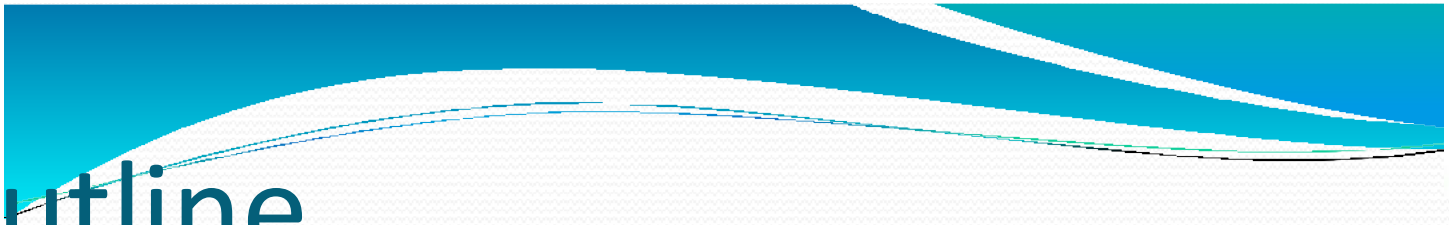
23IT601: Automata theory and compiler design

UNIT-I

Topic: Introduction to finite automata

M. Rajitha
Assistant Professor

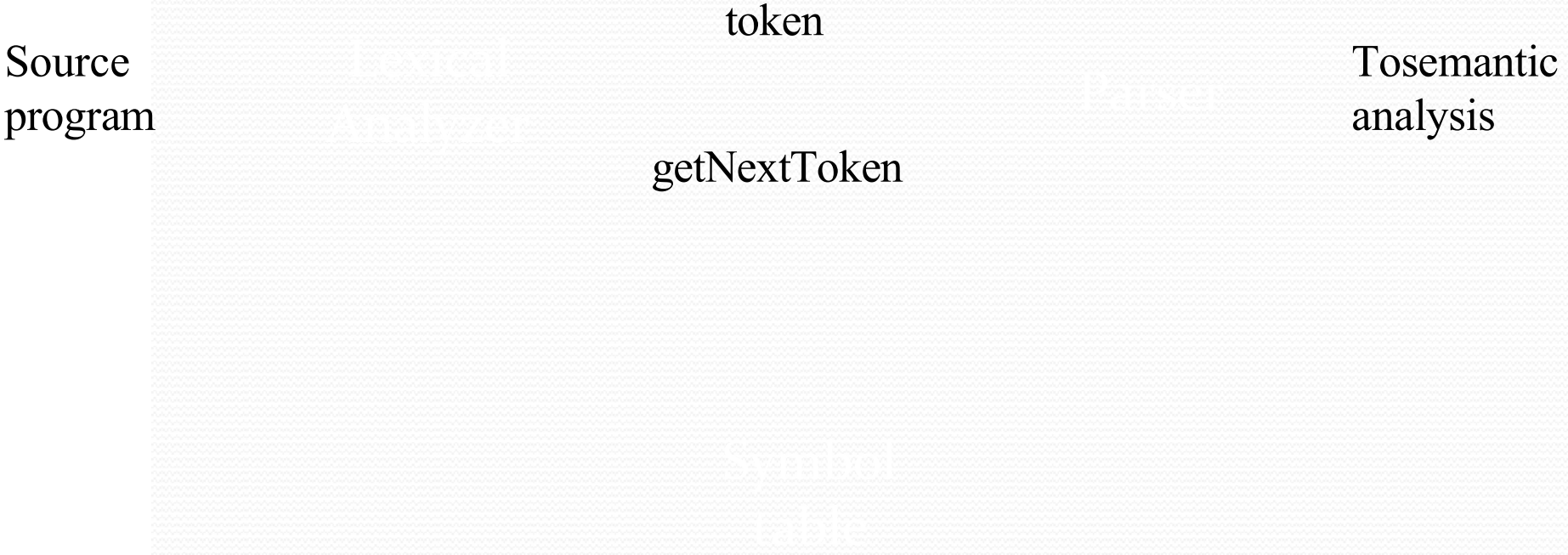
NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India
UGC - Autonomous Institute
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH



Outline

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

Theroleoflexicalanalyzer



WhytoseparateLexicalanalysis and parsing

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

Tokens,PatternsandLexemes

- 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	Informaldescription	Samplelexemes
if	Charactersi,f	if
else	Characterse,l,s,e	else
comparison	< or > or <= or >= or == or !=	<=,!=
id	Letterfollowedbyletteranddigits	pi,score,D2
number	Anynumeric constant	3.14159,0,6.02e23
literal	Anythingbut“sorroundedby“	“coredumped”

printf(“total=%d\n”,score);

Attributesfortokens

- E=M*C**2
  - <id,pointertosymboltableentryforE>
  - <assign-op>
  - <id,pointertosymboltableentryforM>
  - <mult-op>
  - <id,pointertosymboltableentryforC>
  - <exp-op>
  - <number,integervalue2>

Lexicalerrors

- Someerrorsareoutofpoweroflexicalanalyzerto recognize:
  - fi(a==f(x))…
- Howeveritmaybeabletorecognizeerrorslike:
  - d=2r
- Sucherrorsarerecognizedwhennopatternfortokens matches acharactersequence

Error recovery

- Panic mode:successive characters a reignoreduntilwe reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into there main ing input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer need to look ahead some symbols to decide about the token toreturn
  - In C language: we need to look after-,=or<todecide what token to return
  - In Fortran:DO5I=1.25
- We need to introduce at wo buffers cheme to handle large look-aheads safely

Sentinels


```
Switch(*forward++){
    Case of:
            if(forwardisatendoffirstbuffer){ reload
                    second buffer;
                    forward=beginningofsecondbuffer;
            }
            Elseif {forward is at end of secondbuffer){
                    reload first buffer;\
                    forward=beginning offirst buffer;
            }
            else/*eof  within a buffermarks 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 the naisa 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 dentingL(r)

Regulardefinitions

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:

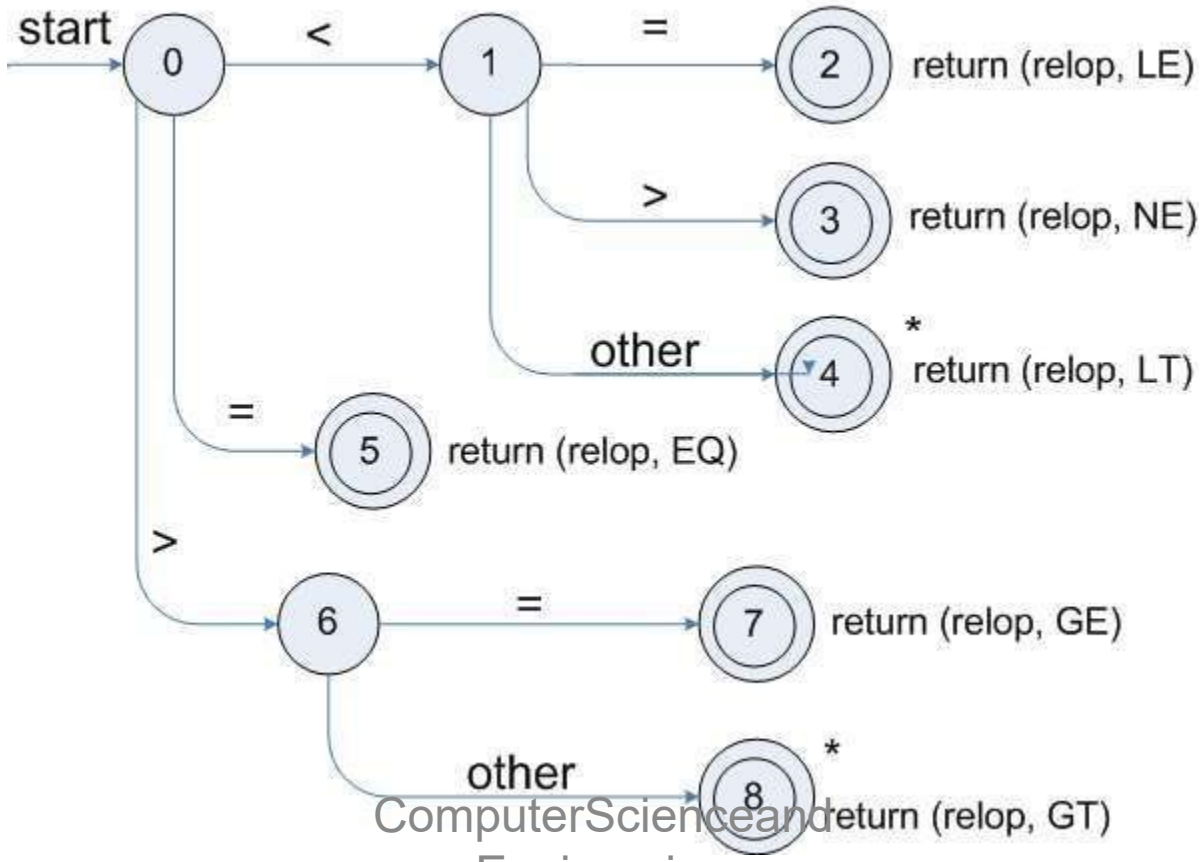
  digit ->[0-9]
  Digits ->digit+
  number->digit(.digits)?(E[+-]?Digit)?
  letter->[A-Za-z_]
  id ->letter(letter|digit)*
  If ->if
  Then ->then
  Else ->else
  Relop ->< |>|<=|>=| =|<>

- We also need to handle whitespaces:

  ws->(blank|tab|newline)+

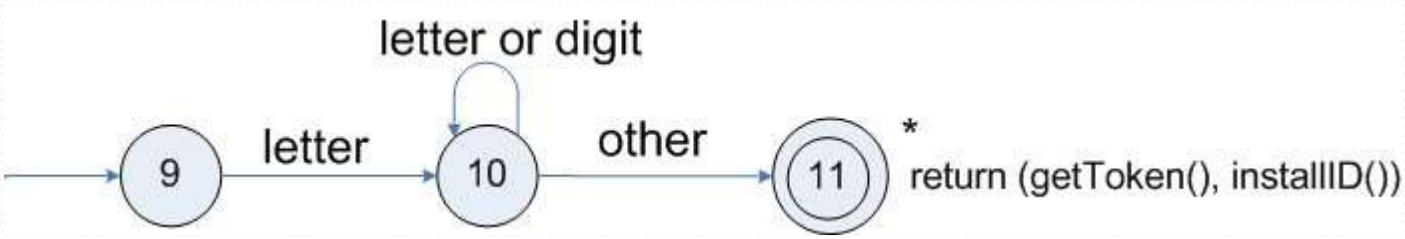
Transition diagrams

- Transition diagram for relop



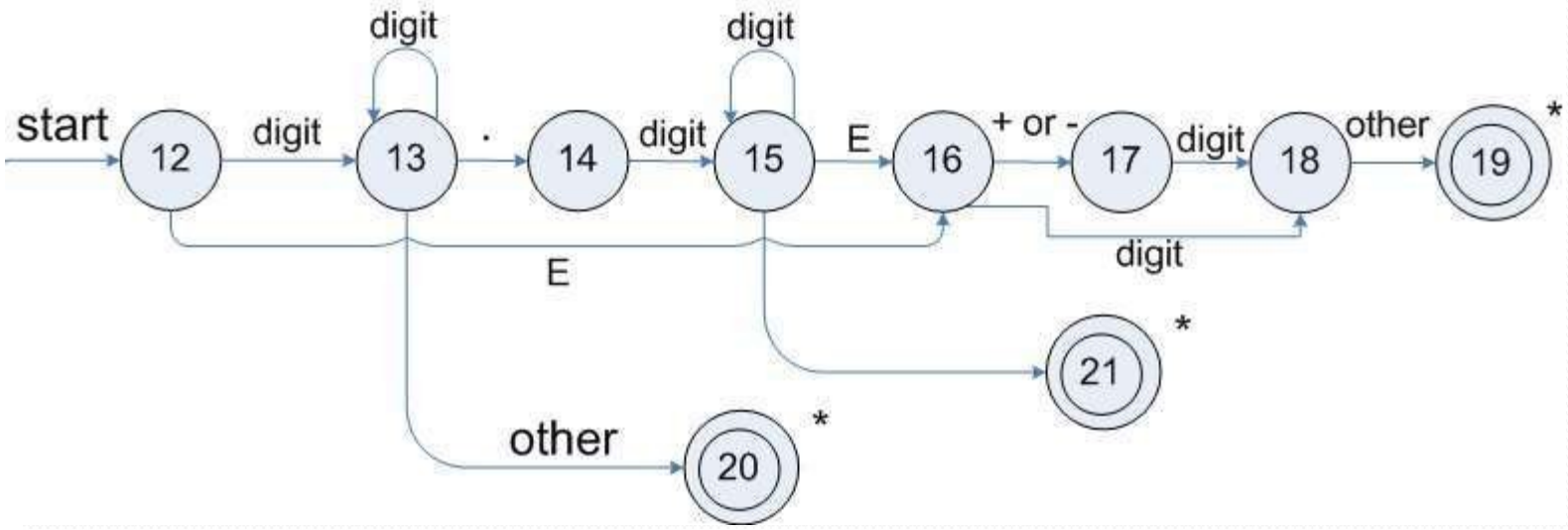
Transition diagrams(cont.)

- Transition diagram for reserved words and identifiers



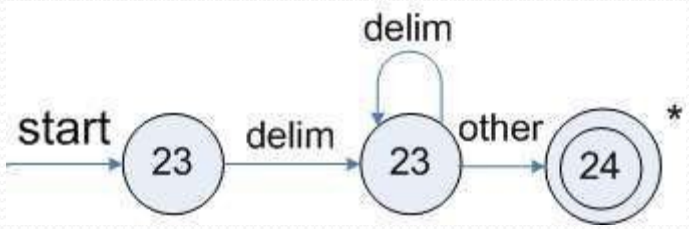
Transition diagrams(cont.)

- Transition diagram for unsigned numbers



Transition diagrams(cont.)

- Transition diagram for whitespace

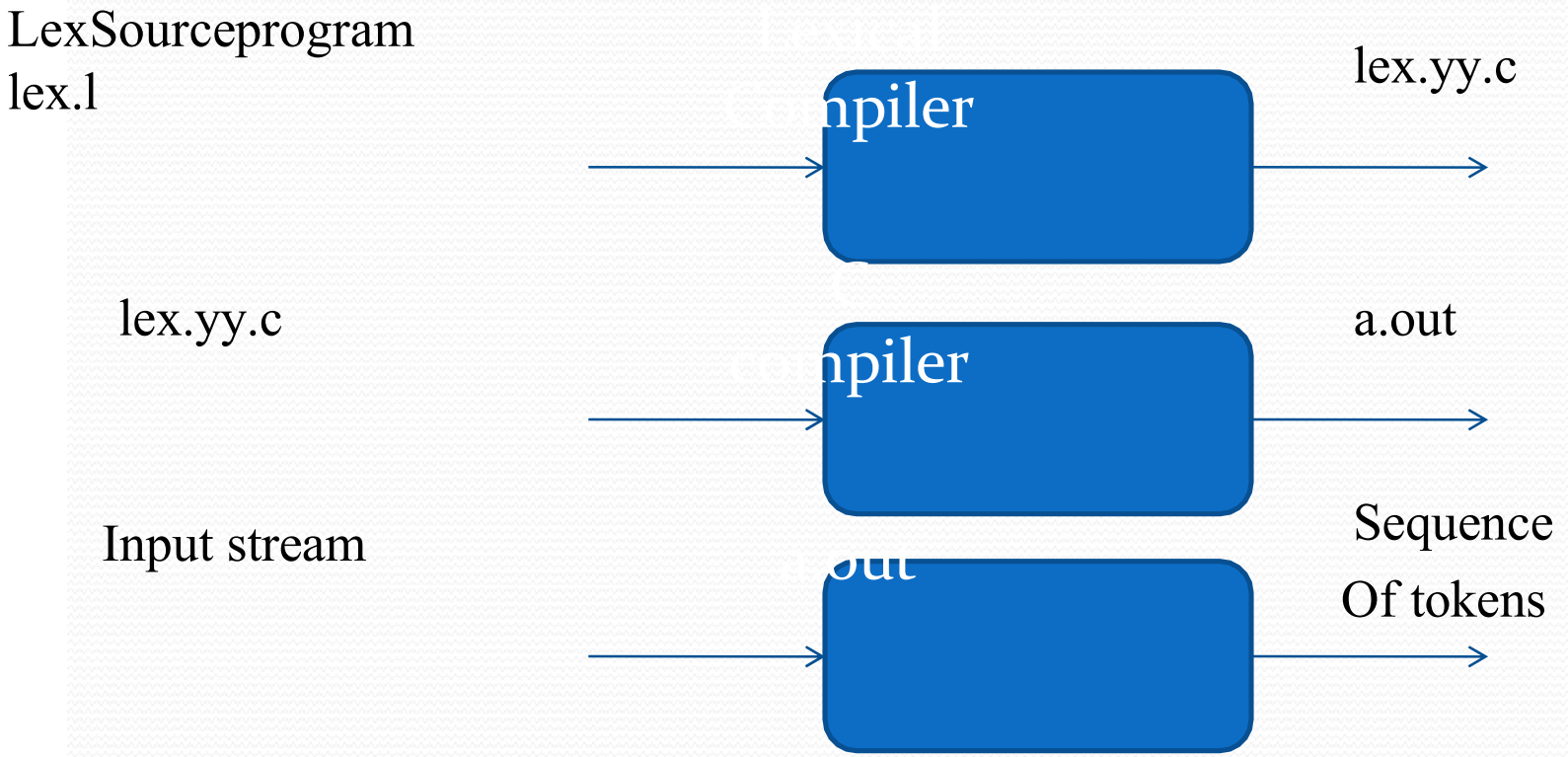


Architecture of a transition-diagram-based lexical analyzer


```
TOKENgetRelop()
{
    TOKENretToken=new(RELOP)
    while(1){                /*repeatcharacterprocessinguntila
                                        returnorfailureoccurs*/
    switch(state){
            caseo:c=nextchar();
                        if(c==‘<‘)state=1;
                        else if (c==‘=‘)state=5;
                        else if (c==‘>’)state=6;
                        else fail();/*lexemeisnotarelop*/ break;
            case1:...
            ...
            case8:retract();
                        retToken.attribute=GT;
                        return(retToken);
```

}

Lexical Analyzer Generator-Lex



Structure of Lex programs

declarations
%%
Translation rules
%%
Auxiliary
functions

Pattern {Action}

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       [o-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)install ID();return(ID);}
{number}    {yylval=(int)install Num();return(NUMBER);}
...
```

```
Int install ID(){/*funtiont o
    lexeme, whose first character is
    pointed to by yy text, and whose
    length is yy leng, into the symbol
    tableand return apointer thereto
    */
}

Int install Num() { /* similar to install
    ID, but puts numerical constants
    into a separatetable*/
}
```

Finite Automata

- Regular expressions=specification
- Finite automata=implementation

- Afinite automata consistsof
  - An input alphabet Σ
  - A set of states S
  - A starts taten
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

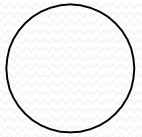
- Is read

Instate s_1 oninput "a" gotostate s_2

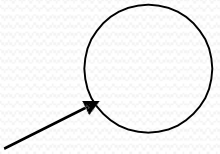
- If end of input
  - If in accepting state=>accept,othewise=>reject
- If no transition possible=>reject

Finite Automata State Graphs

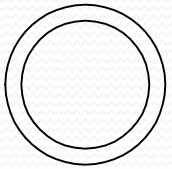
- A state



- The start state

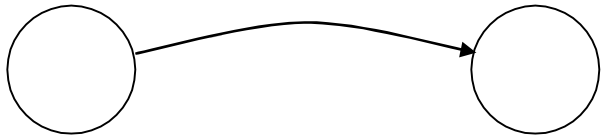


- An accepting state



a

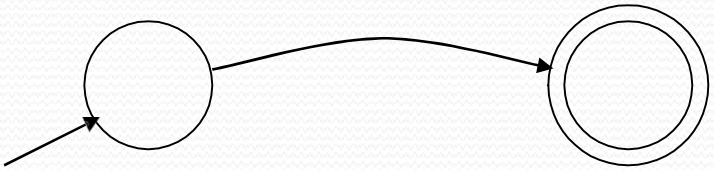
- A transition



A Simple Example

- A finite automaton that accepts only“1”

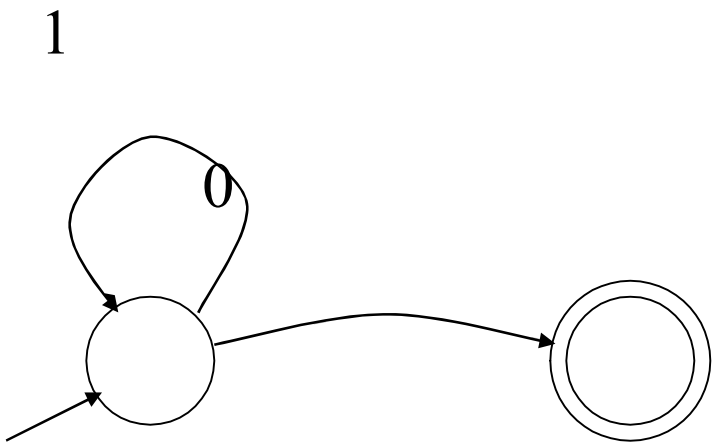
1



- 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

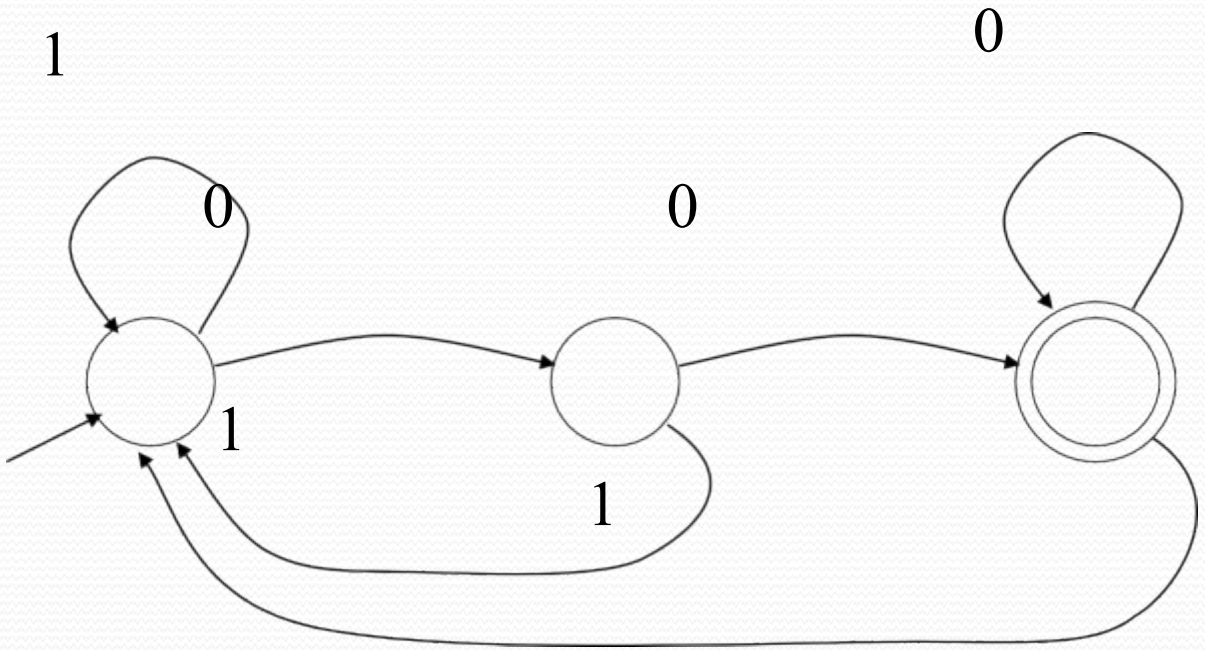
- A finite automaton accepting any number ofi's Followed by a singleo
- Alphabet:{0,1}



- Checkthat"1110"isacceptedbut"110..."is not

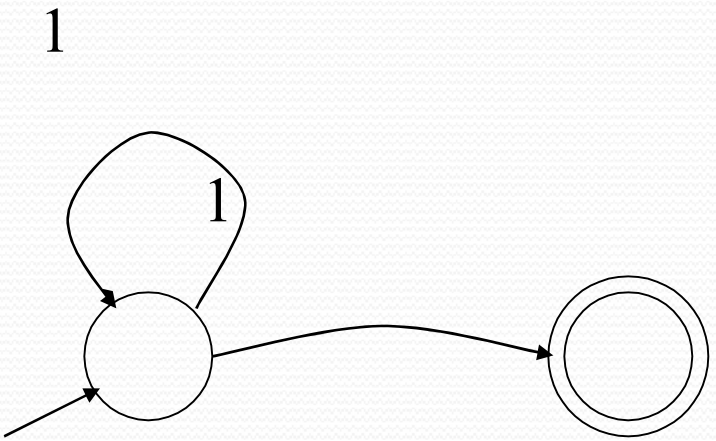
And Another Example

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



And Another Example

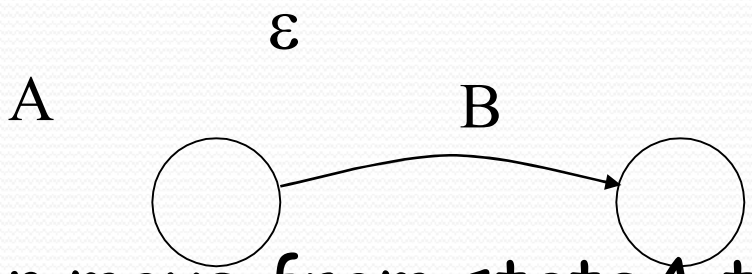
- Alphabet still{0,1 }



- The operation of the automaton is not completely defined by the input
  - Oninput“11”theautomatoncouldbeineitherstate

Epsilon Moves

- Another kind of transition:ε-moves



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

Deterministic and Nondeterministic Automata

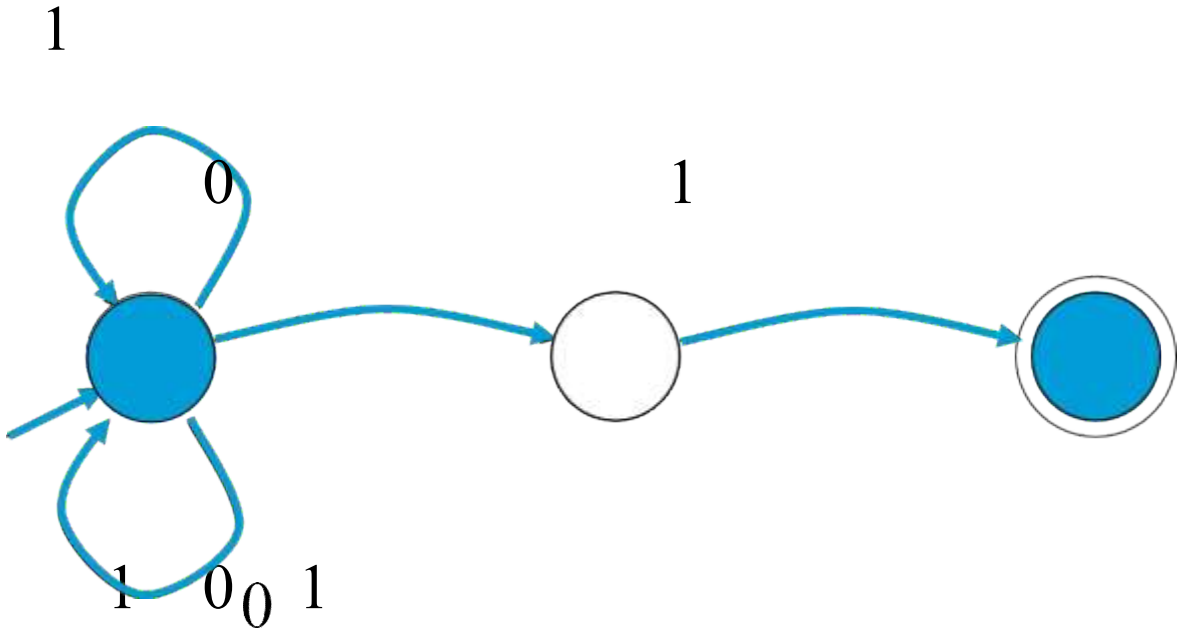
- Deterministic Finite Automata(DFA)
  - One transition per input per state
  - Noε-moves
- Non deterministic Finite Automata(NFA)
  - Can have multiple transitions for one input in a given state
  - Can haveε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to makeε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input:

- Rule:NFA accepts if it can get in a finalst

NFAvs.DFA(1)

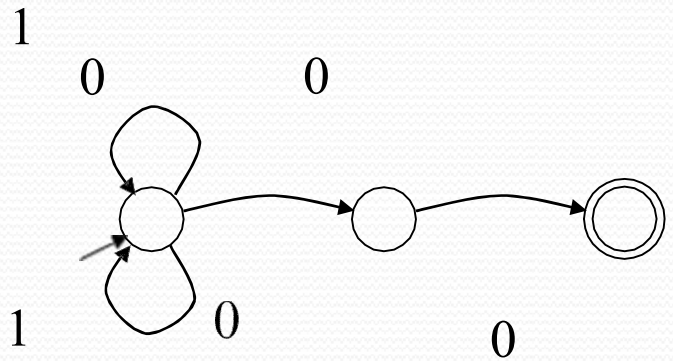
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFA sare easier to implement
  - There are no choices to consider

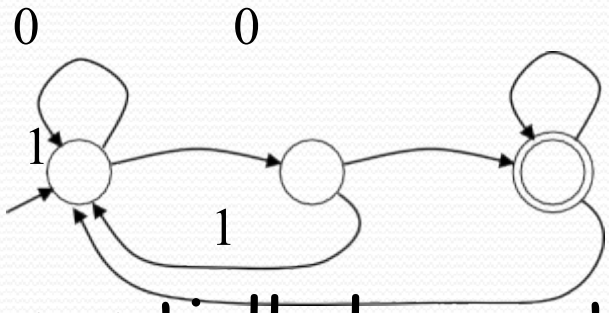
NFAvs.DFA(2)

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

NFA



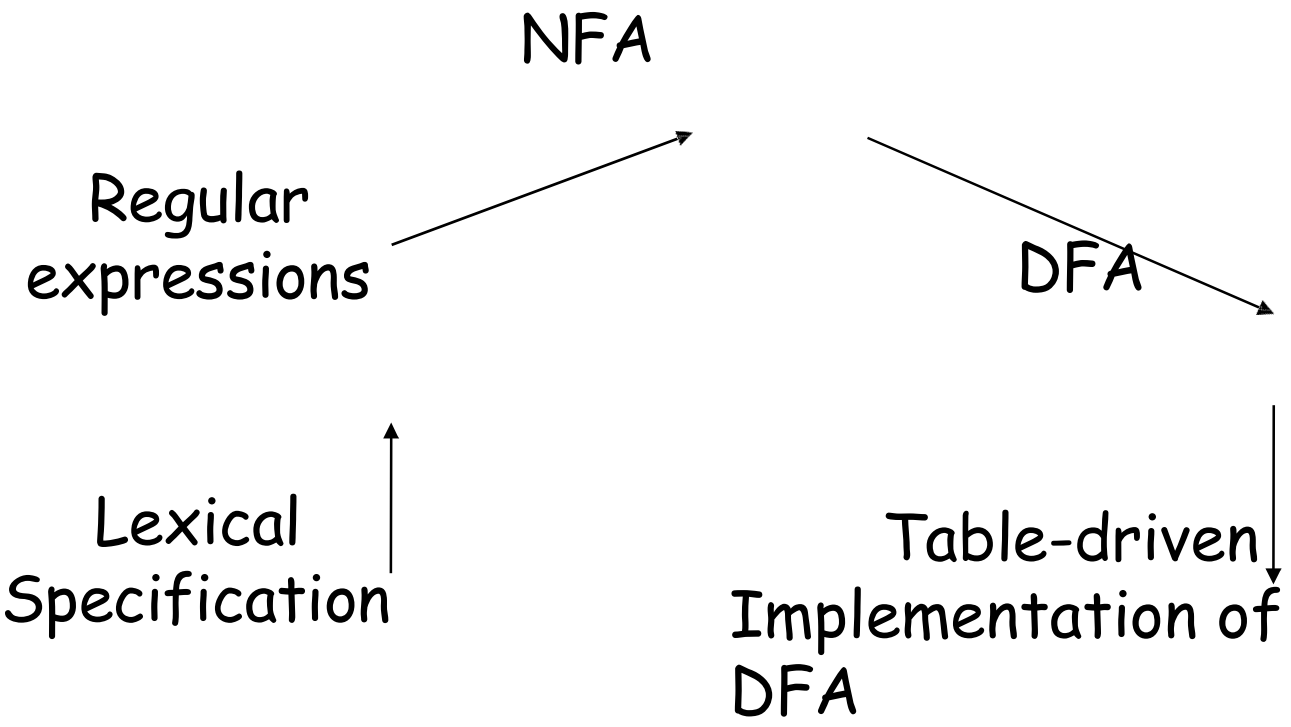
DFA



- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

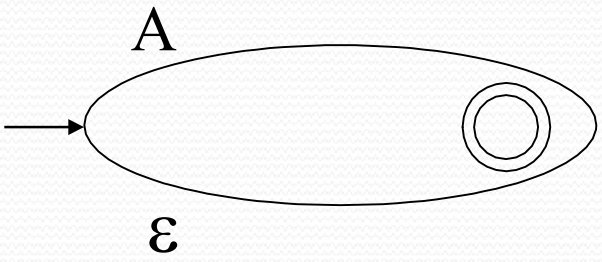
- High-level sketch



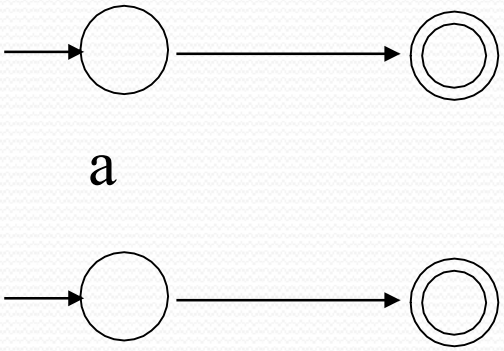
Regular Expressions to NFA(1)

- For each kind of rexp,defineanNFA
  - Notation :NFA for rexpA

- For ε

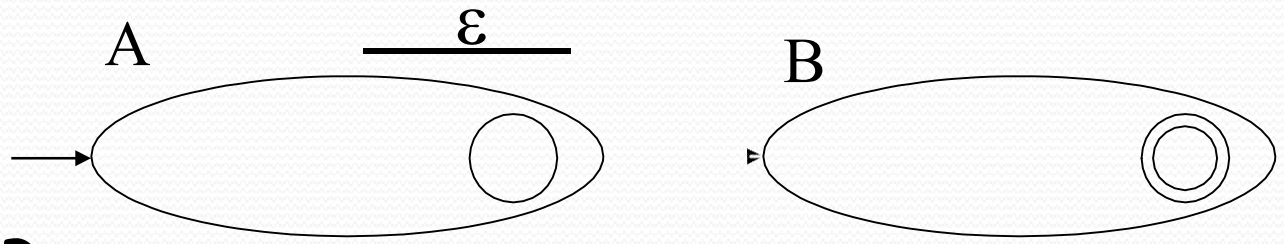


- For input a

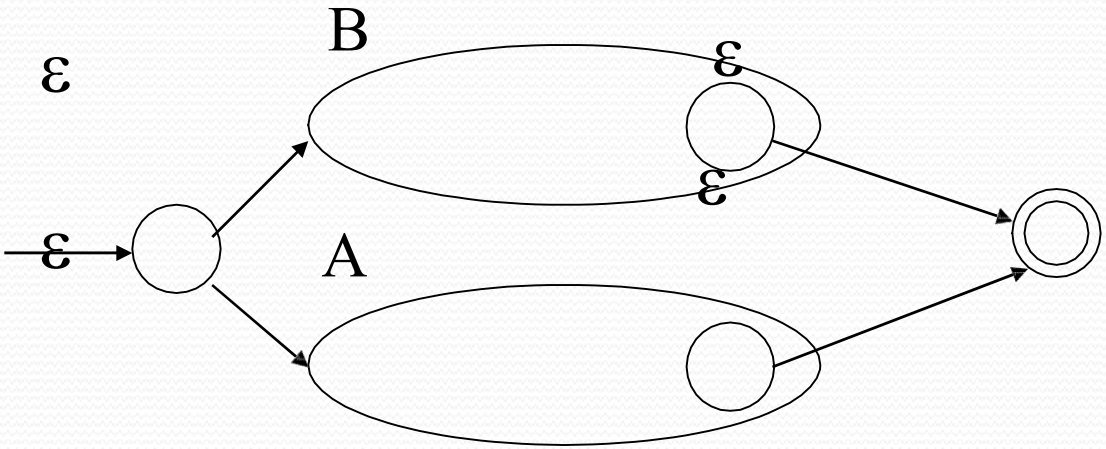


Regular Expressions to NFA(2,

- For AB

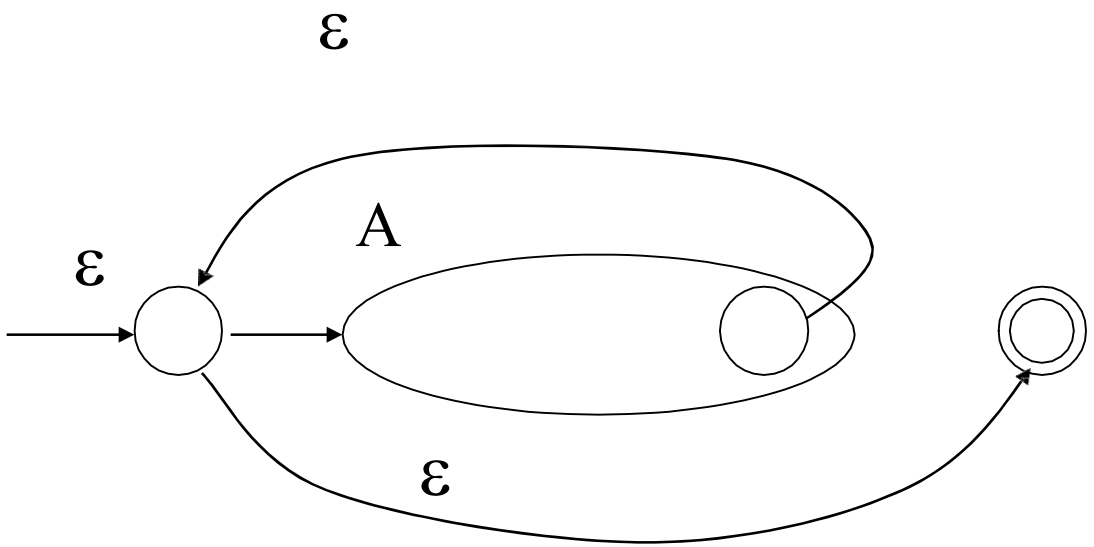


- For A|B



Regular Expressions to NFA(3,

- For A*



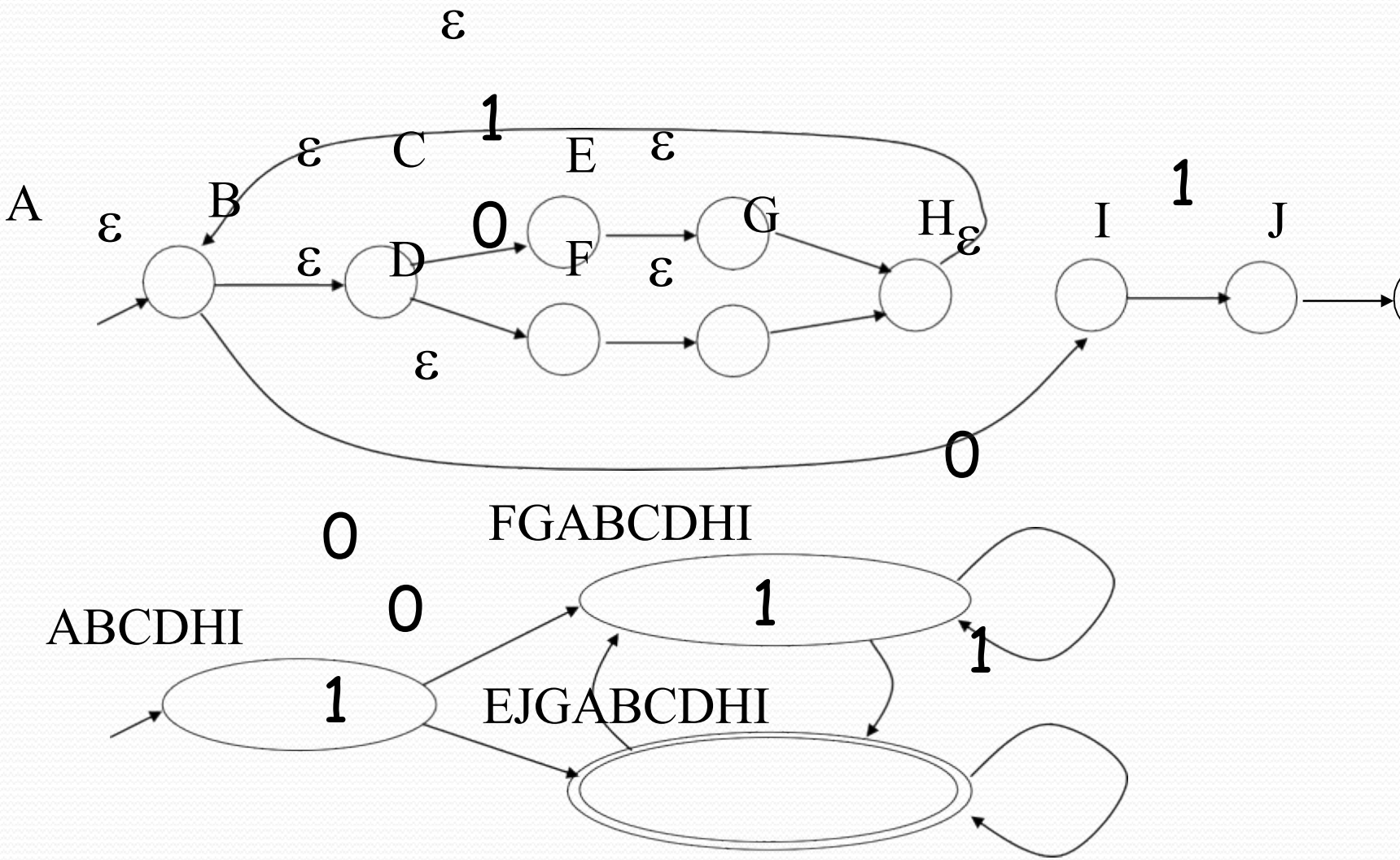
NFA to DFA.The Trick

- Simulate the NFA
- Each state of resulting DFA

  =anon-empty subset of states of theNFA
- Start state

  =the set of NFA states reachable throughε-movesfrom NFA start state
- Add a transition$S \rightarrow^{a} S'$toDFAiff
  - S'is the set of NFA states reachable from the states in S After seeing the inputa
    - consideringε-moves as well

NFA->DFA Example



x=*p

```
LDR1,p                    //R1=p
LDR2,0(R1)                //R2=
   contents(0+contents(R1))
STx,R2                    //x=R2
```

conditional-jump three-address instruction

If x<y goto L

LD R1,x

LD R2,y

SUBR1,R1,R2 BLTZ R1, M

//R1=x

//R2=y

//R1= R1-R2

//ifR1<ojumptoM

costs associated with the addressing modes

- LD R0,R1
- LD R0,M

- LD R1,*100(R2)

cost=1

cost=2

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

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 po
ST    *SP, #here + 16                    // save return addres
BR    callee.codeArea                    // return to caller
```

Branch tocalledprocedure

Return to caller

in caller:

In

Callee:

BR*0(SP)
SUB SP, SP, #calle
r.recordsize

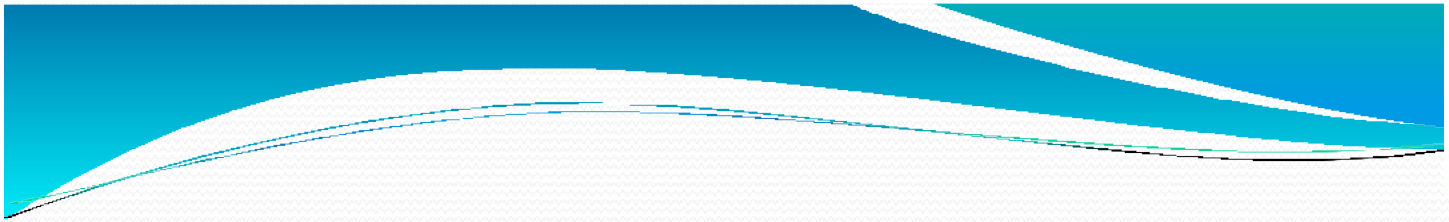
Target code for stack allocation

```
        // 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:  ACTION12
180:  HALT
      ...
                                    // code for p
200:  ACTION3
220:  BR *0(SP)                     // return
      ...
                                    // code for q
300:  ACTION4                       // contains a conditional jum
336:  BR 200                        // call p
344:  SUB SP, SP, #qsize                                  turn address
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 flowgraph

rules for finding leaders

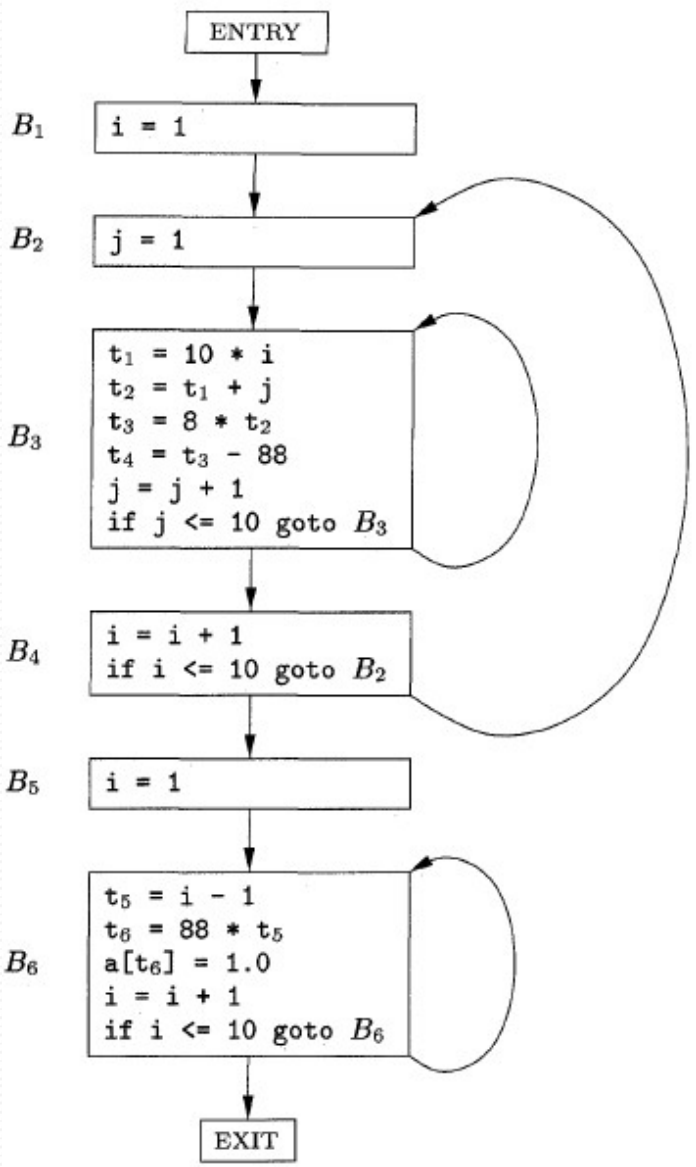
- The first three-address instruction in the intermediate code is a leader.
- Any instruction that is the target of a conditional or unconditional jump is a leader.
- Any instruction that immediately follows a conditional o run conditional jump is a leader.

Intermediatecodetoseta10*10matri 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)
```

Flowgraph based on BasicBlocks



liveness andnext-use information

- We wish to determine for each three address statement x=y+z what the next uses of x,yandzare.
- Algorithm:
  - Attach to statement the information currently found in the symbol table regarding then extuse and livenessof x, y,and z.
  - In the symbol table,set x to"not live"and"non extuse.“
  - In the symbol table,set y and zto"live"and the next uses of yand 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 statements 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 bys.
- 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

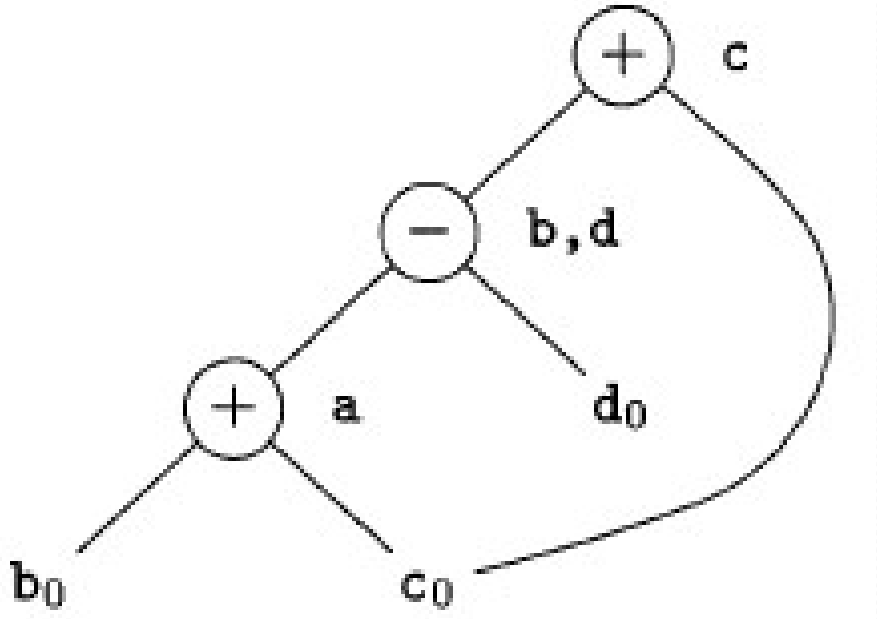
the block.

Code improving transformations

- We can eliminate *local common subexpressions*,that is,instructions that compute a value that has already been computed.
- We can eliminate *deadcode*,thatis,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 timea temporary value needs to be preserved in a register.
- We can apply algebraic law store or deroper and s of three-address instructions,and sometimes thereby simplify t he computation.

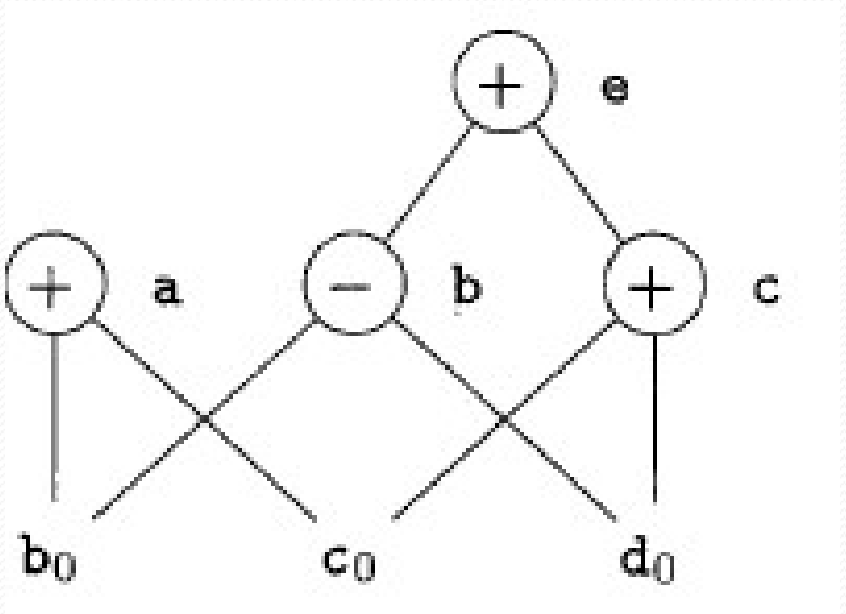
DAG for basicblock

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



DAG for basicblock

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

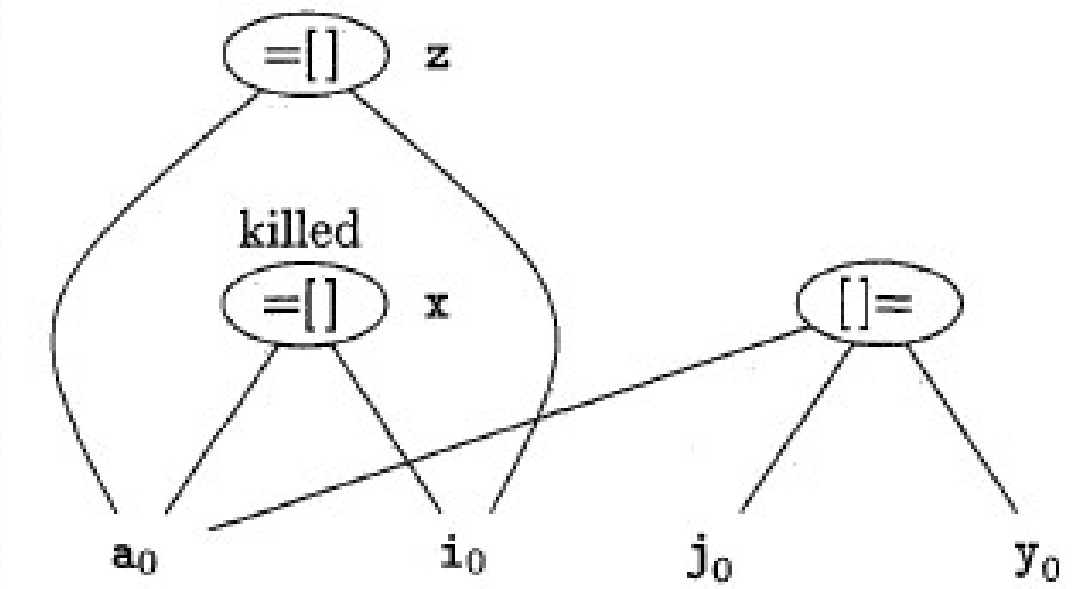


array accesses in a DAG

- An assignment from an array,like x=a[i],isrepresentedby creating anodewithoperator=[]andtwochildrenrepresenting theinitialvalueof thearray,aointhiscase,andtheindexi. Variablex becomesalabelof thisnewnode.
- Anassignmenttoanarray,likea[j]=y,isrepresentedbyanew nodewithoperator[]=andthreechildrenrepresentingao,jand y.Thereisnovariablelabelingthisnode.Whatisdifferentisthat thecreationof thisnode*kills*allcurrentlyconstructednodes whosevaluedependsonao.**A**nodethathasbeenkilledcannot receiveanymorelabels;thatis, itcannotbecomeacommon subexpression.

DAG for a sequence of array assignments

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



Rules for reconstructing thebasicblock from a DAG

- TheorderofinstructionsmustrespecttheorderofnodesintheDAG. Thatis,wecannotcomputeanode'svalueuntilwehavecomputed a value foreach of itschildren.
- Assignmentstoanarraymustfollowallpreviousassignmentsto,or evaluationsfrom,thesamearray,accordingtotheorderofthese instructions inthe original basic block.
- Evaluationsofarrayelementsmustfollowanyprevious(accordingto theoriginalblock)assignmentstothesamearray.Theonly permutationallowedisthattwoevaluationsfromthesamearraymay bedoneineitherorder,aslongasneithercrossesoveranassignmentto that array.
- Anyuseofavariablemustfollowallprevious(accordingtotheoriginal block)procedurecallsorindirectassignmentsthroughapointer.
- Anyprocedurecallorindirectassignmentthroughapointermust followallprevious(accordingtotheoriginalblock)evaluationsofany variable.

principal uses of registers

- Inmostmachinearchitectures, someorallof the operandsofanoperationmustbeinregistersinorder to perform the operation.
- Registersmakegoodtemporaries-placestoholdthe resultofasubexpressionwhilealargerexpressionis beingevaluated,ormoregenerally,aplacetoholda variablethatisusedonlywithinasinglebasicblock.
- Registersareoftenusedtohelpwithrun-timestorage management,forexample,tomanagetherun-time stack,includingthemaintenanceofstackpointersand possiblythetopelementsof thestackitself.

Descriptors for datastructure

- Foreachavailableregister,aregisterdescriptorkeepstrackof the variablenameswhosecurrentvalueisinthatregister.Sincewe shalluseonlythoseregistersthatareavailableforlocaluse withinabasicblock,weassumethatinitially,allregister descriptorsareempty.Asthecodegenerationprogresses,each registerwill hold thevalueof zeroormorenames.
- Foreachprogramvariable,anaddressdescriptorkeepstrackof thelocationorlocationswherethecurrentvalueof thatvariable canbefound.Thelocationmightbearegister,amemory address,astacklocation,orsomesetof morethanoneof these. Theinformationcanbestoredinthesymbol-tableentryforthat variable name.

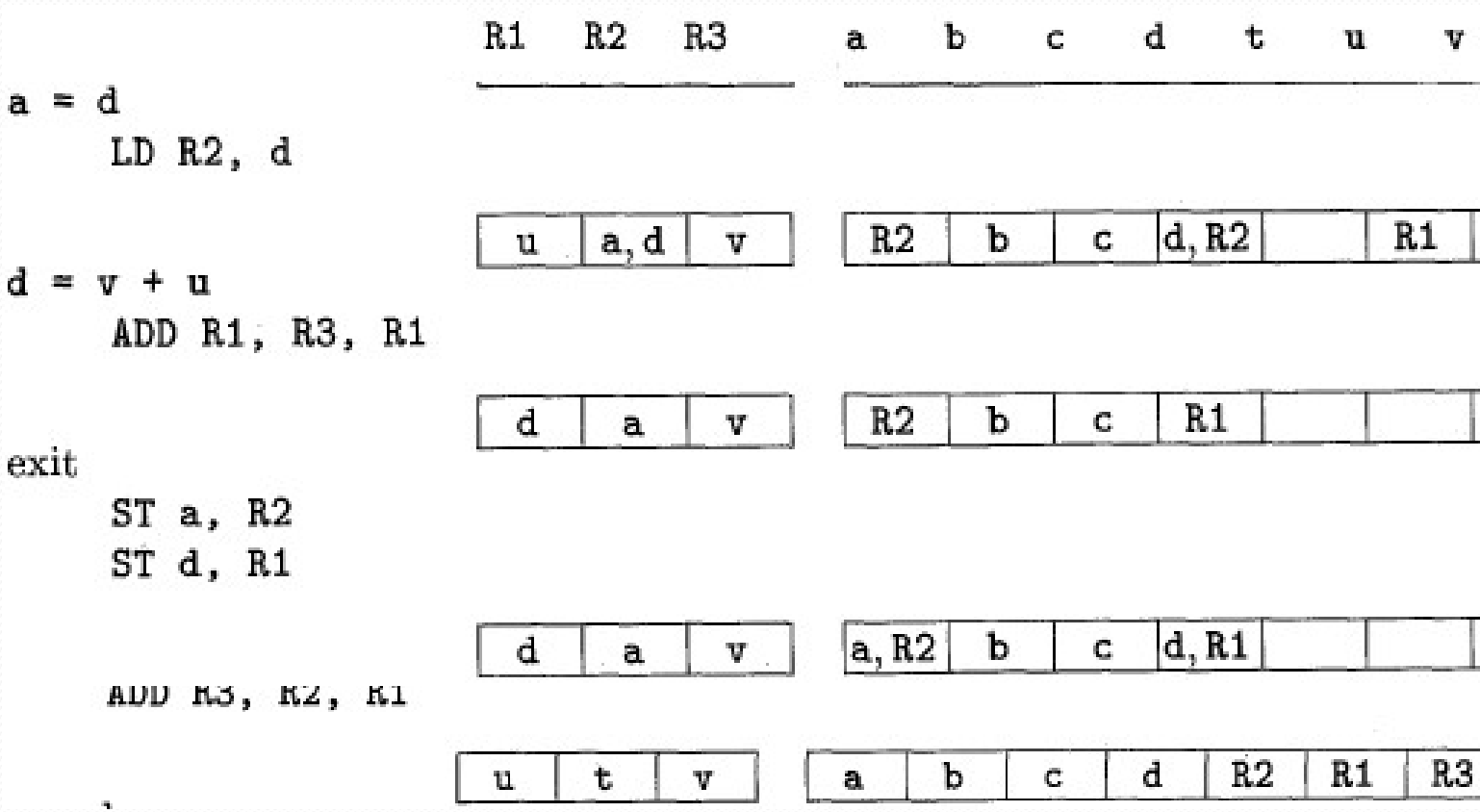
Machine Instructions for Operations

- UsegetReg(x=y+z)toselectregistersforx,y,andz. Call these R_x, $R_y$and R_z.
- If$y$isnotinR_y(accordingtotheregisterdescriptorfor R_y),thenissueaninstructionLD R_y,y',wherey'isone ofthememorylocationsfory(according tothe address descriptor fory).
- Similarly,if$z$isnotinR_z,issueandinstructionLDR_z, z',wherez' isalocationfor *x* .
- IssuetheinstructionADDR_x,R_y,R_z.

Rules for updating the register and address descriptors

- For the instruction LDR, 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

- Ifyiscurrentlyinaregister,pickaregisteralready containingyasR_y.Donotissueamachineinstruction toloadthisregister,asnoneisneeded.
- Ifyisnotinaregister,butthereisaregisterthatis currentlyempty,pickonesuchregisterasR_y.
- Thedifficultcaseoccurswhenyisnotinaregister,and thereisnoregisterthatiscurrentlyempty.Weneedto pickoneoftheallowableregistersanyway,andwe need tomakeitsafetoreuse.

Possibilities for value of R

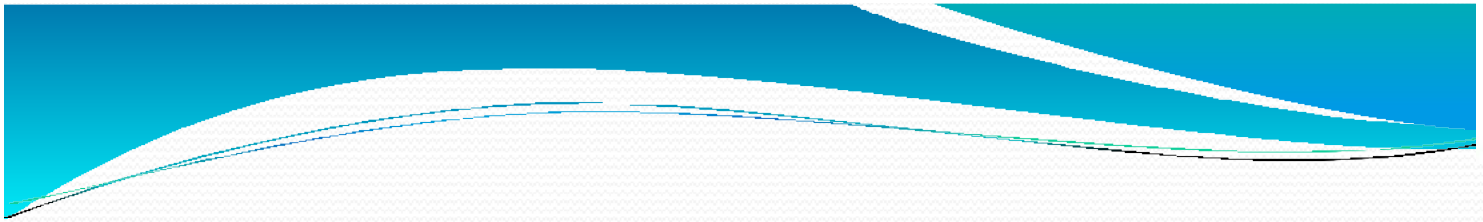
- IftheaddressdescriptorforvsaysthatvissomewherebesidesR, then we are OK.
- Ifvisx,thevaluebeingcomputedbyinstructionI,andxisnot alsooneof theotheroperandsof instructionI (zinthis example),thenweareOK.Thereasonisthatinthiscase,we knowthisvalueof xisneveragaingoingtobeused,soweare free to ignore it.
- Otherwise,ifvisnotused later(thatis,aftertheinstructionI, therearenofurtherusesofv,andifvisliveonexitfromthe block,thenvisrecomputedwithintheblock),thenweareOK.
- IfwearenotOKbyoneofthefirsttwocases,thenweneedto generatethestoreinstruction**ST**v,*R*toplaceacopyofvinits ownmemorylocation.Thisoperationiscalledaspill.

Selection of the register Rx

1. Sinceanewvalueofxisbeingcomputed,aregister thatholdsonlyxisalwaysanacceptablechoicefor Rx.
2. IfyisnotusedafterinstructionI,andRyholdsonlyy afterbeing loaded, RycanalsobeusedasRx.A similaroptionholdsregarding zand Rx.

Possibilities for value of R

- IftheaddressdescriptorforvsaysthatvissomewherebesidesR, then we are OK.
- Ifvisx,thevaluebeingcomputedbyinstructionI,andxisnot alsooneof theotheroperandsof instructionI (zinthis example),thenweareOK.Thereasonisthatinthiscase,we knowthisvalueof xisneveragaingoingtobeused,soweare free to ignore it.
- Otherwise,ifvisnotused later(thatis,aftertheinstructionI, therearenofurtherusesofv,andifvisliveonexitfromthe block,thenvisrecomputedwithintheblock),thenweareOK.
- IfwearenotOKbyoneofthefirsttwocases,thenweneedto generatethestoreinstruction**ST**v,*R*toplaceacopyofvinits ownmemorylocation.Thisoperationiscalledaspill.



Characteristic of peephole optimizations

- Redundant-instructionelimination
- Flow-of-controloptimizations
- Algebraicsimplifications
- Useofmachineidioms

Redundant-instruction elimination

- LDa,Ro

  ST Ro, a

- ifdebug==1goto**L1**

  gotoL2

  LI:printdebugginginformation

  L2:

Flow-of-control optimizations

gotoL1

…

Ll:gotoL2

Canbereplacedby:

goto L2

…

Ll:gotoL2

ifa<bgotoL1

…

Ll:gotoL2

Canbereplacedby: if

a<b goto L2

…

Ll:gotoL2

Global register allocation

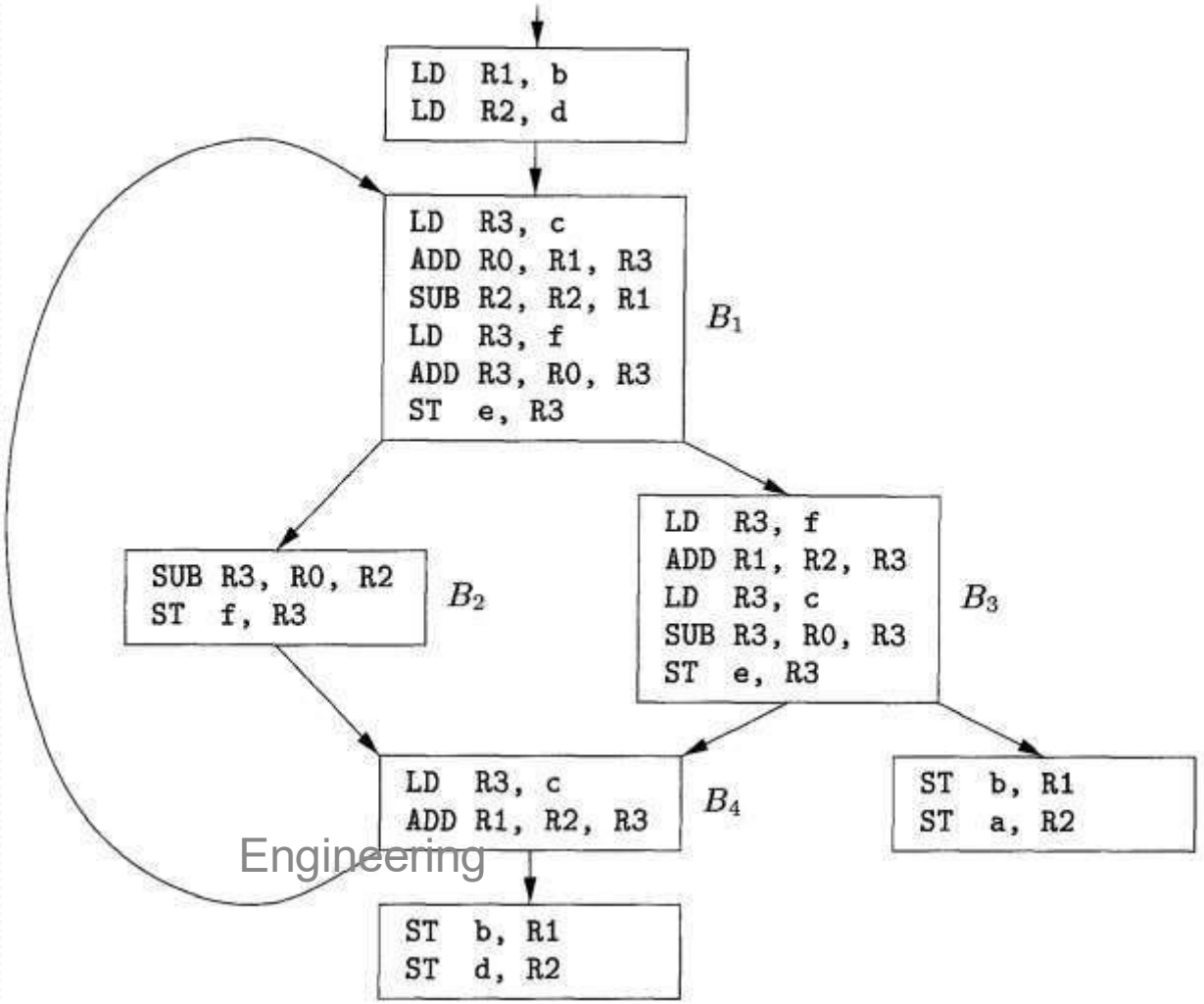
- Previouslyexplainedalgorithmdoeslocal(blockbased) registerallocation
- Thisresultedthatalllivevariablesbestoredattheendof block
- Tosavesomeofthesestoresandtheircorrespondingloads, wemightarrangetoassignregisterstofrequentlyused variablesandkeeptheseregistersconsistentacrossblock boundaries (globally)
- Someoptionsare:
  - Keepvaluesofvariablesusedinloopsinsideregisters
  - Usegraphcoloringapproachformoregloballyallocation

Usage counts

- Fortheloopswecanapproximatethesavingby register allocation as:
  - Sumoverallblocks(B)inaloop(L)
  - Foreachusesofxbeforeanydefinitionintheblockwe addoneunitof saving
  - IfxisliveonexitfromBandisassignedavalueinB, thenweass 2 unitsof saving

Flowgraph of an innerloop

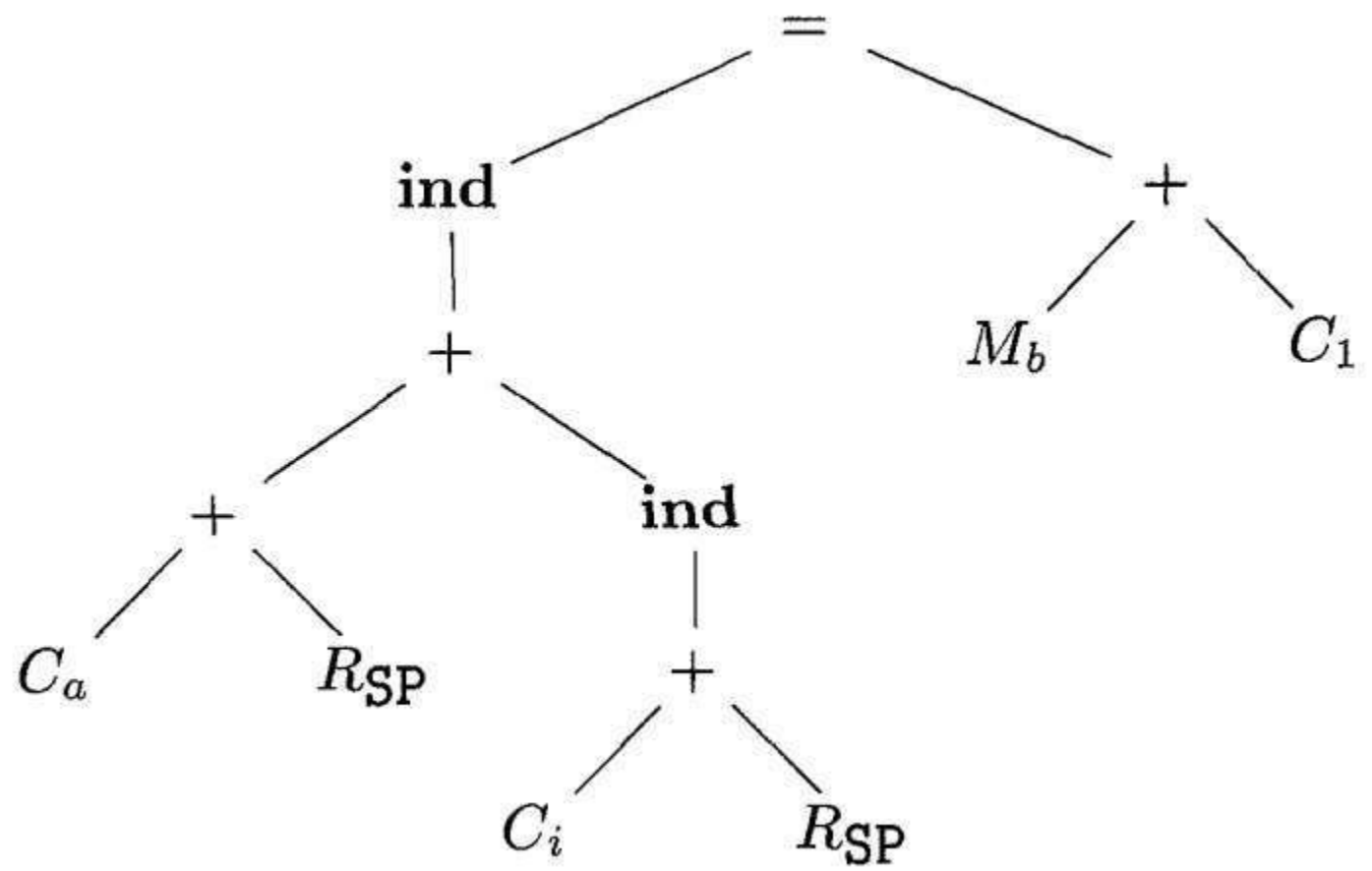
Code sequence using global register assignment



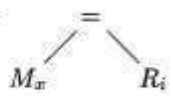
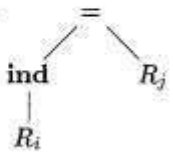
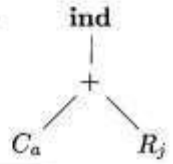
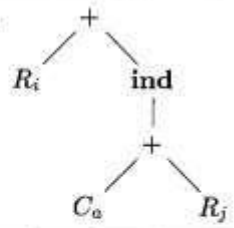
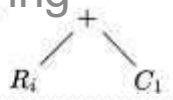
Register allocation by Graph coloring

- Twopassesareused
  - Target-machineinstructionsareselectedasthough therearean infinitenumberof symbolicregisters
  - Assignphysicalregisterstosymbolicones
    - Createaregister-interferencegraph
    - Nodesaresymbolicregistersandedgesconnectstwonodesif oneisliveatapointwheretheotherisdefined.
    - Forexampleinthepreviousexampleanedgeconnectsaandd inthegraph
    - Useagraphcoloringalgorithmtoassignregisters.

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



Tree-rewriting rules

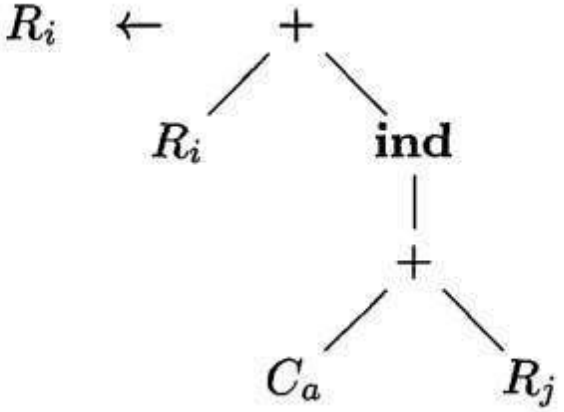
1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ 	{ ST x, Ri }
4)	$M \leftarrow$ 	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ 	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

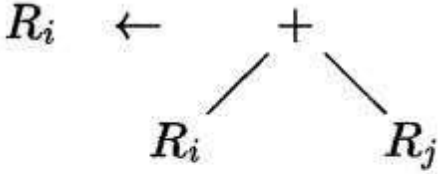
Syntax-directed translation scheme

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

$R_i \leftarrow C_a$



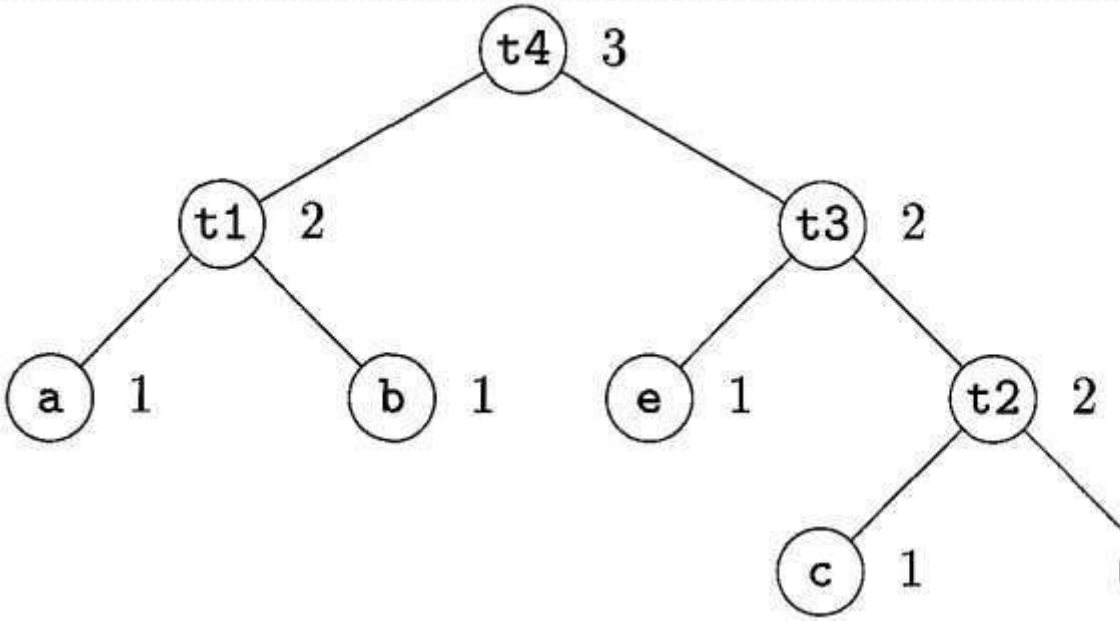


Ershov Numbers

- Labelanyleafı.
- Thelabelofaninteriornodewithonechildisthelabel of its child.
- Thelabelofaninteriornodewithtwochildrenis
  - Thelargerofthelabelsof itschildren,ifthoselabelsare different.
  - Oneplusthelabelofitschildrenifthelabelsarethe 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

- Togeneratemachinecodeforaninteriornodewithlabel*k*andtwo childrenwithequal labels(which must be*k* - l)dothefollowing:
  - Recursivelygeneratecodefortherightchild,usingbase*b*+1.Theresultof therightchildappearsinregisterR_{b+k}.
  - Recursivelygeneratecodefortheleftchild,using baseb;theresultappears in R_{b+k-1}.
  - Generatetheinstruction$OPR_{b+k}, R_{b+k-1}, R_{b+k}$,whereOPistheappropriate operationfortheinteriornodeinquestion.
- Supposewehaveaninteriornodewithlabel *k*andchildrenwithunequal labels.Thenoneofthechildren,whichwe'llcallthe"big"child,haslabel*k* ,andtheotherchild,the"little"child,hassomelabelm<*k*.Dothe followingtogeneratecodeforthisinteriornode,usingbase*b:*
  - Recursivelygeneratecodeforthebigchild,using baseb;theresultappears in register R_{b+k-l}.
  - Recursivelygeneratecodeforthesmallchild,usingbase*b*;theresult appearsinregisterR_{b+m-l}.Notethatsincem<k,neitherR_{b+k-l}norany higher-numbered register is used.
  - Generatetheinstruction$OPR_{b+k-l}, R_{b+m-l}, R_{b+k-1}$ortheinstruction$OPR_{b+k-l}$, R_{b+k-l}, R_{b+m+l},depending onwhetherthebigchild istherightorleftchild, respectively.
- Foraleafrepresentingoperandx,ifthebaseis*b*generatetheinstruction 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

- NodeN hasatleastonechildwithlabelrorgreater.Pickthelarger child(oreitheriftheirlabelsarethesame)tobethe"big"childandlet theotherchild bethe "little" child.
- Recursivelygeneratecodeforthebigchild,usingbaseb=1.Theresult of thisevaluationwillappear inregisterR_r
- Generatethemachineinstruction$ST t_k, R_r$,where$t_k$isatemporary variableusedfortemporaryresultsusedtohelpevaluatenodeswith label k.
- Generatecodeforthelittlechildasfollows.Ifthelittlechildhaslabelr orgreater,pickbaseb=1.If thelabelof thelittlechildis*j*<r,thenpick b=r-*j*.Thenrecursivelyapplythisalgorithmtothelittlechild;theresult appears in R_r.
- Generatetheinstruction$LD R_r$-l,t_k.
- Ifthebigchildistherightchildof N,thengeneratetheinstruction$OP R_r, R_r, R_{r-1}$.Ifthebigchildistheleftchild,generate$OP R, R_r, R_{r-1}, R_r$.

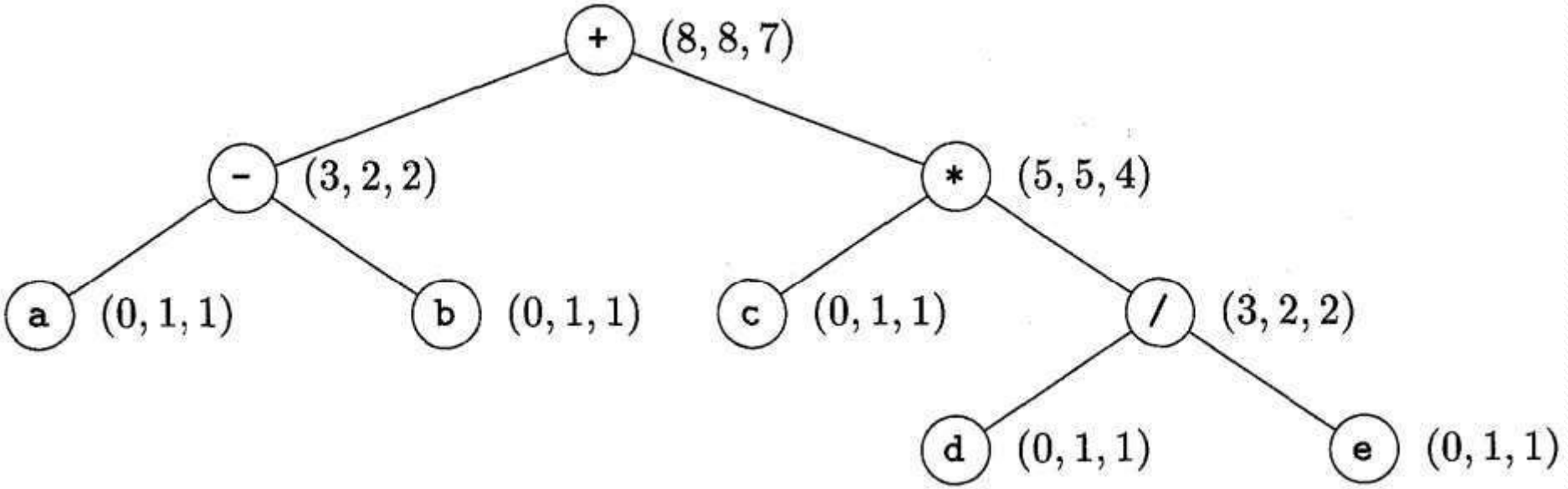
Optimal three-register code using only two registers

```
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

- Computebottom-upforeachnodenoftheexpressiontreeTan arrayCofcosts, inwhichtheithcomponentC[i]istheoptimal costofcomputingthesubtreeS rootedatnintoaregister, assumingiregistersareavailable forthecomputation,for
- TraverseT,usingthecostvectorstodeterminewhichsubtreesof Tmust becomputed intomemory.
- Traverseeachtreeusingthecostvectorsandassociated instructionstogeneratethefinaltargetcode.Thecodeforthe subtreescomputedintomemorylocationsisgeneratedfirst.

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



minimum cost of evaluating the root with two registers available

- Computetheleftsubtreewithtworegistersavailableinto registerRo,computetherightsubtreewithoneregister availableintoregisterR1,andusetheinstructionADDRo, Ro,R1tocomputetheroot.Thissequencehascost 2+5+1=8.
- Computetherightsubtreewithtworegistersavailableinto R l ,computetheleftsubtreewithoneregisteravailable intoRo,andusetheinstruction ADD Ro,Ro, R1.This sequence hascost 4+2+1=7.
- Computetherightsubtreeintomemorylocation M, computetheleftsubtreewithtworegistersavailableinto registerRO,anduseDDRo,Ro,M.This sequencehascost5+2+1=8.

UNIT-III

Syntax-Directed Translation

Outline

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

Introduction

- Wecanassociateinformationwitha language constructbyattachingattributestothegrammar symbols.
- Asyntaxdirecteddefinitionspecifiesthevaluesof attributesbyassociating semanticruleswith the grammar productions.

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

- Wemayalternativelyinsertthesemanticactionsinsidethegrammar

E}

Syntax Directed Definitions

- ASDDisacontextfreegrammarwithattributesand rules
- Attributesareassociatedwithgrammarsymbolsand rules with productions
- Attributesmaybeofmanykinds:numbers,types, table references, strings, etc.
- Synthesizedattributes
  - AsynthesizedattributeatnodeNisdefinedonlyin termsofattributevaluesofchildrenofNandatNit
- Inheritedattributes
  - AninheritedattributeatnodeNisdefinedonlyinterms ofattributevaluesatN'sparent,NitselfandN'ssiblings

Example of S-attributed SDD

Production

1) L->En
2) E->E1 +T
3) E->T
4) T->T1* F
5) T->F
6) F->(E)
7) F->digit

Semantic Rules L.val = E.val

E.val=E1.val+T.val

E.val=T.val

T.val=T1.val*F.val T.val = F.val

F.val=E.val

F.val=digit.lexval

Example of mixed attributes

Production

1) T->FT’

2) T’->*FT’$_1$

3) T’->ε
4) F->digit

SemanticRules

T’.inh=F.val
T.val=T’.syn
T'_1.inh=T’.inh*F.val
T’.syn = T'_1.syn
T’.syn = T’.inh
F.val=F.val=digit.lexval

Evaluation orders for SDD's

- Adependencygraphisusedtodeterminetheorderof computation of attributes
- Dependencygraph
  - Foreachparsetreenode,theparsetreehasanodefor eachattributeassociatedwiththatnode
  - Ifasemanticruledefinesthevalueofsynthesized attributeA.bintermsofthevalueofX.cthenthe dependencygraphhasanedgefromX.ctoA.b
  - Ifasemanticruledefinesthevalueofinheritedattribute B.cintermsofthevalueofX.athenthedependency graphhasanedge fromX.ctoB.c

Ordering the evaluation of attributes

- IfdependencygraphhasanedgefromMtoNthenM mustbeevaluatedbeforetheattributeof N
- ThustheonlyallowableordersofevaluationarethoseSequenceofnodesN1,N2,…,Nksuchthatifthereisan edgefrom Ni toNj then i<j
- Suchanorderingiscalledatopologicalsortofagraph
- Example!

S-Attributed definitions

- AnSDDisS-attributedifeveryattributeissynthesized
- Wecanhaveapost-ordertraversalofparse-treeto evaluateattributesin S-attributeddefinitions

```
postorder(N){
        for(eachchildCof N,fromtheleft)postorder(C);
        evaluatetheattributesassociatedwithnodeN;
}
```

- S-Attributeddefinitionscan be implementedduring bottom-upparsingwithouttheneedtoexplicitlycreate parse trees

L-Attributed definitions

- ASDDisL-Attributediftheedgesindependencygraph goesfromLefttoRightbutnotfromRighttoLeft.
- Moreprecisely,eachattributemustbeeither
  - Synthesized
  - Inherited,butifthereusaproductionA->X1X2…Xnandthere isaninheritedattributeXi.acomputed byaruleassociated withthisproduction,thentherulemayonlyuse:
    - InheritedattributesassociatedwiththeheadA
    - Eitherinheritedorsynthesizedattributesassociatedwiththe occurrencesofsymbolsX1,X2,…,Xi-1locatedtotheleftofXi
    - Inheritedorsynthesizedattributesassociatedwiththisoccurrence of Xiitself, butinsuchawaythatthereisnocycleinthegraph

Application of Syntax Directed Translation

- Typecheckingandintermediatecodegeneration (chapter6)
- Constructionofsyntaxtrees
  - Leafnodes: Leaf(op,val)
  - Interiornode:Node(op,c1,c2,…,ck)
- Example:

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

Syntax tree for L-attributed definition

Production	SemanticRules+
1) E->TE'	E.node=E'.syn E'.inh=T.node
2) E'->+TE1'	E1'.inh=newnode('+',E'.inh,T.node) E'.syn=E1'.syn
3) E'->-TE1'	E1'.inh=newnode('+',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=newLeaf(id,id.entry)
7) T->num	T.node=newLeaf(num,num.val)

Syntax directed translation schemes

- AnSDTisaContextFreegrammarwithprogramfragments embedded within production bodies
- Thoseprogramfragmentsarecalledsemanticactions
- Theycanappearatanypositionwithinproductionbody
- AnySDTcanbeimplementedbyfirstbuildingaparsetree and thenperformingtheactions inaleft-to-rightdepth first order
- TypicallySDT'sareimplementedduringparsingwithout building a parse tree

Postfix translation schemes

- SimplestSDDsarethosethatwecanparsethegrammar bottom-upandtheSDDiss-attributed
- ForsuchcaseswecanconstructSDTwhereeachactionis placedattheendoftheproductionandisexecutedalong withthereductionof thebodytotheheadof that production
- SDT'swithallactionsattherightendsoftheproduction bodies arecalled postfix SDT's

Example of postfix SDT

1) L->En {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 non terminal(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 then on-terminal at the left side of the production
- And put the value on the stack in replace of the right side of production

Example

```
L->En              {print(stack[top-1].val);
                     top=top-1;}
E->E1+T E          {stack[top-2].val=stack[top-2].val+stack.val;
                     top=top-2;}
-> 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

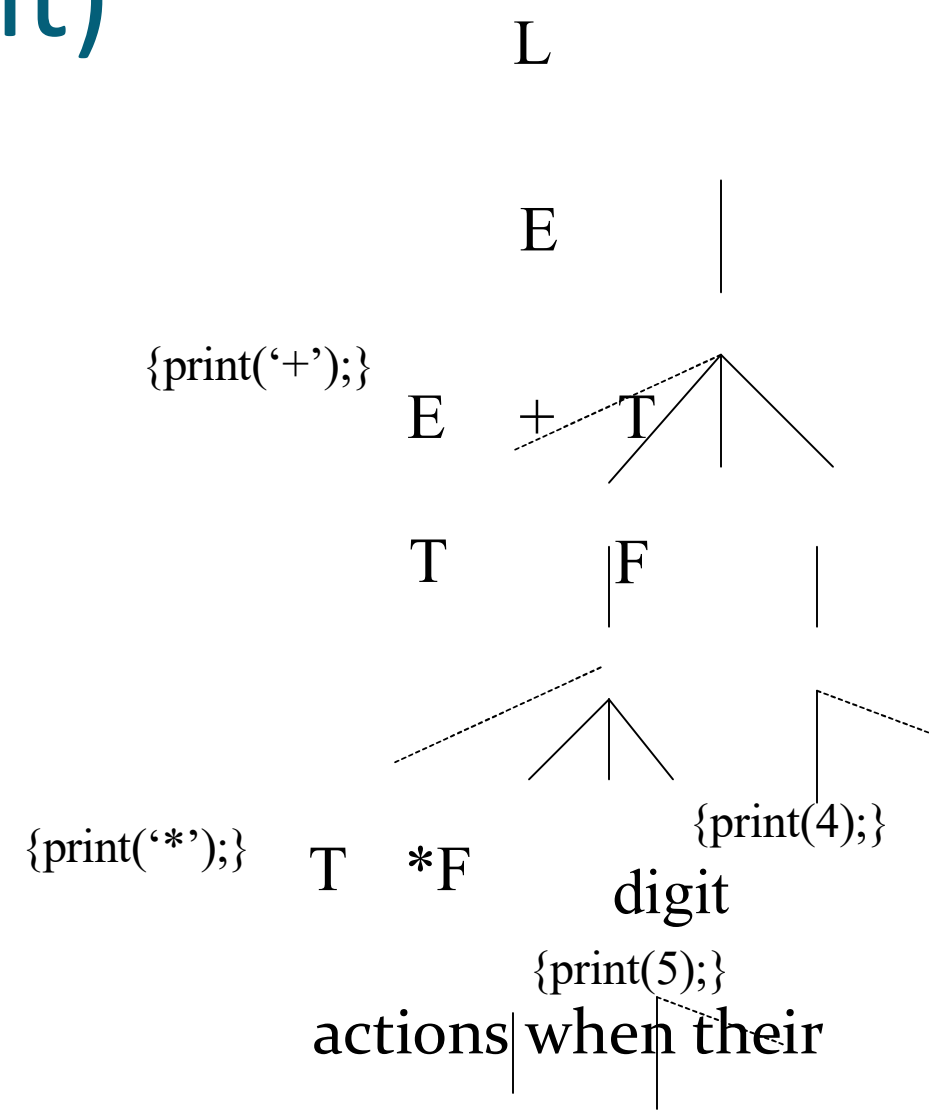
1) L->En
2) E->{print('+');}E1+ T
3) E->T
4) T->{print('*');}T1*F
5) T->F
6) F->(E)

parsing this SDT with a bottom-

7) F->digit{print(digit.lexval);}

SDT's with actions inside productions (cont)

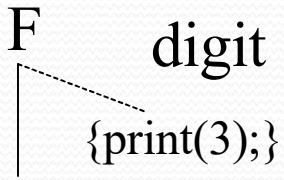
- Any SDT can be implemented as follows
  1. Ignore the actions and produce a parse tree
  2. Examine each interior Node N and add actions As new children at the correct position
  3. Perform a postorder traversal and execute actions when their



nodes

are
visited

digit



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 an on terminal A immediately before that occurrence of A. if several inherited attributes of A are dpendent on one another in anacyclic 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

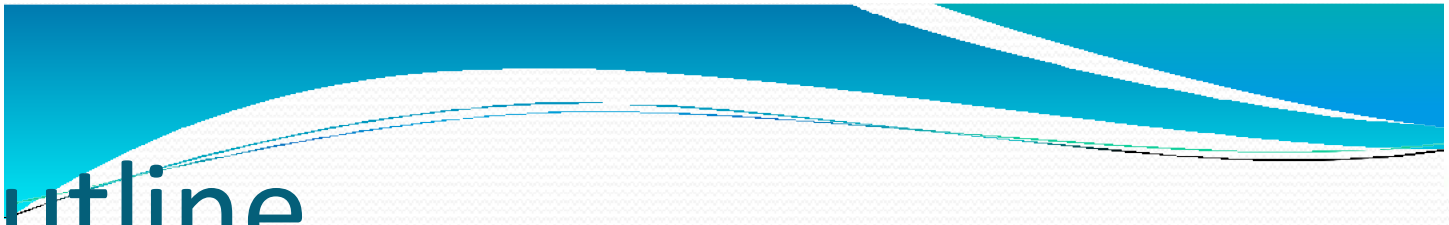
S->while(C) S1

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;}

UNIT-II

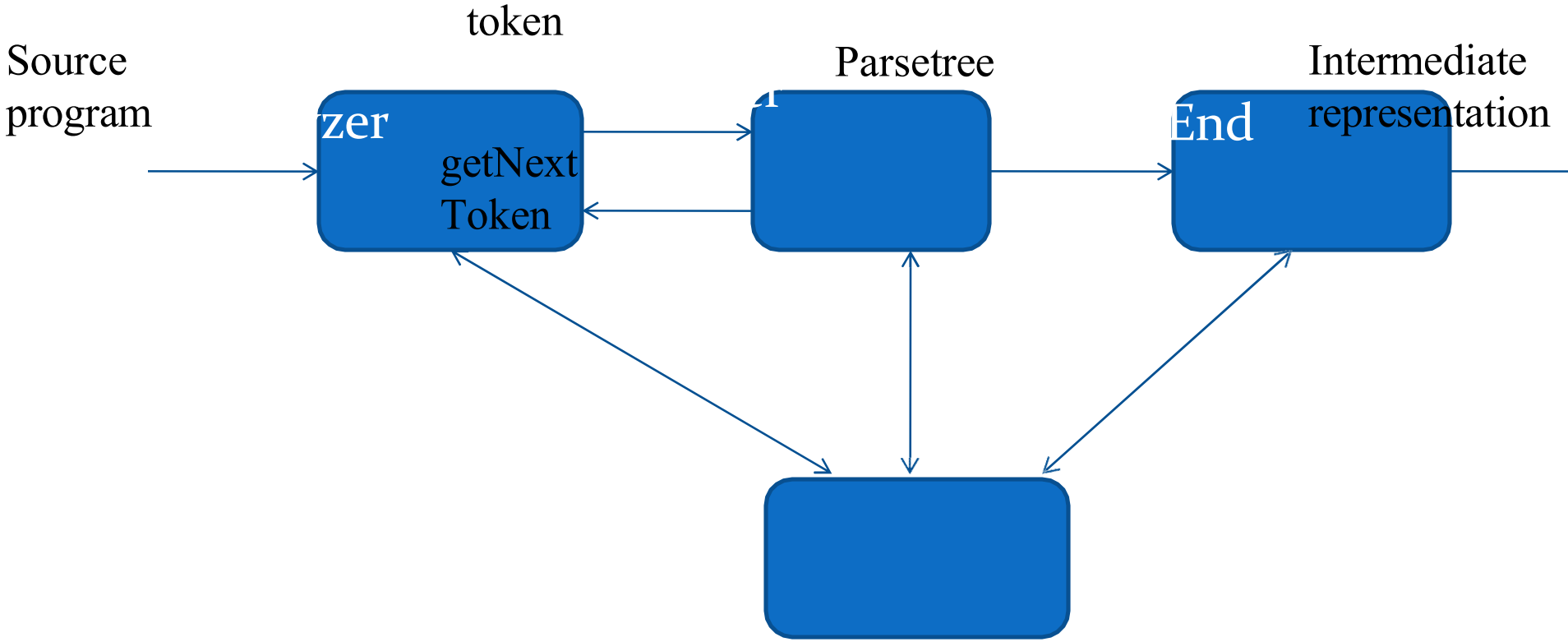
Syntax Analyzer



Outline

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

The role of parser



Uses ofgrammars

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 over head 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
- Non terminals
- Start symbol
- productions

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

Derivations

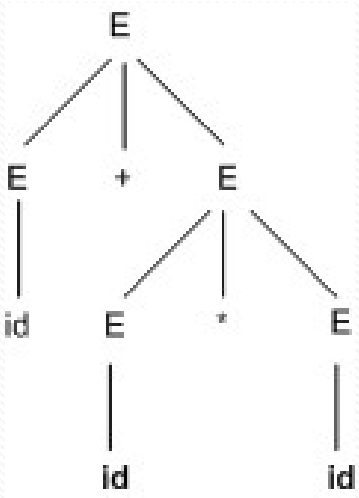
- Productions are treatedasre writing rules to generate a string
- Right most and left most derivations
  - E->E+E|E*E|-E|(E)| **id**
  - Derivations for–**(id+id)**
    - E =>-E=> -(E)=>-(E+E)=>-(**id**+E)=>-(**id**+**id**)

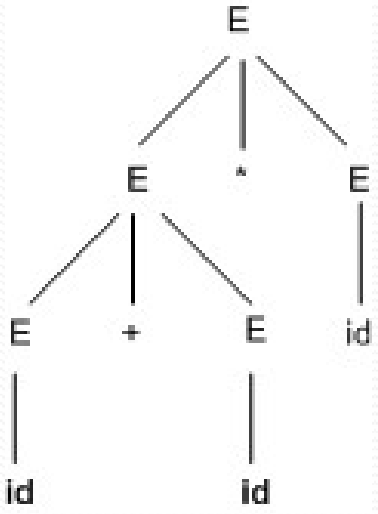
Parse trees

- **-(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 left most derivation
- Or more than one right most derivation
- Example:id+id*id



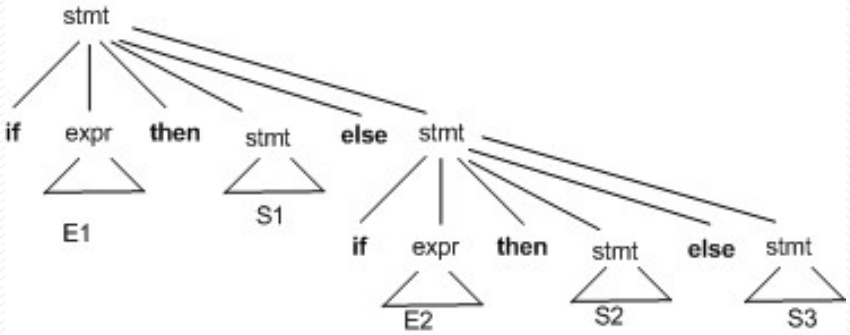


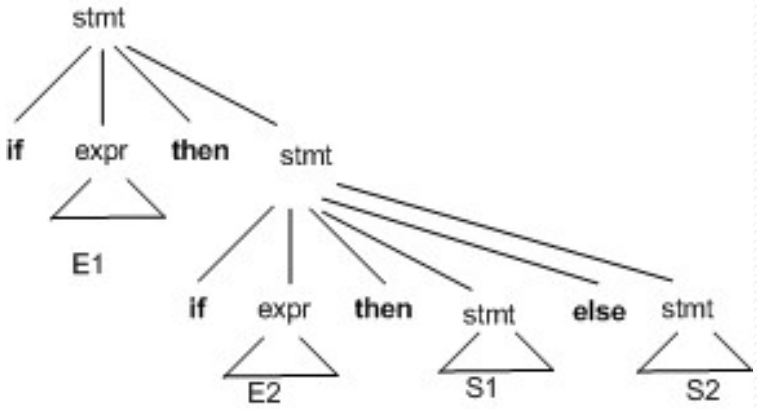
Elimination of ambiguity

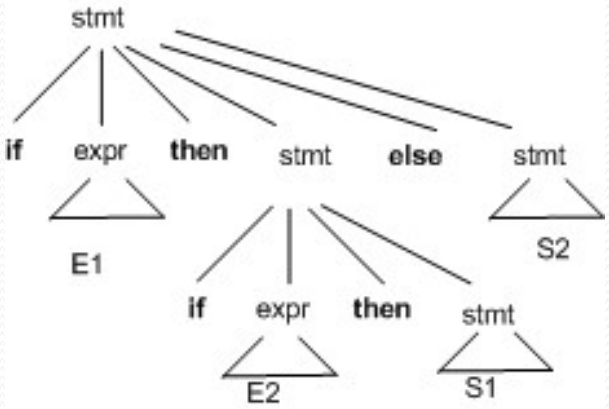
stmt ⟶ **If** expr **then** stmt

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

| **other**

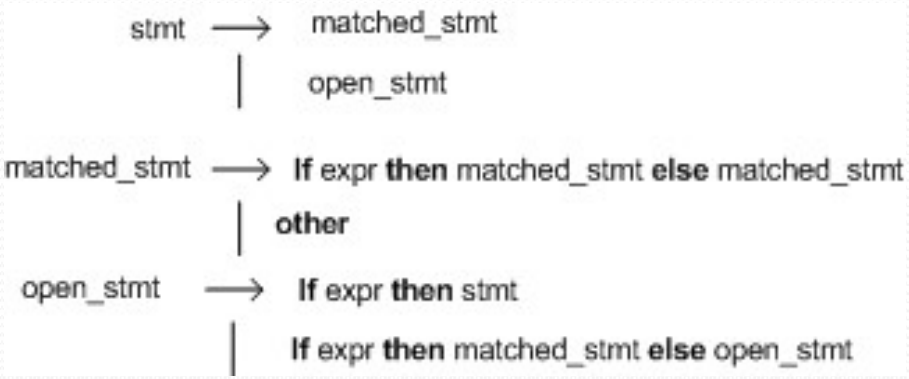






Elimination of ambiguity(cont.)

- 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:
  - ArrangethenonterminalsinsomeorderA1,A2,…,An.
  - For(eachifrom1ton){
    - For(eachjfrom1toi-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-> α β 1| α β 2 Then were place it with
    - A-> α A'
    - A'-> β 1| β 2

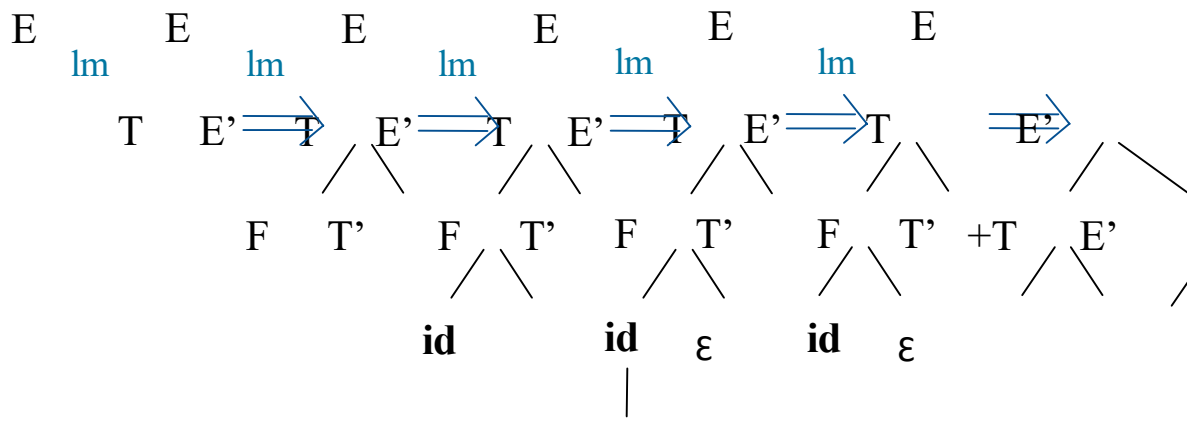
Left factoring(cont.)

- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α <>ε, then replace all of A-productions A-> α β 1| α β 2|... | α β n| γ by
    - A-> α A'| γ
    - A'-> β 1| β 2|...| β n
- Example:
  - S->IEtS|iEtSeS|a
  - E->b

Top-down parser

- A Top-down parser tries to create a parse tree from the root towards the leaf canning input from
- Left to right
- It can be also viewed as finding a left most 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 non terminal
- Execution begins with the procedure for start symbol
- A typical procedure for anon-terminal

```
voidA(){
        chooseanA-production,A->X1X2..Xk for
        (i=1 to k) {
                if(Xiis anonterminal
                        callprocedureXi();
                else if (Xi equals the current input symbol a)
                        advancetheinputtothenextsymbol;
                else/*anerrorhasoccurred*/
        }
}
```

Recursive descentparsing(cont)

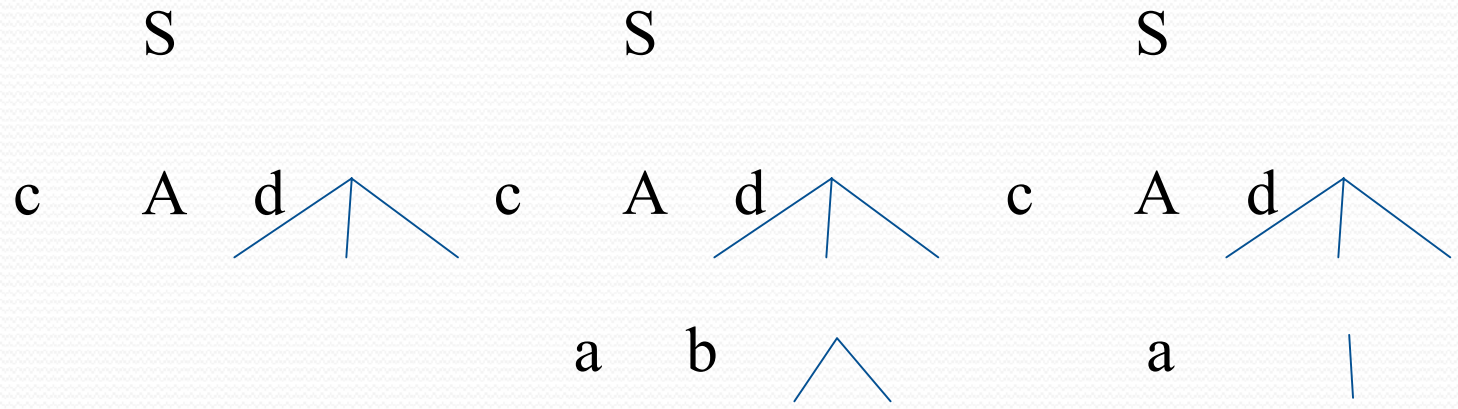
- General recursive descent may require back tracking
- The previous code needs to be modified to allow back tracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- 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 $\alpha \stackrel{*}{=>} \varepsilon$ then is also in First(ε)
- In predictive parsing when we haveA-> $\alpha \mid \beta$,ifFirst(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at then extin put
- Follow(A),for any non terminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have $S \stackrel{*}{=>} \alpha$ Aa β for some α and β then a is in Follow(A)
- If A can be the right most 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 non terminal and X->$Y_1Y_2...Y_k$ is a production for some k>=1,then place a in First(X) if for some ia is in First(Yi) and ε is in all of First(Y_1),...,First(Yi-1)that is$Y_1...Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$. if εis 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 non terminals 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 inFollow(B)
- Example!

LL(1)Grammars

- Predictiveparsersarethoserecursivedescentparsersneedingno backtracking
- GrammarsforwhichwecancreatepredictiveparsersarecalledLL(1)
  - ThefirstLmeansscanninginputfromlefttoright
  - ThesecondLmeansleftmostderivation
  - Andıstandsforusingoneinputsymbolforlookahead
- AgrammarGisLL(1)ifandonlyifwheneverA-> α | β aretwo distinctproductionsofG,thefollowingconditionshold:
  - Fornoterminalado α and β bothderivestringsbeginningwitha
  - Atmostoneof α or β canderiveemptystring
  - If $\alpha \Rightarrow^{*} \varepsilon$ then β doesnotderiveanystringbeginningwitha terminalinFollow(A).

Constructionofpredictive parsing table

- ForeachproductionA-> α ingrammardothe following:
  1. ForeachterminalainFirst(α)addA->inM[A,a]
  2. If ε isinFirst(α), thenforeachterminal bin Follow(A)addA->εtoM[A,b].Ifε isinFirst(α)and $isinFollow(A),addA->εtoM[A,$]aswell
- Ifafterperformingtheabove,thereisnoproduction in M[A,a] thensetM[A,a] toerror

Example

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

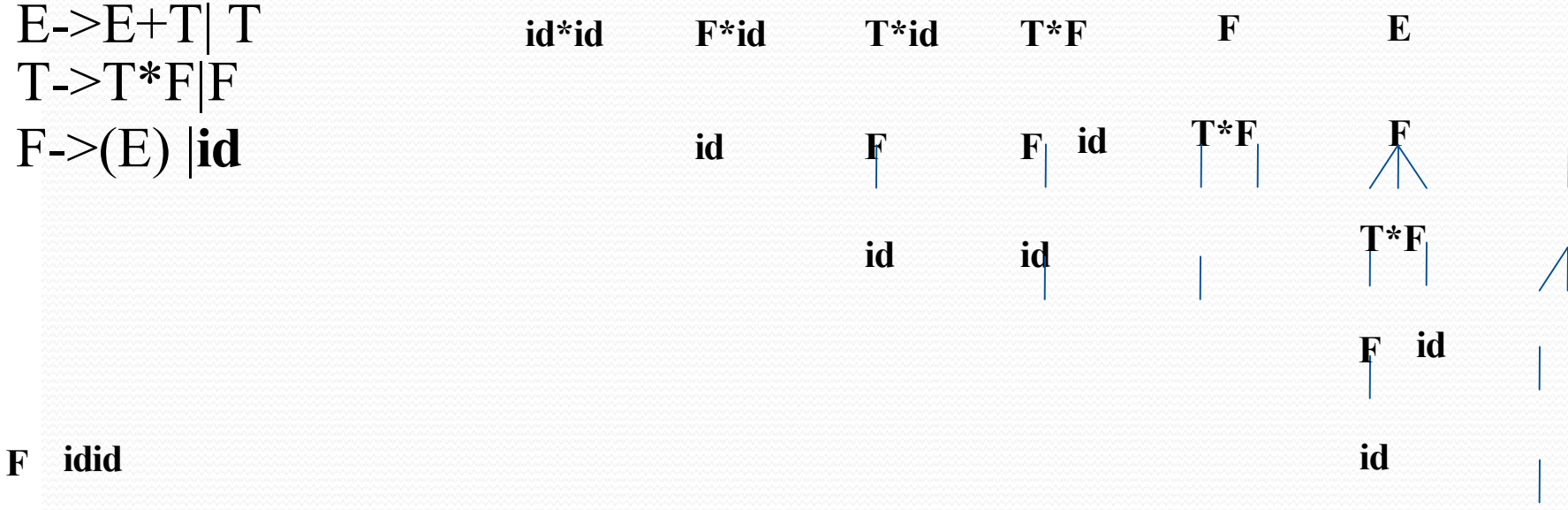
	First	Follow
F	{(,id}	{+,*,),$}
T	{(,id}	{+,),$}
E	{(,id}	{),$}
E’	{+,ε}	{),$}
T’	{*,ε}	{+,),$}

Non-terminal	InputSymbol 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**					

Bottom-upParsing

- Constructsparsetreeforaninputstringbeginningat theleaves(thebottom)andworkingtowardstheroot (the top)
- Example:id*id

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



Shift-reduceparser

- Thegeneralideaistoshiftsomesymbolsofinputto thestackuntilareductioncan beapplied
- Ateachreductionstep,aspecificsubstringmatching thebodyofaproduction isreplaced bythe nonterminal attheheadof theproduction
- Thekeydecisionsduringbottom-upparsingareabout whentoreduceandaboutwhatproductiontoapply
- Areductionisareverseofastepinaderivation
- Thegoalofabottom-upparseristoconstructa derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

Shiftreduceparsing

- Astackisusedtoholdgrammarsymbols
- Handlealwaysappearontopofthestack
- Initialconfiguration:

  Stack Input

  $ w$

- Acceptanceconfiguration

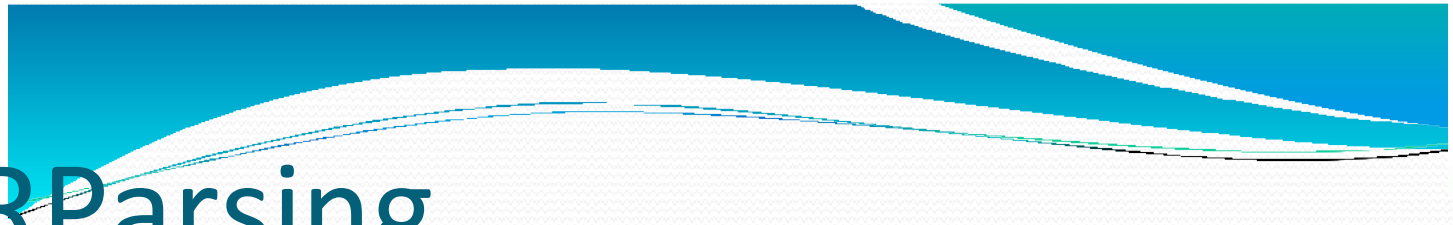
  Stack Input

  $S $

Reduce/reduceconflict

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) …$



LRParsing

- Themostprevalenttypeofbottom-upparsers
- LR(k),mostlyinterestedonparserswithk<=1
- WhyLRparsers?
  - Tabledriven
  - Canbeconstructedtorecognizeallprogramminglanguage constructs
  - Mostgeneralnon-backtrackingshift-reduceparsingmethod
  - Candetectasyntacticerrorassoonasitispossibletodoso
  - ClassofgrammarsforwhichwecanconstructLRparsersare supersetof thosewhichwecanconstructLLparsers

StatesofanLRparser

- Statesrepresentsetofitems
- AnLR(o)itemofGisaproductionofGwiththedotat some positionof the body:
  - ForA->XYZwehavefollowingitems
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - InastatehavingA->.XYZwehopetoseeastring derivablefromXYZnextontheinput.
  - WhataboutA->X.YZ?

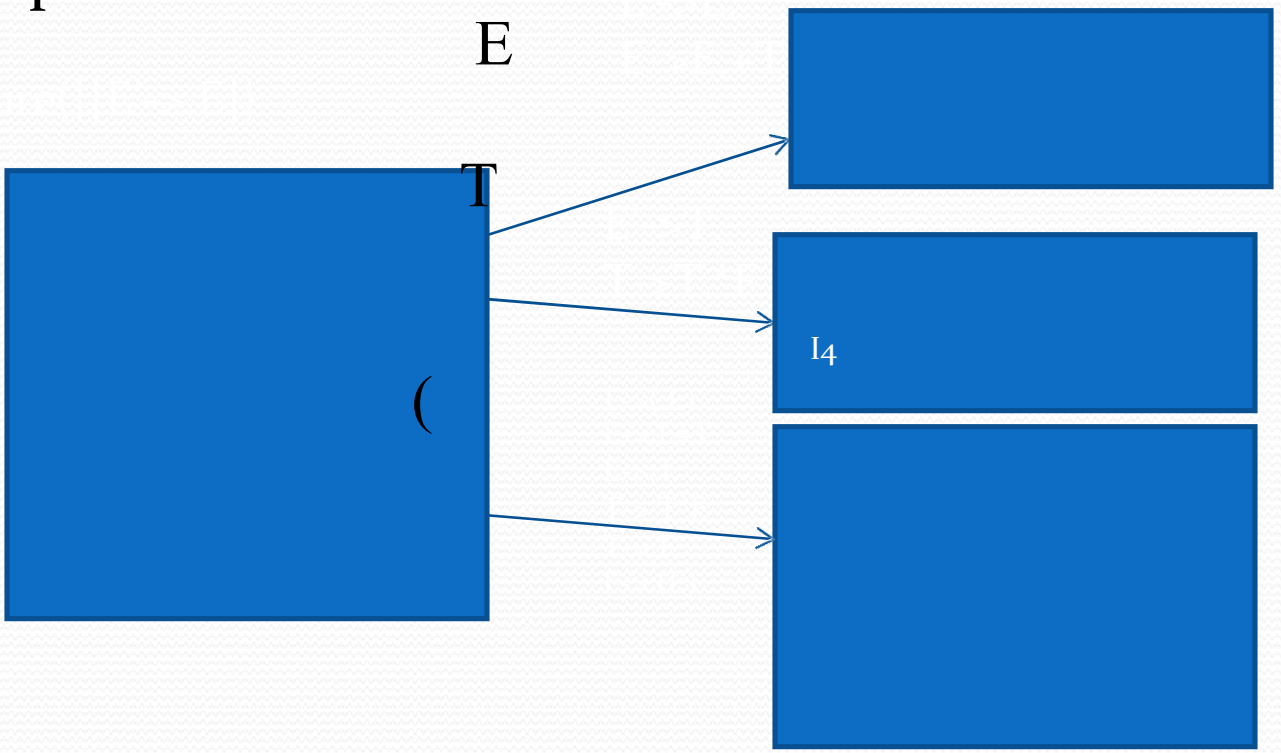
ConstructingcanonicalLR(0) item sets

- Augmentedgrammar:
  - Gwithadditionofaproduction:S'->S
- Closureofitemsets:
  - IfIisasetofitems,closure(I)isasetof itemsconstructedfromIby the following rules:
    - AddeveryiteminItoclosure(I)
    - IfA->α.Bβisinclosure(I)andB->γisaproductionthenaddthe itemB->.γtoclsoure(I).
- Example:

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

Constructingcanonical LR(0) item sets (cont.)

- Goto(I,X)whereI isanitemsetandXisagrammar symbolisclosureofsetofallitems[A->αX.β]where [A->α.X β] is in I
- Example

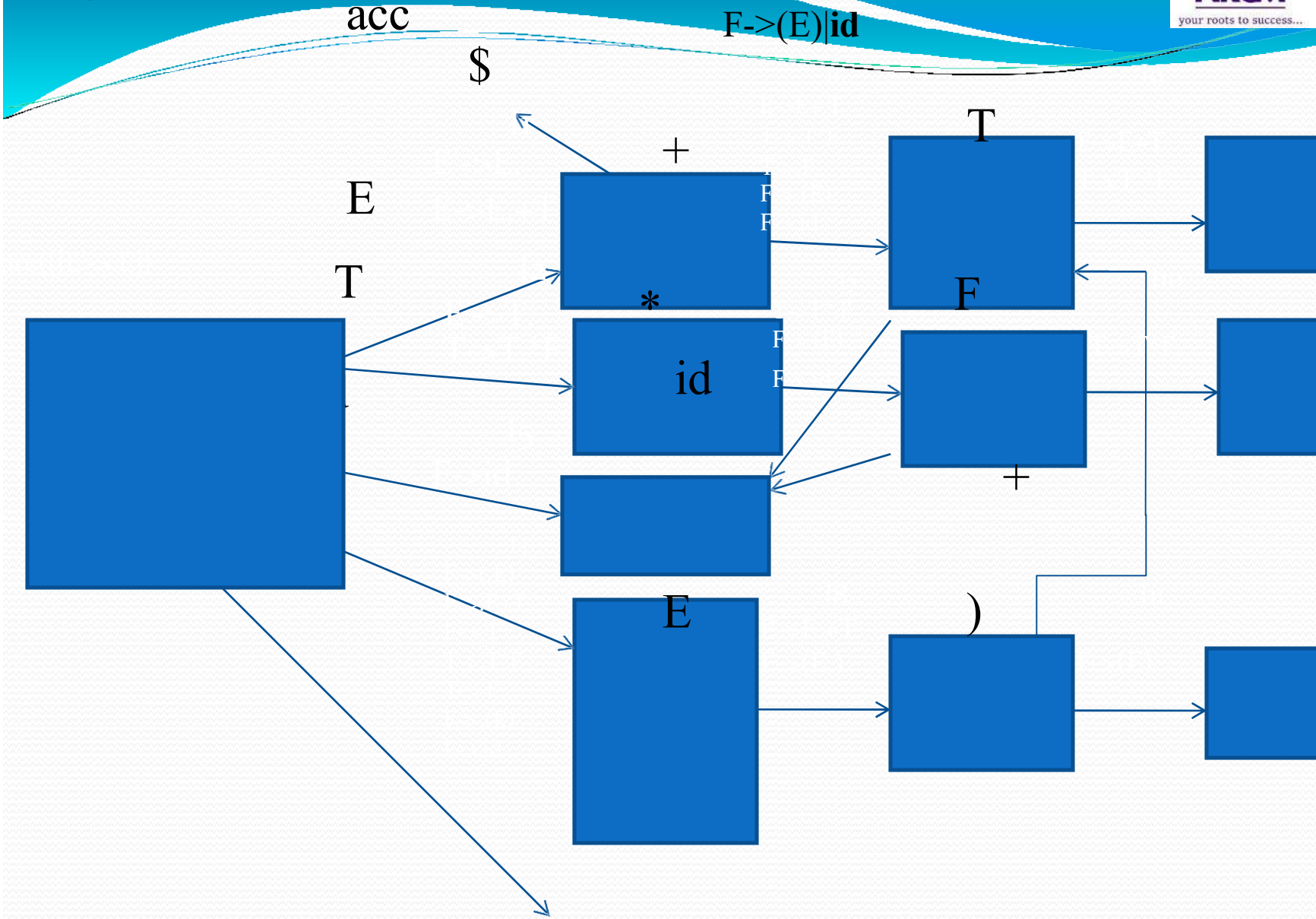


CanonicalLR(0)items

```
Voiditems(G')){
   C=CLOSURE({[S'->.S]});
   repeat
         for(eachsetof itemsI in C)
            for(eachgrammarsymbolX)
               if(GOTO(I,X)isnotemptyandnotinC) add
                  GOTO(I,X) to C;
   untilnonewsetofitemsareaddedtoConaround;
}
```

Example

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

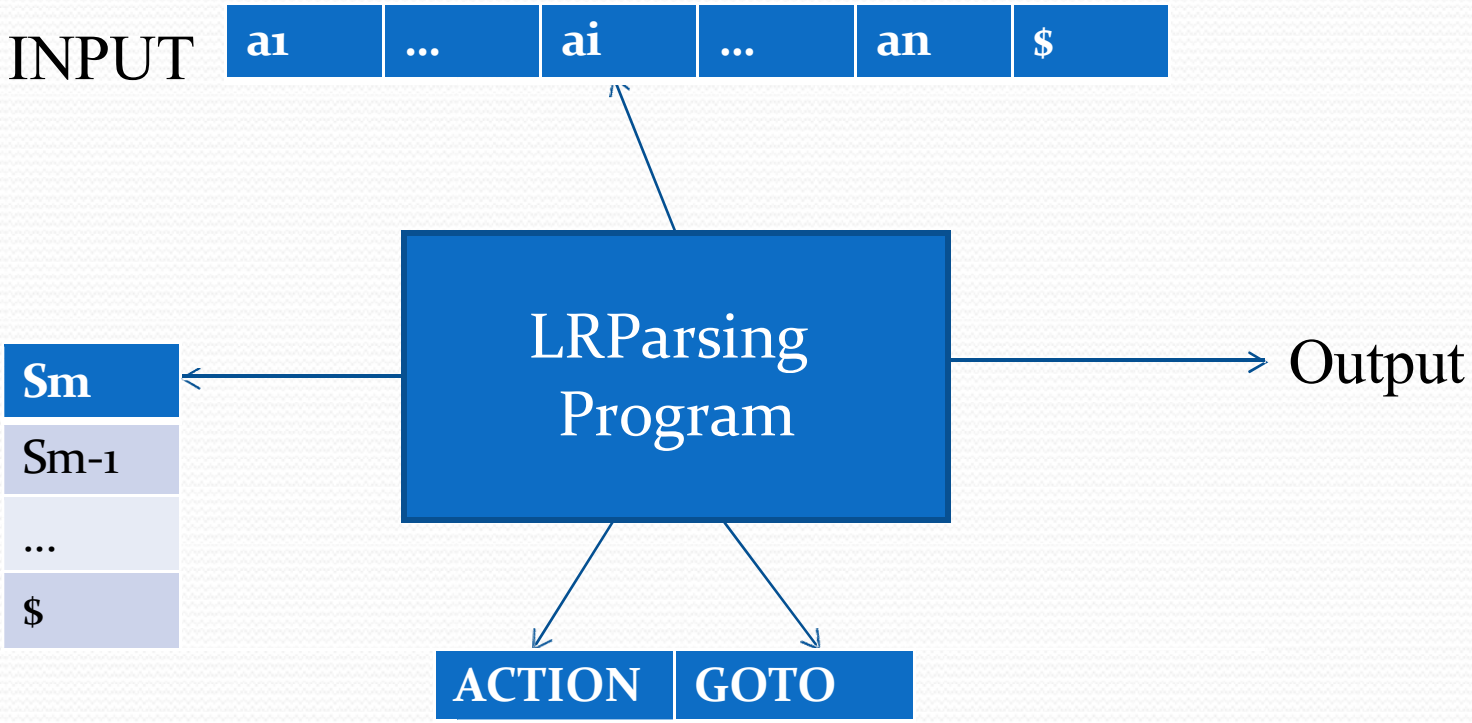


UseofLR(0)automaton

- Example:id*id

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shiftto5
(2)	05	$id	*id$	ReducebyF->id
(3)	03	$F	*id$	ReducebyT->F
(4)	02	$T	*id$	Shiftto7
(5)	027	$T*	id$	Shiftto5
(6)	0275	$T*id	$	ReducebyF->id
(7)	02710	$T*F	$	ReducebyT->T*F
(8)	02	$T	$	ReducebyE->T
(9)	01	$E	$	accept

LR-Parsingmodel



LRparsingalgorithm

```
letabethefirstsymbolofw$;
while(1){/*repeatforever*/
    letsbethestateontopofthestack; if
    (ACTION[s,a] = shiftt) {
            pushtontothestack;
            letabethenextinputsymbol;
    }elseif(ACTION[s,a]=reduceA->β){
            pop |β| symbols of thestack;
            letstatetnowbeontopofthestack;
            push GOTO[t,A] ontothe stack;
            output the production A->β;
    }elseif(ACTION[s,a]=accept)break;/*parsingisdone*/ else
    call error-recoveryroutine;
}
```

Example

(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?

STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R 4	R7		R4	R4			
4	S5			S4			8	2	3
5		R 6	R 6		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			

ConstructingSLRparsingtable

- Method
  - ConstructC={Io,I1,...,In},thecollectionofLR(o)itemsforG'
  - StateiisconstructedfromstateIi:
    - If[A->α.aβ]isinIiandGoto(Ii,a)=Ij,thensetACTION[i,a]to"shiftj"
    - If[A->α.]isinIi,thensetACTION[i,a]to"reduceA->α"forallain follow(A)
    - If{S'->.S]isinIi,thensetACTION[I,$]to"Accept"
  - Ifanyconflictsappearsthenwesaythatthegrammarisnot SLR(1).
  - IfGOTO(Ii,A)=IjthenGOTO[i,A]=j
  - Allentriesnotdefinedbyaboverulesaremade"error"
  - Theinitialstateoftheparseristheoneconstructedfromthe setof itemscontaining [S'->.S]

Examplegrammarwhichisnot 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
=

Shift6

2

ReduceR->

MorepowerfulLRparsers

- Canonical-LRorjustLRmethod
  - Uselookaheadsymbolsforitems:LR(1)items
  - Resultsinalargecollectionofitems
- LALR:lookaheadsareintroducedinLR(0)items

Canonical LR(1)items

- InLR(1)itemseachitemisintheform:[A->α.β,a]
- AnLR(1)item[A->α.β,a]isvalidforaviableprefixγif thereisaderivation$S \overset{*}{\underset{rm}{=}}> \delta Aw => \delta\alpha\beta w$, where
  - Γ=δα
  - Eitheraisthefirstsymbolofw,orwisεandais$
- Example:
  - S->BB
  - B->aB|b

$$S \overset{*}{\underset{rm}{=}}> aaBab => aaaBab$$

Item[B->a.B,a]isvalidforγ=aaa and w=ab

ConstructingLR(1)setsofitem

```
SetOfItemsClosure(I){
    repeat
            for(eachitem[A->α.Bβ,a]inI)
                    for(eachproductionB->γinG')
                            for(eachterminalbinFirst(βa))
                                    add[B->.γ,b]tosetI;
    untilnomoreitemsareaddedtoI;
    returnI;
}

SetOfItemsGoto(I,X){
    initializeJtobetheemptyset;
    for(eachitem[A->α.Xβ,a]inI)
        additem[A->αX.β,a]tosetJ;
    returnclosure(J);
}

voiditems(G'){
    initializeCtoClosure({[S'->.S,$]});
    repeat
            for(eachsetofitemsIinC)
                    for(eachgrammarsymbolX)
                            if(Goto(I,X)isnotemptyandnotinC)
                                    addGoto(I,X)toC;
    untilnonewsetsofitemsareaddedtoC;
}
```

Example

S'->S

S->CC

C->cC

C->d

CanonicalLR(1)parsingtable

- Method
  - ConstructC={Io,I1,...,In},thecollectionofLR(1)itemsforG'
  - StateiisconstructedfromstateIi:
    - If[A->α.aβ,b]isinIiandGoto(Ii,a)=Ij,thensetACTION[i,a]to "shift j"
    - If[A->α.,a]isinIi,thensetACTION[i,a]to"reduceA->α"
    - If{S'->.S,$]isinIi,thensetACTION[I,$]to"Accept"
  - Ifanyconflictsappearsthenwesaythatthegrammarisnot LR(1).
  - IfGOTO(Ii,A)=IjthenGOTO[i,A]=j
  - Allentriesnotdefinedbyaboverulesaremade"error"
  - Theinitialstateoftheparseristheoneconstructedfromthe setof itemscontaining [S'->.S,$]

Example

S'->S

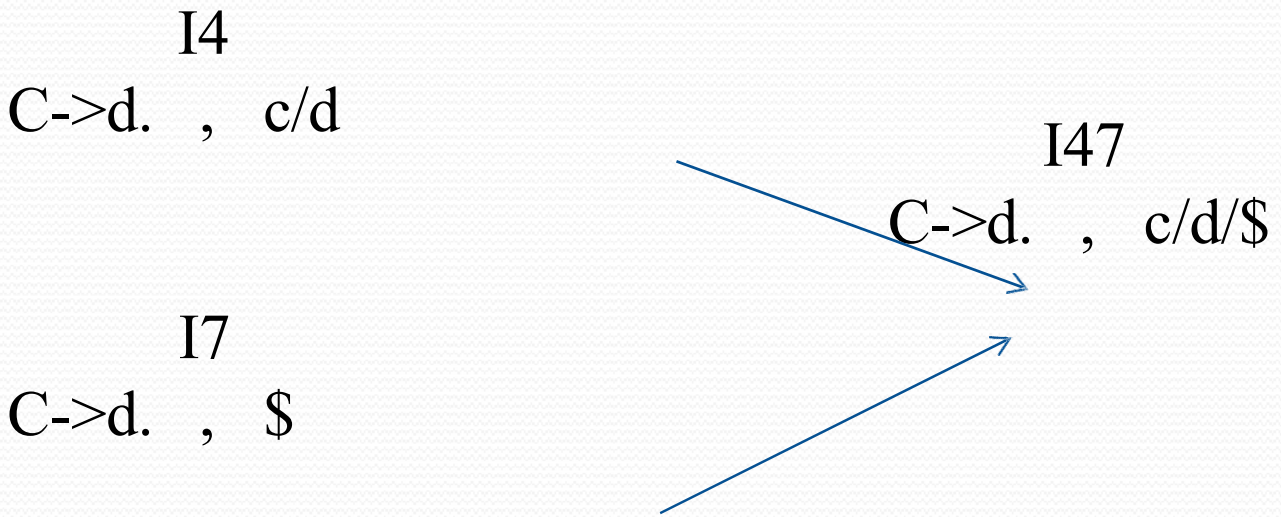
S->CC

C->cC

C->d

LALRParsingTable

- Forthepreviousexamplewehad:



- StatemergescantproduceShift-Reduceconflicts. Why?
- Butitmayproducereduce-reduce conflict

Usingambiguousgrammars

E->E+E

E->E*E

E->(E)

E->id

STATE	ACTON						
	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).

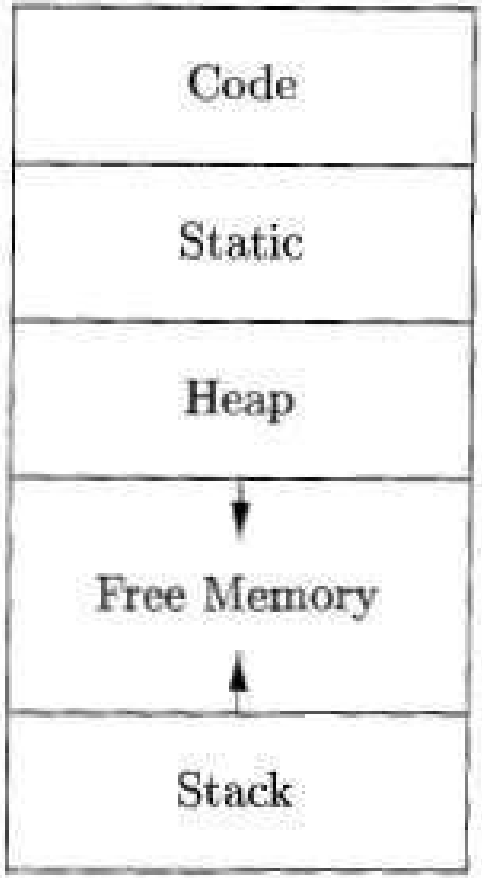
UNIT-IV

Run-TimeEnvironments

Outline

- Compilermustdothestorageallocationandprovide accesstovariables and data
- Memorymanagement
  - Stackallocation
  - Heapmanagement
  - Garbagecollection

StorageOrganization



Staticvs.DynamicAllocation

- Static:Compiletime,Dynamic:Runtimeallocation
- Manycompilersusesomecombinationoffollowing
  - Stackstorage:forlocalvariables,parametersandsoon
  - Heapstorage:Datathatmayoutlivethecalltothe procedure that created it
- Stackallocationisavalidallocationforprocedures sinceprocedurecallsare nested

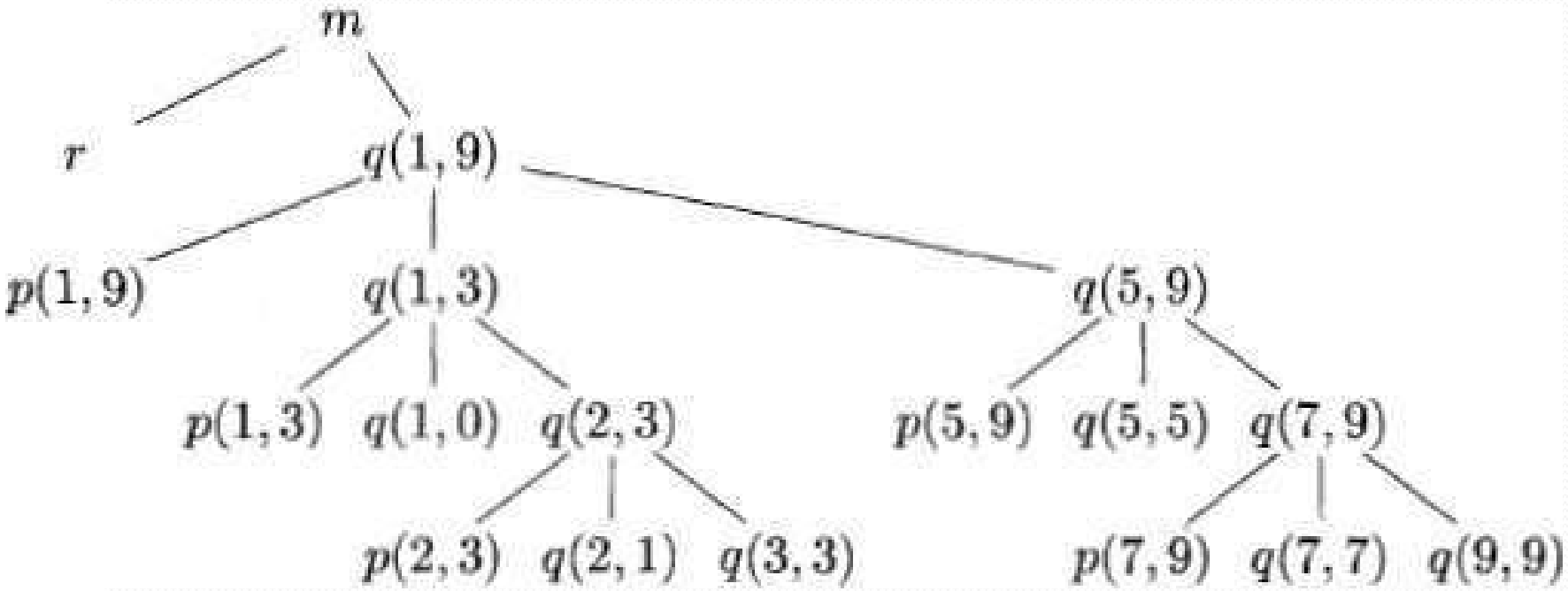
Sketchofaquicksortprogram


```
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);
}
```

ActivationforQuicksort

```
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()
```

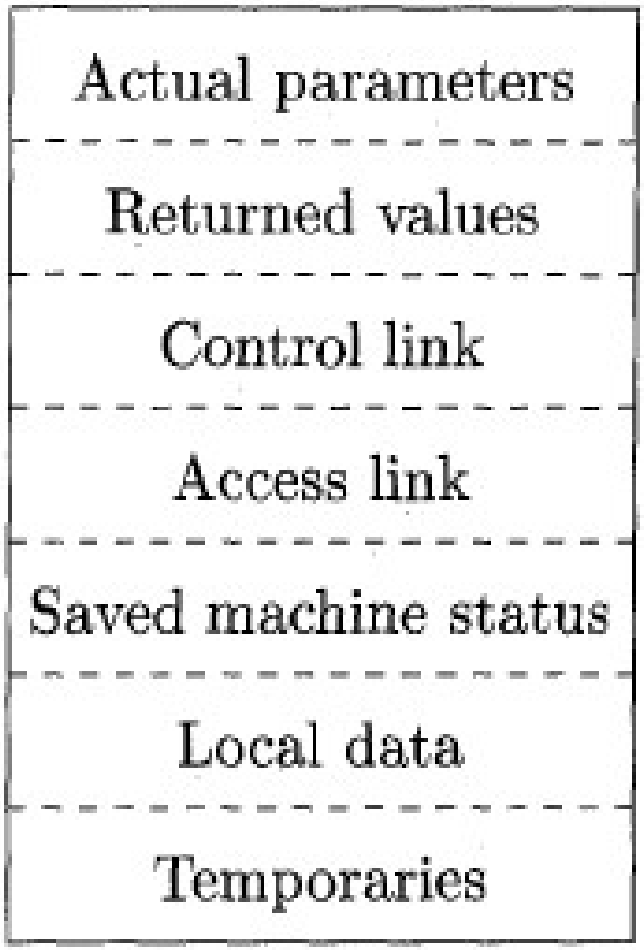
Activationtreerepresentingcallsduri execution of quicksort



Activationrecords

- Procedurecallsandreturnsareusaullymanagedbya run-timestackcalled thecontrol stack.
- Eachliveactivationhasanactivationrecord (sometimes called a frame)
- Therootofactivationtreeisatthebottomofthestack
- Thecurrentexecutionpathspecifiesthecontentofthe stackwiththelastactivationhasrecord inthetopof the stack.

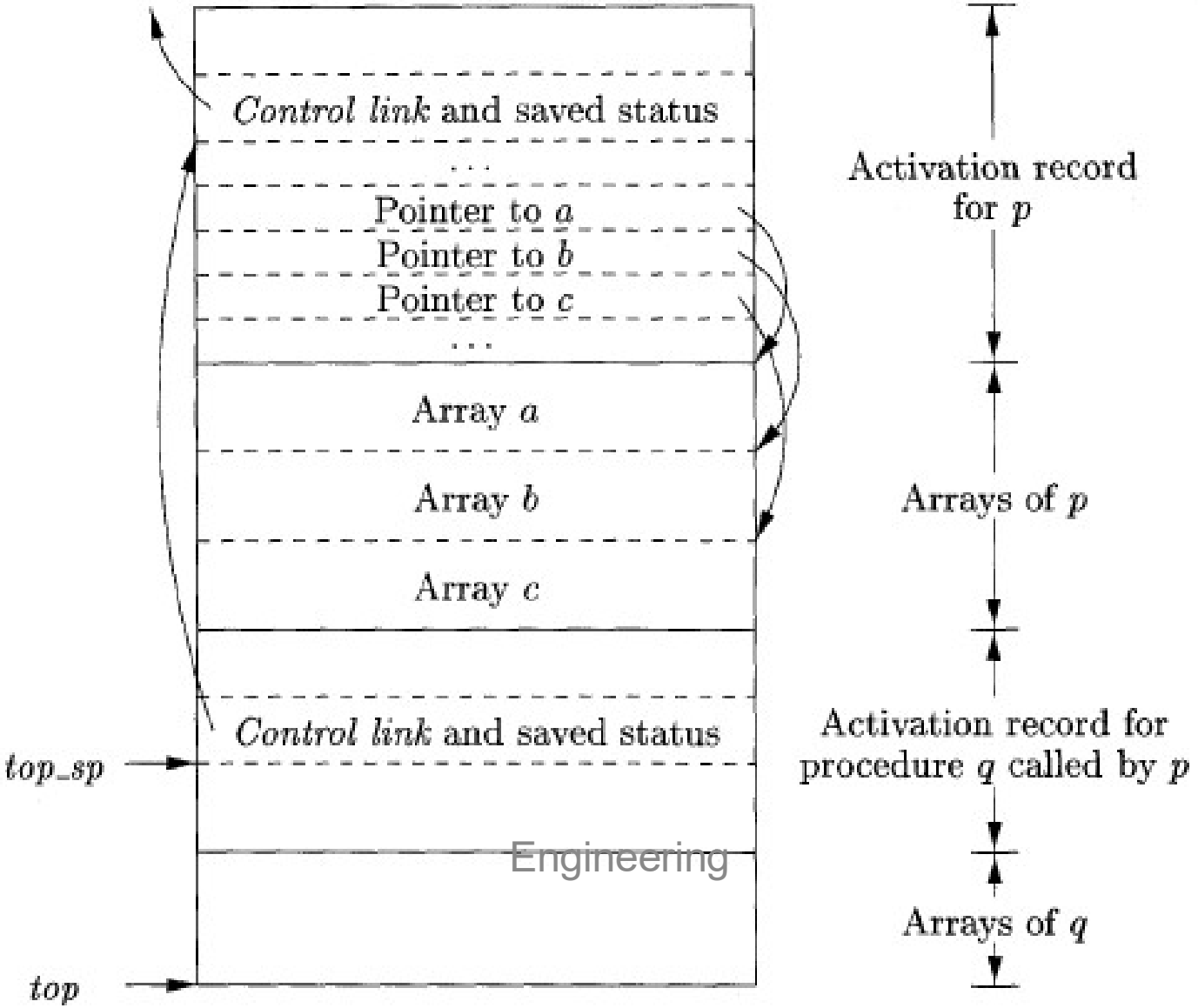
AGeneralActivationRecord



ActivationRecord

- Temporaryvalues
- Localdata
- Asavedmachinestatus
- An“accesslink”
- Acontrollink
- Spaceforthereturnvalueofthecalledfunction
- Theactualparametersusedbythecallingprocedure

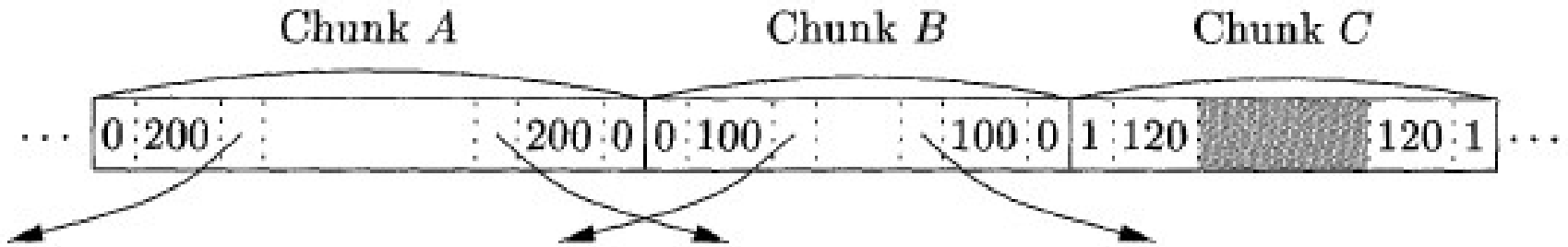
Accesstodynamicallyallocatedarrays



MemoryManager

- Twobasicfunctions:
  - Allocation
  - Deallocation
- Propertiesofmemorymanagers:
  - Spaceefficiency
  - Programefficiency
  - Lowoverhead

Part of a Heap



Introduction

- Thefinalphaseofacompileriscodegenerator
- Itreceivesanintermediaterepresentation(IR)with supplementary information insymbol table
- Producesasemanticallyequivalenttargetprogram
- Codegeneratormaintasks:
  - Instructionselection
  - Registerallocationandassignment
  - Insrtuctionordering

IssuesintheDesignofCode Generator

- Themostimportantcriterionisthatitproducescorrect code
- Inputtothecodegenerator
  - IR+Symboltable
  - Weassumefrontendproduceslow-levelIR,i.e.valuesof namesin itcanbedirectly manipulatedbythemachine instructions.
  - Syntacticandsemanticerrorshavebeenalreadydetected
- Thetargetprogram
  - Commontargetarchitecturesare:RISC,CISCandStack basedmachines
  - InthischapterweuseaverysimpleRISC-likecomputerwith additionof someCISC-likeaddressing modes

complexityofmapping

- thelevelofthe IR
- thenatureoftheinstruction-setarchitecture
- thedesiredqualityofthegeneratedcode.

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
        d, R0
```

Register allocation

- Two sub problems
  - Register allocation: selecting the setoff variables that will reside in register sat 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

ST    R1,t
```

A simple target machine model

- Load operations:LDr,xandLDr1,r2
- Store operations: STx,r
- Computation operations:OPdst,src1,src2
- Unconditional jumps: BRL
- Conditional jumps: Bcondr,LlikeBLTZr,L

Addressing Modes

- Variable name: x
- Indexed address: a(r)likeLDR1,a(R2)means R1=contents(a+ contents (R2))
- Integer indexed by a register:likeLDR1,100(R2)
- Indirect addressing mode:*rand*100(r)
- Immediate constant addressing mode: like LDR1,#100

b=a[i]

```
LDR1,i                    //R1=i

MUL R1, R1, 8             //R1=Rl*8

LD R2, a(R1)

        //R2=contents(a+contents(R1))

STb,R2                    //b=R2
```

a[j]=c

```
LDR1,c                  //R1=c
LDR2,j                  //R2=j
MUL R2, R2, 8           //R2=R2*8
STa(R2), R1
        //contents(a+contents(R2))=R1
```

UNIT-V

Machine-Independent Optimization

Machine independent optimization attempts to improve the intermediate code to get a better target code. The part of the code which is transformed here does not involve any absolute memory location or any CPU registers.

Code Optimization can perform in the following different ways:

(1) Compile Time Evaluation:

(a) z = 5*(45.0/5.0)*r

Perform 5*(45.0/5.0)*rat compile time.

(b) x= 5.7

y= x/3.6

Evaluate x/3.6as5.7/3.6atcompiletime.

(2) Variable Propagation:

Before Optimization the code is:

```
c = a * b
x=a
till
d=x*b+4
```

After Optimization the code is:

```
c = a * b
x=a
till
    d=a*b+4
```

(3) Dead code elimination:

Before elimination the code is:

c = a * b

x=b

till

d=a*b+4

After elimination the code is:

c = a * b

till

d=a*b+4

(4) Code Motion:

It reduces the evaluation frequency of expression.

It brings loop in variant statements out of the loop. do

```
{
   Item =10;
   value value=value+ item;
} while (value<100);
```

//This code can be further optimized as

```
item = 10;
do
{
   Value value=value+ item;
} while (value<100);
```

(5) Induction Variable and Strength Reduction: Streng th reduction is used to replace the high strength operator by the low strength.

An induction variable is used in loop for thefollowing kind of assignment like i = i + constant.

Before reduction the codeis:

After Reduction the codeis:

```
i = 1
t=4
{
   while(t<40)
  y = t;
  t =t +4;
}
```

Data Flow Analysis

To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.

Certain optimization can only be achieved by examining the entire program. It can't be achieve by examining just a portion of the program.

consider the following code:

x = a + b;

x =6 *3

Some optimization needs more global information. For example, consider the following code:

a=1;

b=2;

c=3;

if (....)x=a+ 5;

else x=b+4; c

= x + 1;