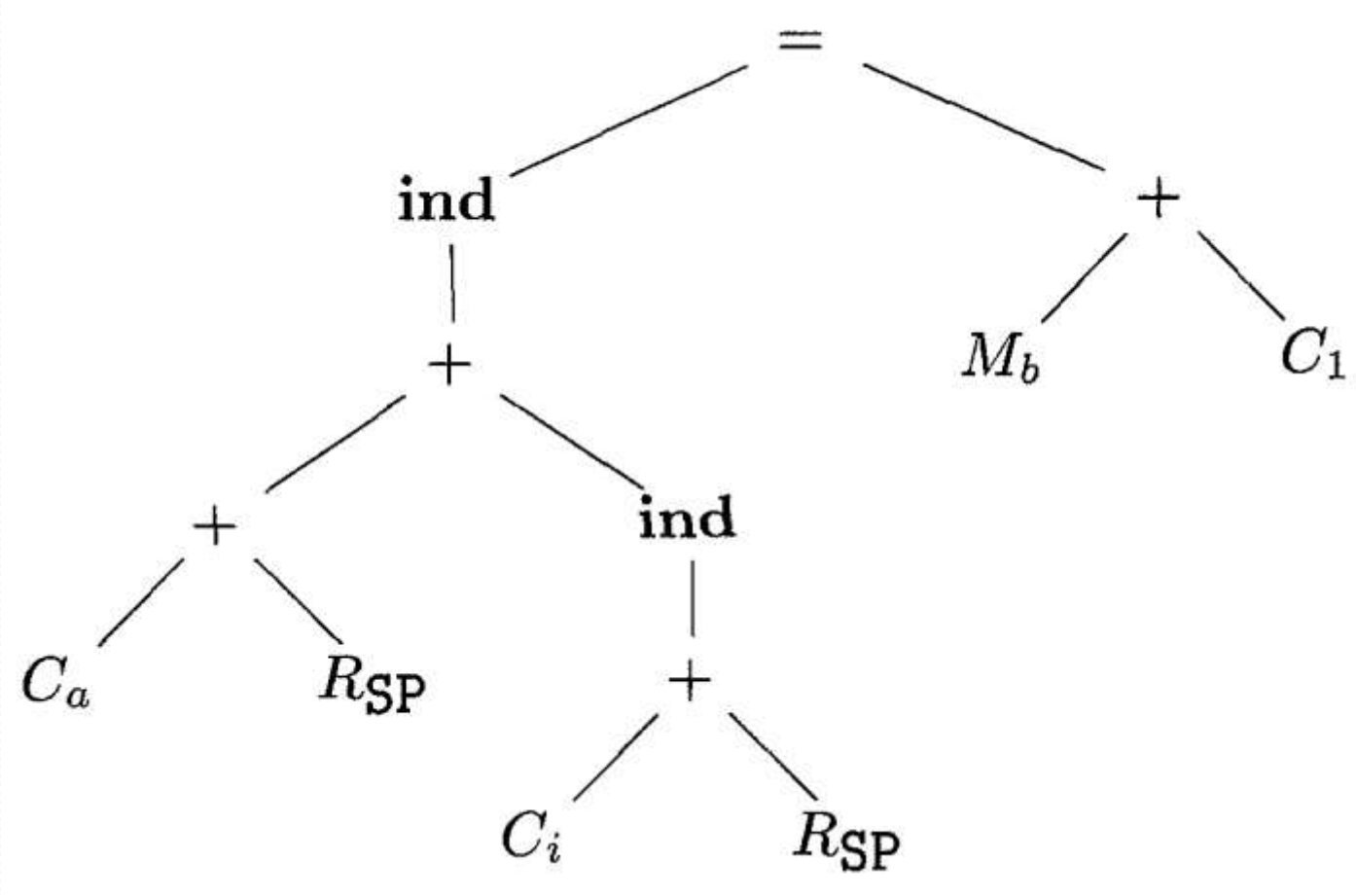
Code sequence using global register assignment



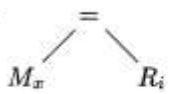
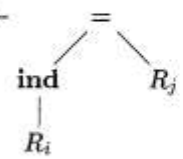
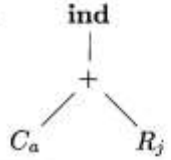
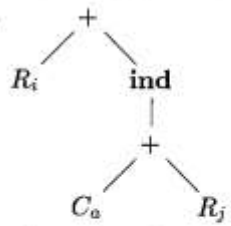
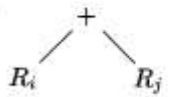
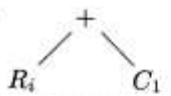
Register allocation by Graph coloring

- Two passes are used
  - Target-machine instructions are selected as though there are an infinite number of symbolic registers
  - Assign physical registers to symbolic ones
    - Create a register-interference graph
    - Nodes are symbolic registers and edges connects two nodes if one is live at a point where the other is defined.
    - For example in the previous example an edge connects a and d in the graph
    - Use a graph coloring algorithm to assign registers.

Intermediate-code tree for a[i]=b+1



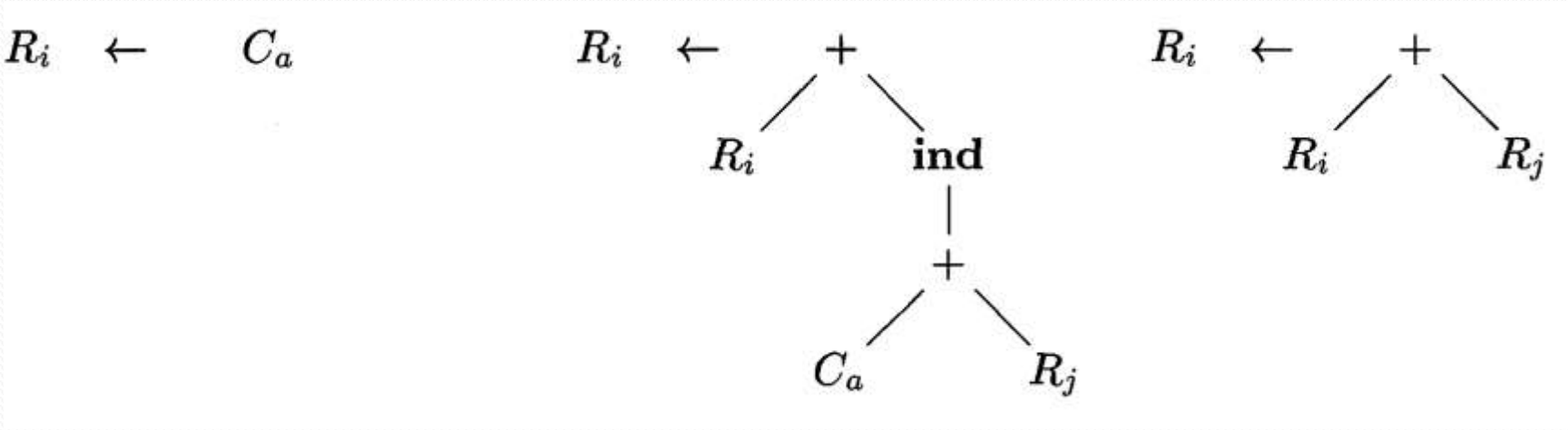
Tree-rewriting rules

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ 	{ ST x, Ri }
4)	$M \leftarrow$ 	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ 	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

Syntax-directed translation scheme

1)	$R_i \rightarrow \mathbf{c}_a$	{ LD Ri, #a }
2)	$R_i \rightarrow M_x$	{ LD Ri, x }
3)	$M \rightarrow = M_x\ R_i$	{ ST x, Ri }
4)	$M \rightarrow = \mathbf{ind}\ R_i\ R_j$	{ ST *Ri, Rj }
5)	$R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$	{ LD Ri, a(Rj) }
6)	$R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \rightarrow + R_i\ R_j$	{ ADD Ri, Ri, Rj }
8)	$R_i \rightarrow + R_i\ \mathbf{c}_1$	{ INC Ri }
9)	$R \rightarrow \mathbf{sp}$	
10)	$M \rightarrow \mathbf{m}$	

An instruction set for tree matching

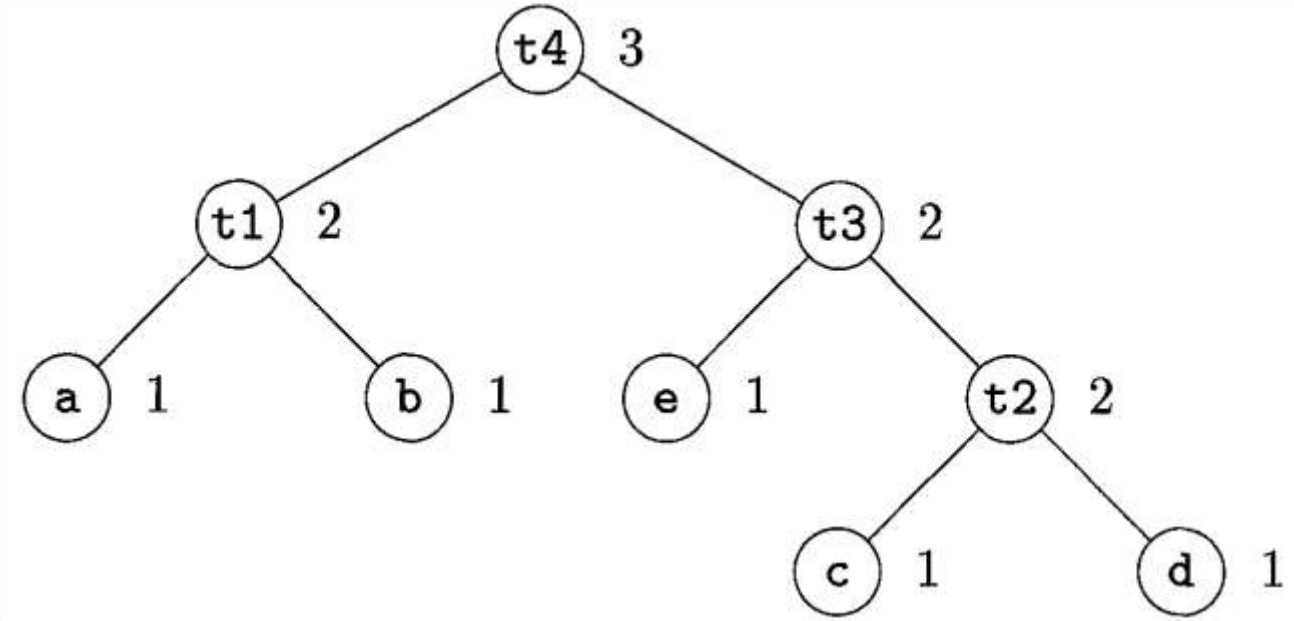


Ershov Numbers

- Label any leaf 1.
- The label of an interior node with one child is the label of its child.
- The label of an interior node with two children is
  - The larger of the labels of its children, if those labels are different.
  - One plus the label of its children if the labels are the same.

A tree labeled with Ershov numbers

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```



Generating code from a labeled expression tree

- To generate machine code for an interior node with label *k* and two children with equal labels (which must be *k* - 1) do the following:
  - Recursively generate code for the right child, using base *b*+1. The result of the right child appears in register R_{b+k}.
  - Recursively generate code for the left child, using base b; the result appears in R_{b+k-1}.
  - Generate the instruction OP R_{b+k}, R_{b+k-1}, R_{b+k}, where OP is the appropriate operation for the interior node in question.
- Suppose we have an interior node with label *k* and children with unequal labels. Then one of the children, which we'll call the "big" child, has label *k* , and the other child, the "little" child, has some label m < *k*. Do the following to generate code for this interior node, using base *b:*
  - Recursively generate code for the big child, using base b; the result appears in register R_{b+k-1}.
  - Recursively generate code for the small child, using base *b;* the result appears in register R_{b+m-1}. Note that since m < k, neither R_{b+k-1} nor any higher-numbered register is used.
  - Generate the instruction OP R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or the instruction OP R_{b+k-1}, R_{b+k-1}, R_{b+m+1}, depending on whether the big child is the right or left child, respectively.
- For a leaf representing operand x, if the base is *b* generate the instruction LD R_b, *x*.

Optimal three-register code

```
LD   R3, d
LD   R2, c
ADD  R3, R2, R3
LD   R2, e
MUL  R3, R2, R3
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
ADD  R3, R2, R3
```

Evaluating Expressions with an Insufficient Supply of Registers

- Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.
- Recursively generate code for the big child, using base b = 1. The result of this evaluation will appear in register R_r
- Generate the machine instruction ST t_k, R_r, where t_k is a temporary variable used for temporary results used to help evaluate nodes with label k.
- Generate code for the little child as follows. If the little child has label r or greater, pick base b=1. If the label of the little child is j<r, then pick b=r-j. Then recursively apply this algorithm to the little child; the result appears in R_r.
- Generate the instruction LD R_r-l, t_k.
- If the big child is the right child of N, then generate the instruction OP R_r, R_r, R_{r-1}. If the big child is the left child, generate OP R_r, R_{r-1}, R_r.

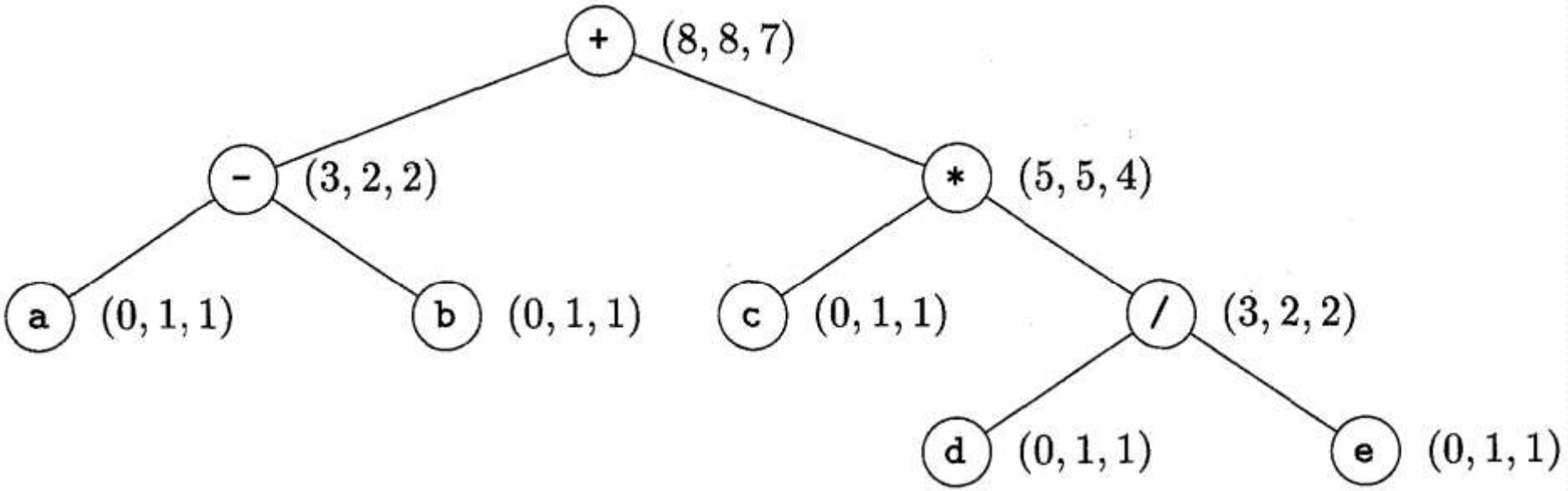
Optimal three-register code using only two registers

```
LD   R2, d
LD   R1, c
ADD  R2, R1, R2
LD   R1, e
MUL  R2, R1, R2
ST   t3, R2
LD   R2, b
LD   R1, a
SUB  R2, R1, R2
LD   R1, t3
ADD  R2, R2, R1
```

Dynamic Programming Algorithm

- Compute bottom-up for each node n of the expression tree T an array *C* of costs, in which the ith component C[i] is the optimal cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for $1 \leq i \leq r$
- Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory.
- Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

Syntax tree for (a-b)+c*(d/e) with cost vector at each node



minimum cost of evaluating the root with two registers available

- Compute the left subtree with two registers available into register Ro, compute the right subtree with one register available into register R1, and use the instruction ADD Ro, Ro, R1 to compute the root. This sequence has cost 2+5+1=8.
- Compute the right subtree with two registers available into R l , compute the left subtree with one register available into Ro, and use the instruction ADD Ro, Ro, R1. This sequence has cost 4+2+1=7.
- Compute the right subtree into memory location M, compute the left subtree with two registers available into register RO, and use the instruction ADD Ro, Ro, M. This sequence has cost 5+2+1=8.

Compiler course

Chapter 5

Syntax Directed Translation

Outline

- Syntax Directed Definitions
- Evaluation Orders of SDD's
- Applications of Syntax Directed Translation
- Syntax Directed Translation Schemes

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Production	Semantic Rule
E->E1+T	E.code=E1.code||T.code||'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions

- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- Inherited attributes
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Example of S-attributed SDD

Production	Semantic Rules
1) L -> E n	L.val = E.val
2) E -> E1 + T	E.val = E1.val + T.val
3) E -> T	E.val = T.val
4) T -> T1 * F	T.val = T1.val * F.val
5) T -> F	T.val = F.val
6) F -> (E)	F.val = E.val
7) F -> digit	F.val = digit.lexval

Example of mixed attributes

Production	Semantic Rules
1) T -> FT’	T’.inh = F.val T.val = T’.syn
2) T’ -> *FT’$_1$	T’$_1$.inh = T’.inh*F.val T’.syn = T’$_1$.syn
3) T’ -> ε	T’.syn = T’.inh
1) F -> digit	F.val = F.val = digit.lexval

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.c to B.c
- Example!

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j
- Such an ordering is called a topological sortof a graph
- Example!

S-Attributed definitions

- An SDD is S-attributed if every attribute is synthesized
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
        for (each child C of N, from the left) postorder(C);
        evaluate the attributes associated with node N;
}
```

- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there us a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Application of Syntax Directed Translation

- Type checking and intermediate code generation (chapter 6)
- Construction of syntax trees
  - Leaf nodes: Leaf(op,val)
  - Interior node: Node(op,c1,c2,…,ck)
- Example:

Production	Semantic Rules
1) E -> E1 + T	E.node=new node(‘+’, E1.node,T.node)
2) E -> E1 - T	E.node=new node(‘-’, E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id,id.entry)
6) T -> num	T.node = new Leaf(num,num.val)

Syntax tree for L-attributed definition

Production	Semantic Rules
1) E -> TE’	E.node=E’.syn E’.inh=T.node
2) E’ -> + TE1’	E1’.inh=new node(‘+’, E’.inh,T.node) E’.syn=E1’.syn
3) E’ -> -TE1’	E1’.inh=new node(‘+’, E’.inh,T.node) E’.syn=E1’.syn
4) E’ -> ∈	E’.syn = E’.inh
5) T -> (E)	T.node = E.node
6) T -> id	T.node=new Leaf(id,id.entry)
7) T -> num	T.node = new Leaf(num,num.val)

Syntax directed translation schemes

- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically SDT's are implemented during parsing without building a parse tree

Postfix translation schemes

- Simplest SDDs are those that we can parse the grammar bottom-up and the SDD is s-attributed
- For such cases we can construct SDT where each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production
- SDT's with all actions at the right ends of the production bodies are called postfix SDT's

Example of postfix SDT

1) L -> E n {print(E.val);}
2) E -> E1 + T {E.val=E1.val+T.val;}
3) E -> T {E.val = T.val;}
4) T -> T1 * F {T.val=T1.val*F.val;}
5) T -> F {T.val=F.val;}
6) F -> (E) {F.val=E.val;}
7) F -> digit {F.val=digit.lexval;}

Parse-Stack implementation of postfix SDT's

- In a shift-reduce parser we can easily implement semantic action using the parser stack
- For each nonterminal (or state) on the stack we can associate a record holding its attributes
- Then in a reduction step we can execute the semantic action at the end of a production to evaluate the attribute(s) of the non-terminal at the leftside of the production
- And put the value on the stack in replace of the rightside of production

Example

```
L -> E n          {print(stack[top-1].val);
                    top=top-1;}
E -> E1 + T       {stack[top-2].val=stack[top-2].val+stack.val;
                    top=top-2;}
E -> T
T -> T1 * F       {stack[top-2].val=stack[top-2].val+stack.val;
                    top=top-2;}
T -> F
F -> (E)          {stack[top-2].val=stack[top-1].val
            top=top-2;}
F -> digit
```

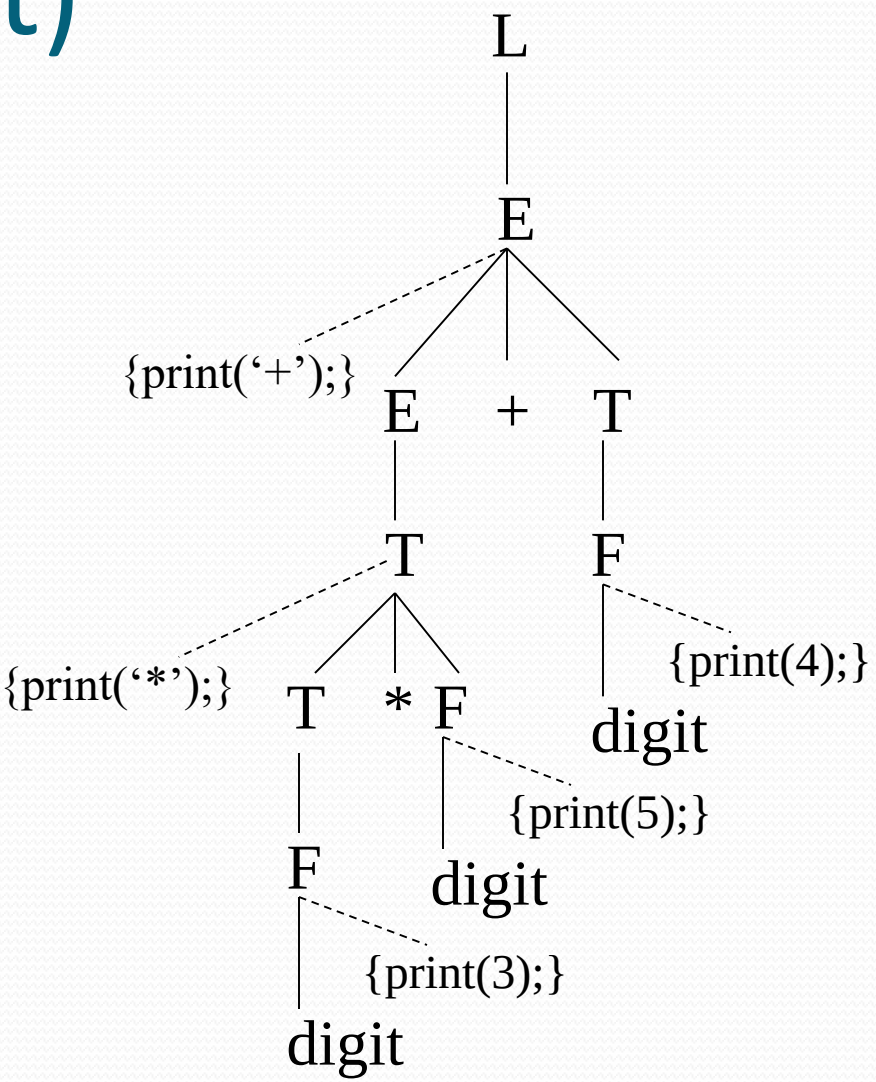
SDT's with actions inside productions

- For a production B->X {a} Y
  - If the parse is bottom-up then we perform action "a" as soon as this occurrence of X appears on the top of the parser stack
  - If the parser is top down we perform "a" just before we expand Y
- Sometimes we cant do things as easily as explained above
- One example is when we are parsing this SDT with a bottom-

1) L -> E n
2) E -> {print('+');} E1 + T
3) E -> T
4) T -> {print('*');} T1 * F
5) T -> F
6) F -> (E)
7) F -> digit {print(digit.lexval);}

SDT's with actions inside productions (cont)

- Any SDT can be implemented as follows
  1. Ignore the actions and produce a parse tree
  2. Examine each interior node N and add actions as new children at the correct position
  3. Perform a postorder traversal and execute actions when their nodes are visited



SDT's for L-Attributed definitions

- We can convert an L-attributed SDD into an SDT using following two rules:
  - Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A. if several inherited attributes of A are dpendent on one another in an acyclic fashion, order them so that those needed first are computed first
  - Place the action of a synthesized attribute for the head of a production at the end of the body of the production

Example

S -> while (C) S1

L1=new();
L2=new();
S1.next=L1;
C.false=S.next;
C.true=L2;
S.code=label||L1||C.code||label||L2||S1.code

S -> while ({L1=new();L2=new();C.false=S.next;C.true=L2;}
C) {S1.next=L1;}
S1{S.code=label||L1||C.code||label||L2||S1.code;}

Readings

- Chapter 5 of the book

Compiler course

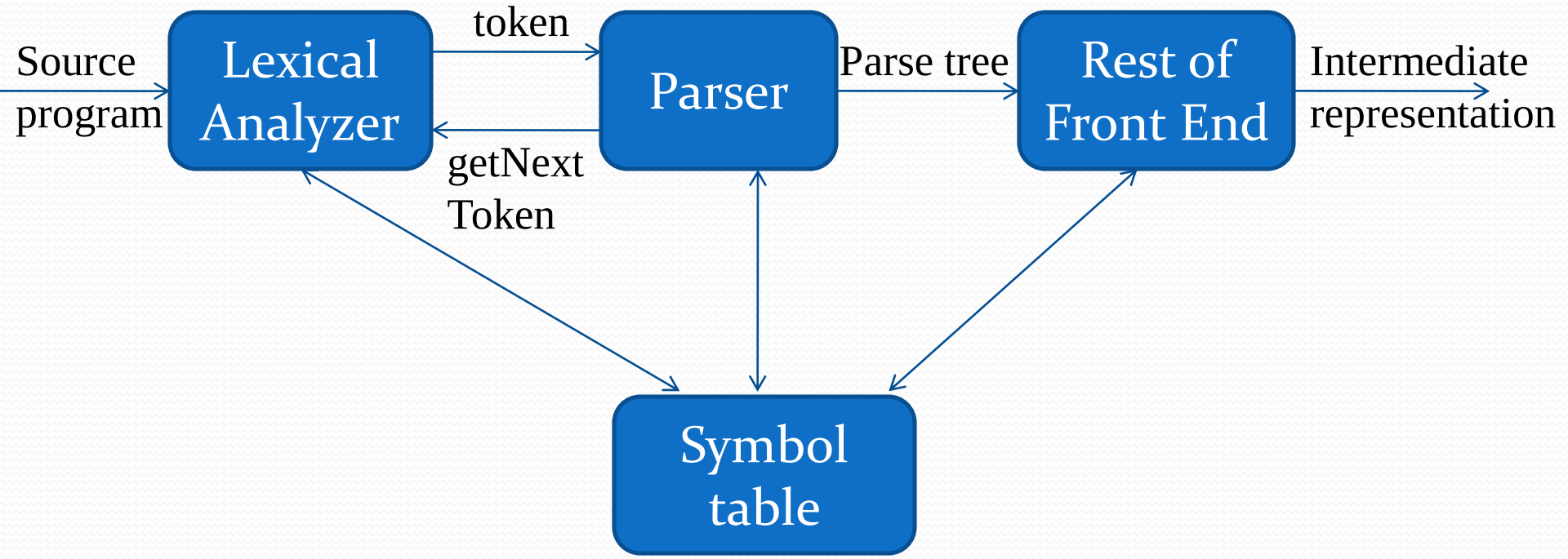
Chapter 4

Syntax Analysis

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

The role of parser



Uses of grammars

E -> E + T | T
T -> T * F | F
F -> (E) | **id**

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct progrms

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

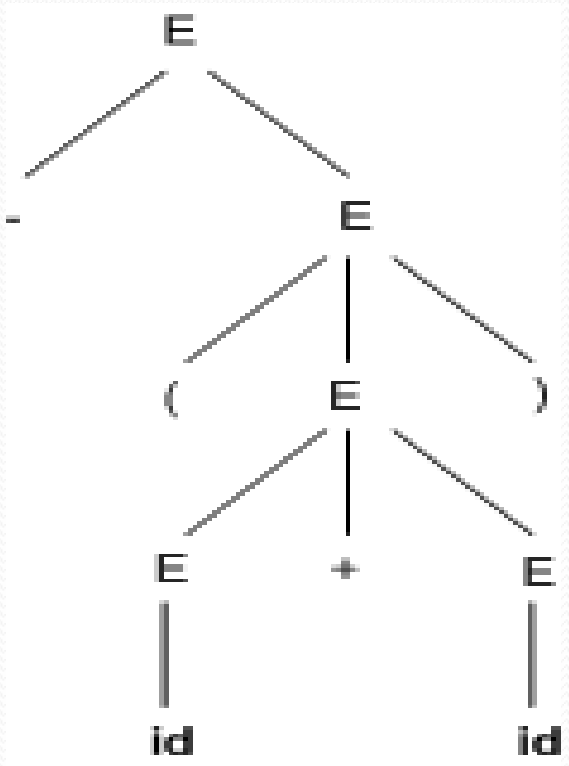
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

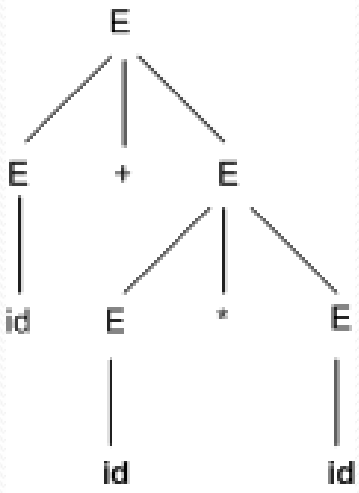
Parse trees

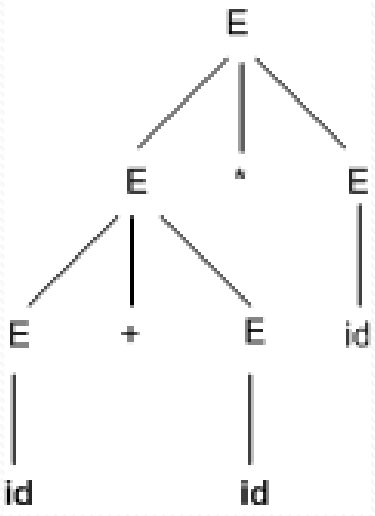
- **-(id+id)**
- E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)



Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id



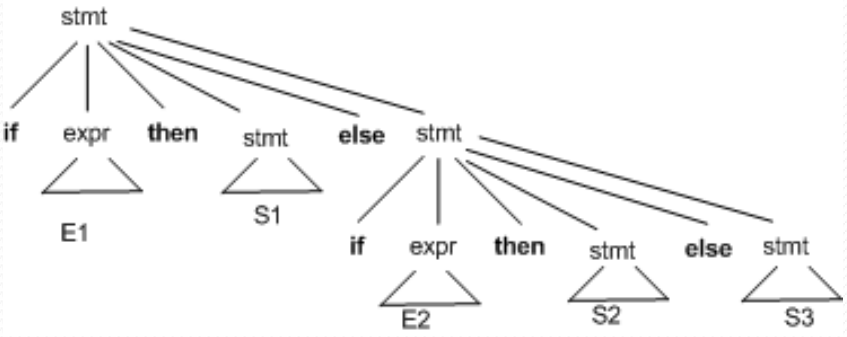


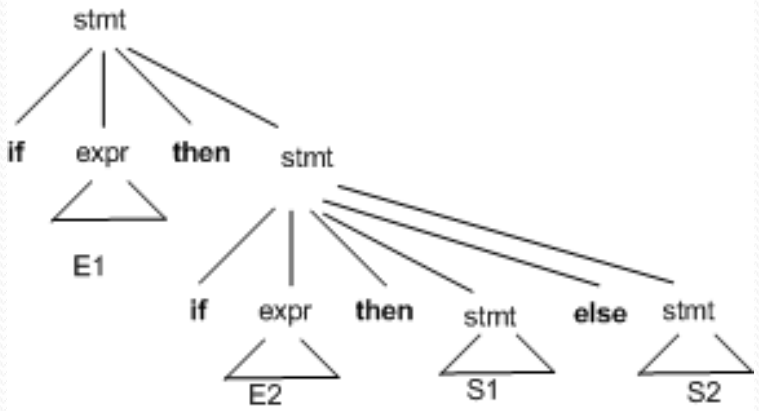
Elimination of ambiguity

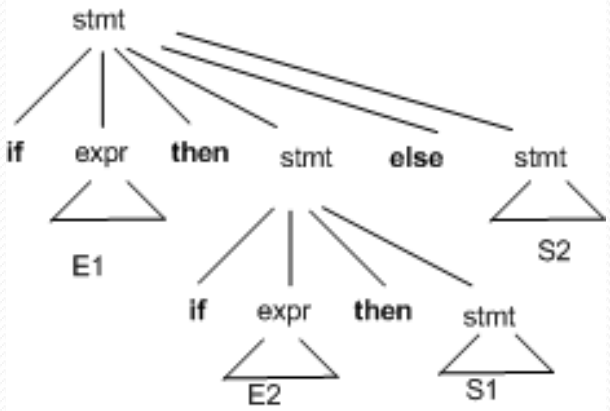
stmt ⟶ **If** expr **then** stmt

| **If** expr **then** stmt **else** stmt

| **other**

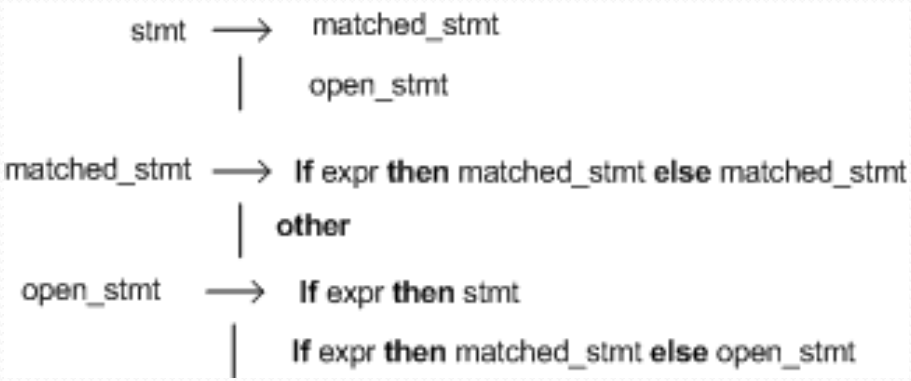






Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched



Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \rightarrow A\ \alpha \mid \beta$
  - We may replace it with
    - $A \rightarrow \beta\ A'$
    - $A' \rightarrow \alpha\ A' \mid \varepsilon$

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order A1,A2,…,An.
  - For (each i from 1 to n) {
    - For (each j from 1 to i-1) {
      - Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta_1 \gamma$ | $\delta_2 \gamma$ | … | $\delta_k \gamma$ where Aj-> δ_1 | δ_2 | … | δ_k are all current Aj productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A-> $\alpha\ \beta_1$ | $\alpha\ \beta_2$ then we replace it with
    - A -> α A'
    - A' -> β_1 | β_2

Left factoring (cont.)

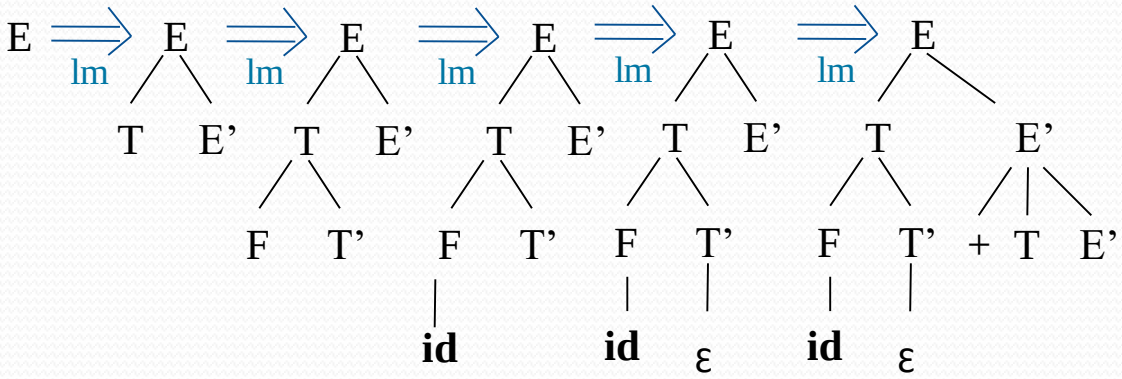
- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha <> \varepsilon$, then replace all of A-productions A-> $\alpha\ \beta 1 \mid \alpha\ \beta 2 \mid \ldots \mid \alpha\ \beta n \mid \gamma$ by
    - A -> α A' | γ
    - A' -> $\beta 1 \mid \beta 2 \mid \ldots \mid \beta n$
- Example:
  - S -> I E t S | i E t S e S | a
  - E -> b

Top Down Parsing

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
                if (Xi is a nonterminal
                        call procedure Xi();
                else if (Xi equals the current input symbol a)
                        advance the input to the next symbol;
                else /* an error has occurred */
        }
}
```

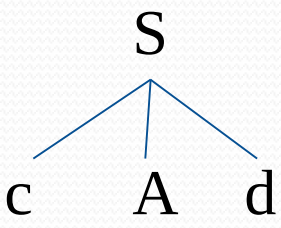
Recursive descent parsing (cont)

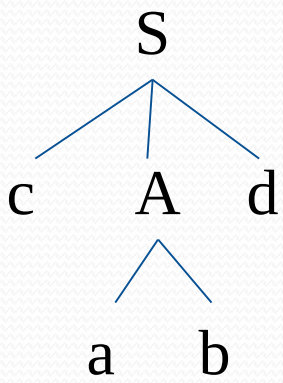
- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

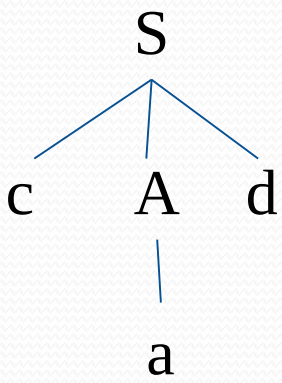
Example

S->cAd
A->ab | a

Input: cad







First and Follow

- First() is set of terminals that begins strings derived from
- If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then is also in First(ε)
- In predictive parsing when we have A-> $\alpha \mid \beta$, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
  1. If X is a terminal then First(X) = {X}.
  2. If X is a nonterminal and X->Y1Y2…Yk is a production for some k>=1, then place a in First(X) if for some i a is in First(Yi) and ε is in all of First(Y1),…,First(Yi-1) that is $Y1…Yi\text{-}1 \overset{*}{\Rightarrow} \varepsilon$. if ε is in First(Yj) for j=1,…,k then add ε to First(X).
  3. If X-> ε is a production then add ε to First(X)
- Example!

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
  1. Place $ in Follow(S) where S is the start symbol
  2. If there is a production A-> α B β then everything in First(β) except ε is in Follow(B).
  3. If there is a production A->B or a production A-> α B β where First(β) contains ε, then everything in Follow(A) is in Follow(B)
- Example!

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
- A grammar G is LL(1) if and only if whenever A-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \Rightarrow^{*} \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

- For each production A-> α in grammar do the following:
  1. For each terminal a in First(α) add A-> in M[A,a]
  2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well
- If after performing the above, there is no production in M[A,a] then set M[A,a] to error

Example

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}

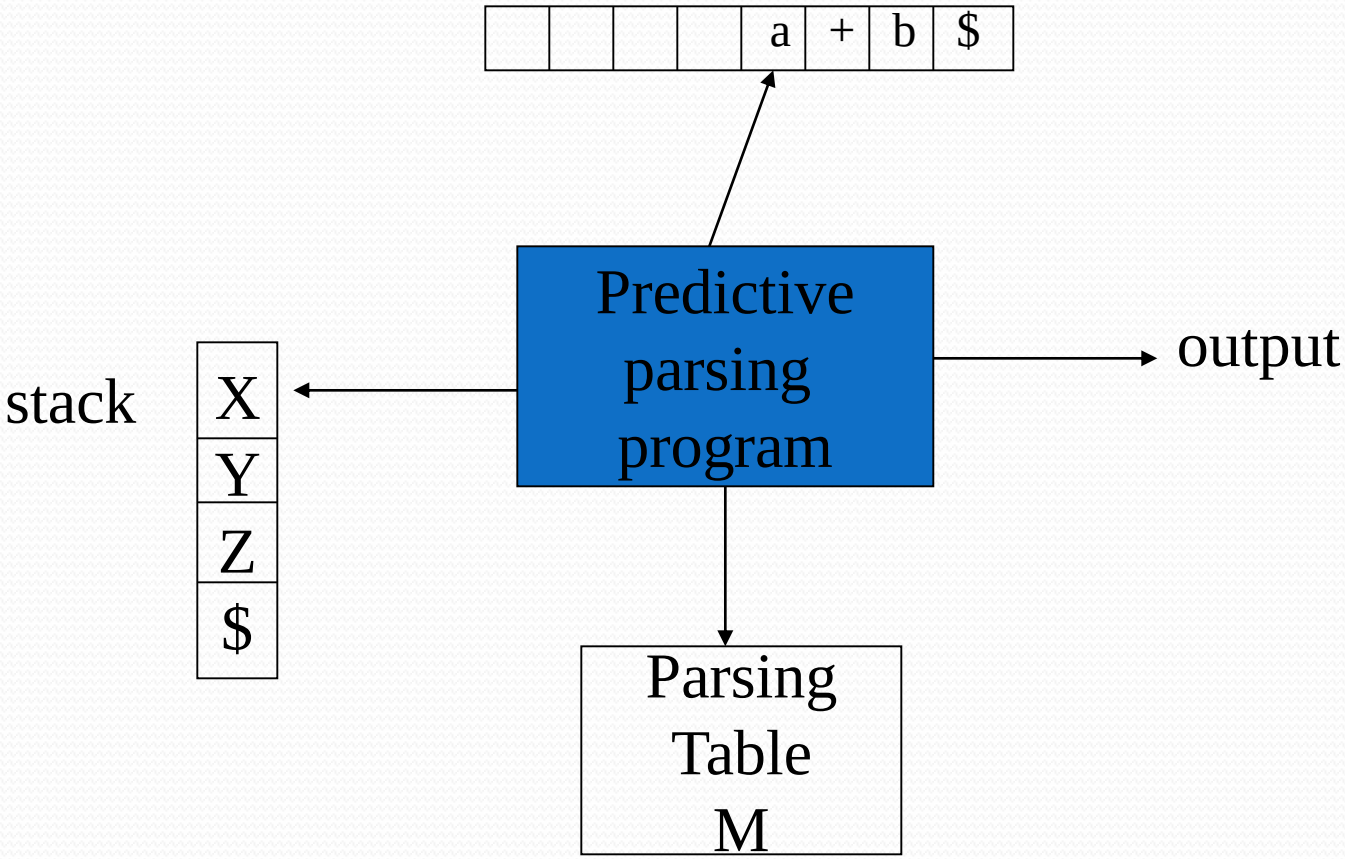
Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | ε
E -> b

Non-terminal	Input Symbol					
	a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> ε S' -> eS			S' -> ε
E		E -> b				

Non-recursive predicting parsing



Predictive parsing algorithm

```
Set ip point to the first symbol of w;
Set X to the top stack symbol;
While (X<>$) { /* stack is not empty */
    if (X is a) pop the stack and advance ip;
    else if (X is a terminal) error();
    else if (M[X,a] is an error entry) error();
    else if (M[X,a] = X->Y1Y2..Yk) {
            output the production X->Y1Y2..Yk;
            pop the stack;
            push Yk,…,Y2,Y1 on to the stack with Y1 on top;
    }
    set X to the top stack symbol;
}
```

Example

- id+id*id$

Matched	Stack	Input	Action
	E$	id+id*id$	

Error recovery in predictive parsing

- Panic mode
  - Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.
  - Add to the synchronization set of lower level construct the symbols that begin higher level constructs
  - Add symbols in First(A) to the synchronization set of nonterminal A
  - If a nonterminal can generate the empty string then the production deriving can be used as a default
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Example

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’	synch	synch
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’	synch		T -> FT’	synch	synch
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> **id**	synch	synch	F -> (E)	synch	synch

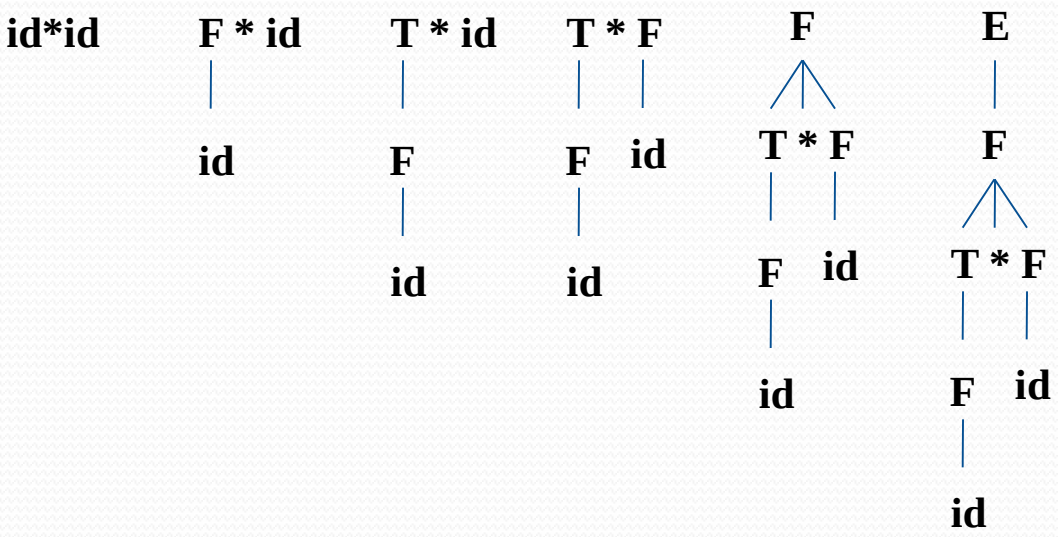
Stack	Input	Action
E$	)id*+id$	Error, Skip)
E$	id*+id$	id is in First(E)
TE’$	id*+id$	
FT’E’$	id*+id$	
idT’E’$	id*+id$	
T’E’$	*+id$	
*FT’E’$	*+id$	
FT’E’$	+id$	Error, M[F,+]=synch
T’E’$	+id$	F has been poped

Bottom-up Parsing

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

E -> E + T | T
T -> T * F | F
F -> (E) | **id**



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

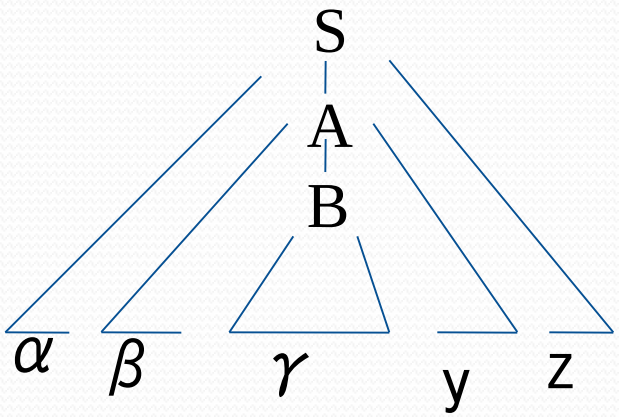
Stack	Input
$S	$

Shift reduce parsing (cont.)

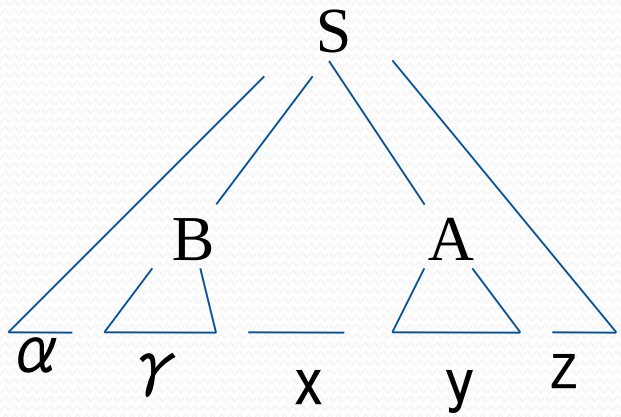
- Basic operations:
  - Shift
  - Reduce
  - Accept
  - Error
- Example: id*id

Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

Handle will appear on top of the stack



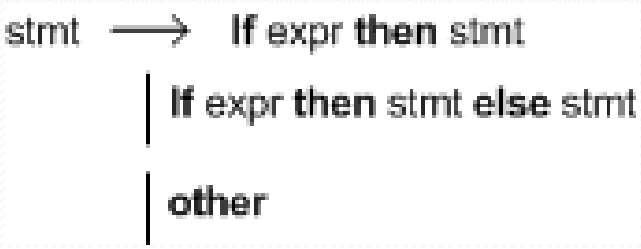
Stack	Input
$ α β γ	yz$
$ α β B	yz$
$ α β By	z$



Stack	Input
$ α γ	xyz$
$ α Bxy	z$

Conflicts during shit reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:



Stack	Input
… if expr then stmt	else …$

Reduce/reduce conflict

stmt -> id(parameter_list)
stmt -> expr:=expr
parameter_list->parameter_list, parameter
parameter_list->parameter
parameter->id
expr->id(expr_list)
expr->id
expr_list->expr_list, expr
expr_list->expr

Stack	Input
… id(id	,id) …$

LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with k<=1
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

Constructing canonical LR(0) item sets

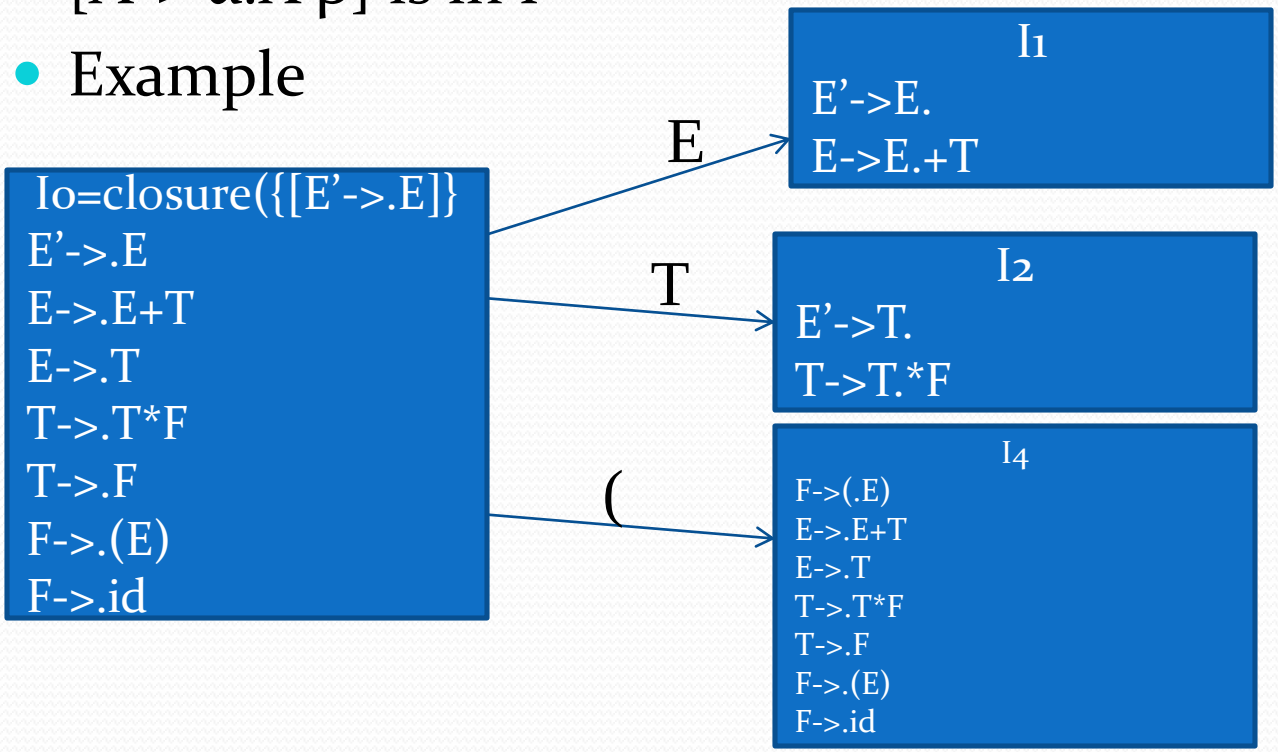
- Augmented grammar:
  - G with addition of a production: S'->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).
- Example:

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

Io=closure({[E'->.E]}
E'->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.X β] is in I
- Example



Closure algorithm

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item A-> α.Bβ in J)
            for (each prodcution B->γ of G)
                if (B->.γ is not in J)
                    add B->.γ to J;
    until no more items are added to J on one round;
    return J;
```

GOTO algorithm

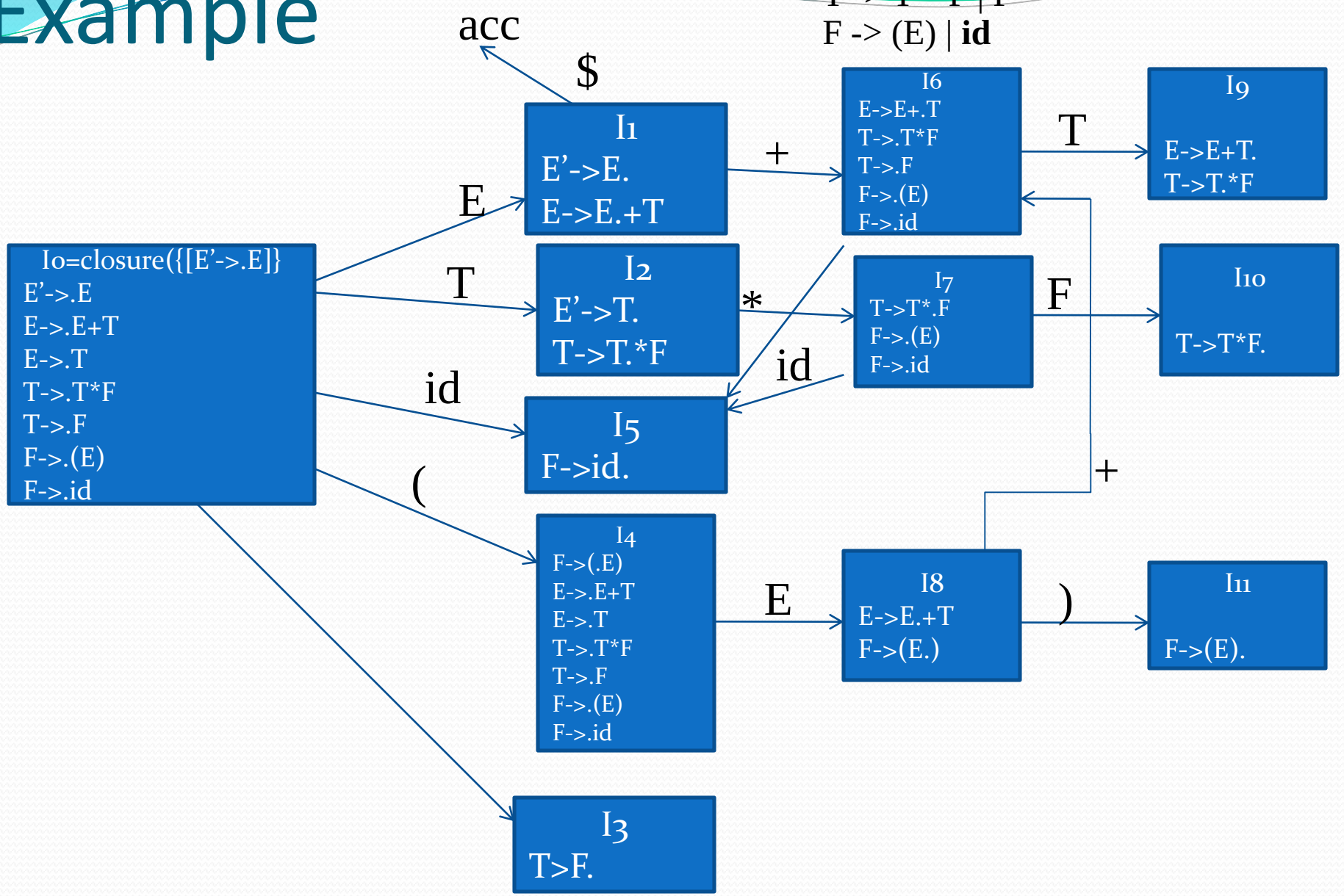
```
SetOfItems  GOTO(I,X) {
   J=empty;
   if (A-> α.X β is in I)
         add CLOSURE(A-> αX. β ) to J;
   return J;
}
```

Canonical LR(0) items

```
Void items(G') {
   C= CLOSURE({[S'->.S]});
   repeat
         for (each set of items I in C)
            for (each grammar symbol X)
               if (GOTO(I,X) is not empty and not in C)
                  add GOTO(I,X) to C;
   until no new set of items are added to C on a round;
}
```

Example

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

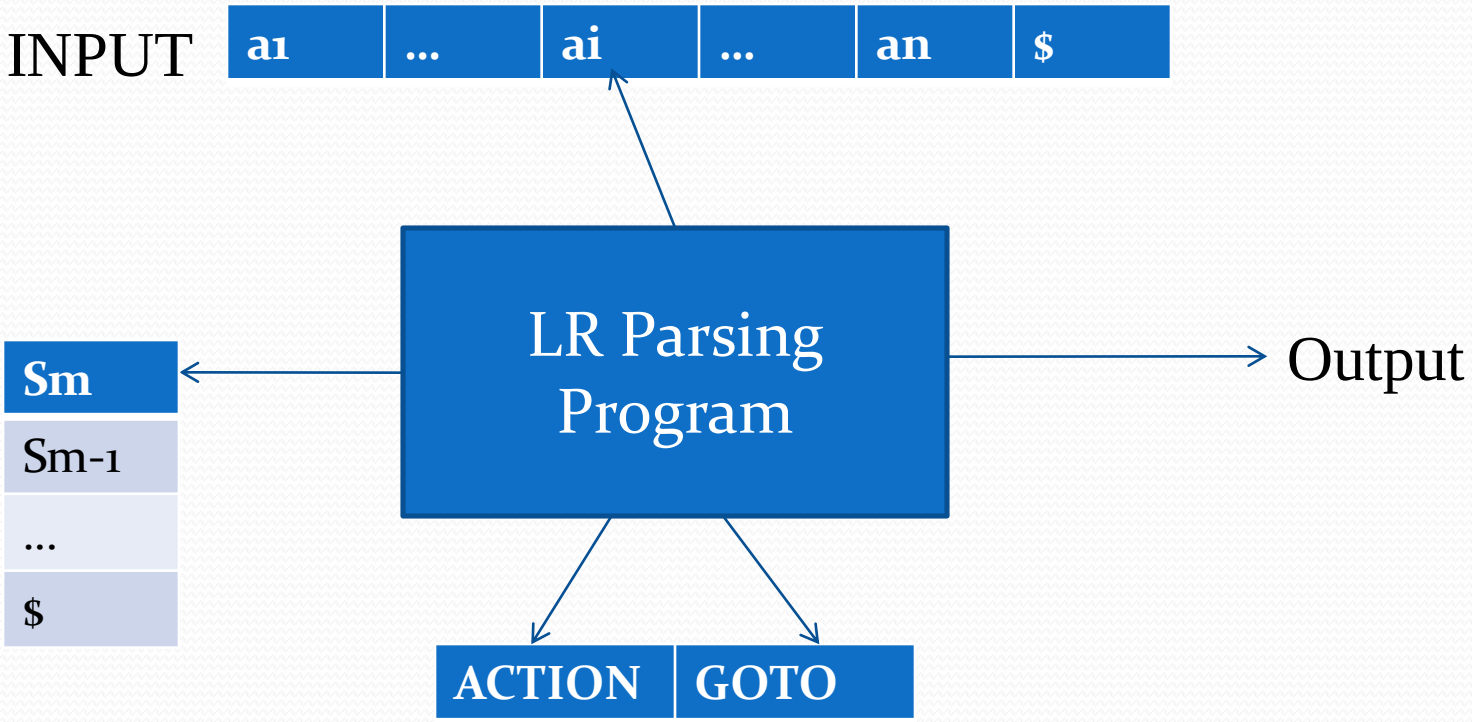


Use of LR(0) automaton

- Example: id*id

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F->id
(3)	03	$F	*id$	Reduce by T->F
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F->id
(7)	02710	$T*F	$	Reduce by T->T*F
(8)	02	$T	$	Reduce by E->T
(9)	01	$E	$	accept

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;
while(1) { /*repeat forever */
    let s be the state on top of the stack;
    if (ACTION[s,a] = shift t) {
            push t onto the stack;
            let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A->β) {
            pop |β| symbols of the stack;
            let state t now be on top of the stack;
            push GOTO[t,A] onto the stack;
            output the production A->β;
    } else if (ACTION[s,a]=accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

Example

STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R4	R7		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			

(0) E'->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F->**id**

id*id+id?

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id
(12)	0163	E+F	$	Reduce by T->F
(13)	0169	E+T`	$	Reduce by E->E+T
(14)	01	E	$	accept

Constructing SLR parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(0) items for G'
  - State i is constructed from state Ii:
    - If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to "shift j"
    - If [A->α.] is in Ii, then set ACTION[i,a] to "reduce A->α" for all a in follow(A)
    - If {S'->.S] is in Ii, then set ACTION[I,$] to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing [S'->.S]

Example grammar which is not SLR(1)

S -> L=R | R
L -> *R | id
R -> L

I0
S'->.S
S -> .L=R
S->.R
L -> .*R |
L->.id
R ->. L

I1
S'->S.

I2
S ->L.=R
R ->L.

I3
S ->R.

I4
L->*.R
R->.L
L->.*R
L->.id

I5
L -> id.

I6
S->L=.R
R->.L
L->.*R
L->.id

I7
L -> *R.

I8
R -> L.

I9
S -> L=R.

	Action =
2	Shift 6 Reduce R->L

More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items

Canonical LR(1) items

- In LR(1) items each item is in the form: [A->α.β,a]
- An LR(1) item [A->α.β,a] is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow} \delta Aw \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where
  - $\Gamma = \delta\alpha$
  - Either a is the first symbol of w, or w is ε and a is $
- Example:
  - S->BB
  - B->aB|b

$S \overset{*}{\Rightarrow} aaBab \underset{rm}{\Rightarrow} aaaBab$

Item [B->a.B,a] is valid for γ=aaa and w=ab

Constructing LR(1) sets of items

```
SetOfItems Closure(I) {
    repeat
        for (each item [A->α.Bβ,a] in I)
            for (each production B->γ in G')
                for (each terminal b in First(βa))
                    add [B->.γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems Goto(I,X) {
    initialize J to be the empty set;
    for (each item [A->α.Xβ,a] in I)
        add item [A->αX.β,a] to set J;
    return closure(J);
}

void items(G'){
    initialize C to Closure({[S'->.S,$]});
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (Goto(I,X) is not empty and not in C)
                    add Goto(I,X) to C;
    until no new sets of items are added to C;
}
```

Example

S’->S

S->CC

C->cC

C->d

Canonical LR(1) parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(1) items for G'
  - State i is constructed from state Ii:
    - If [A->α.aβ, b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to "shift j"
    - If [A->α., a] is in Ii, then set ACTION[i,a] to "reduce A->α"
    - If {S'->.S,$] is in Ii, then set ACTION[I,$] to "Accept"
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing [S'->.S,$]

Example

S’->S

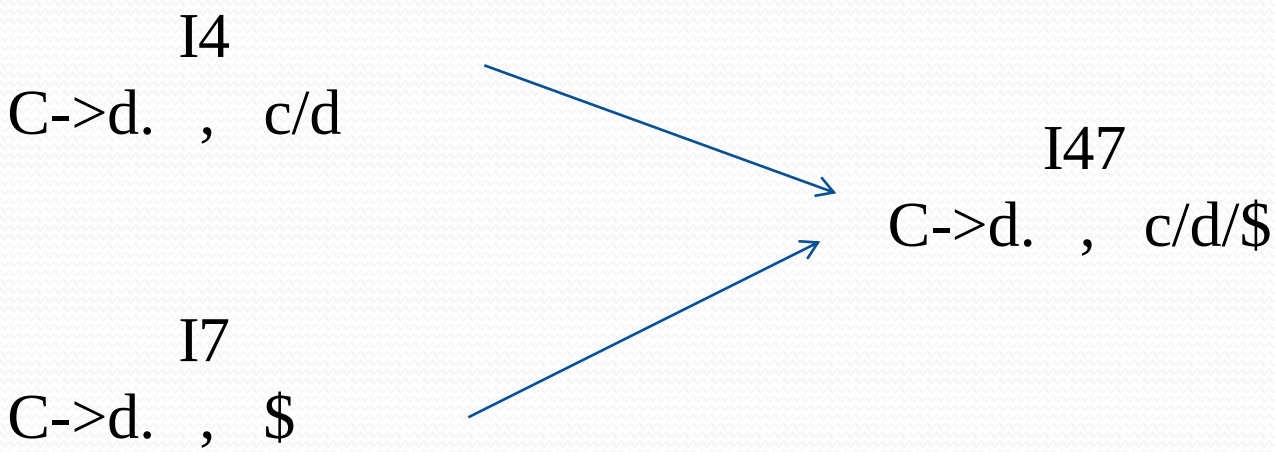
S->CC

C->cC

C->d

LALR Parsing Table

- For the previous example we had:



- State merges cant produce Shift-Reduce conflicts. Why?
- But it may produce reduce-reduce conflict

Example of RR conflict in state merging

S'->S

S -> aAd | bBd | aBe | bAe

A -> c

B -> c

An easy but space-consuming LALR table construction

- Method:
  1. Construct C={I0,I1,…,In} the collection of LR(1) items.
  2. For each core among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  3. Let C'={J0,J1,…,Jm} be the resulting sets. The parsing actions for state i, is constructed from Ji as before. If there is a conflict grammar is not LALR(1).
  4. If J is the union of one or more sets of LR(1) items, that is J = I1 UI2…IIk then the cores of Goto(I1,X), …, Goto(Ik,X) are the same and is a state like K, then we set Goto(J,X) =k.
- This method is not efficient, a more efficient one is discussed in the book

Compaction of LR parsing table

- Many rows of action tables are identical
  - Store those rows separately and have pointers to them from different states
  - Make lists of (terminal-symbol, action) for each state
  - Implement Goto table by having a link list for each nonterinal in the form (current state, next state)

Using ambiguous grammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						GO TO
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Acc	
2	S3		S2				6
3		R4	R4		R4	R4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5				
7		R1	S5		R1	R1	
8		R2	R2		R2	R2	
9		R3	R3		R3	R3	

I0: E'->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E'->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->.id

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

Readings

- Chapter 4 of the book

Compiler course

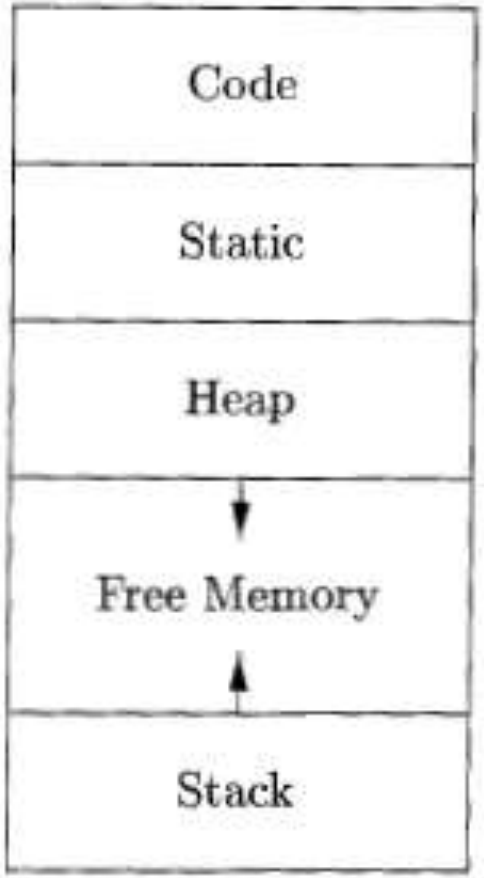
Chapter 7

Run-Time Environments

Outline

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
  - Stack allocation
  - Heap management
  - Garbage collection

Storage Organization



Static vs. Dynamic Allocation

- Static: Compile time, Dynamic: Runtime allocation
- Many compilers use some combination of following
  - Stack storage: for local variables, parameters and so on
  - Heap storage: Data that may outlive the call to the procedure that created it
- Stack allocation is a valid allocation for procedures since procedure calls are nested

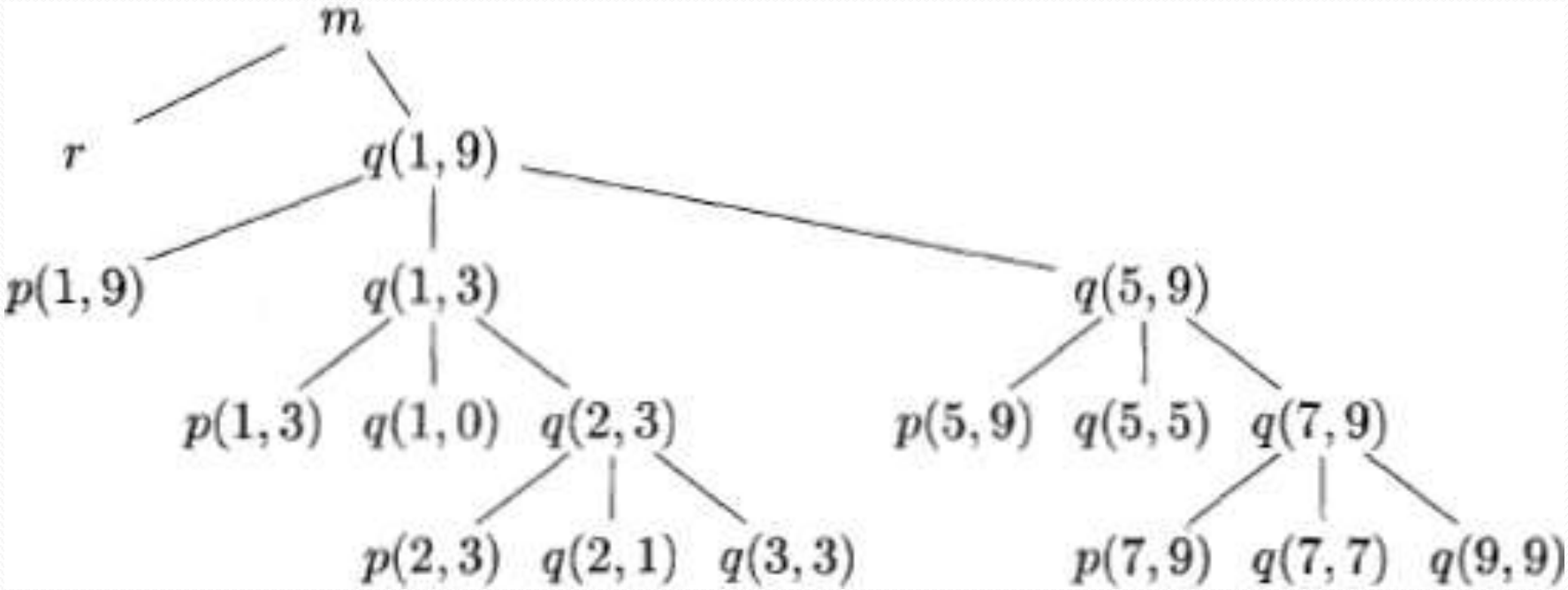
Sketch of a quicksort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Activation for Quicksort

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

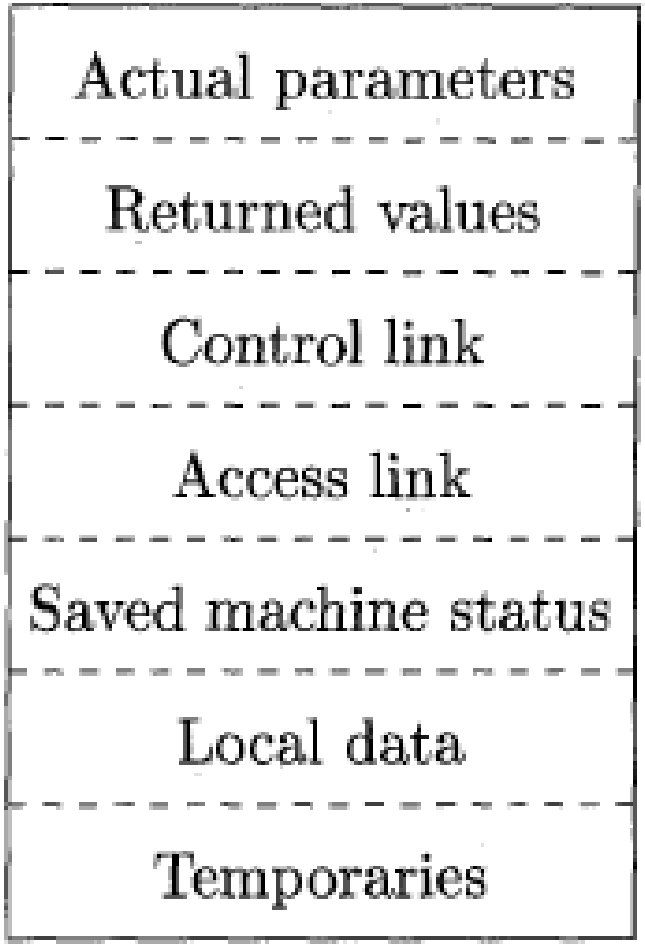
Activation tree representing calls during an execution of quicksort



Activation records

- Procedure calls and returns are usaully managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last activation has record in the top of the stack.

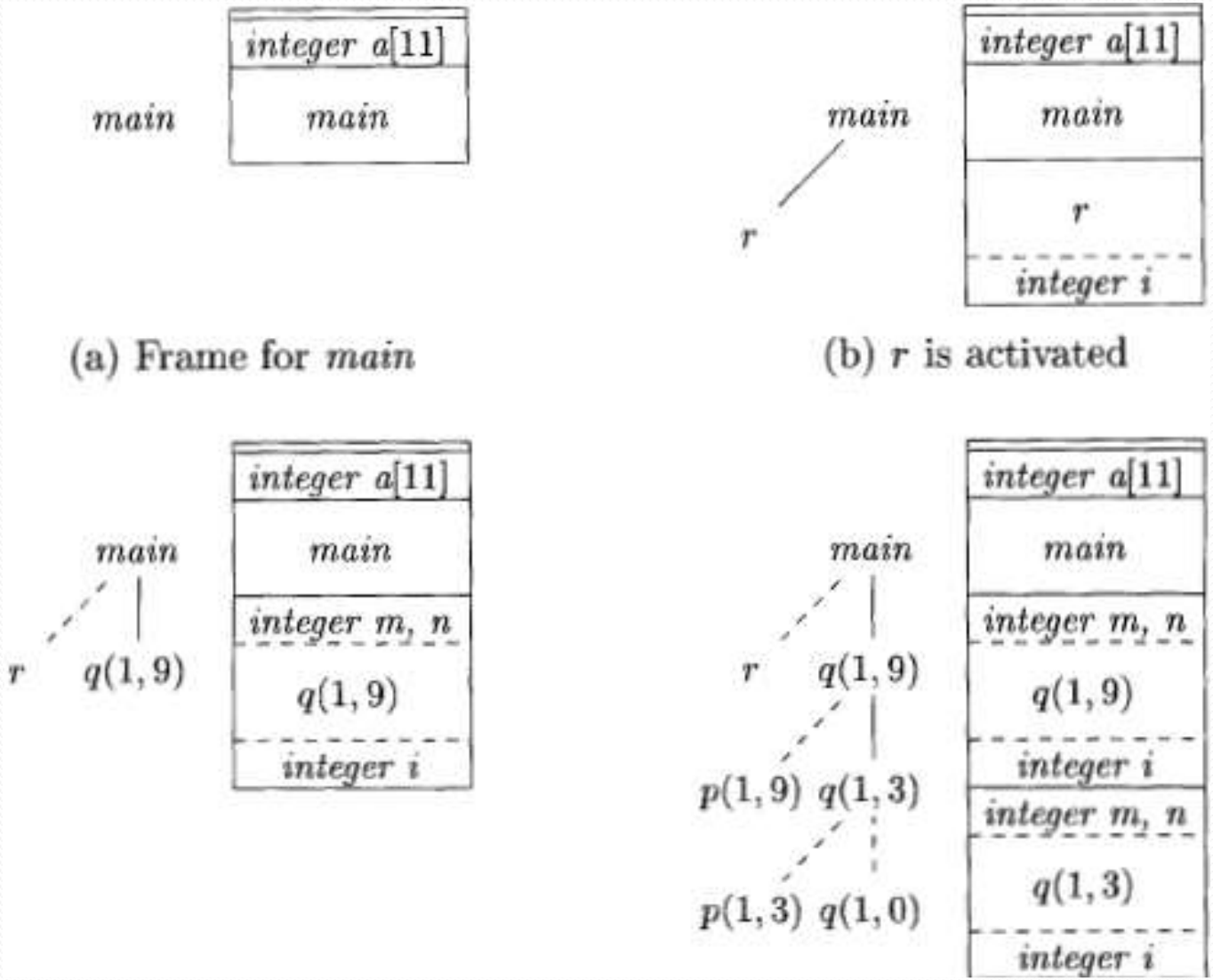
A General Activation Record



Activation Record

- Temporary values
- Local data
- A saved machine status
- An “access link”
- A control link
- Space for the return value of the called function
- The actual parameters used by the calling procedure

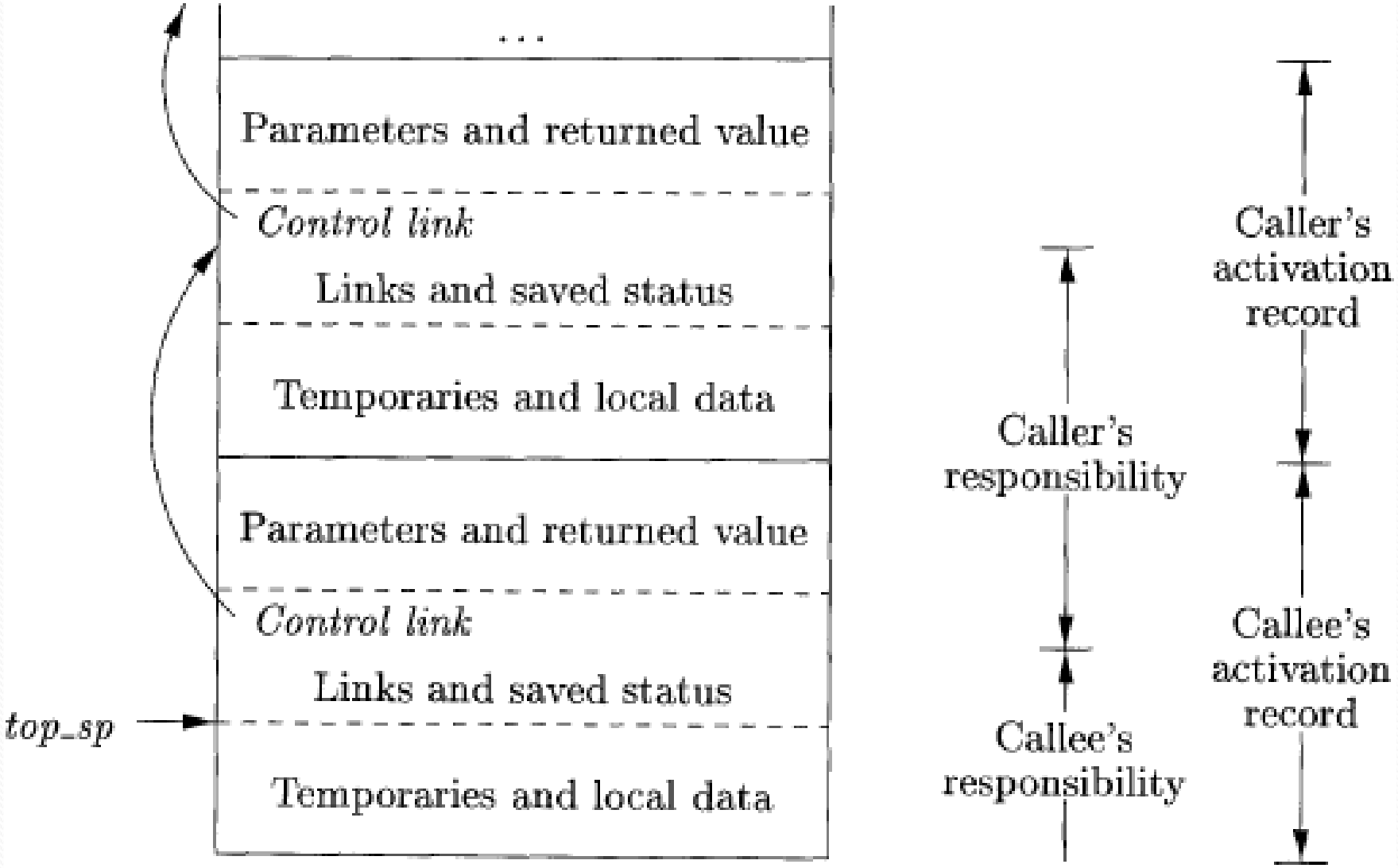
Downward-growing stack of activation records



Designing Calling Sequences

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record
- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields.

Division of tasks between caller and callee



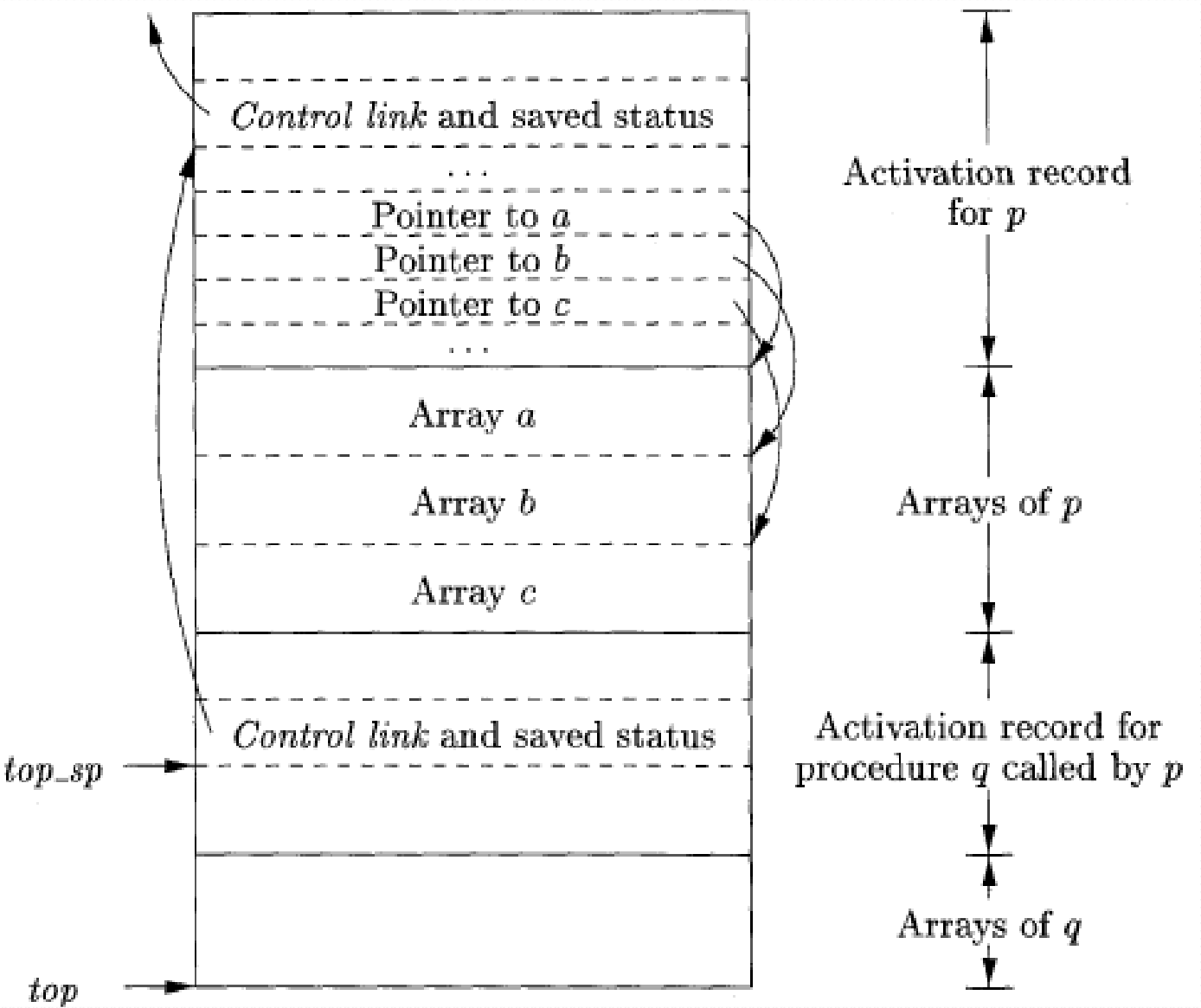
calling sequence

- The caller evaluates the actual parameters
- The caller stores a return address and the old value of *top-sp* into the callee's activation record.
- The callee saves the register values and other status information.
- The callee initializes its local data and begins execution.

corresponding return sequence

- The callee places the return value next to the parameters
- Using information in the machine-status field, the callee restores *top-sp* and other registers, and then branches to the return address that the caller placed in the status field.
- Although *top-sp* has been decremented, the caller knows where the return value is, relative to the current value of *top-sp;* the caller therefore may use that value.

Access to dynamically allocated arrays



ML

- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statement of the form:

  val (name) = (expression)

- Functions are defined using the syntax:

  fun (name) ((arguments)) = (body)

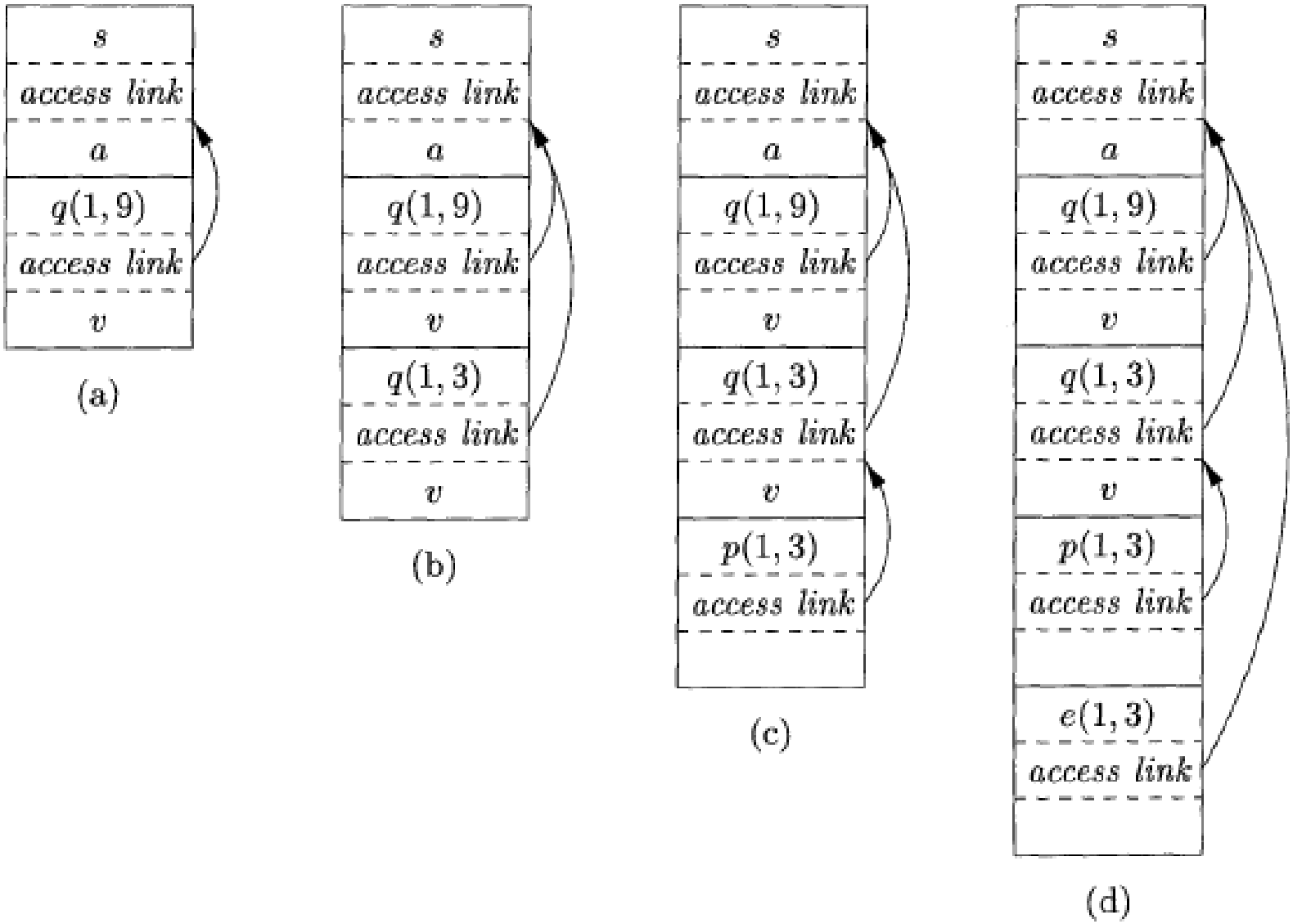
- For function bodies we shall use let-statements of the form:

  let (list of definitions) in (statements) end

A version of quicksort, in ML style, using nested functions

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

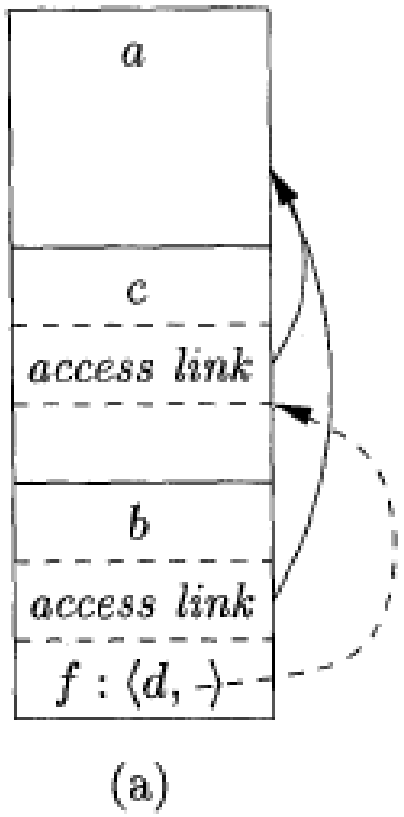
Access links for finding nonlocal data



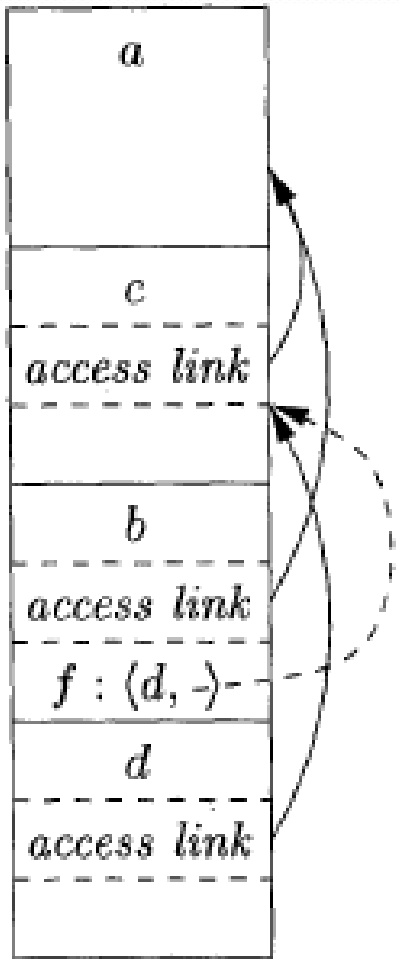
Sketch of ML program that uses function-parameters

```
fun a(x) =
    let
        fun b(f) =
            ... f ... ;
        fun c(y) =
            let
                fun d(z) = ...
            in
                ... b(d) ...
            end
    in
        ... c(1) ...
    end;
```

Actual parameters carry their access link with them

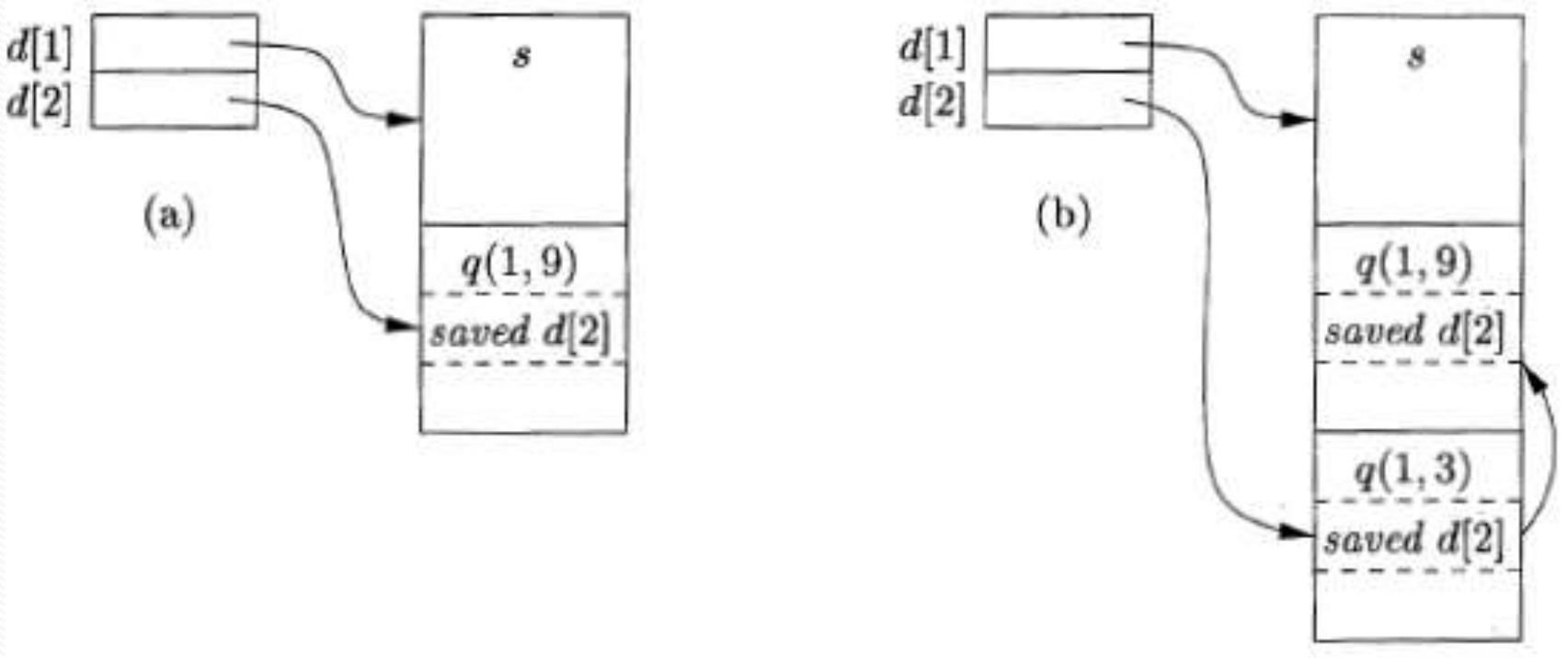


(a)

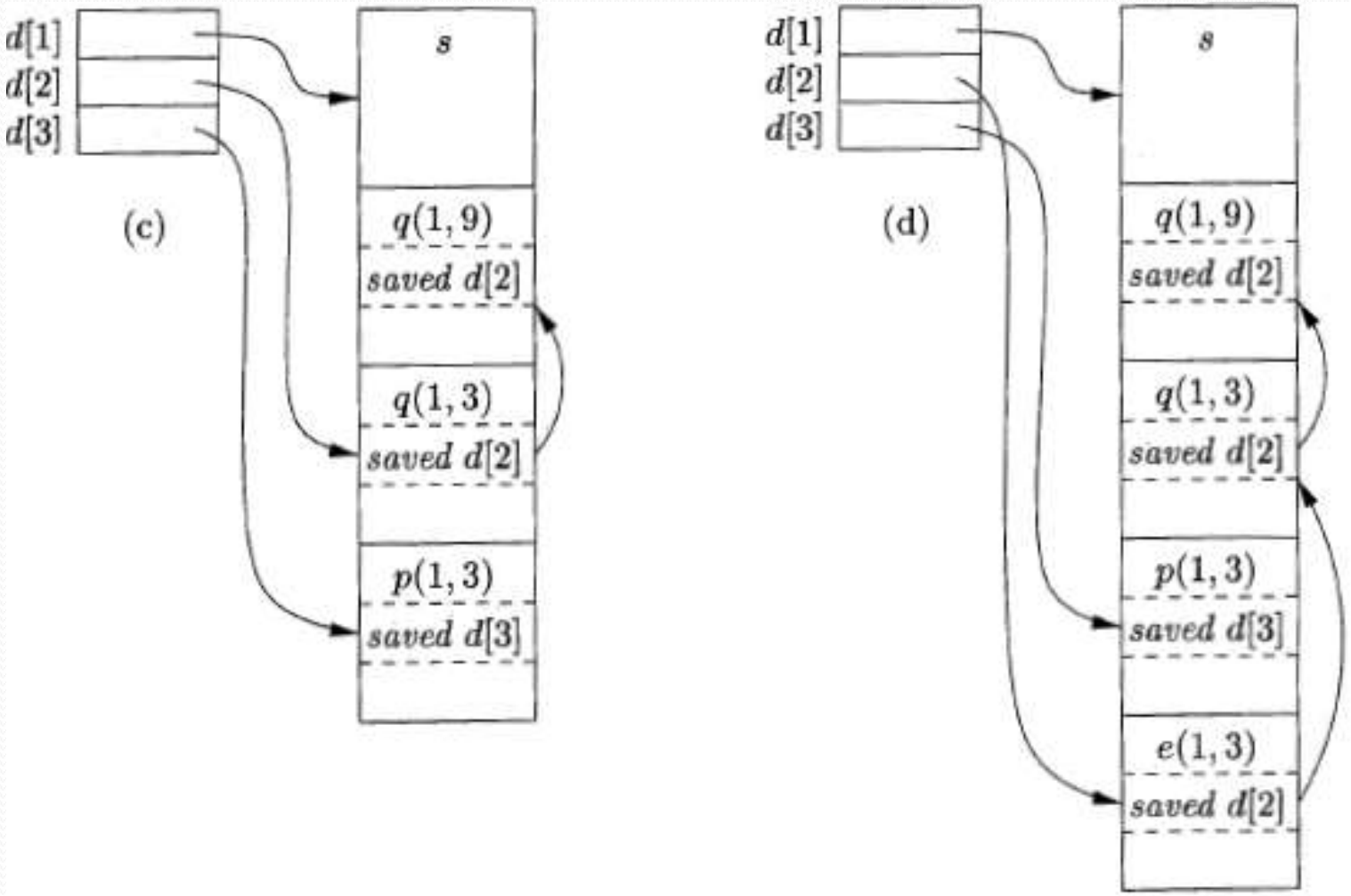


(b)

Maintaining the Display



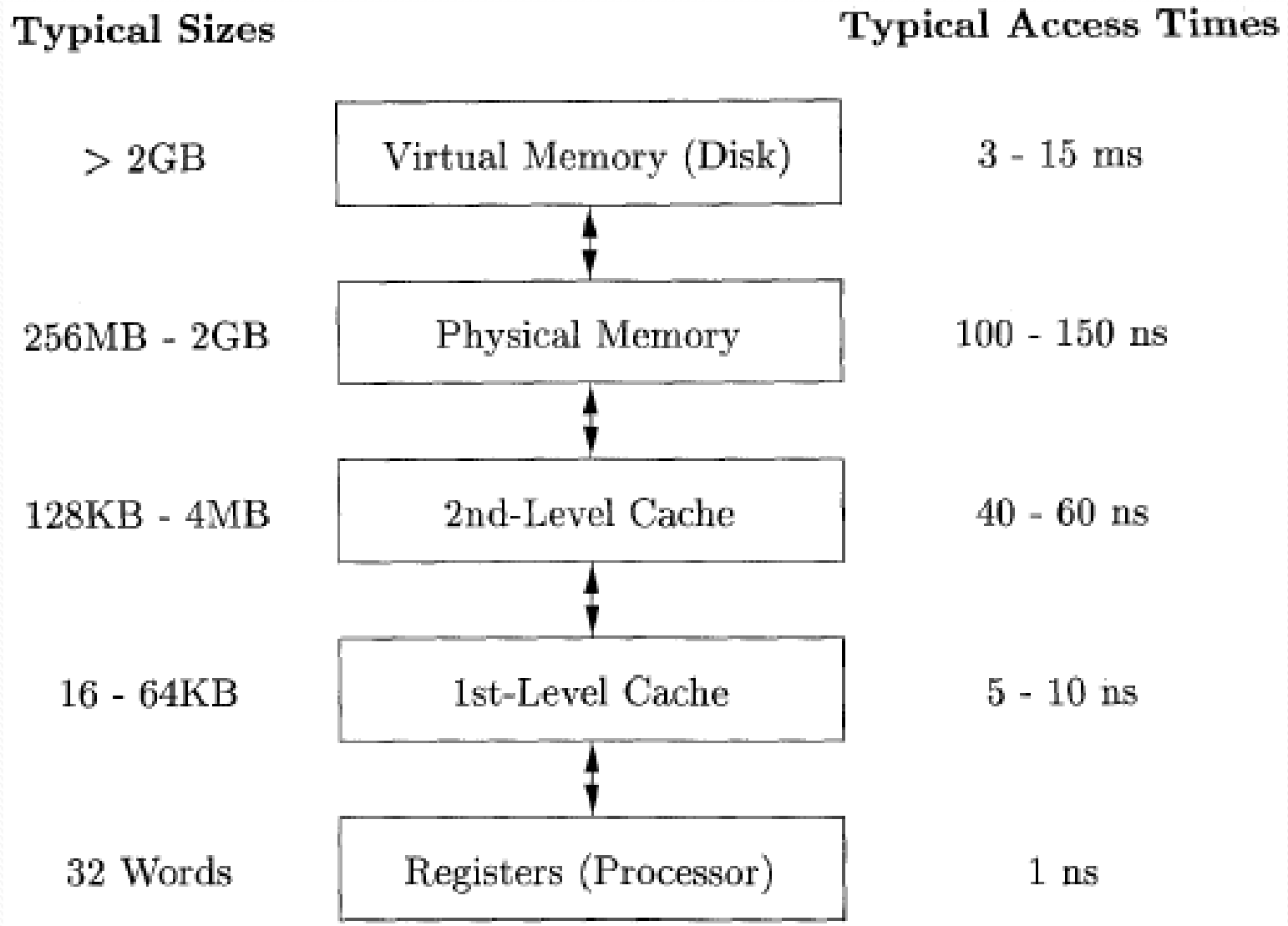
Maintaining the Display (Cont.)



Memory Manager

- Two basic functions:
  - Allocation
  - Deallocation
- Properties of memory managers:
  - Space efficiency
  - Program efficiency
  - Low overhead

Typical Memory Hierarchy Configurations

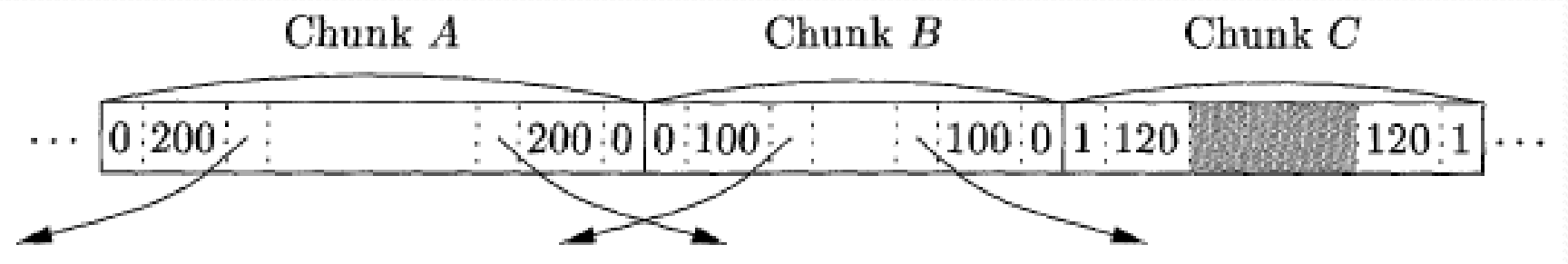


Locality in Programs

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in a typical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

Part of a Heap



Garbage Collection

Reference Counting

Mark-and-Sweep

Short-Pause Methods

The Essence

- Programming is easier if the run-time system "garbage-collects" --- makes space belonging to unusable data available for reuse.
  - Java does it; C does not.
  - But stack allocation in C gets some of the advantage.

Desiderata

1. Speed --- low overhead for garbage collector.
2. Little program interruption.
   - Many collectors shut down the program to hunt for garbage.
3. *Locality* --- data that is used together is placed together on pages, cache-lines.

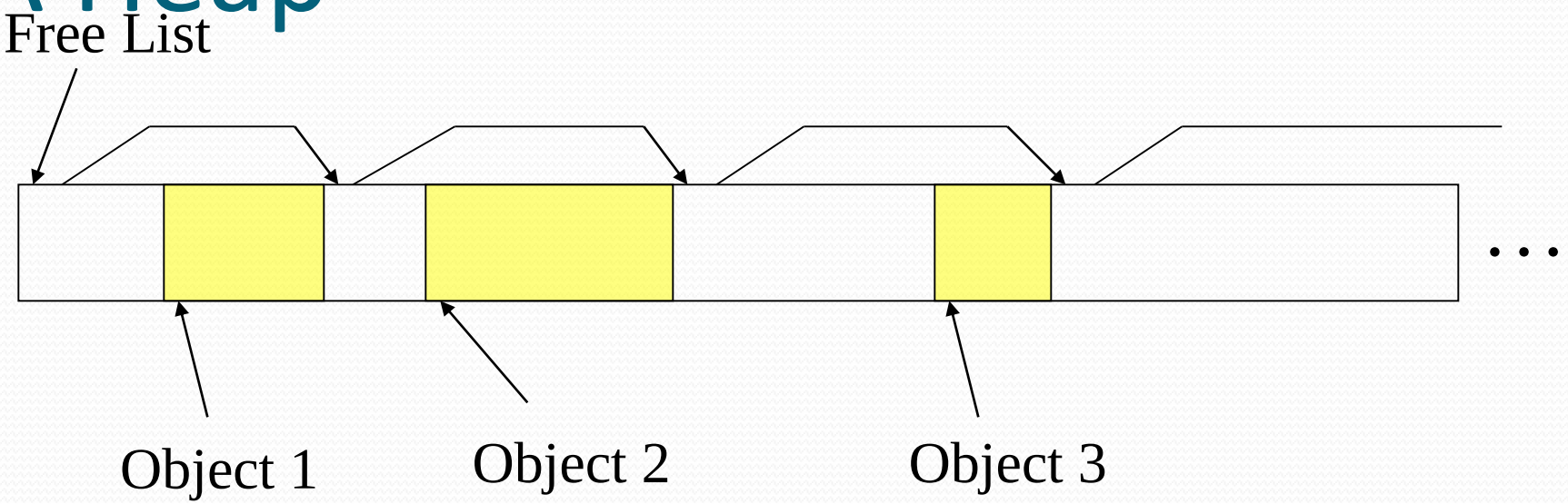
The Model --- (1)

- There is a *root set* of data that is a-priori reachable.
  - Example: In Java, root set = static class variables plus variables on run-time stack.
- *Reachable data* : root set plus anything referenced by something reachable.

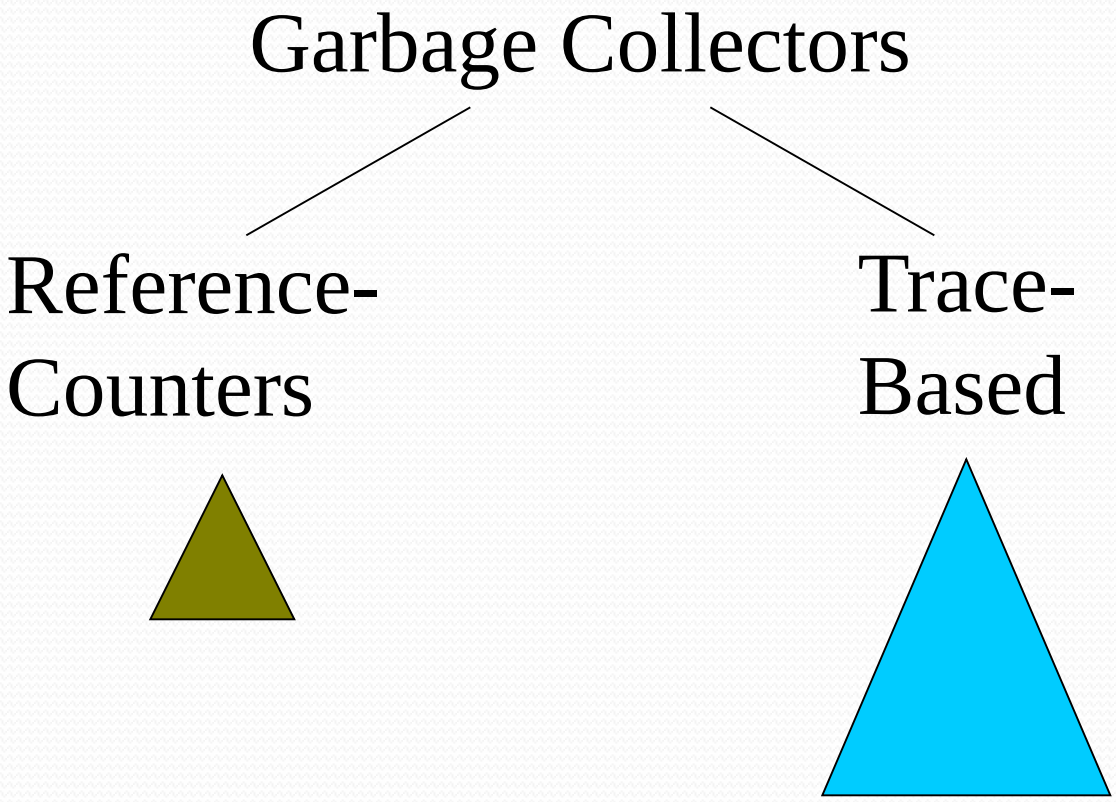
The Model --- (2)

- Things requiring space are "objects."
- Available space is in a *heap* --- large area managed by the run-time system.
  - *Allocator* finds space for new objects.
    - Space for an object is a *chunk*.
  - *Garbage collector* finds unusable objects, returns their space to the heap, and maybe moves objects around in the heap.

A Heap



Taxonomy



Reference Counting

- The simplest (but imperfect) method is to give each object a *reference count* = number of references to this object.
  - OK if objects have no internal references.
- Initially, object has one reference.
- If reference count becomes 0, object is garbage and its space becomes available.

Examples

```
Integer i = new Integer(10);
```

- Integer object is created with RC = 1.

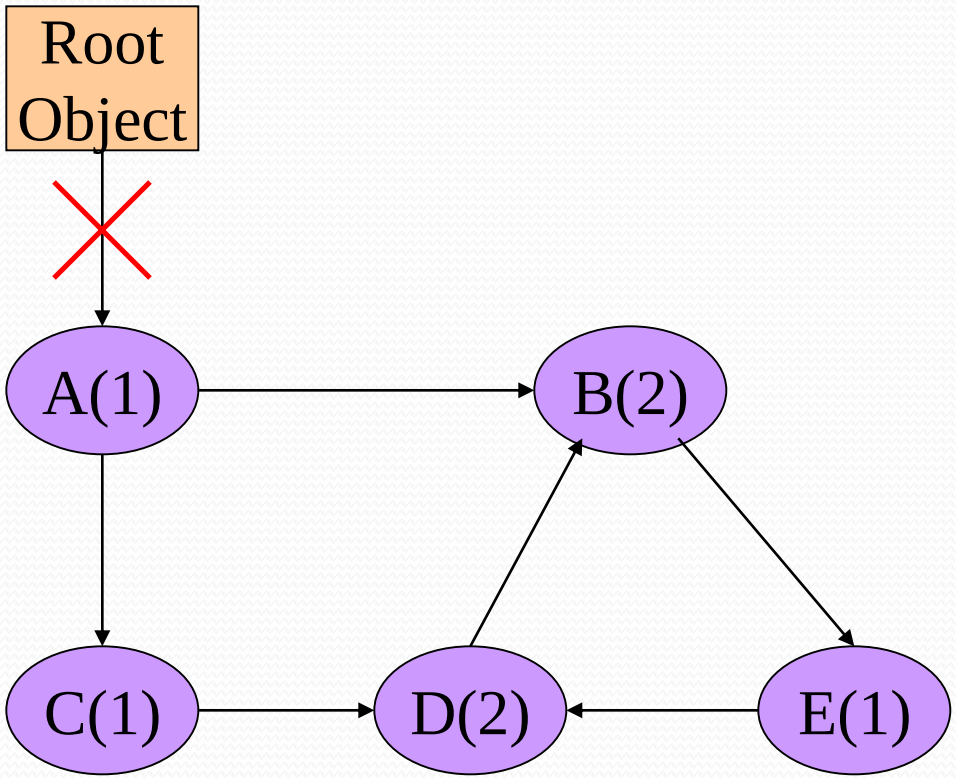
```
j = k;
```
(j, k are Integer references.)

- Object referenced by j has RC--.
- Object referenced by k has RC++.

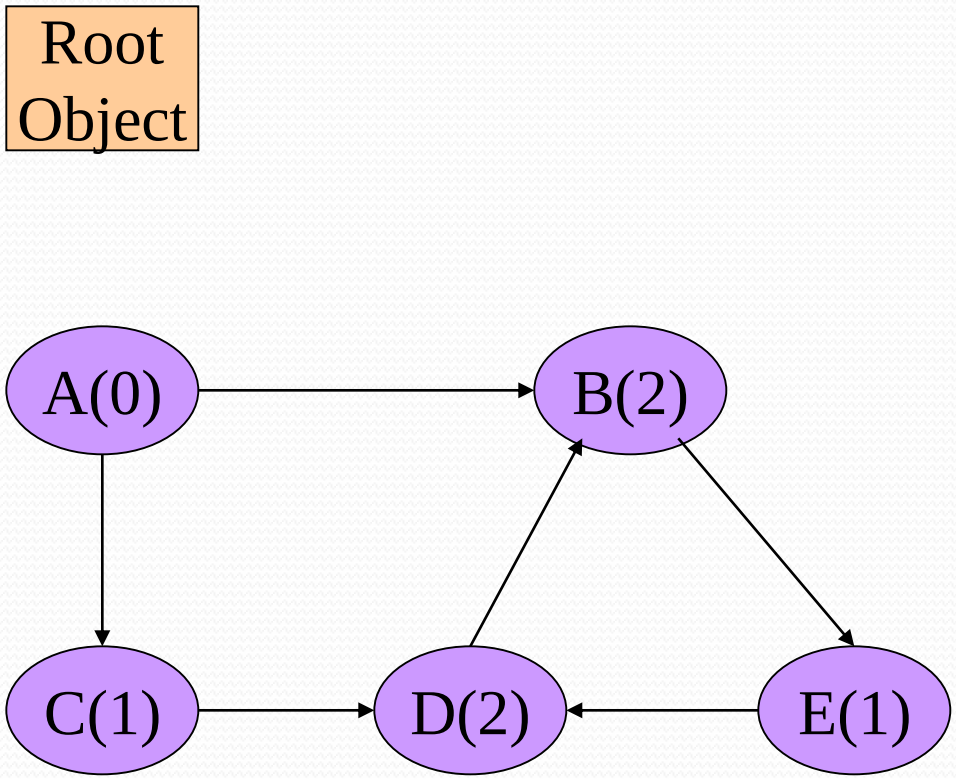
Transitive Effects

- If an object reaches RC=0 and is collected, the references within that object disappear.
- Follow these references and decrement RC in the objects reached.
- That may result in more objects with RC=0, leading to recursive collection.

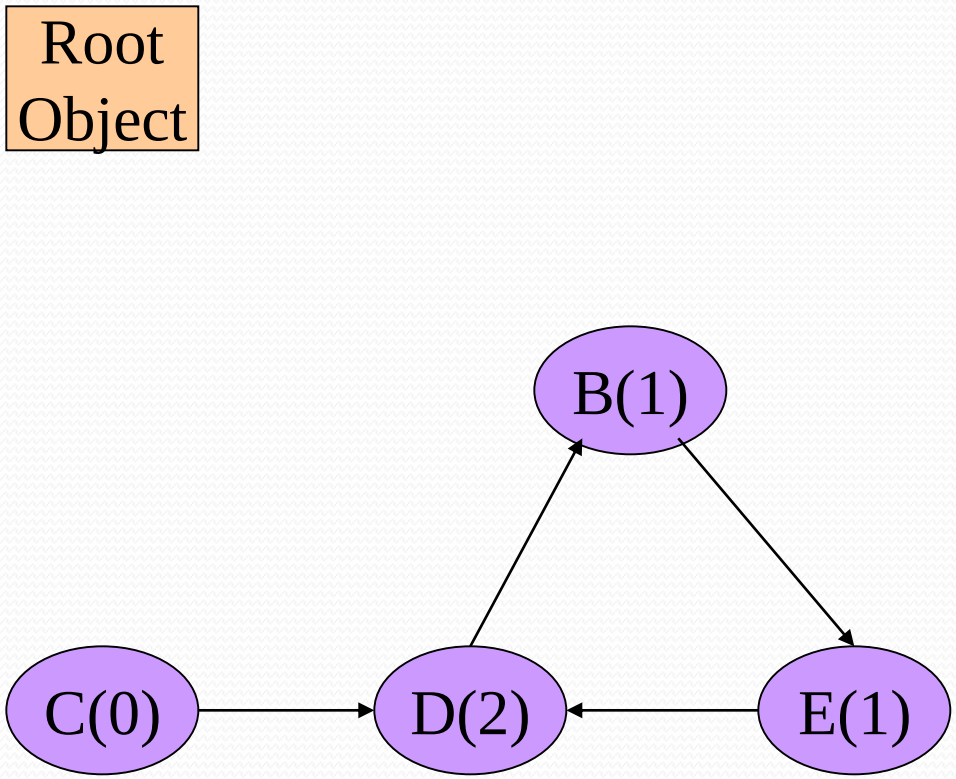
Example: Reference Counting



Example: Reference Counting



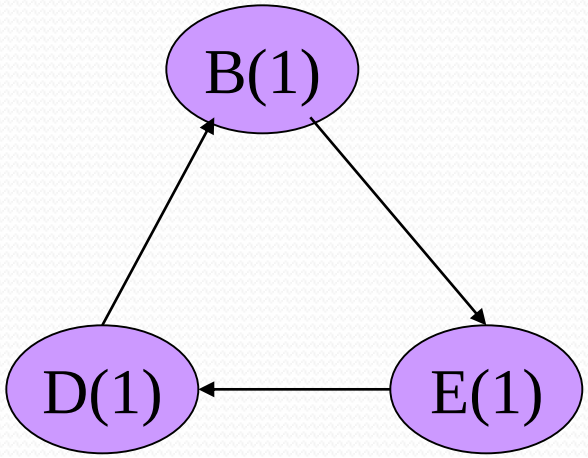
Example: Reference Counting



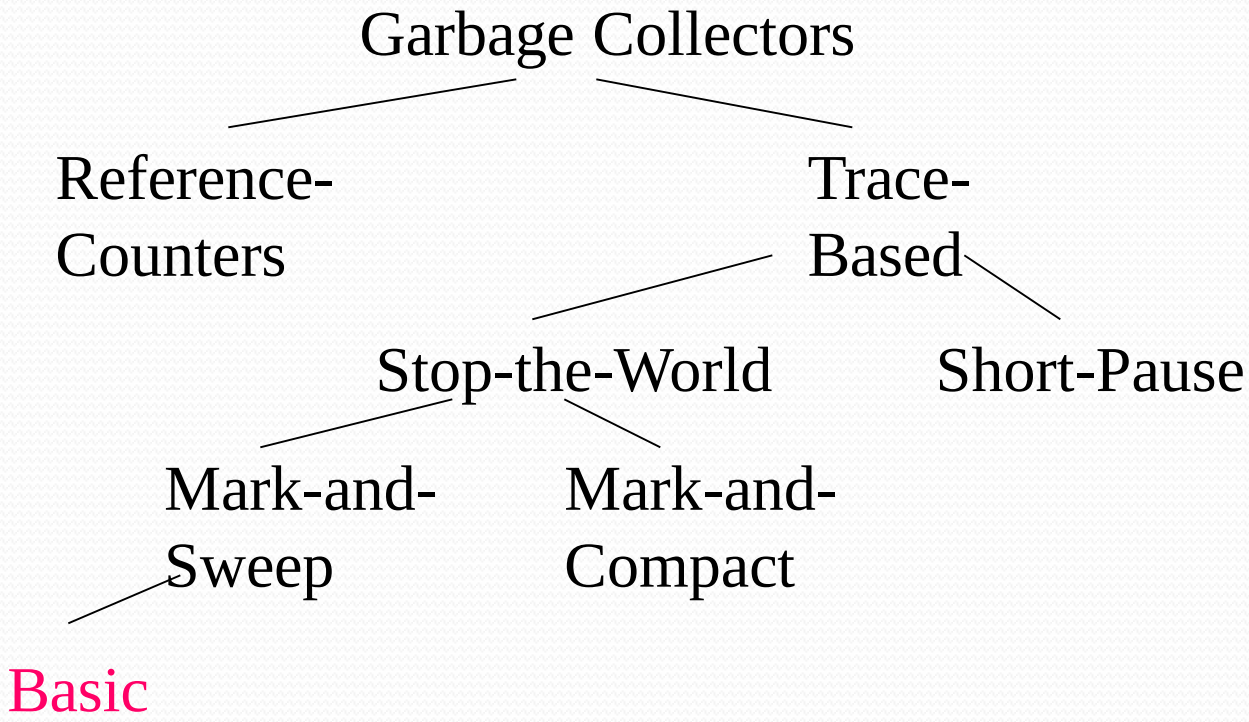
Example: Reference Counting

Root Object

B, D, and E are garbage, but their reference counts are all > 0. They never get collected.



Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Basic

Four States of Memory Chunks

1. *Free* = not holding an object; available for allocation.
2. *Unreached* = Holds an object, but has not yet been reached from the root set.
3. *Unscanned* = Reached from the root set, but its references not yet followed.
4. *Scanned* = Reached and references followed.

Marking

1. Assume all objects in Unreached state.
2. Start with the root set. Put them in state Unscanned.
3. **while** Unscanned objects remain **do**

   examine one of these objects;

   make its state be Scanned;

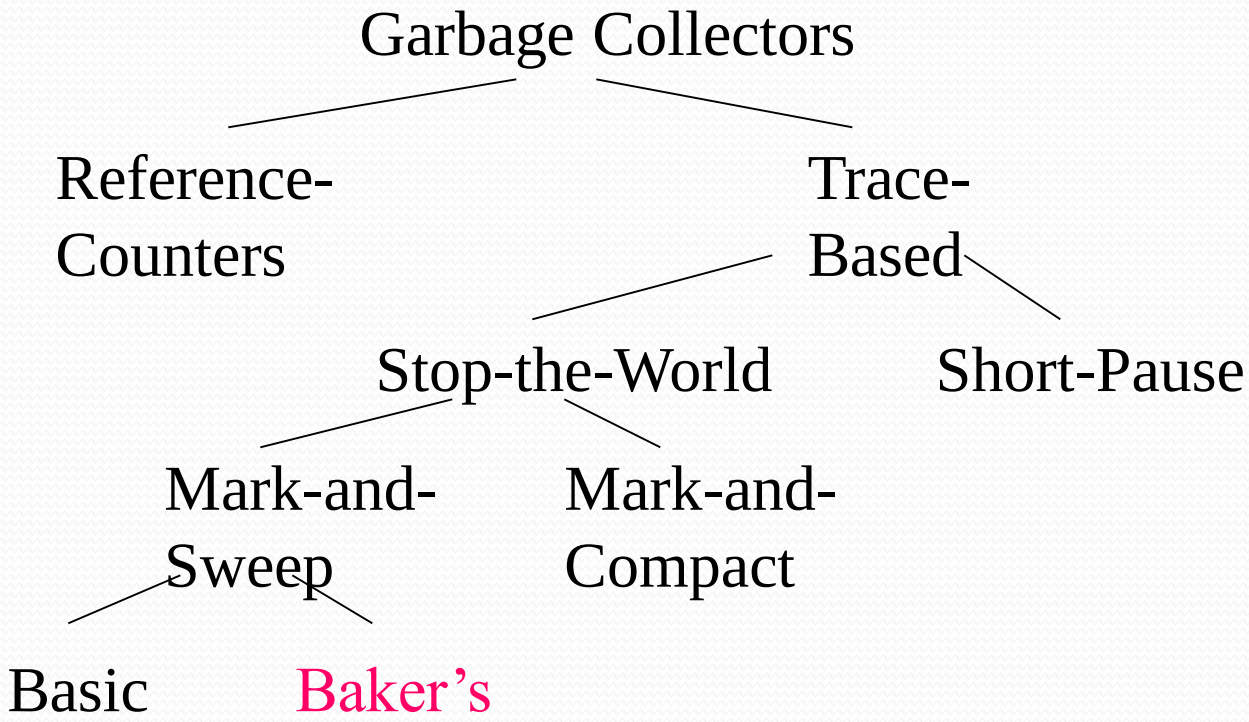
   add all referenced objects to Unscanned if they have not been there;

   end;

Sweeping

- Place all objects still in the Unreached state into the Free state.
- Place all objects in Scanned state into the Unreached state.
  - To prepare for the next mark-and-sweep.

Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Basic
Baker's

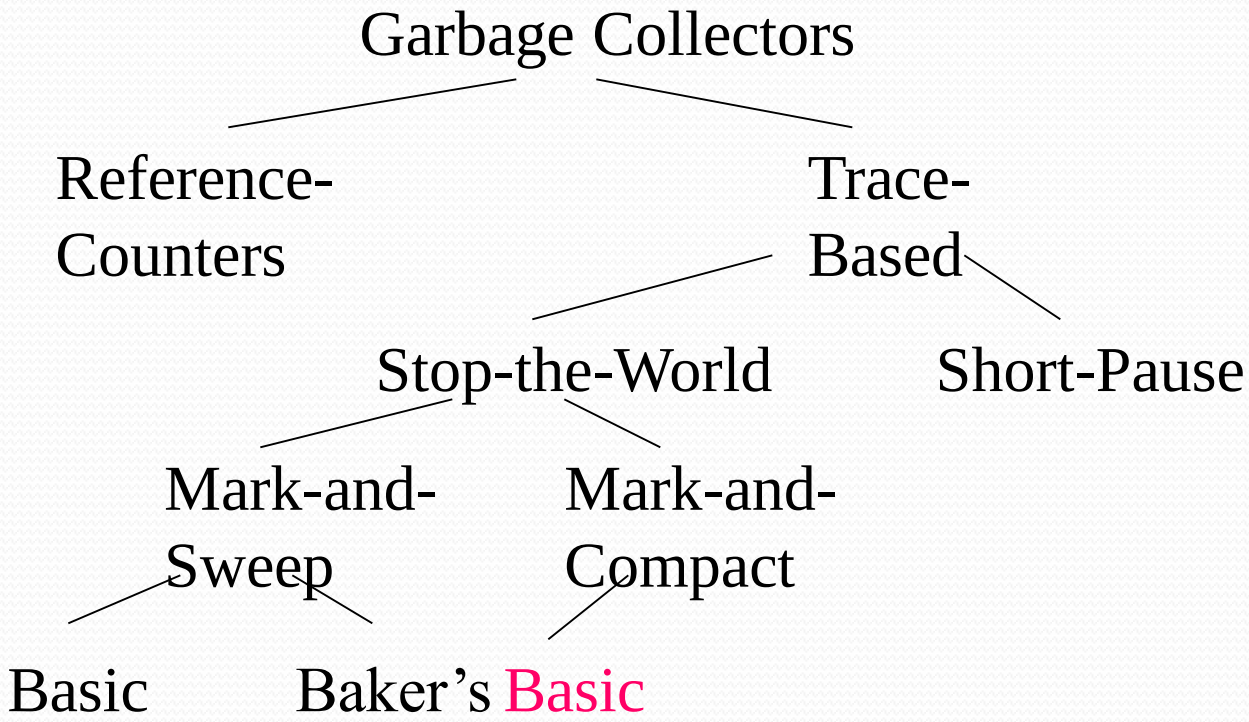
Baker's Algorithm --- (1)

- Problem: The basic algorithm takes time proportional to the heap size.
  - Because you must visit all objects to see if they are Unreached.
- Baker's algorithm keeps a list of all allocated chucks of memory, as well as the Free list.

Baker's Algorithm --- (2)

- Key change: In the sweep, look only at the list of allocated chunks.
- Those that are not marked as Scanned are garbage and are moved to the Free list.
- Those in the Scanned state are put in the Unreached state.
  - For the next collection.

Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Basic
Baker's Basic

Issue: Why Compact?

- *Compact* = move reachable objects to contiguous memory.
- *Locality* --- fewer pages or cache-lines needed to hold the active data.
- *Fragmentation* --- available space must be managed so there is space to store large objects.

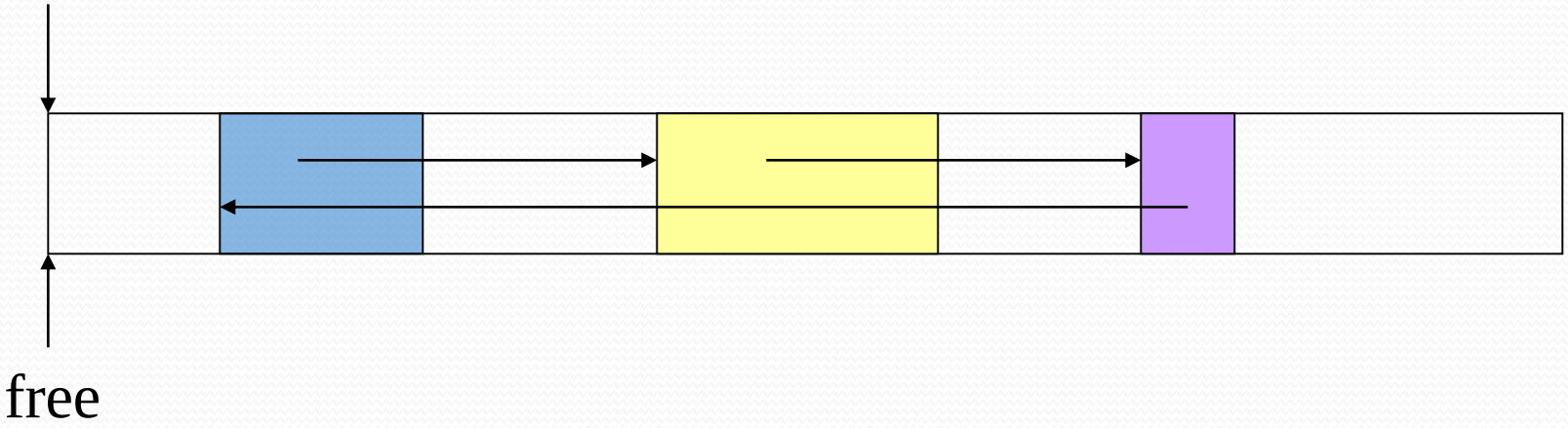
Mark-and-Compact

1. Mark reachable objects as before.
2. Maintain a table (hash?) from reached chunks to new locations for the objects in those chunks.
   - Scan chunks from low end of heap.
   - Maintain pointer *free* that counts how much space is used by reached objects so far.

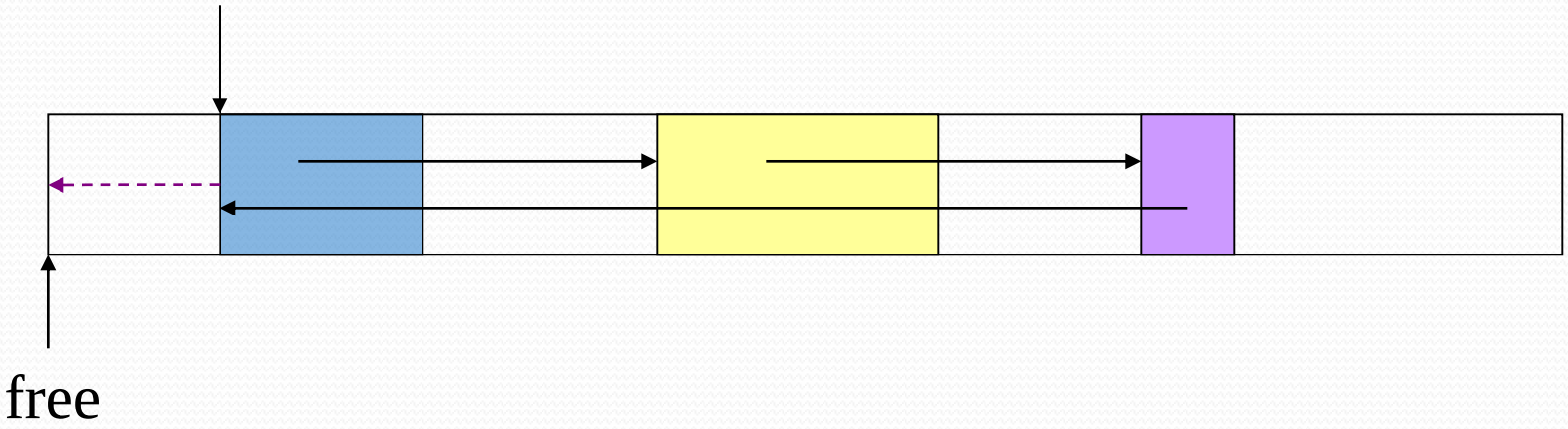
Mark-and-Compact --- (2)

3. Move all reached objects to their new locations, and also retarget all references in those objects to the new locations.
   - Use the table of new locations.
4. Retarget root references.

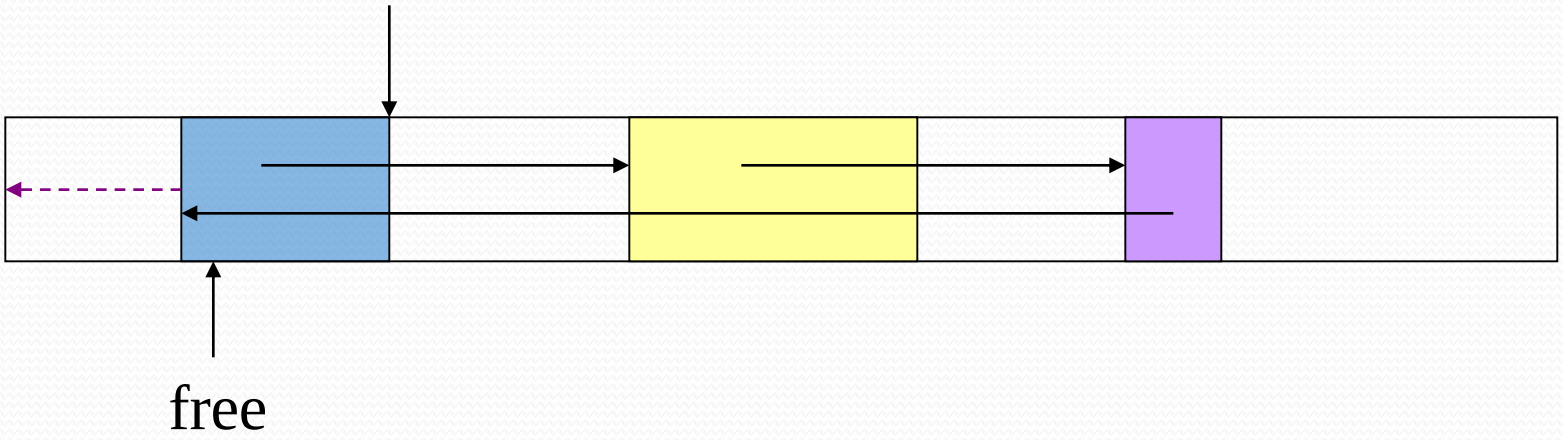
Example: Mark-and-Compact



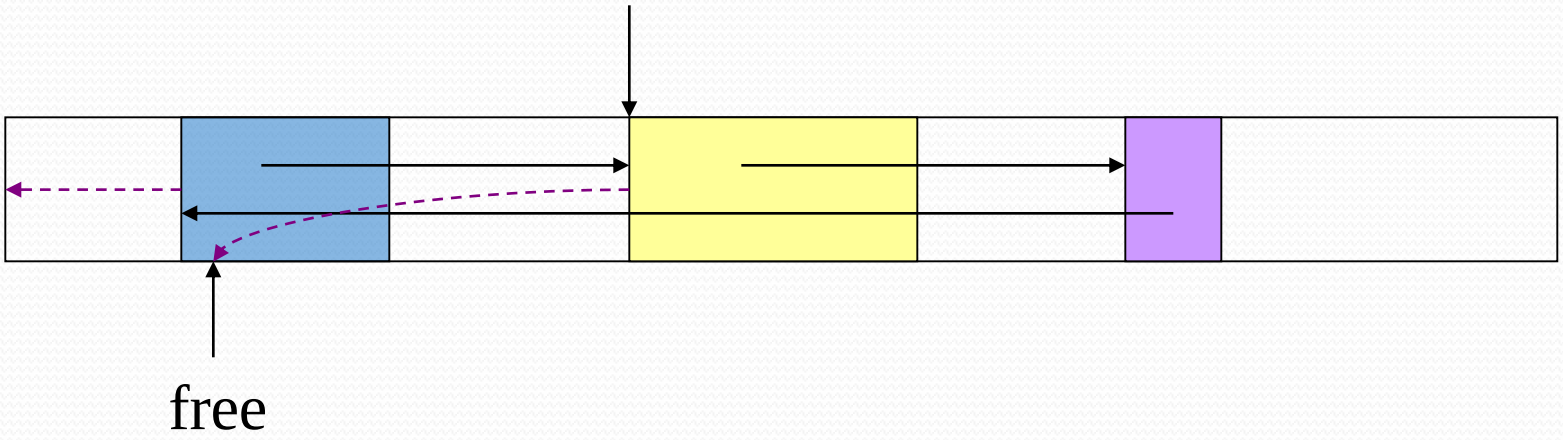
Example: Mark-and-Compact



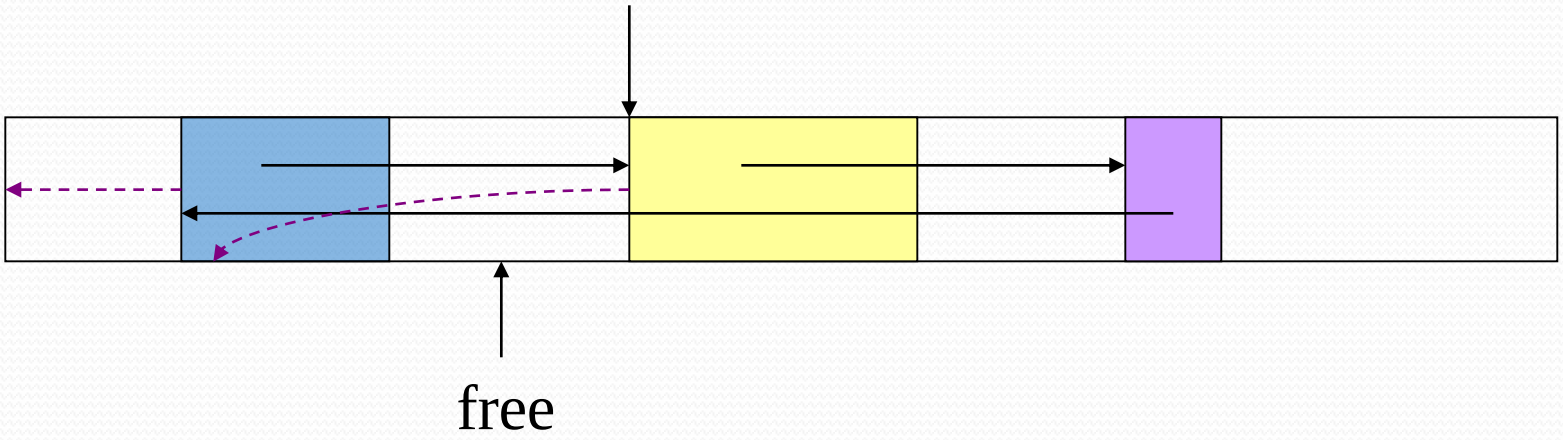
Example: Mark-and-Compact



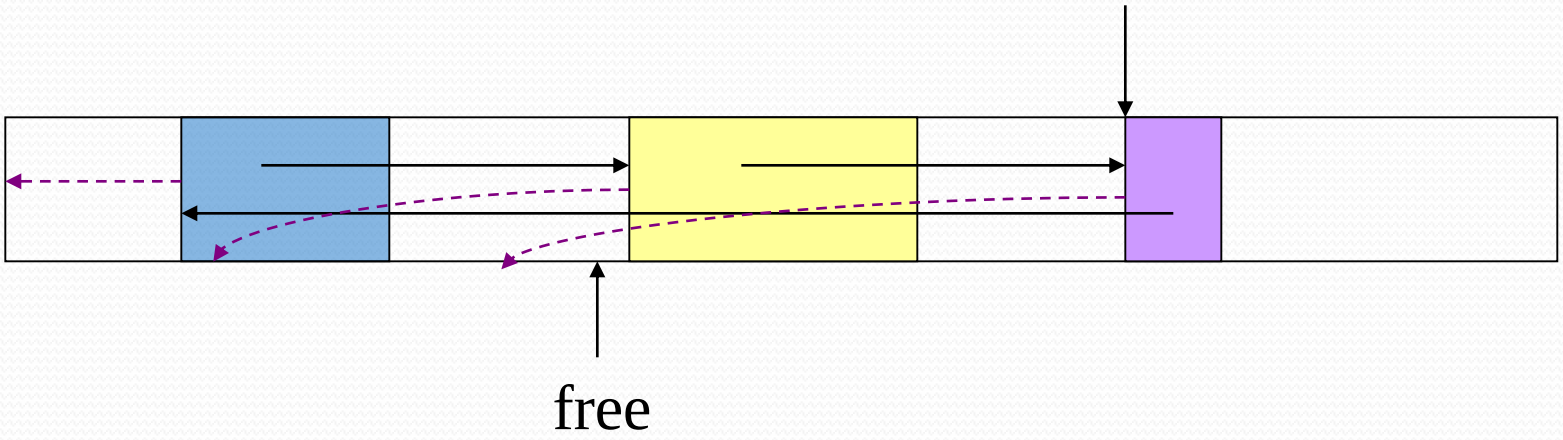
Example: Mark-and-Compact



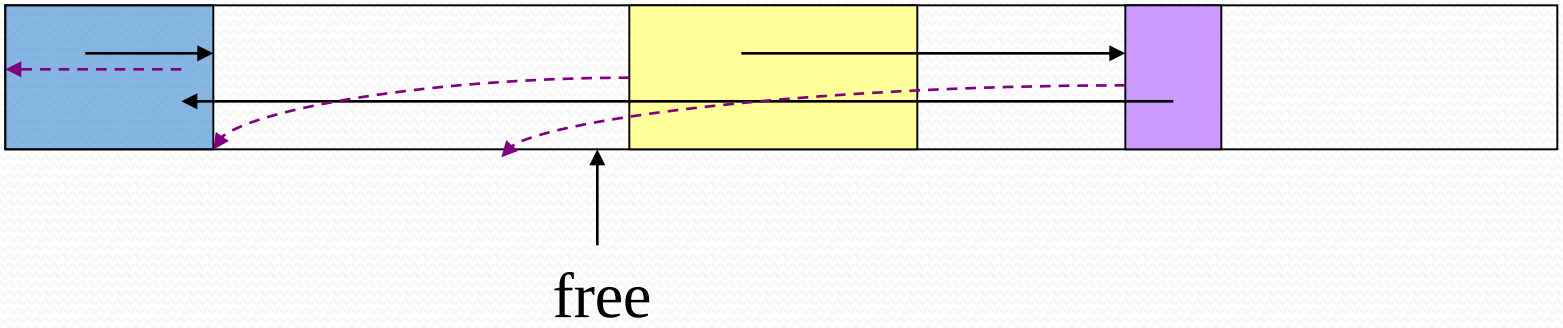
Example: Mark-and-Compact



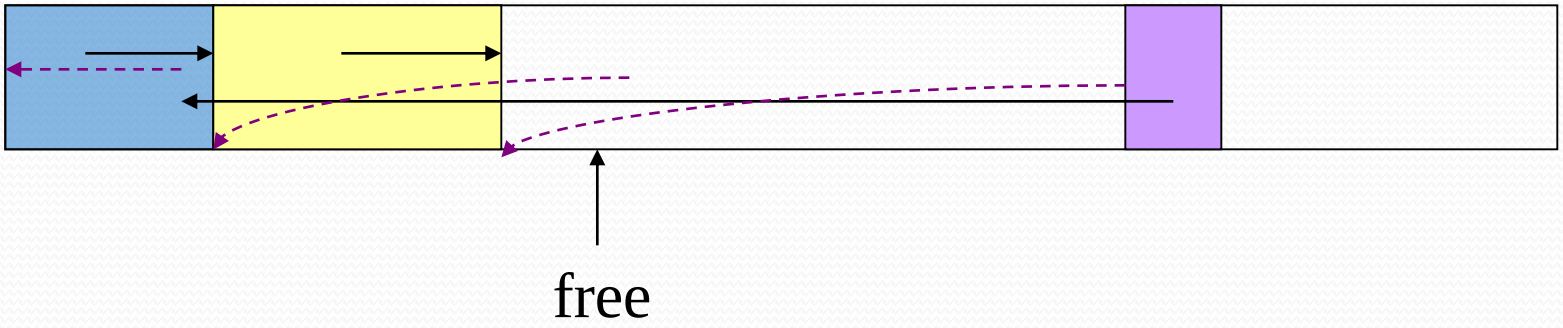
Example: Mark-and-Compact



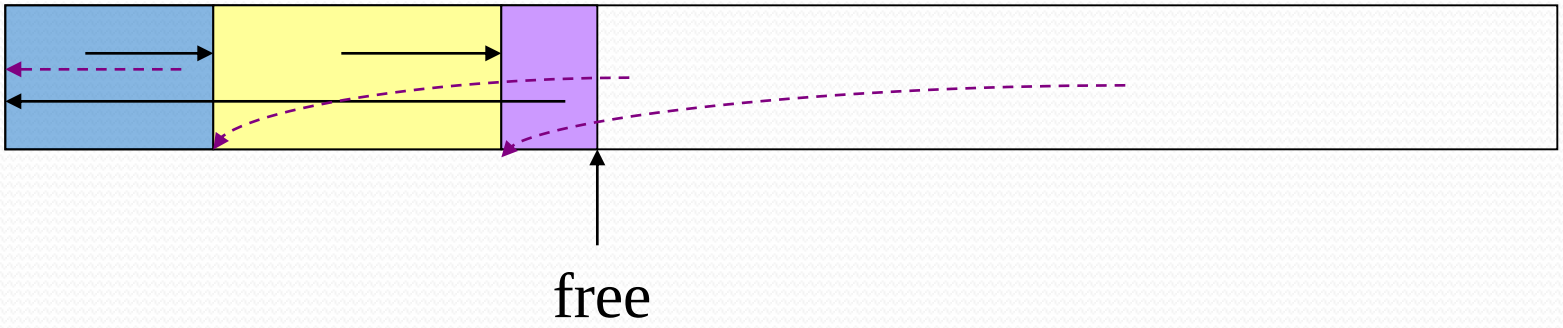
Example: Mark-and-Compact



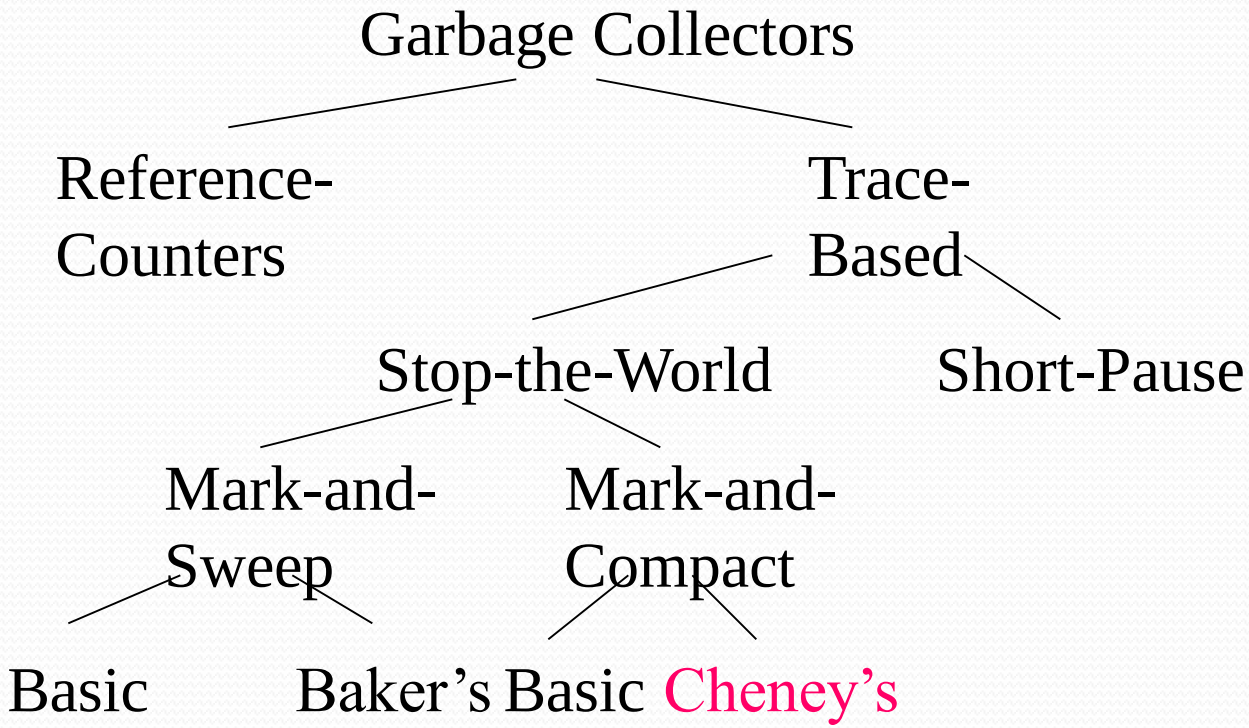
Example: Mark-and-Compact



Example: Mark-and-Compact



Taxonomy

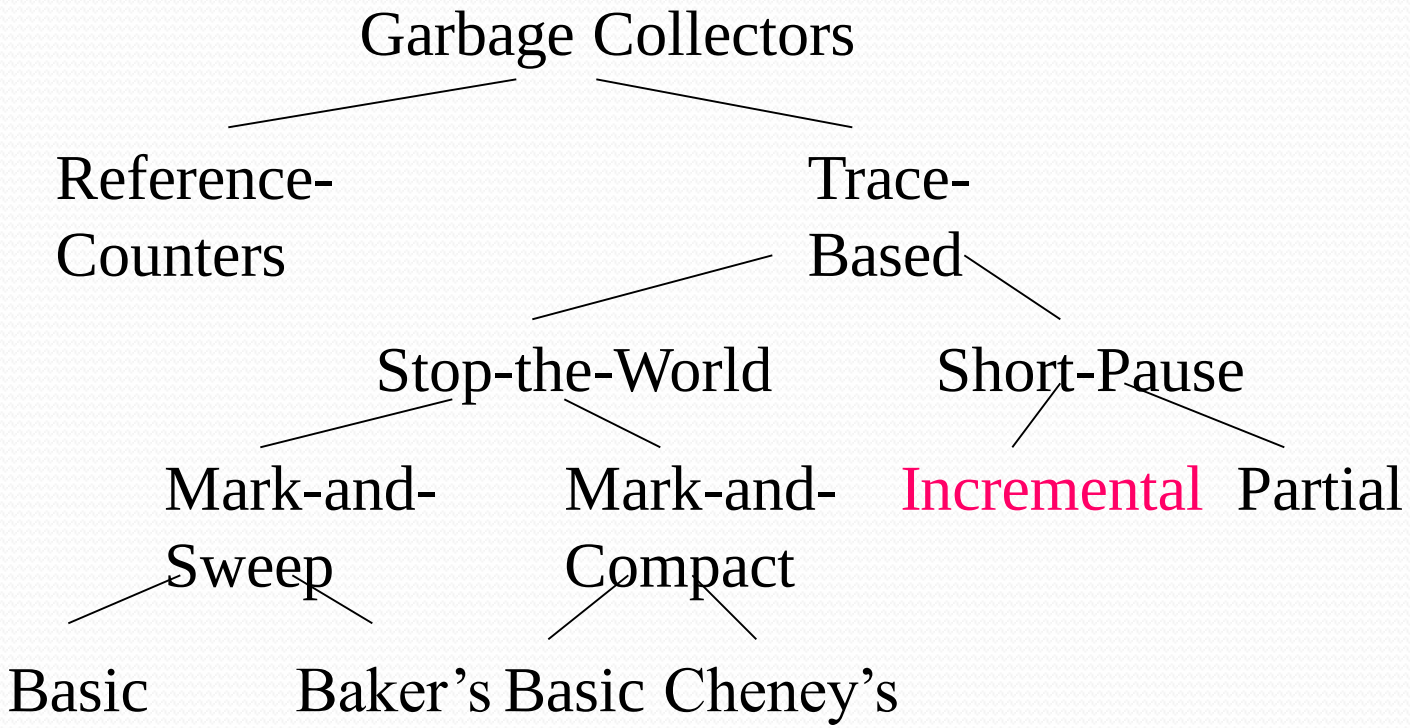


A different Cheney, BTW, so no jokes, please.

Cheney's Copying Collector

- A shotgun approach to GC.
- 2 heaps: Allocate space in one, copy to second when first is full, then swap roles.
- Maintain table of new locations.
- As soon as an object is reached, give it the next free chunk in the second heap.
- As you scan objects, adjust their references to point to second heap.

Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Incremental
Partial
Basic
Baker's Basic
Cheney's

Short-Pause Garbage-Collection

1. *Incremental* --- run garbage collection in parallel with *mutation* (operation of the program).
2. *Partial* --- stop the mutation, but only briefly, to garbage collect a part of the heap.

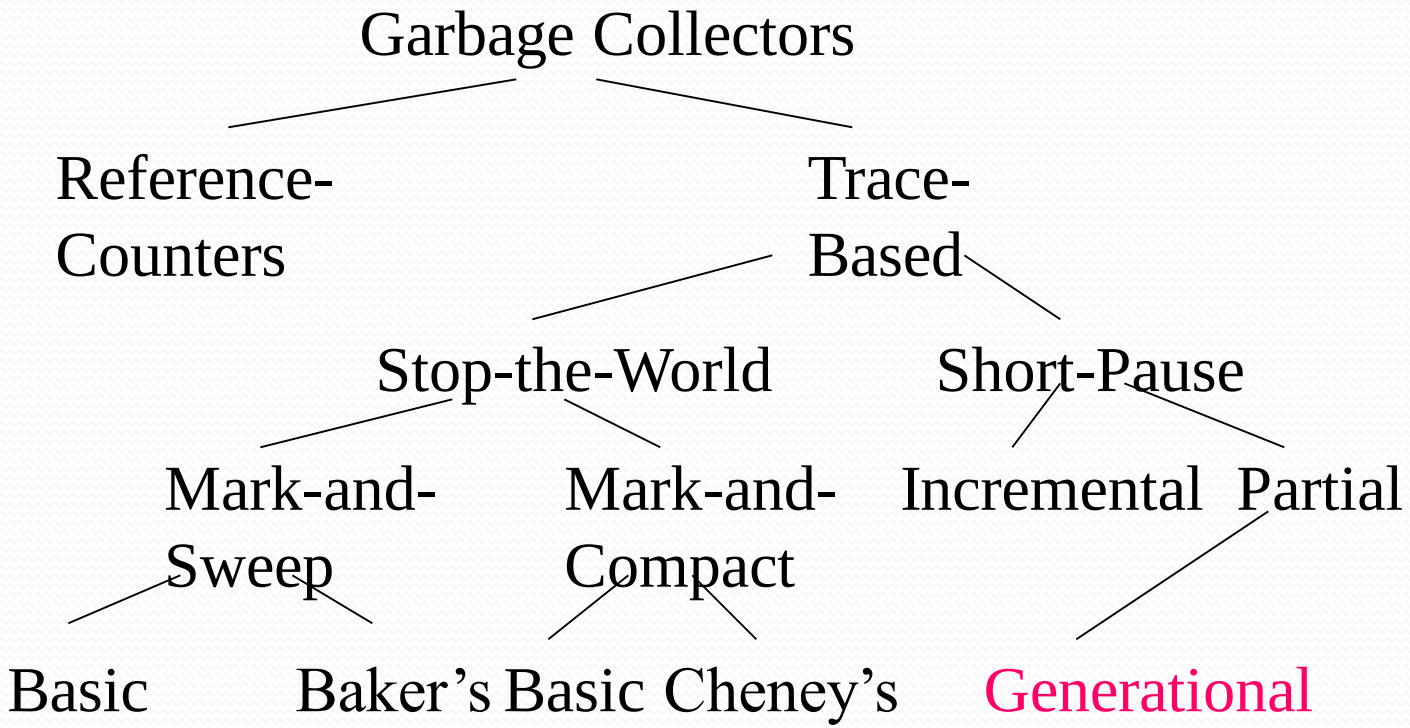
Problem With Incremental GC

- OK to mark garbage as reachable.
- Not OK to GC a reachable object.
- If a reference r within a Scanned object is mutated to point to an Unreached object, the latter may be garbage-collected anyway.
  - Subtle point: How do you point to an Unreached object?

One Solution: *Write Barriers*

- Intercept every write of a reference in a scanned object.
- Place the new object referred to on the Unscanned list.
- A trick: protect all pages containing Scanned objects.
  - A hardware interrupt will invoke the fixup.

Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Incremental
Partial
Basic
Baker's Basic
Cheney's
Generational

The Object Life-Cycle

- “Most objects die young.”
  - But those that survive one GC are likely to survive many.
- Tailor GC to spend more time on regions of the heap where objects have just been created.
  - Gives a better ratio of reclaimed space per unit time.

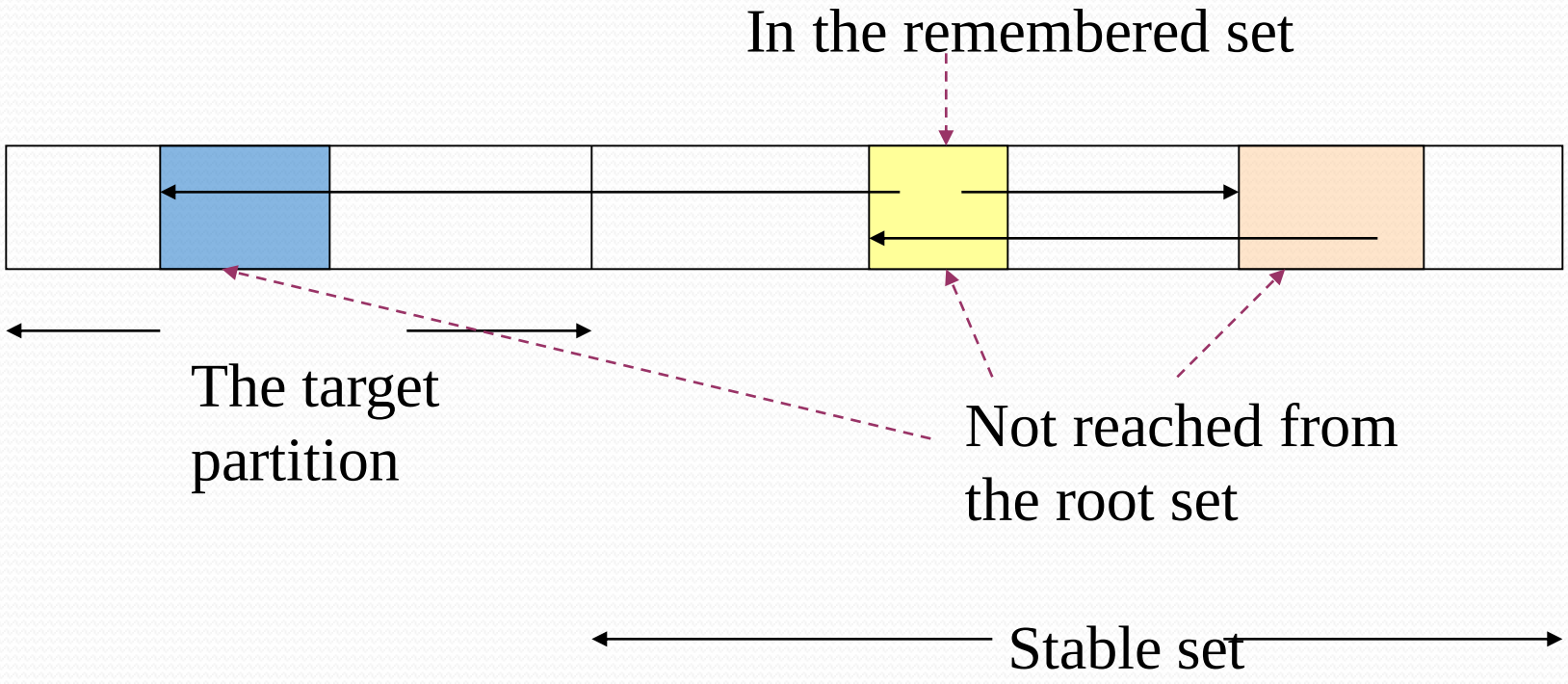
Partial Garbage Collection

- We collect one part(ition) of the heap.
  - The *target* set.
- We maintain for each partition a *remembered* set of those objects outside the partition (the *stable* set) that refer to objects in the target set.
  - Write barriers can be used to maintain the remembered set.

Collecting a Partition

- To collect a part of the heap:
  1. Add the remembered set for that partition to the root set.
  2. Do a reachability analysis as before.
- Note the resulting Scanned set may include garbage.

Example: "Reachable" Garbage



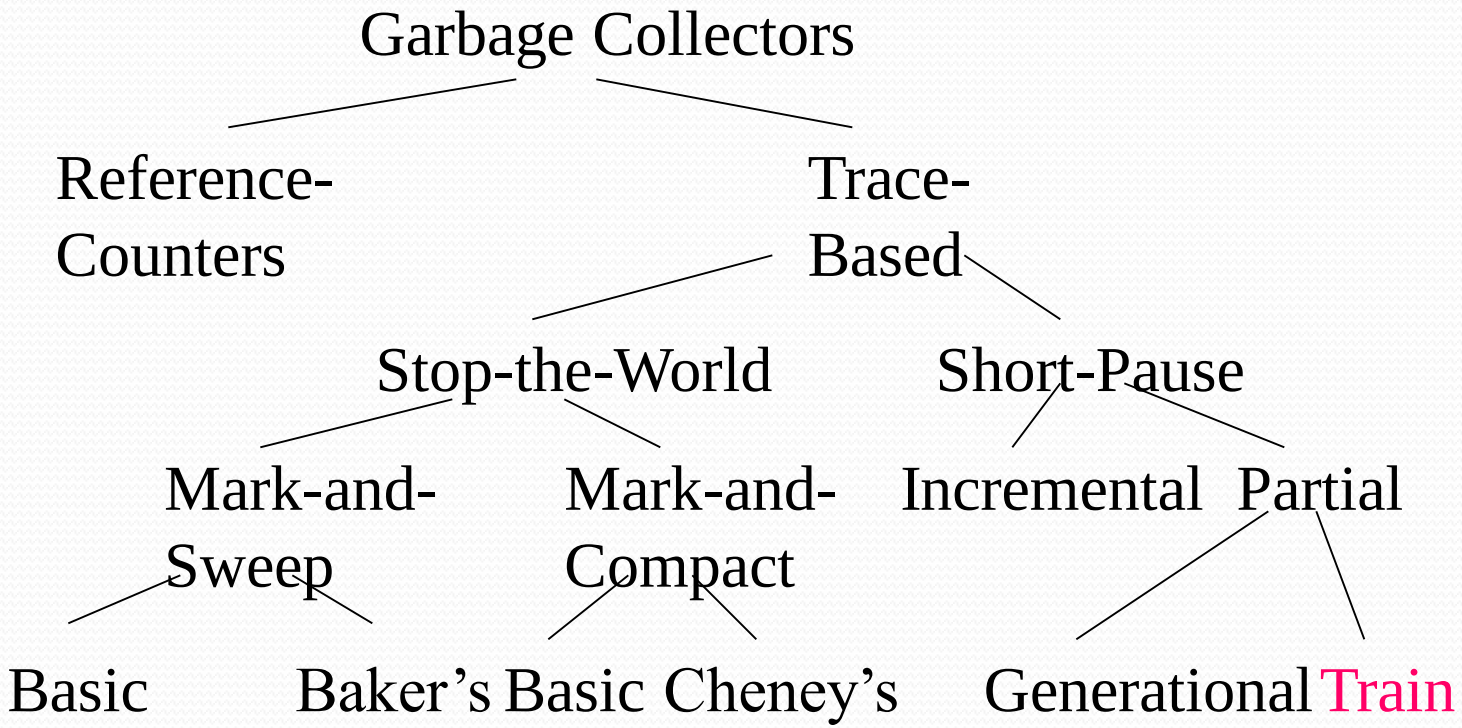
Generational Garbage Collection

- Divide the heap into partitions P0, P1,…
  - Each partition holds older objects than the one before it.
- Create new objects in P0, until it fills up.
- Garbage collect P0 only, and move the reachable objects to P1.

Generational GC --- (2)

- When P1 fills, garbage collect P0 and P1, and put the reachable objects in P2.
- In general: When P*i* fills, collect P0, P1,…,P*i* and put the reachable objects in P(*i* +1).

Taxonomy


Garbage Collectors
Reference-Counters
Trace-Based
Stop-the-World
Short-Pause
Mark-and-Sweep
Mark-and-Compact
Incremental
Partial
Basic
Baker's
Basic
Cheney's
Generational
Train

The Train Algorithm

- Problem with generational GC:
  1. Occasional total collection (last partition).
  2. Long-lived objects move many times.
- Train algorithm useful for long-lived objects.
  - Replaces the higher-numbered partitions in generational GC.

Partitions = “Cars”

Train 1 Car 11 Car 12 Car 13

Train 2 Car 21 Car 22 . . . Car 2*k*

.

.

.

Train *n* Car *n*1 Car *n*2

Organization of Heap

- There can be any number of trains, and each train can have any number of cars.
  - You need to decide on a policy that gives a reasonable number of each.
- New objects can be placed in last car of last train, or start a new car or even a new train.

Garbage-Collection Steps

1. Collect the first car of the first train.
2. Collect the entire first train if there are no references from the root set or other trains.
   - Important: this is how we find and eliminate large, cyclic garbage structures.

Remembered Sets

- Each car has a remembered set of references from later trains and later cars of the same train.
- Important: since we only collect first cars and trains, we never need to worry about "forward" references (to later trains or later cars of the same train).

Collecting the First Car of the First Train

- Do a partial collection as before, using every other car/train as the stable set.
- Move all Reachable objects of the first car somewhere else.
- Get rid of the car.

Moving Reachable Objects

- If object o has a reference from another train, pick one such train and move o to that train.
  - Same car as reference, if possible, else make new car.
- If references only from root set or first train, move o to another car of first train, or create new car.

Panic Mode

- The problem: it is possible that when collecting the first car, nothing is garbage.
- We then have to create a new car of the first train that is essentially the same as the old first car.

Panic Mode --- (2)

- If that happens, we go into *panic mode*, which requires that:
  1. If a reference to any object in the first train is rewritten, we make the new reference a "dummy" member of the root set.
  2. During GC, if we encounter a reference from the "root set," we move the referenced object to another train.

Panic Mode --- (3)

- Subtle point: If references to the first train never mutate, eventually all reachable objects will be sucked out of the first train, leaving cyclic garbage.
- But perversely, the last reference to a first-train object could move around so it is never to the first car.