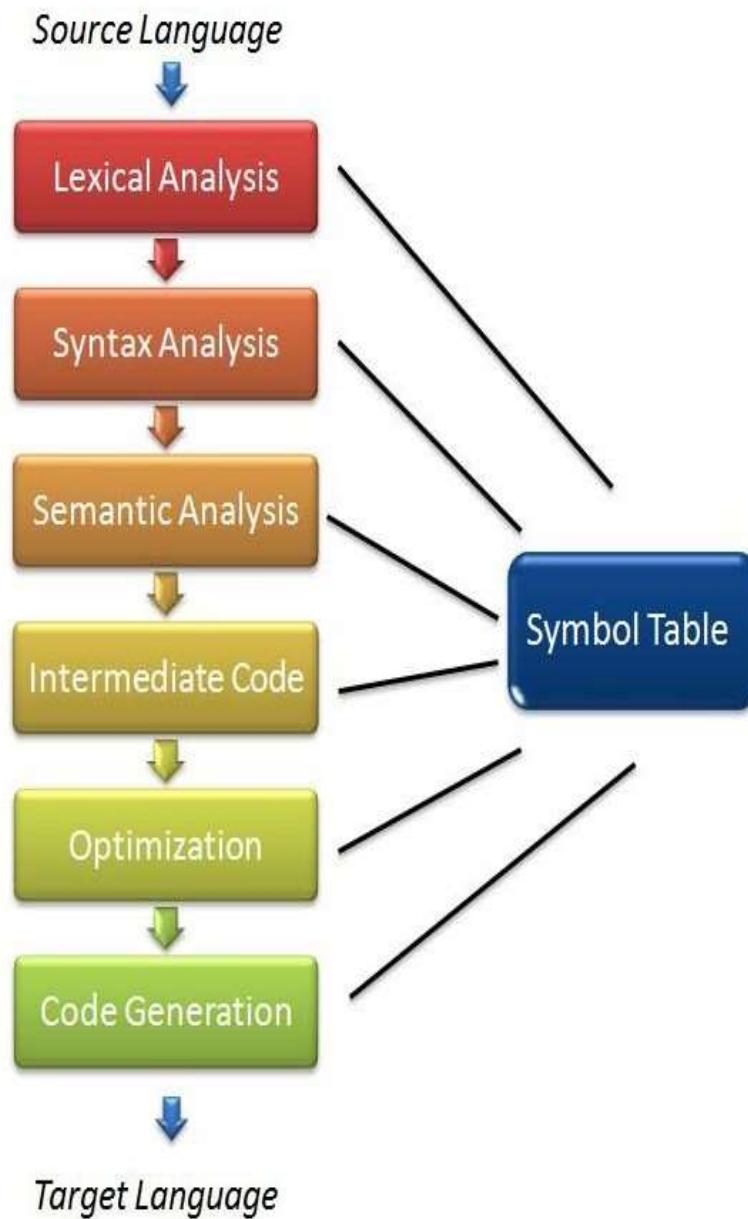


# CS 304-Compiler Design



**Dr. M.Venkatesan**  
Assistant Professor  
Department of Computer Science and  
Engineering  
National Institute of Technology  
Karnataka  
Mangalore  
[venkisakthi77@gmail.com](mailto:venkisakthi77@gmail.com)  
[venkisakthi@nitk.edu.in](mailto:venkisakthi@nitk.edu.in)  
9442314011



# Compiler Design Syllabus

Introduction to language processing; Lexical analysis, Regular languages and finite automata; syntactic analysis, Context-free languages; Semantic analysis and syntax-directed translation; Error analysis; Intermediate representation and intermediate code generation; The procedure abstraction, Run-time environments and storage allocation; Code generation, Instruction selection, Register allocation; Code optimization, Data-flow analysis and control flow analysis.

Books:

- 1.Aho, Lam, Sethi, Ullman Compilers: Principles, Techniques, and Tools, Addison-Wesley, (2007/2013) ISBN-10: 0321486811
- 2.Y. N. Srikant and Priti Shankar: The Compiler Design Handbook: Optimizations and Machine Code Generation, CRC Press, 2002. ISBN 084931240X
- 3.Tremblay and Sorenson: The Theory and Practice of Compiler Writing, McGraw-Hill, 1985.
- 4.Grune, Bal, Jacobs, Langendoen: Modern Compiler Design, John Wiley and Sons, (2000)
- 5.Steven Muchnick: Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997. ISBN 1-558-60320-4.
- 6.Keith Cooper, Linda Torczon: Engineering a Compiler, Morgan Kaufmann; 2 edition (2011)
- 7.Andrew Appel: Modern Compiler Implementation in Java, Cambridge University Press, (2002)

# Definitions

## What is a compiler?

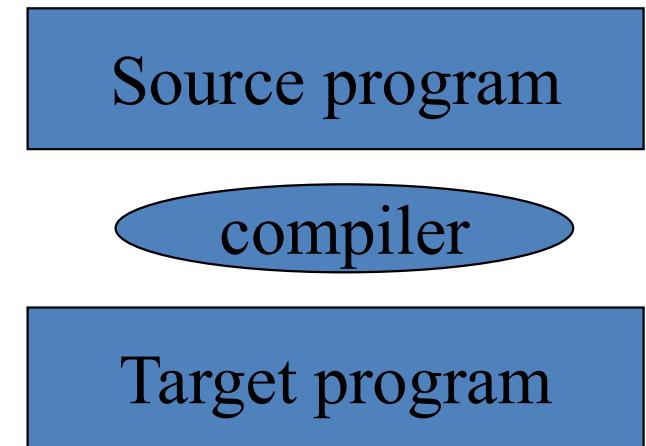
- A program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text (Grune *et al*, 2000).
- A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) (Aho *et al*)

# Examples

- C is typically compiled
- Lisp is typically interpreted
- Java is compiled to bytecodes, which are then interpreted

Also:

- C++ to Intel Core 2/.../Assembly
- C++ to C
- High Performance Fortran (HPF) to Fortran (parallelising compiler)
- C to C (or any language to itself)



# Qualities of a Good Compiler

What qualities would you want in a compiler?

- generates correct code (first and foremost!)
- generates fast code
- conforms to the specifications of the input language
- copes with essentially arbitrary input size, variables, etc.
- compilation time (linearly)proportional to size of source
- good diagnostics
- consistent optimisations
- works well with the debugger

# Principles of Compilation

*The compiler must:*

- *preserve the meaning of the program being compiled.*  
*"improve" the source code in some way.*

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging (transformations obscure the relationship source code vs target)
- Compilation time efficiency (fast or slow compiler?)

# Why study Compilation Technology?

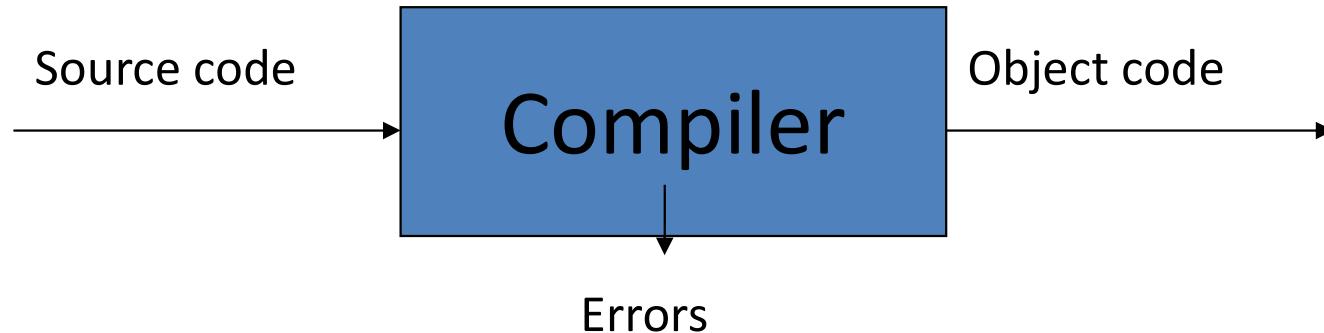
- Success stories (one of the earliest branches in CS)
  - Applying theory to practice (scanning, parsing, static analysis)
  - Many practical applications have embedded languages (eg, tags)
- Practical algorithmic & engineering issues:
  - Approximating really hard (and interesting!) problems
  - Emphasis on efficiency and scalability
  - Small issues can be important!
- Ideas from different parts of computer science are involved:
  - AI: Heuristic search techniques; greedy algorithms - Algorithms: graph algorithms - Theory: pattern matching - Also: Systems, Architecture
- Compiler construction can be challenging and fun:
  - new architectures always create new challenges; success requires mastery of complex interactions; results are useful; opportunity to achieve performance.

# Uses of Compiler Technology

- Most common use: translate a high-level program to object code
  - Program Translation: binary translation, hardware synthesis, ...
- Optimizations for computer architectures:
  - Improve program performance, take into account hardware parallelism, etc...
- Automatic parallelisation or vectorisation
- Performance instrumentation: e.g., -pg option of cc or gcc
- Interpreters: e.g., Python, Ruby, Perl, Matlab, sh, ...
- Software productivity tools
  - Debugging aids: e.g, purify
- Security: Java VM uses compiler analysis to prove “safety” of Java code.
- Text formatters, just-in-time compilation for Java, power management, global distributed computing, ...

**Key: Ability to extract properties of a source program (analysis) and transform it to construct a target program (synthesis)**

# General Structure of a Compiler



**The compiler:-**

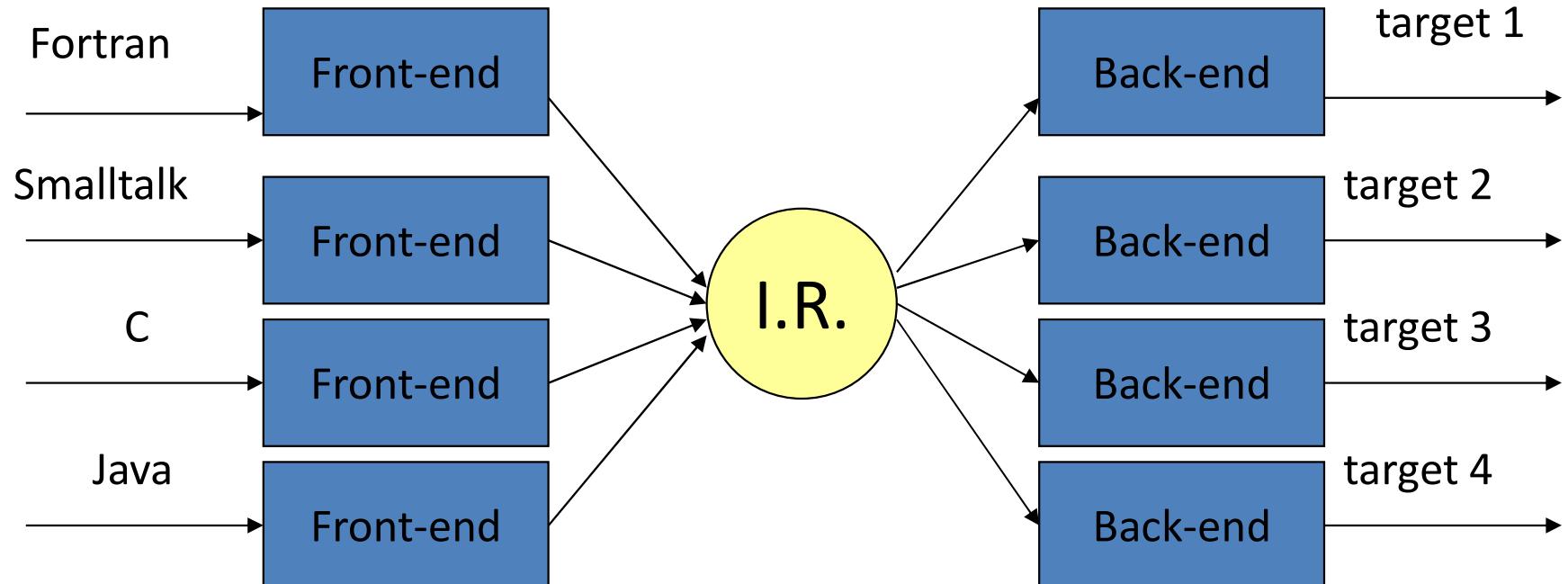
- must generate correct code.
- must recognise errors.
- analyses and synthesises.

# Conceptual Structure: two major phases



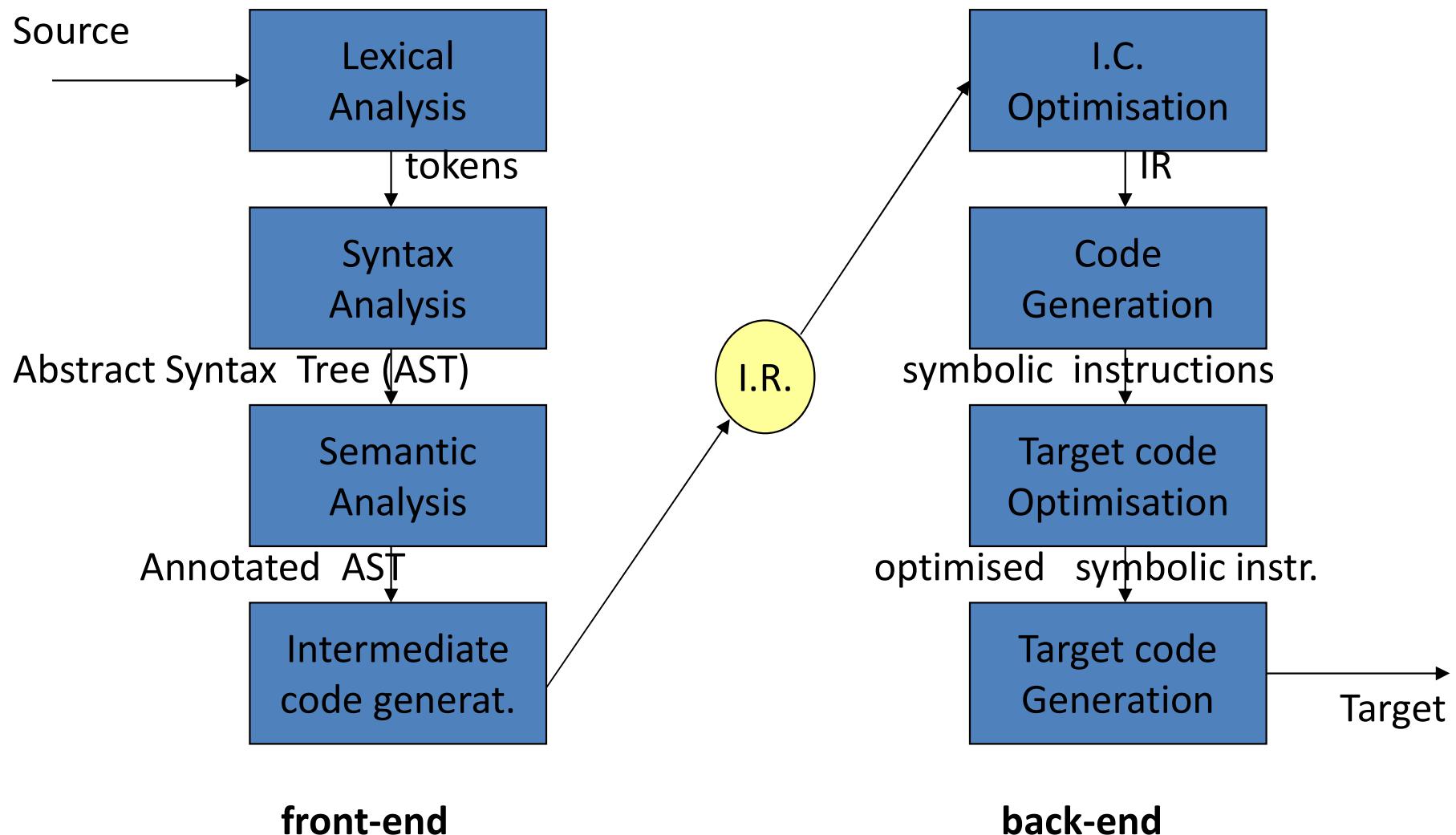
- **Front-end** performs the **analysis** of the source language:
  - Recognises legal and illegal programs and reports errors.
  - “understands” the input program and collects its semantics in an IR.
  - Produces IR and shapes the code for the back-end.
  - Much can be automated.
- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.
  - Needs to conform with system interfaces.
  - Automation has been less successful.
- Typically front-end is **O(n)**, while back-end is **NP-complete**.

# $m \times n$ compilers with $m+n$ components!



- All language specific knowledge must be encoded in the front-end
- All target specific knowledge must be encoded in the back-end

# General Structure of a compiler



# Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognises what sort they are.
- The output is called token and is a pair of the form  $\langle type, lexeme \rangle$  or  $\langle token\_class, attribute \rangle$
- E.g.:  $a=b+c$  becomes  $\langle id, a \rangle \langle =, \rangle \langle id, b \rangle \langle +, \rangle \langle id, c \rangle$
- Needs to record each id attribute: keep a **symbol table**.
- Lexical analysis eliminates white space, etc...
- Speed is important - use a specialised tool: e.g., flex - a tool for generating **scanners**: programs which recognise lexical patterns in text; for more info: `% man flex`

# Syntax (or syntactic) Analysis (Parsing)

- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalise these recursive rules and guide syntax analysis.
- Example:

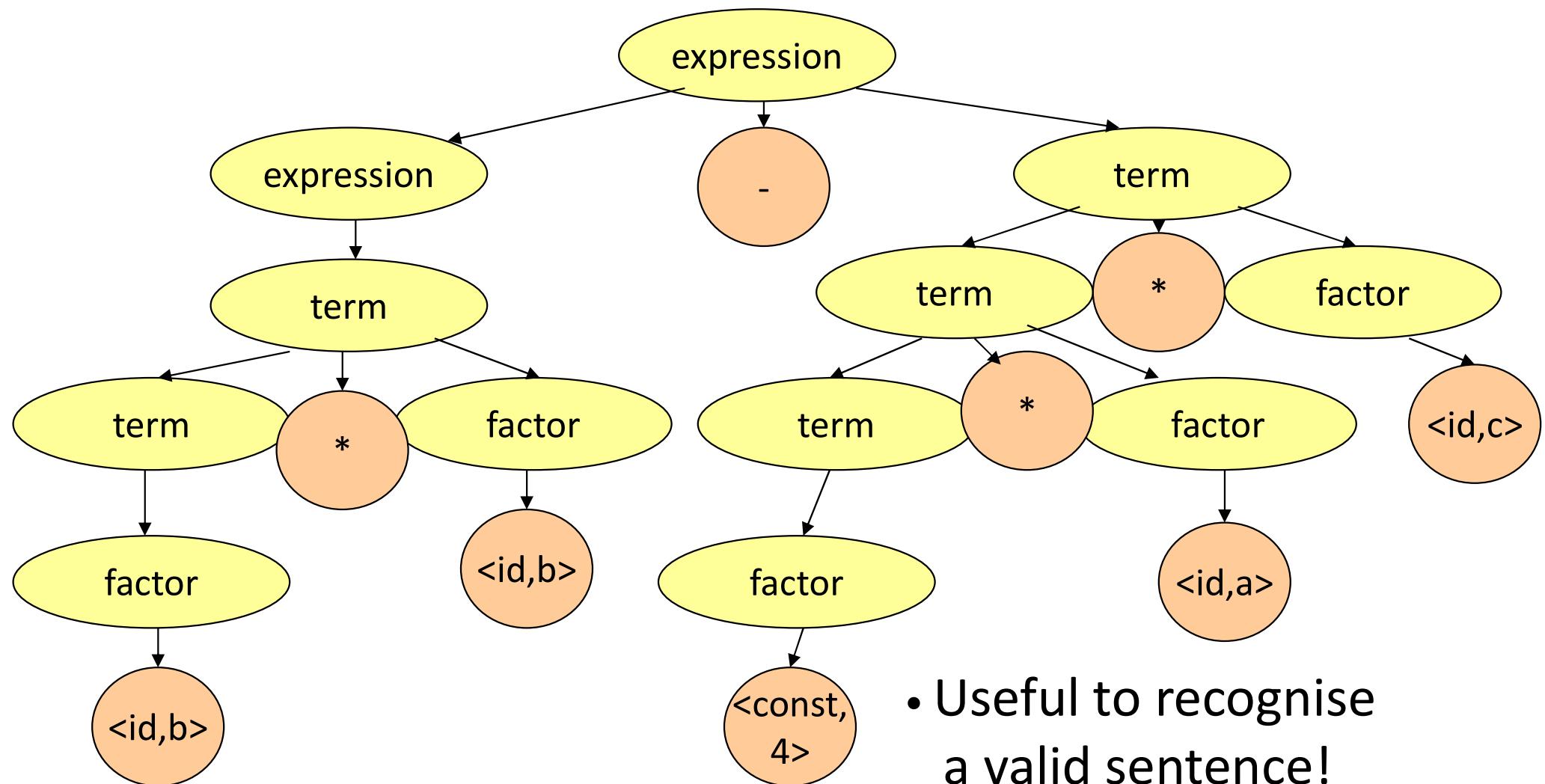
**expression** → **expression** '+' **term** | **expression** '-' **term** | **term**

**term** → **term** '\*' **factor** | **term** '/' **factor** | **factor**

**factor** → **identifier** | **constant** | '(' **expression** ')'

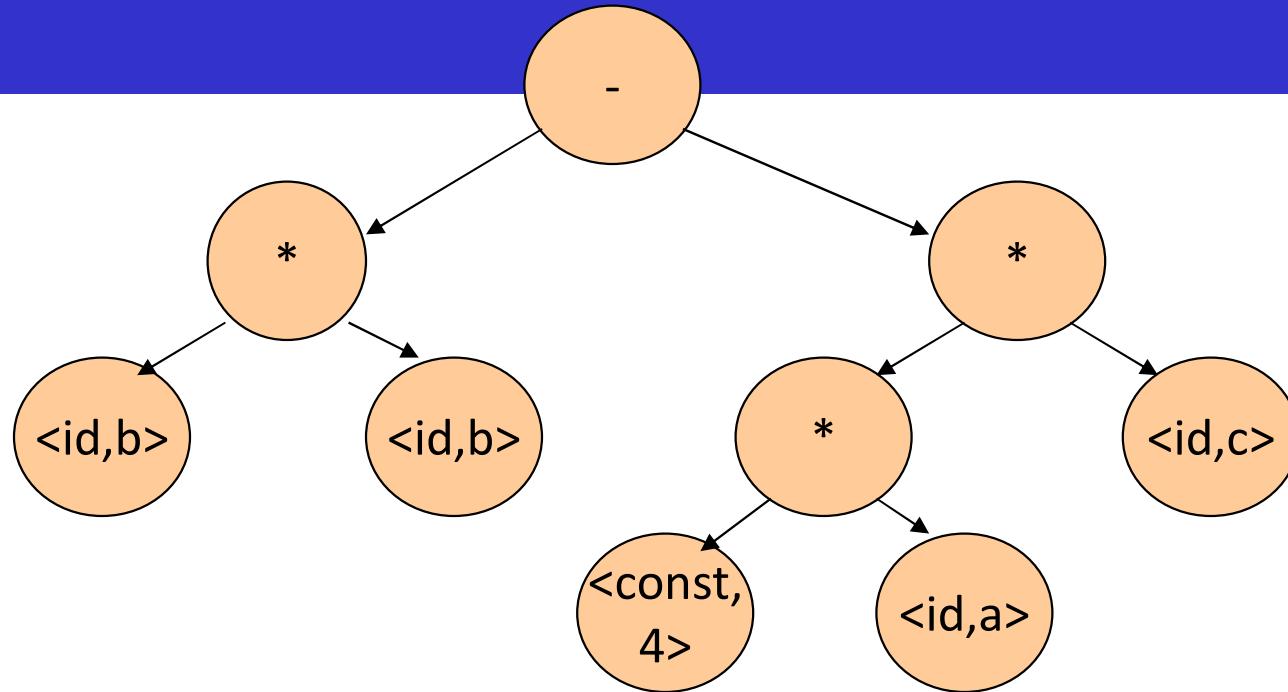
(this grammar defines simple algebraic expressions)

# Parsing: parse tree for $b^*b - 4^*a^*c$



- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

# AST for $b*b-4*a*c$



- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)
- ASTs are one form of IR

# Semantic Analysis (context handling)

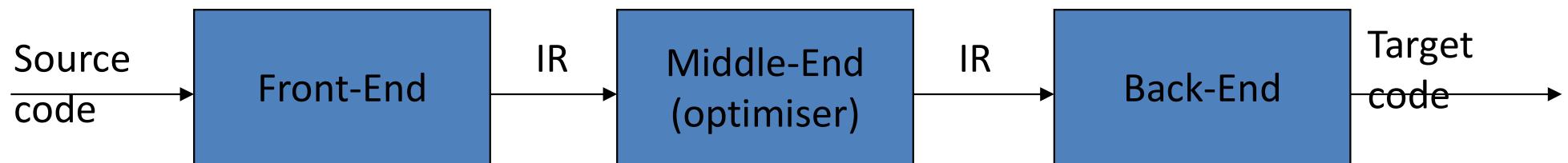
- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.
- Examples:
  - type checking: report error if an operator is applied to an incompatible operand.
  - check flow-of-controls.
  - uniqueness or name-related checks.

# Intermediate code generation

- Translate language-specific constructs in the AST into more general constructs.
- A criterion for the level of “generality”: it should be straightforward to generate the target code from the intermediate representation chosen.
- Example of a form of IR (3-address code):  
`tmp1=4  
tmp2=tmp1*a  
tmp3=tmp2*c  
tmp4=b*b  
tmp5=tmp4 - tmp3`

# Code Optimisation

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code.
- Optimisations can range from trivial (e.g. constant folding) to highly sophisticated (e.g, in-lining).
- For example: replace the first two statements in the example of the previous slide with: `tmp2=4*a`
- Modern compilers perform such a range of optimisations, that one could argue for:



# Code Generation Phase

- Map the AST onto a linear list of target machine instructions in a symbolic form:
  - Instruction selection: a pattern matching problem.
  - Register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem.
  - Instruction scheduling: take advantage of multiple functional units: NP-Complete problem.
- Target, machine-specific properties may be used to optimise the code.
- Finally, machine code and associated information required by the Operating System are generated.

# Some historical notes...

Emphasis of compiler construction research:

- 1945-1960: code generation
  - need to “prove” that high-level programming can produce efficient code (“automatic programming”).
- 1960-1975: parsing
  - proliferation of programming languages
  - study of formal languages reveals powerful techniques.
- 1975-...: code generation and code optimisation

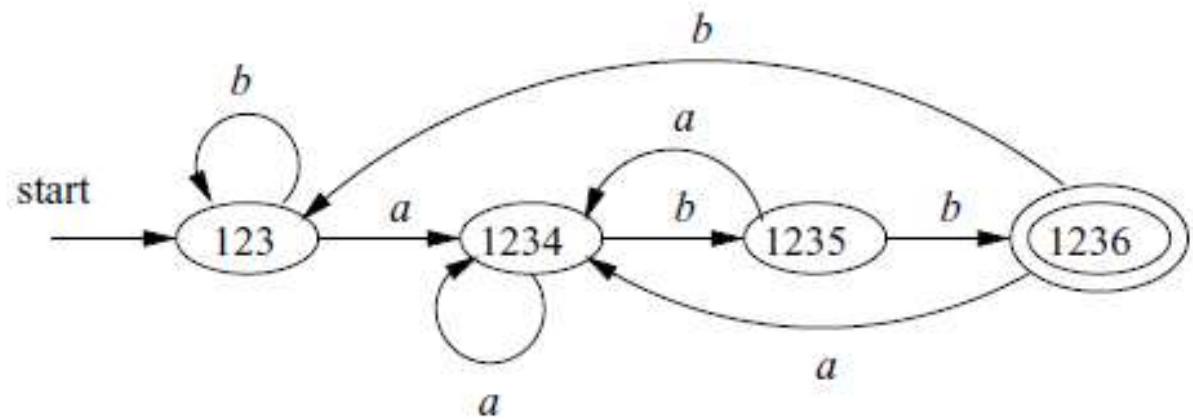
Knuth (1962) observed that “*in this field there has been an unusual amount of parallel discovery of the same technique by people working independently*”

# Historical Notes: the Move to Higher-Level Programming Languages

- Machine Languages (1<sup>st</sup> generation)
- Assembly Languages (2<sup>nd</sup> generation) – early 1950s
- High-Level Languages (3<sup>rd</sup> generation) – later 1950s
- 4<sup>th</sup> generation higher level languages (SQL, Postscript)
- 5<sup>th</sup> generation languages (logic based, eg, Prolog)
- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

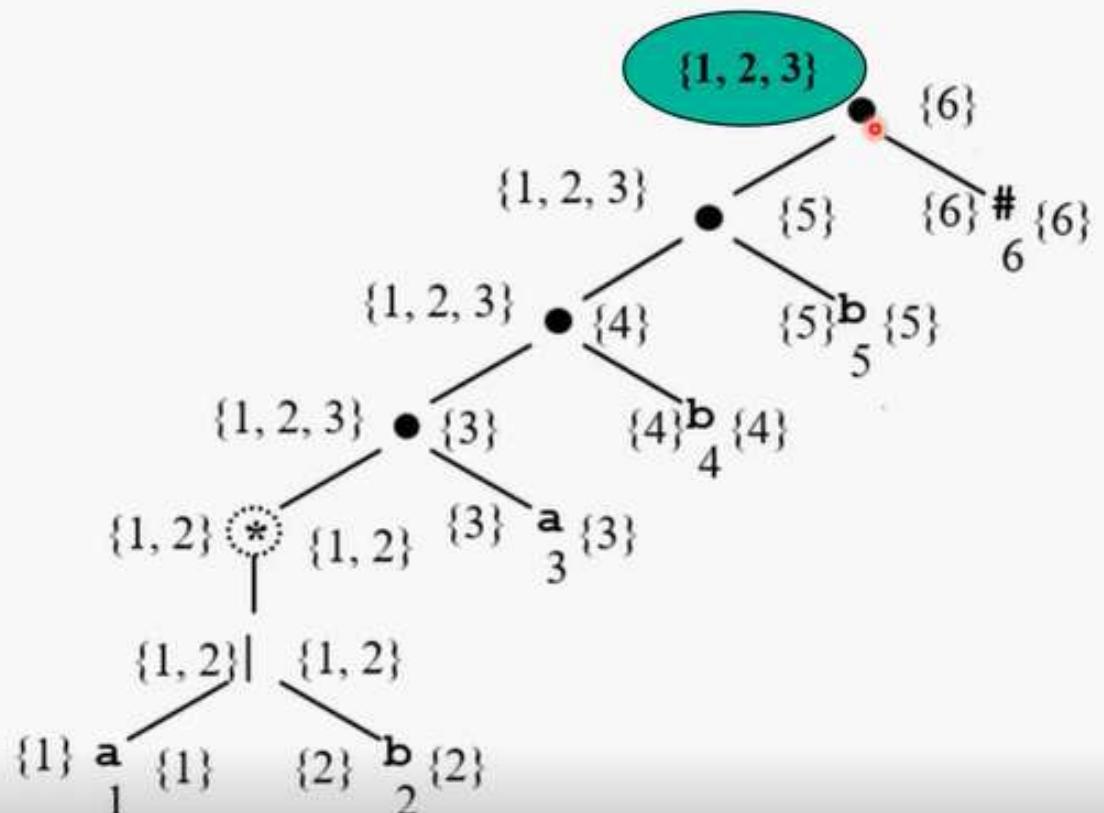
POSITION	$n$	$followpos(n)$
1		$\{1, 2, 3\}$
2		$\{1, 2, 3\}$
3		$\{4\}$
4		$\{5\}$
5		$\{6\}$
6		$\emptyset$

Figure 3.60: The function



$$(a \mid b)^*. a . b . b \#$$

	<b>Node</b>	<i>followpos</i>
a	1	1,2,3
b	2	1,2,3
a	3	4
b	4	5
b	5	6
#	6	-



# Syntax directed translation

Syntax directed translation

Grammar + Semantic rules = SDT

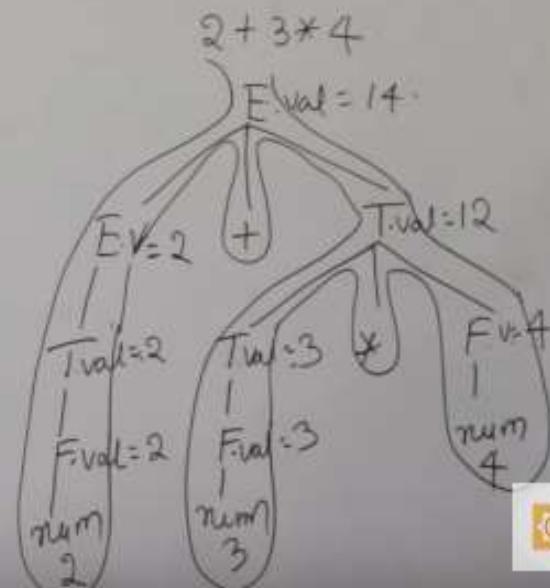
SDT for evaluation of expression

$E \rightarrow E_1 + T \quad \{ E.value = E_1.value + T.value \}$   
 $/ T \quad \{ E.value = T.value \}$

$T \rightarrow T_1 * F \quad \{ T.value = T_1.value * F.value \}$   
 $/ F \quad \{ T.value = F.value \}$

$F \rightarrow \text{num} \quad \{ F.val = \text{num}.lvalue \}$

✓



SOT

$E \rightarrow E + T \{ \text{printf}( "+") ; \}$   
①

$/ T \{ \}$  ②

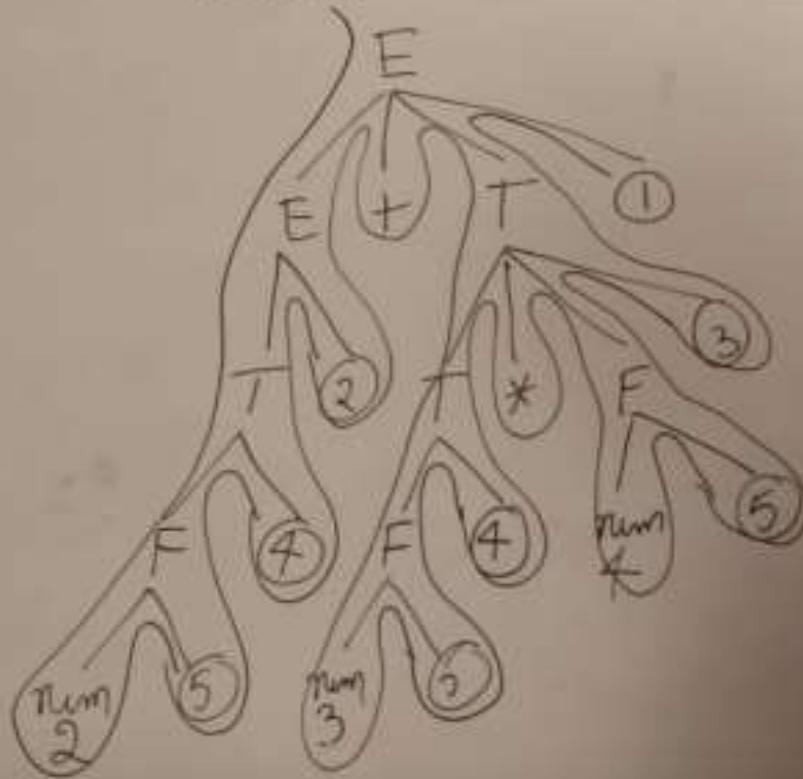
$T \rightarrow T * F \{ \text{printf}( "*") ; \}$   
③

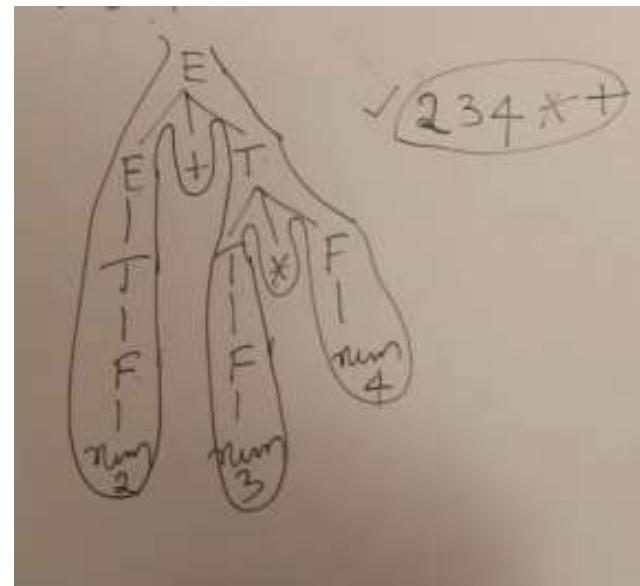
$/ F \{ \}$  ④

$F \rightarrow \text{num} \{ \text{printf}( \text{num}.lval) ; \}$   
⑤

2 + 3 \* 4.

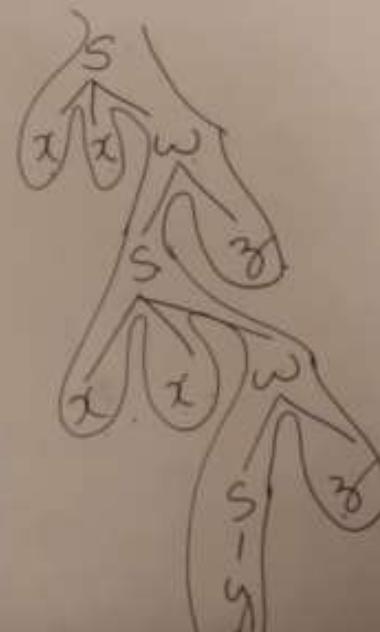
2 3 4 \* +





$S \rightarrow xxw \{ \text{printf}(1); \}$   
 $y \quad \{ \text{printf}(2); \}$   
 $w \rightarrow s z \{ \text{printf}(3); \}$

String  $xxxyzz$



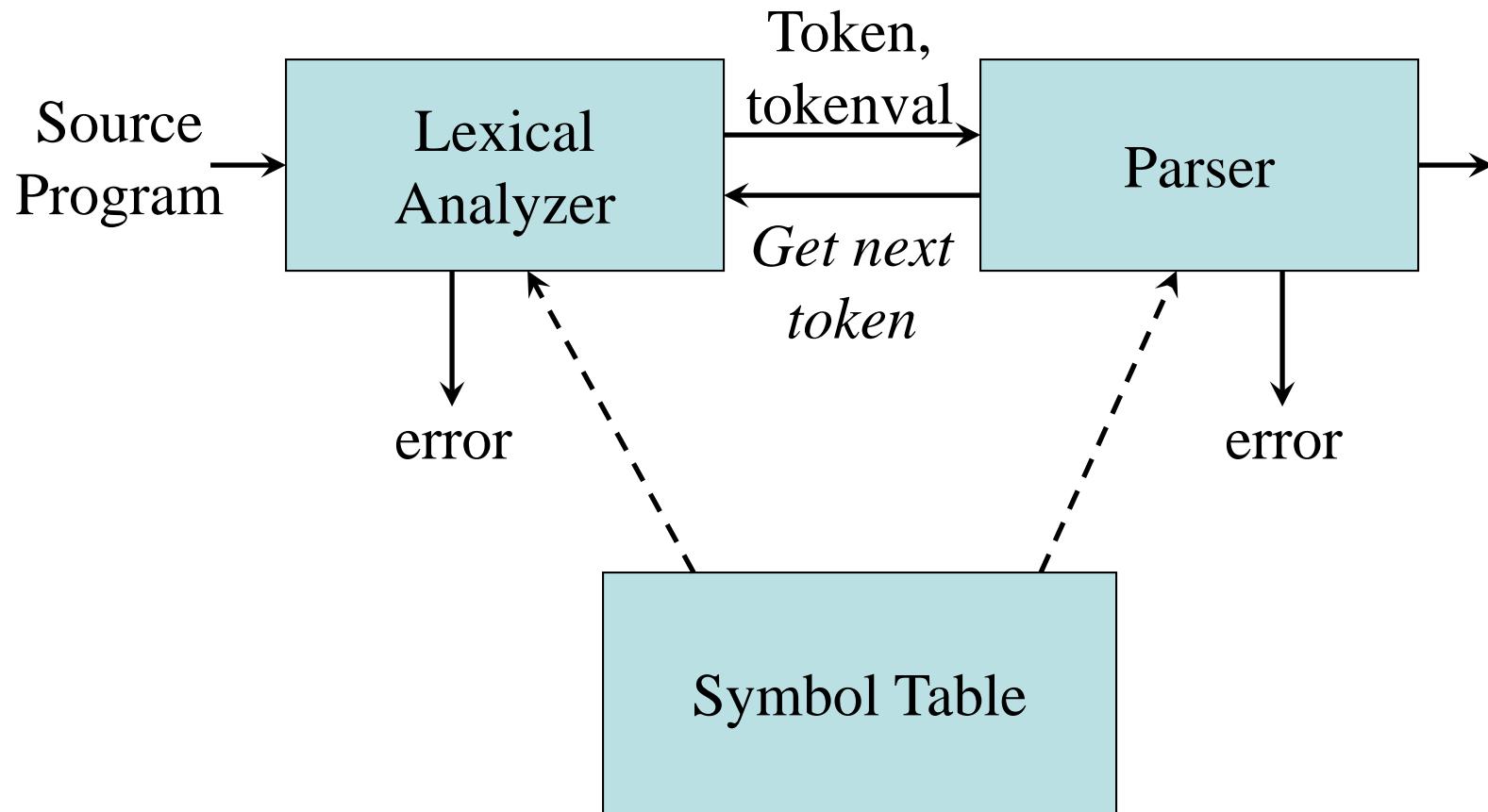
[2 3 1 3 1]

# Lexical Analyzer

# The Reason Why Lexical Analysis is a Separate Phase

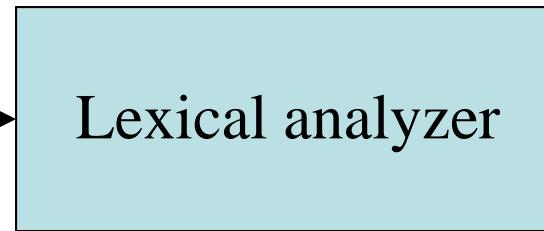
- Simplifies the design of the compiler
  - LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be normalized (e.g. trigraphs)

# Interaction of the Lexical Analyzer with the Parser



# Attributes of Tokens

`y := 31 + 28*x` →



Lexical analyzer

`<id, "y"> <assign, > <num, 31> <+, > <num, 28> <*>, > <id, "x">`

token

**tokenval**

(token attribute)

Parser



# Tokens, Patterns, and Lexemes

- A *token* is a classification of lexical units
  - For example: **id** and **num**
- *Lexemes* are the specific character strings that make up a token
  - For example: **abc** and **123**
- *Patterns* are rules describing the set of lexemes belonging to a token
  - For example: “*letter followed by letters and digits*” and “*non-empty sequence of digits*”

# Specification of Patterns for Tokens: *Definitions*

- An *alphabet*  $\Sigma$  is a finite set of symbols (characters)
- A *string*  $s$  is a finite sequence of symbols from  $\Sigma$ 
  - $|s|$  denotes the length of string  $s$
  - $\varepsilon$  denotes the empty string, thus  $|\varepsilon| = 0$
- A *language* is a specific set of strings over some fixed alphabet  $\Sigma$

# Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings  $x$  and  $y$  is denoted by  $xy$
- The *exponentiation* of a string  $s$  is defined by

$$s^0 = \varepsilon$$

$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that  $s\varepsilon = \varepsilon s = s$

# Specification of Patterns for Tokens: *Language Operations*

- *Union*

$$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$$

- *Concatenation*

$$LM = \{xy \mid x \in L \text{ and } y \in M\}$$

- *Exponentiation*

$$L^0 = \{\varepsilon\}; \quad L^i = L^{i-1}L$$

- *Kleene closure*

$$L^* = \bigcup_{i=0,\dots,\infty} L^i$$

- *Positive closure*

$$L^+ = \bigcup_{i=1,\dots,\infty} L^i$$

# Specification of Patterns for Tokens: *Regular Expressions*

- Basis symbols:
  - $\varepsilon$  is a regular expression denoting language  $\{\varepsilon\}$
  - $a \in \Sigma$  is a regular expression denoting  $\{a\}$
- If  $r$  and  $s$  are regular expressions denoting languages  $L(r)$  and  $M(s)$  respectively, then
  - $r|s$  is a regular expression denoting  $L(r) \cup M(s)$
  - $rs$  is a regular expression denoting  $L(r)L(s)$
  - $r^*$  is a regular expression denoting  $L(r)^*$
  - $(r)$  is a regular expression denoting  $L(r)$
- A language defined by a regular expression is called a *regular set*

# Specification of Patterns for Tokens: *Regular Definitions*

- Regular definitions introduce a naming convention:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

...

$$d_n \rightarrow r_n$$

where each  $r_i$  is a regular expression over

$$\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$$

- Any  $d_j$  in  $r_i$  can be textually substituted in  $r_i$  to obtain an equivalent set of definitions

# Specification of Patterns for Tokens: *Regular Definitions*

- Example:

$$\begin{aligned}\textbf{letter} &\rightarrow \texttt{A} \mid \texttt{B} \mid \dots \mid \texttt{z} \mid \texttt{a} \mid \texttt{b} \mid \dots \mid \texttt{z} \\ \textbf{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \dots \mid \texttt{9} \\ \textbf{id} &\rightarrow \textbf{letter} (\textbf{letter} \mid \textbf{digit})^*\end{aligned}$$

- Regular definitions are not recursive:

$$\textbf{digits} \rightarrow \textbf{digit} \textbf{digits} \mid \textbf{digit} \qquad \textit{wrong!}$$

# Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

$$r^+ = rr^*$$

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

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

- Examples:

**digit** → [0-9]

**num** → **digit**<sup>+</sup> (. **digit**<sup>+</sup>)? ( E (+ | -)? **digit**<sup>+</sup> )?

# Regular Definitions and Grammars

Grammar

$stmt \rightarrow \text{if } expr \text{ then } stmt$

$\text{if } expr \text{ then } stmt \text{ else } stmt$   
 $\epsilon$

$expr \rightarrow term \text{ relop } term$

$term$   
 $term$

$term \rightarrow \text{id}$

$num$

Regular definitions

**if** → **if**

**then** → **then**

**else** → **else**

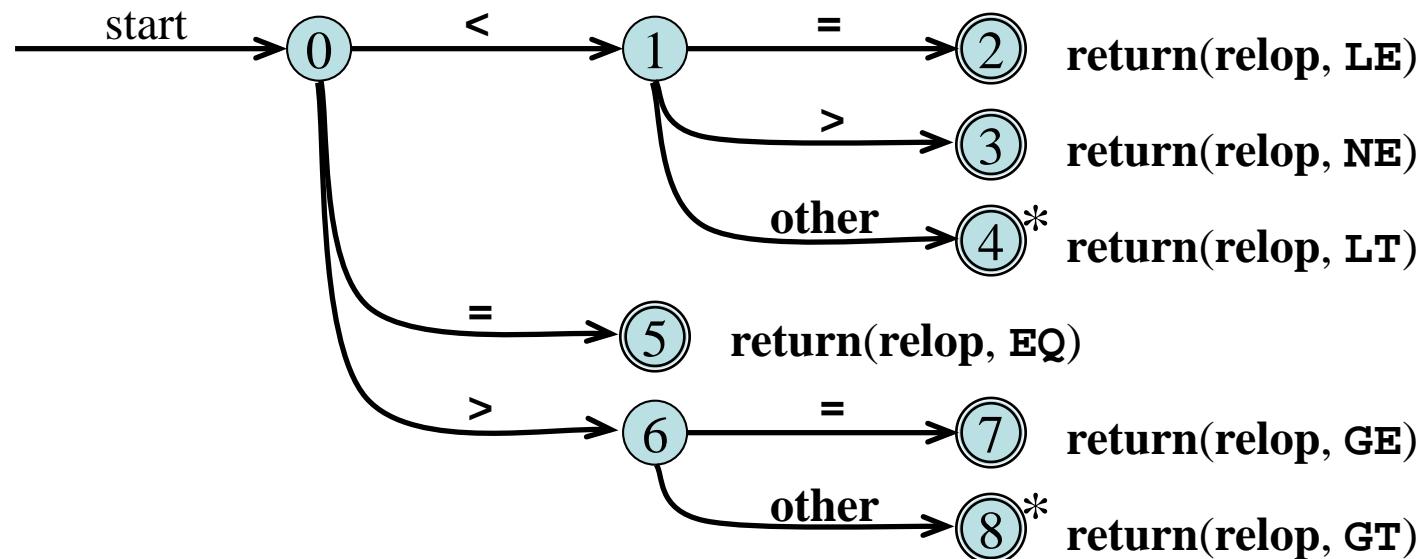
**relop** → < |  $\leq$  |  $\neq$  | > |  $\geq$  | =

**id** → letter ( letter | digit )\*

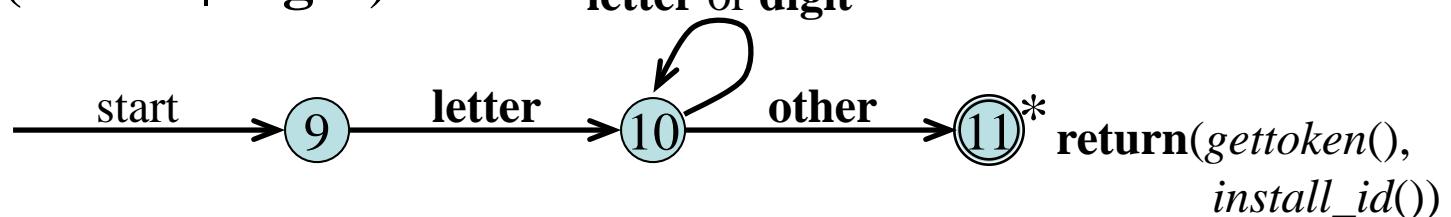
**num** → digit<sup>+</sup> ( . digit<sup>+</sup>)? ( E (+ | -)? digit<sup>+</sup>)?

# Coding Regular Definitions in *Transition Diagrams*

**rellop** → < | <= | <> | > | >= | =



**id** → letter ( letter | digit )<sup>\*</sup>



# Coding Regular Definitions in Transition Diagrams: Code

```

token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
        if (c==blank || c==tab || c==newline) {
            state = 0;
            lexeme_beginning++;
        }
        else if (c=='<') state = 1;
        else if (c=='=') state = 5;
        else if (c=='>') state = 6;
        else state = fail();
        break;
    case 1:
        ...
    case 9: c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
    case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
    ...
}
}

```

Decides the  
next start state  
to check



```

int fail()
{ forward = token_beginning;
switch (start) {
    case 0: start = 9; break;
    case 9: start = 12; break;
    case 12: start = 20; break;
    case 20: start = 25; break;
    case 25: recover(); break;
    default: /* error */
    }
return start;
}

```

# Limits of Regular Languages

---

Not all languages are regular

$$\text{RL's} \subset \text{CFL's} \subset \text{CSL's}$$

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$  *(parenthesis languages)*
- $L = \{ w c w^r \mid w \in \Sigma^* \}$

Neither of these is a regular language *(nor an RE)*

But, this is a little subtle. You can construct DFA's for

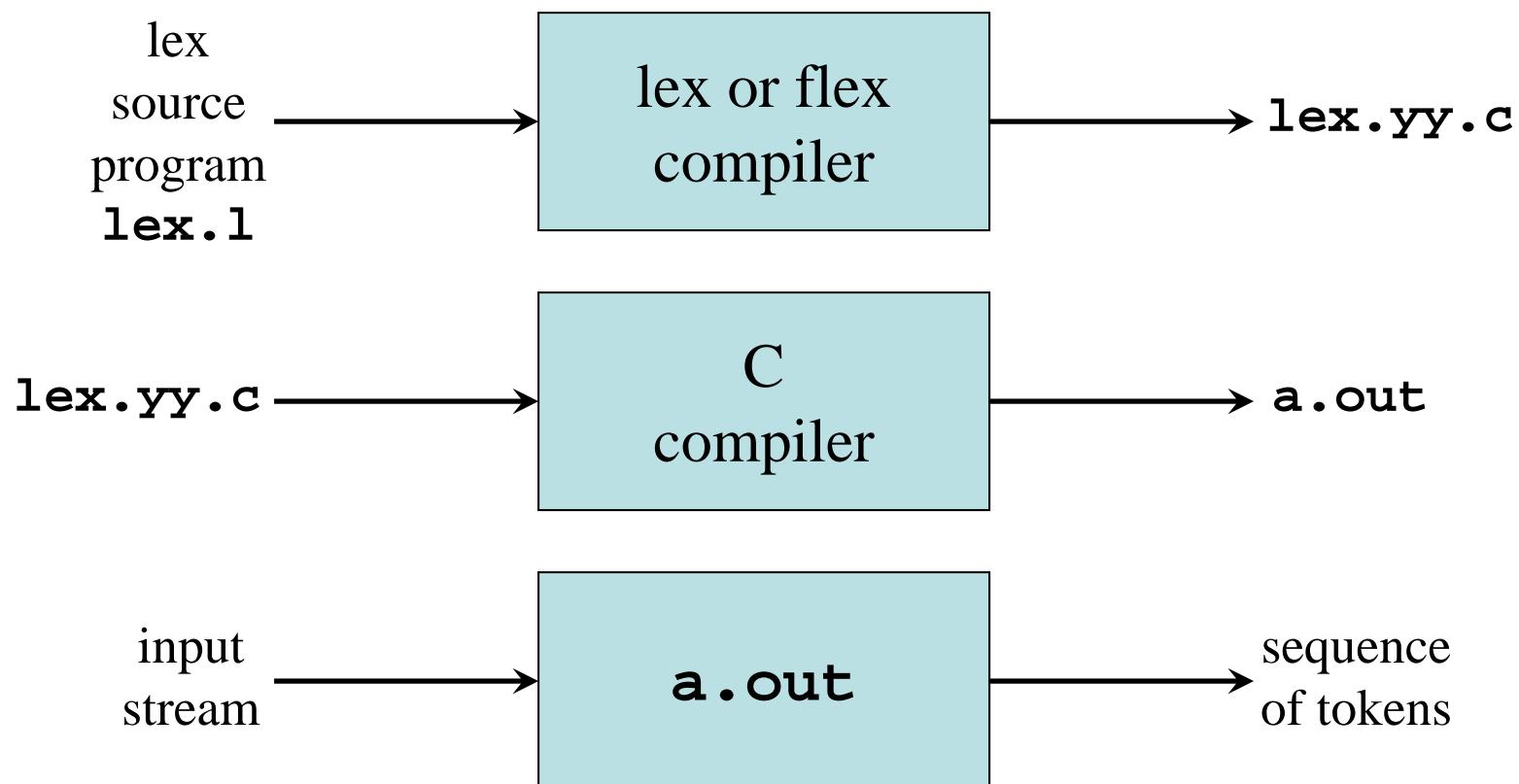
- Strings with alternating 0's and 1's  
 $(\epsilon \mid 1)(01)^*(\epsilon \mid 0)$
- Strings with an even number of 0's and 1's  
See Homework 1!

RE's can count bounded sets and bounded differences

# The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

# Creating a Lexical Analyzer with Lex and Flex



# Lex Specification

- A *lex specification* consists of three parts:  
*regular definitions*, *C declarations in %{ }*  
%%  
*translation rules*  
%%  
*user-defined auxiliary procedures*
- The *translation rules* are of the form:  
$$\begin{array}{ll} p_1 & \{ \textit{action}_1 \} \\ p_2 & \{ \textit{action}_2 \} \\ \dots & \\ p_n & \{ \textit{action}_n \} \end{array}$$

# Regular Expressions in Lex

<b>x</b>	match the character <b>x</b>
<b>\.</b>	match the character <b>.</b>
<b>"string"</b>	match contents of string of characters
<b>.</b>	match any character except newline
<b>^</b>	match beginning of a line
<b>\$</b>	match the end of a line
<b>[xyz]</b>	match one character <b>x</b> , <b>y</b> , or <b>z</b> (use \ to escape -)
<b>[^xyz]</b>	match any character except <b>x</b> , <b>y</b> , and <b>z</b>
<b>[a-z]</b>	match one of <b>a</b> to <b>z</b>
<b>r*</b>	closure (match zero or more occurrences)
<b>r+</b>	positive closure (match one or more occurrences)
<b>r?</b>	optional (match zero or one occurrence)
<b>r<sub>1</sub>r<sub>2</sub></b>	match <b>r<sub>1</sub></b> then <b>r<sub>2</sub></b> (concatenation)
<b>r<sub>1</sub>   r<sub>2</sub></b>	match <b>r<sub>1</sub></b> or <b>r<sub>2</sub></b> (union)
<b>( r )</b>	grouping
<b>r<sub>1</sub>\r<sub>2</sub></b>	match <b>r<sub>1</sub></b> when followed by <b>r<sub>2</sub></b>
<b>{d}</b>	match the regular expression defined by <b>d</b>

# Example Lex Specification 1

Translation  
rules

```
%{  
#include <stdio.h>  
%}  
%%  
[0-9]+ { printf("%s\n", yytext); }  
.|\n    { }  
%%  
main()  
{ yylex(); }  
}
```

Contains  
the matching  
lexeme

Invokes  
the lexical  
analyzer

```
lex spec.1  
gcc lex.yy.c -llex  
./a.out < spec.1
```

# Example Lex Specification 2

Translation  
rules

```
%{  
#include <stdio.h>  
int ch = 0, wd = 0, nl = 0;  
%}  
delim [ \t ]+  
%%  
\n { ch++; wd++; nl++; }  
^{delim} { ch+=yystrlen; }  
{delim} { ch+=yystrlen; wd++; }  
. { ch++; }  
%%  
main()  
{ yylex();  
printf("%8d%8d%8d\n", nl, wd, ch);  
}
```

Regular definition

# Example Lex Specification 3

Translation  
rules

```
%{  
#include <stdio.h>  
%}  
digit      [0-9]  
letter     [A-Za-z]  
id         {letter}({letter}|{digit})*  
%%  
{digit}+   { printf("number: %s\n", yytext); }  
{id}        { printf("ident: %s\n", yytext); }  
.          { printf("other: %s\n", yytext); }  
%%  
main()  
{  yylex();  
}
```

Regular definitions

# Example Lex Specification 4

```
%{ /* definitions of manifest constants */
#define LT (256)
...
%}
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id          {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
```

Return token to parser

```
{ws}        { }
if          {return IF;}
then         {return THEN;}
else         {return ELSE;}
{id}          {yyval = install_id(); return ID;}
{number}     {yyval = install_num(); return NUMBER;}
```

Token attribute

```
"<"        {yyval = LT; return RELOP;}
"<="       {yyval = LE; return RELOP;}
"="         {yyval = EQ; return RELOP;}
"<>"      {yyval = NE; return RELOP;}
">"        {yyval = GT; return RELOP;}
">="       {yyval = GE; return RELOP;}
```

Install **yytext** as identifier in symbol table

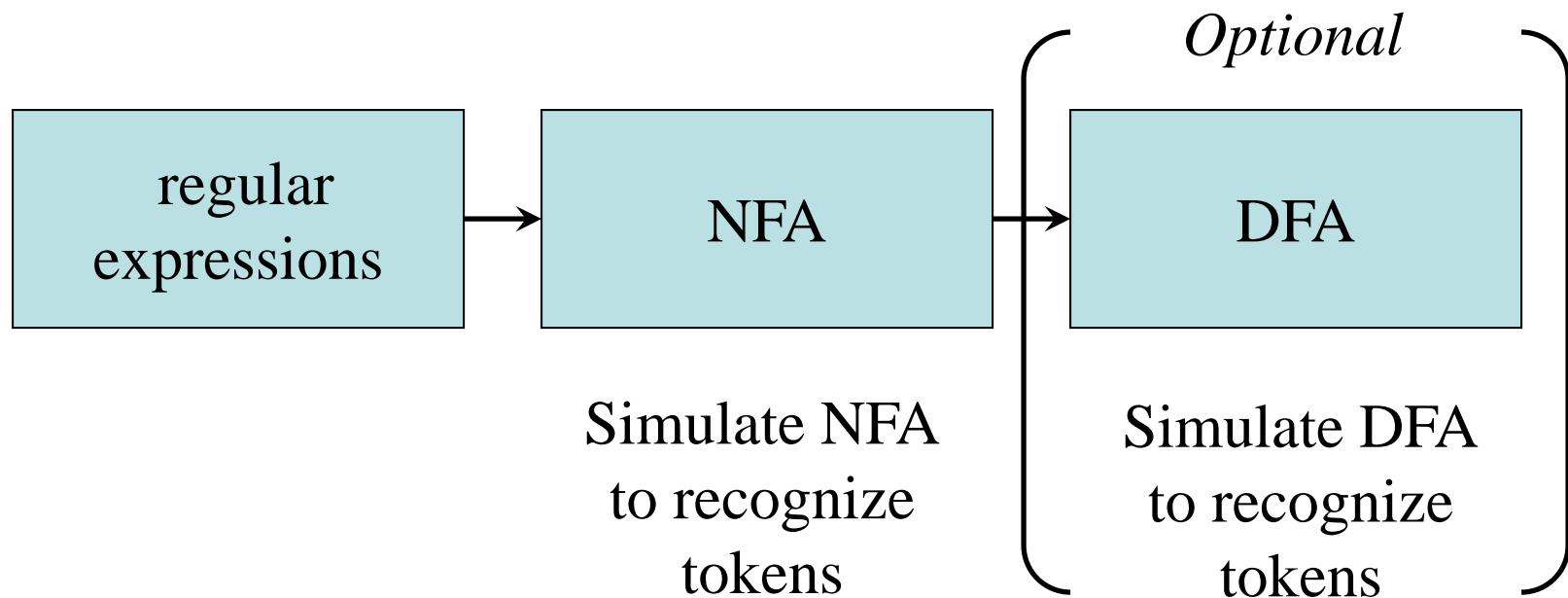
```
%%
```

```
int install_id()
```

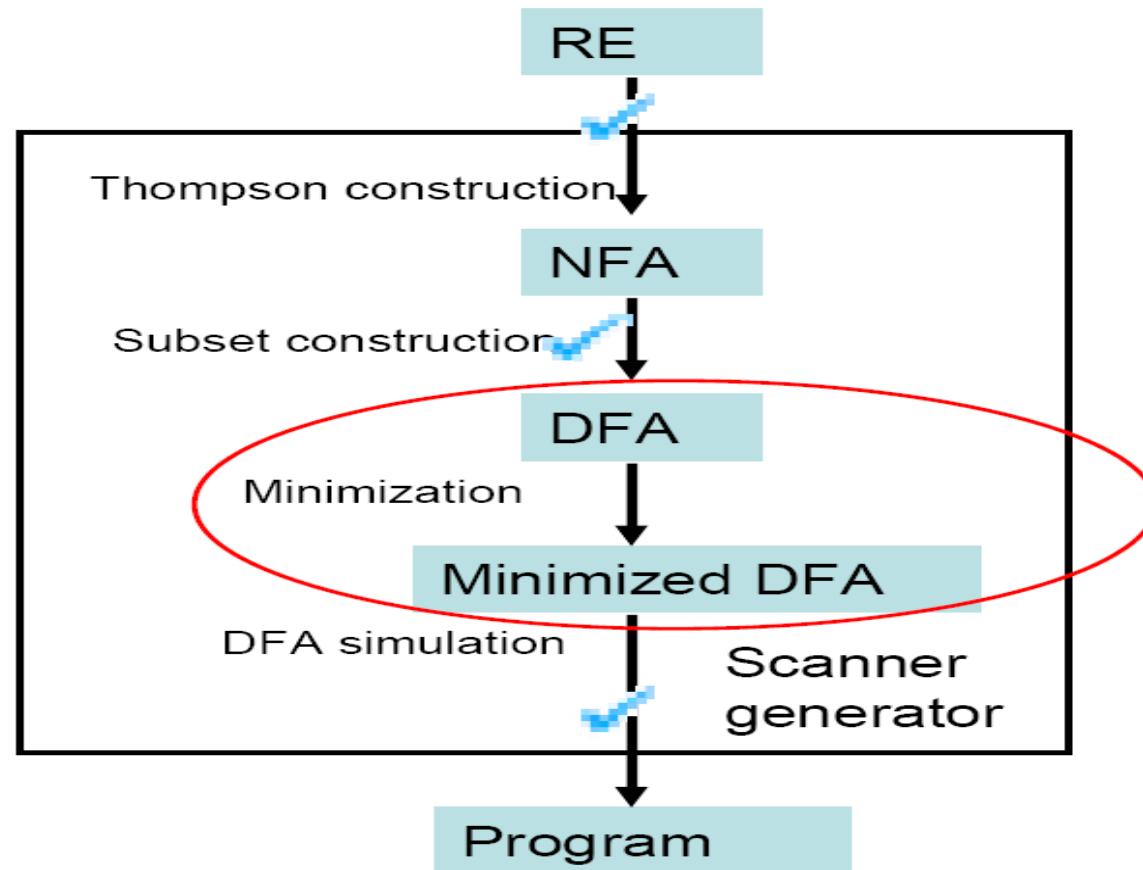
...

# Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA



# Lexical Analyzer Generator – Design

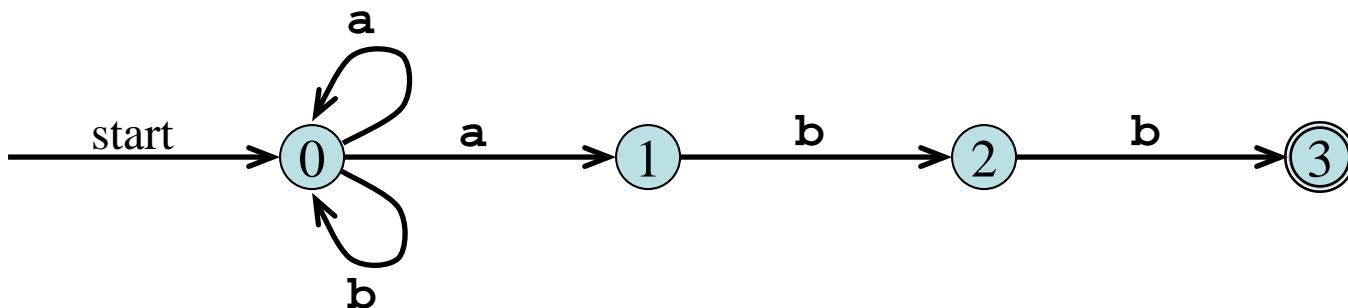


# Nondeterministic Finite Automata

- An NFA is a 5-tuple  $(S, \Sigma, \delta, s_0, F)$  where
  - $S$  is a finite set of *states*
  - $\Sigma$  is a finite set of symbols, the *alphabet*
  - $\delta$  is a *mapping* from  $S \times \Sigma$  to a set of states
  - $s_0 \in S$  is the *start state*
  - $F \subseteq S$  is the set of *accepting* (or *final*) *states*

# Transition Graph

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



$$\begin{aligned}S &= \{0,1,2,3\} \\ \Sigma &= \{\mathbf{a},\mathbf{b}\} \\ s_0 &= 0 \\ F &= \{3\}\end{aligned}$$

# Transition Table

- The mapping  $\delta$  of an NFA can be represented in a *transition table*

$$\begin{aligned}\delta(0,\mathbf{a}) &= \{0,1\} \\ \delta(0,\mathbf{b}) &= \{0\} \\ \delta(1,\mathbf{b}) &= \{2\} \\ \delta(2,\mathbf{b}) &= \{3\}\end{aligned}$$



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

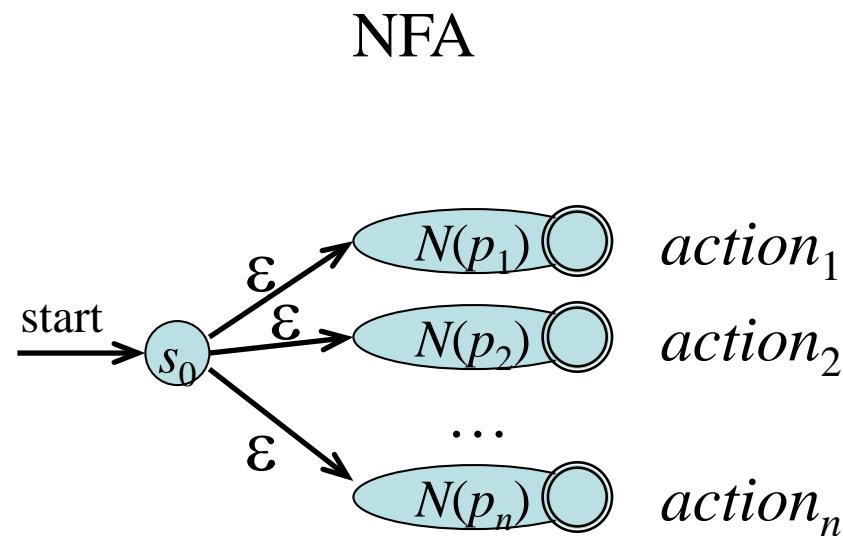
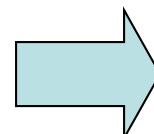
# The Language Defined by an NFA

- An NFA *accepts* an input string  $x$  if and only if there is some path with edges labeled with symbols from  $x$  in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as  $(\mathbf{a} \mid \mathbf{b})^* \mathbf{a} \mathbf{b} \mathbf{b}$  for the example NFA

# Design of a Lexical Analyzer Generator: RE to NFA to DFA

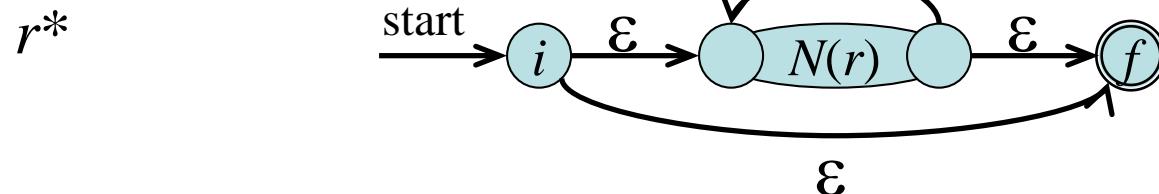
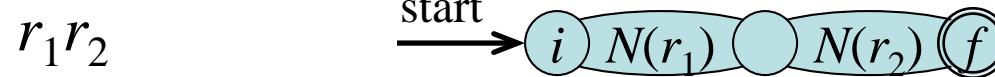
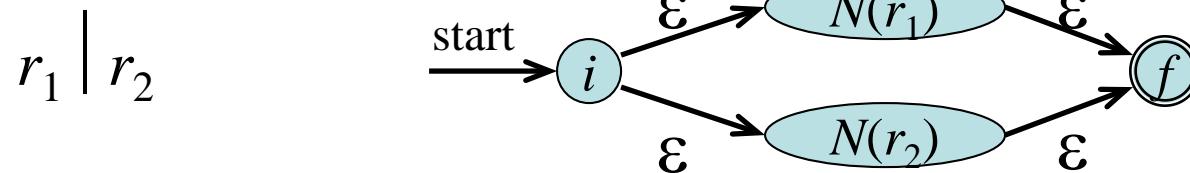
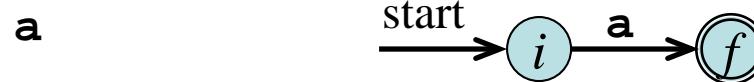
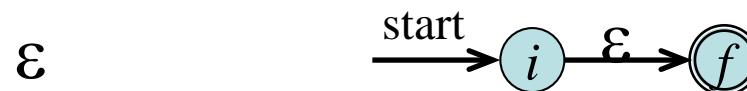
Lex specification with  
regular expressions

$p_1$	$\{ \text{action}_1 \}$
$p_2$	$\{ \text{action}_2 \}$
...	
$p_n$	$\{ \text{action}_n \}$

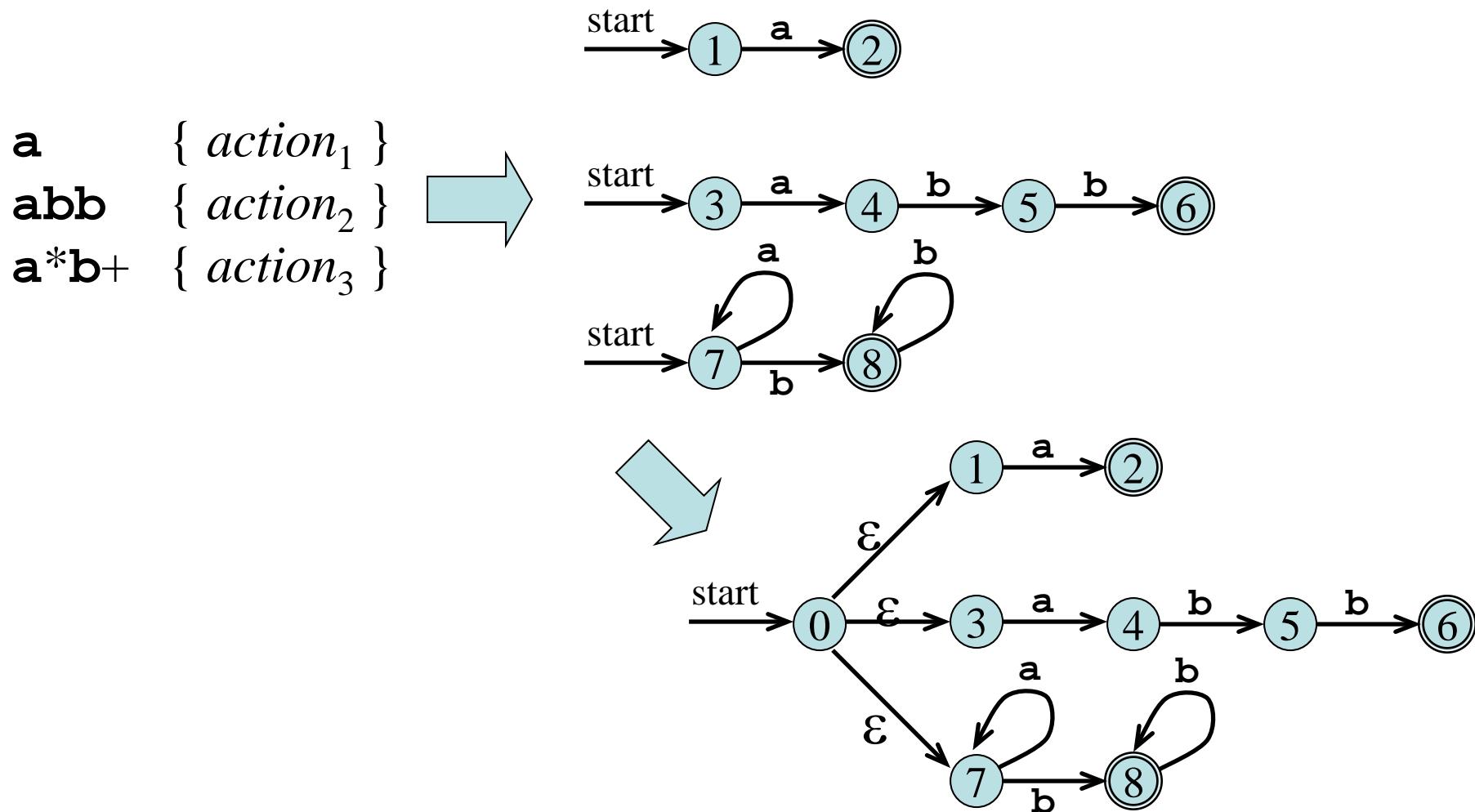


DFA

# From Regular Expression to NFA (Thompson's Construction)

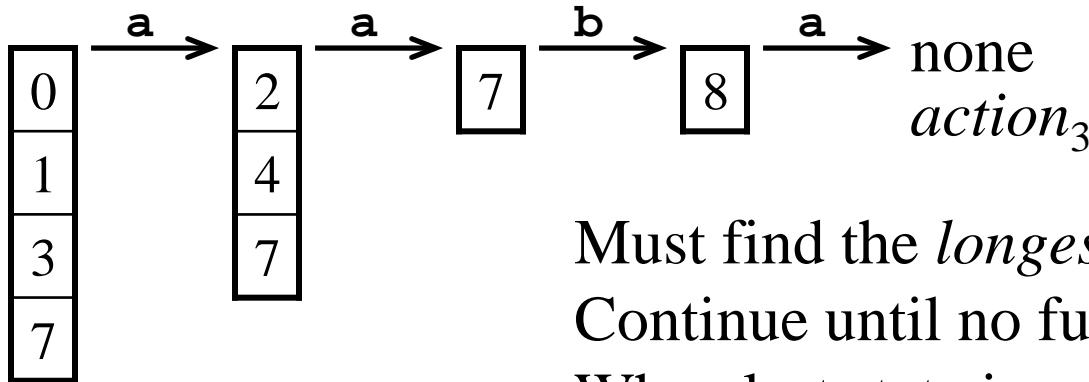
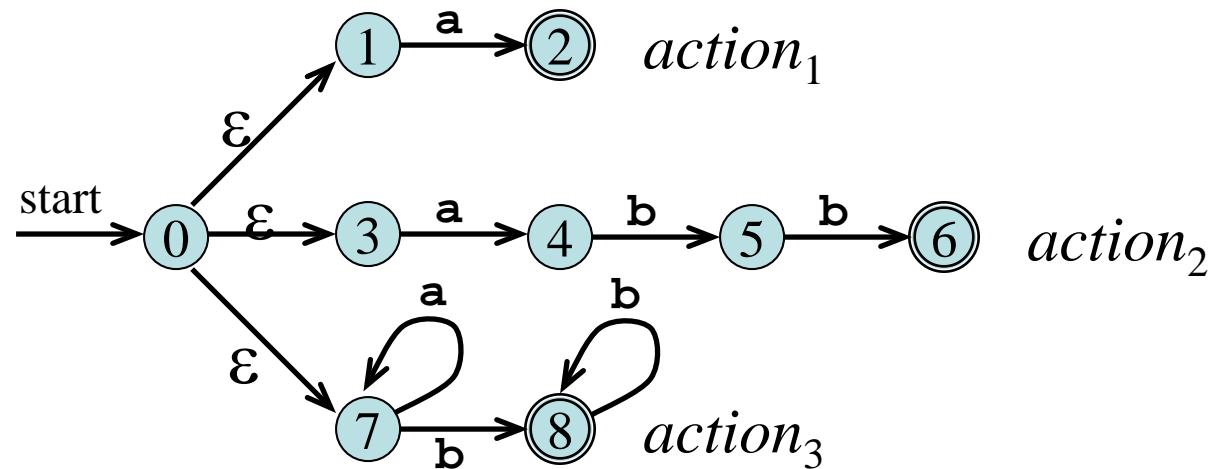


# Combining the NFAs of a Set of Regular Expressions



# Simulating the Combined NFA

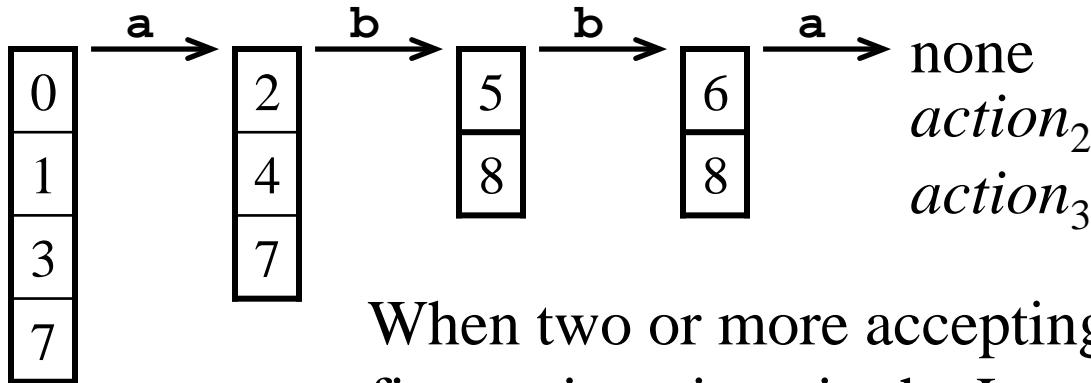
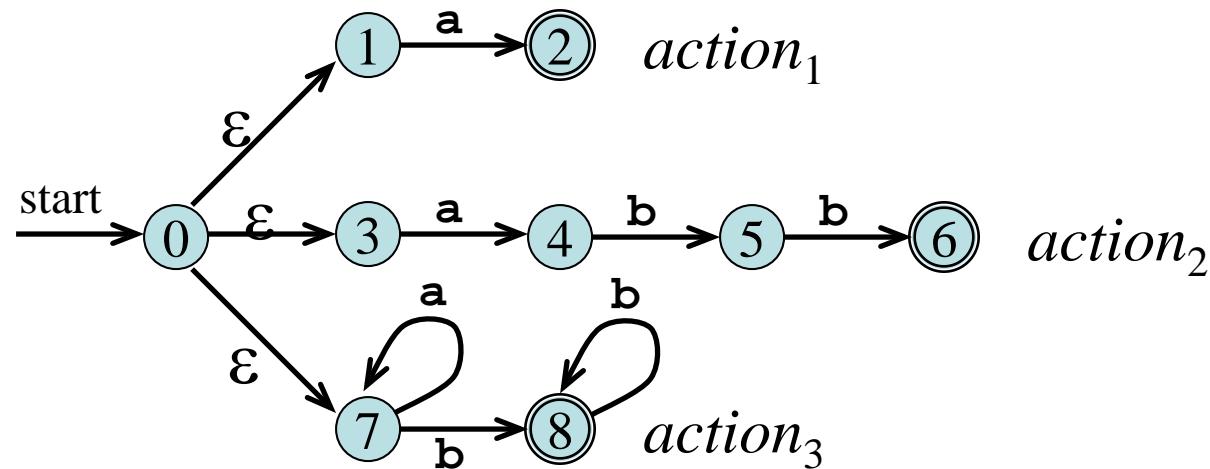
## Example 1



Must find the *longest match*:  
 Continue until no further moves are possible  
 When last state is accepting: execute action

# Simulating the Combined NFA

## Example 2



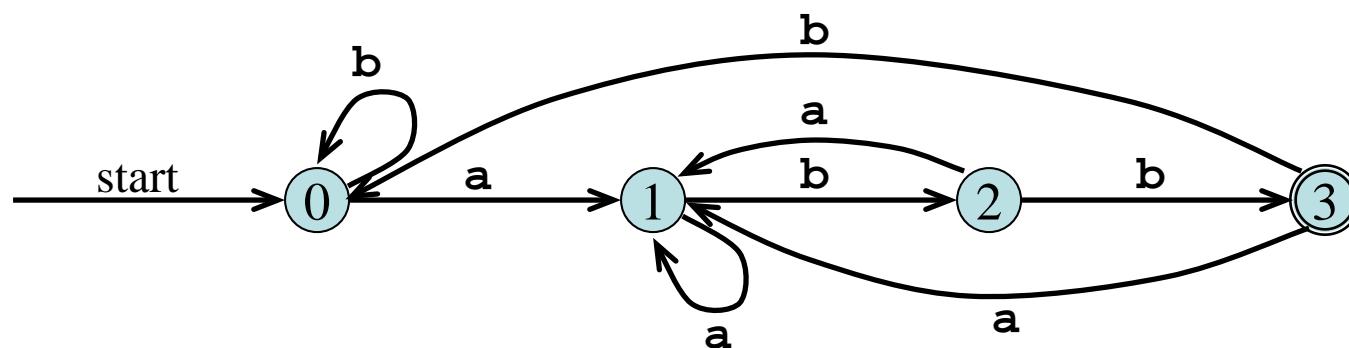
When two or more accepting states are reached, the first action given in the Lex specification is executed

# Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA
  - No state has an  $\varepsilon$ -transition
  - For each state  $s$  and input symbol  $a$  there is at most one edge labeled  $a$  leaving  $s$
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

# Example DFA

A DFA that accepts  $(a \mid b)^*abb$



# Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:

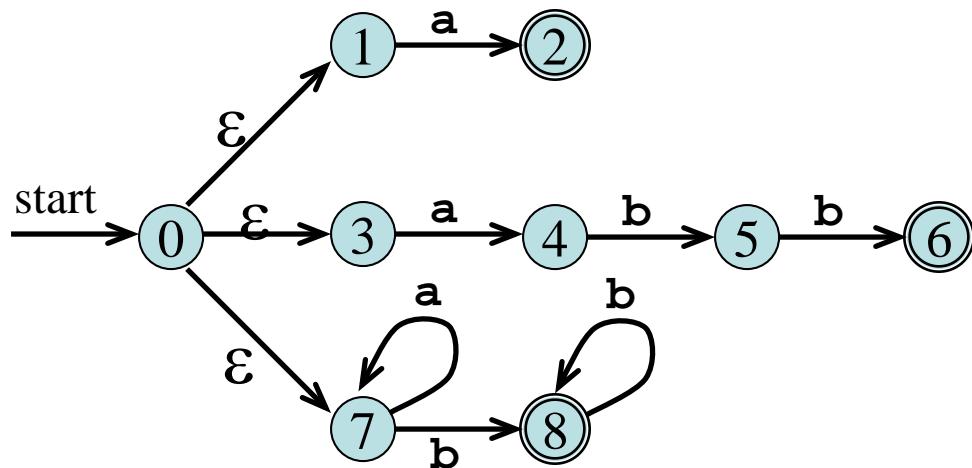
$$\varepsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} t\}$$

$$\varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s)$$

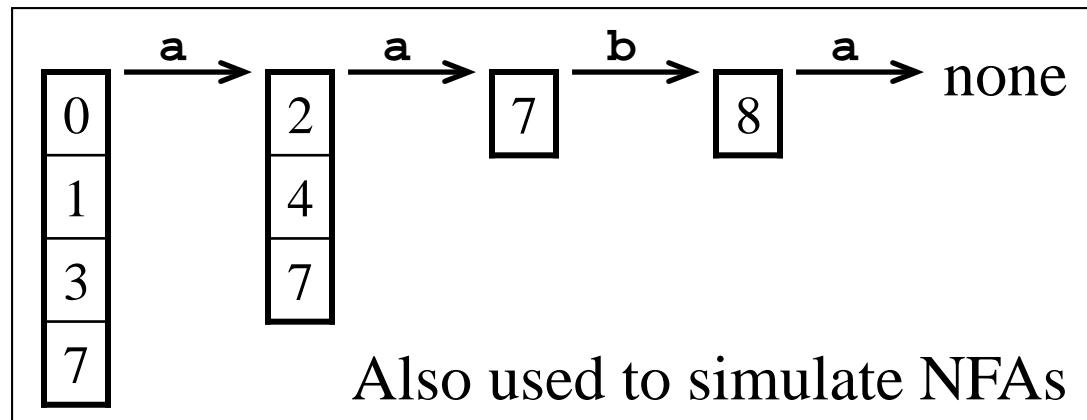
$$\text{move}(T, a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\}$$

- The algorithm produces:  
 $D_{states}$  is the set of states of the new DFA  
consisting of sets of states of the NFA  
 $D_{tran}$  is the transition table of the new DFA

# $\varepsilon$ -closure and move Examples



$\varepsilon\text{-closure}(\{0\}) = \{0,1,3,7\}$   
 $\text{move}(\{0,1,3,7\}, \mathbf{a}) = \{2,4,7\}$   
 $\varepsilon\text{-closure}(\{2,4,7\}) = \{2,4,7\}$   
 $\text{move}(\{2,4,7\}, \mathbf{a}) = \{7\}$   
 $\varepsilon\text{-closure}(\{7\}) = \{7\}$   
 $\text{move}(\{7\}, \mathbf{b}) = \{8\}$   
 $\varepsilon\text{-closure}(\{8\}) = \{8\}$   
 $\text{move}(\{8\}, \mathbf{a}) = \emptyset$



# Simulating an NFA using $\varepsilon$ -closure and move

```

 $S := \varepsilon\text{-closure}(\{s_0\})$ 
 $S_{prev} := \emptyset$ 
 $a := nextchar()$ 
while  $S \neq \emptyset$  do
     $S_{prev} := S$ 
     $S := \varepsilon\text{-closure}(move(S,a))$ 
     $a := nextchar()$ 
end do
if  $S_{prev} \cap F \neq \emptyset$  then
    execute action in  $S_{prev}$ 
    return “yes”
else return “no”

```

# The Subset Construction Algorithm

Initially,  $\varepsilon\text{-closure}(s_0)$  is the only state in  $Dstates$  and it is unmarked  
**while** there is an unmarked state  $T$  in  $Dstates$  **do**

    mark  $T$

**for** each input symbol  $a \in \Sigma$  **do**

$U := \varepsilon\text{-closure}(\text{move}(T,a))$

**if**  $U$  is not in  $Dstates$  **then**

            add  $U$  as an unmarked state to  $Dstates$

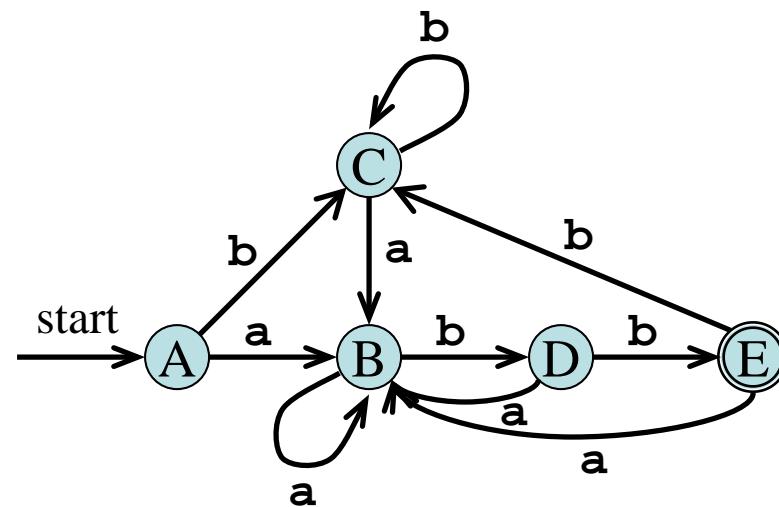
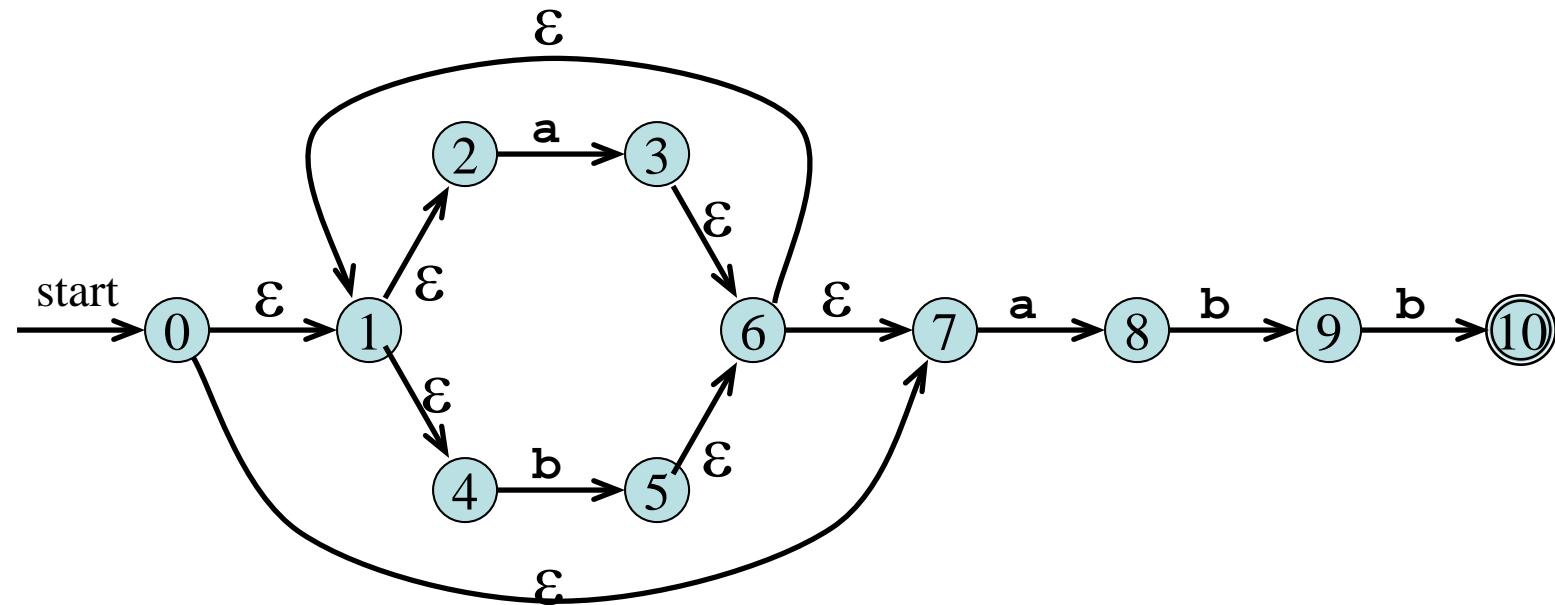
**end if**

$Dtran[T,a] := U$

**end do**

**end do**

# Subset Construction Example 1



*Dstates*

$$A = \{0,1,2,4,7\}$$

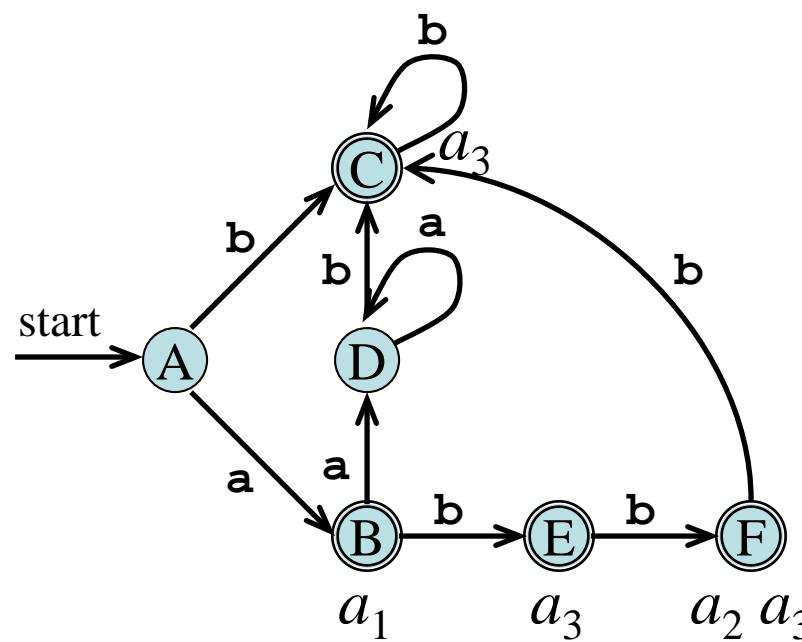
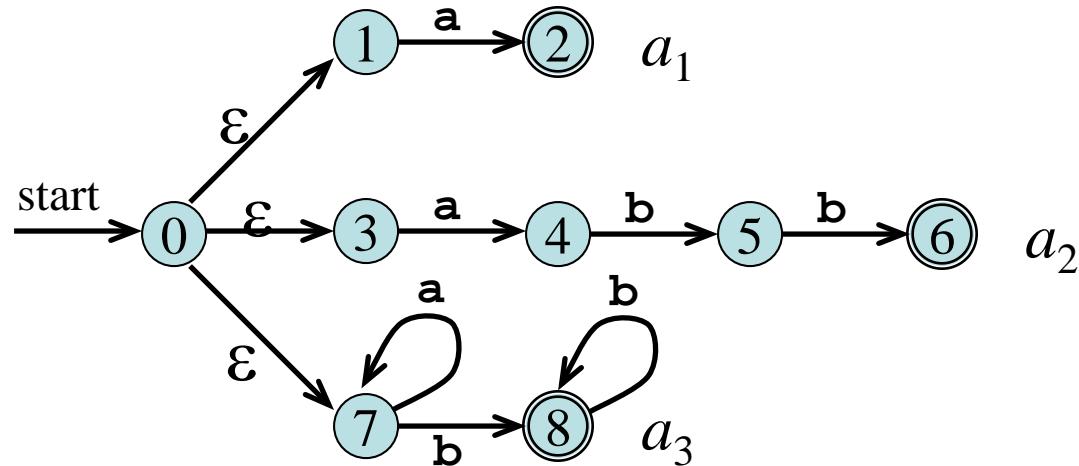
$$B = \{1,2,3,4,6,7,8\}$$

$$C = \{1,2,4,5,6,7\}$$

$$D = \{1,2,4,5,6,7,9\}$$

$$E = \{1,2,4,5,6,7,10\}$$

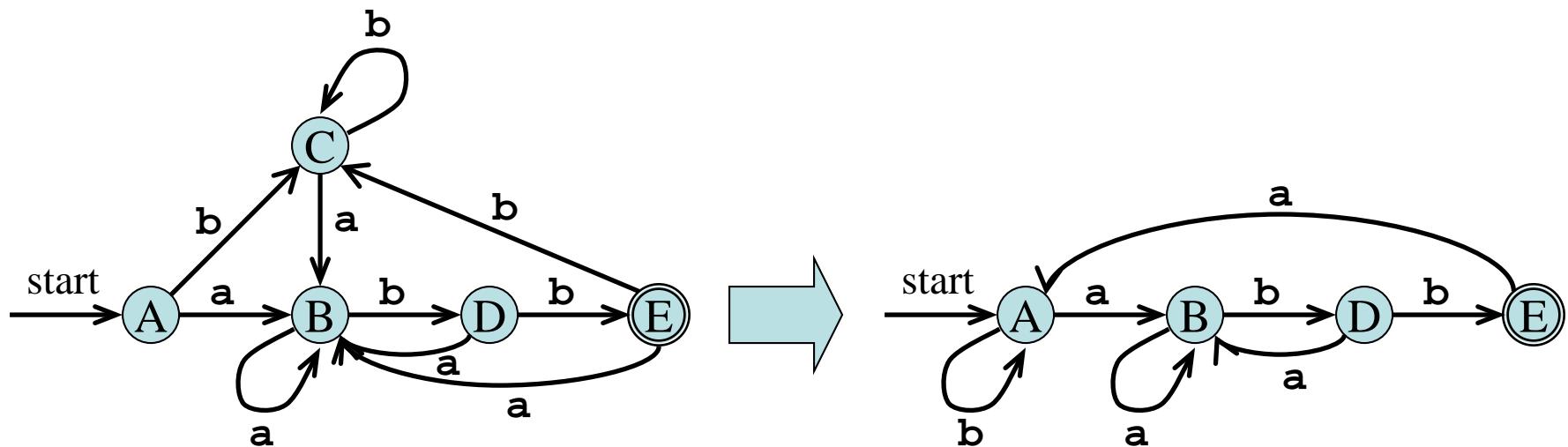
# Subset Construction Example 2



*Dstates*

- $A = \{0,1,3,7\}$
- $B = \{2,4,7\}$
- $C = \{8\}$
- $D = \{7\}$
- $E = \{5,8\}$
- $F = \{6,8\}$

# Minimizing the Number of States of a DFA (Hopcroft's algorithm)



# From Regular Expression to DFA Directly

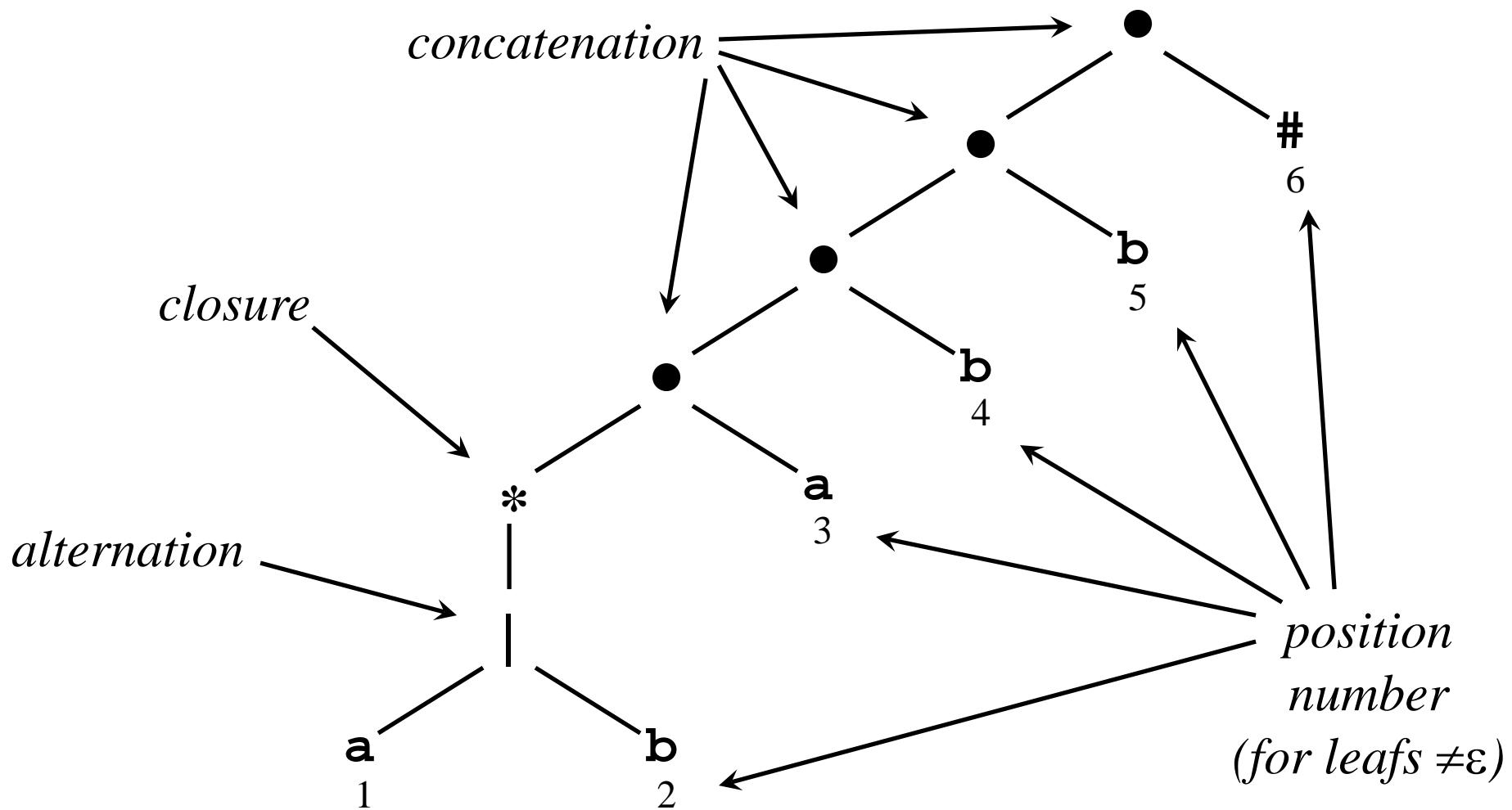
- The “*important states*” of an NFA are those without an  $\epsilon$ -transition, that is if  $move(\{s\}, a) \neq \emptyset$  for some  $a$  then  $s$  is an important state
- The subset construction algorithm uses only the important states when it determines  $\epsilon$ -closure( $move(T, a)$ )

# From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression  $r$  with a special end symbol  $\#$  to make accepting states important: the new expression is  $r\#$
- Construct a syntax tree for  $r\#$
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

# From Regular Expression to DFA

## Directly: Syntax Tree of $(a|b)^*abb\#$



# From Regular Expression to DFA Directly: Annotating the Tree

- $\text{nullable}(n)$ : the subtree at node  $n$  generates languages including the empty string
- $\text{firstpos}(n)$ : set of positions that can match the first symbol of a string generated by the subtree at node  $n$
- $\text{lastpos}(n)$ : the set of positions that can match the last symbol of a string generated be the subtree at node  $n$
- $\text{followpos}(i)$ : the set of positions that can follow position  $i$  in the tree

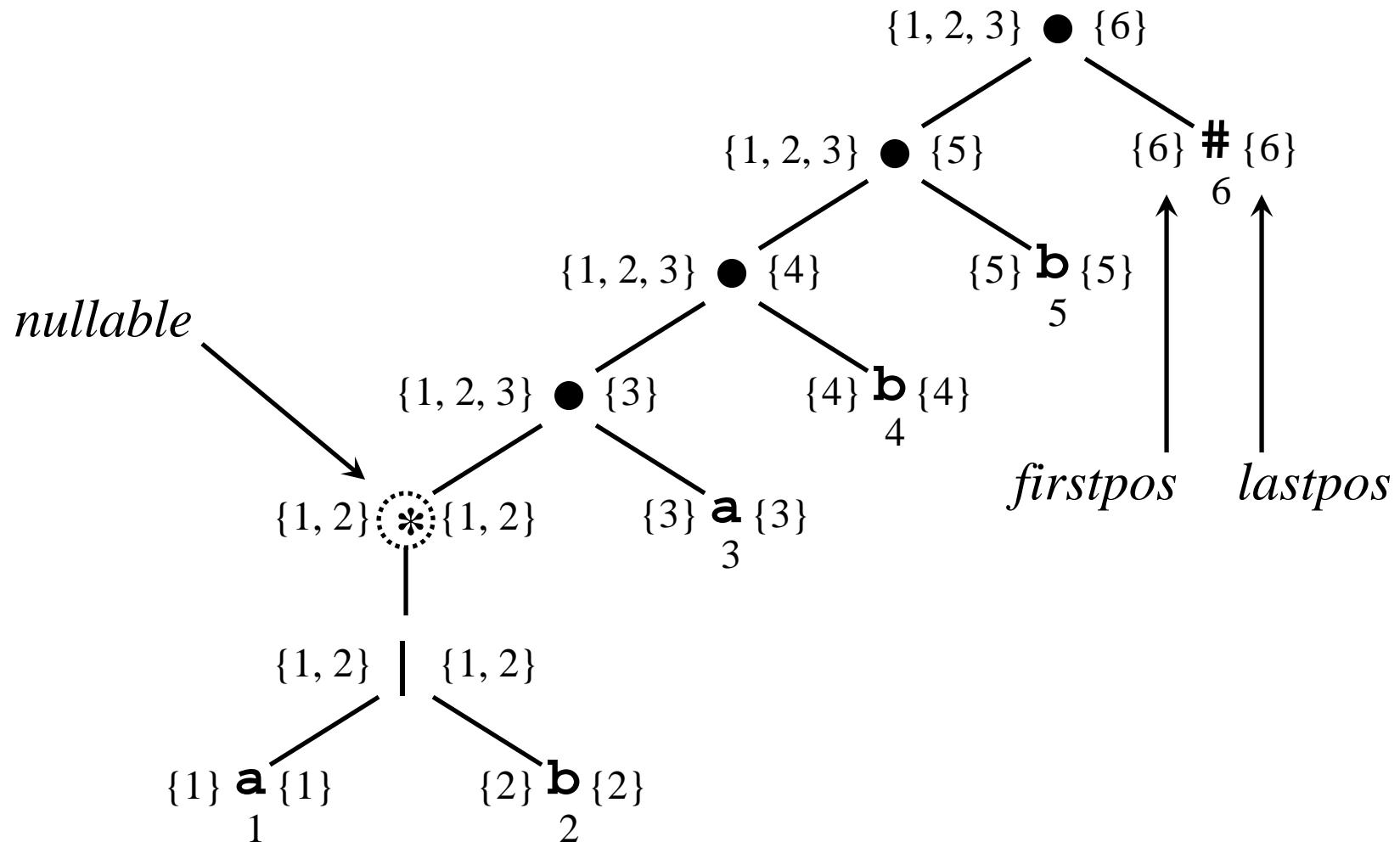
# From Regular Expression to DFA

## Directly: Annotating the Tree

Node $n$	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
Leaf $\epsilon$	true	$\emptyset$	$\emptyset$
Leaf $i$	false	$\{i\}$	$\{i\}$
$\begin{array}{c}   \\ / \backslash \\ c_1 \quad c_2 \end{array}$	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1)$ $\cup$ $\text{firstpos}(c_2)$	$\text{lastpos}(c_1)$ $\cup$ $\text{lastpos}(c_2)$
$\begin{array}{c} \bullet \\ / \backslash \\ c_1 \quad c_2 \end{array}$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	<b>if</b> $\text{nullable}(c_1)$ <b>then</b> $\text{firstpos}(c_1) \cup$ $\text{firstpos}(c_2)$ <b>else</b> $\text{firstpos}(c_1)$	<b>if</b> $\text{nullable}(c_2)$ <b>then</b> $\text{lastpos}(c_1) \cup$ $\text{lastpos}(c_2)$ <b>else</b> $\text{lastpos}(c_2)$
$\begin{array}{c} * \\   \\ c_1 \end{array}$	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

# From Regular Expression to DFA

## Directly: Syntax Tree of $(a|b)^*abb\#$



# From Regular Expression to DFA

## Directly: *followpos*

```
for each node  $n$  in the tree do
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
        for each  $i$  in  $\text{lastpos}(c_1)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(c_2)$ 
        end do
    else if  $n$  is a star-node
        for each  $i$  in  $\text{lastpos}(n)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(n)$ 
        end do
    end if
end do
```

# From Regular Expression to DFA

## Directly: Algorithm

$s_0 := \text{firstpos}(\text{root})$  where  $\text{root}$  is the root of the syntax tree

$D\text{states} := \{s_0\}$  and is unmarked

**while** there is an unmarked state  $T$  in  $D\text{states}$  **do**

mark  $T$

**for** each input symbol  $a \in \Sigma$  **do**

let  $U$  be the set of positions that are in  $\text{followpos}(p)$

for some position  $p$  in  $T$ ,

such that the symbol at position  $p$  is  $a$

**if**  $U$  is not empty and not in  $D\text{states}$  **then**

add  $U$  as an unmarked state to  $D\text{states}$

**end if**

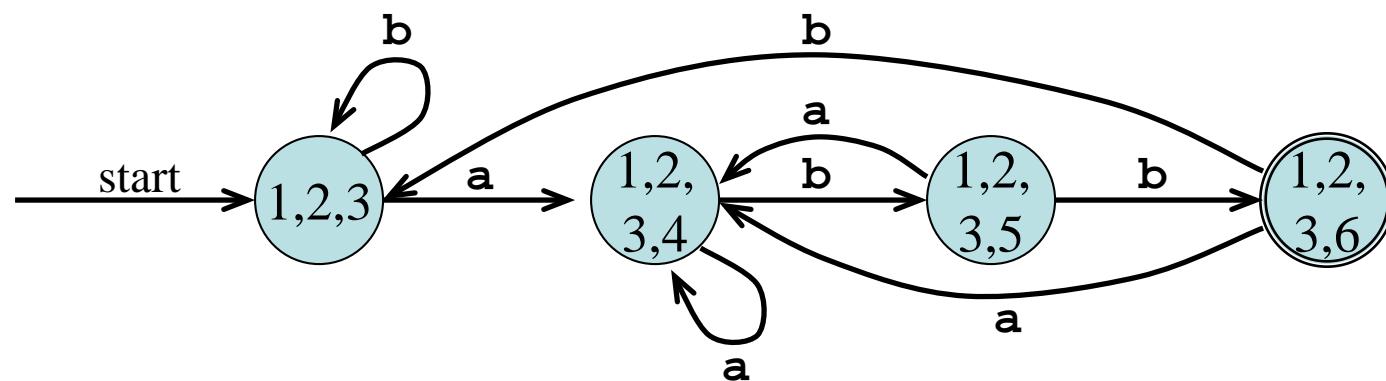
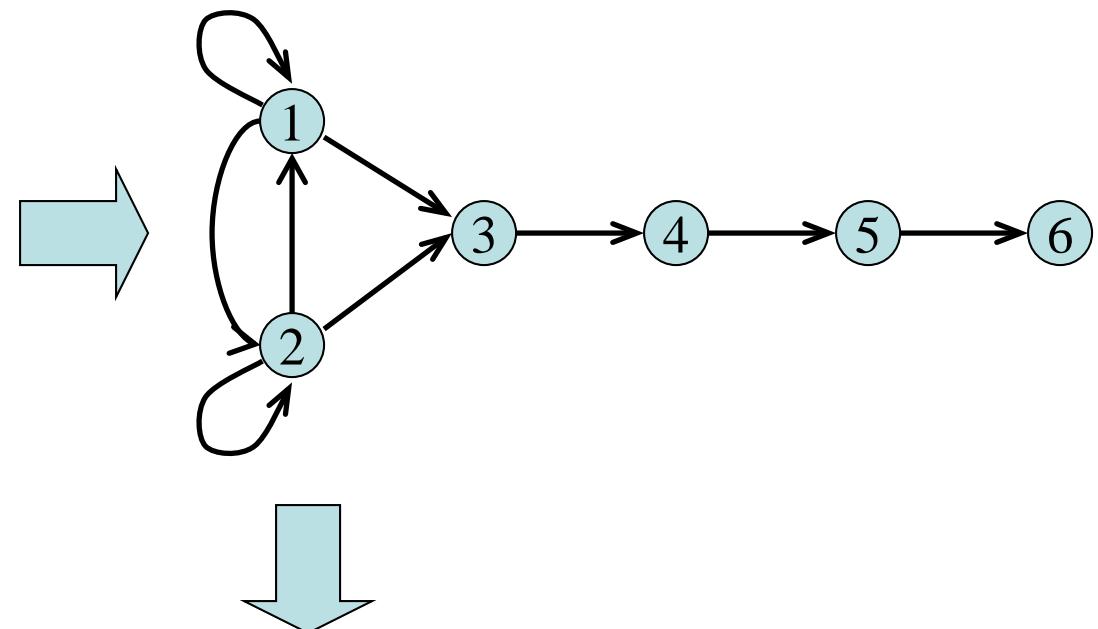
$D\text{tran}[T,a] := U$

**end do**

**end do**

# From Regular Expression to DFA Directly: Example

Node	<i>followpos</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-



# Time-Space Tradeoffs

<i>Automaton</i>	<i>Space (worst case)</i>	<i>Time (worst case)</i>
NFA	$O( r )$	$O( r  \times  x )$
DFA	$O(2^{ r })$	$O( x )$

## DFA Minimization

---

### The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall \alpha \in \Sigma$ , transitions on  $\alpha$  lead to equivalent states (DFA)
- $\alpha$ -transitions to distinct sets  $\Rightarrow$  states must be in distinct sets

A partition  $P$  of  $S$

- Each  $s \in S$  is in exactly one set  $p_i \in P$
- The algorithm iteratively partitions the DFA's states

# John Hopcroft

Initial partition,  $P_0$ , has two sets:  $\{F\}$  &  $\{Q-F\}$       ( $D = (Q, \Sigma, \delta, q_0, F)$ )

Splitting a set ("partitioning a set by  $\underline{a}$ ")

- Assume  $q_a$  &  $q_b \in s$ , and  $\delta(q_a, \underline{a}) = q_x$  &  $\delta(q_b, \underline{a}) = q_y$
- If  $q_x$  &  $q_y$  are not in the same set, then  $s$  must be split
- One state in the final DFA cannot have two transitions on  $\underline{a}$

# DFA minimization: The idea

- **Equivalent states** : Two states  $q$  and  $q'$  in a DFA  $M = (Q, \Sigma, \delta, q_0, F)$  are said to be equivalent if for all strings  $u$  in  $\Sigma^*$ , the states on which  $u$  ends on when read from  $q$  and  $q'$  are both accept, or both non-accept.
- Remove *unreachable states* from start state.
- Remove *dead states*: states that are not final and have transitions to themselves

## The algorithm

- Input: DFA,  $S$  is the set of states,  $F$  is the set of final states.
- Output: minimized equivalent DFA.
- Steps:

$\Pi = (F) (S - F);$

While ( $\Pi$  is changed) {

for each group  $G$  of  $\Pi$  do {

partition  $G$  if there are distinguishable states in  $G$ ;

replace  $G$  by the subgroups found;

}

}

Choose representative state for each group;

Remove dead states;

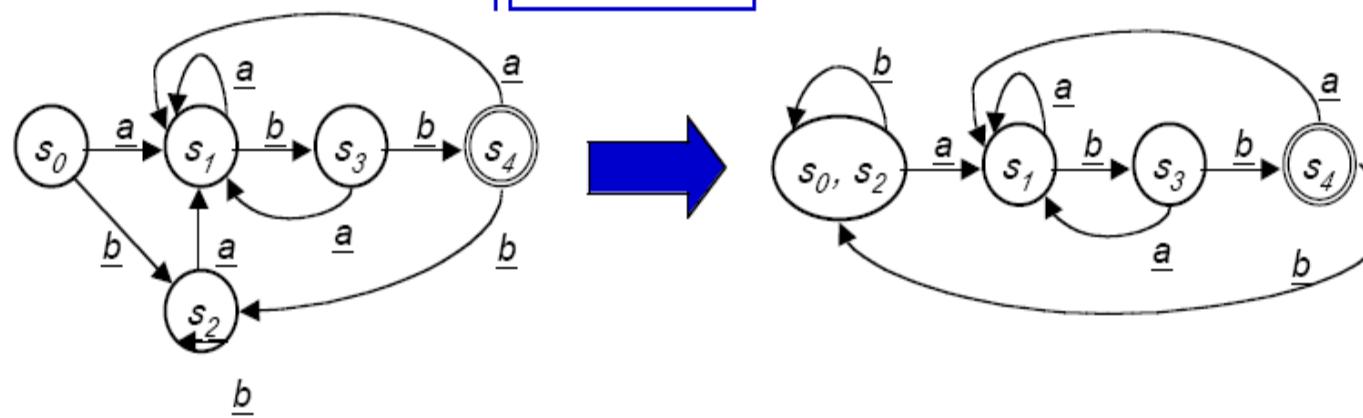
Remove states not reachable from the start state;

## A Detailed Example

Applying the minimization algorithm to the DFA

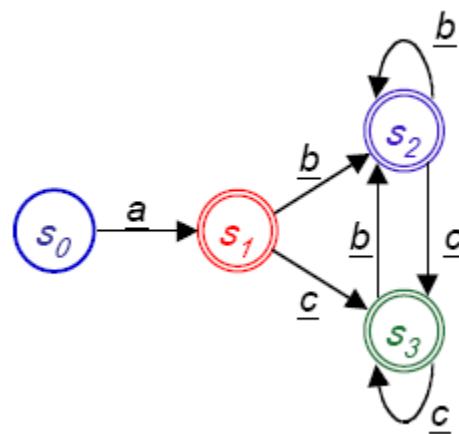
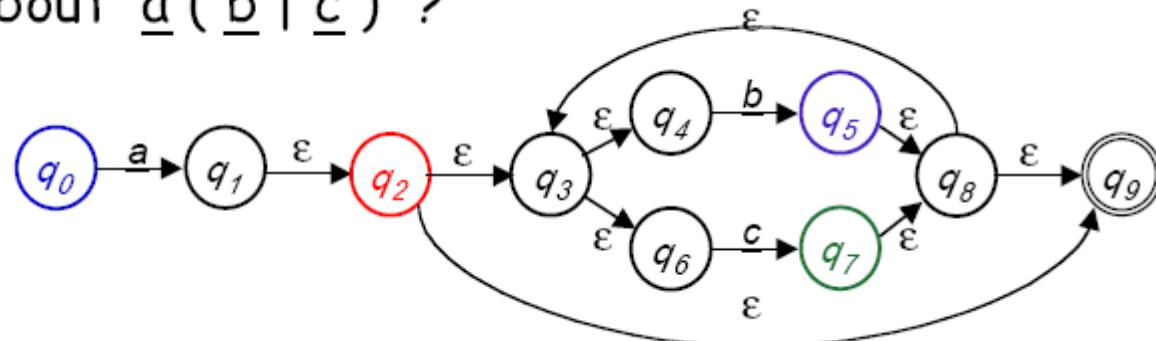
	<i>Current Partition</i>	<i>Worklist</i>	<i>s</i>	<i>Split on <u>a</u></i>	<i>Split on <u>b</u></i>
$P_0$	$\{s_4\} \{s_0, s_1, s_2, s_3\}$	$\{s_4\}$ $\{s_0, s_1, s_2, s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
$P_1$	$\{s_4\} \{s_3\} \{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\} \{s_1\}$
$P_2$	$\{s_4\} \{s_3\} \{s_1\} \{s_0, s_2\}$	$\{s_0, s_2\} \{s_1\}$	$\{s_1\}$	none	none

*final state*

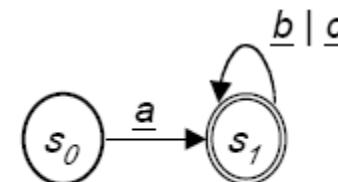


# Example

What about  $\underline{a}(\underline{b} \mid \underline{c})^*$ ?

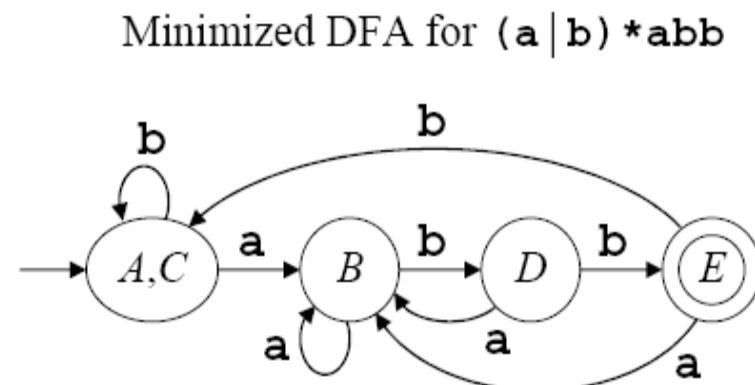
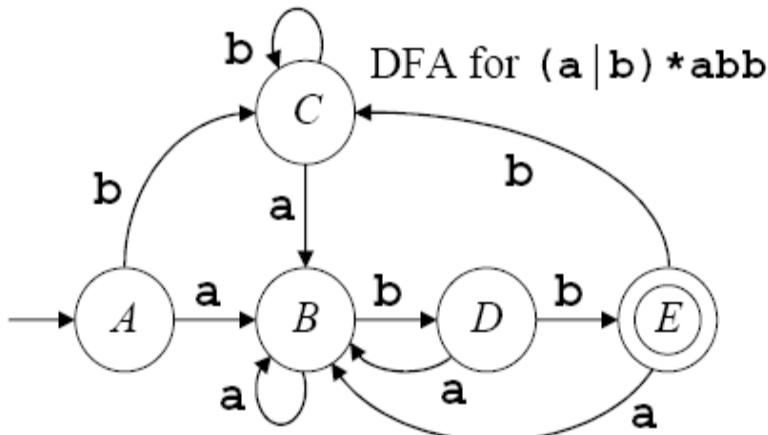


To produce the minimal DFA



# Example on DFA Minimization

- ❖ Consider the DFA for  $(a|b)^*abb$  obtained using subset construction algorithm
- ❖ Initial partition  $\Pi$  consists of 2 groups =  $\{\{A, B, C, D\}, \{E\}\}$
- ❖  $\{A, B, C\}$ -succ under  $b \in \{A, B, C, D\}$ , while  $D$ -succ under  $b$  is  $E$
- ❖ Therefore,  $\Pi_{\text{new}} = \{\{A, B, C\}, \{D\}, \{E\}\}$
- ❖  $\{A, C\}$ -succ under  $b$  is  $C$  while  $B$ -succ under  $b$  is  $D$
- ❖ Therefore,  $\Pi_{\text{new}} = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$
- ❖  $\{A, C\}$ -succ under  $a$  is  $B$ , and  $\{A, C\}$ -succ under  $b$  is  $C$
- ❖  $\{A, C\}$  does not require further partitioning; states  $A$  and  $C$  can be merged
- ❖ Therefore, final  $\Pi = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$



## Building Faster Scanners from the DFA

Table-driven recognizers waste effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in *action()*
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

```

char ← next character;
state ←  $s_0$ ;
call action(state,char);
while (char ≠ eof)
    state ←  $\delta(state,char)$ ;
    call action(state,char);
    char ← next character;

if  $T(state)$  = final then
    report acceptance;
else
    report failure;
```

# Building Faster Scanners from the DFA

A direct-coded recognizer for  $\underline{r} \text{ Digit } \text{Digit}^*$

```
goto  $s_0$ ;
 $s_0$ : word  $\leftarrow \emptyset$ ;
    char  $\leftarrow$  next character;
    if (char = 'r')
        then goto  $s_1$ ;
    else goto  $s_e$ ;
 $s_1$ : word  $\leftarrow$  word + char;
    char  $\leftarrow$  next character;
    if ('0'  $\leq$  char  $\leq$  '9')
        then goto  $s_2$ ;
    else goto  $s_e$ ;
 $s_2$ : word  $\leftarrow$  word + char;
    char  $\leftarrow$  next character;
    if ('0'  $\leq$  char  $\leq$  '9')
        then goto  $s_2$ ;
    else if (char = eof)
        then report success;
    else goto  $s_e$ ;
 $s_e$ : print error message;
        return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

# Minimizing the Number of States in a DFA

Smaller is better!

# Minimal DFA

- Given any DFA, there is an equivalent DFA containing the minimum number of states
- The minimal DFA is unique
- It is possible to directly obtain the minimal DFA from any DFA
- The algorithm presented here is adapted from Aho, Sethi, and Ullman.

# Minimal DFA Algorithm

## 1

- The algorithm starts by partitioning the states in the DFA into sets of states that will ultimately be combined into single states.
- The first partitioning creates 2 sets:
  - One set contains all the accepting states
  - The other set contains all the nonaccepting states
- The process now goes through one or more iterations where it considers the transitions on each character of the alphabet

# Minimal DFA Algorithm

## 2

- Iterate until no further partitioning is possible:
  - For each set  $G$  of states in partition  $\Pi$ , consider the transitions for each input symbol  $a$  from any state in  $G$ .
  - Two states  $s$  and  $t$  belong in the same subgroup iff for all input symbols  $a$ , states  $s$  and  $t$  have transitions into states in the same subgroup of  $\Pi$ .
  - Replace  $G$  in  $\Pi$  by the set of subgroups formed.

# Minimal DFA Algorithm

3

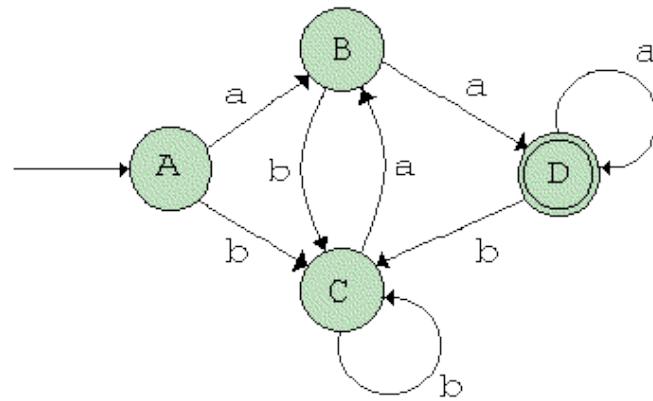
- Choose one state in each group of the partition  $\Pi$  as the representative for that group. The representatives will be the states of the reduced DFA  $M'$ .
- The start state of  $M'$  will be the group that contains the start state of the original DFA.
- Any group that contains an accepting state from the original DFA will be an accepting state of the minimal DFA  $M'$ .

# Minimal DFA Algorithm

4

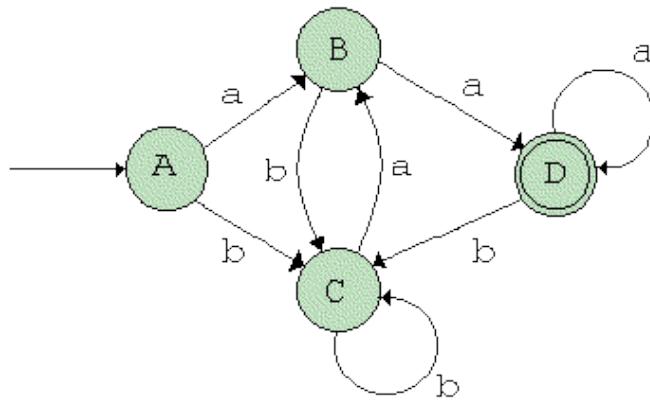
- Remove any dead state **d** from  $M'$ .
  - a dead state is one that has transitions to itself on all input symbols.
  - Any transitions to **d** from other states become undefined.
- Remove any states unreachable from the start state from  $M'$ .

# Example 1: Minimize the DFA



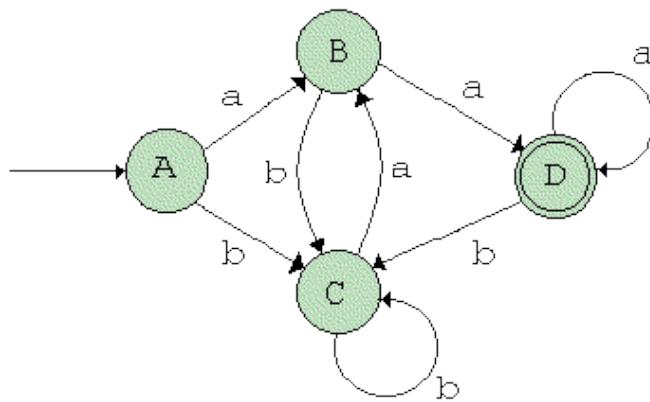
- We start with two groups
  - Accepting states: { D }
  - Nonaccepting states: { A, B, C }
- Since the singleton set { D } cannot be partitioned any further, we concentrate on { A, B, C }

# Example 1, continued



- For input  $a$  and states in group  $\{ A, B, C \}$ 
  - $T(A, a) = B$
  - $T(B, a) = D$  (maps into a different subgroup)
  - $T(C, a) = B$
- We must split the group  $\{ A, B, C \}$  into two subgroups,  $\{ A, C \}$  and  $\{ B \}$

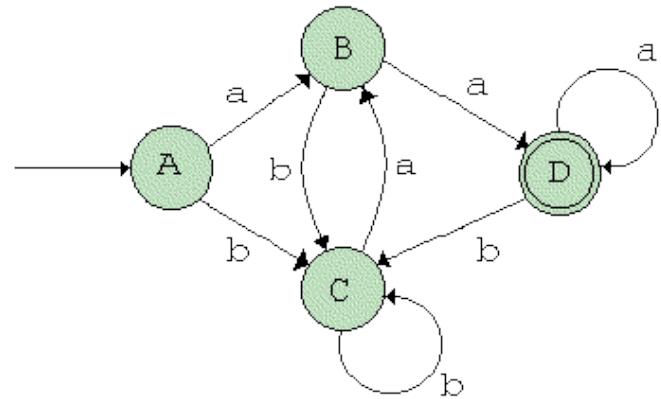
# Example 1, continued



- At this point,  $\Pi = \{ A, C \}, \{ B \}, \{ D \}$
- Consider transitions from  $\{ A, C \}$  on **a** and **b**
  - $T(A, a) = B$   $T(C, a) = B$  (same group)
  - $T(A, b) = C$   $T(C, b) = C$  (same group)
- No further partitioning is possible.

# Example 1, continued

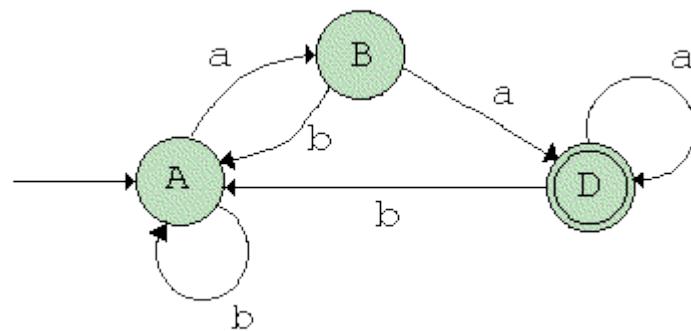
- $\Pi = \{A,C\}, \{B\}, \{D\}$
- Choose A as representative from  $\{A,C\}$ :
  - Remove row C from table
  - Replace all occurrences of C with A
- Resulting minimal DFA is shown on next slide



	a	b	
A	B	C	start
B	D	C	
C	B	C	
D	D	C	accept

# Example 1, conclusion

- Minimal DFA



	a	b	
A	B	A	start
B	D	A	
D	D	A	accept

# Checkpoint: Minimize the DFA

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- $\Pi = \{A, B, C, D, F, G\}, \{E, H\}$
- $T(\{A, B, C, D, F, G\}, a)$ :
  - $T(A, a) = G$
  - $T(B, a) = C$
  - $T(C, a) = B$
  - $T(D, a) = G$
  - $T(F, a) = A$
  - $T(G, a) = B$
  - all map to same group—no repartitioning (yet)

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- $\Pi = \{A, B, C, D, F, G\}, \{E, H\}$
- $T(\{A, B, C, D, F, G\}, b)$ :
  - $T(A, b) = F$
  - $T(B, b) = G$
  - $T(C, b) = D$
  - $T(D, b) = E$  (different group)
  - $T(F, b) = D$
  - $T(G, b) = D$
- Partition  $\{A, B, C, D, F, G\}$  into  $\{A, B, C, F, G\}, \{D\}$

# Checkpoint Solution

	a	b
A	G	F
B	C	G
C	B	D
D	G	E
E	B	H
F	A	D
G	B	D
H	A	E

start      accept      accept

- $\Pi = \{A, B, C, F, G\}, \{D\}, \{E, H\}$
- $T(\{E, H\}, a)$ :
  - $T(E, a) = B$
  - $T(H, a) = A$
  - map into same group
- $T(\{E, H\}, b)$ :
  - $T(E, b) = H$
  - $T(H, b) = E$
  - map into same group
- No repartitioning necessary  
(at least not yet)

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- $\Pi = \{A, B, C, F, G\}, \{D\}, \{E, H\}$
- $T(\{A, B, C, F, G\}, a):$ 
  - $T(A, a) = G$
  - $T(B, a) = C$
  - $T(C, a) = B$
  - $T(F, a) = A$
  - $T(G, a) = B$
  - all map to same group, so no repartitioning results from this

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- $\Pi = \{A, B, C, F, G\}, \{D\}, \{E, H\}$
- $T(\{A, B, C, F, G\}, b)$ :
  - $T(A, b) = F$
  - $T(B, b) = G$
  - $T(C, b) = D$
  - $T(F, b) = D$
  - $T(G, b) = D$
  - $\{A, B\}$  and  $\{C, F, G\}$  map to different groups, so we repartition  $\{A, B, C, F, G\}$  into the groups  $\{A, B\}$  and  $\{C, F, G\}$

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

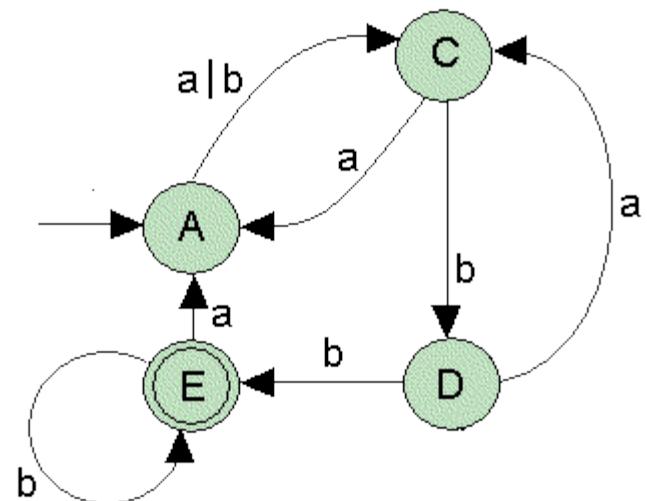
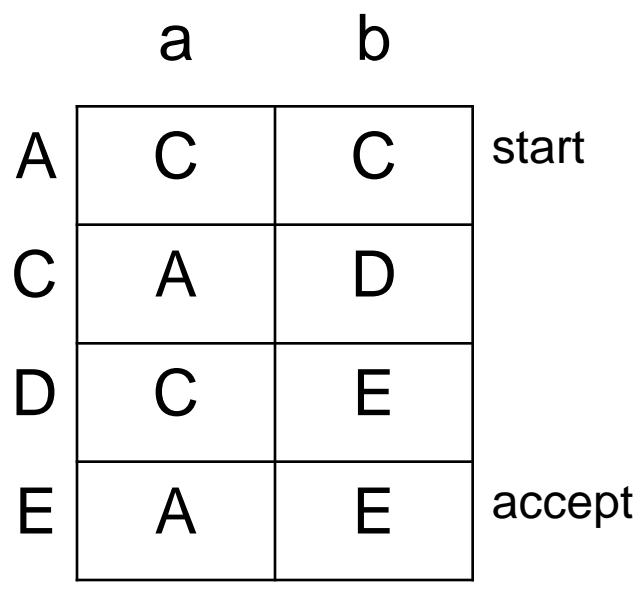
- $\Pi = \{A, B\}, \{C, F, G\}, \{D\}, \{E, H\}$
- $T(\{A, B\}, a) \rightarrow \{C, F, G\}$
- $T(\{A, B\}, b) \rightarrow \{C, F, G\}$
- $T(\{C, F, G\}, a) \rightarrow \{A, B\}$
- $T(\{C, F, G\}, b) \rightarrow \{D\}$
- $T(\{D\}, a) \rightarrow \{C, F, G\}$
- $T(\{D\}, b) \rightarrow \{E, H\}$
- $T(\{E, H\}, a) \rightarrow \{A, B\}$
- $T(\{E, H\}, b) \rightarrow \{E, H\}$
- No further partitioning is possible

# Checkpoint Solution

	a	b	
A	G	F	start
B	C	G	
C	B	D	
D	G	E	
E	B	H	accept
F	A	D	
G	B	D	
H	A	E	accept

- $\Pi = \{A, B\}, \{C, F, G\}, \{D\}, \{E, H\}$
- Representatives:
  - A = {A, B}
  - C = {C, F, G}
  - D = {D}
  - E = {E, H}
- The minimal DFA is shown on the next slide.

# Checkpoint Solution



# **Syntax Analyzer -Top-Down Parsing**

# Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

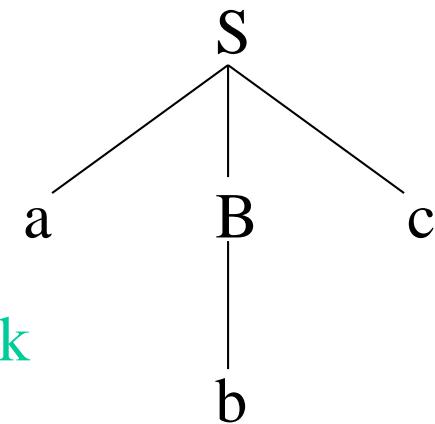
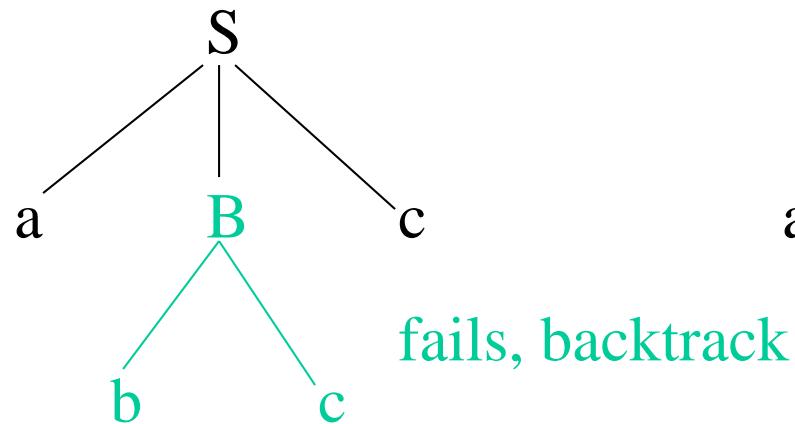
# Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$$S \rightarrow aBc$$

$$B \rightarrow bc \mid b$$

input: abc



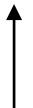
# Predictive Parser

a grammar	→	→	a grammar suitable for predictive parsing (a LL(1) grammar)
eliminate		left	
left recursion		factor	no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$$

input: ... a .....



current token

# Predictive Parser (example)

$\text{stmt} \rightarrow \text{if} \dots \dots \quad |$   
 $\quad \quad \quad \text{while} \dots \dots \quad |$   
 $\quad \quad \quad \text{begin} \dots \dots \quad |$   
 $\quad \quad \quad \text{for} \dots \dots$

- When we are trying to write the non-terminal  $\text{stmt}$ , if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal  $\text{stmt}$ , we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

# Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex:      $A \rightarrow aBb$         (This is only the production rule for A)

```
proc A {  
    - match the current token with a, and move to the next token;  
    - call ‘B’;  
    - match the current token with b, and move to the next token;  
}
```

# Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {  
    case of the current token {  
        'a': - match the current token with a, and move to the next token;  
              - call 'B';  
              - match the current token with b, and move to the next token;  
        'b': - match the current token with b, and move to the next token;  
              - call 'A';  
              - call 'B';  
    }  
}
```

# Recursive Predictive Parsing (cont.)

- When to apply  $\epsilon$ -productions.

$$A \rightarrow aA \mid bB \mid \epsilon$$

- If all other productions fail, we should apply an  $\epsilon$ -production. For example, if the current token is not a or b, we may apply the  $\epsilon$ -production.
- Most correct choice: We should apply an  $\epsilon$ -production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

# Recursive Predictive Parsing (Example)

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

$B \rightarrow bB \mid \epsilon$

$C \rightarrow f$

proc A {

    case of the current token {

        a: - match the current token with a,  
            and move to the next token;

        - call B;  
        - match the current token with e,  
            and move to the next token;

        c: - match the current token with c,  
            and move to the next token;

        - call B;  
        - match the current token with d,  
            and move to the next token;

        f: - call C

}

    } first set of C

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

    proc B {

        case of the current token {

            b: - match the current token with b,  
                and move to the next token;

            - call B

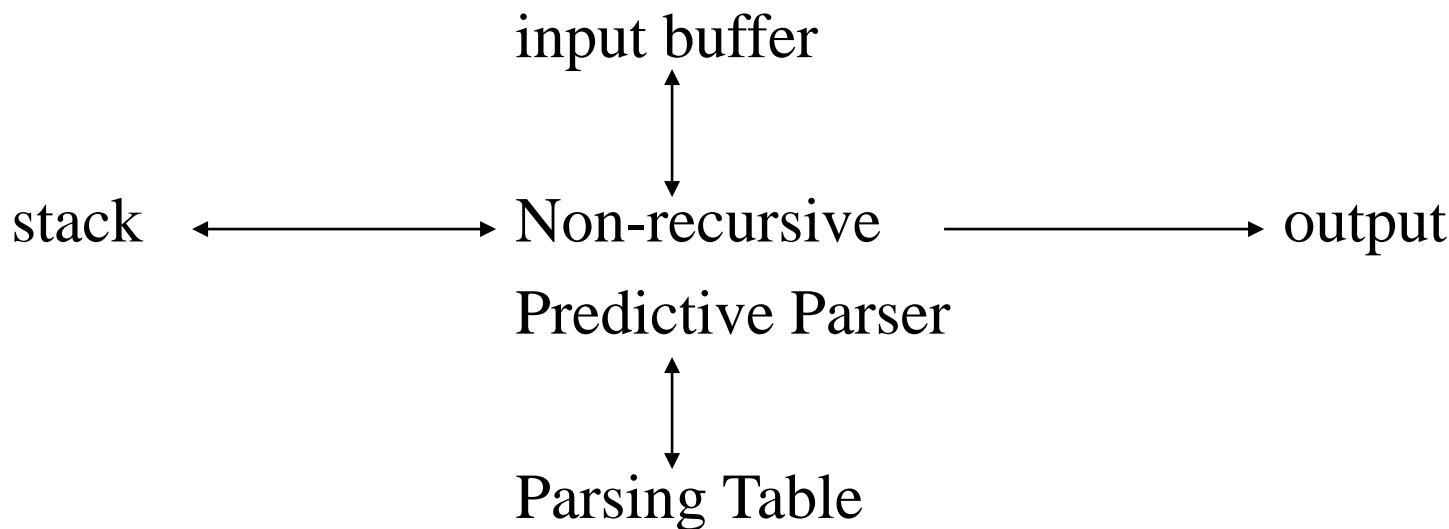
            e,d: do nothing

    }

    } follow set of B

# Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



# LL(1) Parser

## input buffer

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

## output

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

## stack

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

## parsing table

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

# LL(1) Parser – Parser Actions

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

# LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

	a	b	\$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

LL(1) Parsing  
Table

## stack

\$S

\$aBa

\$aB

\$aBb

\$aB

\$aBb

\$aB

\$a

\$

## input

abba\$

abba\$

bba\$

bba\$

ba\$

ba\$

a\$

a\$

\$

## output

$S \rightarrow aBa$

$B \rightarrow bB$

$B \rightarrow bB$

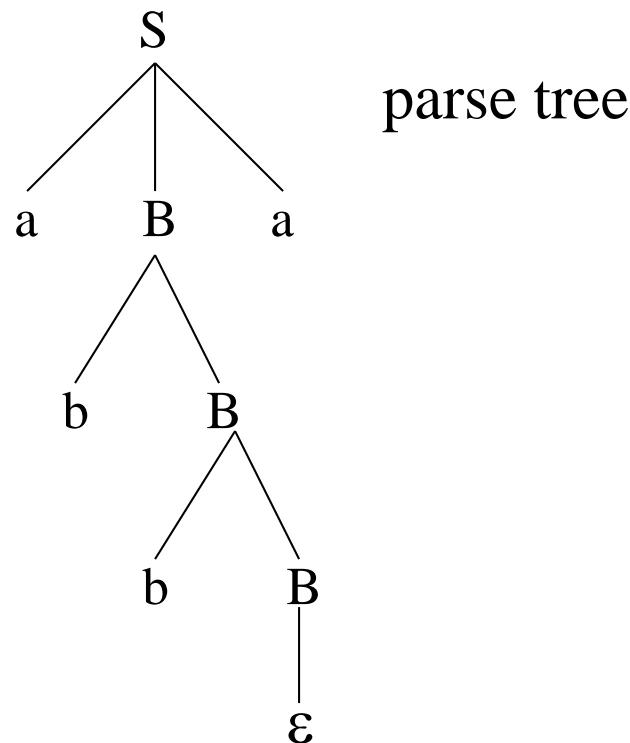
$B \rightarrow \epsilon$

accept, successful completion

# LL(1) Parser – Example1 (cont.)

Outputs:  $S \rightarrow aBa$      $B \rightarrow bB$      $B \rightarrow bB$      $B \rightarrow \epsilon$

Derivation(left-most):  $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



# LL(1) Parser – Example2

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

	<b>id</b>	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

# LL(1) Parser – Example2

<u>stack</u>	<u>input</u>	<u>output</u>
\$E	id+id\$	E → TE'
\$E'T	id+id\$	T → FT'
\$E'T'F	id+id\$	F → id
\$ E'T'id	id+id\$	
\$ E'T'	+id\$	T' → ε
\$ E'	+id\$	E' → +TE'
\$ E'T+	+id\$	
\$ E'T	id\$	T → FT'
\$ E'T'F	id\$	F → id
\$ E'T'id	id\$	
\$ E'T'	\$	T' → ε
\$ E'	\$	E' → ε
\$	\$	accept

# Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST      FOLLOW
- **FIRST( $\alpha$ )** is a set of the terminal symbols which occur as first symbols in strings derived from  $\alpha$  where  $\alpha$  is any string of grammar symbols.
- if  $\alpha$  derives to  $\epsilon$ , then  $\epsilon$  is also in  $\text{FIRST}(\alpha)$  .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal  $a$  is in  $\text{FOLLOW}(A)$  if  $S \xrightarrow{*} \alpha A a \beta$
  - $\$$  is in  $\text{FOLLOW}(A)$  if  $S \xrightarrow{*} \alpha A$

# Compute FIRST for Any String X

- If X is a terminal symbol  $\rightarrow \text{FIRST}(X)=\{X\}$
- If X is a non-terminal symbol and  $X \rightarrow \epsilon$  is a production rule  $\rightarrow \epsilon$  is in  $\text{FIRST}(X)$ .
- If X is a non-terminal symbol and  $X \rightarrow Y_1Y_2..Y_n$  is a production rule
  - $\rightarrow$  if a terminal **a** in  $\text{FIRST}(Y_i)$  and  $\epsilon$  is in all  $\text{FIRST}(Y_j)$  for  $j=1,...,i-1$  then **a** is in  $\text{FIRST}(X)$ .
  - $\rightarrow$  if  $\epsilon$  is in all  $\text{FIRST}(Y_j)$  for  $j=1,...,n$  then  $\epsilon$  is in  $\text{FIRST}(X)$ .
- If X is  $\epsilon$   $\rightarrow \text{FIRST}(X)=\{\epsilon\}$
- If X is  $Y_1Y_2..Y_n$ 
  - $\rightarrow$  if a terminal **a** in  $\text{FIRST}(Y_i)$  and  $\epsilon$  is in all  $\text{FIRST}(Y_j)$  for  $j=1,...,i-1$  then **a** is in  $\text{FIRST}(X)$ .
  - $\rightarrow$  if  $\epsilon$  is in all  $\text{FIRST}(Y_j)$  for  $j=1,...,n$  then  $\epsilon$  is in  $\text{FIRST}(X)$ .

# FIRST Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

$\text{FIRST}(F) = \{( , id\}$

$\text{FIRST}(T') = \{ *, \epsilon \}$

$\text{FIRST}(T) = \{( , id\}$

$\text{FIRST}(E') = \{ +, \epsilon \}$

$\text{FIRST}(E) = \{( , id\}$

$\text{FIRST}(TE') = \{( , id\}$

$\text{FIRST}(+TE') = \{ + \}$

$\text{FIRST}(\epsilon) = \{ \epsilon \}$

$\text{FIRST}(FT') = \{( , id\}$

$\text{FIRST}(*FT') = \{ * \}$

$\text{FIRST}(\epsilon) = \{ \epsilon \}$

$\text{FIRST}((E)) = \{ ( \}$

$\text{FIRST}(id) = \{ id \}$

# Compute FOLLOW (for non-terminals)

- If  $S$  is the start symbol  $\rightarrow \$$  is in  $\text{FOLLOW}(S)$
- if  $A \rightarrow \alpha B \beta$  is a production rule  
 $\rightarrow$  everything in  $\text{FIRST}(\beta)$  is  $\text{FOLLOW}(B)$  except  $\epsilon$
- If ( $A \rightarrow \alpha B$  is a production rule) or  
( $A \rightarrow \alpha B \beta$  is a production rule and  $\epsilon$  is in  $\text{FIRST}(\beta)$ )  
 $\rightarrow$  everything in  $\text{FOLLOW}(A)$  is in  $\text{FOLLOW}(B)$ .

We apply these rules until nothing more can be added to any follow set.

# FOLLOW Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

$\text{FOLLOW}(E) = \{ \$, ) \}$

$\text{FOLLOW}(E') = \{ \$, ) \}$

$\text{FOLLOW}(T) = \{ +, ), \$ \}$

$\text{FOLLOW}(T') = \{ +, ), \$ \}$

$\text{FOLLOW}(F) = \{ +, *, ), \$ \}$

# Constructing LL(1) Parsing Table -- Algorithm

- for each production rule  $A \rightarrow \alpha$  of a grammar G
  - for each terminal  $a$  in  $\text{FIRST}(\alpha)$ 
    - add  $A \rightarrow \alpha$  to  $M[A,a]$
  - If  $\epsilon$  in  $\text{FIRST}(\alpha)$ 
    - for each terminal  $a$  in  $\text{FOLLOW}(A)$  add  $A \rightarrow \alpha$  to  $M[A,a]$
  - If  $\epsilon$  in  $\text{FIRST}(\alpha)$  and  $\$$  in  $\text{FOLLOW}(A)$ 
    - add  $A \rightarrow \alpha$  to  $M[A,\$]$
- All other undefined entries of the parsing table are error entries.

# Constructing LL(1) Parsing Table -- Example

$E \rightarrow TE'$

$\text{FIRST}(TE') = \{(, id\}$

→  $E \rightarrow TE'$  into  $M[E, ()]$  and  $M[E, id]$

$E' \rightarrow +TE'$

$\text{FIRST}(+TE') = \{+\}$

→  $E' \rightarrow +TE'$  into  $M[E', +]$

$E' \rightarrow \epsilon$

$\text{FIRST}(\epsilon) = \{\epsilon\}$

→ none

but since  $\epsilon$  in  $\text{FIRST}(\epsilon)$   
and  $\text{FOLLOW}(E') = \{ \$, ) \}$

→  $E' \rightarrow \epsilon$  into  $M[E', \$]$  and  $M[E', )]$

$T \rightarrow FT'$

$\text{FIRST}(FT') = \{(, id\}$

→  $T \rightarrow FT'$  into  $M[T, ()]$  and  $M[T, id]$

$T' \rightarrow *FT'$

$\text{FIRST}(*FT') = \{ *\}$

→  $T' \rightarrow *FT'$  into  $M[T', *]$

$T' \rightarrow \epsilon$

$\text{FIRST}(\epsilon) = \{\epsilon\}$

→ none

but since  $\epsilon$  in  $\text{FIRST}(\epsilon)$   
and  $\text{FOLLOW}(T') = \{ \$, ), + \}$

→  $T' \rightarrow \epsilon$  into  $M[T', \$]$ ,  $M[T', )]$  and  $M[T', +]$

$F \rightarrow (E)$

$\text{FIRST}((E)) = \{()\}$

→  $F \rightarrow (E)$  into  $M[F, ()]$

$F \rightarrow id$

$\text{FIRST}(id) = \{ id \}$

→  $F \rightarrow id$  into  $M[F, id]$

# LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-head symbol do determine parser action  
 $\downarrow$   
LL(1)  
↑  
input scanned from left to right

left most derivation

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

# A Grammar which is not LL(1)

$$S \rightarrow i C t S E \mid a$$

$$E \rightarrow e S \mid \epsilon$$

$$C \rightarrow b$$

$$\text{FOLLOW}(S) = \{ \$, e \}$$

$$\text{FOLLOW}(E) = \{ \$, e \}$$

$$\text{FOLLOW}(C) = \{ t \}$$

$$\text{FIRST}(iCtSE) = \{ i \}$$

$$\text{FIRST}(a) = \{ a \}$$

$$\text{FIRST}(eS) = \{ e \}$$

$$\text{FIRST}(\epsilon) = \{ \epsilon \}$$

$$\text{FIRST}(b) = \{ b \}$$

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e S$			$E \rightarrow \epsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem → ambiguity

# A Grammar which is not LL(1) (cont.)

- What do we have to do if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$ 
    - ➔ any terminal that appears in  $\text{FIRST}(\beta)$  also appears in  $\text{FIRST}(A\alpha)$  because  $A\alpha \Rightarrow \beta\alpha$ .
    - ➔ If  $\beta$  is  $\epsilon$ , any terminal that appears in  $\text{FIRST}(\alpha)$  also appears in  $\text{FIRST}(A\alpha)$  and  $\text{FOLLOW}(A)$ .
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 
    - ➔ any terminal that appears in  $\text{FIRST}(\alpha\beta_1)$  also appears in  $\text{FIRST}(\alpha\beta_2)$ .
- An ambiguous grammar cannot be a LL(1) grammar.

# Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules  $A \rightarrow \alpha$  and  $A \rightarrow \beta$ 
  1. Both  $\alpha$  and  $\beta$  cannot derive strings starting with same terminals.
  2. At most one of  $\alpha$  and  $\beta$  can derive to  $\epsilon$ .
  3. If  $\beta$  can derive to  $\epsilon$ , then  $\alpha$  cannot derive to any string starting with a terminal in  $\text{FOLLOW}(A)$ .

# Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry  $M[A,a]$  is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

# Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few changes as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

# Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as *synch* to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which is on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

# Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \epsilon$

$A \rightarrow a \mid cAd$

$\text{FOLLOW}(S) = \{\$\}$

$\text{FOLLOW}(A) = \{b, d\}$

	a	b	c	d	e	\$
S	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow e$	$S \rightarrow \epsilon$
A	$A \rightarrow a$	<i>sync</i>	$A \rightarrow cAd$	<i>sync</i>	<i>sync</i>	<i>sync</i>

<u>stack</u>	<u>input</u>	<u>output</u>
\$S	aab\$	$S \rightarrow AbS$
\$SbA	aab\$	$A \rightarrow a$
\$Sba	aab\$	
\$Sb	ab\$	Error: missing b, inserted
\$S	ab\$	$S \rightarrow AbS$
\$SbA	ab\$	$A \rightarrow a$
\$Sba	ab\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \epsilon$
\$	\$	accept

<u>stack</u>	<u>input</u>	<u>output</u>
\$S	ceadb\$	$S \rightarrow AbS$
\$SbA	ceadb\$	$A \rightarrow cAd$
\$SbdAc	ceadb\$	
\$SbdA	eadb\$	Error:unexpected e (illegal A)
		(Remove all input tokens until first b or d, pop A)
\$Sbd	db\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \epsilon$
\$	\$	accept

# Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

# **Syntax Analyser- Bottom Up Parsing**

# Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

$S \Rightarrow \dots \Rightarrow \omega$  (the right-most derivation of  $\omega$ )  
 $\leftarrow$  (the bottom-up parser finds the right-most derivation in the reverse order)
- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will be replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

# Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string     $\rightarrow$     the starting symbol  
reduced to
- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation:

$$S \xrightarrow[\text{rm}]{}^* \omega$$

Shift-Reduce Parser finds:

$$\omega \xleftarrow[\text{rm}]{} \dots \xleftarrow[\text{rm}]{} S$$

# Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

input string: aa**a**bb

$A \rightarrow aA \mid a$

a**a**Abb

$B \rightarrow bB \mid b$

aA**b**b

↓ reduction

**aABb**

S

$S \xrightarrow{rm} aABb \xrightarrow{rm} aA**bb** \xrightarrow{rm} aaA**bb** \xrightarrow{rm} aaabb$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

# Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form  $\gamma$  ( $\equiv \alpha\beta\omega$ ) is a production rule  $A \rightarrow \beta$  and a position of  $\gamma$  where the string  $\beta$  may be found and replaced by  $A$  to produce the previous right-sentential form in a rightmost derivation of  $\gamma$ .

$$S \xrightarrow[\text{rm}]{*} \alpha A \omega \xRightarrow[\text{rm}]{*} \alpha \beta \omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that  $\omega$  is a string of terminals.

# Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S = \gamma_0 \xrightarrow{\text{rm}} \gamma_1 \xrightarrow{\text{rm}} \gamma_2 \xrightarrow{\text{rm}} \cdots \xrightarrow{\text{rm}} \gamma_{n-1} \xrightarrow{\text{rm}} \gamma_n = \omega$$

← input string

- Start from  $\gamma_n$ , find a handle  $A_n \rightarrow \beta_n$  in  $\gamma_n$ , and replace  $\beta_n$  in by  $A_n$  to get  $\gamma_{n-1}$ .
- Then find a handle  $A_{n-1} \rightarrow \beta_{n-1}$  in  $\gamma_{n-1}$ , and replace  $\beta_{n-1}$  in by  $A_{n-1}$  to get  $\gamma_{n-2}$ .
- Repeat this, until we reach S.

# A Shift-Reduce Parser

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T^*F \mid F$$
$$F \rightarrow (E) \mid id$$

Right-Most Derivation of  $id + id^* id$

$$E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow E+T^*id \Rightarrow E+F^*id$$
$$\Rightarrow E+id^*id \Rightarrow T+id^*id \Rightarrow F+id^*id \Rightarrow id + id^* id$$

## Right-Most Sentential Form

$id$ + $id^*id$

$F$ + $id^*id$

$T$ + $id^*id$

$E + \underline{id}^*id$

$E + \underline{F}^*id$

$E + T^*\underline{id}$

$E + \underline{T^*F}$

$E+T$

$E$

## Reducing Production

$F \rightarrow id$

$T \rightarrow F$

$E \rightarrow T$

$F \rightarrow id$

$T \rightarrow F$

$F \rightarrow id$

$T \rightarrow T^*F$

$E \rightarrow E+T$

Handles are red and underlined in the right-sentential forms.

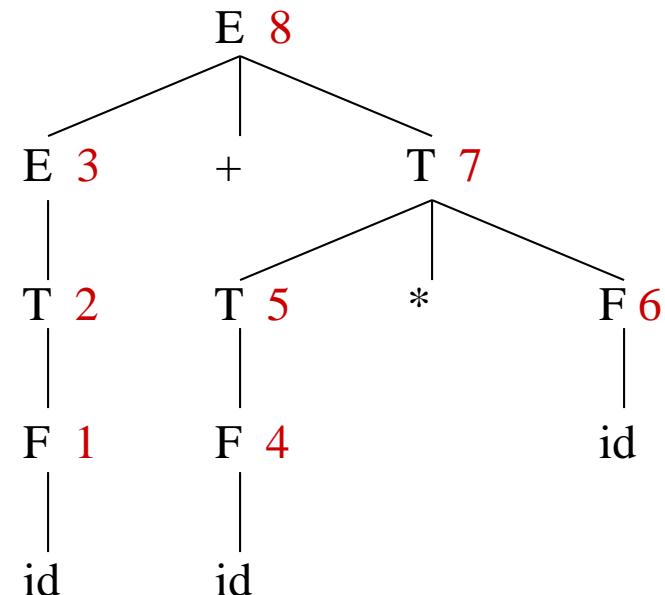
# A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.
- Initial stack just contains only the end-marker \$.
- The end of the input string is marked by the end-marker \$.

# A Stack Implementation of A Shift-Reduce Parser

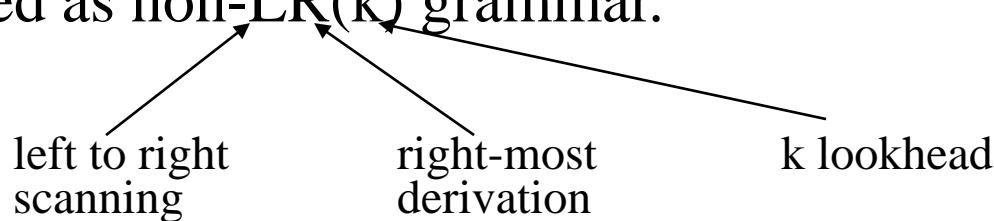
<u>Stack</u>	<u>Input</u>	<u>Action</u>
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by F → id
\$F	+id*id\$	reduce by T → F
\$T	+id*id\$	reduce by E → T
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by F → id
\$E+F	*id\$	reduce by T → F
\$E+T	*id\$	shift
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by F → id
\$E+T*F	\$	reduce by T → T*F
\$E+T	\$	reduce by E → E+T
\$E	\$	accept

Parse Tree



# Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as **non-LR( $k$ ) grammar**.



- An ambiguous grammar can never be a LR grammar.

# Shift-Reduce Parsers

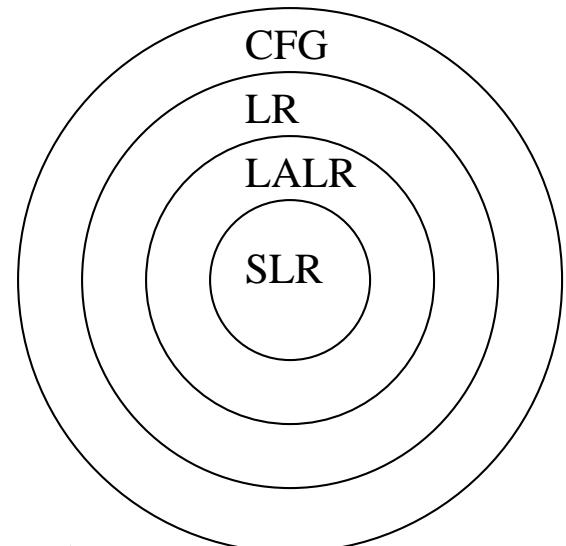
- There are two main categories of shift-reduce parsers

## 1. Operator-Precedence Parser

- simple, but only a small class of grammars.

## 2. LR-Parsers

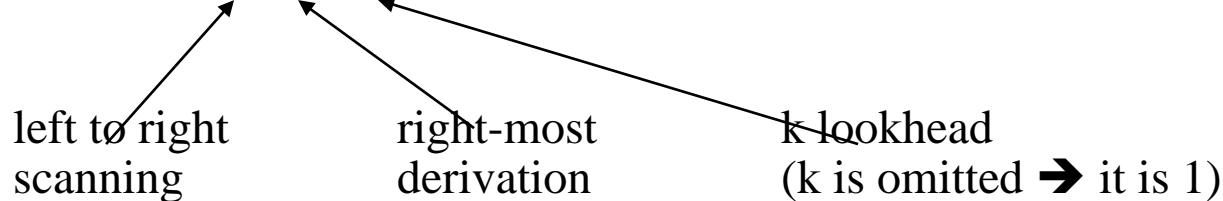
- covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (lookhead LR parser)
- SLR, LR and LALR work same, only their parsing tables are different.



# LR Parsers

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

LR( $k$ ) parsing.

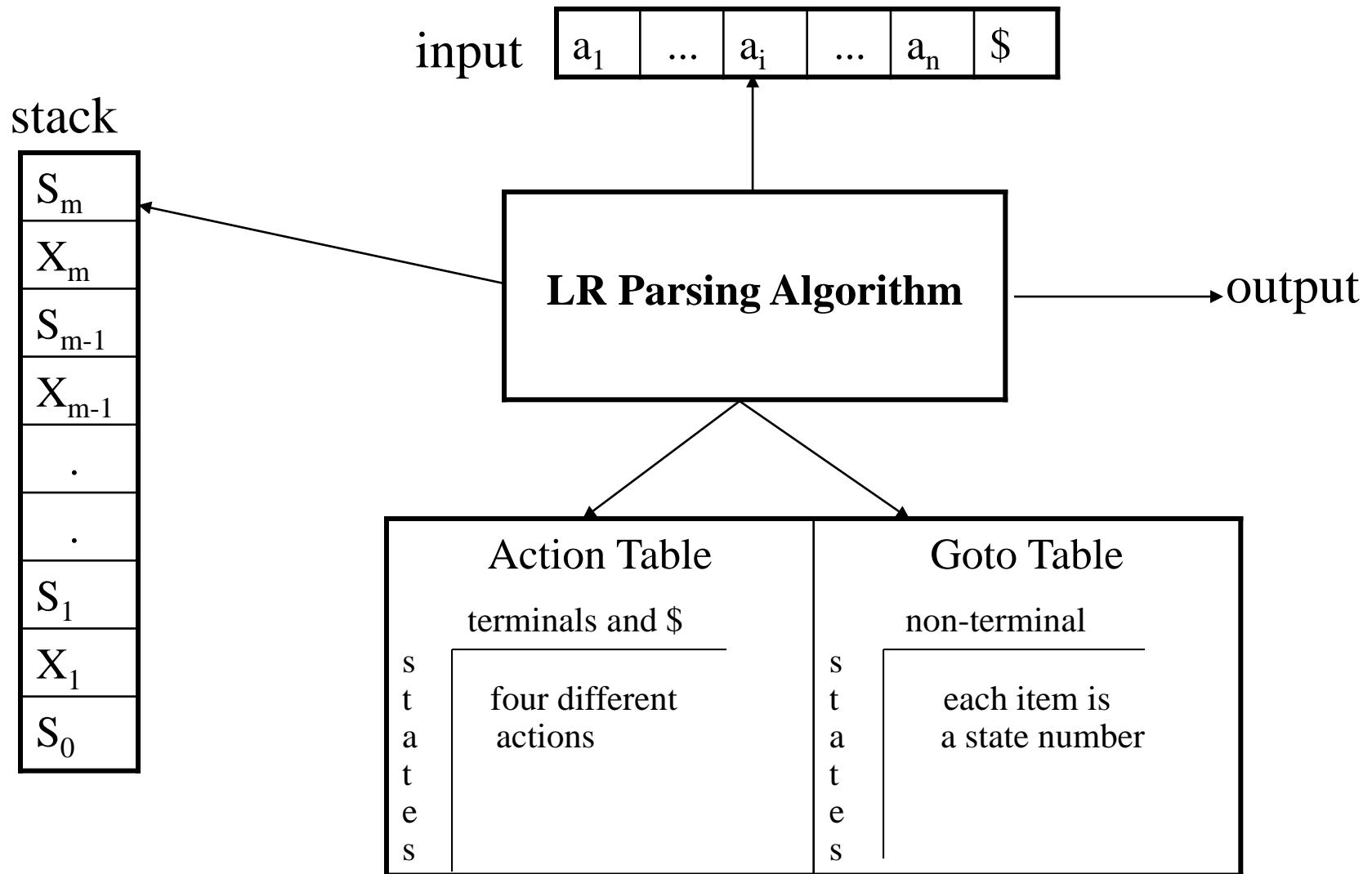


- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.  
 $\text{LL}(1)\text{-Grammars} \subset \text{LR}(1)\text{-Grammars}$
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

# LR Parsers

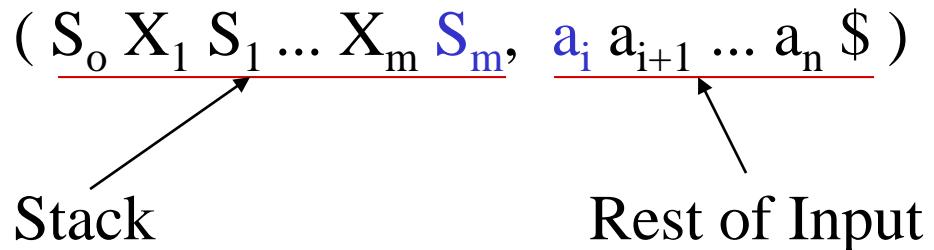
- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

# LR Parsing Algorithm



# A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:



- $S_m$  and  $a_i$  decides the parser action by consulting the parsing action table. (*Initial Stack* contains just  $S_o$ )
- A configuration of a LR parsing represents the right sentential form:

$$X_1 \dots X_m a_i a_{i+1} \dots a_n \$$$

# Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack  
 $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_m S_m \textcolor{red}{a_i s}, a_{i+1} \dots a_n \$)$
2. **reduce A→β** (or **rn** where n is a production number)
  - pop  $2|\beta| (=r)$  items from the stack;
  - then push **A** and **s** where **s=goto[s<sub>m-r</sub>,A]**  
 $(S_o X_1 S_1 \dots X_m S_m, a_i a_{i+1} \dots a_n \$) \rightarrow (S_o X_1 S_1 \dots X_{m-r} \textcolor{red}{S}_{m-r} A s, a_i \dots a_n \$)$
  - Output is the reducing production reduce  $A \rightarrow \beta$
3. **Accept** – Parsing successfully completed
4. **Error** -- Parser detected an error (an empty entry in the action table)

# Reduce Action

- pop  $2|\beta|$  ( $=r$ ) items from the stack; let us assume that  $\beta = Y_1 Y_2 \dots Y_r$
- then push  $A$  and  $s$  where  $s=\text{goto}[s_{m-r}, A]$

(  $S_o X_1 S_1 \dots X_{m-r} S_{m-r} Y_1 S_{m-r} \dots Y_r S_m, a_i a_{i+1} \dots a_n \$$  )  
→ (  $S_o X_1 S_1 \dots X_{m-r} S_{m-r} A s, a_i \dots a_n \$$  )

- In fact,  $Y_1 Y_2 \dots Y_r$  is a handle.

$X_1 \dots X_{m-r} A a_i \dots a_n \$ \Rightarrow X_1 \dots X_m Y_1 \dots Y_r a_i a_{i+1} \dots a_n \$$

# (SLR) Parsing Tables for Expression Grammar

- 1)  $E \rightarrow E + T$
- 2)  $E \rightarrow T$
- 3)  $T \rightarrow T^* F$
- 4)  $T \rightarrow F$
- 5)  $F \rightarrow (E)$
- 6)  $F \rightarrow id$

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

# Actions of A (S)LR-Parser -- Example

<u>stack</u>	<u>input</u>	<u>action</u>	<u>output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by F→id	F→id
0F3	*id+id\$	reduce by T→F	T→F
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by F→id	F→id
0T2*7F10	+id\$	reduce by T→T*F	T→T*F
0T2	+id\$	reduce by E→T	E→T
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by F→id	F→id
0E1+6F3	\$	reduce by T→F	T→F
0E1+6T9	\$	reduce by E→E+T	E→E+T
0E1	\$	accept	

# Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0)** item of a grammar G is a production of G a dot at the some position of the right side.
- Ex:  $A \rightarrow aBb$       Possible LR(0) Items:       $A \rightarrow \bullet aBb$   
(four different possibility)       $A \rightarrow a \bullet Bb$   
     $A \rightarrow aB \bullet b$   
     $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar:*  
 $G'$  is  $G$  with a new production rule  $S' \rightarrow S$  where  $S'$  is the new starting symbol.

# The Closure Operation

- If  $I$  is a set of LR(0) items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of LR(0) items constructed from  $I$  by the two rules:
  1. Initially, every LR(0) item in  $I$  is added to  $\text{closure}(I)$ .
  2. If  $A \rightarrow \alpha \bullet B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production rule of  $G$ ; then  $B \rightarrow \bullet \gamma$  will be in the  $\text{closure}(I)$ .We will apply this rule until no more new LR(0) items can be added to  $\text{closure}(I)$ .

# The Closure Operation -- Example

$E' \rightarrow E$

$\text{closure}(\{E' \rightarrow \bullet E\}) =$

$E \rightarrow E + T$

{  $E' \rightarrow \bullet E$  ← kernel items

$E \rightarrow T$

$E \rightarrow \bullet E + T$

$T \rightarrow T^* F$

$E \rightarrow \bullet T$

$T \rightarrow F$

$T \rightarrow \bullet T^* F$

$F \rightarrow (E)$

$T \rightarrow \bullet F$

$F \rightarrow \text{id}$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet \text{id}$  }

$\text{FOLLOW}(E) = \{\$, +, )\}$

$\text{FOLLOW}(T) = \{\$, +, ), *\}$

$\text{FOLLOW}(F) = \{\$, +, ), *\}$

# Goto Operation

- If  $I$  is a set of LR(0) items and  $X$  is a grammar symbol (terminal or non-terminal), then  $\text{goto}(I, X)$  is defined as follows:
  - If  $A \rightarrow \alpha \bullet X \beta$  in  $I$   
then every item in **closure**( $\{A \rightarrow \alpha X \bullet \beta\}$ ) will be in  $\text{goto}(I, X)$ .

Example:

$$I = \{ E' \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, \\ T \rightarrow \bullet T^* F, T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), F \rightarrow \bullet \text{id} \}$$

$$\text{goto}(I, E) = \{ E' \rightarrow E \bullet, E \rightarrow E \bullet + T \}$$

$$\text{goto}(I, T) = \{ E \rightarrow T \bullet, T \rightarrow T \bullet * F \}$$

$$\text{goto}(I, F) = \{ T \rightarrow F \bullet \}$$

$$\text{goto}(I, ()) = \{ F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T^* F, T \rightarrow \bullet F, \\ F \rightarrow \bullet (E), F \rightarrow \bullet \text{id} \}$$

$$\text{goto}(I, \text{id}) = \{ F \rightarrow \text{id} \bullet \}$$

# Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- *Algorithm:*

$C$  is { closure( $\{S' \rightarrow \bullet S\}$ ) }

**repeat** the followings until no more set of LR(0) items can be added to  $C$ .

**for each** I in  $C$  and each grammar symbol X

**if** goto(I,X) is not empty and not in  $C$

add goto(I,X) to  $C$
- goto function is a DFA on the sets in C.

# The Canonical LR(0) Collection -- Example

$I_0: E' \rightarrow .E$   $I_1: E' \rightarrow E.$   $I_6: E \rightarrow E+.T$   
 $E \rightarrow .E+T$   ~~$E \rightarrow E.+T$~~

$E \rightarrow .T$

$T \rightarrow .T^*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

~~$E \rightarrow E+.T$~~

$I_2: E \rightarrow T.$

$T \rightarrow T.^*F$

$I_3: T \rightarrow F.$

$I_4: F \rightarrow (.E)$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T^*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_5: F \rightarrow id.$

$I_9: E \rightarrow E+T.$   
 ~~$T \rightarrow T^*F$~~

$T \rightarrow .F$

~~$F \rightarrow (E)$~~

$F \rightarrow .id$

$I_7: T \rightarrow T.^*F$   
 $F \rightarrow .(E)$

$F \rightarrow .id$

$I_8: F \rightarrow (E.)$   
 $E \rightarrow E.+T$

$i9$   
 $E \rightarrow E+T.$   
 $T \rightarrow T.^*F$

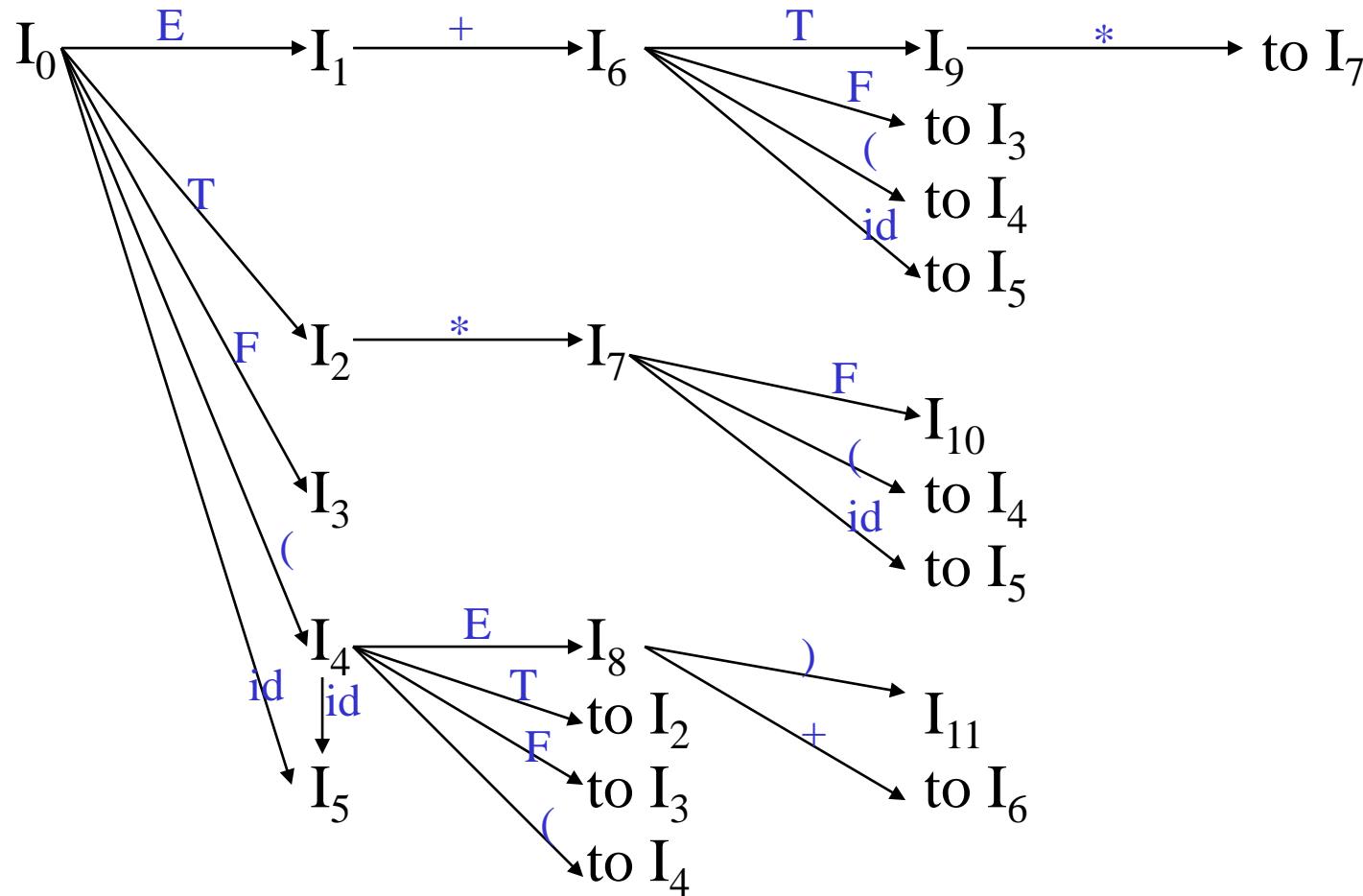
$I_{10}: T \rightarrow T^*F.$

$I_{11}: F \rightarrow (E).$

$|1$   
 $E' \rightarrow E.$   
 $E \rightarrow E.+T$

$|6$   
 $E \rightarrow E+.T$   
 $T \rightarrow T^*F$   
 $T \rightarrow .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .id$

# Transition Diagram (DFA) of Goto Function



# Constructing SLR Parsing Table

(of an augmented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.

$$C \leftarrow \{I_0, \dots, I_n\}$$

2. Create the parsing action table as follows

- If  $a$  is a terminal,  $A \rightarrow \alpha.a\beta$  in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a]$  is *shift j*.
- If  $A \rightarrow \alpha.$  is in  $I_i$ , then  $\text{action}[i, a]$  is *reduce A  $\rightarrow \alpha$*  for all  $a$  in  $\text{FOLLOW}(A)$  where  $A \neq S'$ .
- If  $S' \rightarrow S.$  is in  $I_i$ , then  $\text{action}[i, \$]$  is *accept*.
- If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table

- for all non-terminals  $A$ , if  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains  $S' \rightarrow .S$

# Parsing Tables of Expression Grammar

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

# **SLR(1) Grammar**

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

# **shift/reduce and reduce/reduce conflicts**

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule  $i$  or  $j$  for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar  $G$  has a conflict, we say that that grammar is not SLR grammar.

# Conflict Example

$S \rightarrow L=R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$I_0: S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

$I_1: S' \rightarrow S.$

$I_2: S \rightarrow L.=R$   
 $R \rightarrow L.$

$I_3: S \rightarrow R.$

$I_6: S \rightarrow L=.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

$I_9: S \rightarrow L=R.$

Problem

$\text{FOLLOW}(R) = \{=, \$\}$

= shift 6

reduce by  $R \rightarrow L$

shift/reduce conflict

$I_4: L \rightarrow *.R$

$R \rightarrow .L$

$L \rightarrow .*R$

$L \rightarrow .id$

$I_7: L \rightarrow *R.$

$I_8: R \rightarrow L.$

# Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$I_0: S' \rightarrow .S$   
 $S \rightarrow .AaAb$   
 $S \rightarrow .BbBa$   
 $A \rightarrow .$   
 $B \rightarrow .$

Problem

$\text{FOLLOW}(A) = \{a, b\}$

$\text{FOLLOW}(B) = \{a, b\}$

a  $\begin{cases} \xrightarrow{\text{reduce by } A \rightarrow \epsilon} \\ \xrightarrow{\text{reduce by } B \rightarrow \epsilon} \end{cases}$

reduce/reduce conflict

b  $\begin{cases} \xrightarrow{\text{reduce by } A \rightarrow \epsilon} \\ \xrightarrow{\text{reduce by } B \rightarrow \epsilon} \end{cases}$

reduce/reduce conflict

# Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state  $i$  makes a reduction by  $A \rightarrow \alpha$  when the current token is  $a$ :
  - if the  $A \rightarrow \alpha.$  in the  $I_i$  and  $a$  is  $\text{FOLLOW}(A)$
- In some situations,  $\beta A$  cannot be followed by the terminal  $a$  in a right-sentential form when  $\beta\alpha$  and the state  $i$  are on the top stack. This means that making reduction in this case is not correct.

$$S \rightarrow AaAb$$
$$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$$
$$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$$
$$S \rightarrow BbBa$$
$$A \rightarrow \epsilon$$
$$Aab \Rightarrow \epsilon ab$$
$$Bba \Rightarrow \epsilon ba$$
$$B \rightarrow \epsilon$$
$$AaAb \Rightarrow Aa \epsilon b$$
$$BbBa \Rightarrow Bb \epsilon a$$

# LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

$$A \rightarrow \alpha \bullet \beta, a$$

where **a** is the look-head of the LR(1) item  
(**a** is a terminal or end-marker.)

## LR(1) Item (cont.)

- When  $\beta$  (in the LR(1) item  $A \rightarrow \alpha \cdot \beta, a$ ) is not empty, the look-head does not have any affect.
- When  $\beta$  is empty ( $A \rightarrow \alpha \cdot, a$ ), we do the reduction by  $A \rightarrow \alpha$  only if the next input symbol is **a** (not for any terminal in  $\text{FOLLOW}(A)$ ).
- A state will contain  $A \rightarrow \alpha \cdot, a_1 \quad \dots \quad A \rightarrow \alpha \cdot, a_n$  where  $\{a_1, \dots, a_n\} \subseteq \text{FOLLOW}(A)$

...

$A \rightarrow \alpha \cdot, a_n$

# Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

**closure(I)** is: ( where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if  $A \rightarrow \alpha.B\beta, a$  in closure(I) and  $B \rightarrow \gamma$  is a production rule of G; then  $B \rightarrow .\gamma, b$  will be in the closure(I) for each terminal b in FIRST( $\beta a$ ) .

# goto operation

- If  $I$  is a set of LR(1) items and  $X$  is a grammar symbol (terminal or non-terminal), then  $\text{goto}(I, X)$  is defined as follows:
  - If  $A \rightarrow \alpha.X\beta, a$  in  $I$   
then every item in  $\text{closure}(\{A \rightarrow \alpha X. \beta, a\})$  will be in  $\text{goto}(I, X)$ .

# Construction of The Canonical LR(1) Collection

- *Algorithm:*

$C$  is  $\{ \text{closure}(\{S' \rightarrow .S, \$\}) \}$

**repeat** the followings until no more set of LR(1) items can be added to  $C$ .

**for each**  $I$  in  $C$  and each grammar symbol  $X$

**if**  $\text{goto}(I, X)$  is not empty and not in  $C$

            add  $\text{goto}(I, X)$  to  $C$

- $\text{goto}$  function is a DFA on the sets in  $C$ .

# A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \cdot \beta, a_1$$

...

$$A \rightarrow \alpha \cdot \beta, a_n$$

can be written as

$$A \rightarrow \alpha \cdot \beta, a_1/a_2/\dots/a_n$$

# Canonical LR(1) Collection -- Example

$$S \rightarrow AaAb$$

$$S \rightarrow BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

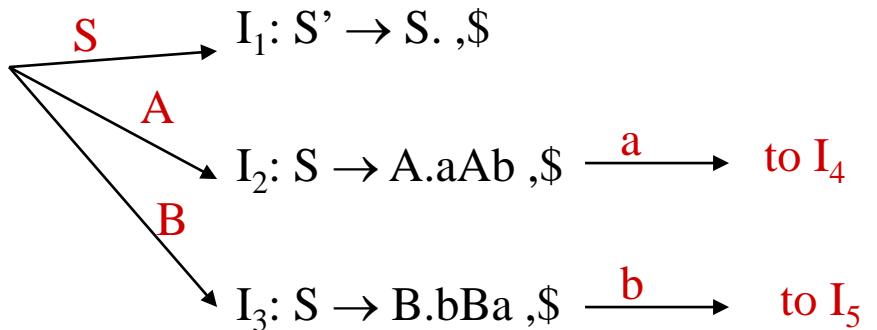
$$I_0: S' \rightarrow .S , \$$$

$$S \rightarrow .AaAb , \$$$

$$S \rightarrow .BbBa , \$$$

$$A \rightarrow . , a$$

$$B \rightarrow . , b$$



$$I_4: S \rightarrow Aa.Ab , \$ \xrightarrow{A} I_6: S \rightarrow AaA.b , \$ \xrightarrow{a} I_8: S \rightarrow AaAb. , \$$$

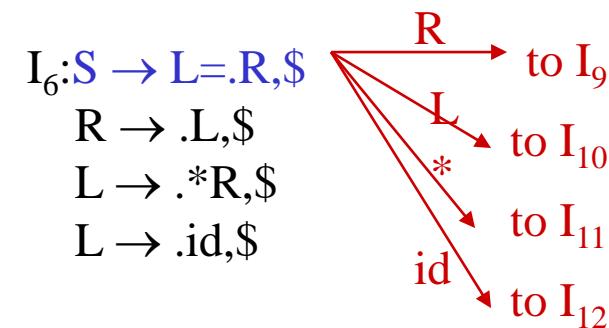
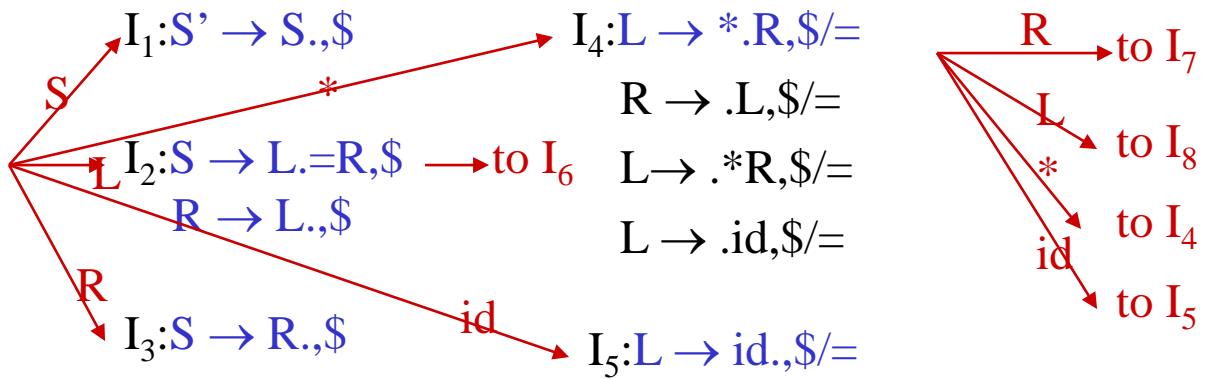
$A \rightarrow . , b$

$$I_5: S \rightarrow Bb.Ba , \$ \xrightarrow{B} I_7: S \rightarrow BbB.a , \$ \xrightarrow{b} I_9: S \rightarrow BbBa. , \$$$

$B \rightarrow . , a$

# Canonical LR(1) Collection – Example2

$S' \rightarrow S$	$I_0: S' \rightarrow .S, \$$
1) $S \rightarrow L=R$	$S \rightarrow .L=R, \$$
2) $S \rightarrow R$	$S \rightarrow .R, \$$
3) $L \rightarrow *R$	$L \rightarrow .*R, \$/=$
4) $L \rightarrow id$	$L \rightarrow .id, \$/=$
5) $R \rightarrow L$	$R \rightarrow .L, \$$



$I_7: L \rightarrow *R., \$/=$

$I_8: R \rightarrow L., \$/=$

$I_9: S \rightarrow L=R., \$$

$I_{10}: R \rightarrow L., \$$

$I_{11}: L \rightarrow *.R, \$$   
 $R \rightarrow .L, \$$   
 $L \rightarrow .*R, \$$   
 $L \rightarrow .id, \$$

$I_{12}: L \rightarrow id., \$$

$I_{13}: L \rightarrow *R., \$$

$I_4$  and  $I_{11}$

$I_5$  and  $I_{12}$

$I_7$  and  $I_{13}$

$I_8$  and  $I_{10}$

# Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for  $G'$ .

$$C \leftarrow \{I_0, \dots, I_n\}$$

2. Create the parsing action table as follows

- If  $a$  is a terminal,  $A \rightarrow \alpha \bullet a\beta, b$  in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a]$  is *shift j*.
- If  $A \rightarrow \alpha \bullet, a$  is in  $I_i$ , then  $\text{action}[i, a]$  is *reduce A  $\rightarrow \alpha$*  where  $A \neq S'$ .
- If  $S' \rightarrow S \bullet, \$$  is in  $I_i$ , then  $\text{action}[i, \$]$  is *accept*.
- If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table

- for all non-terminals  $A$ , if  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains  $S' \rightarrow .S, \$$

# LR(1) Parsing Tables – (for Example2)

	<b>id</b>	*	=	\$	S	L	R
<b>0</b>	s5	s4			1	2	3
<b>1</b>				acc			
<b>2</b>			s6	r5			
<b>3</b>				r2			
<b>4</b>	s5	s4				8	7
<b>5</b>			r4	r4			
<b>6</b>	s12	s11				10	9
<b>7</b>			r3	r3			
<b>8</b>			r5	r5			
<b>9</b>				r1			
<b>10</b>				r5			
<b>11</b>	s12	s11				10	13
<b>12</b>				r4			
<b>13</b>				r3			

no shift/reduce or  
no reduce/reduce conflict



so, it is a LR(1) grammar

# LALR Parsing Tables

- LALR stands for LookAhead LR.
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- *yacc* creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

# Creating LALR Parsing Tables

Canonical LR(1) Parser → LALR Parser  
shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

# The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$$I_1 : L \rightarrow id_{\bullet}, = \quad \xrightarrow{\hspace{1cm}} \quad \text{A new state:} \quad I_{12} : L \rightarrow id_{\bullet}, = \\ I_{12} : L \rightarrow id_{\bullet}, \$$$

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
  - In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

# Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.  
$$C = \{I_0, \dots, I_n\} \rightarrow C' = \{J_1, \dots, J_m\} \quad \text{where } m \leq n$$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If  $J = I_1 \cup \dots \cup I_k$  since  $I_1, \dots, I_k$  have same cores  
 $\rightarrow$  cores of  $\text{goto}(I_1, X), \dots, \text{goto}(I_k, X)$  must be same.
  - So,  $\text{goto}(J, X) = K$  where  $K$  is the union of all sets of items having same cores as  $\text{goto}(I_1, X)$ .
- If no conflict is introduced, the grammar is LALR(1) grammar.  
(We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

# Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha \bullet, a \quad \text{and} \quad B \rightarrow \beta \bullet a\gamma, b$$

- This means that a state of the canonical LR(1) parser must have:

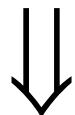
$$A \rightarrow \alpha \bullet, a \quad \text{and} \quad B \rightarrow \beta \bullet a\gamma, c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

# Reduce/Reduce Conflict

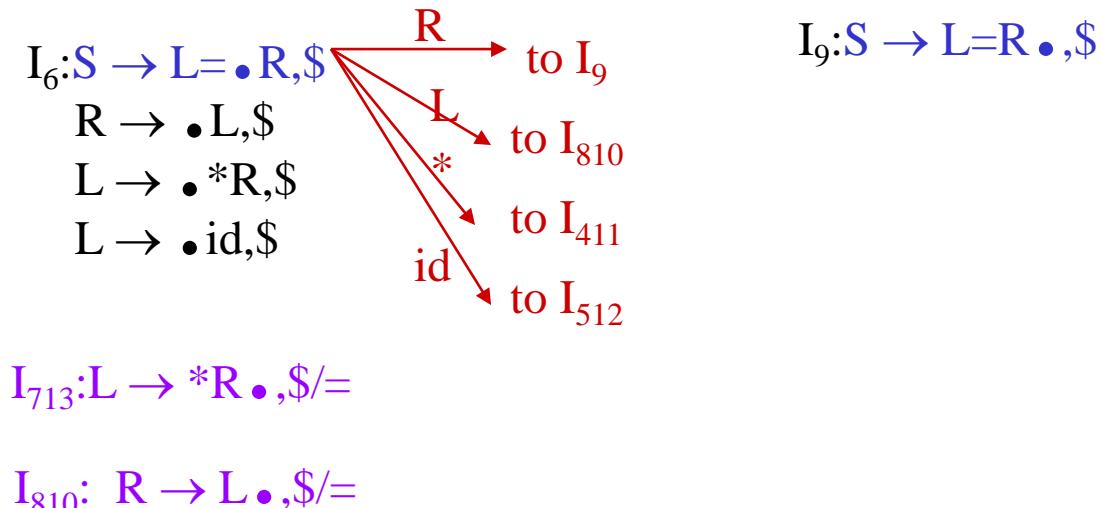
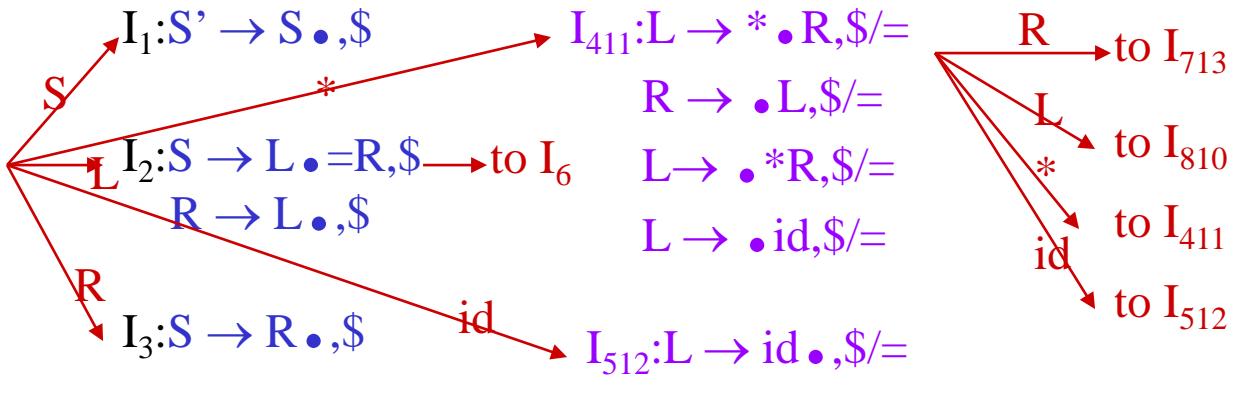
- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$I_1 : A \rightarrow \alpha \bullet, a$$
$$B \rightarrow \beta \bullet, b$$
$$I_2 : A \rightarrow \alpha \bullet, b$$
$$B \rightarrow \beta \bullet, c$$

$$I_{12} : A \rightarrow \alpha \bullet, a/b$$
$$B \rightarrow \beta \bullet, b/c$$

→ reduce/reduce conflict

# Canonical LALR(1) Collection – Example2

$S' \rightarrow S$	$I_0: S' \rightarrow \bullet S, \$$
1) $S \rightarrow L=R$	$S \rightarrow \bullet L=R, \$$
2) $S \rightarrow R$	$S \rightarrow \bullet R, \$$
3) $L \rightarrow *R$	$L \rightarrow \bullet *R, \$/=$
4) $L \rightarrow id$	$L \rightarrow \bullet id, \$/=$
5) $R \rightarrow L$	$R \rightarrow \bullet L, \$$



Same Cores  
 $I_4$  and  $I_{11}$

$I_5$  and  $I_{12}$

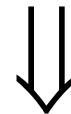
$I_7$  and  $I_{13}$

$I_8$  and  $I_{10}$

# LALR(1) Parsing Tables – (for Example2)

	<b>id</b>	*	=	\$	S	L	R
<b>0</b>	s5	s4			1	2	3
<b>1</b>				acc			
<b>2</b>			s6	r5			
<b>3</b>				r2			
<b>4</b>	s5	s4				8	7
<b>5</b>			r4	r4			
<b>6</b>	s12	s11				10	9
<b>7</b>			r3	r3			
<b>8</b>			r5	r5			
<b>9</b>				r1			

no shift/reduce or  
no reduce/reduce conflict



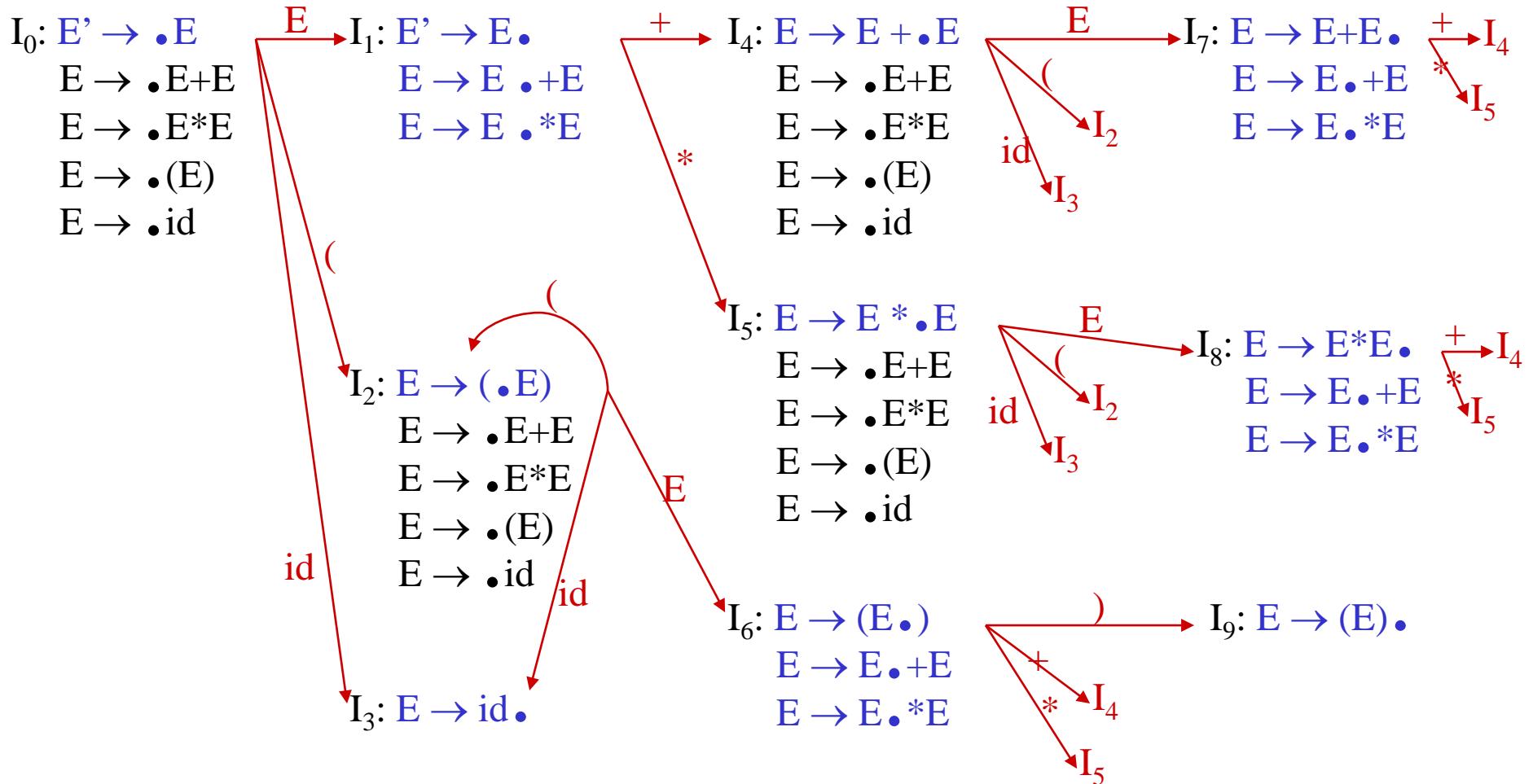
so, it is a LALR(1) grammar

# Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$$\begin{array}{ccc} E \rightarrow E+E \mid E^*E \mid (E) \mid id & \xrightarrow{\quad\quad\quad} & \begin{array}{c} E \rightarrow E+T \mid T \\ \\ T \rightarrow T^*F \mid F \\ \\ F \rightarrow (E) \mid id \end{array} \end{array}$$

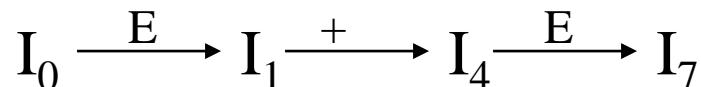
# Sets of LR(0) Items for Ambiguous Grammar



# SLR-Parsing Tables for Ambiguous Grammar

$$\text{FOLLOW}(E) = \{ \$, +, *, ) \}$$

State  $I_7$  has shift/reduce conflicts for symbols  $+$  and  $*$ .



when current token is  $+$

shift  $\rightarrow +$  is right-associative

reduce  $\rightarrow +$  is left-associative

when current token is  $*$

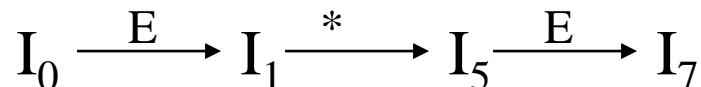
shift  $\rightarrow *$  has higher precedence than  $+$

reduce  $\rightarrow +$  has higher precedence than  $*$

# SLR-Parsing Tables for Ambiguous Grammar

$$\text{FOLLOW}(E) = \{ \$, +, *, ) \}$$

State  $I_8$  has shift/reduce conflicts for symbols  $+$  and  $*$ .



when current token is  $*$

shift  $\rightarrow *$  is right-associative

reduce  $\rightarrow *$  is left-associative

when current token is  $+$

shift  $\rightarrow +$  has higher precedence than  $*$

reduce  $\rightarrow *$  has higher precedence than  $+$

# SLR-Parsing Tables for Ambiguous Grammar

	Action	Goto					
	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		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

# Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

# Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state **s**).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow **A**.
  - The symbol **a** is simply in FOLLOW(**A**), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal **A** is normally is a basic programming block (there can be more than one choice for **A**).
  - stmt, expr, block, ...

# Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

# Syntax Directed Translation

Parsing is the act of checking that a sentence can be produced by a given grammar. However, in a compiler, one must do more than this. Eventually, code must be produced (i.e., a translation) and one way to do this is by defining semantic actions for various parts of the grammar. This is called *syntax directed translation*.

Syntax directed translation consists of a context-free grammar, a set of attributes for each grammar symbol and a collection of semantic rules associated with each production. The attributes are either *synthesized* or *inherited*. A simple way to envision this is to think of each grammar symbol as a record in the parse tree, where attributes are a name of a field in the record.

Attributes are computed by the semantic actions associated with the rule. If an attribute is computed from only child nodes, it is *synthesized*. If it computed from parent and/or sibling nodes, it is *inherited*. We call a parse tree with the attributes an *annotated parse*

*tree* and the act of adding the attributes *decorating* or *annotating* the parse tree.

## Infix to Postfix Example

$$E \rightarrow E_1 + T \{E = E_1.val \mid\mid T.val \mid\mid +\}$$

$$E \rightarrow E_1 - T \{E = E_1.val \mid\mid T.val \mid\mid -\}$$

$$E \rightarrow T \{E.val = T.val\}$$

$$T \rightarrow 0 \mid \dots \mid 9 \{T.val = \text{number}\}$$

(The  $\mid\mid$  operator is concatenation. It is assumed the *val* attribute is actually a string)

Using the above scheme, we can translate from infix to postfix for simple additive expressions. If you draw out the parse tree for some expression and annotate the parse tree, you can translate

3 + 2 - 4

into

3 2 + 4 -

by evaluating the semantic actions from the bottom-up.

It is also possible to do the same thing in a slightly different manner.

$$E \rightarrow E_1 + T \{ \text{print } ('+') ; \}$$

$$E \rightarrow E_1 - T \{ \text{print } ('-') ; \}$$

$$E \rightarrow T$$

$$T \rightarrow 0 | \dots | 9 \{ \text{print } ('%d', \text{ number}) ; \}$$

The correct translation can now be achieved by doing a postorder traversal (a.k.a. depth-first order) of the annotated parse tree.

Notice there are various ways in which to make the order of the semantic actions specific. For bottom-up parsing, the first technique is more suitable. It is known as a *simple translation scheme* because the output is merely the concatenation of the attributes for the grammar symbols on the right side of the production.

# Synthesized and Inherited Attributes

**Synthesized Attributes** are those that can be determined in terms of the attributes of child nodes. Thus, for a production  $A \rightarrow \alpha$ , we can determine the values for attributes in  $A$  by only looking at attributes in  $\alpha$ . A syntax directed translation with only synthesized attributes is called an *S-attributed definition*. S-attributed definitions are easily implemented by semantic actions in an LR parser since both evaluate from the bottom up.

**Inherited Attributes** are those that are determined in terms of attributes of sibling and/or parent nodes. If  $X$  is a grammar symbol with inherited attributes, then for some production  $A \rightarrow \alpha X \beta$ , attributes in  $X$  are determined by a combination of attributes in  $A$ ,  $\alpha$  or  $\beta$ .

# Syntax-Directed Definitions

Every grammar rule of the form  $A \rightarrow \alpha$  has a set of semantic actions associated with it. They are of the form  $b = f(c_1, c_2, \dots, c_k)$  where:

- $b$  is a synthesized attribute of  $A$  and  $c_i$  are attributes of grammar symbols of the production
- $b$  is an inherited attribute of one of the grammar symbols in  $\alpha$  and  $c_i$  are attributes of grammar symbols in the production.

In either case, we say that  $b$  *depends* on  $c_1, c_2, \dots, c_k$ .

Terminals are considered to only have synthesized attributes whose values are usually supplied by the lexical analyzer. The start symbol is assumed to have only synthesized attributes.

# Dependency Graphs

We can create a dependency graph for the attributes for a syntax directed translation by drawing a graph with a directed edge from  $c_i$  to  $b$  when attribute  $b$  depends on  $c_i$ . If a semantic action only creates a side effect (for example, a print statement or a function call), then we can introduce a dummy attribute for that action.

Attributes must be evaluated in the order defined by the dependency graph. If the dependency graph for some parse tree of the underlying grammar has a cycle, the syntax-directed definition is said to be circular. If the syntax-directed definition is circular, the attributes cannot be evaluated. However, there is no efficient algorithm to test for circularity in syntax-directed definitions.

After creating the dependency graph, we can perform a *topological sort* on the graph (assuming it has no cycles). Note that a dependency graph is a directed, acyclic graph. Thus, we can impose an

ordering on the nodes in the graph such that for any edge  $(m_i, m_j)$ ,  $i < j$ . After creating this graph, we can list, in order, the evaluation of the semantic rules of the entire syntax-directed definition.

1. Create the parse tree
2. Create the dependency graph for the parse tree
3. Do a topological sort of the dependency graph

# Syntax Trees

A *syntax tree* is a condensed form of parse tree. It used to separate translation from parsing. Usually, syntax are more convenient to deal with than full parse trees.

A node in a syntax tree is usually an operator or language construct with the arguments as children. For example:

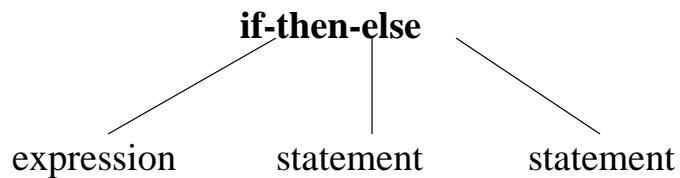


Figure 1: Syntax tree node.

The noticeable difference between a syntax tree and a parse tree is that “useless” nodes are removed.

## Example:

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.\text{nptr} = \text{mknode} ('+', E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow E_1 - T$	$E.\text{nptr} = \text{mknode} ('-', E_1.\text{nptr}, T.\text{nptr})$
$E \rightarrow T$	$E.\text{nptr} = T.\text{nptr}$
$T \rightarrow T_1 * F$	$T.\text{nptr} = \text{mknode} ('\ast', T_1.\text{nptr}, F.\text{nptr})$
$T \rightarrow T_1 / F$	$T.\text{nptr} = \text{mknode} ('\text{/}', T_1.\text{nptr}, F.\text{nptr})$
$T \rightarrow F$	$T.\text{nptr} = F.\text{nptr}$
$T \rightarrow (E)$	$T.\text{nptr} = E.\text{nptr}$
$F \rightarrow \text{id}$	$F.\text{nptr} = \text{mkleaf} (\text{id}, \text{id.entry})$
$F \rightarrow \text{num}$	$F.\text{nptr} = \text{mkleaf} (\text{num}, \text{num.val})$

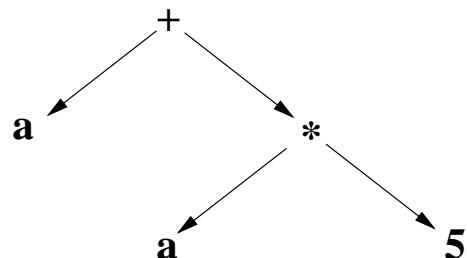


Figure 2: Syntax tree for  $a + a * 5$ .

# Directed Acyclic Graphs (DAGs)

We can use a DAG for creating syntax trees as well. The advantage to DAGs is that they avoid redundant subtrees. Essentially, any common nodes are merged into a single node. This is done by altering the procedures `mknode` and `mkleaf` to first check to see if the node already exists.

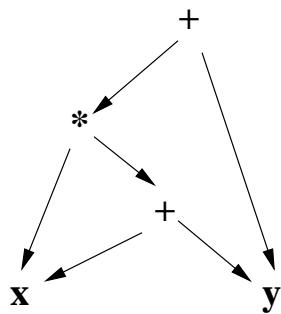


Figure 3: DAG for  $x * (x + y) + y$ .

While DAGs avoid redundant code, they can be inefficient and problematic later on if there are side effects that change values at some point.

In order to implement a DAG, usually the nodes are stored in an array and searched when a new node

is to be created. A signature can be made through a (operator, expr, expr) triple, thus allowing the value to be hashed. This creates a much faster method for searching as opposed to simply scanning a large array with all the nodes. Plus, it makes addition and removal of nodes more efficient.

# Syntax Directed Translation- Evaluation of Attributes

## Evaluation of S-attributed Definitions

Recall that an S-attributed syntax-directed definition contains only synthesized attributes. Thus, in some production  $A \rightarrow \alpha$ , we can compute attributes for  $A$  with attributes for  $\alpha$ . This makes S-attributed definitions very easily computed in conjunction with a bottom-up parser.

If we keep values for attributes on the parser stack (or another stack maintained concurrently with the parser stack), we can access these attributes when rules are reduced.

top →

State	Val
...	...
X	X.x
Y	Y.x
Z	Z.x
...	...

Table 1: Parser stack with attributes.

If we have a production  $A \rightarrow XYZ$ , we can compute

`A.x` (the attribute(s) of `A`) by accessing `val[top]`, `val[top-1]` and `val[top-2]`.

`bison/yacc` has a mechanism for accessing stack values, with the `$N` and `$$`, as we have already seen. Of course, types must be specified in order for it to work in `bison`.

## L-attributed Definitions

L-attributed definitions encompass a larger class than that of S-attributed definitions. A syntax-directed definition is L-attributed if each inherited attribute of  $X_j$  on the right side of  $A \rightarrow X_1 X_2 \dots X_n$  with  $1 \leq j \leq n$  only depends on

1. The symbols  $X_1 \dots X_{j-1}$ .
2. The inherited attributes of  $A$ .

Note that an S-attributed grammar is also L-attributed since it doesn't contain any inherited attributes and the above restrictions apply only to definitions with inherited attributes (i.e., it's trivially true).

# Evaluating L-attributes

The definition of L-attributes lends itself very nicely to a depth-first traversal for evaluation (that is, start at the root and recursively visit the children left-to-right). This is because symbols on the right side of a production can only rely on symbols to the left of that symbol. Thus, you can evaluate the attributes with a function:

```
function dfvisit (node)
  foreach child m of node from left to right do
    evaluate inherited attributes of m
    dfvisit (m)
  end
  evaluate synthesized attributes of node
```

# Translation Schemes

A translation scheme is a context-free grammar where attributes are associated with each symbol and actions are specified within the productions, usually with braces ( $\{\}$ ). Note that this is not the same as a syntax-directed definition where semantics have simply been associated with the grammar rules. We must actually specify *when* the actions are to take place.

Here again is an infix-to-postfix translator, as a translation scheme:

$$E \rightarrow TR$$

$$R \rightarrow \text{addop } T \{ \text{print(addop.val)} \} R_1$$

$$R \rightarrow \epsilon$$

$$T \rightarrow \text{num } \{ \text{print(num.val)} \}$$

With S-attributed definitions, translation schemes are essentially the same as the definition (attributes

are computed at the end). However, with L-attributed definitions, you must follow some rules:

1. An inherited attribute for a symbol must be computed in an action before that symbol.
2. No action can refer to a synthesized attribute of a symbol to the right.
3. A synthesized attribute is computed at the end of the rule.

**Example:**

$$S \rightarrow A_1 A_2 \{A1.in = 1; A2.in = 2;\}$$
$$A \rightarrow a \{\text{print } (A.in);\}$$

This translation scheme fails condition 1. To fix it, we must put the action before the print statement:

$$S \rightarrow \{A1.in = 1; A2.in = 2;\} A_1 A_2$$
$$A \rightarrow a \{\text{print } (A.in);\}$$

# Evaluating L-attributed Definitions

## Bottom-up

While it may not seem intuitive, it is possible to evaluate L-attributed definitions using a bottom-up technique. If you haven't noticed, all L-attributed definitions have been evaluated top-down (like the previous example). Earlier, bottom-up evaluation relied on actions being executed just prior to the rule being reduced.

To perform actions inside of a production (like the previous translation scheme), you insert "dummy" rules whose sole purpose is to execute a given action.

$$E \rightarrow TR$$

$$R \rightarrow \text{addop } T M R_1$$

$$M \rightarrow \epsilon \{ \text{print(addop.val)} \}$$

$$R \rightarrow \epsilon$$

$$T \rightarrow \text{num } \{ \text{print(num.val)} \}$$

bison forces you to put actions at the end of the rule, but you can embed actions inside the rule. In this case, bison will automatically generate a dummy rule to perform the action given.

## A Caveat:

Inserting these embedded actions can cause shift/reduce conflicts. Observe:

```
%%
thing: abcd | abcz
abcd: 'A' 'B' 'C' 'D'
abcz: 'A' 'B' 'C' 'D'
-----
%%
thing: abcd | abcz
abcd: 'A' 'B' {action} 'C' 'D'
abcz: 'A' 'B' 'C' 'D'
```

The first is unambiguous. The second, with the dummy rule, makes it such that the parser does not know whether to shift 'C' or reduce the dummy rule.

Part of the reason that these dummy rules work is that when the rule is entered, the context is known and it is allowed to reference items below itself on the stack (note that each dummy rule is uniquely named, else this wouldn't work). If context can be *assured*, it is quite legal to reference below the top of the stack to get attribute values. (see Tables 3 and 4)

When context cannot be assured, you can use dummy variables to make sure back-references work.

$S \rightarrow aAC$	C.i = A.s
$S \rightarrow bABC$	C.i = A.s
$C \rightarrow c$	C.s = g(C.i)
<hr/>	
$S \rightarrow aAC$	C.i = A.s
$S \rightarrow bABMC$	M.i = A.s; C.i = M.s
$M \rightarrow \epsilon$	M.s = M.i
$C \rightarrow c$	C.s = g(C.i)

Table 2: Inserting  $M$  allows assures that C.i gets the needed value.

Translation scheme	Code Fragment
$D \rightarrow T \{L.in = T.type\} L$	
$T \rightarrow \text{int } \{T.type = \text{integer}\}$	$\text{val}[ntop] = \text{integer}$
$T \rightarrow \text{real } \{T.type = \text{real}\}$	$\text{val}[ntop] = \text{real}$
$L \rightarrow \{L1.in = L.in\}$	
$L_1, \text{id } \{\text{addtype}\}$	$\text{addtype}(\text{val}[top], \text{val}[top - 3])$
$L \rightarrow \text{id } \{\text{addtype}\}$	$\text{addtype}(\text{val}[top], \text{val}[top-1])$

Table 3: ‘top’ (top of the stack before a reduction), ‘ntop’ (after).

Input	State	Production
real p, q, r	-	
p, q, r	<b>real</b>	
p, q, r	$T$	$T \rightarrow \text{real}$
, q, r	$T p$	
, q, r	$T L$	$L \rightarrow \text{id}$
q, r	$T L ,$	
,	$T L , q$	
,	$T L$	$L \rightarrow L, \text{id}$
r	$T L ,$	
	$T L , r$	
	$T L$	$L \rightarrow L, \text{id}$
	$D$	$D \rightarrow TL$

Table 4: Note how  $T$  is always just below  $L$ .

# Avoiding Inherited Attributes

One way of dealing with inherited attributes is to avoid them whenever possible. This usually involves changing the grammar in some way. For identifiers, you could make the type come at the end of the declaration.

Another method, if possible based on the dependencies, is to do a combination of bottom-up and top-down. While parsing and building a parse/syntax tree, you calculate synthesized attributes. When you get to the point where you have the needed values, you descend the tree and calculate as needed.

For example, the identifier declarations in C. Since you can have a long list of identifiers with the same type, build the list of identifiers then, when you get to the node to reduce it to a declaration, run through the list and insert the type into the symbol table for each identifier.

# Syntax directed translation

Syntax directed translation

Grammar + Semantic rules = SDT

SDT for evaluation of expression

$E \rightarrow E_1 + T \quad \{ E.value = E_1.value + T.value \}$   
 $/ T \quad \{ E.value = T.value \}$

$T \rightarrow T_1 * F \quad \{ T.value = T_1.value * F.value \}$   
 $/ F \quad \{ T.value = F.value \}$

$F \rightarrow \text{num} \quad \{ F.val = \text{num}.lvalue \}$

✓

```
graph TD; Root["2 + 3 * 4"] -- "+" --> Node2["E.val = 14"]; Root -- "*" --> Node3["T.val = 12"]; Root -- "F" --> Node4["F.val = 4"]; Node2 -- "2" --> Node5["num 2"]; Node3 -- "3" --> Node6["T.val = 3"]; Node3 -- "4" --> Node7["F.val = 4"]; Node5 -- "2" --> Node8["num 3"]; Node6 -- "3" --> Node9["num 4"];
```

SOT

$E \rightarrow E + T \{ \text{printf}( "+") ; \}$

$/ T \quad \{ \}$  ②

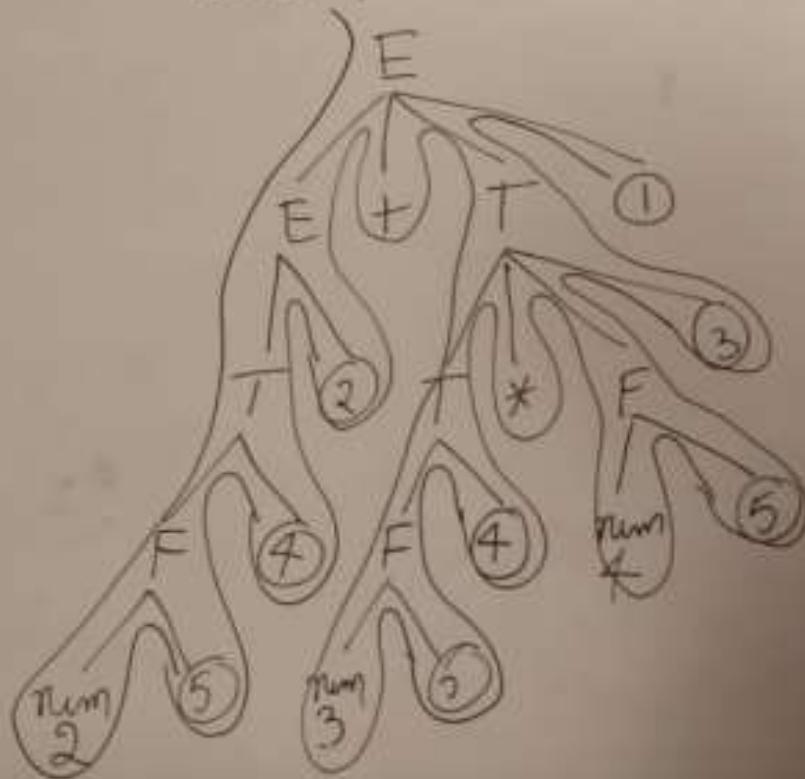
$T \rightarrow T * F \{ \text{printf}( "*") ; \}$

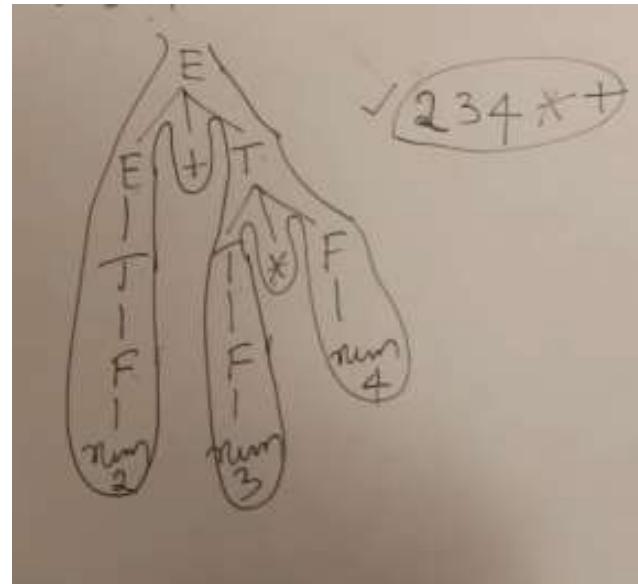
$/ F \quad \{ \}$  ③

$F \rightarrow \text{num} \{ \text{printf}( \text{num}.lval) ; \}$

2 + 3 \* 4.

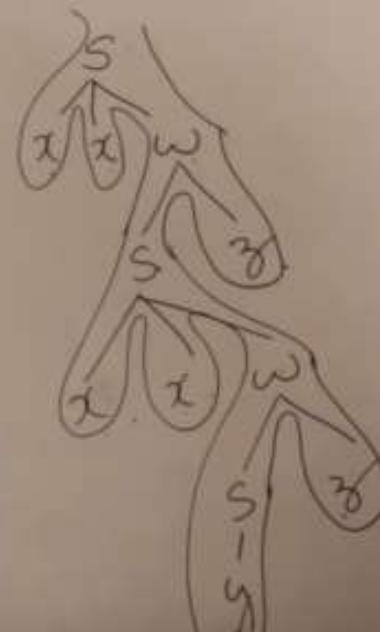
2 3 4 \* +





$S \rightarrow xxw \{ \text{printf}(1); \}$   
 $y \quad \{ \text{printf}(2); \}$   
 $w \rightarrow s z \{ \text{printf}(3); \}$

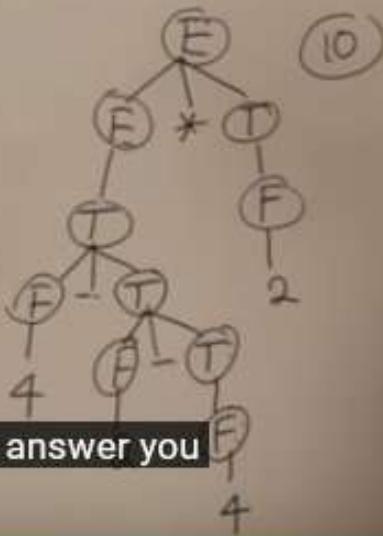
String  $xxxyzz$



[2 3 1 3 1]

$E \rightarrow E, * T \quad \{E.\text{val} = E.\text{val} * T.\text{val},\}$   
 /  $T \quad \{E.\text{val} = T.\text{val},\}$   
 $T \rightarrow F - T \quad \{T.\text{val} = F.\text{val} - T.\text{val},\}$   
 /  $F \quad \{T.\text{val} = F.\text{val},\}$   
 $F \rightarrow 2 \quad \{F.\text{val} = 2,\}$   
 /  $4 \quad \{F.\text{val} = 4,\}$

$$w = ((4 - (2 - 4)) * 2) \quad 12$$



production and find out the answer you  
can

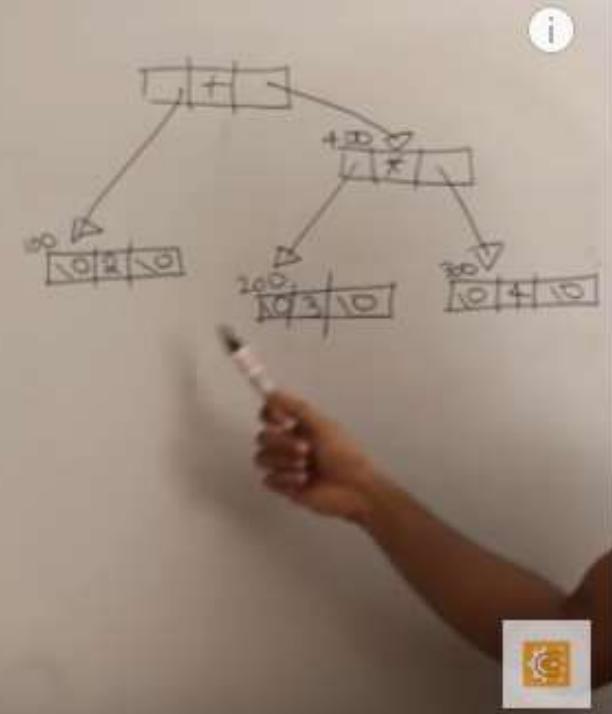
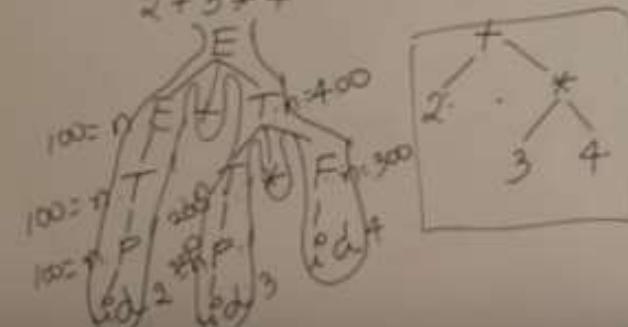
SLD to build Syntax tree:-

$E \rightarrow E + T \quad \{E.\text{nptn} = \text{mknode}(E.\text{nptn}, '+', T.\text{nptn});\}$   
  |  
   $T \quad \{E.\text{nptn} = T.\text{nptn};\}$

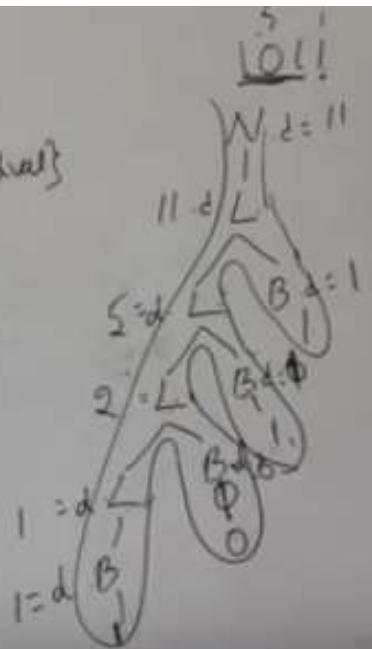
$T \rightarrow T * F \quad \{T.\text{nptn} = \text{mknode}(T.\text{nptn}, '*', F.\text{nptn});\}$   
  |  
   $F \quad \{T.\text{nptn} = F.\text{nptn};\}$

$F \rightarrow \text{id.} \quad \{F.\text{nptn} = \text{mknode}(\text{null}, \text{id.name}, \text{null});\}$

$2 + 3 * 4$



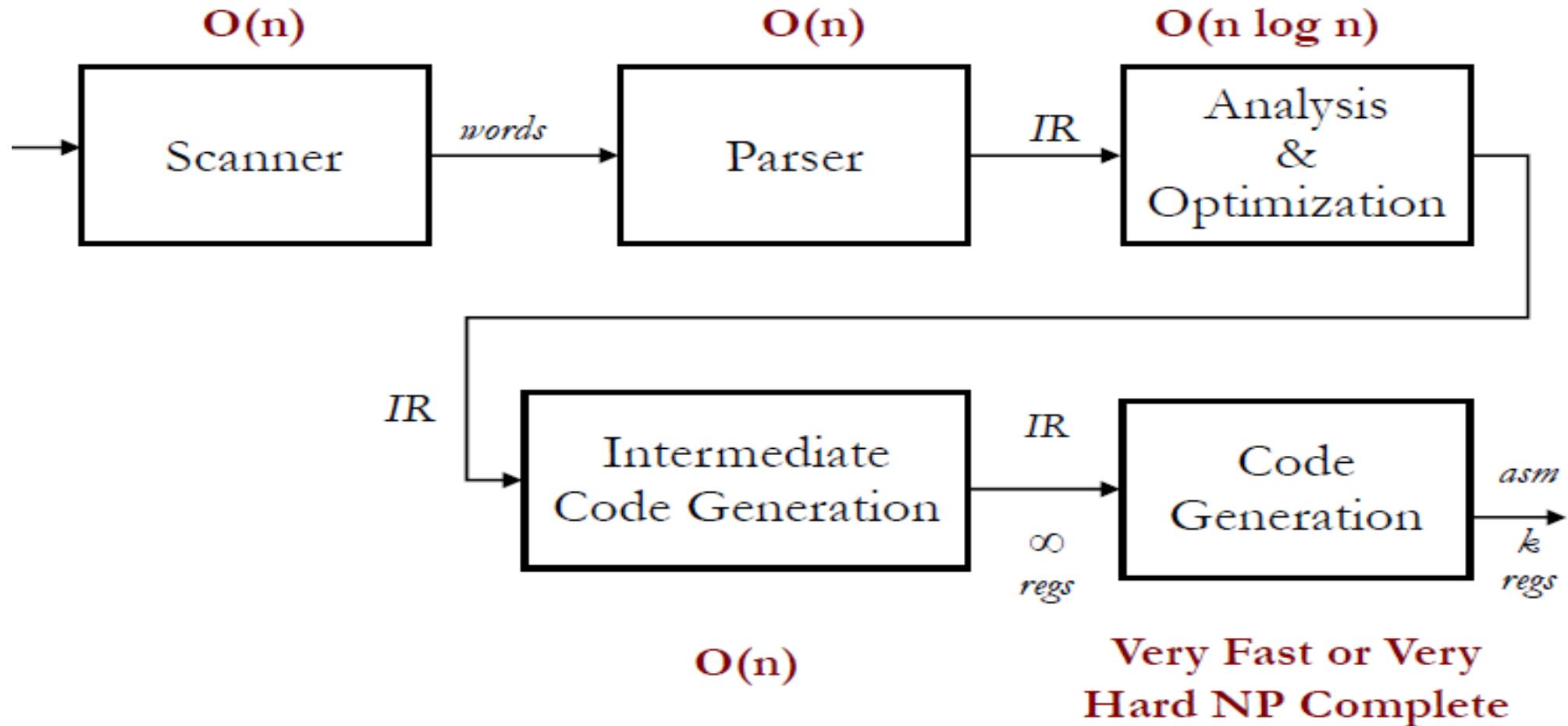
$$\begin{array}{l}
 N \rightarrow L \quad \left\{ \begin{array}{l} N \cdot \text{dual} = L \cdot \text{dual} \end{array} \right\} \\
 L \rightarrow L, B \quad \left\{ \begin{array}{l} L \cdot \text{dual} = L, \text{dual} * 2 + B \cdot \text{dual} \end{array} \right\} \\
 /B \quad \left\{ \begin{array}{l} L \cdot \text{dual} = B \cdot \text{dual} \end{array} \right\} \\
 B \rightarrow O \quad \left\{ \begin{array}{l} B \cdot \text{dual} = 0 \end{array} \right\} \\
 /I \quad \left\{ \begin{array}{l} B \cdot \text{dual} = 1 \end{array} \right\}
 \end{array}$$



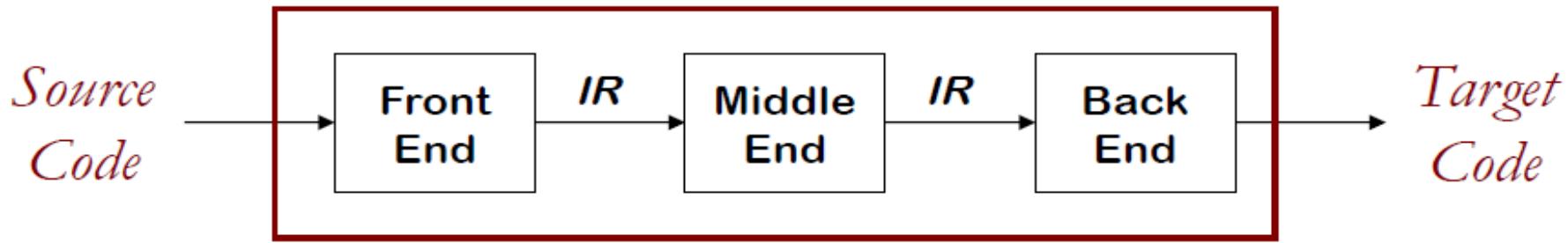
# CS 304 Compiler Design

Intermediate Code Generation

# Structure of a Compiler

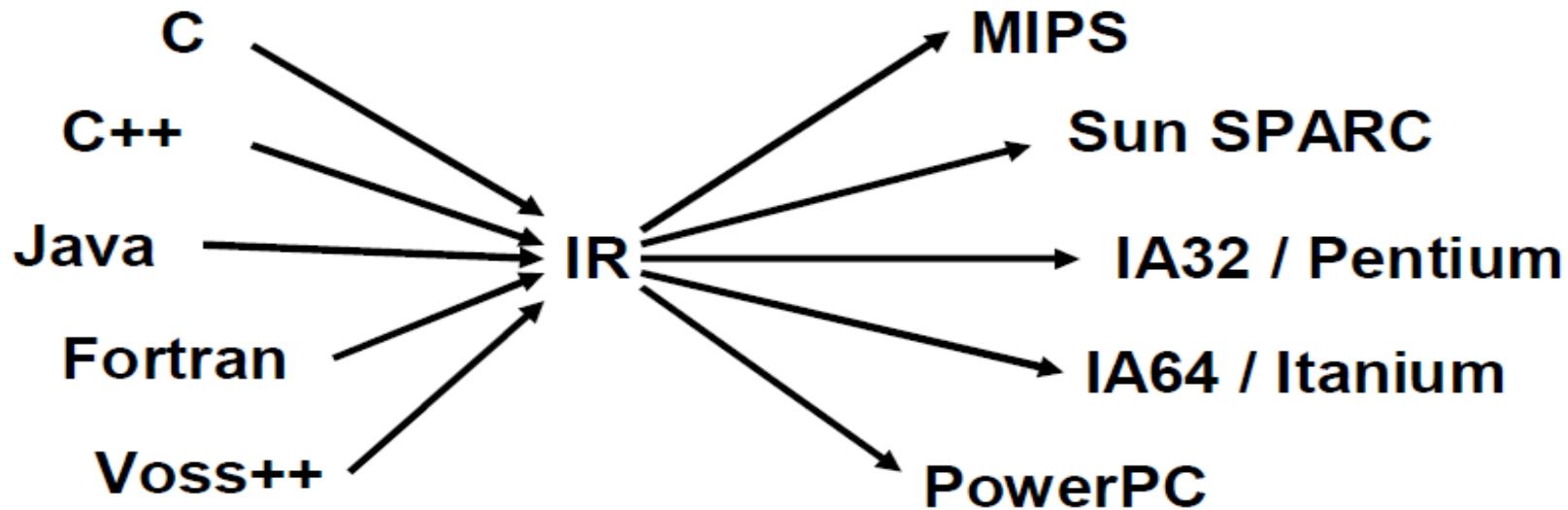


# Intermediate Representations



- Front End - Produces an Intermediate Representation (*IR*)
  - Middle End - Transforms the *IR* into an equivalent *IR* that runs more efficiently
  - Back End - Transforms the *IR* into native Code
- 
- *IR* Encodes the compiler's Knowledge of the Program
  - Middle End usually consists of Several Passes

# Why Use an IR?



- Good Software Engineering
  - Portability
  - Reuse

# Important IR properties

- Decisions in *IR* design affect the speed and efficiency of the compiler
- Some important *IR* Properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction
- Importance of different properties varies
  - Selecting an appropriate *IR* for a compiler is critical

# Types of Intermediate Representations

Three Major Categories

- Structural
    - Graphically oriented
    - Heavily used in source-to-source translators
    - Tend to be large
  - Linear
    - Pseudo-code for an abstract machine
    - Level of abstraction varies
    - Simple, compact data structures
    - Easier to rearrange
  - Hybrid
    - Combination of graphs and linear code
- Examples:  
Trees, DAGs
- Examples:  
3 address code  
Stack machine code
- Example:  
Control-Flow Graph

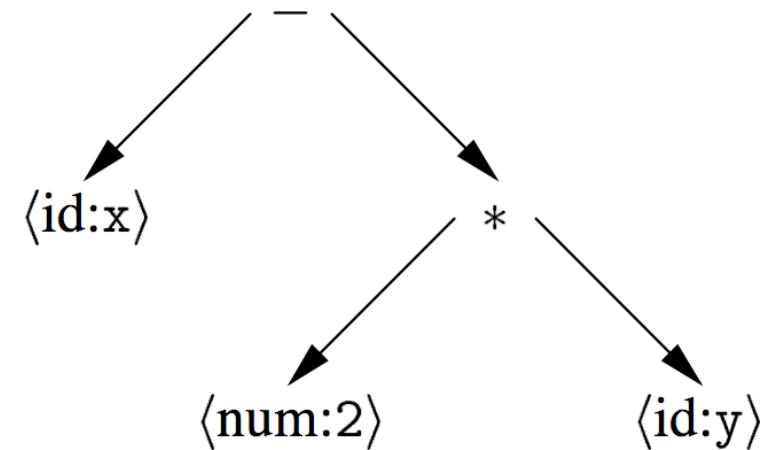
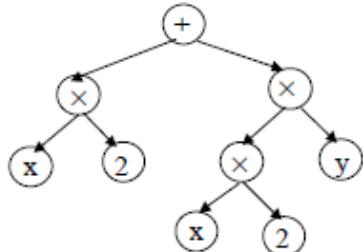
# Abstract syntax tree

An AST is a parse tree with nodes for most non-terminals removed.

*Since the program is already parsed, non-terminals needed to establish precedence and associativity can be collapsed!*

redundancy in  
Representation

$x \times 2 + x \times 2 \times y$



A linear operator form of this tree  
(postfix) would be:

x 2 y \* -

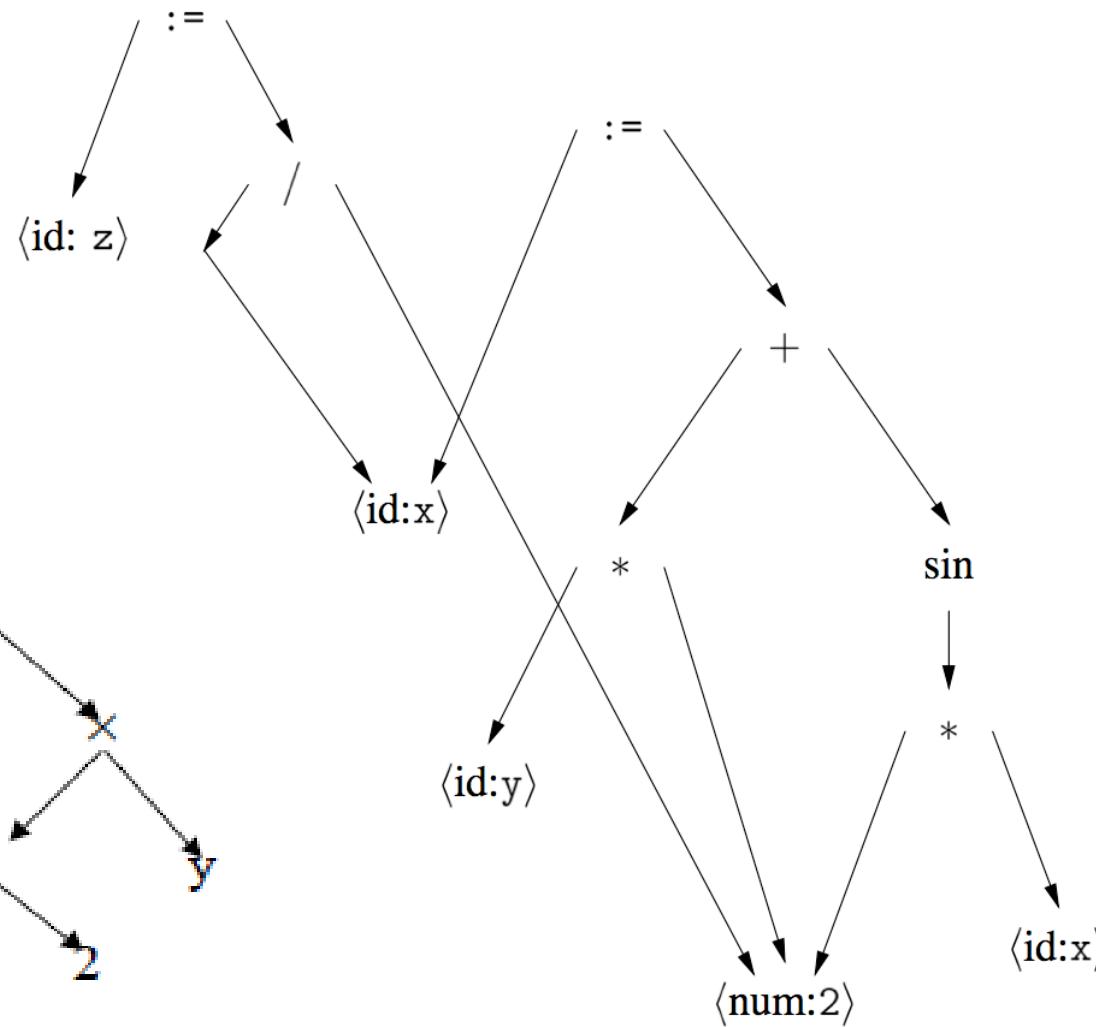
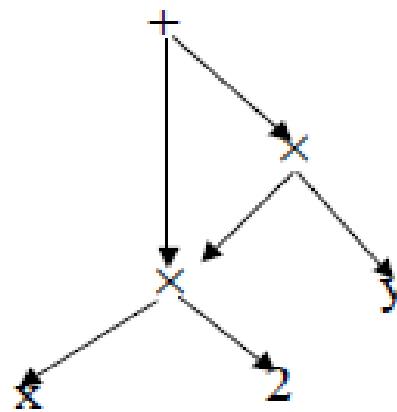
# Directed acyclic graph

A DAG is an AST with unique, shared nodes for each value.

```
x := 2 * y + sin(2*x)
z := x / 2
```

- eliminates redundancy by using a graph instead of a tree
- Good for simple optimization, not for CSA

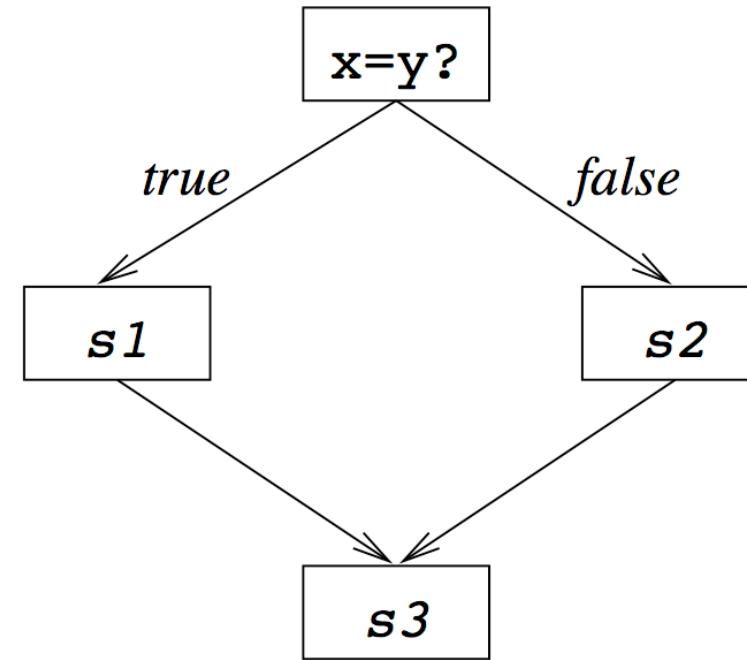
$x \times 2 + x \times 2 \times y$



# Control flow graph

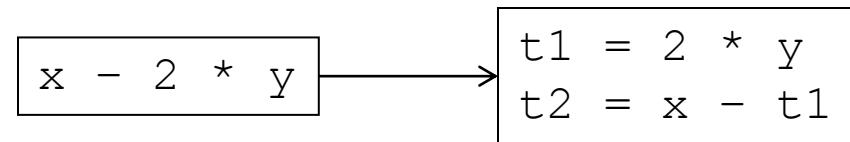
- A CFG models *transfer of control* in a program
  - nodes are basic blocks (straight-line blocks of code)
  - edges represent *control flow* (loops, if/else, goto ...)

```
if x = y then  
    S1  
else  
    S2  
end  
S3
```



# 3-address code

- Statements take the form:  $x = y \text{ op } z$ 
  - single operator and at most three names



- > Advantages:
  - compact form
  - names for intermediate values

# Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

$$x \leftarrow y \text{ } op \text{ } z$$

With 1 operator (op) and, at most, 3 names (x, y, & z)

Example:

$$z \leftarrow x - 2 * y$$

becomes

$$\begin{array}{l} t \leftarrow 2 * y \\ z \leftarrow x - t \end{array}$$

Advantages:

- Resembles many machines
- Introduces a new set of names t
- Compact form

# Three Address Instructions

There are a variety of TAC statements available:

- Assignments:  $x := y \text{ op } z$  where op is a binary operator (arithmetic or logical)
- Assignments:  $x := \text{op } y$  where op is a unary operator (minus, logical, etc.)

- Copy:  $x := y$
- Jump: `goto L` where L is a label
- Conditional Jump: `if x relop y then goto L` where `relop` is a relational operator that generates a true or false value (usually 1 or 0). Some IRs only allow `if x goto L`
- Indexed Statements:  $x[i] := y$  or  $x := y[i]$  where i is the offset from memory location x or y.
- Address/Pointer:  $x := \&y$ ,  $x := *y$ ,  $*x := y$
- Procedure Calls:

param 1

param 2

...

param n call proc, n

the expression  $x = a * b + c$  would look like this  
in three-address code:

```
t1 := a * b
t2 := t1 + c
x  := t2
```

Temporaries are explicit in three-address code.

# Function Call Example

Source Code	Three Address Instructions
<pre>y = p(a, b+1)</pre>	<pre>t1 = a t2 = b + 1 putparam t1 putparam t2 y = call p, 2</pre>
<pre>int p(x, z) {     return x+z; }</pre>	<pre>getparam z getparam x t3 = x + z return t3 return</pre>

# Loop Example

Source Code

```
do  
    i = i + 1;  
while (a[i] < v);
```

Three Address Instructions

```
L: t1 = i + 1  
    i = t1  
    t2 = i * 8  
    t3 = a[t2]  
    if t3 < v goto L
```

# Loop Example

Source Code

```
do  
    i = i + 1;  
while (a[i] < v);
```

Three Address Instructions

```
L: t1 = i + 1  
    i = t1  
t2 = i * 8  
    t3 = a[t2]  
    if t3 < v goto L
```

Where did this  
come from ?

## Implementing Three-Address Code

The primary methods of implementing three-address code is with triples or quadruples. Triples are more of a space-saving method and more useful when memory is tight. Quadruples are simpler and if space allows, it is this author's opinion that you use them.

If you've noticed with TAC, there are up to four parts: the operator and one to three operands. We can represent this with a table:

Op	arg1	arg2	result
+	y	z	x
:=	y		x
param	x1		
goto	label		
relop	x	y	label

Quadruple representation

# Three Address Code: Quadruples

Naïve representation of three address code

- Table of  $k * 4$  small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used “quads”

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	y	
loadI	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruples

# SDT for Three Address Code Generation

- Attributes for the Non-Terminals, E and S
  - Location (in terms of temporary variable) of the value of an expression: E.place. If E.place is  $t_1$  it means that the value of E is saved in  $t_1$ .
  - The Code that Evaluates the Expressions or Statement: E.code
  - Markers for beginning and end of sections of the code S.begin, S.end
    - For simplicity these are symbolic labels.
    - Markers are inherited attributes
- Semantic Actions in Productions of the Grammar
  - Functions to create temporaries newtemp, and labels newlabel
  - Auxiliary functions to enter symbols and lookup types corresponding to declarations in a *symbol table*.
  - To generate the code we use the function gen which creates a list of instructions to be emitted later and can generate symbolic labels corresponding to next instruction of a list.
  - Use of append function on lists of instructions.
  - Generate code in post-order traversal of the AST

# Assignment Statements

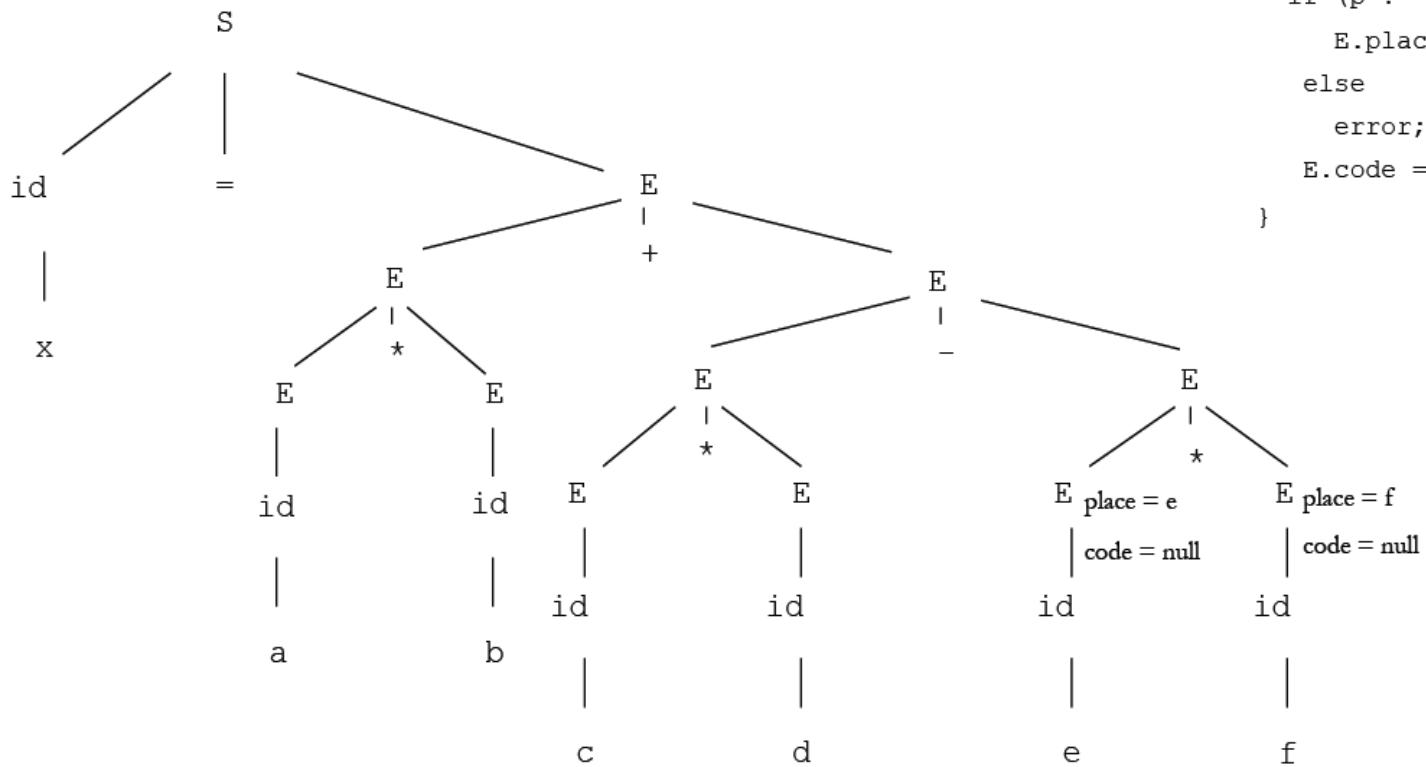
```
S → id = E { p = lookup(id.name);
              if (p != NULL) {
                  S.code = gen(p '=' E.place);
              } else {
                  error;
                  S.code = nulllist;
              }
}

E → E1 + E2 { E.place = newtemp();
                     E.code = append(E1.code, E2.code,
                                     gen(E.place '=' E1.place '+' E2.place));
}
E → E1 * E2 { E.place = newtemp();
                     E.code = append(E1.code, E2.code,
                                     gen(E.place '=' E1.place '*' E2.place));
}
E → - E1 { E.place = newtemp();
               E.code = append(E1.code, gen(E.place '=' '-' E1.place)); }

E → (E1) { E.place = E1.place; E.code = E1.code; }
E → id { p = lookup(id.name);
           if (p != NULL)
               E.place = p;
           else
               error;
           E.code = nulllist;
}
```

# Assignment: Example

$x = a * b + c * d - e * f;$

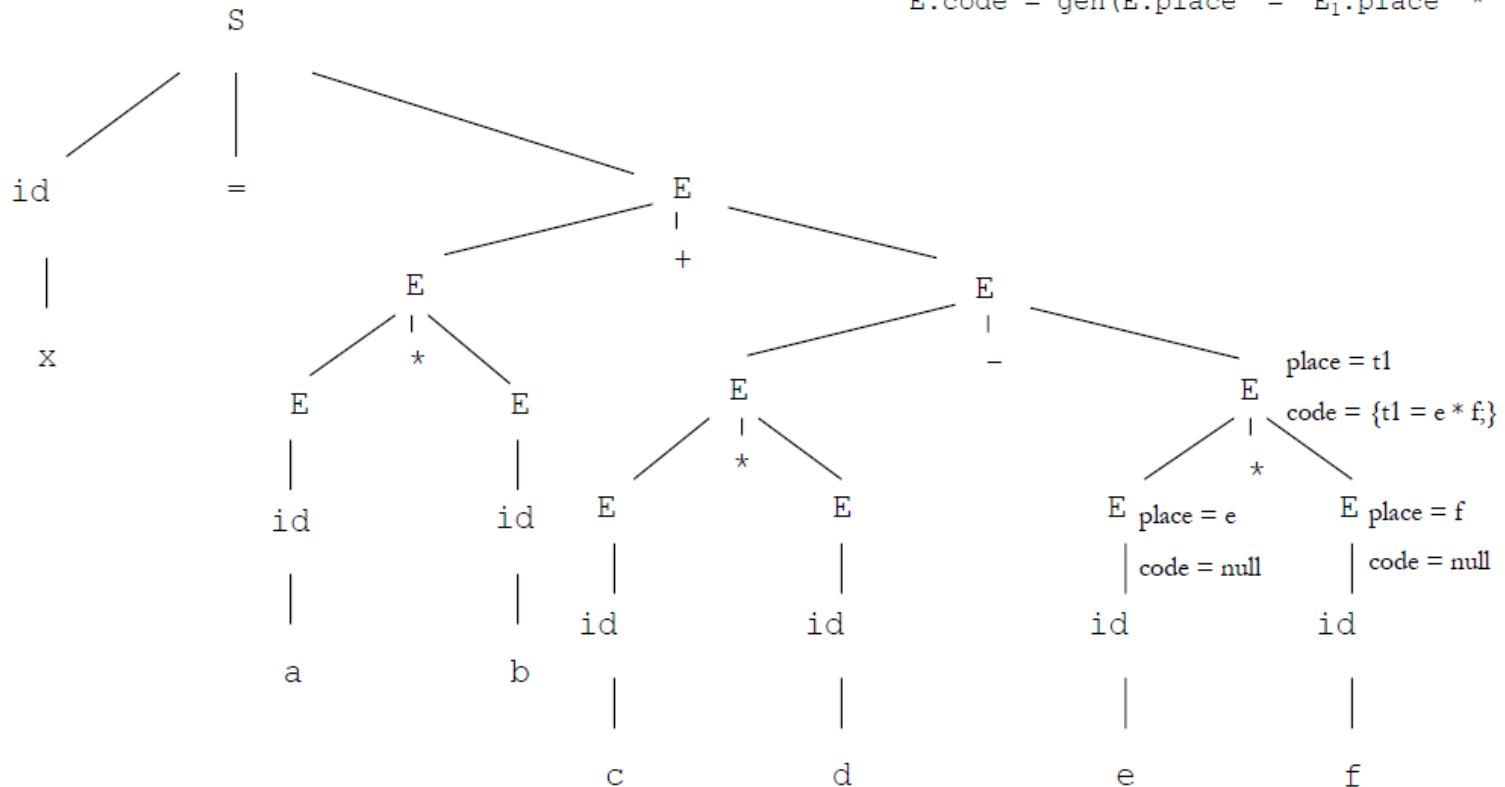


Production:

```
E → id      { p = lookup(id.name);  
               if (p != NULL)  
                 E.place = p;  
               else  
                 error;  
               E.code = null list;  
 }
```

# Assignment: Example

$x = a * b + c * d - e * f;$

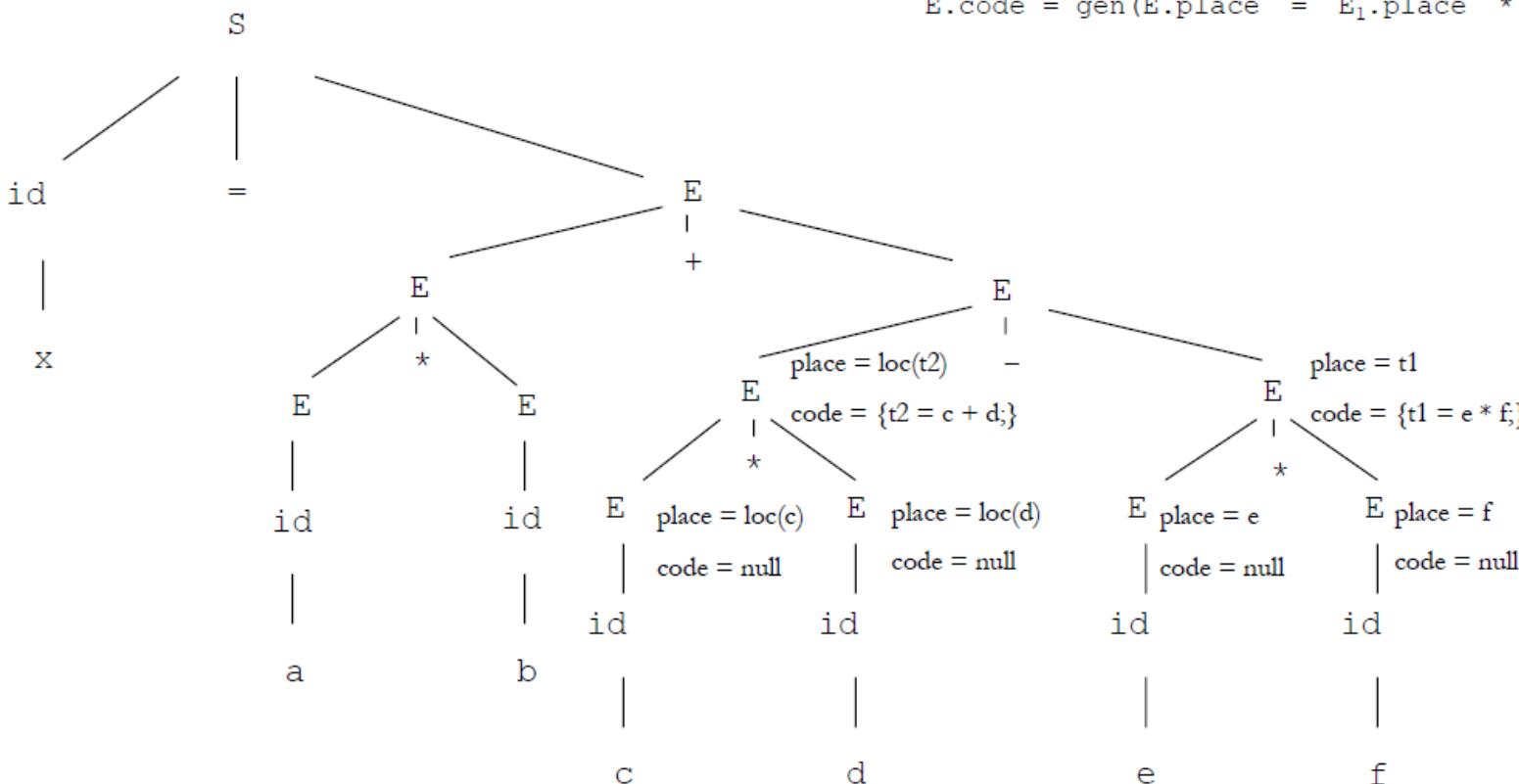


Production:

```
E → E1 * E2 {E.place = newtemp();  
E.code = gen(E.place '=' E1.place '*' E2.place);}
```

# Assignment: Example

$x = a * b + c * d - e * f;$



Production:

$E \rightarrow E_1 * E_2 \quad \{E.place = newtemp();$

$E.code = gen(E.place '=' E_1.place '*' E_2.place); \}$

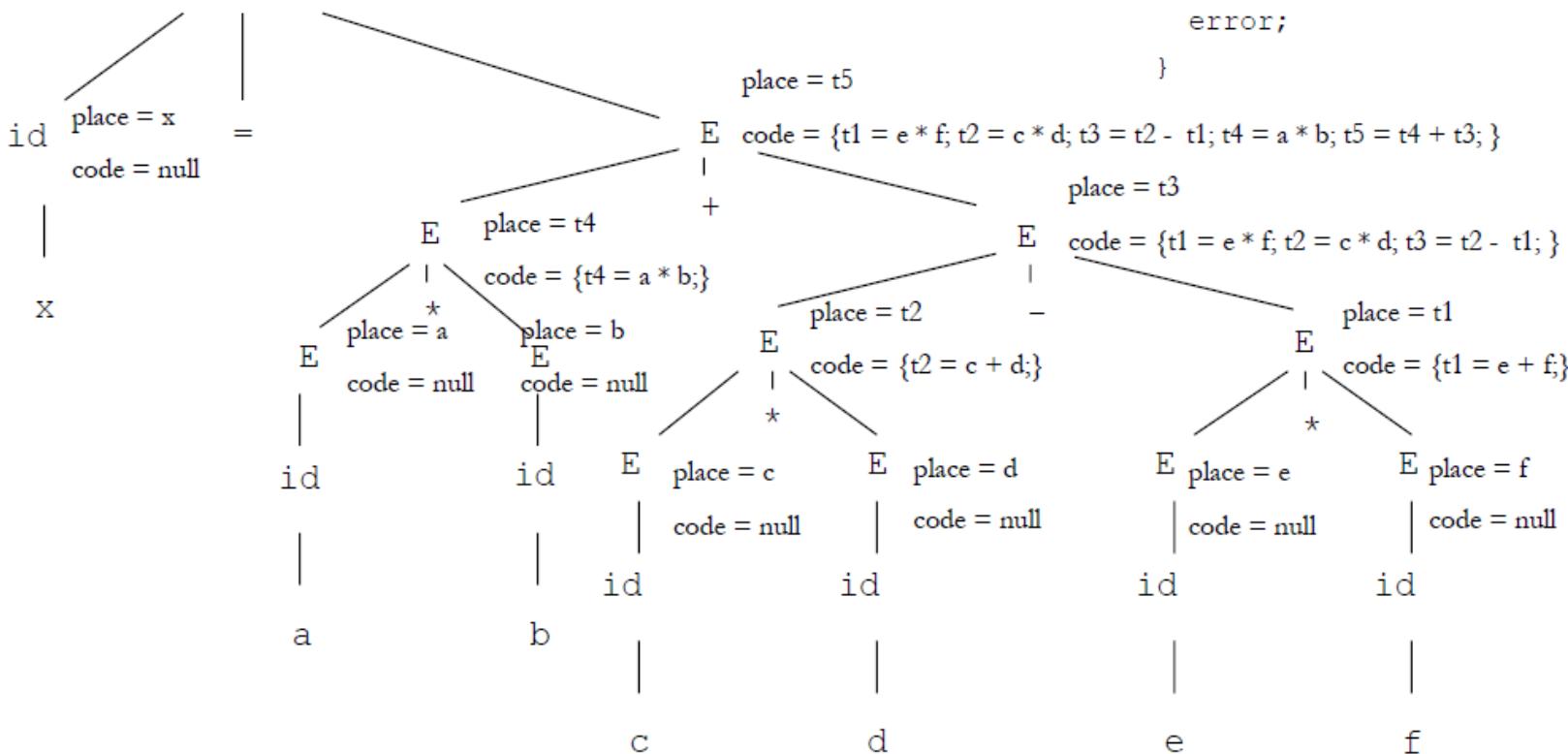
# Assignment: Example

Production:

$x = a * b + c * d - e * f;$

```
s → id = E { p = lookup(id.name);
if (p != NULL)
    E.code = append(E.code,
                    gen(p '=' E.place));
else
    error;
}
```

S code = {t1 = e \* f; t2 = c \* d; t3 = t2 - t1; t4 = a \* b; t5 = t4 + t3; x = t5; }



# Assignment: Example

x = a \* b + c \* d - e \* f;

t1 = e \* f;

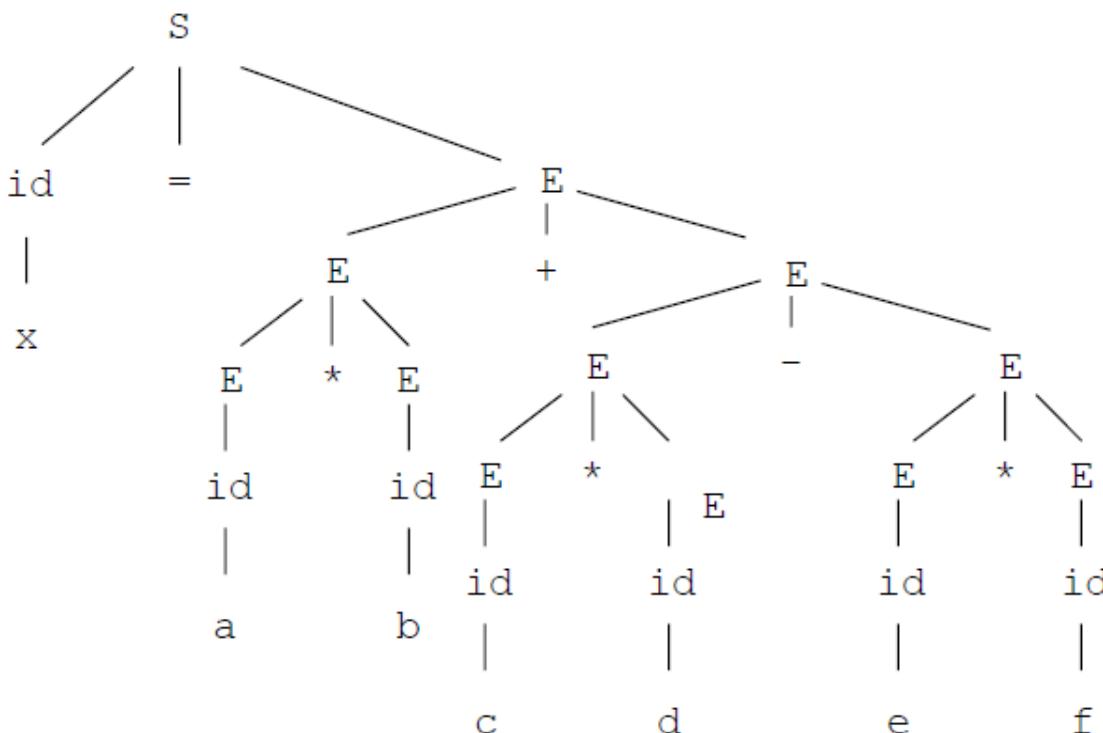
t2 = c \* d;

t3 = t2 - t1;

```
t4 = a * b;
```

t5 = t4 + t3;

x = t5;

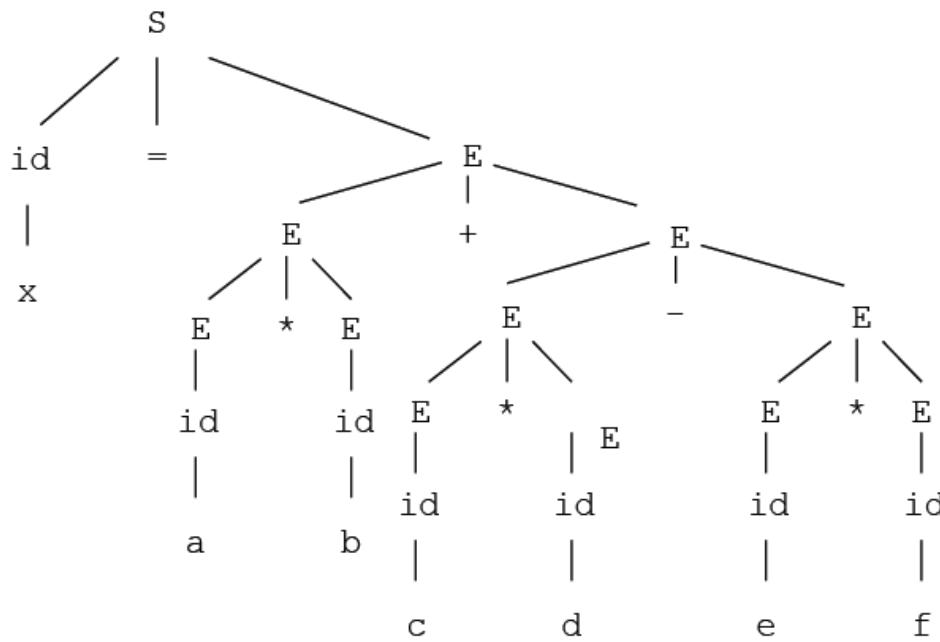


# Reusing Temporary Variables

- Temporary Variables
  - Short lived
  - Used for Evaluation of Expressions
  - Clutter the Symbol Table
- Change the newtemp Function
  - Keep track of when a value created in a temporary is used
  - Use a counter to keep track of the number of active temporaries
  - When a temporary is used in an expression decrement counter
  - When a temporary is generated by newtemp increment counter
  - Initialize counter to zero (0)
- Alternatively, can be done as a post-processing pass...

## Assignment: Example

x = a \* b + c \* d - e \* f;



```
// c = 0  
t1 = e * f;           // c = 1  
t2 = c * d;           // c = 2  
t1 = t2 - t1;         // c = 1  
t2 = a * b;           // c = 2  
t1 = t2 + t1;         // c = 1  
x = t1;               // c = 0
```

- Only 2 Temporary Variables and hence only 2 Registers are Needed

# SDT Scheme for Boolean Expressions

- Two Basic Code Generation Flavors
  - Use boolean **and**, **or** and **not** instructions (like arithmetic).
  - Control-flow (or positional code) defines **true** or **false** of predicate.
- Arithmetic Evaluation
  - Simpler to generate code as just eagerly evaluate the expression.
  - Associate ‘1’ or ‘0’ with outcome of predicates and combine with logic instructions.
  - Use the same SDT scheme explained for arithmetic operations.
- Control Flow Evaluation (**short circuit evaluation - later**)
  - More efficient in many cases.
  - Complications:
    - Need to Know Address to Jump To in Some Cases
    - Solution: Two Additional Attributes
      - `nextstat` (Inherited) Indicates the next symbolic location to be generated
      - `laststat` (Synthesized) Indicates the last location filled
      - As code is generated down and up the tree attributes are filled with the correct values

# Arithmetic Scheme: Grammar and Actions

```
E → false    || E.place = newtemp()
          E.code = {gen(E.place = 0)}
          E.laststat = E.nextstat + 1

E → true     || E.place = newtemp()
          E.code = {gen(E.place = 1)}
          E.laststat = E.nextstat + 1

E → (E1)  || E.place = E1.place;
          E.code = E1.code;
          E1.nextstat = E.nextstat
          E.laststat = E1.laststat

E → not E1 || E.place = newtemp()
          E.code = append(E1.code, gen(E.place = not E1.place))
          E1.nextstat = E.nextstat
          E.laststat = E1.laststat + 1
```

# Arithmetic Scheme: Grammar and Actions

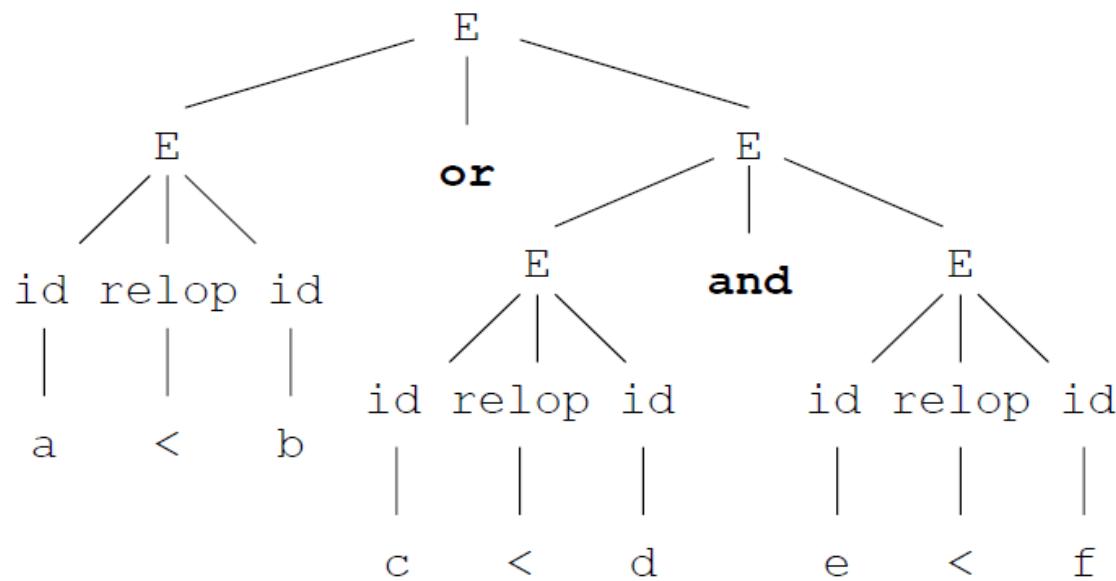
```
E → E1 or E2 || E.place = newtemp()
    E.code = append(E1.code, E2.code, gen(E.place = E1.place or E2.place))
    E1.nextstat = E.nexstat
    E2.nextstat = E1.laststat
    E.laststat = E2.laststat + 1

E → E1 and E2 || E.place = newtemp()
    E.code = append(E1.code, E2.code, gen(E.place = E1.place and E2.place))
    E1.nextstat = E.nexstat
    E2.nextstat = E1.laststat
    E.laststat = E2.laststat + 1

E → id1 relop id2 || E.place = newtemp()
    E.code = gen(if id1.place relop id2.place goto E.nextstat+3)
    E.code = append(E.code, gen(E.place = 0))
    E.code = append(E.code, gen(goto E.nextstat+2))
    E.code = append(E.code, gen(E.place = 1))
    E.laststat = E.nextstat + 4
```

# Boolean Expressions: Example

a < b or c < d and e < f

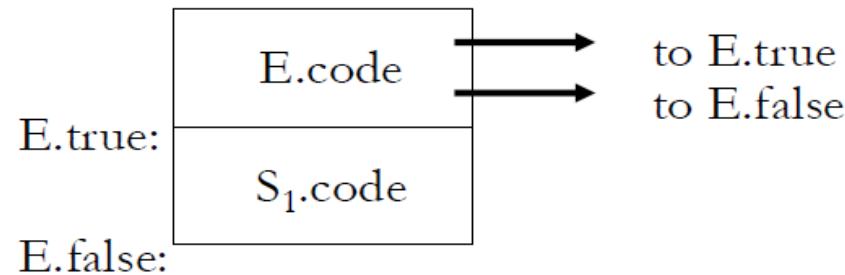


00: if a < b goto 03  
01: t1 = 0  
02: goto 04  
03: t1 = 1  
04: if c < d goto 07  
05: t2 = 0  
06: goto 08  
07: t2 = 1  
08: if e < f goto 11  
09: t3 = 0  
10: goto 12  
11: t3 = 1  
12: t4 = t2 and t3  
13: t5 = t1 or t4

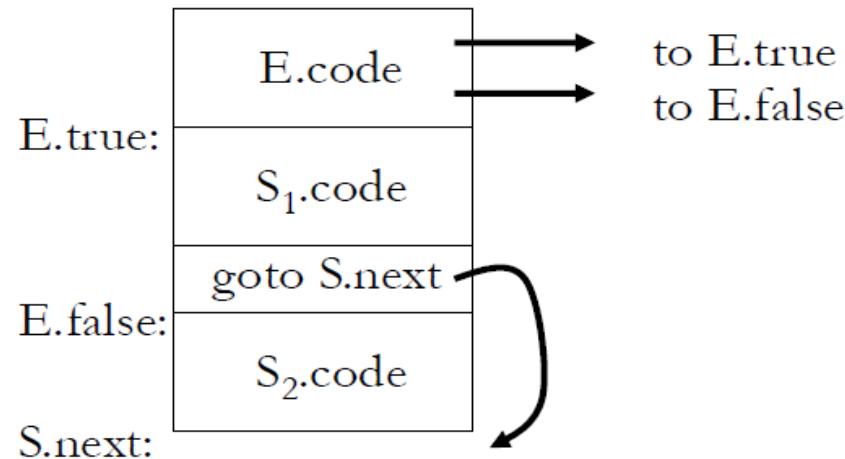
# Control Flow Statements: Code Layout

- Attributes:
  - $E.\text{true}$ : label to which control flows if  $E$  is true (Inherited)
  - $E.\text{false}$ : label to which control flows if  $E$  is false (Inherited)
  - $S.\text{next}$ : inherited attribute with the symbolic label of the code following  $S$

$S \rightarrow \text{if } E \text{ then } S_1$

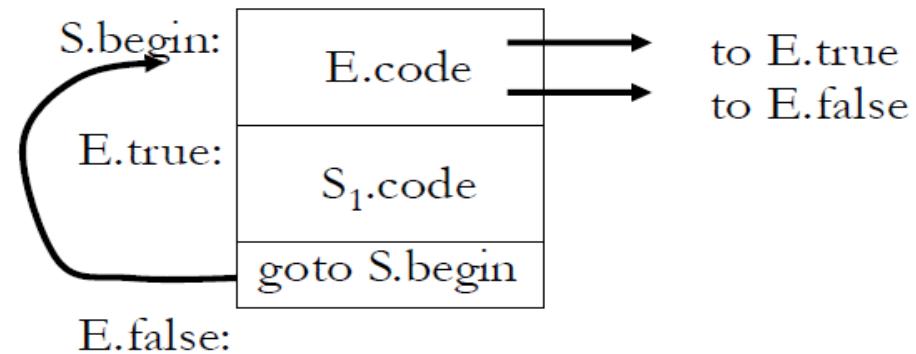


$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$



# Code Layout

$S \rightarrow \text{while } E \text{ do } S_1$

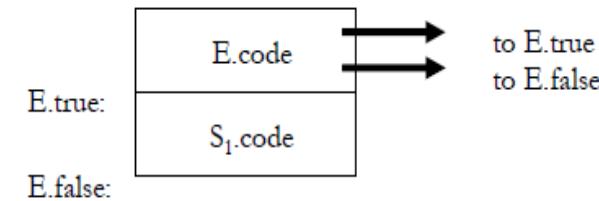


- Difficulty: Need to know where to jump to
  - Introduce a Symbolic labels using the newlabel function
  - Use Inherited Attributes
  - Back-patch it with the actual value (**later...**)

# Grammar and Actions

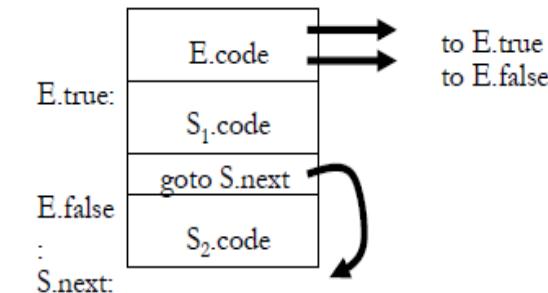
$S \rightarrow \text{if } E \text{ then } S_1 \parallel$

```
E.true = newlabel()
E.false = S.next
S1.next = S.next
S.code = append(E.code, gen(E.true:), S1.code)
```



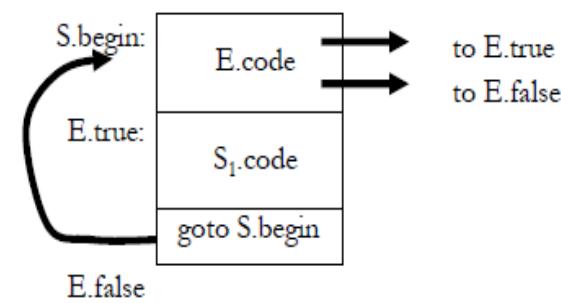
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \parallel$

```
E.true = newlabel()
E.false = newlabel()
S1.next = S.next
S2.next = S.next
S.code = append(E.code, gen(E.true:), S1.code,
                gen(goto S.next), gen(E.false :), S2.code)
```



$S \rightarrow \text{while } E \text{ do } S_1 \parallel$

```
S.begin = newlabel()
E.true = newlabel()
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:), E.code, gen(E.true:), S1.code,
                gen(goto S.begin))
```



# Control Flow Translation of Boolean Expressions

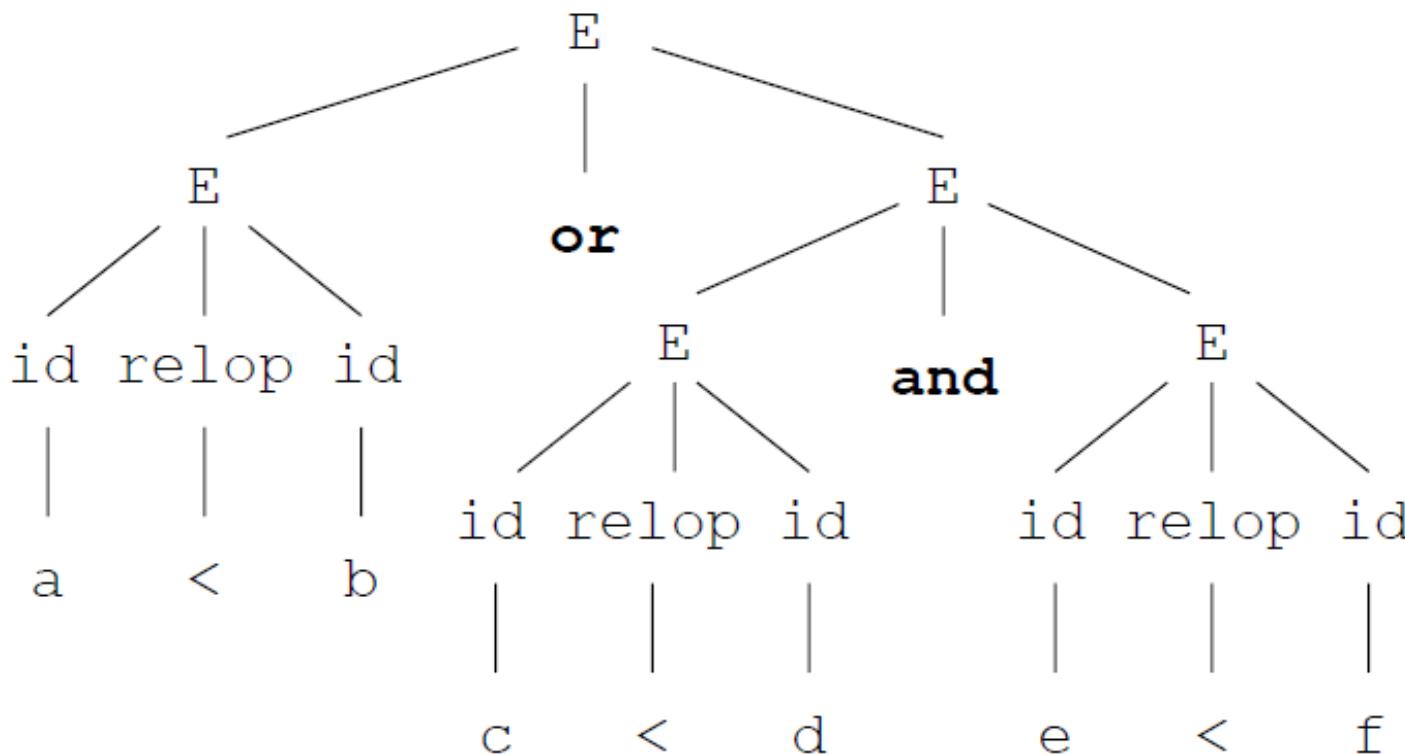
- Short-Circuit Evaluation
  - No need to Evaluate portions of the Expression if the outcome is already determined
  - Examples:
    - $E_1 \text{ or } E_2$  : need not evaluate  $E_2$  if  $E_1$  is known to be **true**.
    - $E_1 \text{ and } E_2$  : need not evaluate  $E_2$  if  $E_1$  is known to be **false**.
- Use Control Flow
  - *Jump* over code that evaluates boolean terms of the expression
  - Use Inherited  $E.\text{false}$  and  $E.\text{true}$  attributes and link evaluation of  $E$

# Control Flow for Boolean Expressions

$E \rightarrow id_1 \text{ relop } id_2 \parallel$	E.code = append(gen(if id <sub>1</sub> .place relop id <sub>2</sub> .place goto E.true), gen(goto E.false))		
$E \rightarrow \text{true} \parallel$	E.code = gen(goto E.true)	$E \rightarrow \text{not } E_1 \parallel$	E <sub>1</sub> .true = E.false E <sub>1</sub> .false = E.true E.code = E <sub>1</sub> .code
$E \rightarrow \text{false} \parallel$	E.code = gen(goto E.false)		
$E \rightarrow E_1 \text{ or } E_2 \parallel$		$E \rightarrow ( E_1 ) \parallel$	
	E <sub>1</sub> .true = E.true E <sub>1</sub> .false = newlabel E <sub>2</sub> .true = E.true E <sub>2</sub> .false = E.false E.code = append(E <sub>1</sub> .code, gen(E <sub>1</sub> .false:), E <sub>2</sub> .code)		E <sub>1</sub> .true = E.true E <sub>1</sub> .false = E.false E.code = E <sub>1</sub> .code
$E \rightarrow E_1 \text{ and } E_2 \parallel$			
	E <sub>1</sub> .false = E.false E <sub>1</sub> .true = newlabel E <sub>2</sub> .true = E.true E <sub>2</sub> .false = E.false E.code = append(E <sub>1</sub> .code, gen(E <sub>1</sub> .true:), E <sub>2</sub> .code)		

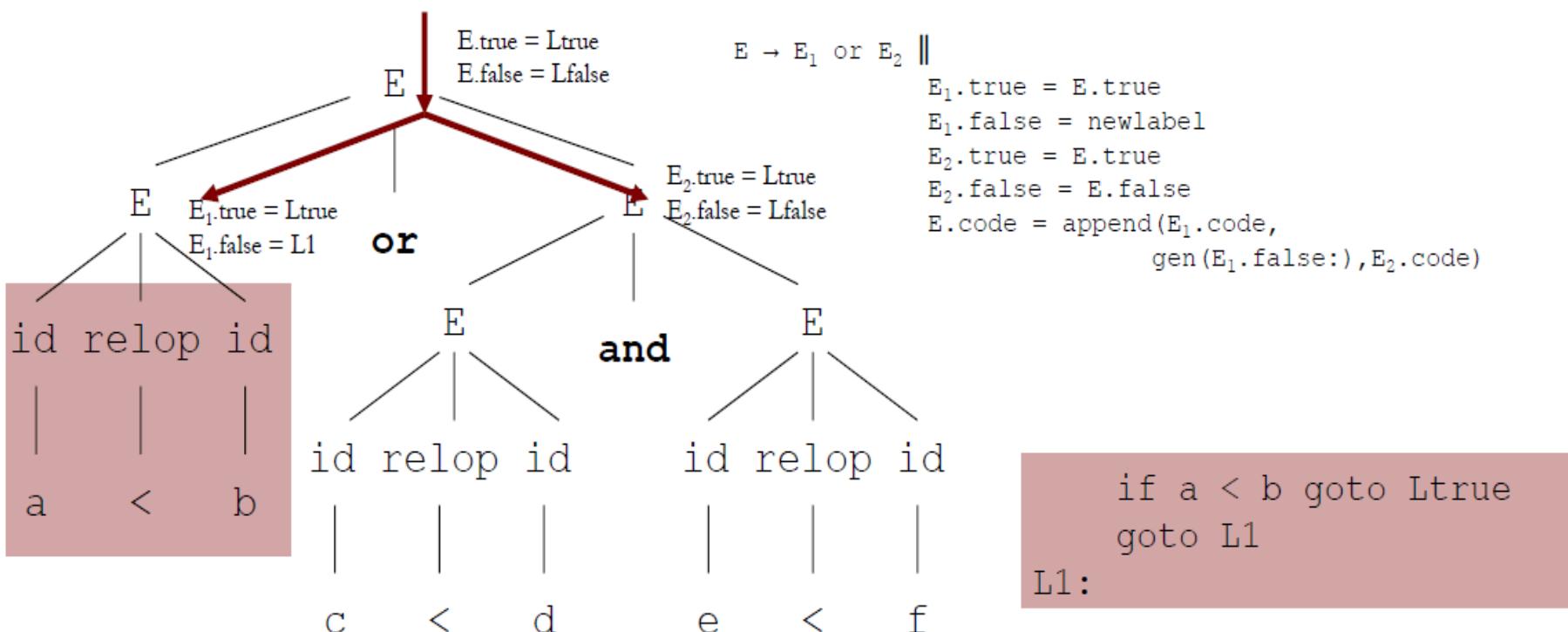
# Short Circuit Evaluation

a < b **or** c < d **and** e < f



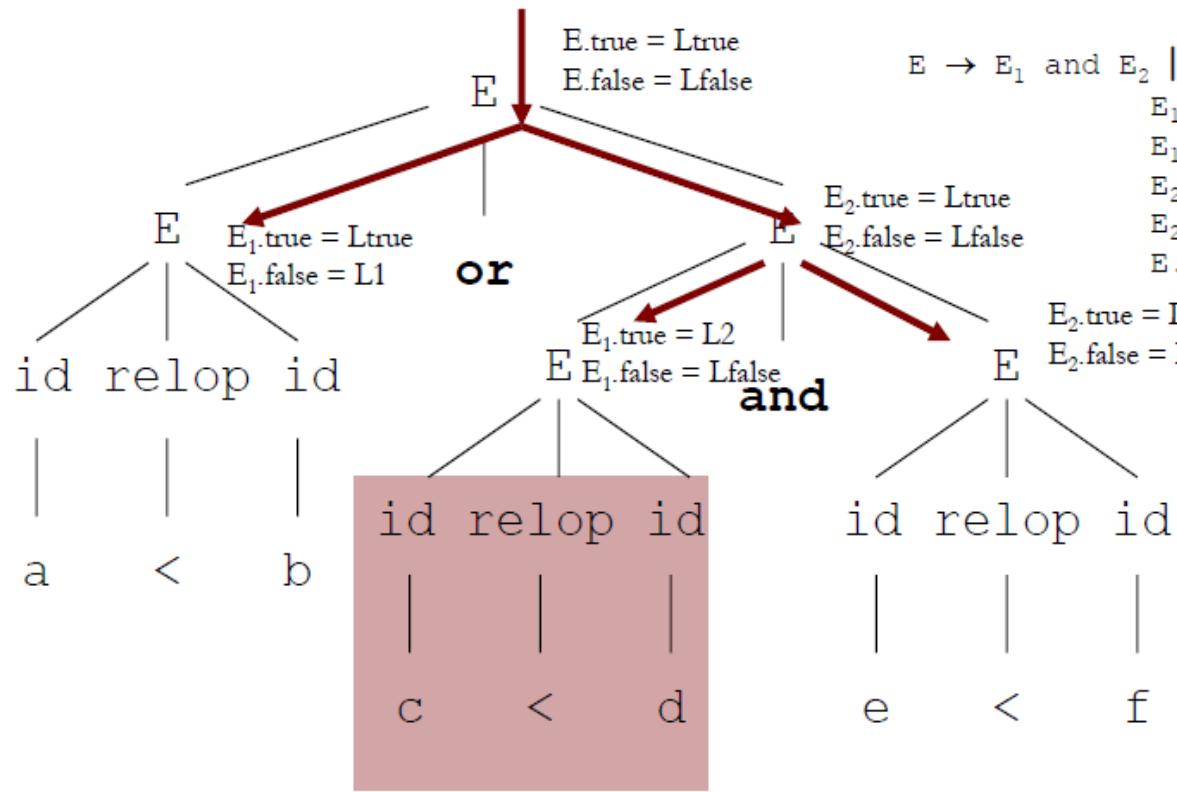
# Short Circuit Evaluation

$a < b \text{ or } c < d \text{ and } e < f$



# Short Circuit Evaluation

$a < b \text{ or } c < d \text{ and } e < f$



$E \rightarrow \text{id}_1 \text{ relop id}_2 \parallel$   
 $E.\text{code} = \text{append}($   
     $\text{gen}(\text{if } \text{id}_1.\text{place} \text{ relop } \text{id}_2.\text{place} \text{ goto } E.\text{true}),$   
     $\text{gen}(\text{goto } E.\text{false}))$

$E \rightarrow E_1 \text{ and } E_2 \parallel$   
 $E_1.\text{false} = E.\text{false}$   
 $E_1.\text{true} = \text{newlabel}$   
 $E_2.\text{true} = E.\text{true}$   
 $E_2.\text{false} = E.\text{false}$   
 $E.\text{code} = \text{append}(E_1.\text{code},$   
     $\text{gen}(E_1.\text{true}:), E_2.\text{code}))$

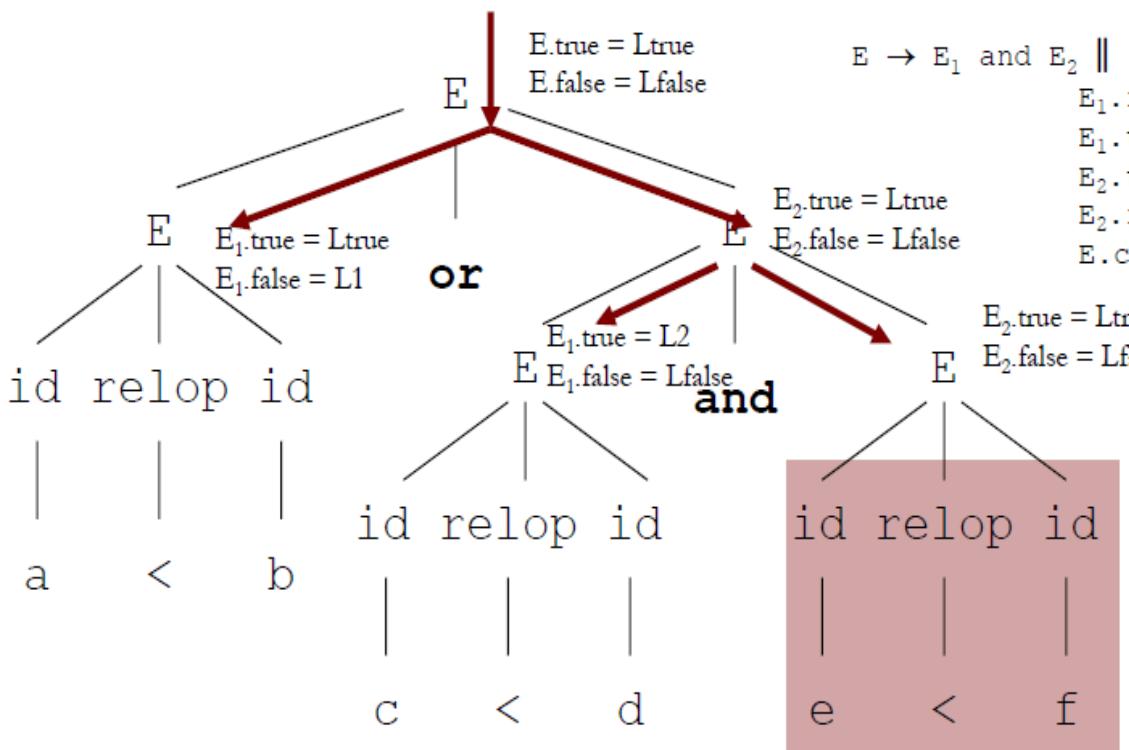
if  $a < b$  goto Ltrue  
goto L1

L1: if  $c < d$  goto L2  
goto Lfalse

L2:

# Short Circuit Evaluation

a < b **or** c < d **and** e < f



```

E → id1 relop id2 ||
      E.code = append(
          gen(if id1.place relop id2.place goto E.true),
          gen(goto E.false))

```

```

E → E1 and E2 ||

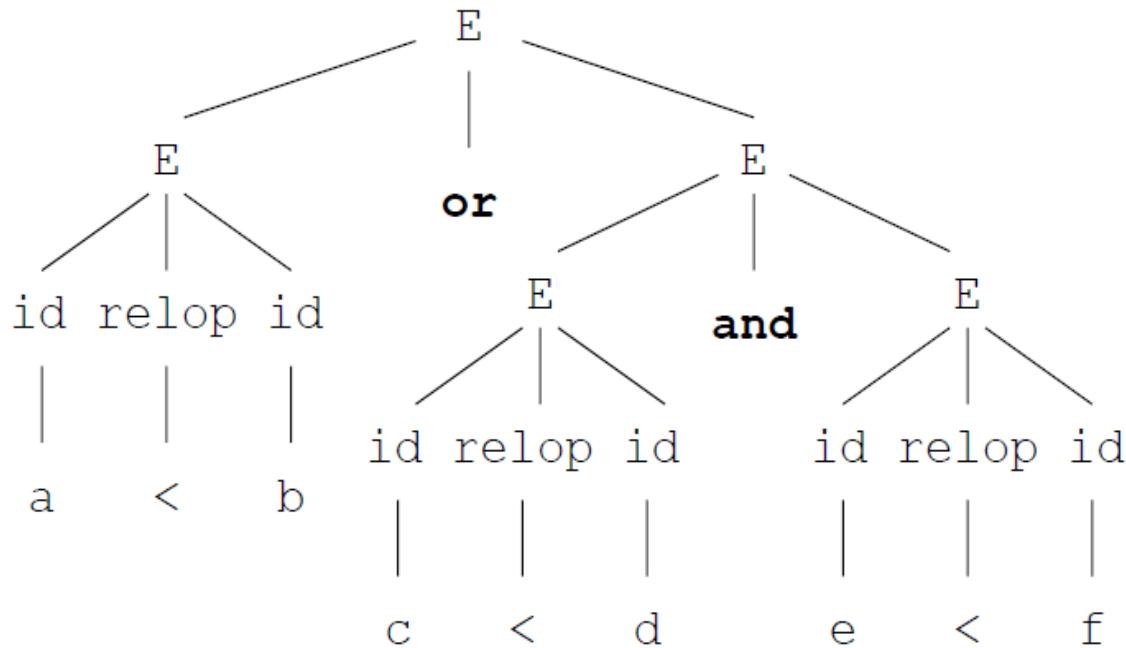
    E1.false = E.false
    E1.true = newlabel
    E2.true = E.true
    E2.false = E.false
    E.code = append(E1.code,
                                gen(E1.true:), E2.code)

```

```
if a < b goto Ltrue
goto L1
L1: if c < d goto L2
      goto Lfalse
L2: if e < f goto Ltrue
      goto Lfalse
```

# Short Circuit Evaluation

a < b **or** c < d **and** e < f



if a < b goto Ltrue  
goto L1

L1: if c < d goto L2  
goto Lfalse

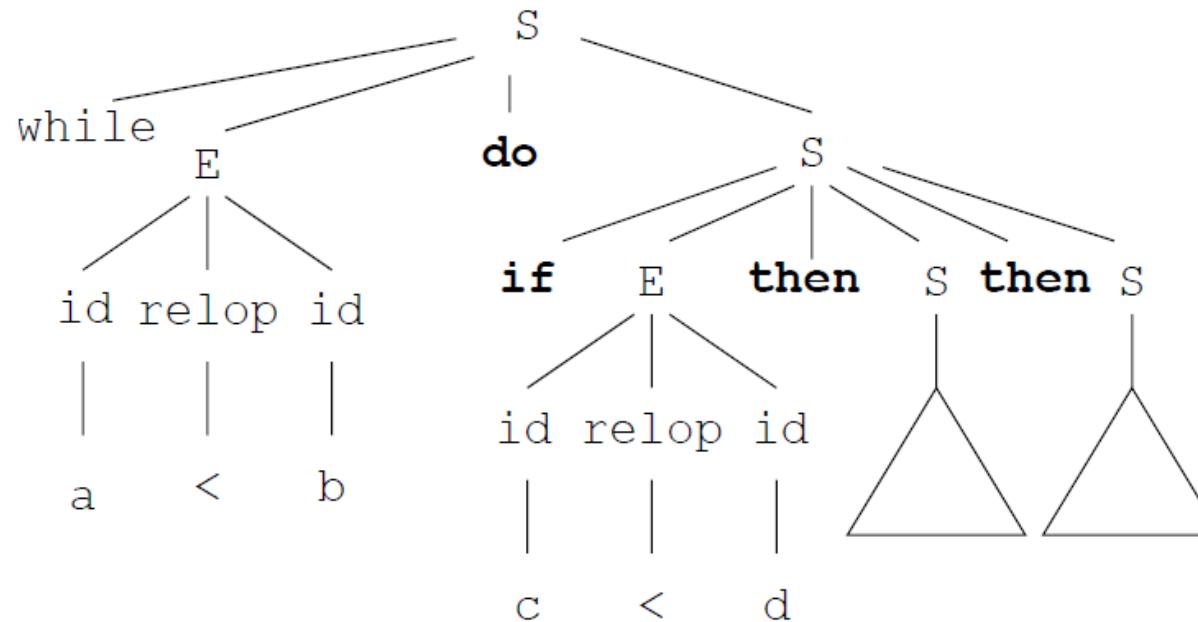
L2: if e < f goto Ltrue  
goto Lfalse

# Combining Boolean and Control Flow Statements

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y - z
```

$S \rightarrow \text{while } E \text{ do } S_1 \quad ||$

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:), E.code,
                gen(E.true:), S1.code,
                gen(goto S.begin))
```

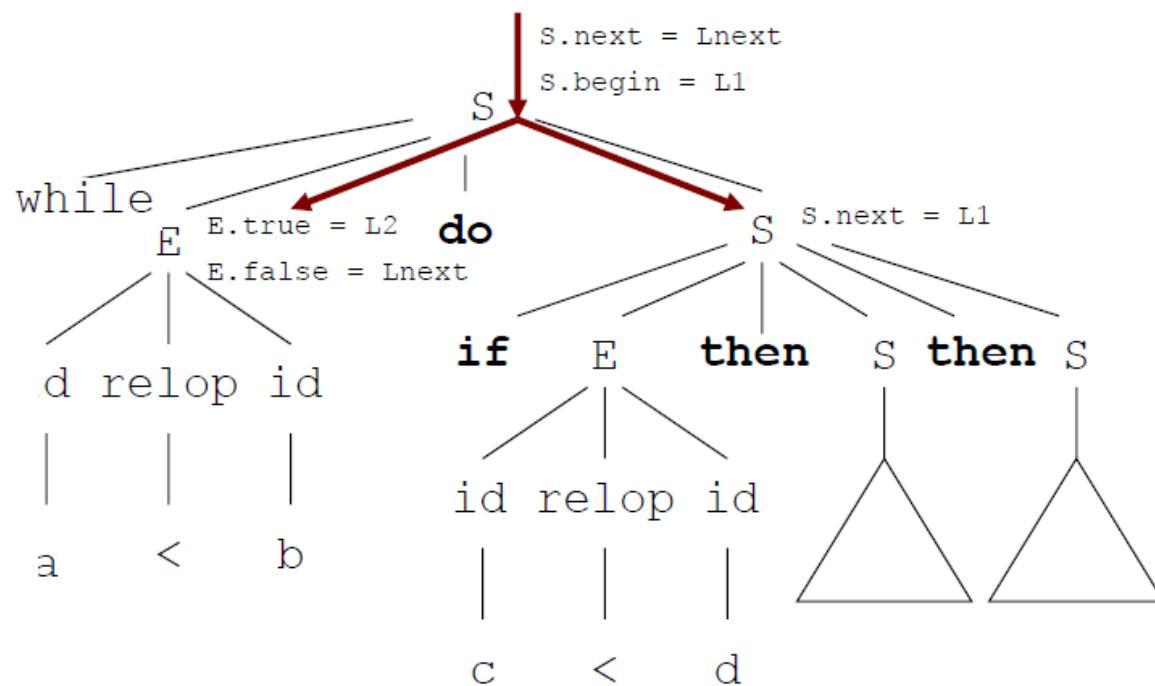


# Combining Boolean and Control Flow Statements

```
while a < b do
    if c < d then
        x = y + z
    else
        x = y - z
```

$S \rightarrow \text{while } E \text{ do } S_1 \quad ||$

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:), E.code,
                gen(E.true:), S1.code,
                gen(goto S.begin))
```



# Combining Boolean and Control Flow Statements

```

while a < b do
    if c < d then
        x = y + z
    else
        x = y - z

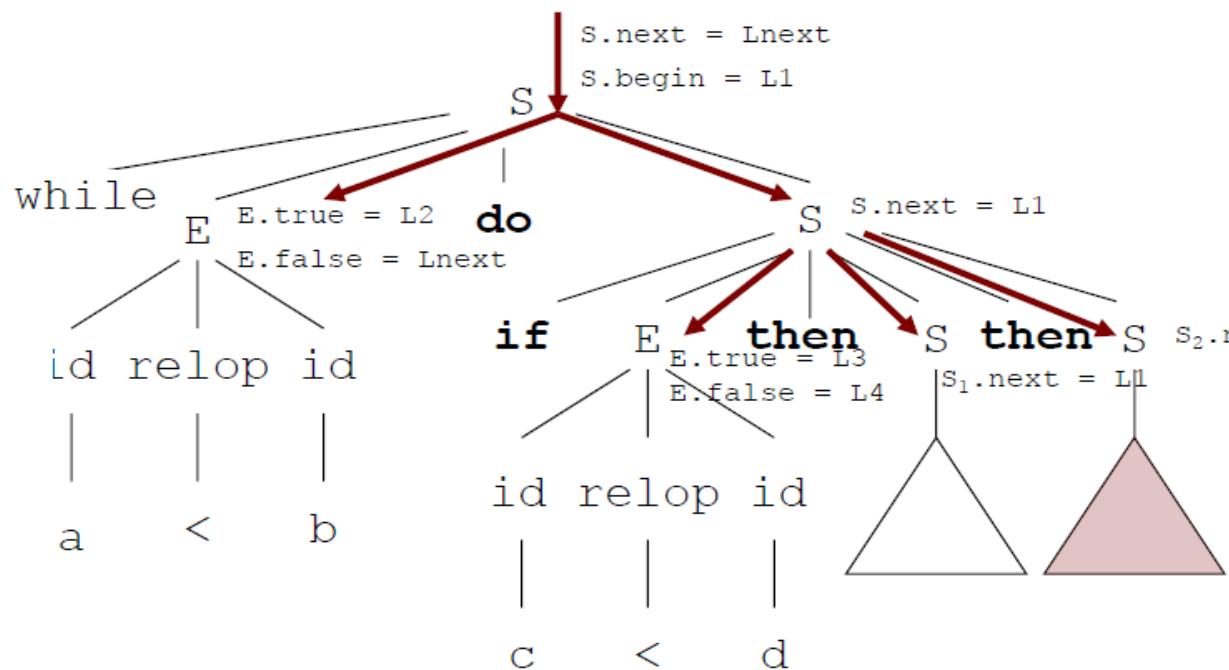
```

$S \rightarrow \text{while } E \text{ do } S_1 \parallel$

```

S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:), E.code,
                gen(E.true:), S1.code,
                gen(goto S.begin))

```



L1: if a < b goto L2

goto Lnext

L2: if c < d goto L3

goto L4

L3: t1 = x + z

x = t1

goto L1

L4: t2 = x - z

x = t2

goto L1

Lnext:

# Case Statement: Code Layout

```
switch E                                code to Evaluate E into t  
begin                                     goto Ltest  
    case V1: S1                      L1: code for S1  
    case V2: S2                      goto Lnext  
    ...                                     L2: code for S2  
    case Vn-1: Sn-1                 goto Lnext  
    default: Sn                         ...  
                                         Ln-1: code for Sn-1  
                                         goto Lnext  
end                                         Ln: code for Sn  
                                         goto Lnext  
  
Ltest: if t = V1 goto L1           }  
      if t = V2 goto L2  
      ...  
      if t = Vn-1 goto Ln-1  
      goto Ln  
  
Lnext:
```

**Linear Search**

# SDT scheme for Case Statements

Issue: Need to Save the Labels and Values for the various cases for the test code at the end

- Use a Right-Recursive Grammar (next slide)
  - Use queue to save pairs (value, label) for generation of search code
  - In the end pop values from queue to generate the linear search
- Use a Left-Recursive Grammar
  - Cleaner: No queue is needed
  - Use of the parsing stack to accumulate the non-terminals and corresponding attribute

# Grammar and Actions

```
S → switch E List end ||
    S.code = append(E.code, gen('goto Ltest'), List.code, gen('Ltest:'))
    while(queue not empty) do {
        (vi,Li) = pop.queue;
        if (vi = default)
            S.code = append(S.code, gen('goto Li'));
        else
            S.code = append(S.code, gen('if t = vi goto Li'));

Case → case Value : S ||
    Case.code = append(gen('Li:'), S.code, gen('goto Lnext'));
    queue.push((Value.val,Li));

List → Case ; List1 ||
    List.code = append(Case.code, List1.code);

List → default : S ||
    List.code = append(gen('Li:'), S.code, gen('goto Lnext'));
    queue.push((default,Li))

List → ε ||
    List.code = append(gen('Li:'), gen('goto Lnext'));
    queue.push((default,Li))
```

# Boolean Expressions Revisited

- Use Additional  $\varepsilon$ -Production

- Just a Marker M
  - Label Value M.addr

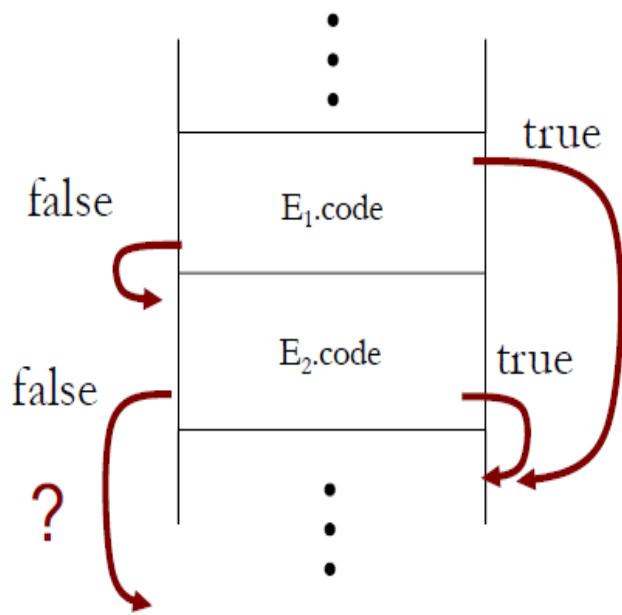
$$\begin{array}{ll}(1) & E \rightarrow E_1 \text{ or } M \ E_2 \\(2) & \quad | \quad E_1 \text{ and } M \ E_2 \\(3) & \quad | \quad \text{not } E_1 \\(4) & \quad | \quad (E_1) \\(5) & \quad | \quad id_1 \text{ relop } id_2 \\(6) & \quad | \quad \text{true} \\(7) & \quad | \quad \text{false} \\(8) & M \rightarrow \varepsilon\end{array}$$

- Attributes:

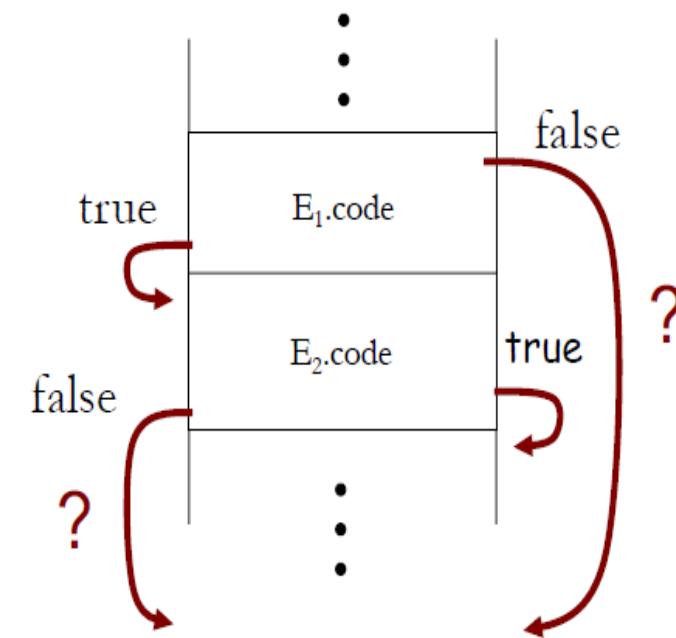
- E.trueList: code places that need to be filled-in corresponding to the evaluation of E as “true”.
  - E.falseList: same for “false”

# Boolean Expressions: Code Outline

$E_1 \text{ or } E_2$



$E_1 \text{ and } E_2$



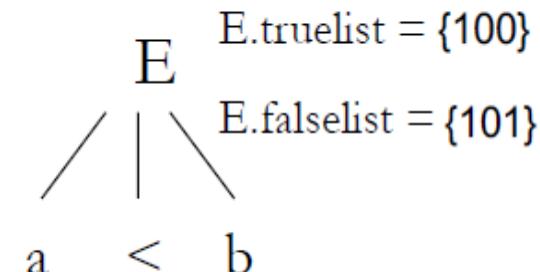
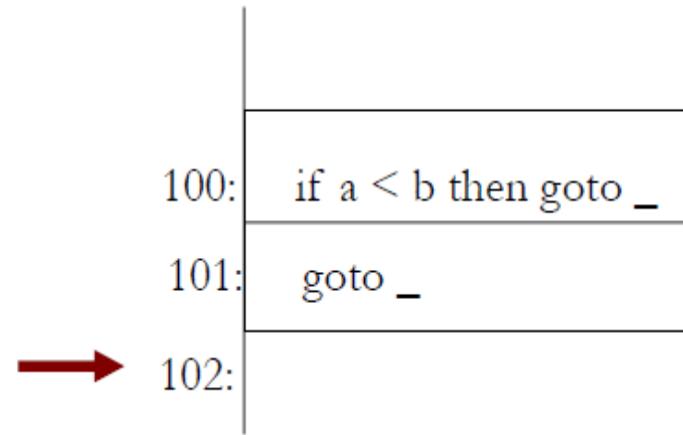
# Auxiliary Functions

- Functions:
  - makeList(i): make a list with the label i
  - merge(p1,p2): creates a new list of labels with lists p1 and p2
  - backPatch(p,i): fills the locations in p with the address i
  - newAddress() : returns a new symbolic address in sequence and increments the value for the next call
- Array of Instructions
  - Linearly sequence of instructions
  - Function emit to generate actual instructions in the array
  - Symbolic Addresses

# Using the Actions & List Attributes

5)  $E \rightarrow id_1 \text{ relop } id_2$

```
|| E.truelist := makeList(newlabel);  
    E.falselist := makeList(newlabel);  
    emit("if id1.place relop id2.place goto _");  
    emit("goto _");
```



(3)  $E \rightarrow \text{not } E_1$       ||  $E.\text{truelist} := E_1.\text{falselist}; E.\text{falselist} := E_1.\text{truelist};$

(4)  $E \rightarrow (E_1)$       ||  $E.\text{truelist} := E_1.\text{truelist}; E.\text{falselist} := E_1.\text{falselist};$

(5)  $E \rightarrow id_1 \text{ relop } id_2$       ||  $E.\text{truelist} := \text{makeList(nextAddr())};$   
                         $E.\text{falselist} := \text{makeList(nextAddr())};$   
                         $\text{emit("if } id_1.\text{place relop.op } id_2.\text{place goto } \underline{\quad});$   
                         $\text{emit("goto } \underline{\quad});$

(6)  $E \rightarrow \text{true}$       ||  $E.\text{truelist} := \text{makeList(nextAddr())}; \text{emit("goto } \underline{\quad});$

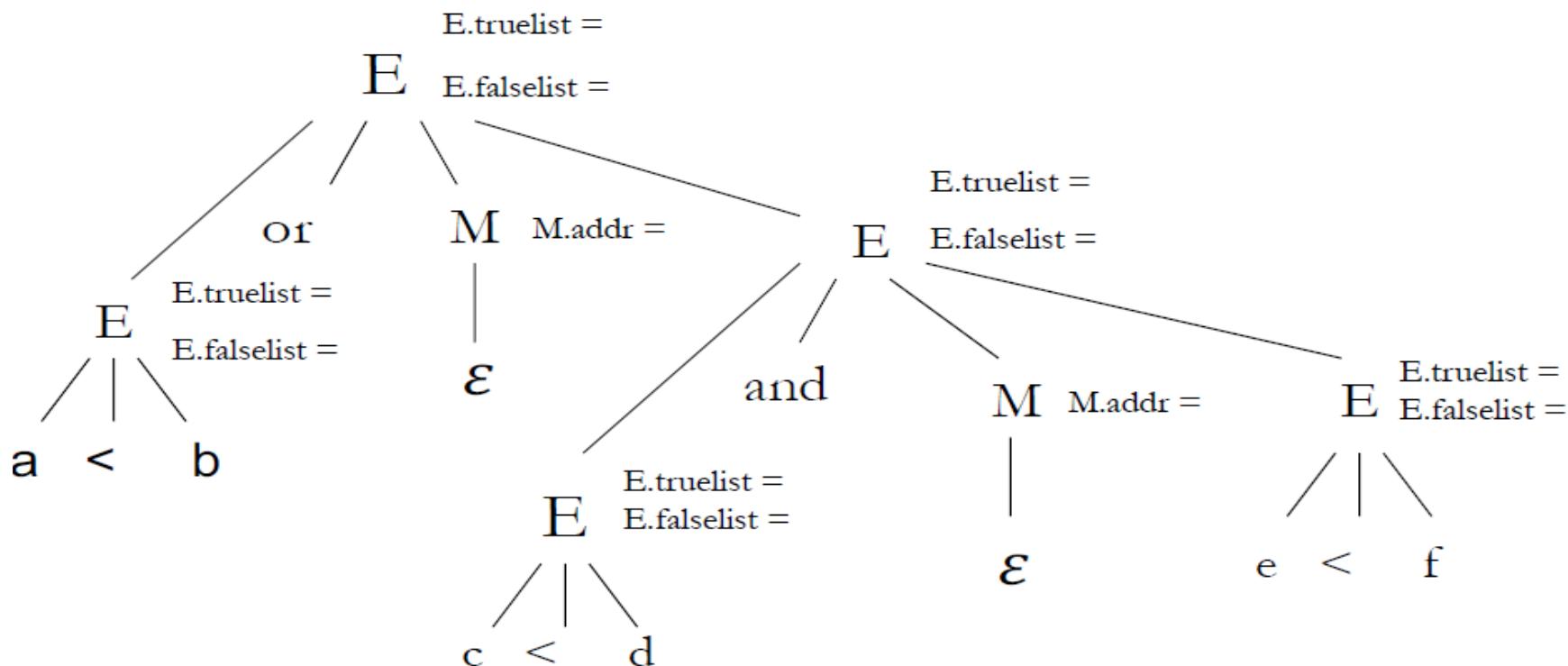
(7)  $E \rightarrow \text{false}$       ||  $E.\text{falselist} := \text{makeList(nextAddr())}; \text{emit("goto } \underline{\quad});$

# More Actions

- (1)  $E \rightarrow E_1 \text{ or } M\ E_2$       || backpatch( $E_1.\text{falselist}, M.\text{Addr}$ );  
     $E.\text{truelist} := \text{merge}(E_1.\text{truelist}, E_2.\text{truelist})$ ;  
     $E.\text{falselist} := E_2.\text{falselist}$ ;
- (2)  $E \rightarrow E_1 \text{ and } M\ E_2$       || backpatch( $E_1.\text{truelist}, M.\text{Addr}$ );  
     $E.\text{truelist} := E_2.\text{truelist}$ ;  
     $E.\text{falselist} := \text{merge}(E_1.\text{falselist}, E_2.\text{falselist})$ ;
- (8)  $M \rightarrow \varepsilon$       ||  $M.\text{Addr} := \text{nextAddr}$ ;

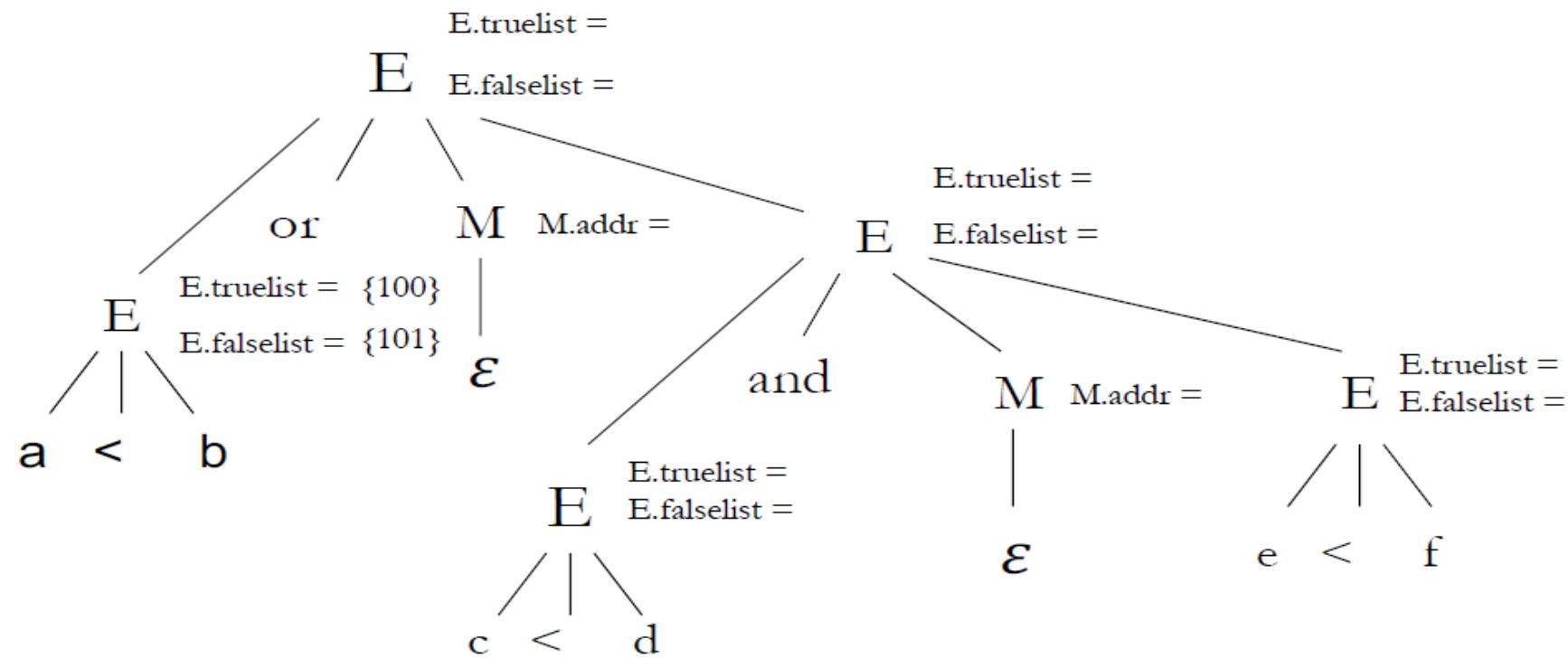
# Back-Patching Example

	Executing Action	Generated Code
E	E.truelist E.falselist	
M	M.addr	



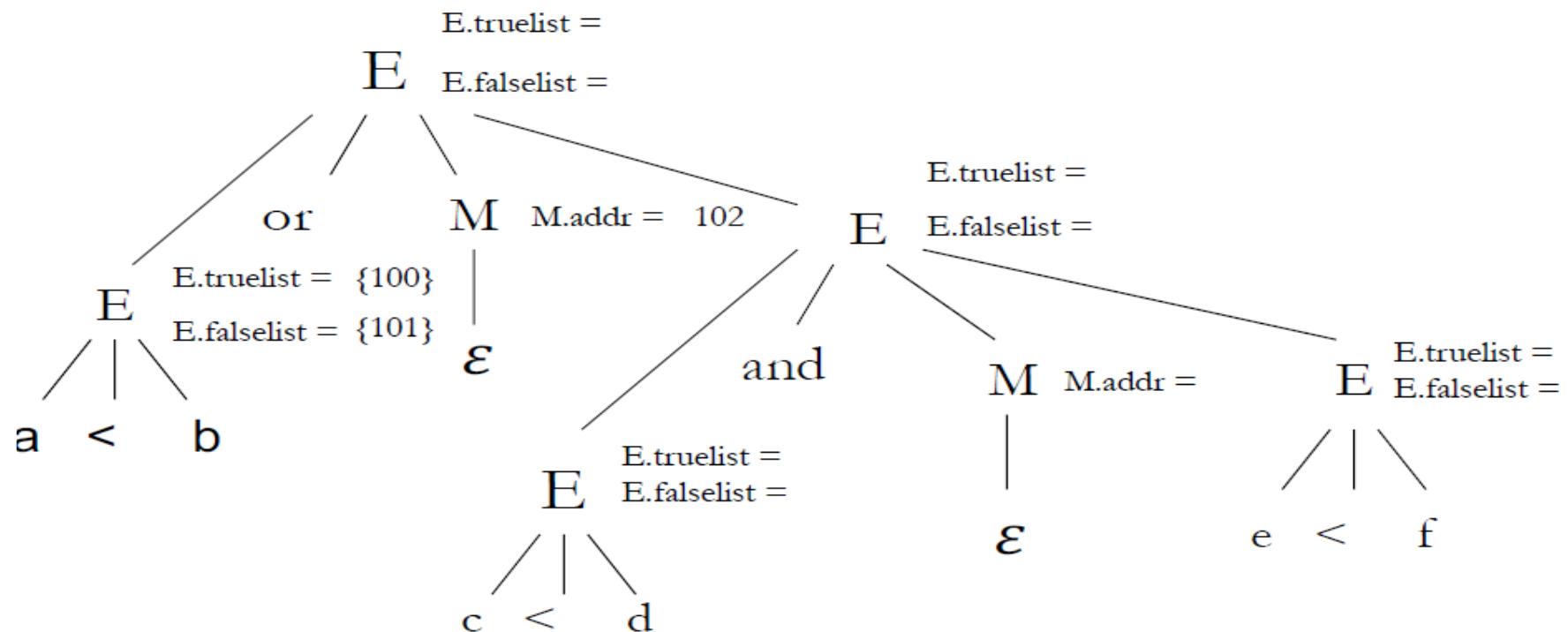
# Back-Patching Example

	Executing Action	Generated Code
E E.truelist E E.falselist	<pre>E.truelist := makelist(nextquad()); E.falselist := makelist(nextquad());</pre>	<pre>100: if a &lt; b goto _ 101: goto _</pre>
M M.addr	<pre>emit("if id1.place relop.op id2.place goto _"); emit("goto _");</pre>	



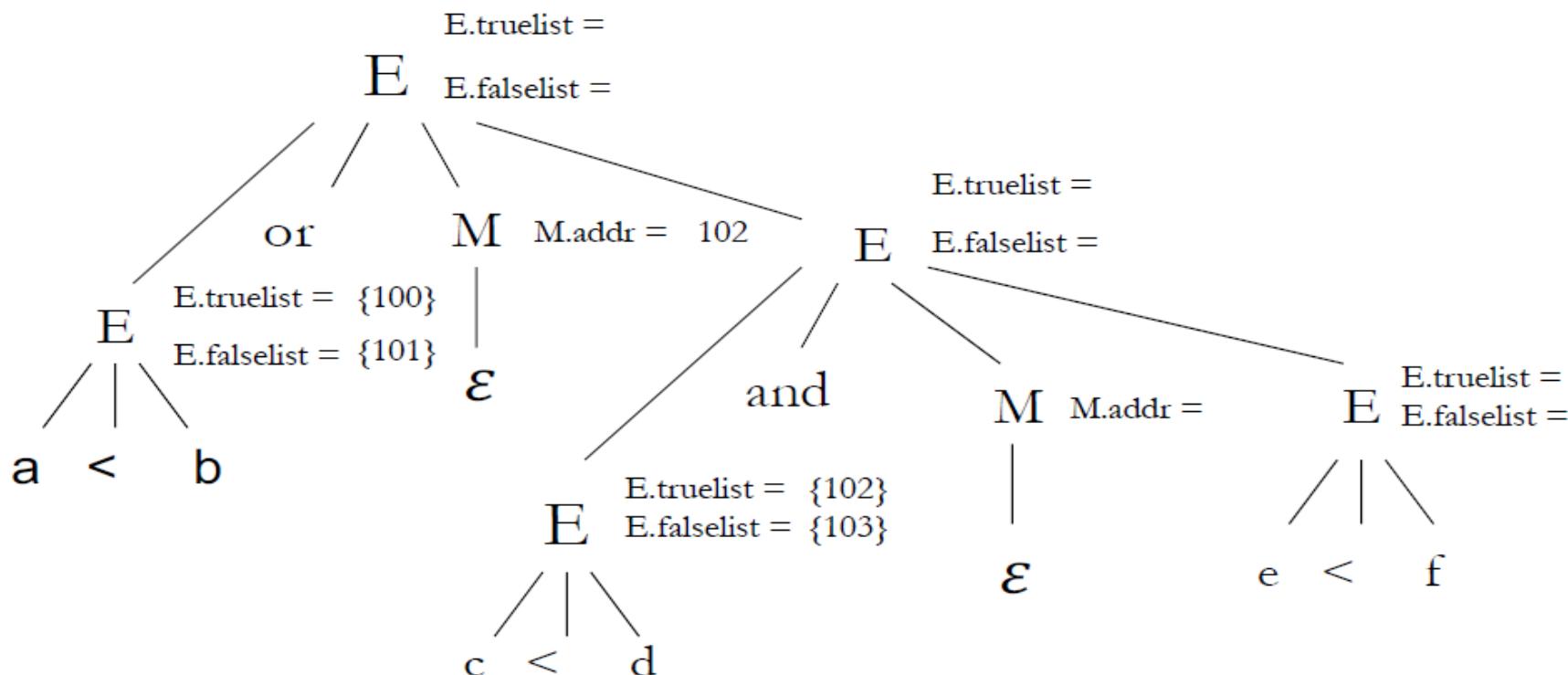
# Back-Patching Example

	Executing Action	Generated Code
E    E.truelist E    E.falsestlist	M.quad = nextquad();	100: if a < b goto _
M    M.addr		101: goto _



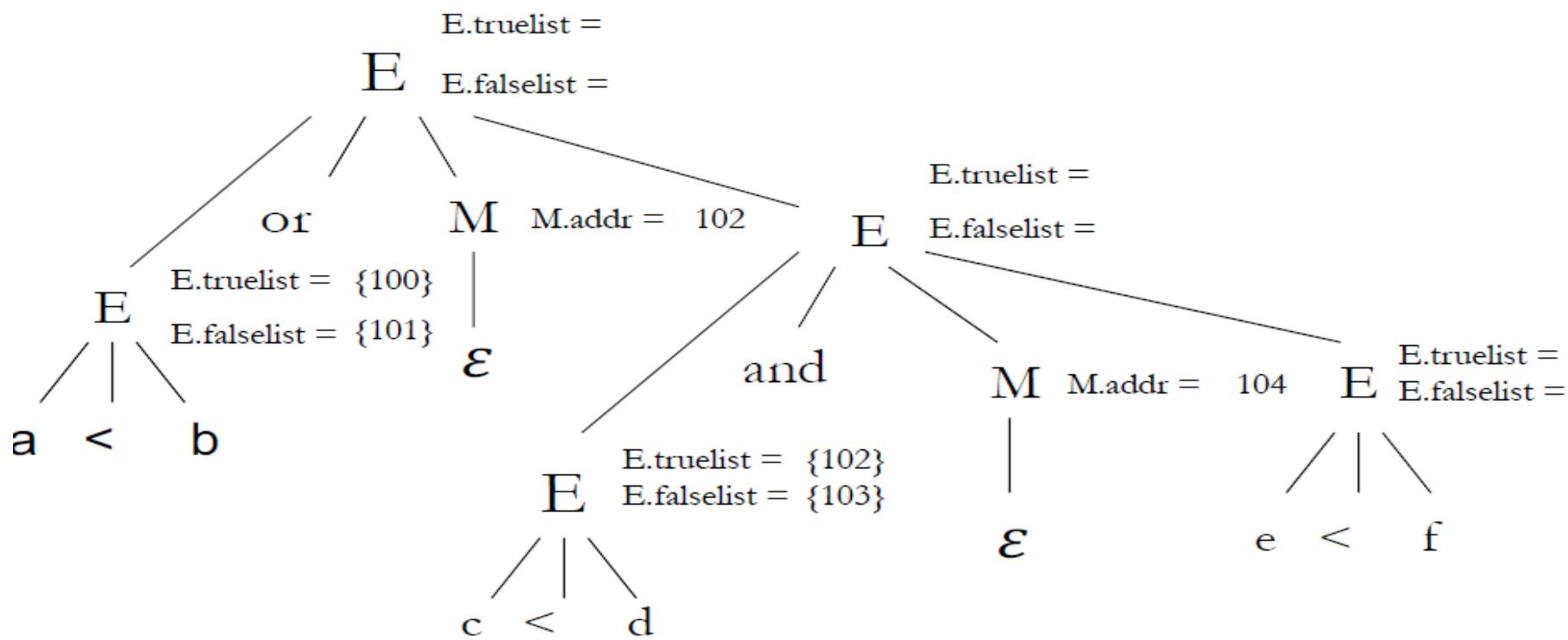
# Back-Patching Example

	Executing Action	Generated Code
E    E.truelist E    E.falsestlist	E.truelist := makelist(nextquad()); E.falsestlist := makelist(nextquad());	100: if a < b goto _
M    M.addr	emit("if id1.place relop.op id2.place goto _"); emit("goto _");	101: goto _ 102: if c < d goto _ 103: goto _



# Back-Patching Example

	Executing Action	Generated Code
E    E.truelist E    E.falselist	M.quad = nextquad();	100: if a < b goto _ 101: goto _
M    M.addr		102: if c < d goto _ 103: goto _



# Back-Patching Example

	Executing Action	Generated Code
E E.truelist E E.falselist	E.truelist := makelist(nextquad()); E.falselist := makelist(nextquad());	100: if a < b goto _
M M.addr	emit("if id1.place relop.op id2.place goto _"); emit("goto _");	101: goto _ 102: if c < d goto _ 103: goto _ 104: if e < f goto _
	E.truelist = E.falselist =	105: goto _
	or      M M.addr = 102	
	E E.truelist = {100} E.falselist = {101}	
a < b		
	E E.truelist = {102} E.falselist = {103}	
c < d		
	and      M M.addr = 104	
	E E.truelist = {104} E.falselist = {105}	
e < f		

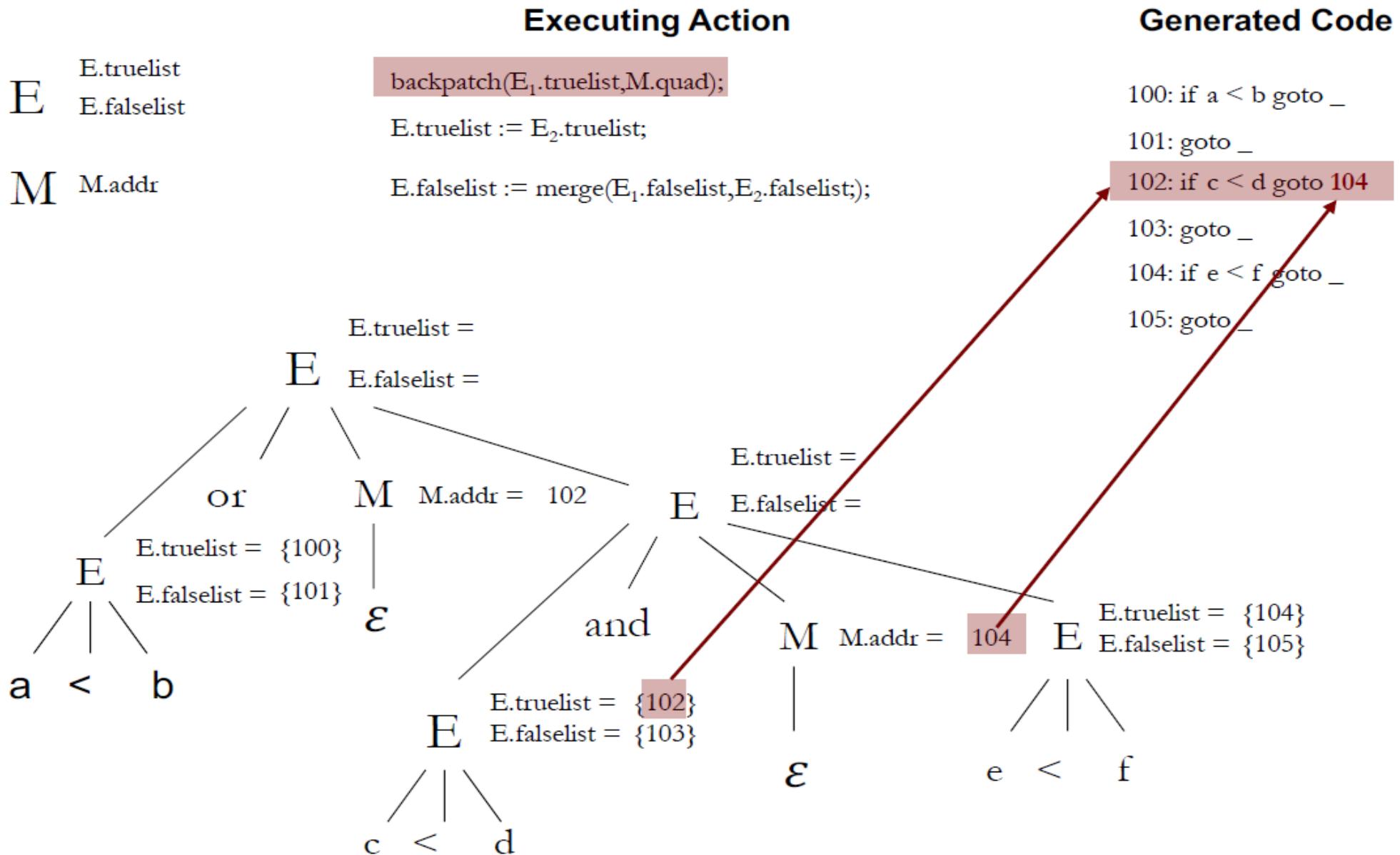
# Back-Patching Example

	Executing Action	Generated Code
E	<p>E.truelist E.falselist</p> <p>backpatch(E<sub>1</sub>.truelist,M.quad); E.truelist := E<sub>2</sub>.truelist;</p>	<p>100: if a &lt; b goto _ 101: goto _</p>
M	<p>M.addr</p> <p>E.falselist := merge(E<sub>1</sub>.falselist,E<sub>2</sub>.falselist);</p>	<p>102: if c &lt; d goto _ 103: goto _ 104: if e &lt; f goto _ 105: goto _</p>
	<p>E.truelist =</p>	

# Back-Patching Example

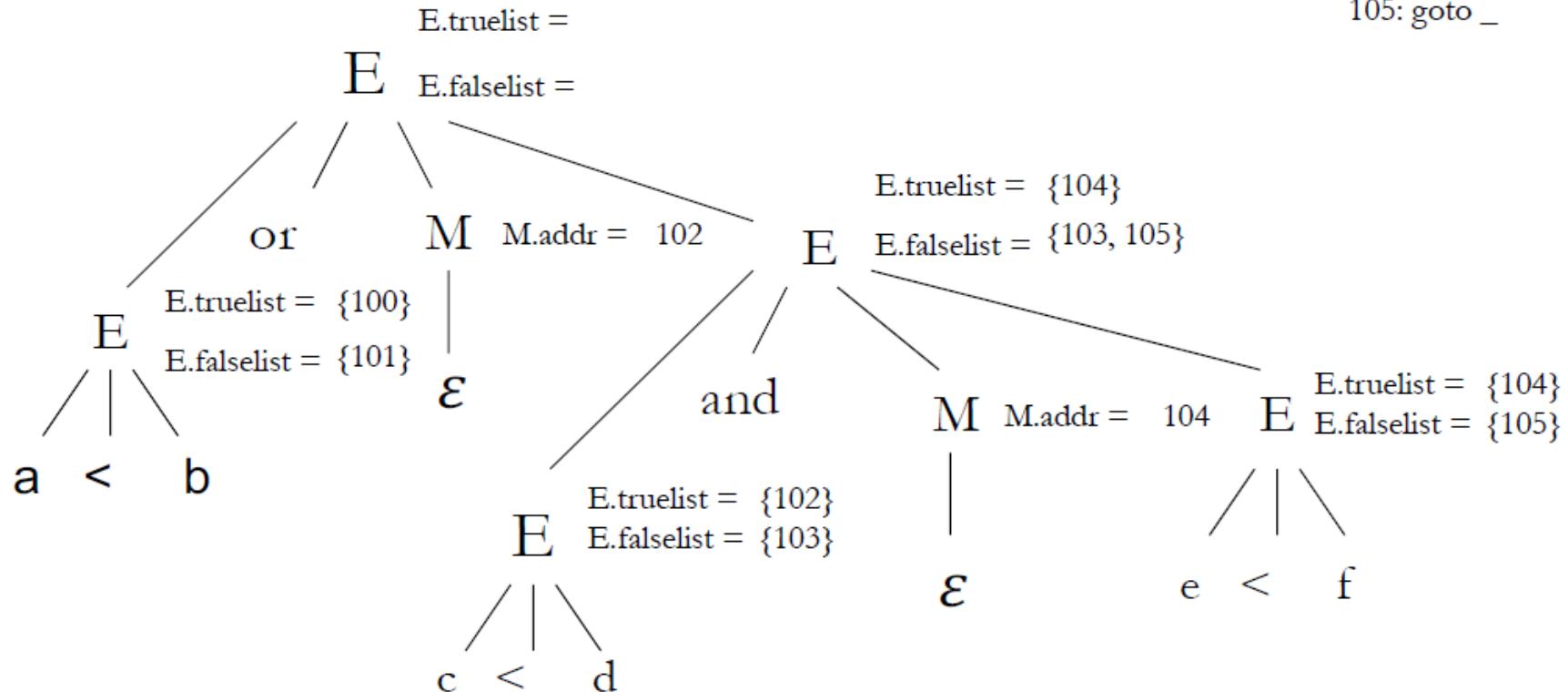
	Executing Action	Generated Code
E	<p>E.truelist E.falsestlist</p> <pre>backpatch(E<sub>1</sub>.truelist,M.quad); E.truelist := E<sub>2</sub>.truelist;</pre>	100: if a < b goto _
M	<p>M.addr</p> <pre>E.falsestlist := merge(E<sub>1</sub>.falsestlist,E<sub>2</sub>.falsestlist);</pre>	101: goto _ 102: if c < d goto _ 103: goto _ 104: if e < f goto _
	<p>E.truelist =</p> <pre>E.truelist = E.falsestlist = or E.truelist = {100} E.falsestlist = {101} M M.addr = 102 E truelist = {102} E.falsestlist = {103} and E truelist = {104} E.falsestlist = {105} E E e &lt; f</pre>	105: goto _

# Back-Patching Example



# Back-Patching Example

	Executing Action	Generated Code
E E.truelist E.falselist	backpatch(E <sub>1</sub> .truelist,M.quad); E.truelist := E <sub>2</sub> .truelist;	100: if a < b goto _ 101: goto _
M M.addr	E.falselist := merge(E <sub>1</sub> .falselist,E <sub>2</sub> .falselist);	102: if c < d goto 104 103: goto _
	E.truelist =	104: if e < f goto _ 105: goto _



# Back-Patching Example

	Executing Action	Generated Code
E	<p>E.trueclist E.falseclist</p> <p>backpatch(E<sub>1</sub>.trueclist,M.quad); E.trueclist := E<sub>2</sub>.trueclist;</p>	<p>100: if a &lt; b goto _</p> <p>101: goto 102</p>
M	<p>M.addr</p> <p>E.falseclist := merge(E<sub>1</sub>.falseclist,E<sub>2</sub>.falseclist);</p>	<p>102: if c &lt; d goto 104</p> <p>103: goto _</p>
	<p>E.trueclist = {100, 104}</p>	<p>104: if e &lt; f goto _</p>
	<p>E</p> <p>E.trueclist = {100, 104} E.falseclist = {103, 105}</p>	<p>105: goto _</p>
	<p>or</p> <p>M M.addr = 102</p>	
	<p>E</p> <p>E.trueclist = {100} E.falseclist = {101}</p>	
	<p>a &lt; b</p>	
	<p>E</p> <p>E.trueclist = {102} E.falseclist = {103}</p>	
	<p>c &lt; d</p>	
	<p>and</p>	
	<p>E</p> <p>E.trueclist = {104} E.falseclist = {105}</p>	
	<p>M M.addr = 104</p>	
	<p>E</p> <p>E.trueclist = {104} E.falseclist = {105}</p>	
	<p>e &lt; f</p>	

# Back-Patching Example

$S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$

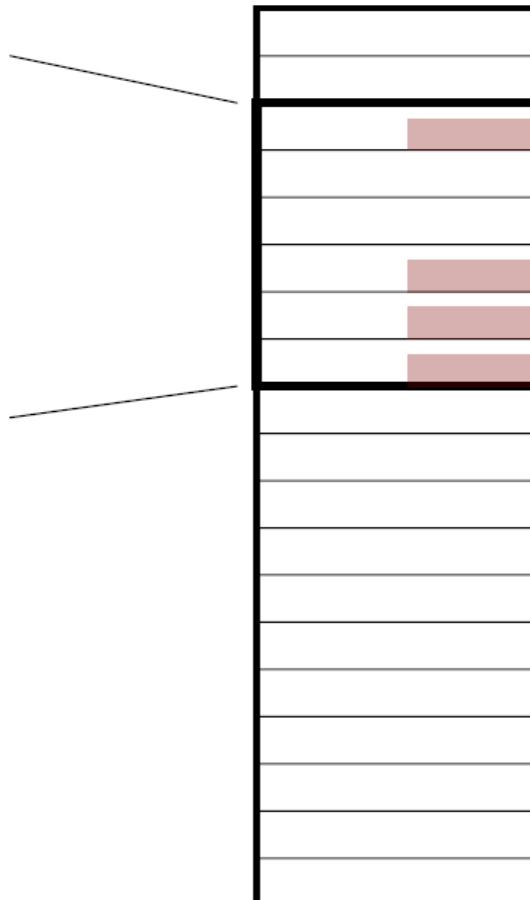
```
backpatch(S1.nextlist, M1.addr);  
backpatch(E.truelist,M2.addr);  
S.nextlist := E.falselist;  
emit("goto M1.addr");
```

E

```
100: if a < b goto _  
101: goto 102  
102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```

E.truelist = {100, 104}

E.falselist = {103, 105}



# Back-Patching Example

$S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$

```
backpatch(S1.nextlist, M1.addr);  
backpatch(E.truelist,M2.addr);  
S.nextlist := E.falselist;  
emit("goto M1.addr");
```

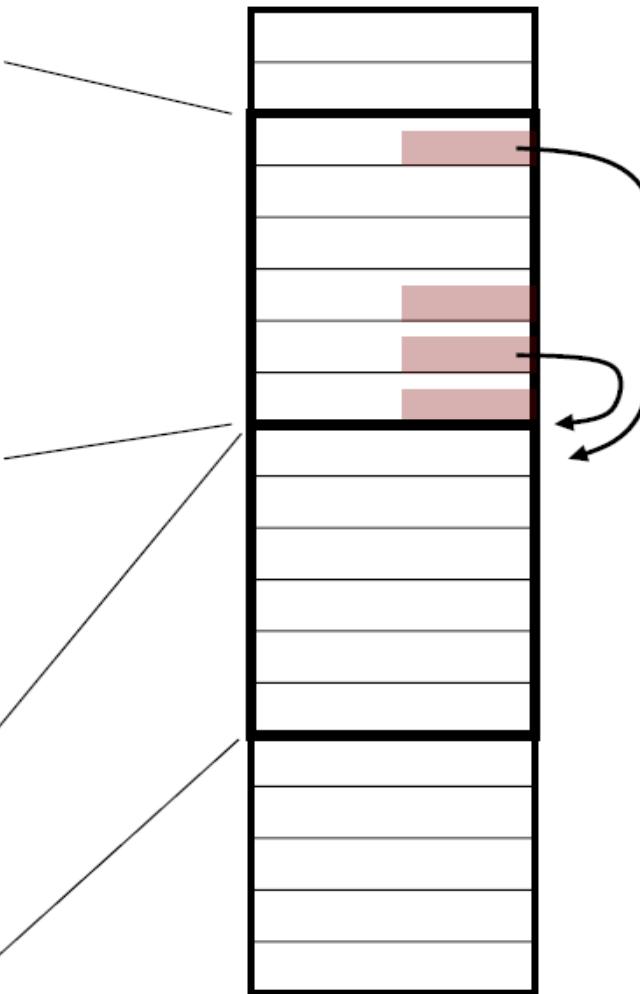
**E**

```
100: if a < b goto _  
101: goto 102  
102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```

$E.\text{truelist} = \{100, 104\}$

$E.\text{falselist} = \{103, 105\}$

**S<sub>1</sub>**



# Back-Patching Example

$S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$

```
backpatch( $S_1.\text{nextlist}$ ,  $M_1.\text{addr}$ );  
backpatch( $E.\text{truelist}$ ,  $M_2.\text{addr}$ );  
 $S.\text{nextlist} := E.\text{falseList};$   
emit("goto  $M_1.\text{addr}$ ");
```

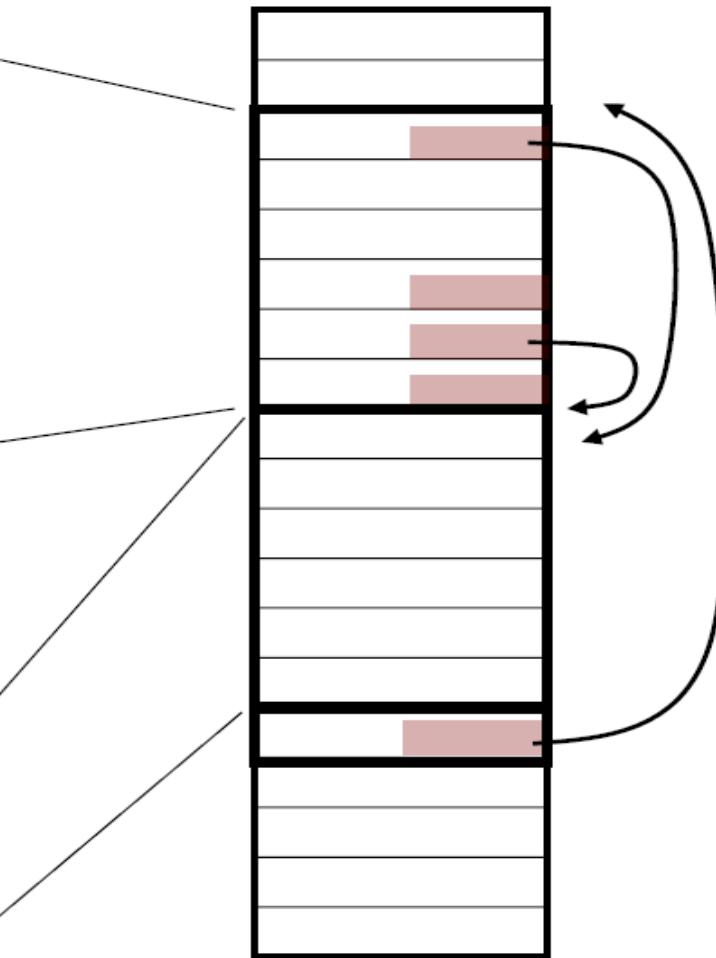
**E**

```
100: if a < b goto _  
101: goto 102  
102: if c < d goto 104  
103: goto _  
104: if e < f goto _  
105: goto _
```

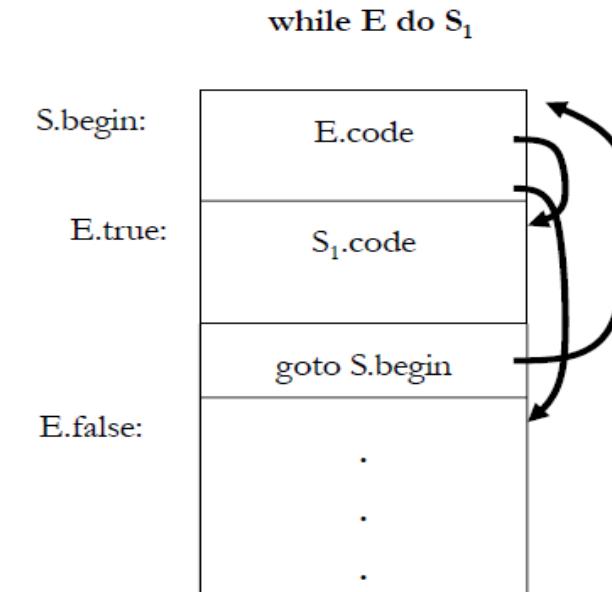
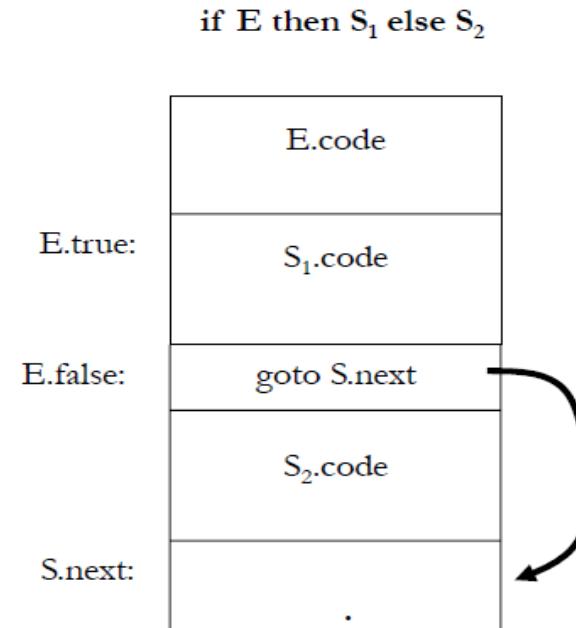
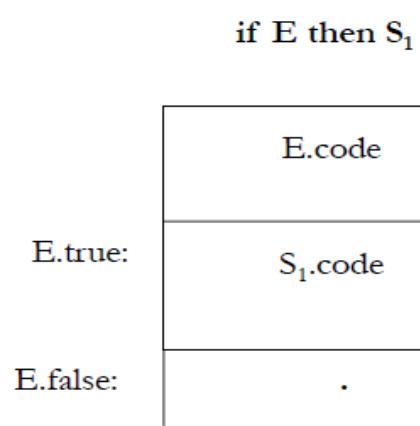
$E.\text{truelist} = \{100, 104\}$

$E.\text{falseList} = \{103, 105\}$

**$S_1$**



# Control Flow Code Structures



# Control Flow Constructs: Conditionals

- Add the `nextlist` attribute to `S` and `N`
    - denotes the set of locations in the `S` code to be patched with the address that follows the execution of `S`
    - Can be either due to control flow or fall-through
- (1)  $S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2 \quad \| \quad \text{backpatch}(E.\text{truelist}, M_1.\text{addr});$   
 $\qquad \qquad \qquad \text{backpatch}(E.\text{falselist}, M_2.\text{addr});$   
 $\qquad \qquad \qquad S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, \text{merge}(N.\text{nextlist}, S_2.\text{nextlist}));$
- (2)  $N \rightarrow \epsilon \quad \| \quad N.\text{nextlist} := \text{makelist}(\text{nextAddr});$   
 $\qquad \qquad \qquad \text{emit}(\text{"goto } \underline{\quad} \text{"}); \}$
- (3)  $M \rightarrow \epsilon \quad \| \quad M.\text{quad} := \text{nextAddr};$
- (4)  $S \rightarrow \text{if } E \text{ then } M S_1 \quad \| \quad \text{backpatch}(E.\text{truelist}, M.\text{addr});$   
 $\qquad \qquad \qquad S.\text{nextlist} := \text{merge}(E.\text{falselist}, S_1.\text{nextlist});$

# Control Flow Constructs: Loops

(5)  $S \rightarrow \text{while } M_1 \text{ E do } M_2 S_1 \quad \| \quad \text{backpatch}(S_1.\text{nextlist}, M_1.\text{addr}),;$   
 $\qquad \qquad \qquad \text{backpatch}(E.\text{truelist}, M_2.\text{addr});$   
 $\qquad \qquad \qquad S.\text{nextlist} := E.\text{falselist};$   
 $\qquad \qquad \qquad \text{emit}(\text{"goto } M_1.\text{addr}") );$

# Sequencing: List of Statements

- Additional Symbols
  - L for list of statements
  - S for Assignment statement

(6)  $S \rightarrow \text{begin } L \text{ end} \quad \| \quad S.\text{nextlist} = L.\text{nextlist};$

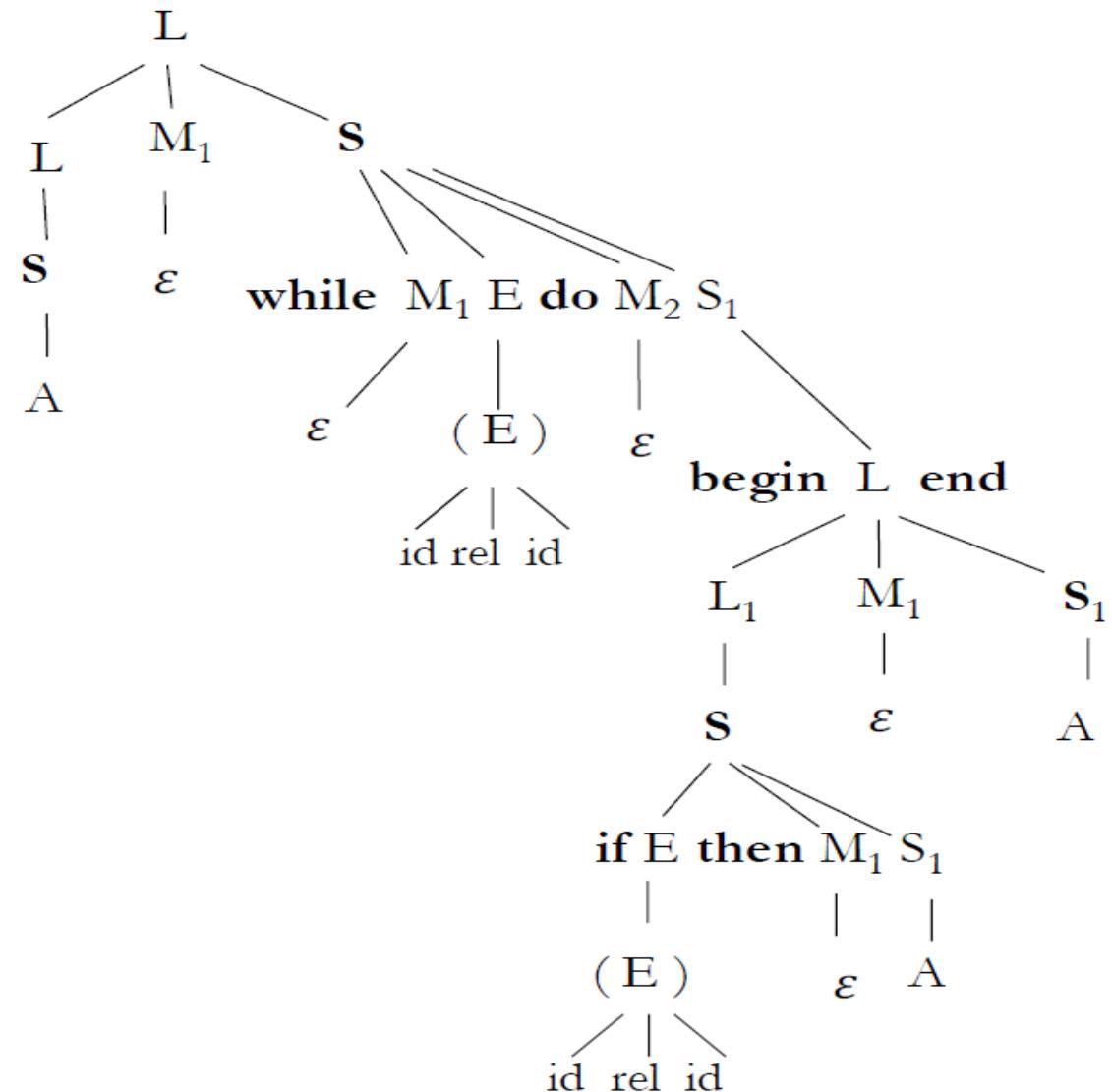
(7)  $S \rightarrow A \quad \| \quad S.\text{nextlist} = \text{nil};$

(8)  $L \rightarrow L_1; M S \quad \| \quad \text{backpatch}(L_1.\text{nextlist}, M.\text{addr}); \quad L.\text{nextlist} = S.\text{nextlist};$

(9)  $L \rightarrow S \quad \| \quad L.\text{nextlist} = S.\text{nextlist};$

# Extended Example

```
i = 0;  
while (i < n) do  
    begin  
        if(a <= k) then  
            a = a + 1;  
        i++;  
    end
```



# Summary

- Intermediate Code Generation
  - Using Syntax-Directed Translation Schemes
  - Conditional
  - Boolean using Short-Circuit Evaluation
  - Control-Flow
- Back-Patching
  - Allows Code Generation in a Single Pass

# Outline

## Compiler Optimization

- Local/pipeline optimizations
- Other sources of optimization
- Recognizing loops
- Dataflow analysis

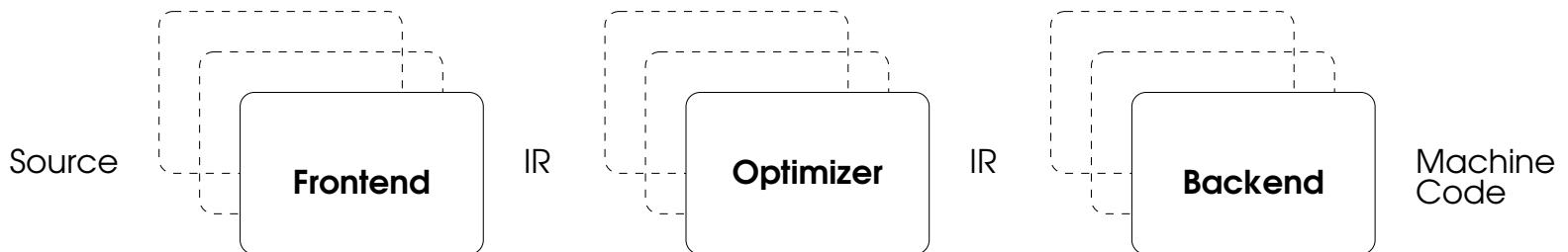
# Outline (cont'd)

## Optimizing (cont'd)

- Algorithms for global optimizations
- Alias analysis
- Code generation
  - Instruction selection
  - Register allocation
  - Instruction scheduling: improving ILP

Optimizations for cache behavior

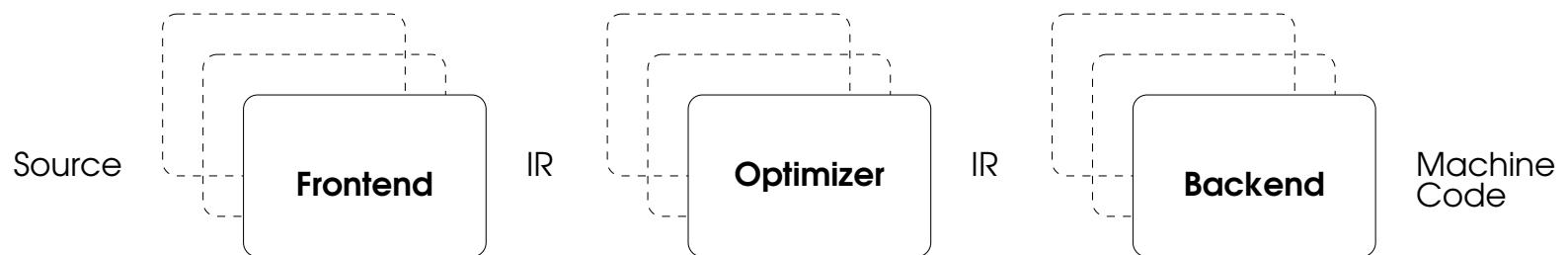
# Compilers



## Frontend

- Dependent on source language
- Lexical Analysis
- Parsing
- Semantic analysis (e.g., type checking)

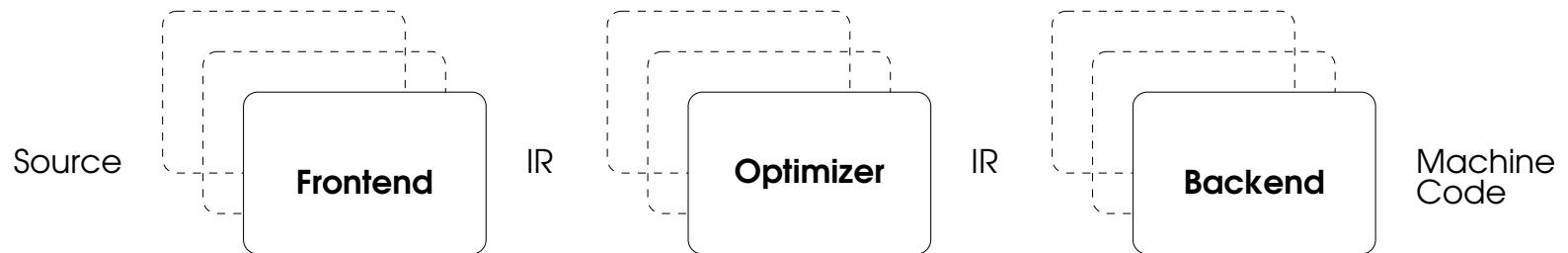
# Compilers



## Optimizer

- Independent part of compiler
- Different optimizations possible
- IR to IR translation

# Compilers



- Backend
  - Dependent on target
  - Code selection
  - Code scheduling
  - Register allocation
  - Peephole optimization

# Intermediate Representation (IR)

Flow graph

- Nodes are **basic blocks**
  - Basic blocks are single entry and single exit
- Edges represent control-flow
- Abstract Machine Code
  - Including the notion of functions and procedures
- Symbol table(s) keep track of scope and binding information about names

# Partitioning into basic blocks

1. Determine the leaders, which are:
  - The first statement
  - Any statement that is the target of a jump
  - Any statement that immediately follows a jump
2. For each leader its basic block consists of the leader and all statements up to but not including the next leader

## Partitioning into basic blocks (cont'd)

```
BB1 [ 1 prod=0  
      2 i=1  
      3 t1=4*i  
      4 t2=a[t1]  
      5 t3=4*i  
      6 t4=b[t3]  
      7 t5=t2*t4  
      8 t6=prod+t5  
      9 prod=t6  
     10 t7=i+i  
     11 i=t7  
     12 if i < 21 goto 3 ]  
  
BB2 [ ]
```

# Intermediate Representation (cont'd)

Structure within a basic block:

Abstract Syntax Tree (AST)

- Leaves are labeled by variable names or constants
- Interior nodes are labeled by an operator

Directed Acyclic Graph (DAG)

C-like

3 address statements (like we have already seen)

# Directed Acyclic Graph

Like ASTs:

- Leaves are labeled by variable names or constants
- Interior nodes are labeled by an operator

Nodes can have variable names attached that contain the value of that expression

Common subexpressions are represented by multiple edges to the same expression

## DAG creation

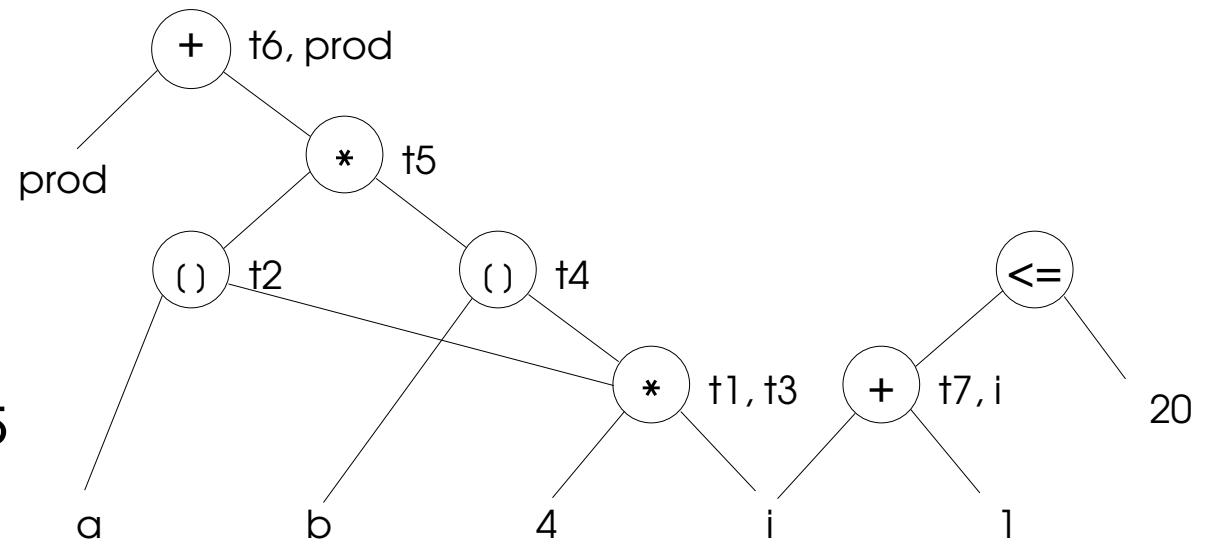
Suppose the following three address statements:

1.  $x = y \text{ op } z$
2.  $x = \text{op } y$
3.  $x = y$

$if(i <= 20) \dots$  will be treated like case 1 with  $x$  undefined

# DAG example

```
1  t1 = 4 * i
2  t2 = a[t1]
3  t3 = 4 * i
4  t4 = b[t3]
5  t5 = t2 * t4
6  t6 = prod + t5
7  prod = t6
8  t7 = i + 1
9  i = t7
10 if (i <= 20) goto 1
```



# Local optimizations

On basic blocks in the intermediate representation

- Machine independent optimizations

As a post code-generation step (often called **peephole optimization**)

- On a small “instruction window” (often a basic block)
- Includes machine specific optimizations

# Transformations on basic blocks

## Examples

### Function-preserving transformations

- Common subexpression elimination
- Constant folding
- Copy propagation
- Dead-code elimination
- Temporary variable renaming
- Interchange of independent statements

# **Transformations on basic blocks (cont'd)**

Algebraic transformations

Machine dependent eliminations/transformations

- Removal of redundant loads/stores
- Use of machine idioms

# Common subexpression elimination

If the same expression is computed more than once it is called a common subexpression

If the result of the expression is stored, we don't have to recompute it

- Moving to a DAG as IR, common subexpressions are automatically detected!

$$\begin{array}{ll} x = & + b \\ & \dots & \Rightarrow & \dots \\ y = & a + b & y = & x \end{array}$$

# Constant folding

Compute constant expression at compile time

- May require some emulation support

$$x = + 5 \qquad \qquad x = 8$$

$$\dots \qquad \qquad \Rightarrow \qquad \dots$$

$$y = x \qquad \qquad y = 16$$

# Copy propagation

Propagate original values when copied

Target for dead-code elimination

$$x = y \qquad \qquad x = y$$

$$\dots \qquad \qquad \Rightarrow \qquad \dots$$

$$z = x \qquad \qquad z = y$$

## **Dead-code elimination**

A variable  $x$  is dead at a statement if it is not used after that statement

An assignment  $x = y + z$  where  $x$  is dead can be safely eliminated

Requires live-variable analysis (discussed later on)

# Temporary variable renaming

$$\begin{array}{ll} 1 = & + b \\ & = 1 \\ \dots & \end{array} \qquad \Rightarrow \qquad \begin{array}{ll} 1 = & + b \\ & = 1 \\ \dots & \end{array}$$
$$\begin{array}{ll} 1 = d - e & t3 = d - e \\ c = t1 + 1 & c = t3 + 1 \end{array}$$

- If each statement that defines a temporary defines a new temporary, then the basic block is in **normal-form**
  - Makes some optimizations at BB level a lot simpler (e.g. common subexpression elimination, copy propagation, etc.)

# Algebraic transformations

There are many possible algebraic transformations

Usually only the common ones are implemented

$$x = x + 0$$

$$x = x - 1$$

$$x = x \quad \Rightarrow x = x << 1$$

$$x = x \Rightarrow x = x - x$$

# Machine dependent eliminations/transformations

Removal of redundant loads/stores

```
1  mov R0, a  
2  mov a, R0          // can be removed
```

Removal of redundant jumps, for example

```
1      beq ..., $Lx           bne ..., $Ly  
2      j $Ly                 ⇒   $Lx:    ...  
3  $Lx:    ...
```

Use of machine idioms, e.g.,

- Auto increment/decrement addressing modes
- SIMD instructions

Etc., etc. (see practical assignment)

# Other sources of optimizations

## Global optimizations

- Global common subexpression elimination
- Global constant folding
- Global copy propagation, etc.

## Loop optimizations

They all need some dataflow analysis on the flow graph

# Loop optimizations

## Code motion

- Decrease amount of code inside loop

Take a loop-invariant expression and place it before the loop

$$\text{while } (i \leq \text{lim}_i - ) \Rightarrow = \text{lim}_i - 2$$
$$\text{while } (i \leq t)$$

# Loop optimizations (cont'd)

## Induction variable elimination

Variables that are locked to the iteration of the loop are called  
**induction variables**

Example: in `for ( i = 0 ; i < 10 ; i++ )` *i* is an induction variable

Loops can contain more than one induction variable, for example,  
hidden in an array lookup computation

- Often, we can eliminate these extra induction variables

# Loop optimizations (cont'd)

## Strength reduction

Strength reduction is the replacement of expensive operations by cheaper ones (algebraic transformation)

- Its use is not limited to loops but can be helpful for induction variable elimination

$$\begin{array}{ll} i = i + 1 & i = i + 1 \\ 1 = i * 4 & \Rightarrow t1 = t1 + 4 \\ t2 = a[t1] & t2 = a[t1] \\ \text{if } (i < 10) \text{ goto top} & \text{if } (i < 10) \text{ goto top} \end{array}$$

# Loop optimizations (cont'd)

## Induction variable elimination (2)

Note that in the previous strength reduction we have to initialize 1 before the loop

After such strength reductions we can eliminate an induction variable

$$\begin{array}{l} i = i + 1 \\ t1 = t1 + 4 \\ t2 = a[t1] \\ \text{if } (i < 10) \text{ goto top} \end{array} \Rightarrow \begin{array}{l} 1 = 1 + 4 \\ t2 = a[t1] \\ \text{if } (t1 < 0) \text{ goto top} \end{array}$$

# Finding loops in flow graphs

## Dominator relation

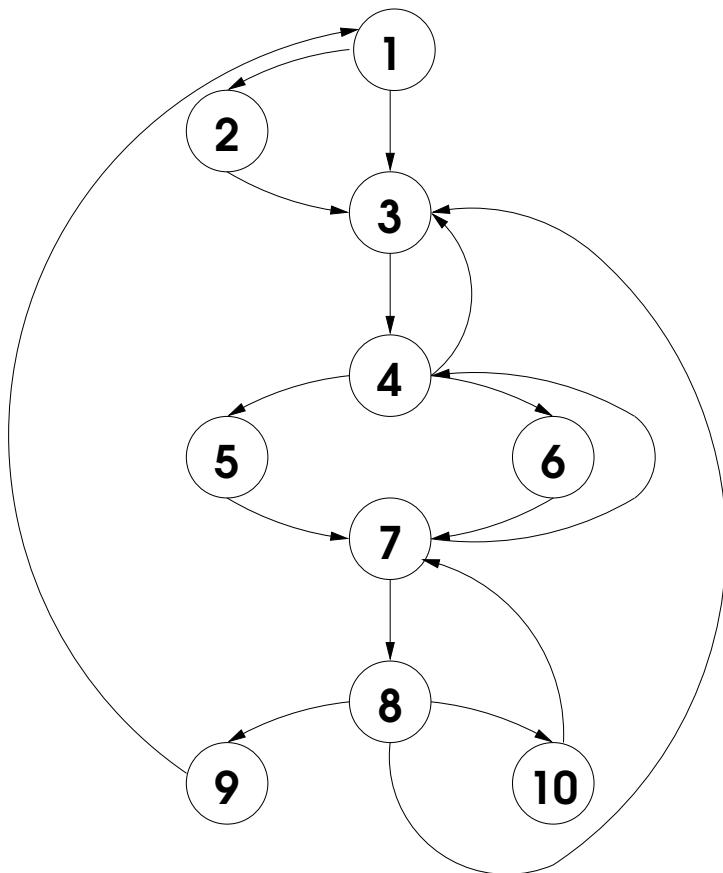
Node A dominates node B if all paths to node B go through node A

A node always dominates itself

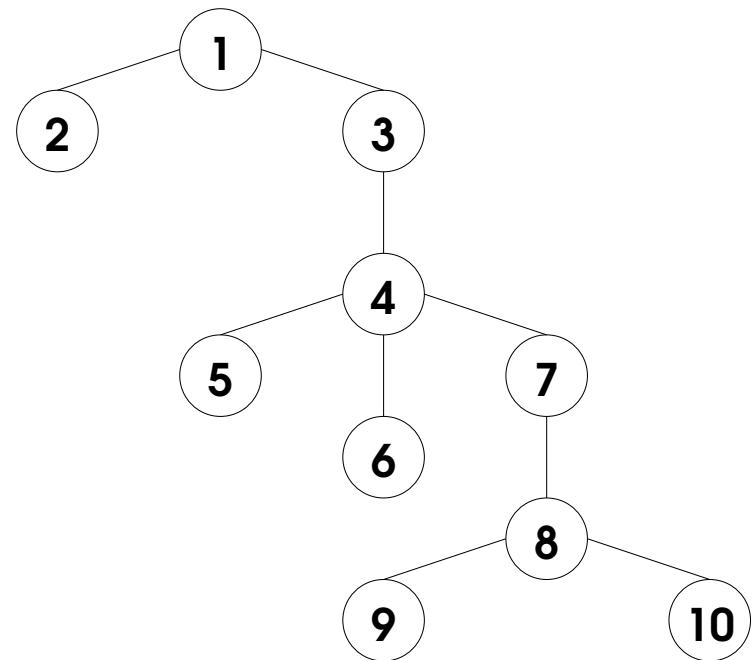
We can construct a tree using this relation: the Dominator tree

# Dominator tree example

Flow graph



Dominator tree



# Natural loops

A loop has a single entry point, **the header**, which dominates the loop

- There must be a path back to the header
- Loops can be found by searching for edges of which their heads dominate their tails, called the **backedges**
- Given a backedge  $n \rightarrow d$ , the **natural loop** is  $d$  plus the nodes that can reach  $n$  without going through  $d$

## Finding natural loop of $\rightarrow d$

```
procedure insert( $m$ ) {  
    if (not  $m \in loop$ ) {  
         $loop = loop \cup m$   
        push( $m$ )  
    }  
}  
  
 $st\_ck = \emptyset$   
 $loop = \{d\}$   
insert( )  
while ( $st\_ck \neq \emptyset$ ) {  
     $m = pop()$   
    for ( $p \in pred(m)$ ) insert( $p$ )  
}
```

## Natural loops (cont'd)

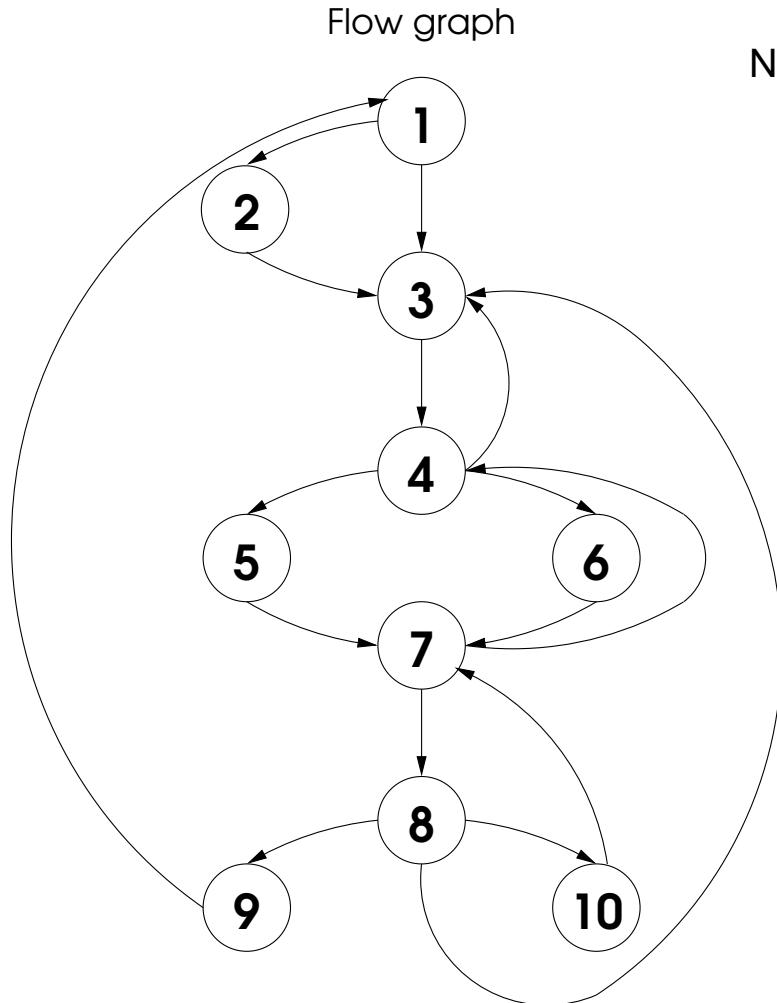
When two backedges go to the same header node, we may join the resulting loops

When we consider two natural loops, they are either completely disjoint or one is nested inside the other

The nested loop is called an **inner loop**

A program spends most of its time inside loops, so loops are a target for optimizations. This especially holds for inner loops!

# Our example revisited



Natural loops:

1. backedge  $10 \rightarrow 7$ :  $\{7,8,10\}$  (the inner loop)
2. backedge  $7 \rightarrow 4$ :  $\{4,5,6,7,8,10\}$
3. backedges  $4 \rightarrow 3$  and  $8 \rightarrow 3$ :  $\{3,4,5,6,7,8,10\}$
4. backedge  $9 \rightarrow 1$ : the entire flow graph

# Reducible flow graphs

A flow graph is reducible when the edges can be partitioned into forward edges and backedges

The forward edges must form an acyclic graph in which every node can be reached from the initial node

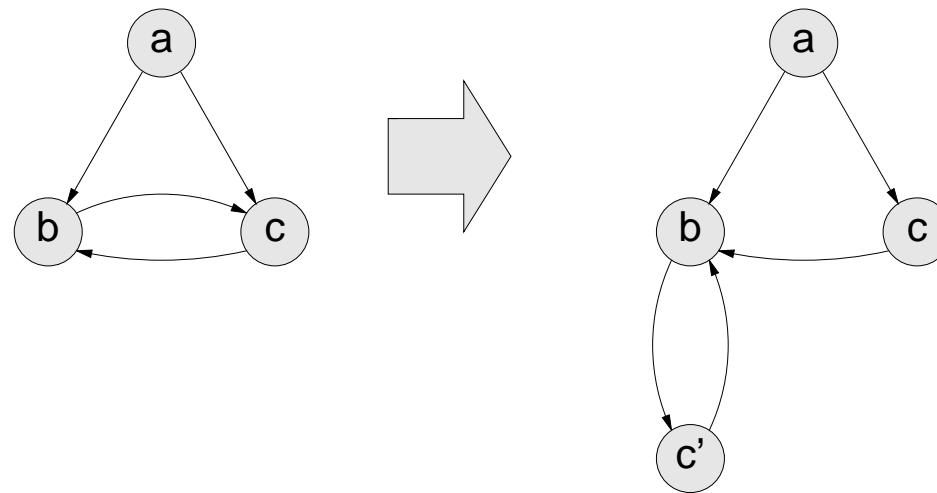
- Exclusive use of structured control-flow statements such as `if-then-else`, `while` and `break` produces reducible control-flow

Irreducible control-flow can create loops that cannot be optimized

## Reducible flow graphs (cont'd)

Irreducible control-flow graphs can always be made reducible

This usually involves some duplication of code



# Dataflow analysis

Data analysis is needed for global code optimization, e.g.:

- Is a variable live on exit from a block? Does a definition reach a certain point in the code?
- Dataflow equations are used to collect dataflow information
  - A typical dataflow equation has the form
    - o  $[S] = ge[S] \cup (i[S] - kil[S])$
- The notion of generation and killing depends on the dataflow analysis problem to be solved
- Let's first consider Reaching Definitions analysis for structured programs

# Reaching definitions

A definition of a variable  $x$  is a statement that assigns or may assign a value to  $x$

- An assignment to  $x$  is an **unambiguous** definition of  $x$
- An **ambiguous** assignment to  $x$  can be an assignment to a pointer or a function call where  $x$  is passed by reference

## Reaching definitions (cont'd)

When  $x$  is defined, we say the definition is generated

An unambiguous definition of  $x$  kills all other definitions of  $x$

When all definitions of  $x$  are the same at a certain point, we can use this information to do some optimizations

Example: all definitions of  $x$  define  $x$  to be 1. Now, by performing constant folding, we can do strength reduction if  $x$  is used in

$$z = y - x$$

# Dataflow analysis for reaching definitions

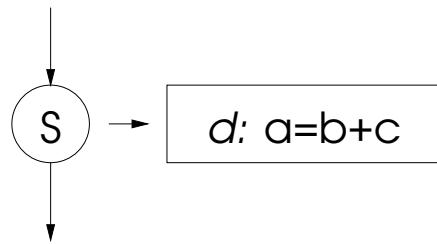
During dataflow analysis we have to examine every path that can be taken to see which definitions reach a point in the code

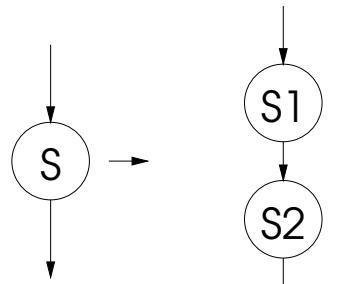
Sometimes a certain path will never be taken, even if it is part of the flow graph

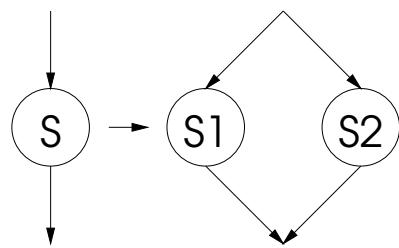
Since it is undecidable whether a path can be taken, we simply examine all paths

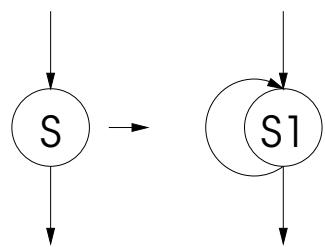
This won't cause false assumptions to be made for the code: it is a conservative simplification

- It merely causes optimizations not to be performed



$$\begin{aligned} \text{gen}[S] &= \{d\} \\ \text{kill}[S] &= D_a - \{d\} \\ \text{out}[S] &= \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S]) \end{aligned}$$


$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2]) \\ \text{kill}[S] &= \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2]) \\ \text{in}[S_1] &= \text{in}[S] \\ \text{in}[S_2] &= \text{out}[S_1] \\ \text{out}[S] &= \text{out}[S_2] \end{aligned}$$


$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_1] \cup \text{gen}[S_2] \\ \text{kill}[S] &= \text{kill}[S_1] \cap \text{kill}[S_2] \\ \text{in}[S_1] &= \text{in}[S_2] = \text{in}[S] \\ \text{out}[S] &= \text{out}[S_1] \cup \text{out}[S_2] \end{aligned}$$


$$\begin{aligned} \text{gen}[S] &= \text{gen}[S_1] \\ \text{kill}[S] &= \text{kill}[S_1] \\ \text{in}[S_1] &= \text{in}[S] \cup \text{gen}[S_1] \\ \text{out}[S] &= \text{out}[S_1] \end{aligned}$$

## Dealing with loops

The in-set to the code inside the loop is the in-set of the loop plus the out-set of the loop:  $i[S1] = in[S] \cup out[S1]$

- The out-set of the loop is the out-set of the code inside:  
 $out[S] = out[S1]$

Fortunately, we can also compute  $out[S1]$  in terms of  $in[S1]$ :

$$o[S1] = ge[S1] \cup (i[S1] - kil[S1])$$

## Dealing with loops (cont'd)

$I = \text{in}[S1]$ ,  $= \text{out}[S1]$ ,  $J = \text{in}[S]$ ,  $G = \text{gen}[S]$  and  $K = \text{kill}[S]$

$$I = I \cup O$$

- $O = G \cup (I - K)$

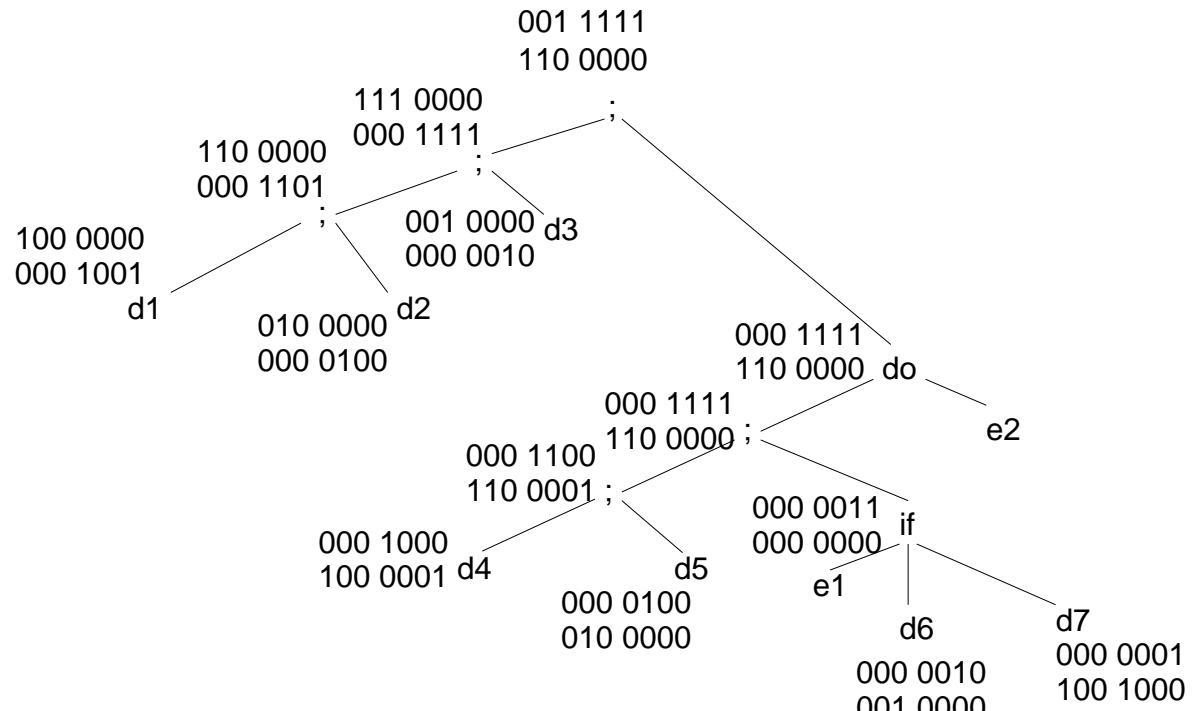
Assume  $O = \emptyset$ , then  $I^1 = J$

$$O^1 = G \cup (I^1 - K) = G \cup (J - K)$$

- $I^2 = J \cup O^1 = J \cup G \cup (J - K) = J \cup G$
- $O^2 = G \cup (I^2 - K) = G \cup (J \cup G - K) = G \cup (J - K)$
- $O^1 = O^2$  so  $\text{in}[S1] = \text{in}[S] \cup \text{gen}[S1]$  and  $\text{out}[S] = \text{out}[S1]$

# Reaching definitions example

```
d1    i = m - 1  
d      j = n  
d3    a = u1  
do  
d4    i = i + 1  
d5    j = j - 1  
if (e1)  
d6    a = u2  
else  
d7    i = u3  
while (e2)
```



In reality, dataflow analysis is often performed at the granularity of basic blocks rather than statements

# Iterative solutions

Programs in general need not be made up out of structured control-flow statements

We can do dataflow analysis on these programs using an iterative algorithm

The equations (at basic block level) for reaching definitions are:

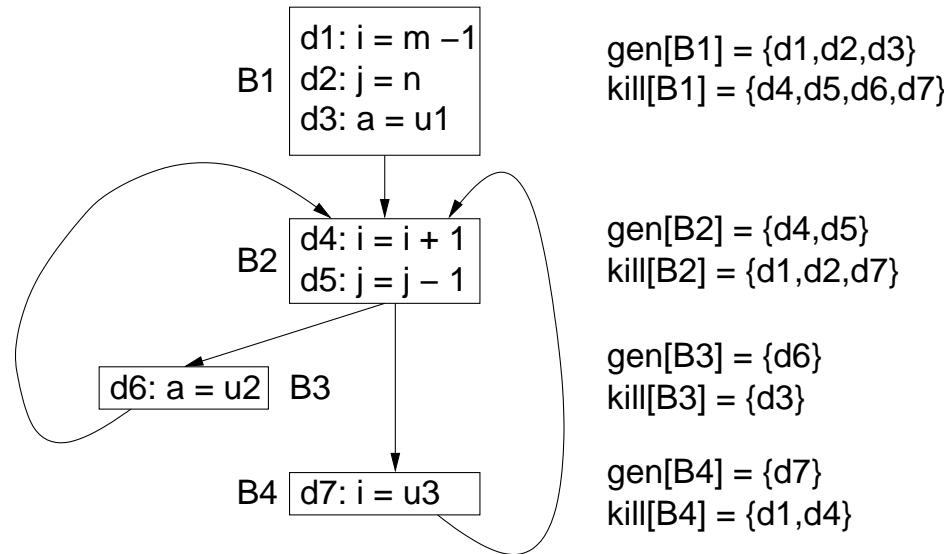
$$i[B] = \bigcup_{P \in pred(B)} out[P]$$

$$out[B] = gen[B] \cup (i[B] - kil[B])$$

# Iterative algorithm for reaching definitions

```
for (each block B)  $o_{[ ]} = gen[B]$ 
do {
    change = false
    for (each block B) {
         $i_{[ ]} = \bigcup_{P \in pred(B)} o_{[P]}$ 
        oldout =  $out[B]$ 
         $o_{[ ]} = ge_{[ ]} \cup (i_{[ ]} - kil_{[ ]})$ 
        if ( $out[B] = \text{oldout}$ ) change = true
    }
} while (change)
```

# Reaching definitions: an example



Block B	Initial		Pass 1		Pass 2	
	$in[B]$	$out[B]$	$in[B]$	$out[B]$	$in[B]$	$out[B]$
B1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

# Available expressions

An expression  $e$  is available at a point  $p$  if every path from the initial node to  $p$  evaluates  $e$ , and the variables used by  $e$  are not changed after the last evaluations

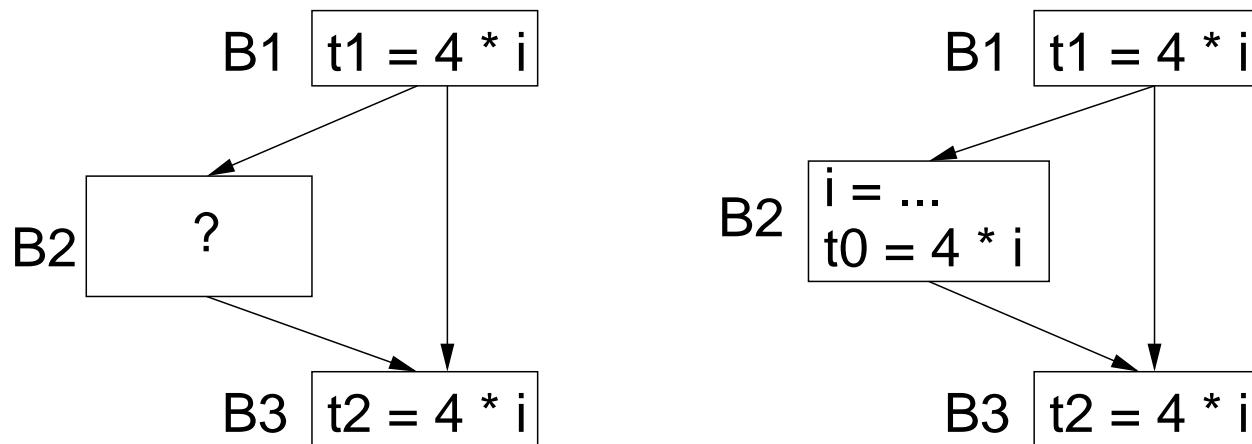
- An available expression  $e$  is killed if one of the variables used by  $e$  is assigned to

An available expression  $e$  is generated if it is evaluated

Note that if an expression  $e$  is assigned to a variable used by  $e$ , this expression will not be generated

## Available expressions (cont'd)

Available expressions are mainly used to find common subexpressions



## Available expressions (cont'd)

Dataflow equations:

$$o[ ] = e\_ge[ ] \cup (i[ ] - e\_kil[ ])$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \text{ for } B \text{ not initial}$$

$$i[1] = \emptyset \text{ where } B1 \text{ is the initial block}$$

The confluence operator is intersection instead of the union!

# Liveness analysis

A variable is live at a certain point in the code if it holds a value that may be needed in the future

Solve backwards:

- Find use of a variable
- This variable is live between statements that have found use as next statement
- Recurse until you find a definition of the variable

# Dataflow for liveness

Using the sets  $se[\ ]$  and  $def[\ ]$

- $def[B]$  is the set of variables assigned values in  $B$  prior to any use of that variable in  $B$
- $se[B]$  is the set of variables whose values may be used in  $B$  prior to any definition of the variable

A variable comes live into a block (in  $i[B]$ ), if it is either used before redefinition or it is live coming out of the block and is not redefined in the block

A variable comes live out of a block (in  $out[B]$ ) if and only if it is live coming into one of its successors

## Dataflow equations for liveness

$$i[ ] = use[B] \cup (out[B] - def[ ])$$

$$out[B] = \bigcup_{S \in succ[B]} in[S]$$

Note the relation between reaching-definitions equations: the roles of *in* and *out* are interchanged

# Algorithms for global optimizations

## Global common subexpression elimination

First calculate the sets of available expressions

For every statement  $s$  of the form  $x = y + z$  where  $y + z$  is available do the following

- Search backwards in the graph for the evaluations of  $y + z$
- Create a new variable  $u$
- Replace statements  $\quad = y + z$  by  $\quad = y + z; \quad =$
- Replace statement  $s$  by  $x =$

# Copy propagation

Suppose a copy statement  $s$  of the form  $x = y$  is encountered. We may now substitute a use of  $x$  by a use of  $y$  if

- Statement  $s$  is the only definition of  $x$  reaching the use
- On every path from statement  $s$  to the use, there are no assignments to  $y$

## Copy propagation (cont'd)

To find the set of copy statements we can use, we define a new dataflow problem

An occurrence of a copy statement generates this statement

An assignment to  $x$  or  $y$  kills the copy statement  $x = y$

Dataflow equations:

$$out[B] = c\_gen[B] \cup (i\_B - c\_kil[B])$$

$$in[B] = \bigcap_{P \in pred(B)} out[P] \text{ for } B \text{ not initial}$$

$$i\_B = \emptyset \text{ where } B_1 \text{ is the initial block}$$

## Copy propagation (cont'd)

For each copy statement  $s: x = y$  do

- Determine the uses of  $x$  reached by this definition of  $x$
- Determine if for each of those uses this is the only definition reaching it ( $s \in i [use]$ )
- If so, remove  $s$  and replace the uses of  $x$  by uses of  $y$

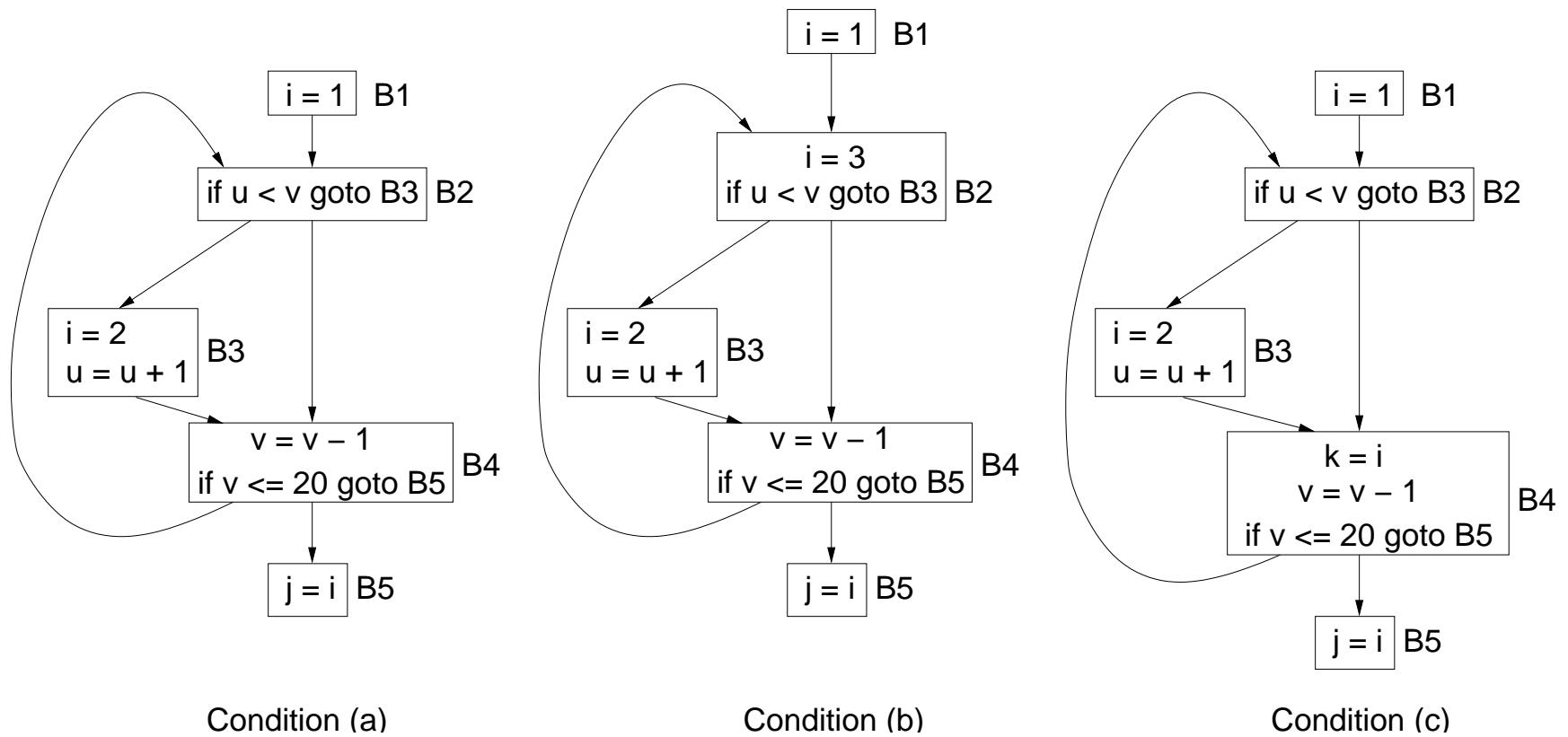
# Detection of loop-invariant computations

1. Mark **invariant** those statements whose operands are constant or have reaching definitions outside the loop
2. Repeat step 3 until no new statements are marked invariant
3. Mark invariant those statements whose operands either are constant, have reaching definitions outside the loop, or have one reaching definition that is marked invariant

# Code motion

1. Create a pre-header for the loop
2. Find loop-invariant statements
3. For each statement  $s$  defining  $x$  found in step 2, check that
  - (a) it is in a block that dominate all exits of the loop
  - (b)  $x$  is not defined elsewhere in the loop
  - (c) all uses of  $x$  in the loop can only be reached from this statement  $s$
4. Move the statements that conform to the pre-header

# Code motion (cont'd)



# Detection of induction variables

A basic induction variable  $i$  is a variable that only has assignments of the form  $i = i + c$

Associated with each induction variable  $j$  is a triple  $(i, c, d)$  where  $i$  is a basic induction variable and  $c$  and  $d$  are constants such that  $j = c * i + d$

- In this case  $j$  belongs to the family of  $i$
- The basic induction variable  $i$  belongs to its own family, with the associated triple  $(i, 1, 0)$

## Detection of induction variables (cont'd)

Find all basic induction variables in the loop

Find variables  $k$  with a single assignment in the loop with one of the following forms:

- $k = j * b, k = b * j, k = j/b, k = j + b, k = b + j$ , where  $b$  is a constant and  $j$  is an induction variable

If  $j$  is not basic and in the family of  $i$  then there must be

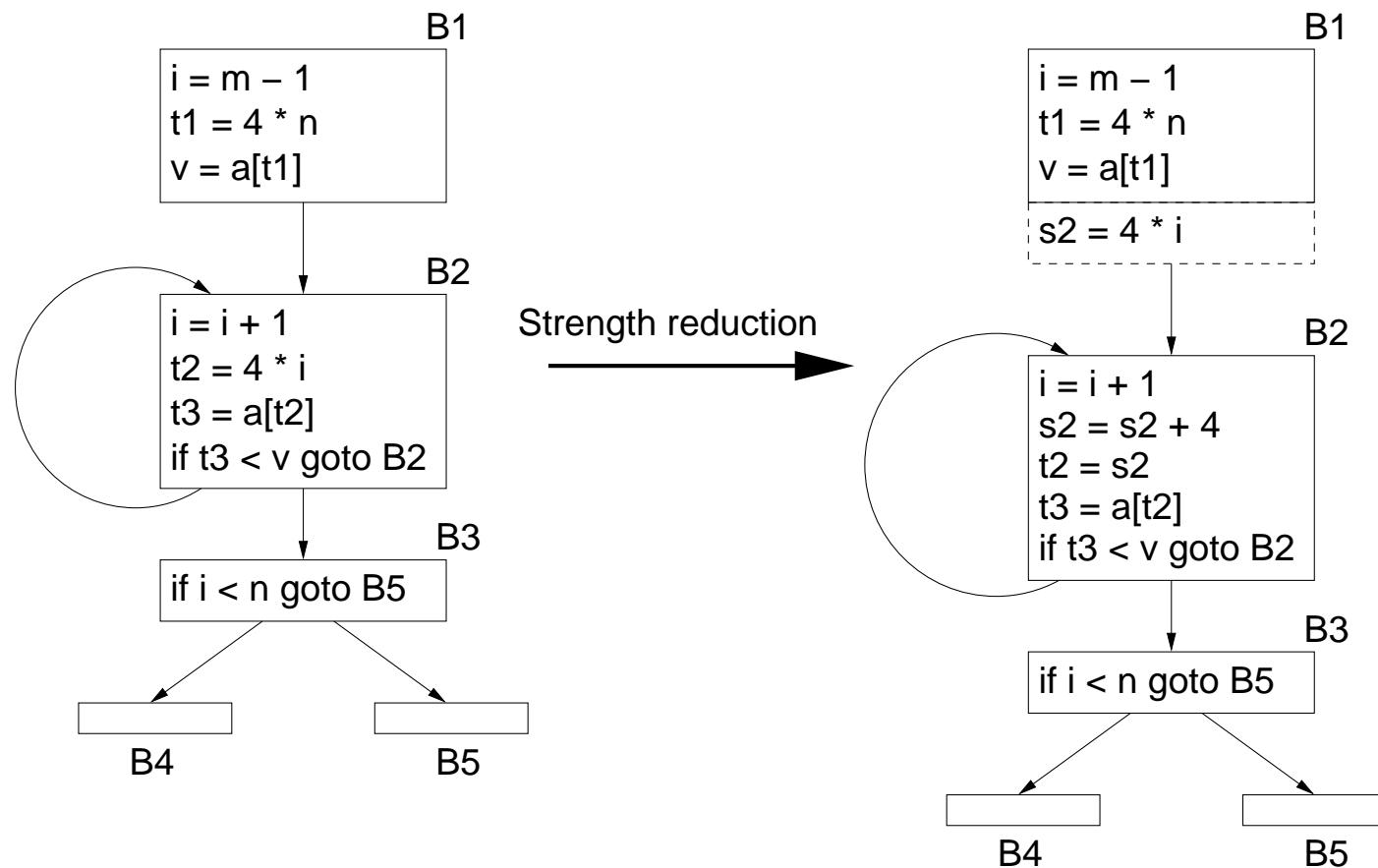
- No assignment of  $i$  between the assignment of  $j$  and  $k$
- No definition of  $j$  outside the loop that reaches  $k$

# Strength reduction for induction variables

Consider each basic induction variable  $i$  in turn. For each variable  $j$  in the family of  $i$  with triple  $(i, c, d)$ :

- Create a new variable  $s$
- Replace the assignment to  $j$  by  $j = s$
- Immediately after each assignment  $i = i$         append  
$$s = s + c$$
- Place  $s$  in the family of  $i$  with triple  $(i, c, d)$
- Initialize  $s$  in the preheader:  $s = c * i + d$

# Strength reduction for induction variables (cont'd)



# Elimination of induction variables

Consider each basic induction variable  $i$  only used to compute other induction variables and tests

Take some  $j$  in  $i$ 's family such that  $c$  and  $d$  from the triple  $(i, c, d)$  are simple

Rewrite tests `if (i relop x) to`

$$= c * x + d; \text{ if } (j \text{ relop } )$$

- Delete assignments to  $i$  from the loop
- Do some copy propagation to eliminate  $j = s$  assignments formed during strength reduction

# Alias Analysis

Aliases, e.g. caused by pointers, make dataflow analysis more complex (uncertainty regarding what is defined and used:  $x = *p$  might use any variable)

Use dataflow analysis to determine what a pointer might point to

$in[B]$  contains for each pointer  $p$  the set of variables to which  $p$  could point at the beginning of block

- Elements of  $i[B]$  are pairs  $(p, v)$  where  $p$  is a pointer and  $v$  a variable, meaning that  $p$  might point to  $v$
- $o[B]$  is defined similarly for the end of

## Alias Analysis (cont'd)

Define a function  $s_B$  such that  $s_B(i[B]) = \text{out}[B]$

- $ra\ s_B$  is composed of  $s_s$ , for each statement  $s$  of block
  - If  $s$  is  $p =$  or  $p = \pm c$  in case  $\square$  is an array, then
$$s_s(S) = (S - \{(p, b) \mid \text{any variable } b\}) \cup \{(p, \square)\}$$
  - If  $s$  is  $p = q \pm c$  for pointer  $q$  and nonzero integer  $c$ , then
$$\begin{aligned} ra\ s_s(S) &= (S - \{(p, b) \mid \text{any variable } b\}) \\ &\cup \{(p, b) \mid (q, b) \in S \text{ and } b \text{ is an array variable}\} \end{aligned}$$
  - If  $s$  is  $p = q$ , then
$$\begin{aligned} ra\ s_s(S) &= (S - \{(p, b) \mid \text{any variable } b\}) \\ &\cup \{(p, b) \mid (q, b) \in S\} \end{aligned}$$

## Alias Analysis (cont'd)

- If  $s$  assigns to pointer  $p$  any other expression, then

$$s_s(S) = S - \{(p, b \mid \text{any variable } b\}$$

- If  $s$  is not an assignment to a pointer, then  $ra \ s_s(S) = S$

Dataflow equations for alias analysis:

$$o [ ] = s_B(i [ ])$$

$$i [ ] = \bigcup_{P \in pred(B)} out[P]$$

where  $s_B(S) = s_{s_k}(\dots s_{s_{k-1}}(\dots (tra \ s_{s_1}(S$

# Alias Analysis (cont'd)

How to use the alias dataflow information? Examples:

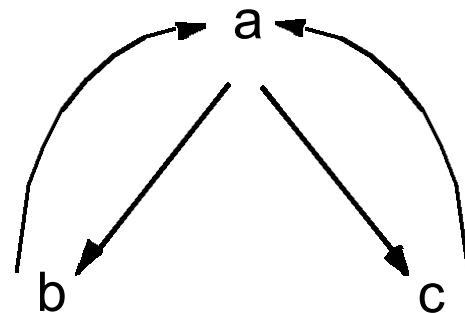
- In reaching definitions analysis (to determine  $ge$  and  $kil$ )  
statement  $p =$  generates a definition of every variable  $b$  such that  $p$  could point to  $b$   
 $\rightarrow *p = a$  kills definition of  $b$  only if  $b$  is not an array and is the only variable  $p$  could possibly point to (to be conservative)
- In liveness analysis (to determine  $def$  and  $se$ )  
 $p = a$  uses  $p$  and  $a$ . It defines  $b$  only if  $b$  is the unique variable that  $p$  might point to (to be conservative)  
 $\rightarrow = *p$  defines , and represents the use of  $p$  and a use of any variable that could point to

# **Compiler Optimization**

## **LICM: Loop Invariant Code Motion**

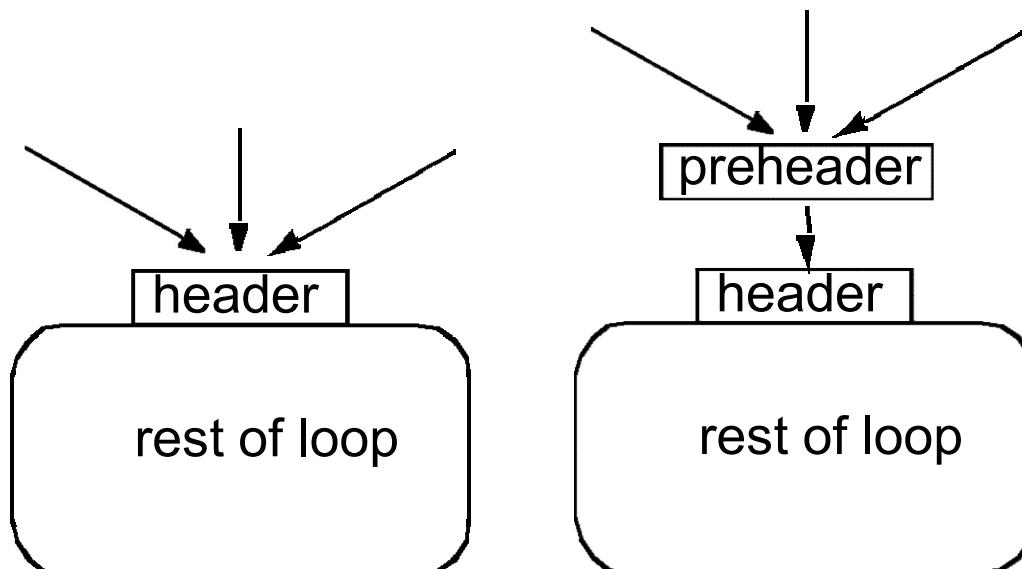
# Inner Loops

- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Combine as one



# Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop

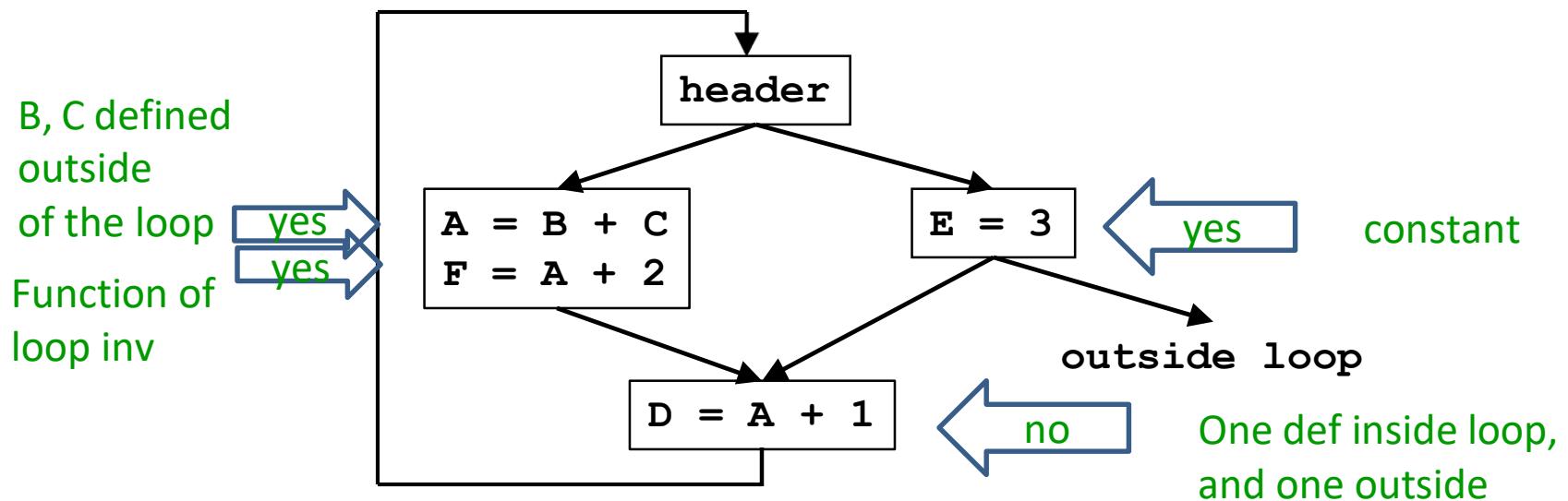


# Finding Loops: Summary

- Define loops in graph theoretic terms
- Definitions and algorithms for:
  - Dominators
  - Back edges
  - Natural loops

# Loop-Invariant Computation and Code Motion

- A loop-invariant computation:
  - a computation whose value does not change as long as control stays within the loop
- Code motion:
  - to move a statement within a loop to the preheader of the loop



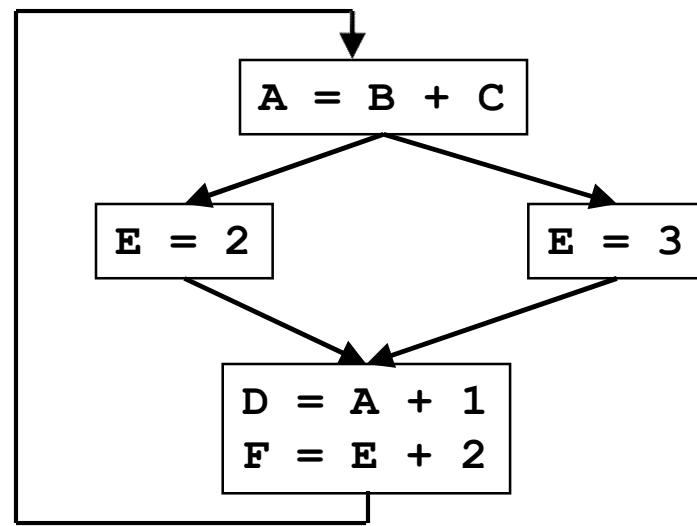
# Algorithm

- **Observations**
  - Loop invariant
    - operands are defined outside loop or invariant themselves
  - Code motion
    - not all loop invariant instructions can be moved to preheader
- **Algorithm**
  - Find invariant expressions
  - Conditions for code motion
  - Code transformation

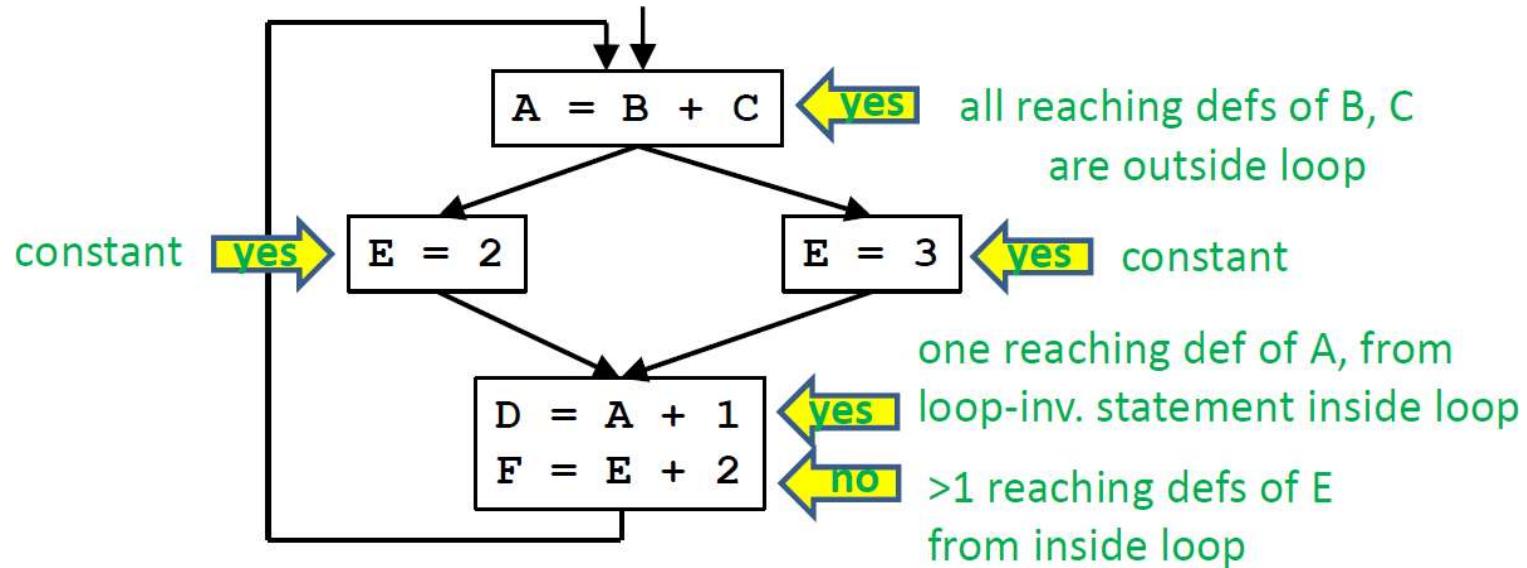
# Detecting Loop Invariant Computation

- Compute reaching definitions
  - Mark INVARIANT if
    - all the definitions of B and C that reach a statement  $A=B+C$  are outside the loop
      - constant B, C?
  - Repeat: Mark INVARIANT if
    - all reaching definitions of B are outside the loop, or
    - there is exactly one reaching definition for B, and it is from a loop-invariant statement inside the loop
    - similarly for C
- until no changes to set of loop-invariant statements occur.

# Example

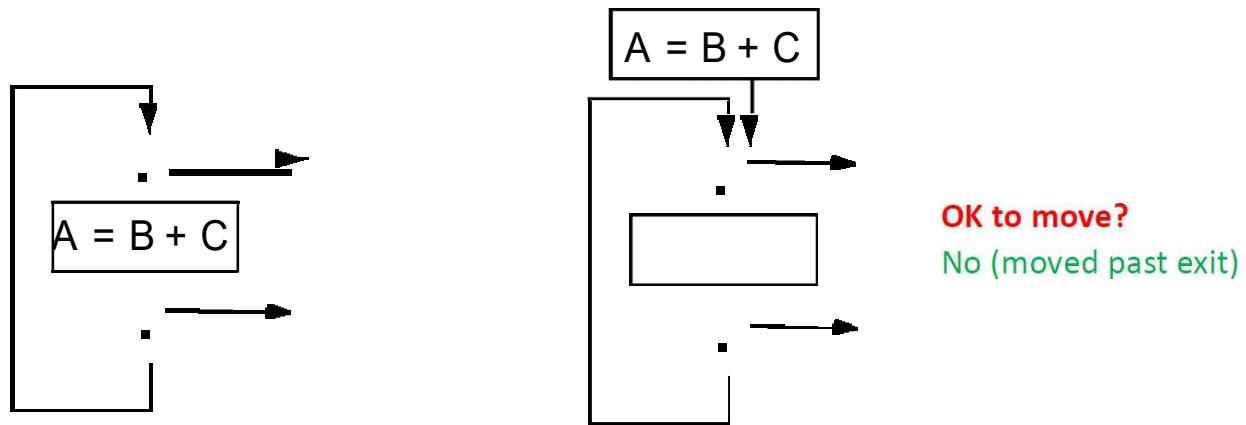


# Example

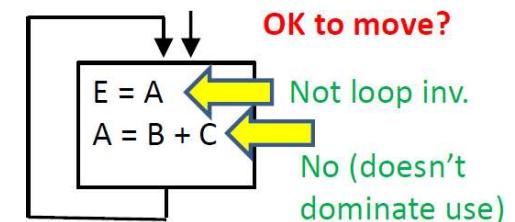


# Conditions for Code Motion

- **Correctness:** Movement does not change semantics of program
- **Performance:** Code is not slowed down



- **Basic idea:** defines once and for all
  - control flow: once?  
Code dominates all exists
  - other definitions: for all?  
No other definition
  - other uses: for all?  
Dominates use or no other reaching defs to use

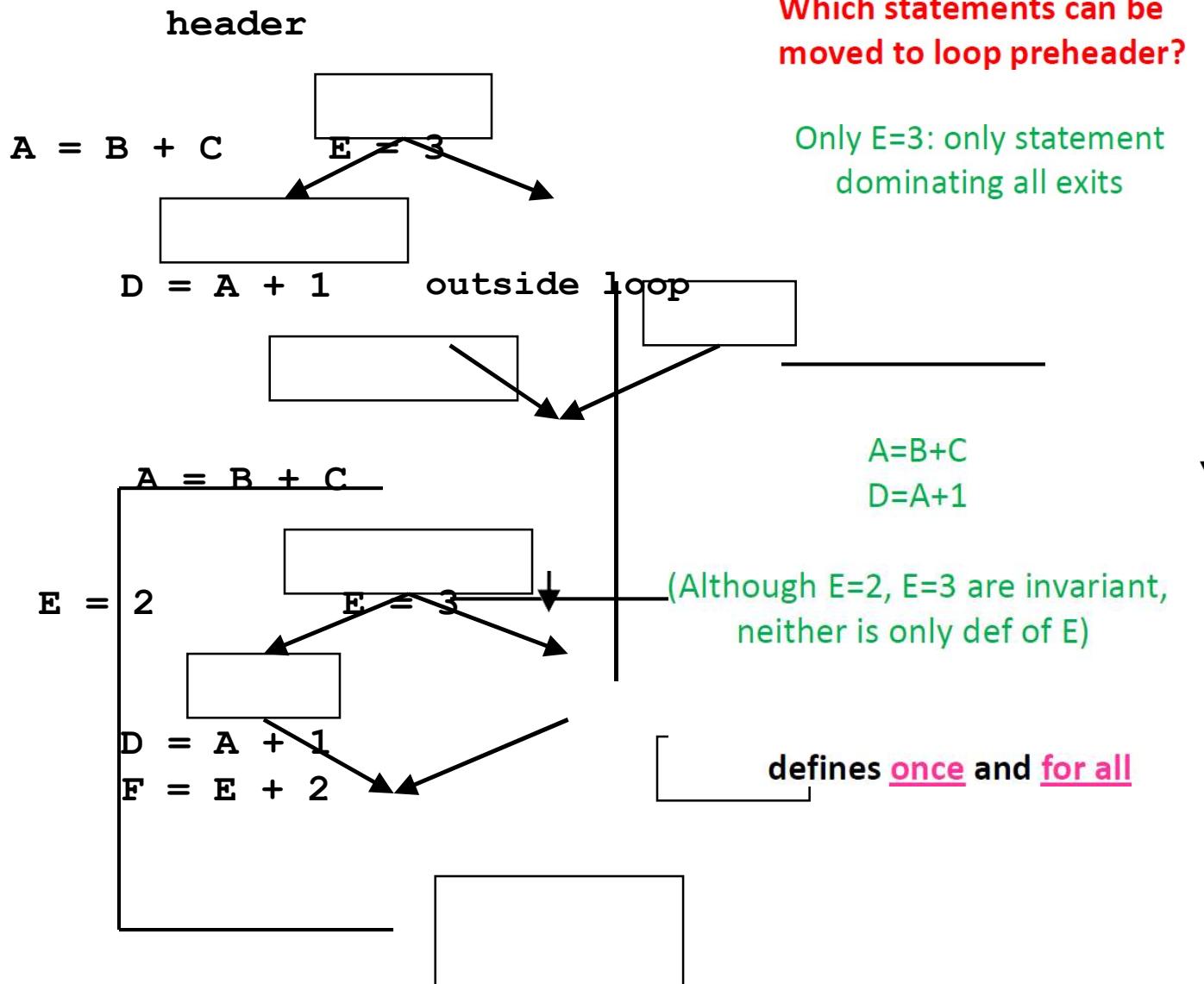


# Code Motion Algorithm

Given: a set of nodes in a loop

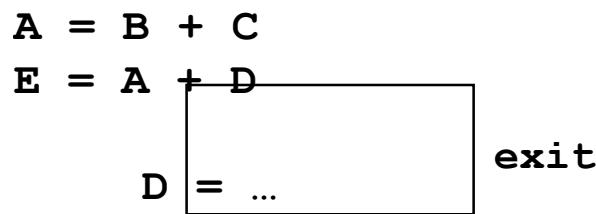
- **Compute reaching definitions**
- **Compute loop invariant computation**
- **Compute dominators**
- **Find the exits of the loop (i.e., nodes with successor outside loop)**
- **Candidate statement for code motion:**
  - loop invariant
  - in blocks that dominate all the exits of the loop
  - assign to variable not assigned to elsewhere in the loop
  - in blocks that dominate all blocks in the loop that use the variable assigned
- **Perform a depth-first search of the blocks**
  - Move candidate to preheader if all the invariant operations it depends upon have been moved

# Examples

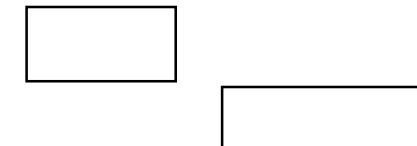


# More Aggressive Optimizations

- **Gamble on: most loops get executed**
  - Can we relax constraint of dominating all exits?



Can relax if destination not live after loop  
& can compute in preheader  
w/o causing an exception



Ensures preheader  
executes only  
if enter loop

- **Landing pads**

While p do s → if p {  
    preheader  
    repeat  
        s  
    until not p;  
}

# LICM Summary

- Precise definition and algorithm for loop invariant computation
- Precise algorithm for code motion
- Use of reaching definitions and dominators in optimizations

# **Induction Variables and Strength Reduction**

- I. Overview of optimization
- II. Algorithm to find induction variables

# Example

```
FOR i = 0 to 100
    A[i] = 0;

i = 0

L2: IF i>=100 GOTO L1
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i+1
    GOTO L2

L1:
```

# Definitions

- A **basic induction variable** is
  - a variable  $X$  whose only definitions within the loop are assignments of the form:  
$$X = X + c \text{ or } X = X - c,$$
where  $c$  is either a **constant** or a **loop-invariant variable**.
- An **induction variable** is
  - a **basic induction variable**, or
  - a variable **defined once** within the loop, whose value is a **linear function** of some **basic induction variable** at the time of the definition:  
$$A = c_1 * B + c_2$$
- The **FAMILY of a basic induction variable  $B$**  is
  - the set of induction variables  $A$  such that each time  $A$  is assigned in the loop, the value of  $A$  is a linear function of  $B$ .

# Optimizations

## 1. Strength reduction:

- A is an induction variable in family of basic induction variable B ( $A = c_1 * B + c_2$ )
  - Create new variable:  $A'$
  - Initialization in preheader:  $A' = c_1 * B + c_2;$
  - Track value of B: add after  $B=B+x$ :  $A'=A'+x*c_1;$
  - Replace assignment to A:  $A=A'$

# Optimizations (continued)

## 2. Optimizing **non-basic** induction variables

- copy propagation
- dead code elimination

## 3. Optimizing **basic** induction variables

- Eliminate basic induction variables used only for
  - calculating other induction variables and loop tests
- Algorithm:
  - Select an **induction variable A in the family of B**, preferably with simple constants ( $A = c_1 * B + c_2$ ).
  - Replace a comparison such as

```
if B > x goto L1
```

with

```
if (A' > c1 * x + c2) goto L1
```

(assuming  $c_1$  is positive)
  - **if B is live** at any exit from the loop, **recompute it from A'**
    - After the exit,  $B = (A' - c_2) / c_1$

# II. Basic Induction Variables

- **A BASIC induction variable in a loop L**
  - a variable  $X$  whose **only definitions within L** are assignments of the form:  
 $X = X+c$  or  $X = X-c$ , where  $c$  is either a constant or a loop-invariant variable.
- **Algorithm:** can be detected by scanning L
- **Example:**

```
k = 0;
for (i = 0; i < n; i++) {
    k = k + 3;
    ... = m;
    if (x < y)
        k = k + 4;
    if (a < b)
        m = 2 * k;
    k = k - 2;
    ... = m;
```

*Each iteration may execute a different number of increments/decrements!!*

# Strength Reduction Algorithm

- **Key idea:**
  - For each induction variable A, ( $A = c_1 * B + c_2$  at time of definition)
    - variable  $A'$  holds expression  $c_1 * B + c_2$  at all times
    - replace definition of A with  $A=A'$  only when executed
- **Result:**
  - Program is correct
  - Definition of A does not need to refer to B

# Finding Induction Variable Families

- Let B be a basic induction variable
  - Find all induction variables A in family of B:
    - $A = c_1 * B + c_2$   
(where B refers to the value of B at time of definition)
- Conditions:
  - If A has a single assignment in the loop L, and assignment is one of:

$A = B * c$   
 $A = c * B$   
 $A = B / c$  (assuming  $A$  is real)  
 $A = B + c$   
 $A = c + B$   
 $A = B - c$   
 $A = c - B$

- OR, ... (next page)

# Finding Induction Variable Families (continued)

Let D be an induction variable in the family of B ( $D = c_1 * B + c_2$ )

- If A has a single assignment in the loop L, and assignment is one of:

```
A = D * C
A = C * D
A = D / C      (assuming A is real)
A = D + C
A = C + D
A = D - C
A = C - D
```

- No definition of D outside L reaches the assignment to A
- Between the lone point of assignment to D in L and the assignment to A, there are no definitions of B

# Summary

- **Precise definitions of induction variables**
- **Systematic identification of induction variables**
- **Strength reduction**
- **Clean up:**
  - eliminating basic induction variables
    - used in other induction variable calculations
    - replacement of loop tests
  - eliminating other induction variables
    - standard optimizations



# Data-flow Analysis

Y.N. Srikant

Department of Computer Science and Automation  
Indian Institute of Science  
Bangalore 560 012

NPTEL Course on Compiler Design

# Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point  $p_1$  to point  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that for each  $i = 1, 2, \dots, n - 1$ , either
  - 1  $p_i$  is the point immediately preceding a statement and  $p_{i+1}$  is the point immediately following that same statement, or
  - 2  $p_i$  is the end of some block and  $p_{i+1}$  is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

# Uses of Data-flow Analysis

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

# Data-Flow Analysis Schema

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of definitions in the program
  - A particular data-flow value is a set of definitions
- $IN[s]$  and  $OUT[s]$ : data-flow values *before* and *after* each statement  $s$
- The *data-flow problem* is to find a solution to a set of constraints on  $IN[s]$  and  $OUT[s]$ , for all statements  $s$

## Data-Flow Analysis Schema (2)

- Two kinds of constraints
  - Those based on the semantics of statements (*transfer functions*)
  - Those based on flow of control
- A DFA schema consists of
  - A control-flow graph
  - A direction of data-flow (forward or backward)
  - A set of data-flow values
  - A confluence operator (normally set union or intersection)
  - Transfer functions for each block
- We always compute *safe* estimates of data-flow values
- A decision or estimate is *safe* or *conservative*, if it never leads to a change in what the program computes (after the change)
- These safe values may be either subsets or supersets of actual values, based on the application

# The Reaching Definitions Problem

- We *kill* a definition of a variable  $a$ , if between two points along the path, there is an assignment to  $a$
- A definition  $d$  reaches a point  $p$ , if there is a path from the point immediately following  $d$  to  $p$ , such that  $d$  is not *killed* along that path
- Unambiguous and ambiguous definitions of a variable

$a := b+c$

(unambiguous definition of 'a')

...

$*p := d$

(ambiguous definition of 'a', if 'p' may point to variables other than 'a' as well; hence does not kill the above definition of 'a')

...

$a := k-m$

(unambiguous definition of 'a'; kills the above definition of 'a')

## The Reaching Definitions Problem(2)

- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both  $a=2$  and  $a=4$  reach the point after the complete if-then-else statement, even though the statement  $a=4$  is not reached by control flow

```
if (a==b) a=2; else if (a==b) a=4;
```

# The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

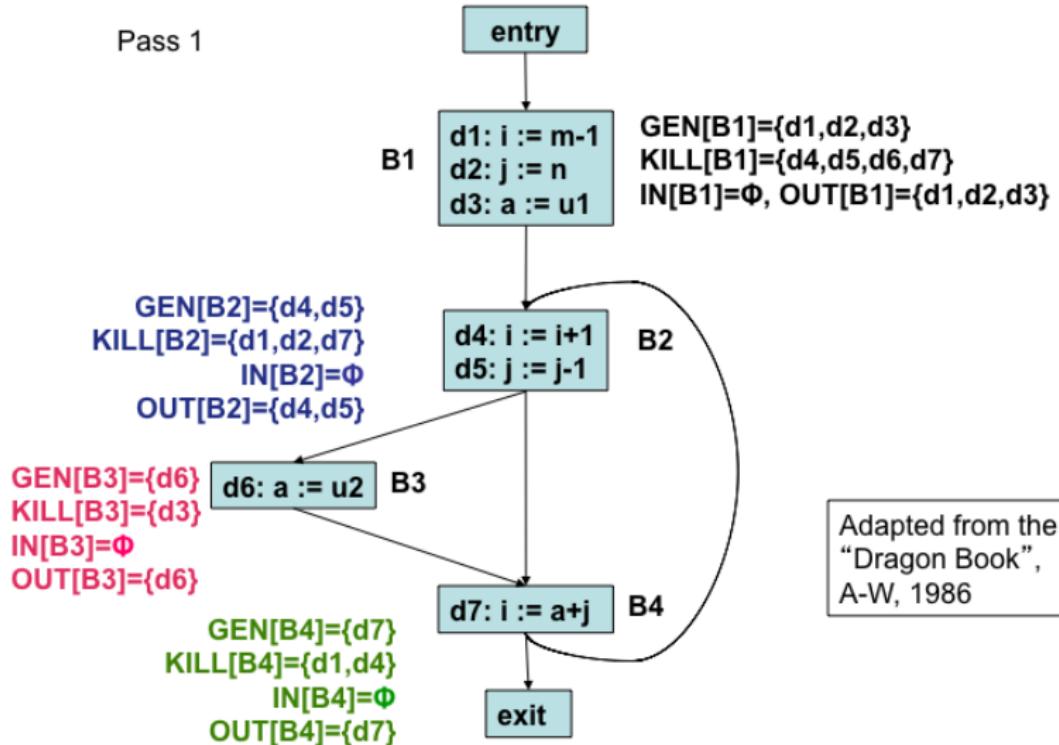
*P is a predecessor of B*

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

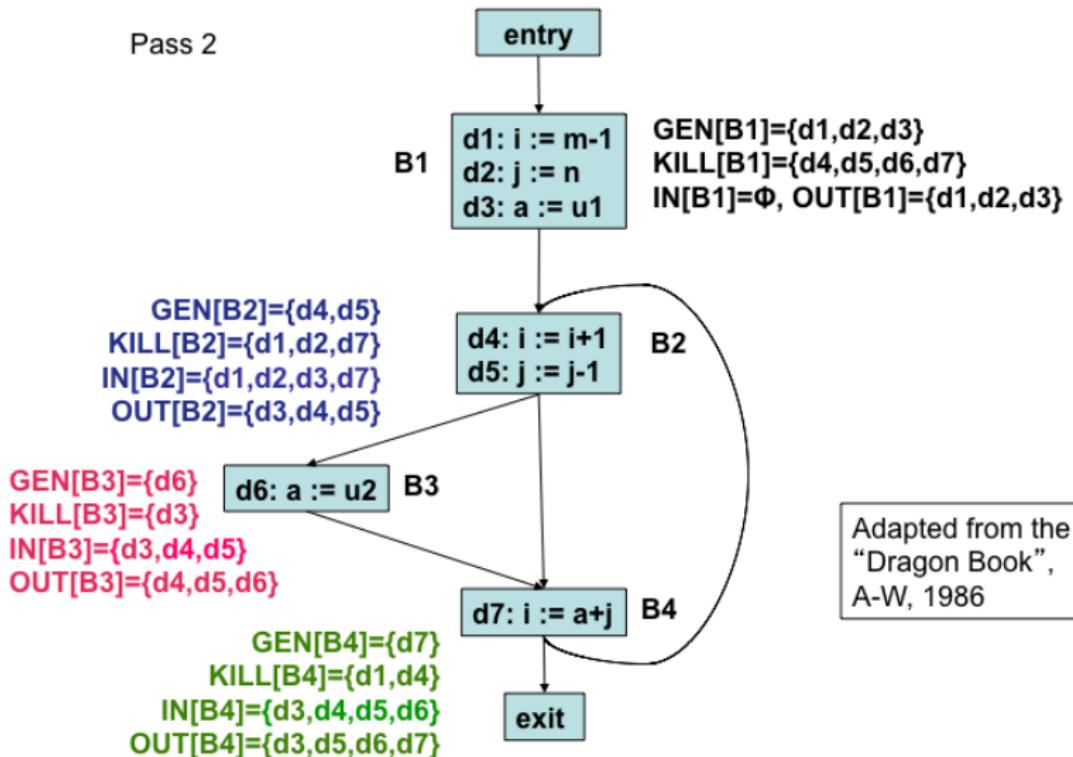
$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

- If some definitions reach  $B_1$  (entry), then  $IN[B_1]$  is initialized to that set
- Forward flow DFA problem (since  $OUT[B]$  is expressed in terms of  $IN[B]$ ), confluence operator is  $\cup$
- $GEN[B]$  = set of all definitions inside  $B$  that are “visible” immediately after the block - *downwards exposed* definitions
- $KILL[B]$  = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in  $B$

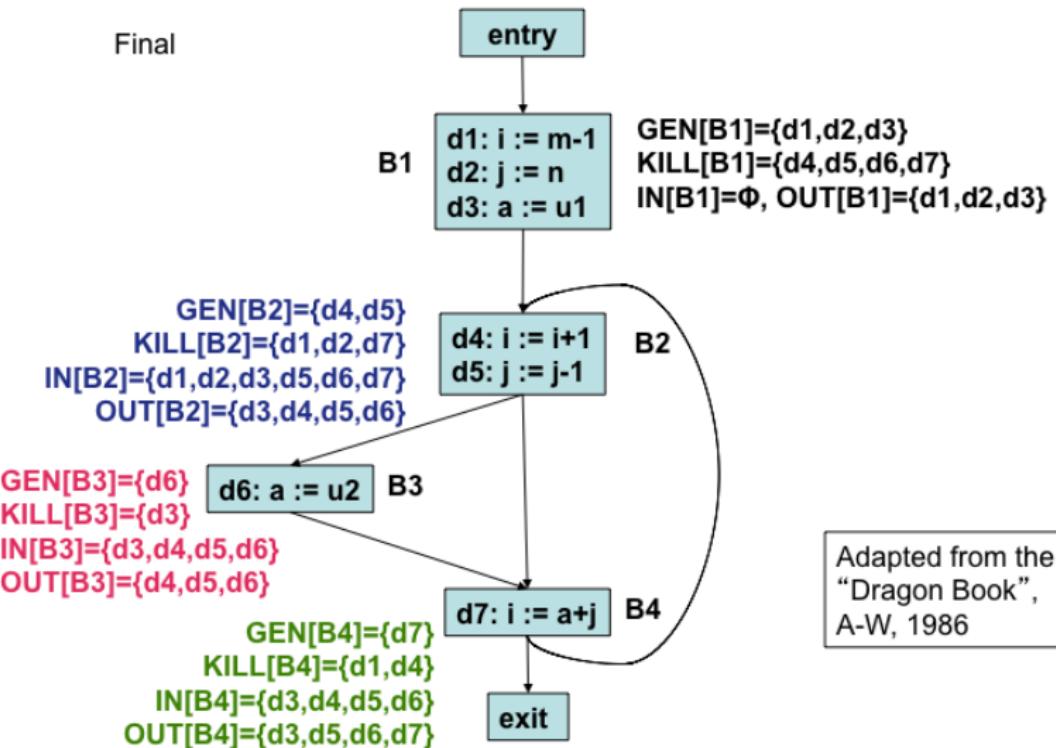
# Reaching Definitions Analysis: An Example - Pass 1



# Reaching Definitions Analysis: An Example - Pass 2



# Reaching Definitions Analysis: An Example - Final



# An Iterative Algorithm for Computing Reaching Definitions

for each block  $B$  do {  $IN[B] = \phi$ ;  $OUT[B] = GEN[B]$ ; }

$change = true$ ;

while  $change$  do {  $change = false$ ;

for each block  $B$  do {

$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];$$

$$oldout = OUT[B];$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);$$

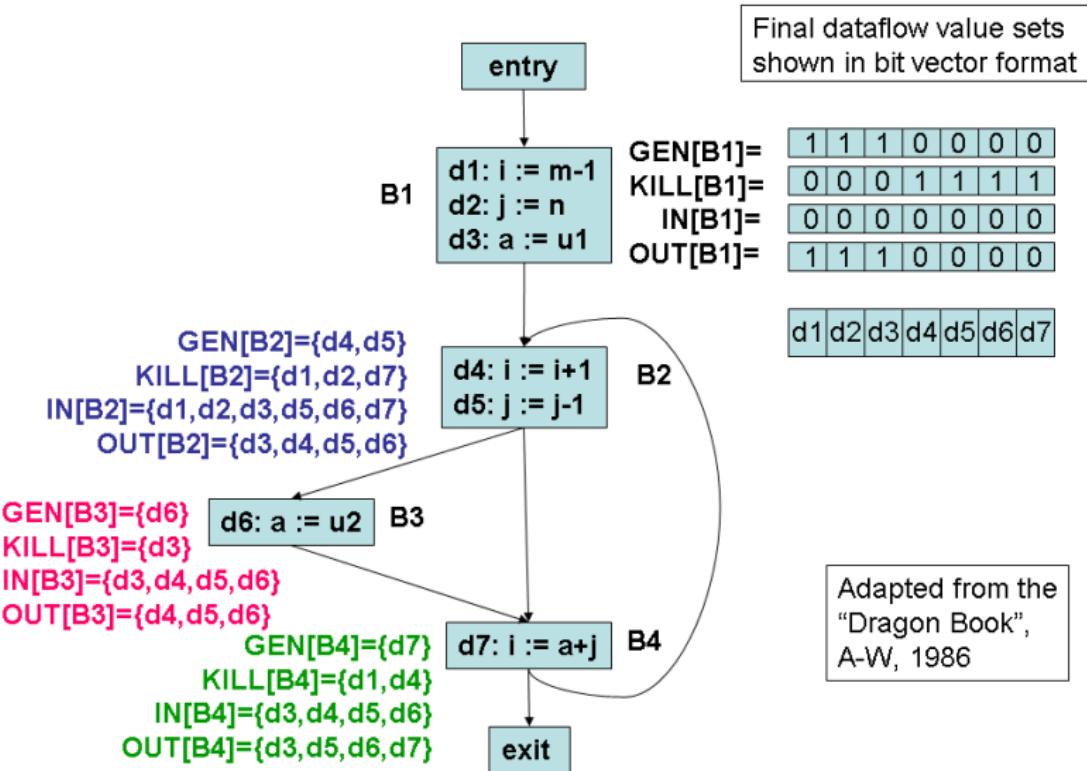
if ( $OUT[B] \neq oldout$ )  $change = true$ ;

}

}

- $GEN$ ,  $KILL$ ,  $IN$ , and  $OUT$  are all represented as bit vectors with one bit for each definition in the flow graph

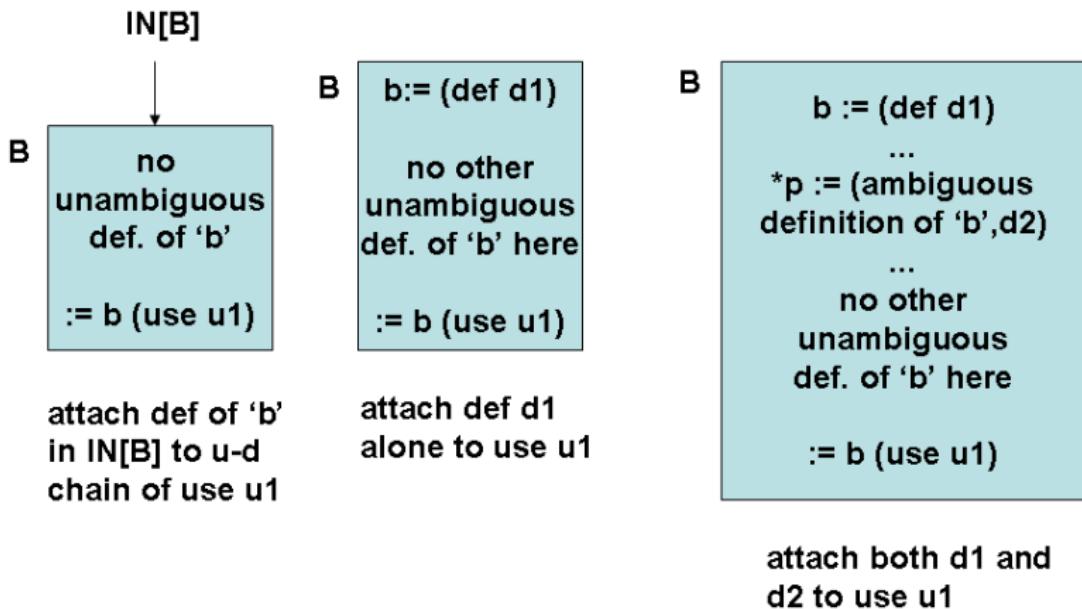
# Reaching Definitions: Bit Vector Representation



## Use-Definition Chains (u-d chains)

- Reaching definitions may be stored as u-d chains for convenience
- A u-d chain is a list of a use of a variable and all the definitions that reach that use
- u-d chains may be constructed once reaching definitions are computed
- **case 1:** If use  $u_1$  of a variable  $b$  in block B is preceded by no unambiguous definition of  $b$ , then attach all definitions of  $b$  in  $IN[B]$  to the u-d chain of that use  $u_1$  of  $b$
- **case 2:** If any unambiguous definition of  $b$  precedes a use of  $b$ , then *only that definition* is on the u-d chain of that use of  $b$
- **case 3:** If any ambiguous definitions of  $b$  precede a use of  $b$ , then each such definition for which no unambiguous definition of  $b$  lies between it and the use of  $b$ , are on the u-d chain for this use of  $b$

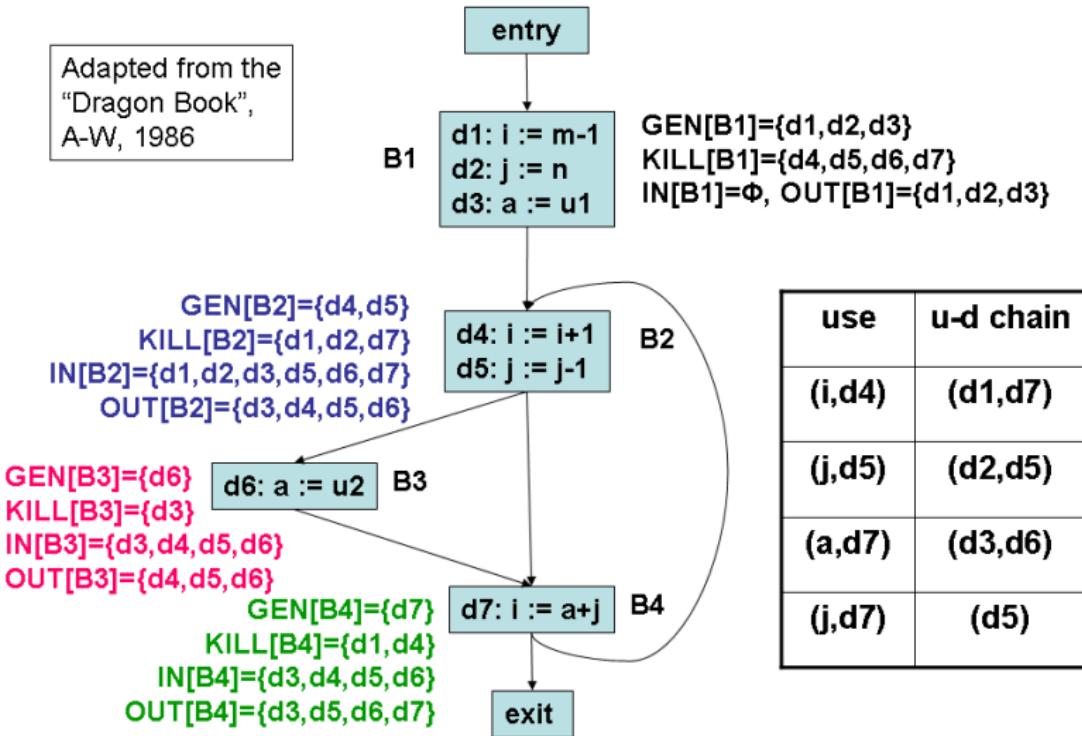
# Use-Definition Chain Construction



Three cases while constructing  
u-d chains from the reaching  
definitions

# Use-Definition Chain Example

Adapted from the  
"Dragon Book",  
A-W, 1986



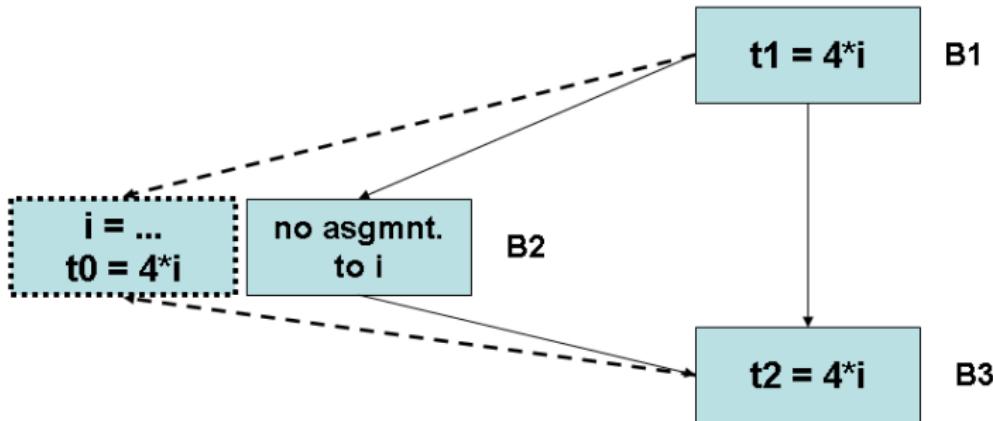
use	u-d chain
(i,d4)	(d1,d7)
(j,d5)	(d2,d5)
(a,d7)	(d3,d6)
(j,d7)	(d5)

# Available Expression Computation

- Sets of expressions constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is  $\cap$
- An expression  $x + y$  is *available* at a point  $p$ , if every path (not necessarily cycle-free) from the initial node to  $p$  evaluates  $x + y$ , and after the last such evaluation, prior to reaching  $p$ , there are no subsequent assignments to  $x$  or  $y$
- A block *kills*  $x + y$ , if it assigns (or may assign) to  $x$  or  $y$  and does not subsequently recompute  $x + y$ .
- A block *generates*  $x + y$ , if it definitely evaluates  $x + y$ , and does not subsequently redefine  $x$  or  $y$

## Available Expression Computation(2)

- Useful for global common sub-expression elimination
- $4 * i$  is a CSE in  $B_3$ , if it is available at the entry point of  $B_3$  i.e., if  $i$  is not assigned a new value in  $B_2$  or  $4 * i$  is recomputed after  $i$  is assigned a new value in  $B_2$  (as shown in the dotted box)



# Available Expression Computation (3)

- The data-flow equations

$$IN[B] = \bigcap_{\substack{P \text{ is a predecessor of } B}} OUT[P], \text{ } B \text{ not initial}$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$

$$IN[B1] = \phi$$

$$IN[B] = U, \text{ for all } B \neq B1 \text{ (initialization only)}$$

- $B1$  is the initial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$  is always  $\phi$
- $U$  is the universal set of all expressions
- Initializing  $IN[B]$  to  $\phi$  for all  $B \neq B1$ , is restrictive

# Computing e\_gen and e\_kill

- For statements of the form  $x = a$ , step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

$e_{\text{gen}}[q] = A$      $\mathbf{q} \bullet$   
 $x = y + z$

$\mathbf{p} \bullet$

## Computing $e_{\text{gen}}[p]$

- $A = A \mathbf{U} \{y+z\}$
- $A = A - \{\text{all expressions involving } x\}$
- $e_{\text{gen}}[p] = A$

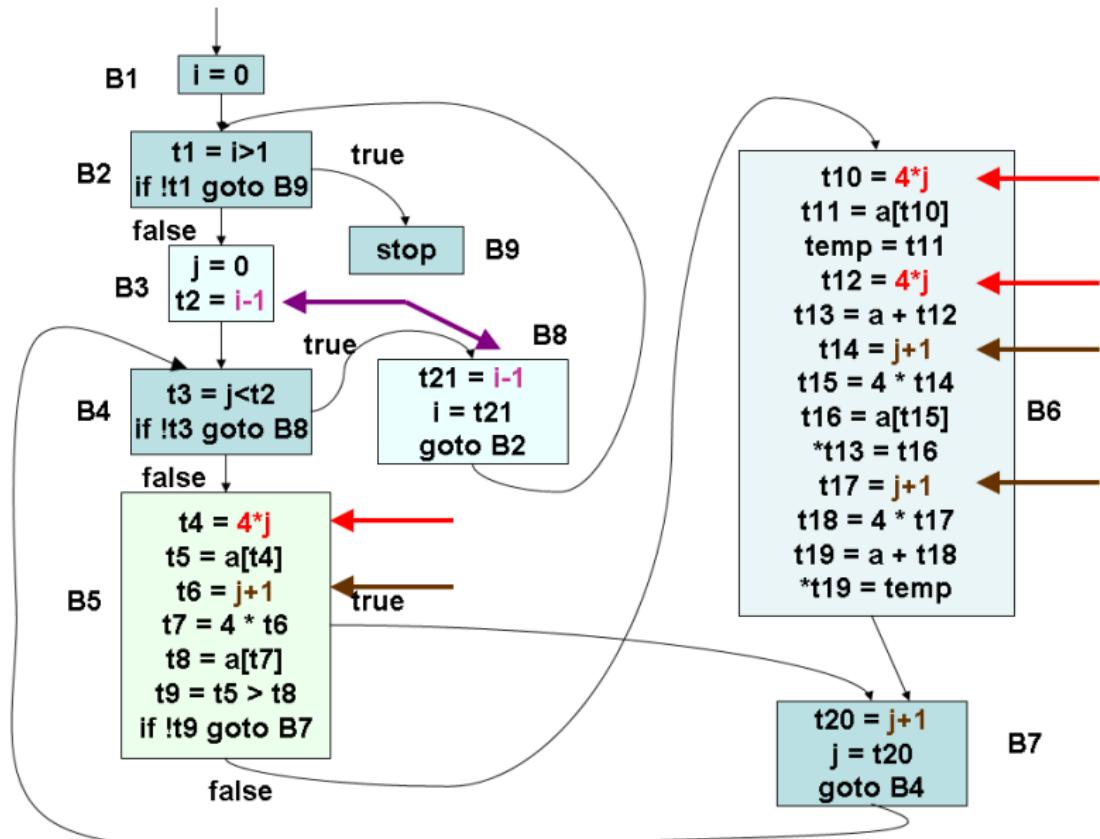
$e_{\text{kill}}[q] = A$      $\mathbf{q} \bullet$   
 $x = y + z$

$\mathbf{p} \bullet$

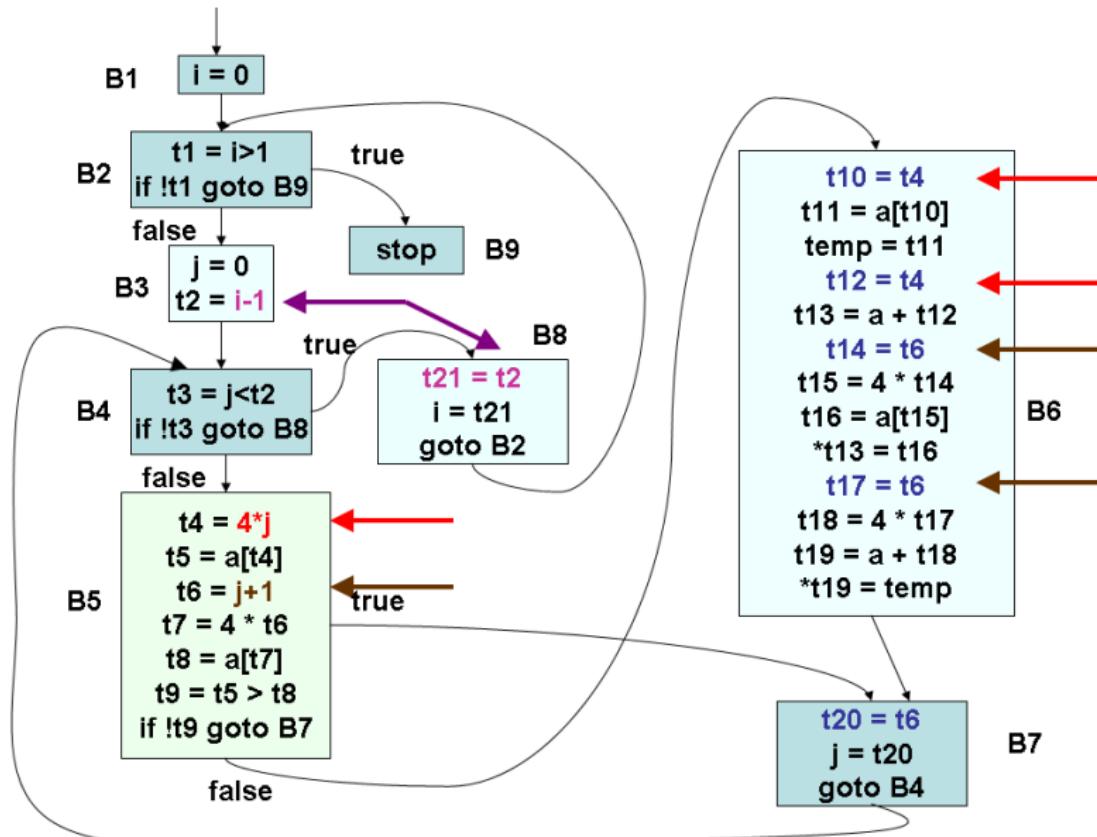
## Computing $e_{\text{kill}}[p]$

- $A = A - \{y+z\}$
- $A = A \mathbf{U} \{\text{all expressions involving } x\}$
- $e_{\text{kill}}[p] = A$

# Available Expression Computation - An Example



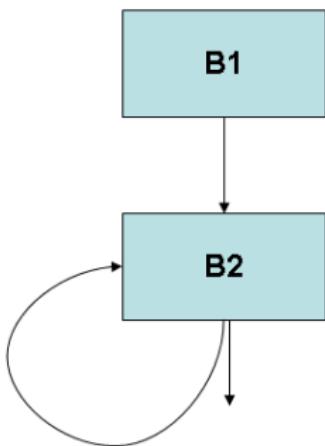
# Available Expression Computation - An Example (2)



# An Iterative Algorithm for Computing Available Expressions

```
for each block  $B \neq B_1$  do { $OUT[B] = U - e\_kill[B];$  }  
/* You could also do  $IN[B] = U;$  */  
/* In such a case, you must also interchange the order of */  
/*  $IN[B]$  and  $OUT[B]$  equations below */  
change = true;  
while change do { change = false;  
    for each block  $B \neq B_1$  do {  
  
 $IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P];$   
 $oldout = OUT[B];$   
 $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B]);$   
    if ( $OUT[B] \neq oldout$ ) change = true;  
}
```

# Initializing $\text{IN}[B]$ to $\phi$ for all $B$ can be restrictive



Let  $e_{\text{gen}}[B2]$  be  $G$  and  $e_{\text{kill}}[B2]$  be  $K$

$$\text{IN}[B2] = \text{OUT}[B1] \cap \text{OUT}[B2]$$

$$\text{OUT}[B2] = G \cup (\text{IN}[B2] - K)$$

$$\text{IN}^0[B2] = \Phi, \text{OUT}^1[B2] = G$$

$$\text{IN}^1[B2] = \text{OUT}[B1] \cap G$$

$$\begin{aligned}\text{OUT}^2[B2] &= G \cup ((\text{OUT}[B1] \cap G) - K) \\ &= G \cup G = G\end{aligned}$$

Note that  $(\text{OUT}[B1] \cap G)$  is always smaller than  $G$

---

$$\text{IN}^0[B2] = \mathbf{U}, \text{OUT}^1[B2] = \mathbf{U} - K$$

$$\begin{aligned}\text{IN}^1[B2] &= \text{OUT}[B1] \cap (\mathbf{U} - K) \\ &= \text{OUT}[B1] - K\end{aligned}$$

$$\begin{aligned}\text{OUT}^2[B2] &= G \cup ((\text{OUT}[B1] - K) - K) \\ &= G \cup (\text{OUT}[B1] - K)\end{aligned}$$

This set  $\text{OUT}[B2]$  is larger and more intuitive, but still correct

# Live Variable Analysis

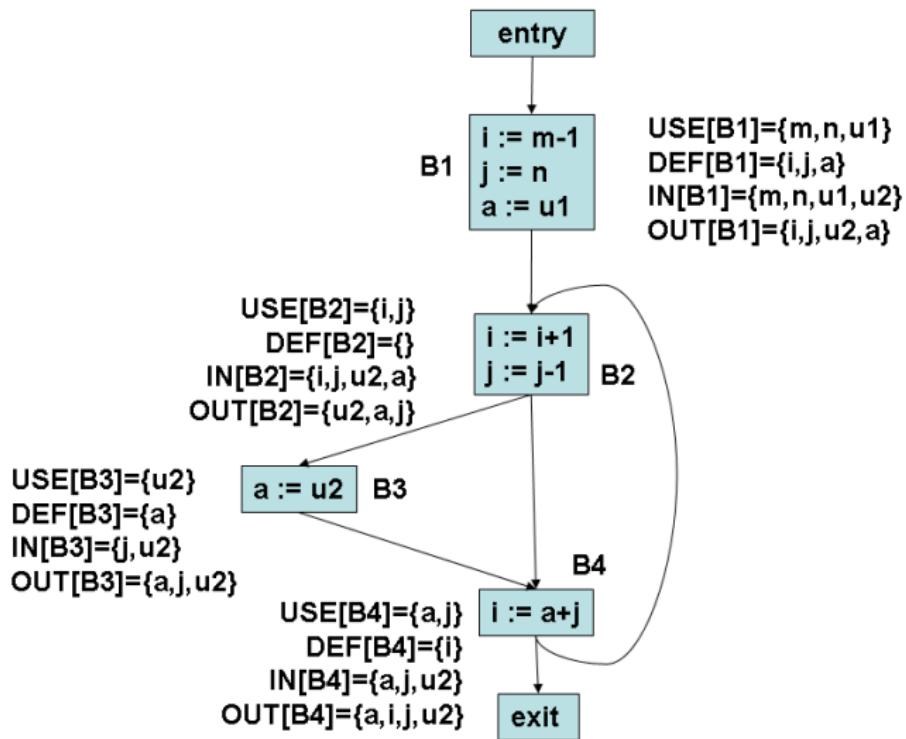
- The variable  $x$  is *live* at the point  $p$ , if the value of  $x$  at  $p$  could be used along some path in the flow graph, starting at  $p$ ; otherwise,  $x$  is *dead* at  $p$
- Sets of variables constitute the domain of data-flow values
- Backward flow problem, with confluence operator  $\bigcup$
- $IN[B]$  is the set of variables live at the beginning of  $B$
- $OUT[B]$  is the set of variables live just after  $B$
- $DEF[B]$  is the set of variables definitely assigned values in  $B$ , prior to any use of that variable in  $B$
- $USE[B]$  is the set of variables whose values may be used in  $B$  prior to any definition of the variable

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

# Live Variable Analysis: An Example



## Definition-Use Chains (d-u chains)

- For each definition, we wish to attach the statement numbers of the uses of that definition
- Such information is very useful in implementing register allocation, loop invariant code motion, etc.
- This problem can be transformed to the data-flow analysis problem of computing for a point  $p$ , the set of uses of a variable (say  $x$ ), such that there is a path from  $p$  to the use of  $x$ , that does not redefine  $x$ .
- This information is represented as sets of  $(x, s)$  pairs, where  $x$  is the variable used in statement  $s$
- In live variable analysis, we need information on whether a variable is used later, but in  $(x, s)$  computation, we also need the statement numbers of the uses
- The data-flow equations are similar to that of LV analysis
- Once  $IN[B]$  and  $OUT[B]$  are computed, d-u chains can be computed using a method similar to that of u-d chains

# Data-flow Analysis for (x,s) pairs

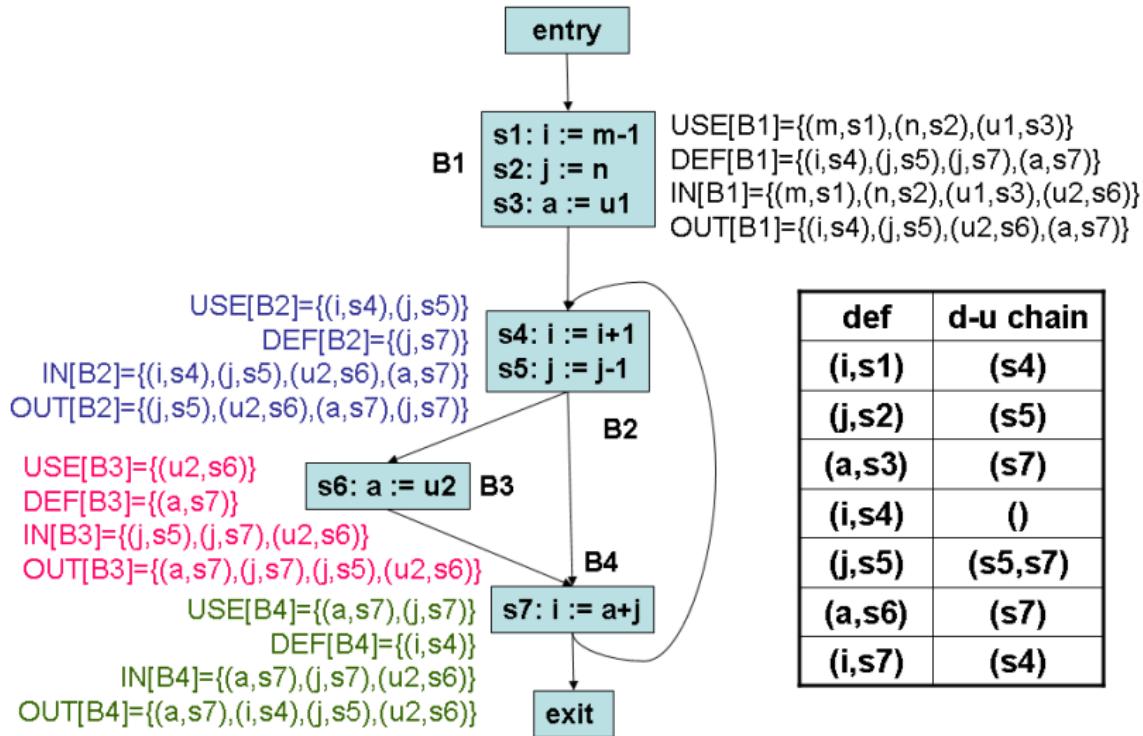
- Sets of pairs  $(x,s)$  constitute the domain of data-flow values
- Backward flow problem, with confluence operator  $\bigcup$
- $USE[B]$  is the set of pairs  $(x, s)$ , such that  $s$  is a statement in  $B$  which uses variable  $x$  and such that no prior definition of  $x$  occurs in  $B$
- $DEF[B]$  is the set of pairs  $(x, s)$ , such that  $s$  is a statement which uses  $x$ ,  $s$  is *not in*  $B$ , and  $B$  contains a definition of  $x$
- $IN[B]$  ( $OUT[B]$ , resp.) is the set of pairs  $(x, s)$ , such that statement  $s$  uses variable  $x$  and the value of  $x$  at  $IN[B]$  ( $OUT[B]$ , resp.) has not been modified along the path from  $IN[B]$  ( $OUT[B]$ , resp.) to  $s$

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

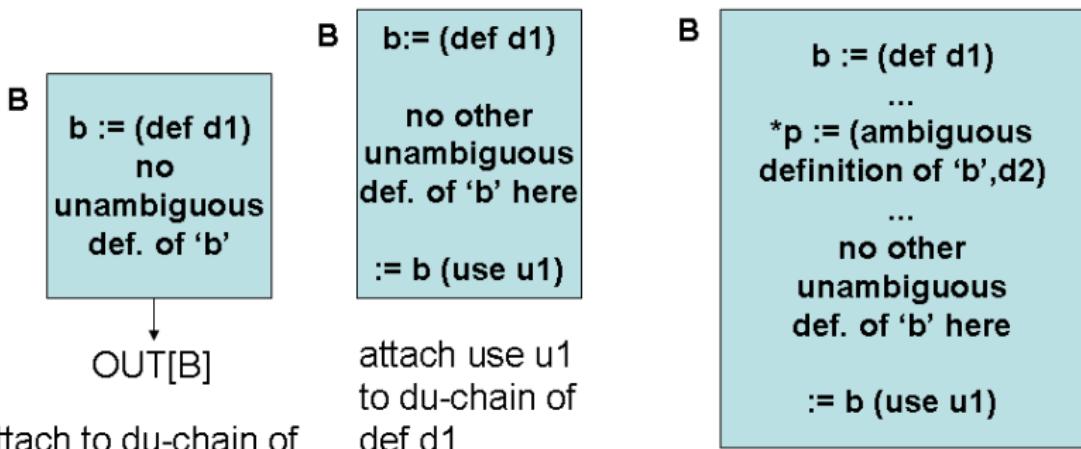
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

# Definition-Use Chain Example



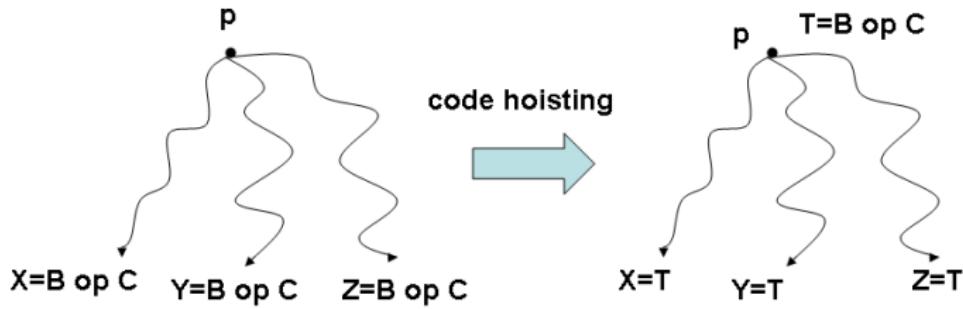
# Definition-Use Chain Construction



Three cases while constructing  
d-u chains from the  $(x, s)$  pairs

# Very Busy Expressions or Anticipated Expressions

- An expression  $B \text{ op } C$  is very busy or anticipated at a point  $p$ , if along every path from  $p$ , we come to a computation of  $B \text{ op } C$  before any computation of  $B$  or  $C$
- Useful in code hoisting and partial redundancy elimination
- Code hoisting does not reduce time, but reduces space
- We must make sure that no use of  $B \text{ op } C$  (from X, Y, or Z below) has any definition of  $B$  or  $C$  reaching it without passing through  $p$



## Very Busy Expressions or Anticipated Expressions (2)

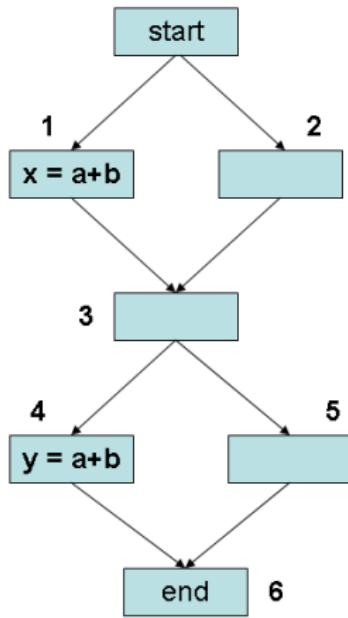
- Sets of expressions constitute the domain of data-flow values
- Backward flow analysis with  $\cap$  as confluence operator
- $V\_USE[n]$  is the set of expressions  $B \text{ op } C$  computed in  $n$  with no prior definition of  $B$  or  $C$  in  $n$
- $V\_DEF[n]$  is the set of expressions  $B \text{ op } C$  in  $U$  (the universal set of expressions) for which either  $B$  or  $C$  is defined in  $n$ , prior to any computation of  $B \text{ op } C$

$$OUT[n] = \bigcap_{\substack{S \text{ is a successor of } n}} IN[S]$$

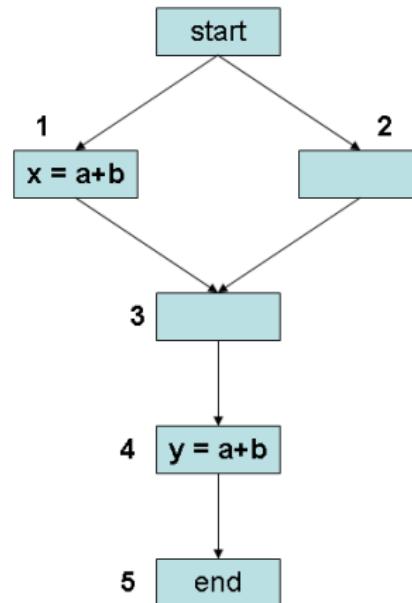
$$IN[n] = V\_USE[n] \bigcup (OUT[n] - V\_DEF[n])$$

$$IN[n] = U, \text{ for all } n \text{ (initialization only)}$$

# Anticipated Expressions - An Example



(a)



(b)

**a+b** is anticipated at: entry to 1 and 4  
**a+b** is not anticipated at: all other points

**a+b** is anticipated at all points,  
except at exit of 4 and entry of 5

## The Reaching Definitions Problem

- Domain of data-flow values: sets of definitions
- Direction: Forwards
- Confluence operator:  $\cup$
- Initialization:  $IN[B] = \phi$
- Equations:

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

# Data-Flow Problems: A Summary - 2

## The Available Expressions Problem

- Domain of data-flow values: sets of expressions
- Direction: Forwards
- Confluence operator:  $\cap$
- Initialization:  $IN[B] = U$
- Equations:

$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$

$$IN[B_1] = \phi$$

## The Live Variable Analysis Problem

- Domain of data-flow values: sets of variables
- Direction: backwards
- Confluence operator:  $\cup$
- Initialization:  $IN[B] = \phi$
- Equations:

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

## The Anticipated Expressions (Very Busy Expressions) Problem

- Domain of data-flow values: sets of expressions
- Direction: backwards
- Confluence operator:  $\cap$
- Initialization:  $IN[B] = U$
- Equations:

$$OUT[B] = \bigcap_{S \text{ is a successor of } B} IN[S]$$
$$IN[B] = V\_USE[B] \bigcup (OUT[B] - V\_DEF[B])$$