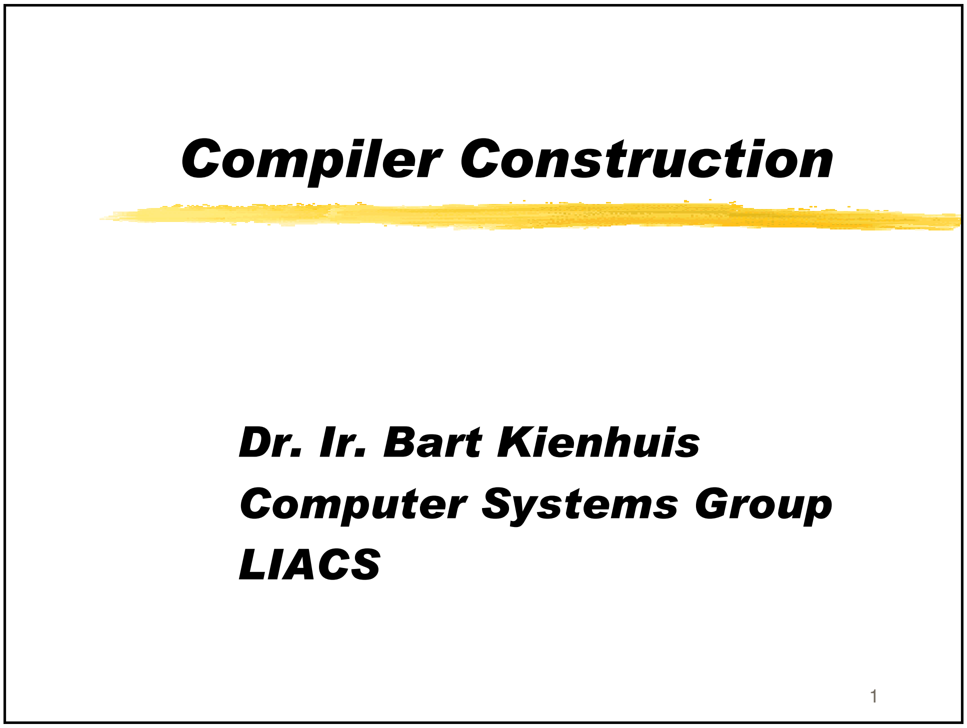
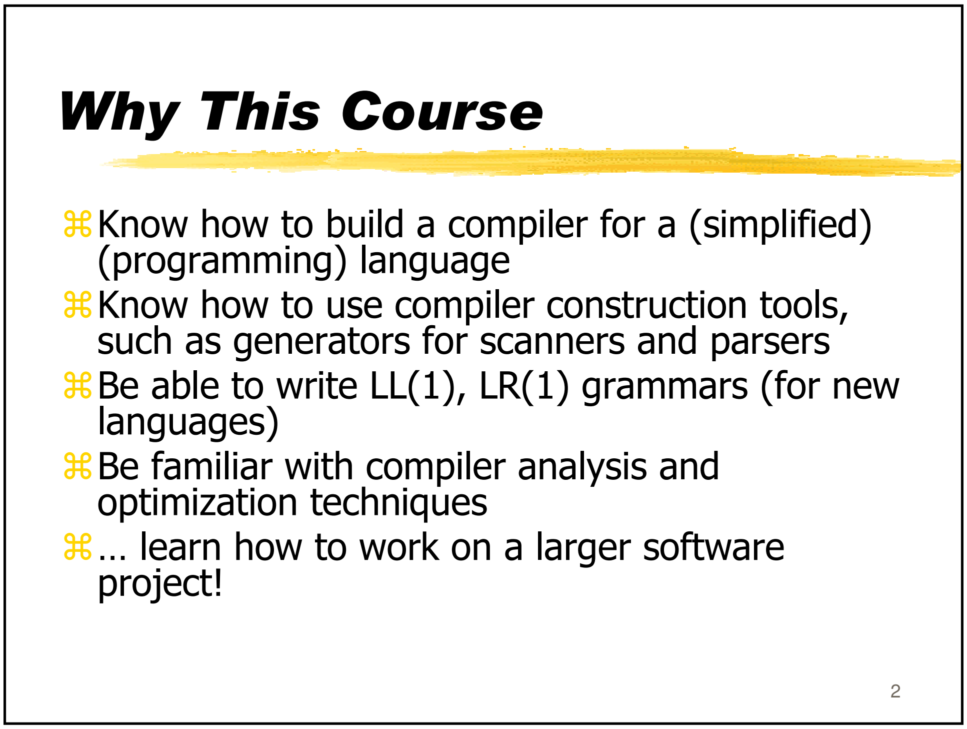


Compiler Construction

Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS



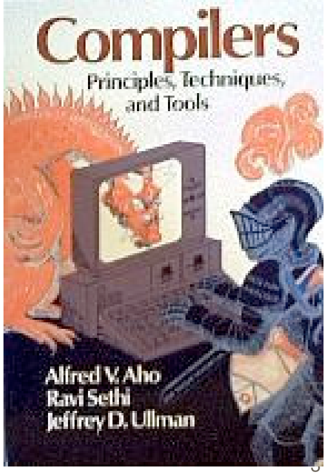
Why This Course

- Know how to build a compiler for a (simplified) (programming) language
- Know how to use compiler construction tools, such as generators for scanners and parsers
- Be able to write LL(1), LR(1) grammars (for new languages)
- Be familiar with compiler analysis and optimization techniques
- … learn how to work on a larger software project!

Course Outline

- In class, we discuss the theory using the ‘dragon’ book by Aho et al.
- In the practicum, the theory is applied when building a compiler that converts Pascal code to MIPS instructions.

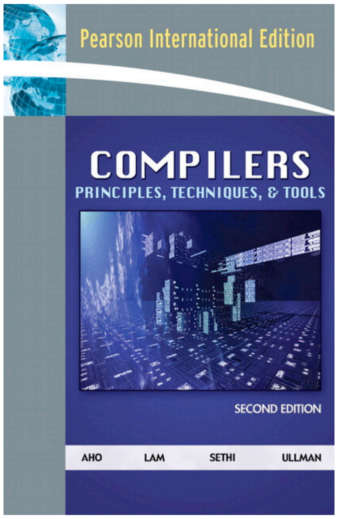
A.V. Aho, R. Sethi, en J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986, ISBN: 0-201-10088-6.

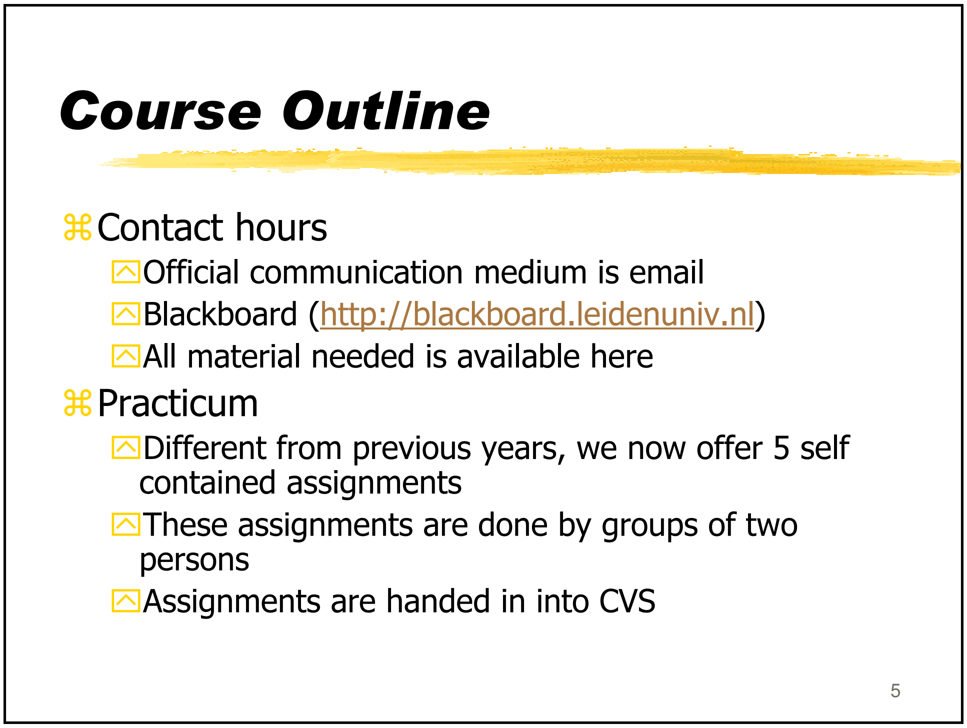


New edition

- Dragon book has been revised in 2006
- In Second edition good improvements are made

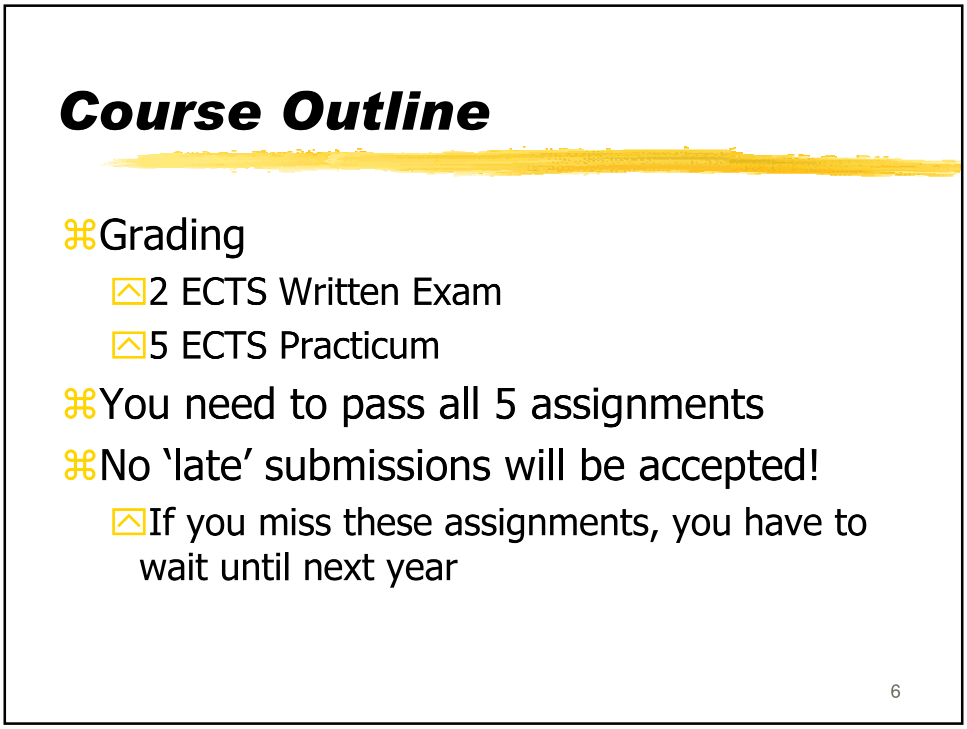
- **Publisher:** Addison Wesley; 2 edition (August 31, 2006)
- **Language:** English
- **ISBN-10:** 0321486811


Pearson International Edition
COMPILERS
PRINCIPLES, TECHNIQUES, & TOOLS
SECOND EDITION
AHO
LAM
SETHI
ULLMAN



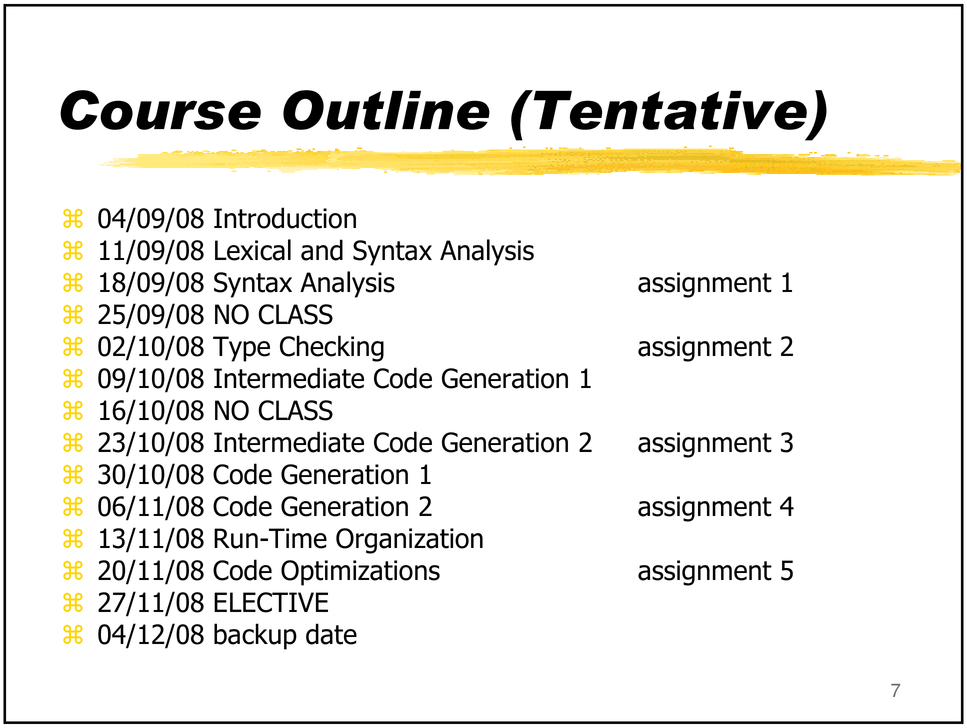
Course Outline

- Contact hours
  - Official communication medium is email
  - Blackboard (http://blackboard.leidenuniv.nl)
  - All material needed is available here
- Practicum
  - Different from previous years, we now offer 5 self contained assignments
  - These assignments are done by groups of two persons
  - Assignments are handed in into CVS



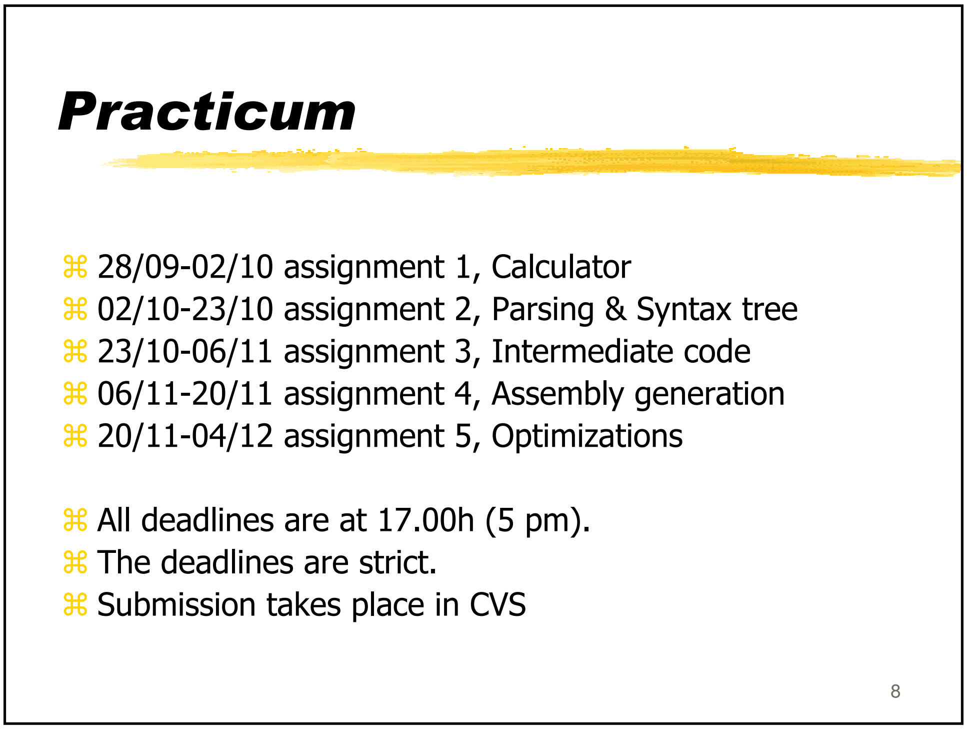
Course Outline

- Grading
  - 2 ECTS Written Exam
  - 5 ECTS Practicum
- You need to pass all 5 assignments
- No ‘late’ submissions will be accepted!
  - If you miss these assignments, you have to wait until next year



Course Outline (Tentative)

- 04/09/08 Introduction
- 11/09/08 Lexical and Syntax Analysis
- 18/09/08 Syntax Analysis assignment 1
- 25/09/08 NO CLASS
- 02/10/08 Type Checking assignment 2
- 09/10/08 Intermediate Code Generation 1
- 16/10/08 NO CLASS
- 23/10/08 Intermediate Code Generation 2 assignment 3
- 30/10/08 Code Generation 1
- 06/11/08 Code Generation 2 assignment 4
- 13/11/08 Run-Time Organization
- 20/11/08 Code Optimizations assignment 5
- 27/11/08 ELECTIVE
- 04/12/08 backup date



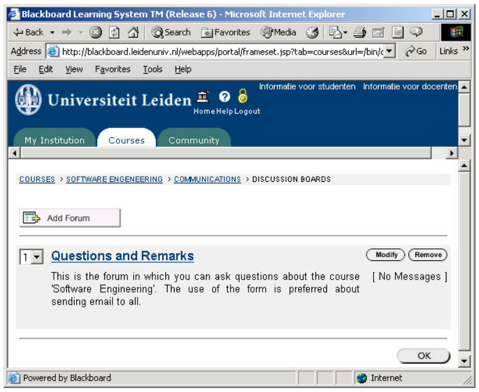
Practicum

- 28/09-02/10 assignment 1, Calculator
- 02/10-23/10 assignment 2, Parsing & Syntax tree
- 23/10-06/11 assignment 3, Intermediate code
- 06/11-20/11 assignment 4, Assembly generation
- 20/11-04/12 assignment 5, Optimizations

- All deadlines are at 17.00h (5 pm).
- The deadlines are strict.
- Submission takes place in CVS

Blackboard Coco

- Please enroll on-line to Coco in Blackboard
- Communication about Coco is shared between everyone.
- Use the ‘Forum’ option to ask me questions.
- If you ask me directly, I will submit also to the forum.


Blackboard Learning System TM (Release 6) - Microsoft Internet Explorer

Address http://blackboard.leidenuniv.nl/webapps/portal/frameset.jsp?tab=courses&url=/bin/c

COURSES > SOFTWARE ENGENEERING > COMMUNICATIONS > DISCUSSION BOARDS

1 Questions and Remarks

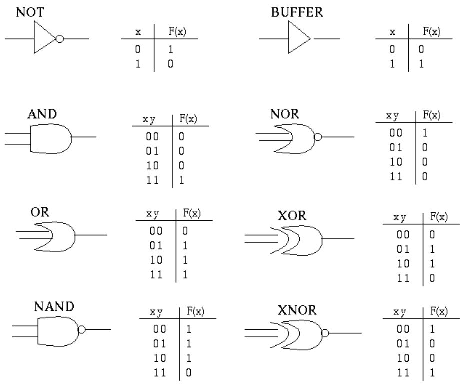
This is the forum in which you can ask questions about the course 'Software Engineering'. The use of the form is preferred about sending email to all.

[No Messages]

Introduction

- Compiler Construction
- Missing Link between
  - Digital Technique
    - Boolean Logic
    - Flip-Flops
  - Computer Architectures
    - Memory
    - Instructions

Digital Technique



NOT

x	F(x)
0	1
1	0

BUFFER

x	F(x)
0	0
1	1

AND

x y	F(x)
0 0	0
0 1	0
1 0	0
1 1	1

NOR

x y	F(x)
0 0	1
0 1	0
1 0	0
1 1	0

OR

x y	F(x)
0 0	0
0 1	1
1 0	1
1 1	1

XOR

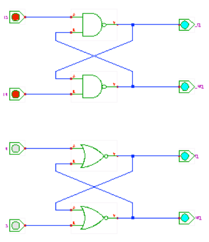
x y	F(x)
0 0	0
0 1	1
1 0	1
1 1	0

NAND

x y	F(x)
0 0	1
0 1	1
1 0	1
1 1	0

XNOR

x y	F(x)
0 0	1
0 1	0
1 0	0
1 1	1



/S	/R	Q	NQ	NAND
0	0	1	1	(forbidden)
0	1	1	0	
1	0	0	1	
1	1	Q*	NQ*	(store)

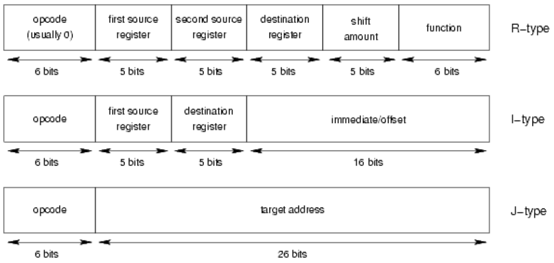
S	R	Q	NQ	NOR
0	0	Q*	NQ*	(store)
0	1	0	1	
1	0	1	0	
1	1	0	0	(forbidden)

Boolean Logic

Flip Flops

11

Computer Architecture

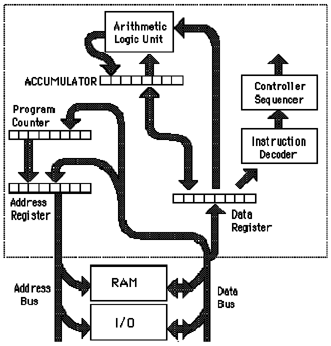


opcode (usually 0)	first source register	second source register	destination register	shift amount	function	R–type
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

opcode	first source register	destination register	immediate/offset	I–type
6 bits	5 bits	5 bits	16 bits	

opcode	target address	J–type
6 bits	26 bits	

MIPS Instruction Set

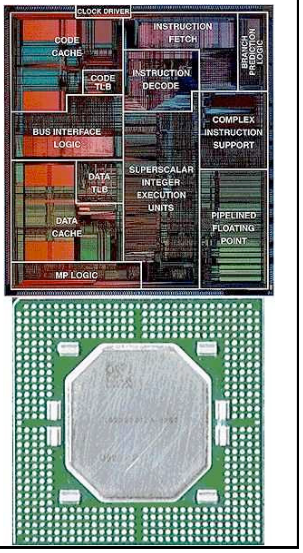


Computer Architecture

12

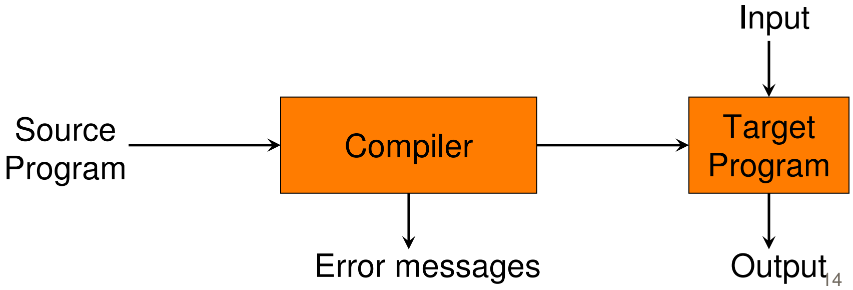
MIPS Instruction Set

opcode field	opcode	instruction format
000010	**j**	J-type
000011	**jal**	J-type
000100	**beq**	I-type
000101	**bne**	I-type
001000	**Addi**	I-type
001001	**Addiu**	I-type
001010	**Slti**	I-type
001011	**Sltiu**	I-type
001100	**Andi**	I-type



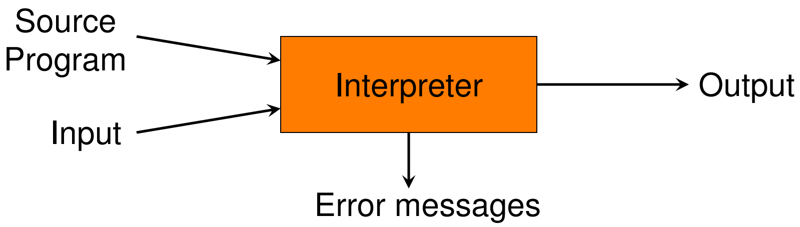
Compilers and Interpreters

- "*Compilation*"
  - Translation of a program written in a source language into a semantically equivalent program written in a target language



Compilers and Interpreters (cont'd)

- "*Interpretation*"
  - Performing the operations implied by the source program



15

The Analysis-Synthesis Model of Compilation

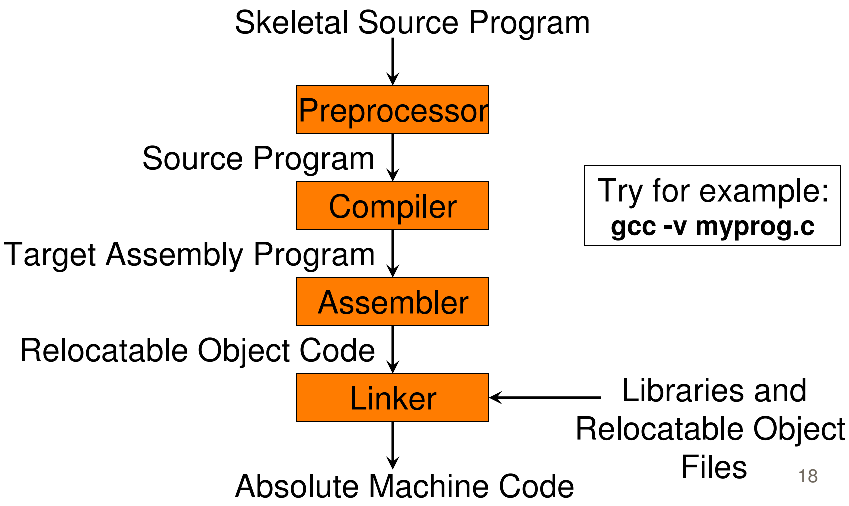
- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

16

Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. doxygen)
- *Static checkers* (e.g. lint and splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Preprocessors, Compilers, Assemblers, and Linkers



Skeletal Source Program

Preprocessor

Source Program

Compiler

Target Assembly Program

Assembler

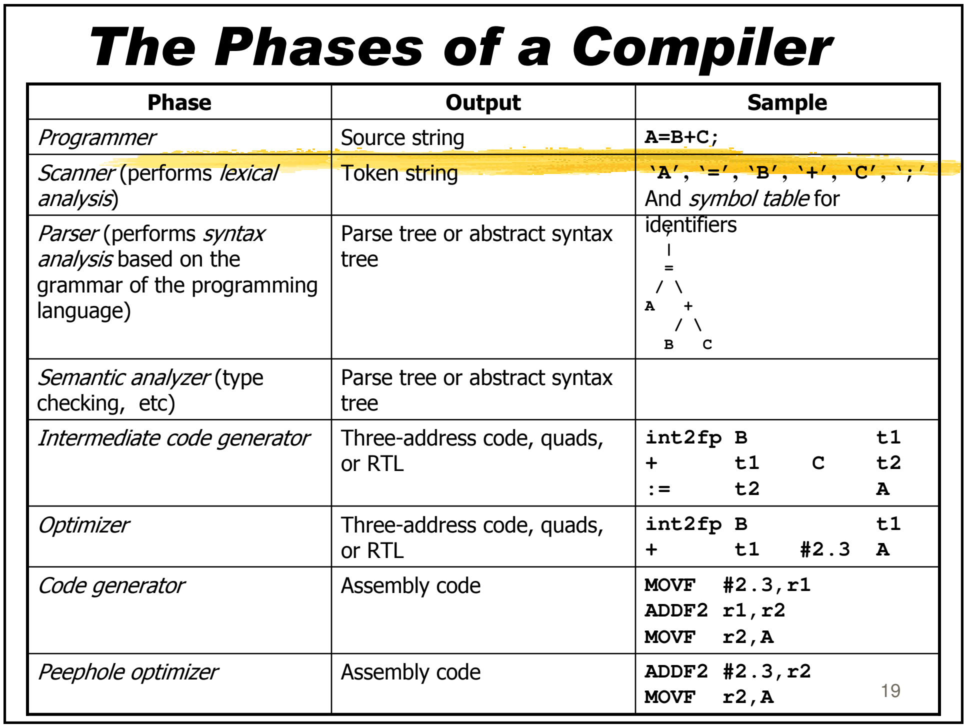
Relocatable Object Code

Linker

Libraries and Relocatable Object Files

Absolute Machine Code

Try for example:
gcc -v myprog.c



The Phases of a Compiler

Phase	Output	Sample
Programmer	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* for identifiers

Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	`|`
		`=`
		`/ \`
		`A   +`
		`/ \`
		`B   C`

Semantic analyzer (type checking, etc)	Parse tree or abstract syntax tree	

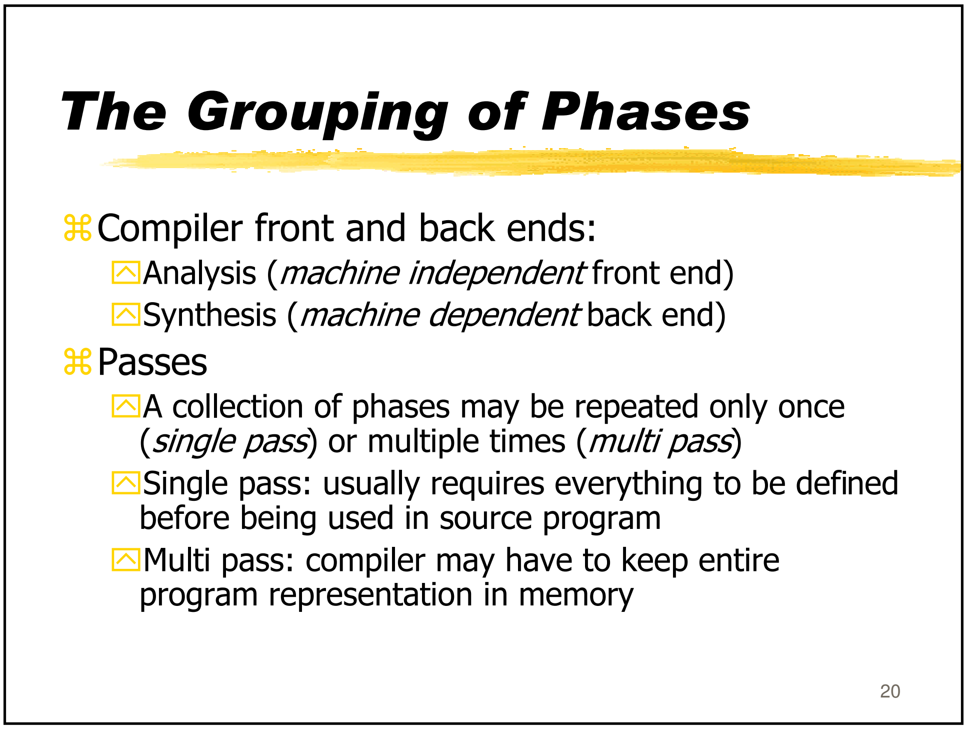
Intermediate code generator	Three-address code, quads, or RTL	`int2fp B       t1`
		`+      t1   C  t2`
		`:=     t2      A`

Optimizer	Three-address code, quads, or RTL	`int2fp B       t1`
		`+      t1 #2.3 A`

Code generator	Assembly code	`MOVF  #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF  r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2`
		`MOVF  r2,A`

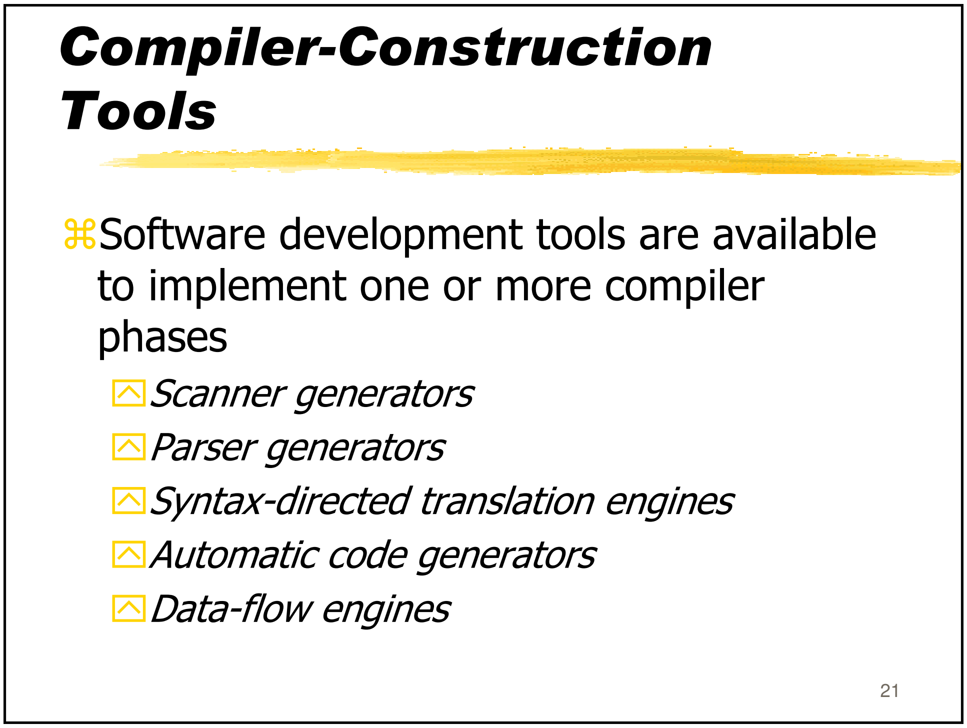
19



The Grouping of Phases

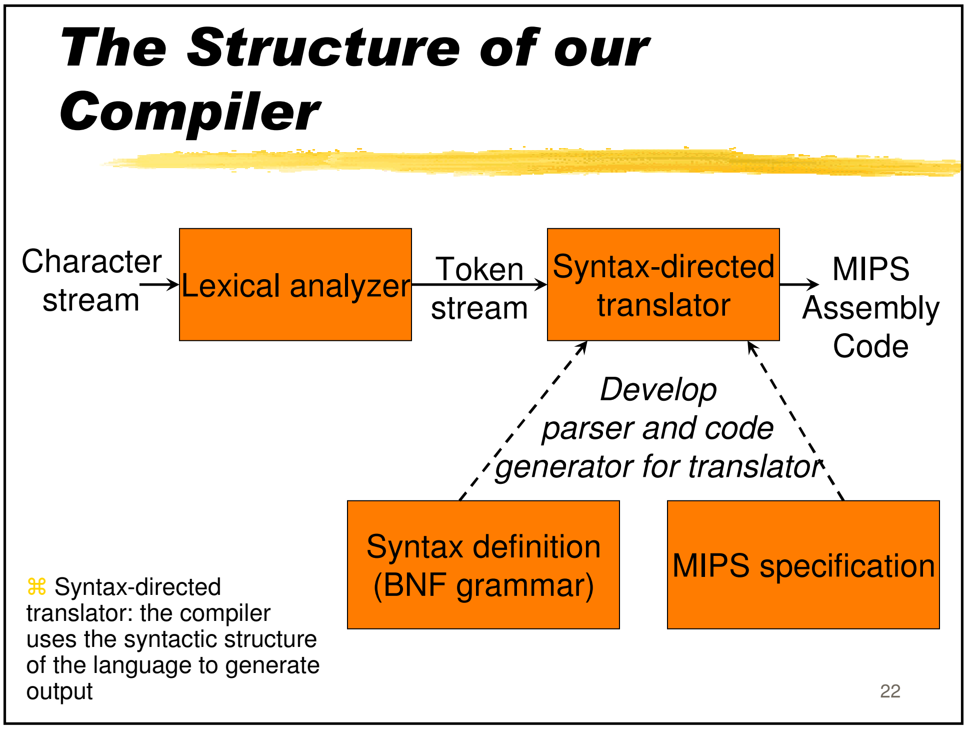
- Compiler front and back ends:
  - Analysis (*machine independent* front end)
  - Synthesis (*machine dependent* back end)
- Passes
  - A collection of phases may be repeated only once (*single pass*) or multiple times (*multi pass*)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory

20



Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - *Scanner generators*
  - *Parser generators*
  - *Syntax-directed translation engines*
  - *Automatic code generators*
  - *Data-flow engines*



The Structure of our Compiler

Character stream
Lexical analyzer
Token stream
Syntax-directed translator
MIPS Assembly Code
Develop parser and code generator for translator
Syntax definition (BNF grammar)
MIPS specification

- Syntax-directed translator: the compiler uses the syntactic structure of the language to generate output

Syntax Definition

- Context-free grammar is a 4-tuple with
  - A set of tokens (*terminal* symbols)
  - A set of *nonterminals*
  - A set of *productions*
  - A designated *start symbol*

Example Grammar

Context-free grammar for simple expressions:

$$G = <\{list, digit\}, \{\mathbf{+}, \mathbf{-}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9}\}, P, list>$$

with productions P =

$$list \rightarrow list \ \mathbf{+} \ digit$$

$$list \rightarrow list \ \mathbf{-} \ digit$$

$$list \rightarrow digit$$

$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Derivation

- Given a CF grammar we can determine the set of all *strings* (sequences of tokens) generated by the grammar using *derivation*
  - We begin with the start symbol
  - In each step, we replace one nonterminal in the current *sentential form* with one of the right-hand sides of a production for that nonterminal

Derivation for the Example Grammar

$$
\begin{aligned}
&\underline{\mathit{list}}\\
&\Rightarrow \underline{\mathit{list}} + \mathit{digit}\\
&\Rightarrow \underline{\mathit{list}} - \mathit{digit} + \mathit{digit}\\
&\Rightarrow \underline{\mathit{digit}} - \mathit{digit} + \mathit{digit}\\
&\Rightarrow \mathbf{9} - \underline{\mathit{digit}} + \mathit{digit}\\
&\Rightarrow \mathbf{9} - \mathbf{5} + \underline{\mathit{digit}}\\
&\Rightarrow \mathbf{9} - \mathbf{5} + \mathbf{2}
\end{aligned}
$$

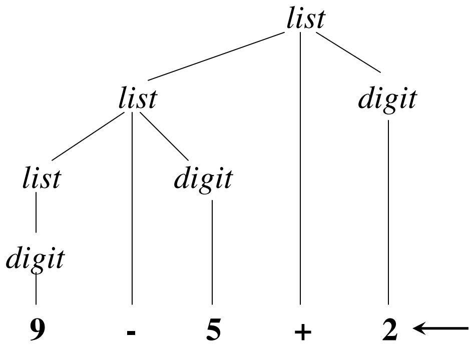
This is an example *leftmost derivation*, because we replaced the leftmost nonterminal (underlined) in each step

Parse Trees

- The root of the tree is labeled by the start symbol
- Each leaf of the tree is labeled by a terminal (=token) or ε (=empty)
- Each interior node is labeled by a nonterminal
- If $A \rightarrow X_1\, X_2 \ldots X_n$ is a production, then node A has children $X_1, X_2, \ldots, X_n$ where X_i is a (non)terminal or ε

Parse Tree for the Example Grammar

Parse tree of the string **9-5+2** using grammar *G*



The sequence of leafs is called the *yield* of the parse tree

Ambiguity

Consider the following context-free grammar:

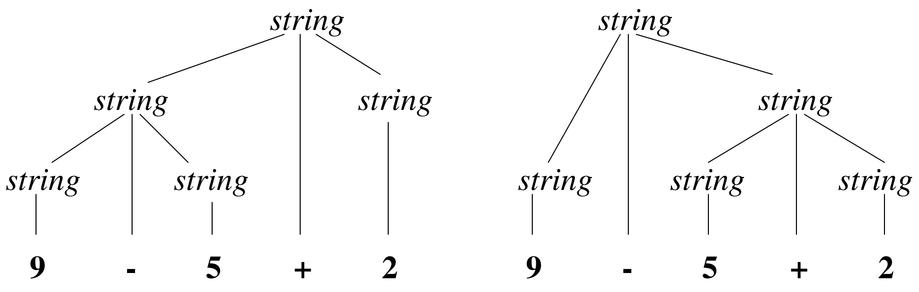
$$G = <\{\mathit{string}\}, \{\mathbf{+},\mathbf{-},\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7},\mathbf{8},\mathbf{9}\}, P, \mathit{string}>$$

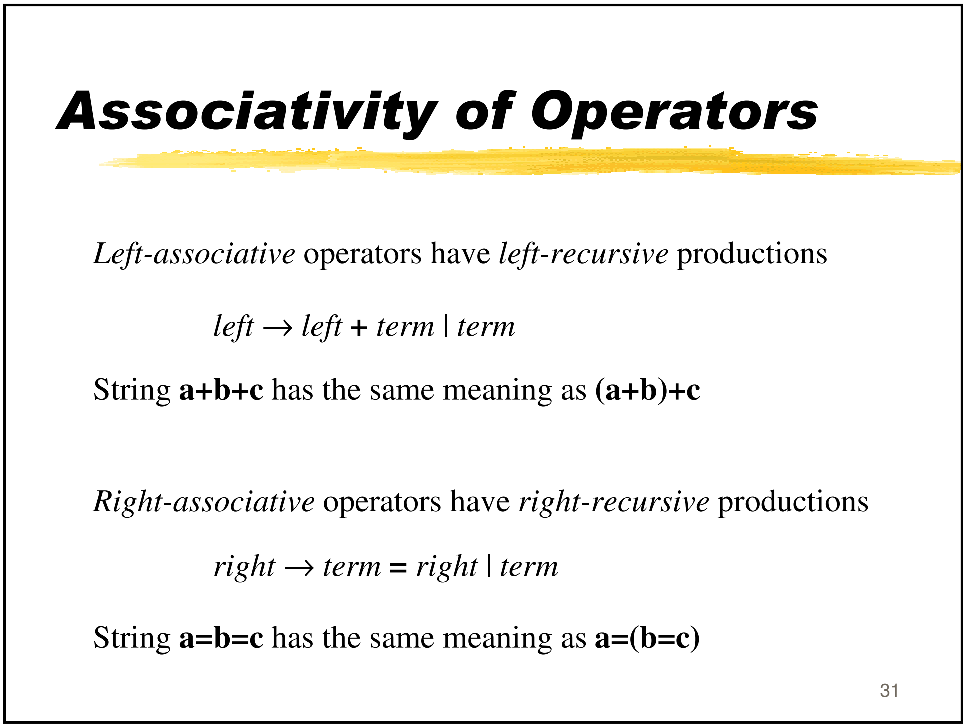
with production P =

$$\mathit{string} \rightarrow \mathit{string}\ \mathbf{+}\ \mathit{string} \mid \mathit{string}\ \mathbf{-}\ \mathit{string} \mid \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9}$$

This grammar is *ambiguous*, because more than one parse tree generates the string **9-5+2**

Ambiguity (cont'd)





Associativity of Operators

Left-associative operators have *left-recursive* productions

left → *left* **+** *term* | *term*

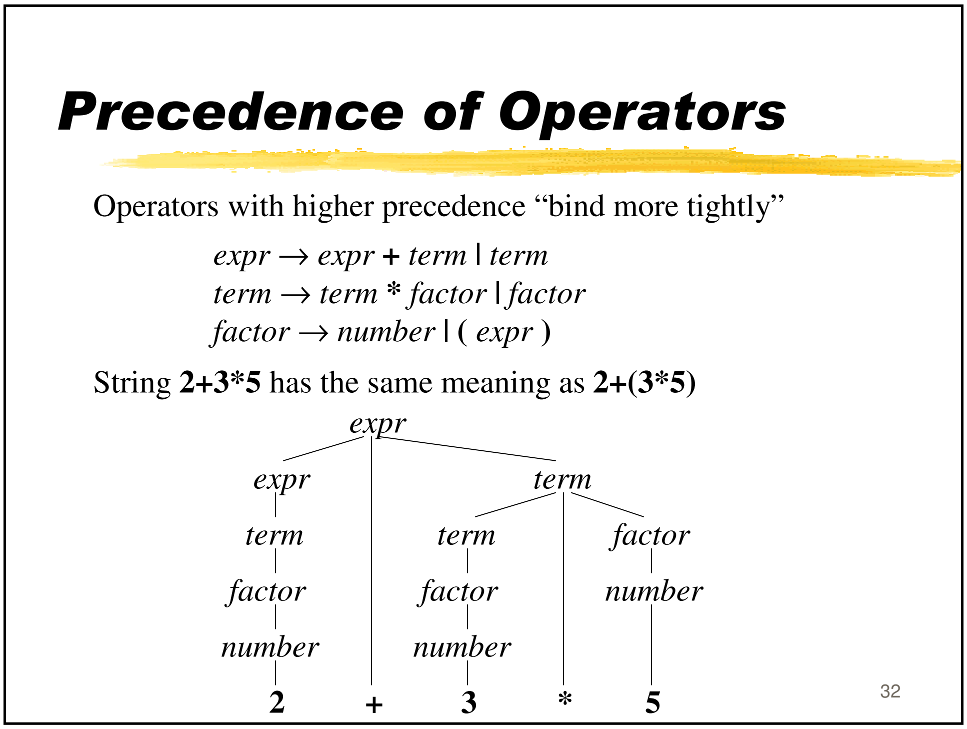
String **a+b+c** has the same meaning as **(a+b)+c**

Right-associative operators have *right-recursive* productions

right → *term* **=** *right* | *term*

String **a=b=c** has the same meaning as **a=(b=c)**

31



Precedence of Operators

Operators with higher precedence “bind more tightly”

expr → *expr* **+** *term* | *term*
term → *term* ***** *factor* | *factor*
factor → *number* | **(** *expr* **)**

String **2+3*5** has the same meaning as **2+(3*5)**

expr
expr
term
term
term
factor
factor
factor
number
number
number
2
+
3
*
5

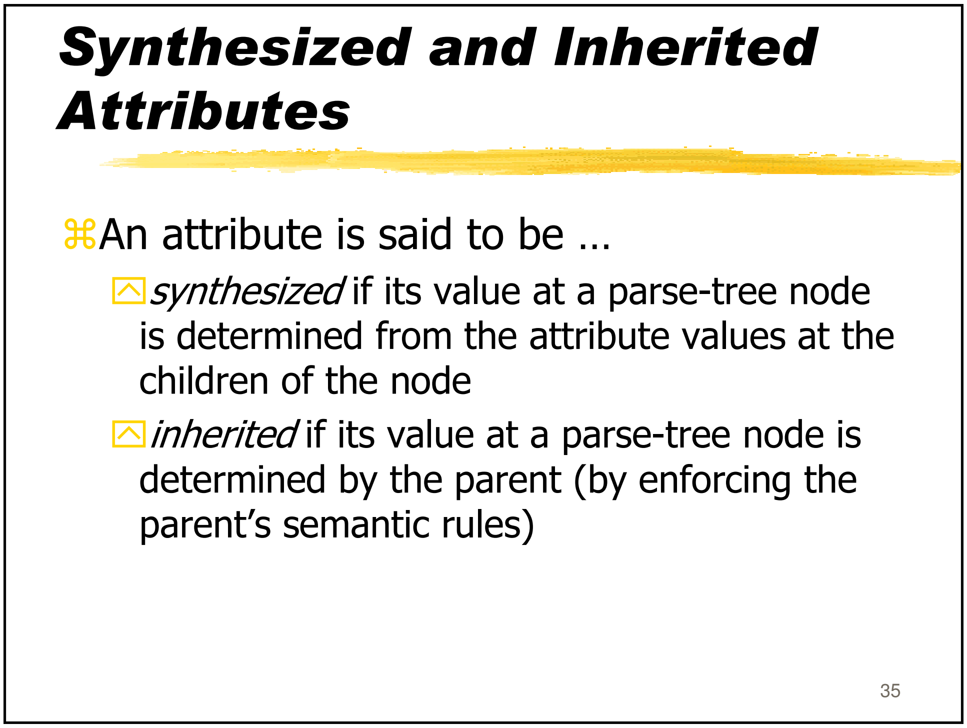
32

Syntax of Statements

```
stmt → id := expr
            | if expr then stmt
            | if expr then stmt else stmt
            | while expr do stmt
            | begin opt_stmts end
opt_stmts → stmt ; opt_stmts
            | ε
```

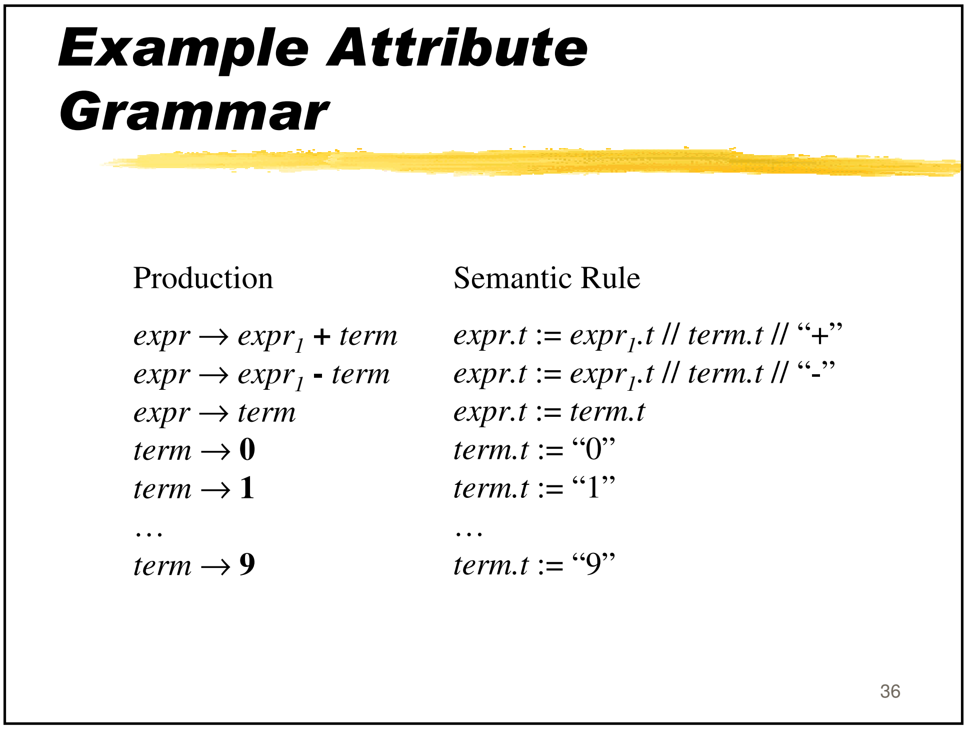
Syntax-Directed Translation

- ⌘Uses a CF grammar to specify the syntactic structure of the language
- ⌘AND associates a set of *attributes* with (non)terminals
- ⌘AND associates with each production a set of *semantic rules* for computing values for the attributes
- ⌘The attributes contain the translated form of the input after the computations are completed



Synthesized and Inherited Attributes

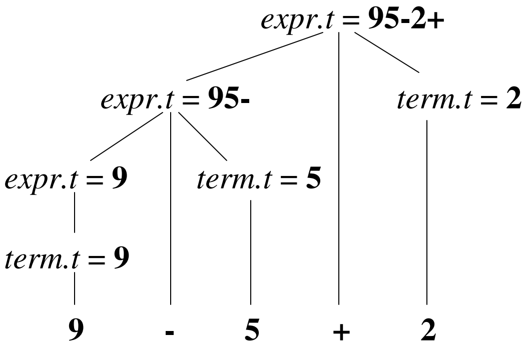
- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent’s semantic rules)



Example Attribute Grammar

Production	Semantic Rule
$expr \rightarrow expr_1$ **+** $term$	$expr.t := expr_1.t \,\|\|\, term.t \,\|\|$ “+”
$expr \rightarrow expr_1$ **-** $term$	$expr.t := expr_1.t \,\|\|\, term.t \,\|\|$ “-”
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow$ **0**	$term.t :=$ “0”
$term \rightarrow$ **1**	$term.t :=$ “1”
…	…
$term \rightarrow$ **9**	$term.t :=$ “9”

Example Annotated Parse Tree

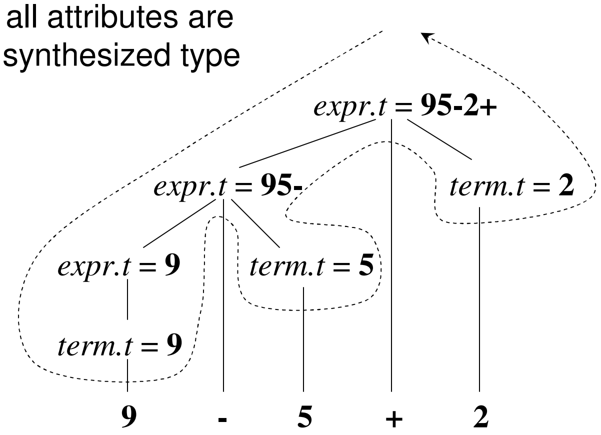


Depth-First Traversals

```
procedure visit(n : node);
begin
   for each child m of n, from left to right do
      visit(m);
   evaluate semantic rules at node n
end
```

Depth-First Traversals (Example)

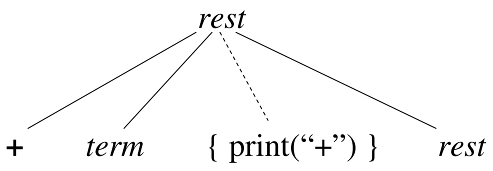
Note: all attributes are of the synthesized type



Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

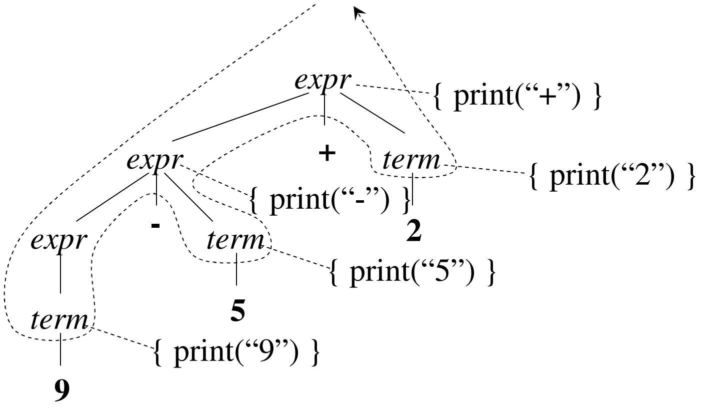
$$\mathit{rest} \rightarrow \mathbf{+}\ \mathit{term}\ \underbrace{\{\ \text{print(“+”)}\ \}}_{\text{Embedded semantic action}}\ \mathit{rest}$$



Example Translation Scheme

expr → *expr* **+** *term*	{ print(“+”) }
expr → *expr* **-** *term*	{ print(“-”) }
expr → *term*	
term → **0**	{ print(“0”) }
term → **1**	{ print(“1”) }
…	…
term → **9**	{ print(“9”) }

Example Translation Scheme (cont’d)



Translates **9-5+2** into postfix **95-2+**

Parsing

- Parsing = *process of determining if a string of tokens can be generated by a grammar*
- For any CF grammar there is a parser that takes at most $O(n^3)$ time to parse a string of *n* tokens
- Linear algorithms suffice for parsing programming language
- *Top-down parsing* "constructs" parse tree from root to leaves
- *Bottom-up parsing* "constructs" parse tree from leaves to root

Predictive Parsing

- *Recursive descent parsing* is a top-down parsing method
  - Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens
  - When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information
- *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations

Example Predictive Parser (Grammar)

```
  type → simple
       | ^ id
       | array [ simple ] of type
simple → integer
       | char
       | num dotdot num
```

45

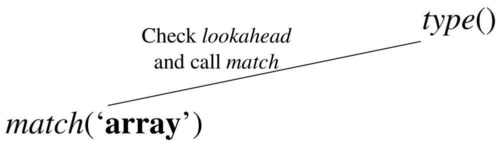
Example Predictive Parser (Program Code)

```
procedure match(t : token);
begin
   if lookahead = t then
      lookahead := nexttoken()
   else error()
end;

procedure type();
begin
   if lookahead in { ‘integer’, ‘char’, ‘num’ } then
      simple()
   else if lookahead = ‘^’ then
      match(‘^’); match(id)
   else if lookahead = ‘array’ then
      match(‘array’); match(‘[‘); simple();
      match(‘]’); match(‘of’); type()
   else error()
end;
```

```
procedure simple();
begin
   if lookahead = ‘integer’ then
      match(‘integer’)
   else if lookahead = ‘char’ then
      match(‘char’)
   else if lookahead = ‘num’ then
      match(‘num’);
      match(‘dotdot’);
      match(‘num’)
   else error()
end;
```

46

Example Predictive Parser (Execution Step 1)



type()

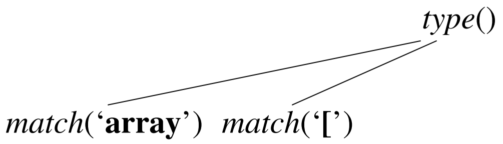
Check *lookahead* and call *match*

match(‘**array**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 2)

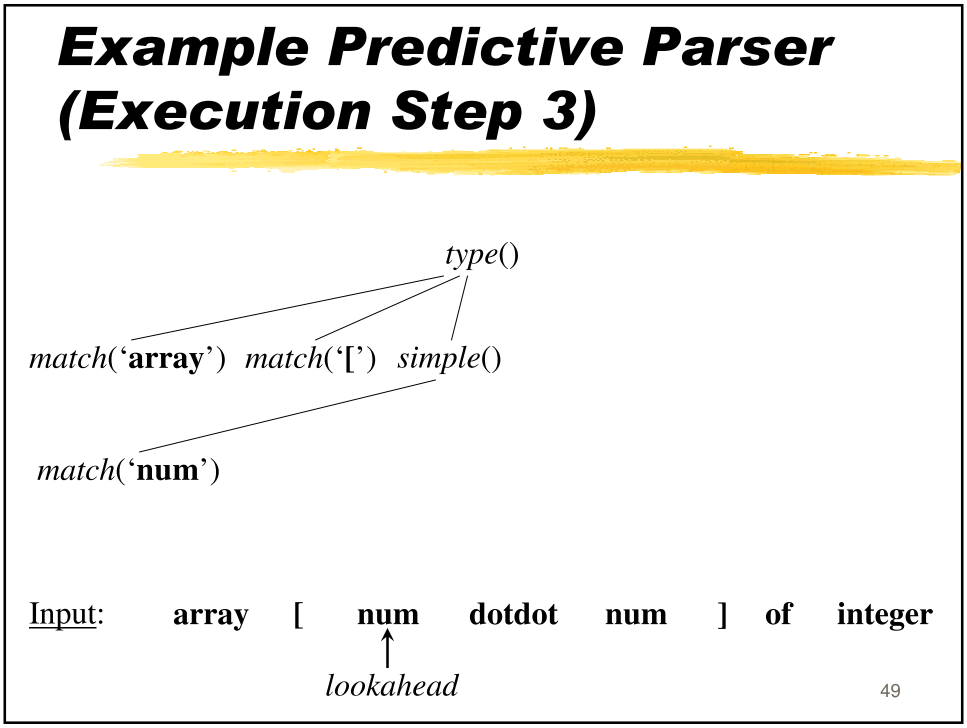


type()

match(‘**array**’) *match*(‘**[**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*



Example Predictive Parser (Execution Step 3)

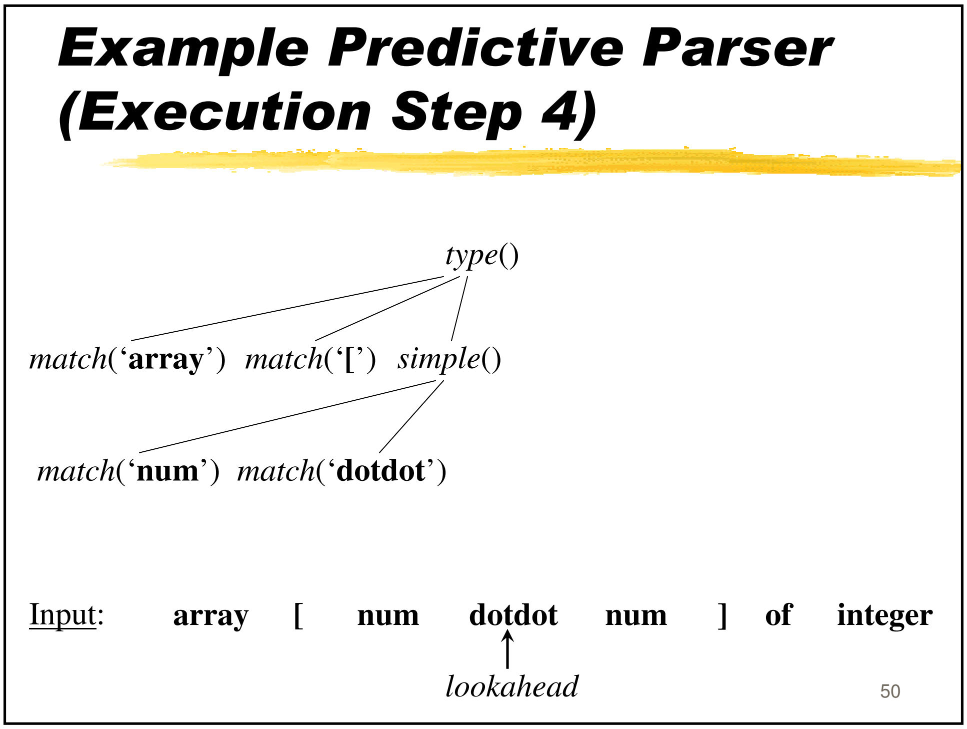
type()

match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*



Example Predictive Parser (Execution Step 4)

type()

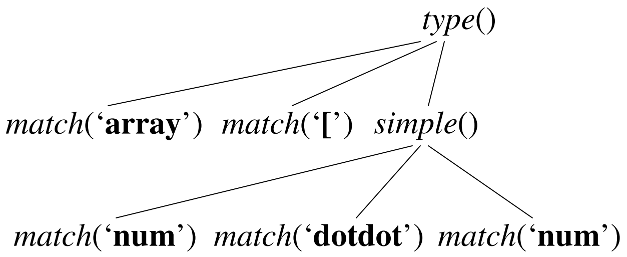
match(‘**array**’) *match*(‘**[**’) *simple*()

match(‘**num**’) *match*(‘**dotdot**’)

Input: **array [num dotdot num] of integer**

↑ *lookahead*

Example Predictive Parser (Execution Step 5)


type()

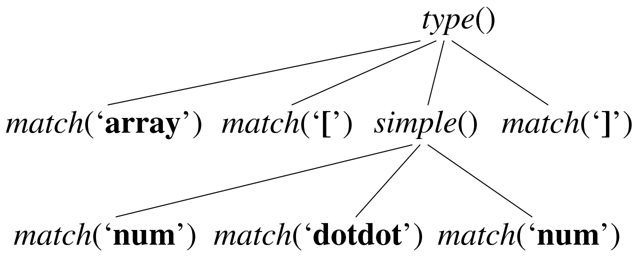
match('**array**') *match*('[') *simple*()

match('**num**') *match*('**dotdot**') *match*('**num**')

Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead* (at **num**)

Example Predictive Parser (Execution Step 6)

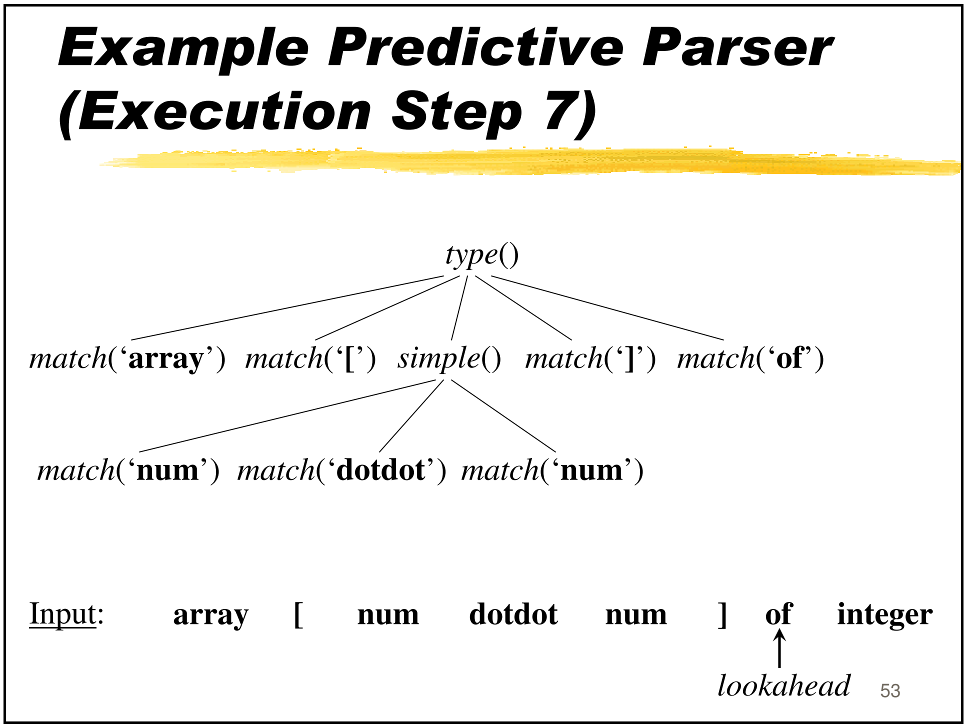

type()

match('**array**') *match*('[') *simple*() *match*(']')

match('**num**') *match*('**dotdot**') *match*('**num**')

Input: **array** **[** **num** **dotdot** **num** **]** **of** **integer**

↑ *lookahead* (at **]**)



Example Predictive Parser (Execution Step 7)

type()

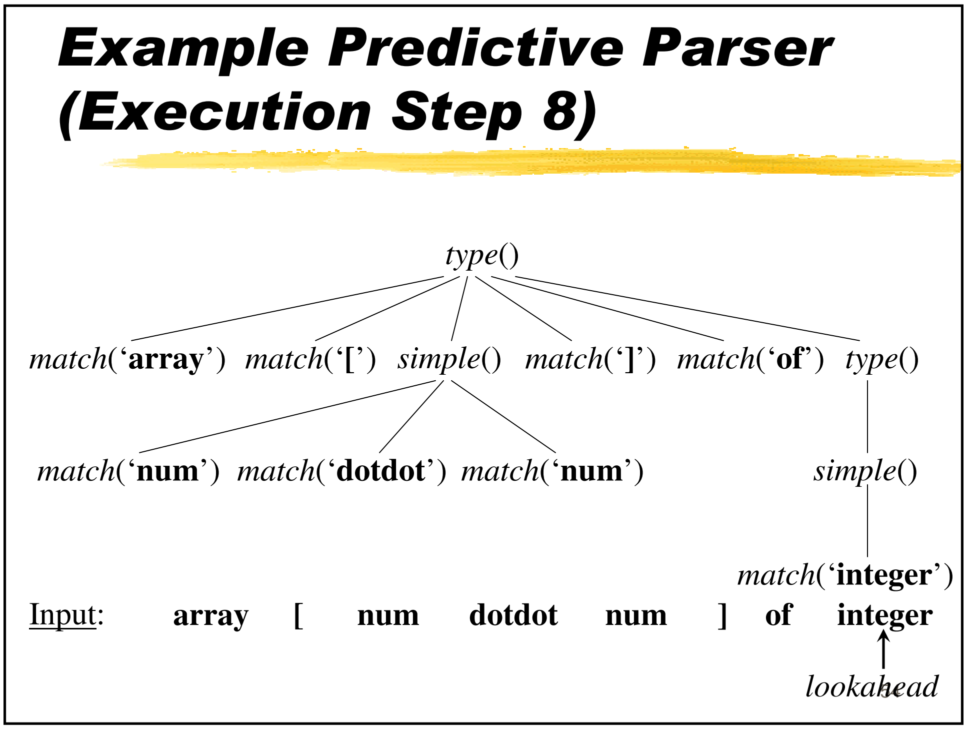
match('**array**') *match*('**[**') *simple*() *match*('**]**') *match*('**of**')

match('**num**') *match*('**dotdot**') *match*('**num**')

Input: **array [num dotdot num] of integer**

lookahead

53



Example Predictive Parser (Execution Step 8)

type()

match('**array**') *match*('**[**') *simple*() *match*('**]**') *match*('**of**') *type*()

match('**num**') *match*('**dotdot**') *match*('**num**') *simple*()

match('**integer**')

Input: **array [num dotdot num] of integer**

lookahead

FIRST

FIRST(α) is the set of terminals that appear as the first symbols of one or more strings generated from α

type → *simple*
  | **^ id**
  | **array [** *simple* **] of** *type*
simple → **integer**
  | **char**
  | **num dotdot num**

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(**^ id**) = { **^** }
FIRST(*type*) = { **integer**, **char**, **num**, **^**, **array** }

Using FIRST

We use FIRST to write a predictive parser as follows

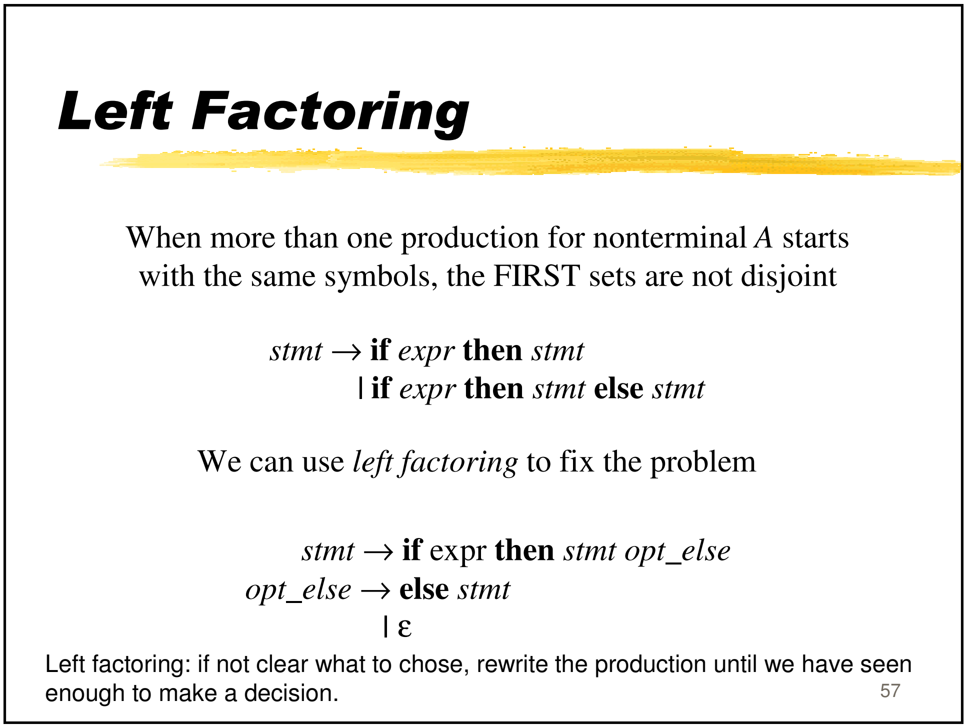
expr → *term rest*
rest → **+** *term rest*
  | **-** *term rest*
  | ε

```
procedure rest();
begin
   if lookahead in FIRST(+ term rest) then
      match('+'); term(); rest()
   else if lookahead in FIRST(- term rest) then
      match('-'); term(); rest()
   else return
end;
```

When a nonterminal *A* has two (or more) productions as in

$A \to \alpha$
  $| \; \beta$

Then FIRST (α) and FIRST(β) must be disjoint for predictive parsing to work



Left Factoring

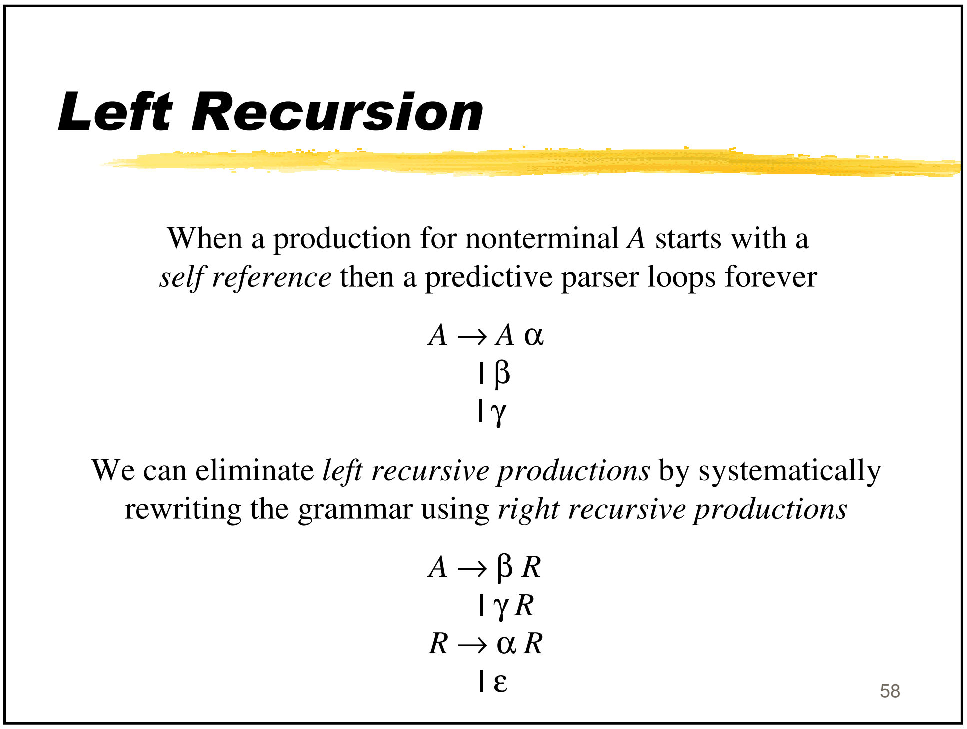
When more than one production for nonterminal *A* starts with the same symbols, the FIRST sets are not disjoint

$$\begin{aligned} \mathit{stmt} &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\ &\mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \end{aligned}$$

We can use *left factoring* to fix the problem

$$\begin{aligned} \mathit{stmt} &\to \textbf{if}\ \text{expr}\ \textbf{then}\ \mathit{stmt}\ \mathit{opt\_else} \\ \mathit{opt\_else} &\to \textbf{else}\ \mathit{stmt} \\ &\mid \varepsilon \end{aligned}$$

Left factoring: if not clear what to chose, rewrite the production until we have seen enough to make a decision.



Left Recursion

When a production for nonterminal *A* starts with a *self reference* then a predictive parser loops forever

$$\begin{aligned} A &\to A\ \alpha \\ &\mid \beta \\ &\mid \gamma \end{aligned}$$

We can eliminate *left recursive productions* by systematically rewriting the grammar using *right recursive productions*

$$\begin{aligned} A &\to \beta\ R \\ &\mid \gamma\ R \\ R &\to \alpha\ R \\ &\mid \varepsilon \end{aligned}$$

A Translator for Simple Expressions

expr → *expr* **+** *term*	{ print(“+”) }
expr → *expr* **-** *term*	{ print(“-”) }
expr → *term*	
term → **0**	{ print(“0”) }
term → **1**	{ print(“1”) }
…	…
term → **9**	{ print(“9”) }

After left recursion elimination:

expr → *term rest*
rest → **+** *term* { print(“+”) } *rest* | **-** *term* { print(“-”) } *rest* | ε
term → **0** { print(“0”) }
term → **1** { print(“1”) }
…
term → **9** { print(“9”) }

expr → *term rest*

rest → **+** *term* { print(“+”) } *rest*
| **-** *term* { print(“-”) } *rest*
| ε

term → **0** { print(“0”) }
term → **1** { print(“1”) }
…
term → **9** { print(“9”) }

```
main()
{   lookahead = getchar();
    expr();
}
expr()
{   term();
    while (1) /* optimized by inlining rest()
                 and removing recursive calls */
    {   if (lookahead == ‘+’)
        {   match(‘+’); term(); putchar(‘+’);
        }
        else if (lookahead == ‘-’)
        {   match(‘-’); term(); putchar(‘-’);
        }
        else break;
    }
}
term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{   printf(“Syntax error\n”);
    exit(1);
}
```

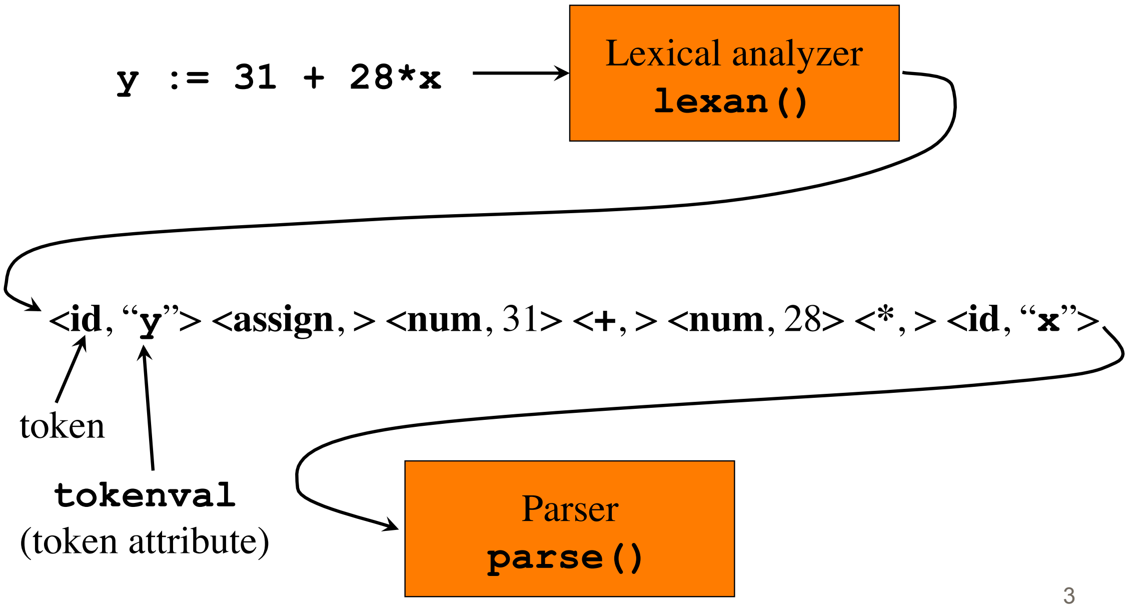
Lexical Analysis

Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS

Adding a Lexical Analyzer

- Typical tasks of the lexical analyzer:
  - Remove white space and comments
  - Encode constants as tokens
  - Recognize keywords
  - Recognize identifiers

The Lexical Analyzer



Token Attributes

factor → (*expr*)
| **num** { print(**num**.value) }

```
#define NUM 256 /* token returned by lexan */

factor()
{   if (lookahead == ‘(‘)
    {   match(‘(‘); expr(); match(‘)’);
    }
    else if (lookahead == NUM)
    {   printf(“ %d “, tokenval); match(NUM);
    }
    else error();
}
```

Symbol Table

The symbol table is globally accessible (to all phases of the compiler)

Each entry in the symbol table contains a string and a token value:

```
struct entry
{   char *lexptr; /* lexeme (string) */
    int token;
};
struct entry symtable[];
```

`insert(s, t)`: returns array index to new entry for string **`s`** token **`t`**

`lookup(s)`: returns array index to entry for string **`s`** or 0

Possible implementations:
- simple C code (see textbook)
- hashtables

Identifiers

factor → (*expr*)
| **id** { print(**id**.string) }

```
#define ID 259 /* token returned by lexan() */

factor()
{   if (lookahead == ‘(‘)
    {   match(‘(‘); expr(); match(‘)’);
    }
    else if (lookahead == ID)
    {   printf(“ %s “, symtable[tokenval].lexptr);
        match(NUM);
    }
    else error();
}
```

Handling Reserved Keywords

We simply initialize the global symbol table with the set of keywords

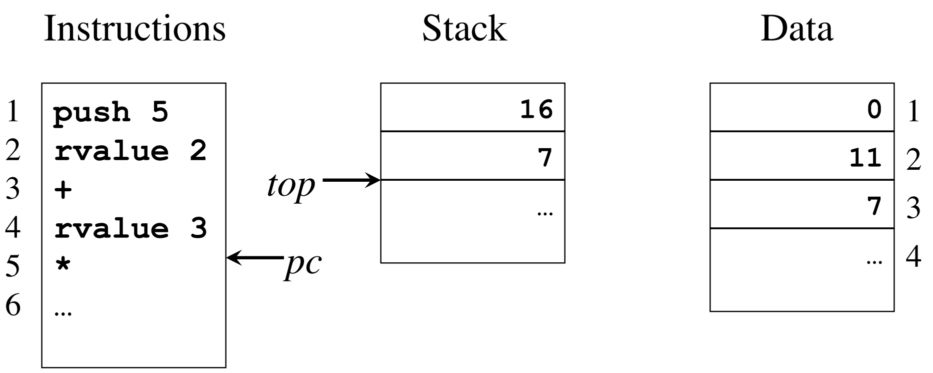
```
/* global.h */
#define DIV 257 /* token */
#define MOD 258 /* token */
#define ID  259 /* token */

/* init.c */
insert(“div”, DIV);
insert(“mod”, MOD);

/* lexer.c */
int lexan()
{   …
    tokenval = lookup(lexbuf);
    if (tokenval == 0)
        tokenval = insert(lexbuf, ID);
    return symtable[p].token;
}
```

Handling Reserved Keywords (cont’d)

morefactors → **div** *factor* { print(‘DIV’) } *morefactors*
| **mod** *factor* { print(‘MOD’) } *morefactors*
| …

```
/* parser.c */
morefactors()
{   if (lookahead == DIV)
    {   match(DIV); factor(); printf(“DIV”); morefactors();
    }
    else if (lookahead == MOD)
    {   match(MOD); factor(); printf(“MOD”); morefactors();
    }
    else
        …
}
```

Abstract Stack Machines



Instructions

1 push 5
2 rvalue 2
3 +
4 rvalue 3
5 *
6 …

Stack

16
7
…

Data

0	1
11	2
7	3
…	4

Generic Instructions for Stack Manipulation

push *v*	push constant value *v* onto the stack
rvalue *l*	push contents of data location *l*
lvalue *l*	push address of data location *l*
pop	discard value on top of the stack
:=	the r-value on top is placed in the l-value below it and both are popped
copy	push a copy of the top value on the stack
+	add value on top with value below it pop both and push result
-	subtract value on top from value below it pop both and push result
***, /,** …	ditto for other arithmetic operations
<, &, …	ditto for relational and logical operations

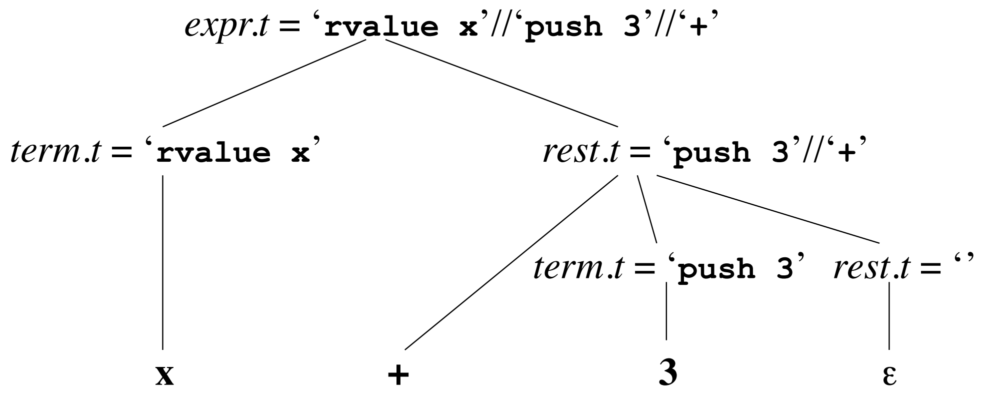
Generic Control Flow Instructions

`label` *l*	label instruction with *l*
`goto` *l*	jump to instruction labeled *l*
`gofalse` *l*	pop the top value, if zero then jump to *l*
`gotrue` *l*	pop the top value, if nonzero then jump to *l*
`halt`	stop execution
`jsr` *l*	jump to subroutine labeled *l*, push return address
`return`	pop return address and return to caller

Syntax-Directed Translation of Expressions

expr → *term rest* { *expr*.*t* := *term*.*t* // *rest*.*t* }
rest → **+** *term* $rest_1$ { *rest*.*t* := *term*.*t* // ‘+’ // $rest_1$.*t* }
rest → **-** *term* $rest_1$ { *rest*.*t* := *term*.*t* // ‘-’ // $rest_1$.*t* }
rest → ε { *rest*.*t* := ‘’ }
term → **num** { *term*.*t* := ‘`push` ’ // **num**.*value* }
term → **id** { *term*.*t* := ‘`rvalue` ’ // **id**.*lexeme* }

Syntax-Directed Translation of Expressions (cont'd)



expr.t = '`rvalue x`'//'`push 3`'//'`+`'

term.t = '`rvalue x`'

rest.t = '`push 3`'//'`+`'

term.t = '`push 3`'

rest.t = ''

x

+

3

ε

Translation Scheme to Generate Abstract Machine Code

expr → *term moreterms*
moreterms → **+** *term* { print('`+`') } *moreterms*
moreterms → **-** *term* { print('`-`') } *moreterms*
moreterms → ε
term → *factor morefactors*
morefactors → ***** *factor* { print('`*`') } *morefactors*
morefactors → **div** *factor* { print('`DIV`') } *morefactors*
morefactors → **mod** *factor* { print('`MOD`') } *morefactors*
morefactors → ε
factor → **(** *expr* **)**
factor → **num** { print('`push `' // **num**.*value*) }
factor → **id** { print('`rvalue `' // **id**.*lexeme*) }

Translation Scheme to Generate Abstract Machine Code (cont'd)

stmt → **id :=** { print('`lvalue `' // **id**.*lexeme*) } *expr* { print('`:=`') }

`lvalue` **id**.*lexeme*
code for *expr*
`:=`

Translation Scheme to Generate Abstract Machine Code (cont'd)

stmt → **if** *expr* { *out* := newlabel(); print('`gofalse `' // *out*) }
then *stmt* { print('`label `'// *out*) }

code for *expr*
`gofalse` *out*
code for *stmt*
`label` *out*

Translation Scheme to Generate Abstract Machine Code (cont’d)

stmt → **while** { *test* := newlabel(); print(‘`label` ’ // *test*) }
expr { *out* := newlabel(); print(‘`gofalse` ’ // *out*) }
do *stmt* { print(‘`goto` ’ // *test* // ‘`label` ’ // *out*) }

`label` *test*
code for *expr*
`gofalse` *out*
code for *stmt*
`goto` *test*
`label` *out*

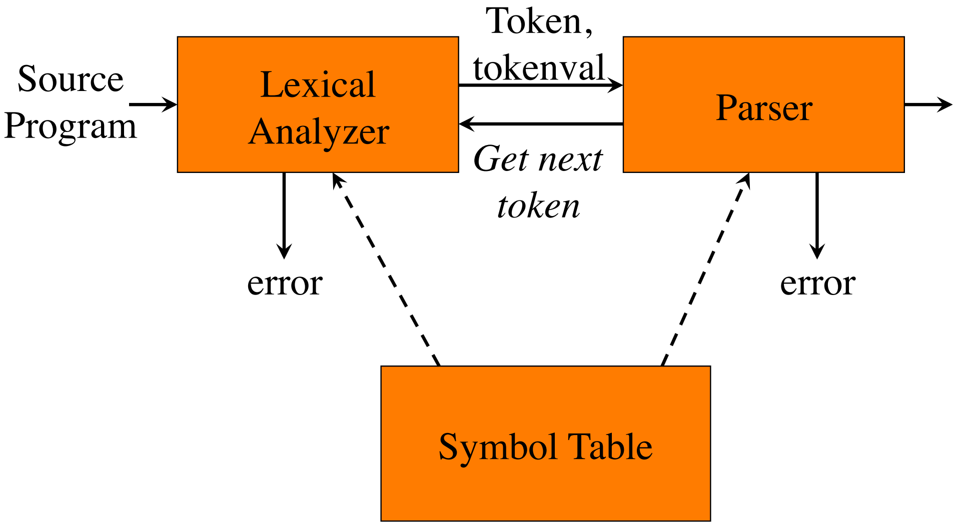
Translation Scheme to Generate Abstract Machine Code (cont’d)

start → *stmt* { print(‘`halt`’) }
stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt* **;** *opt_stmts* | ε

The Reason Why Lexical Analysis is a Separate Phase

- Simplifies the design of the compiler
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be more easily translated

Interaction of the Lexical Analyzer with the Parser



Source Program

Lexical Analyzer

Token, tokenval

Get next token

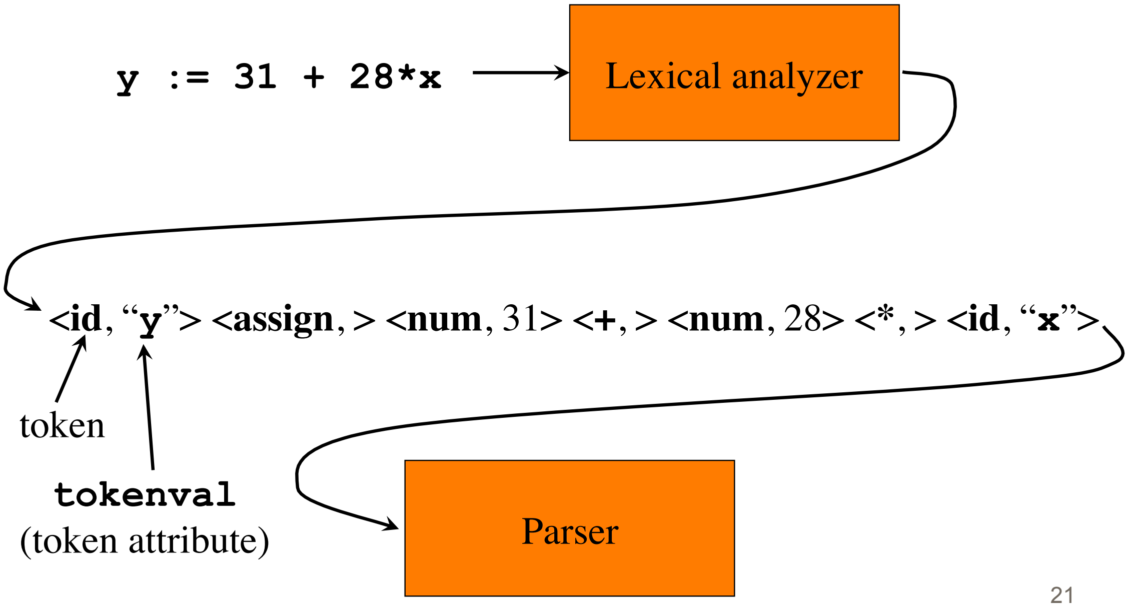
Parser

error

error

Symbol Table

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: Terminology

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

Specification of Patterns for Tokens: String Operations

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

  $s^0 = \varepsilon$

  $s^i = s^{i-1}s$ 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*
  $L^0 = \{\varepsilon\};\ L^i = L^{i-1}L$
- *Kleene closure*
  $L^* = \cup_{i=0,\dots,\infty}\ L^i$
- *Positive closure*
  $L^+ = \cup_{i=1,\dots,\infty}\ L^i$

Specification of Patterns for Tokens: Regular Expressions

- Basis symbols:
  - ε 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
  - $r \mid s$ is a regular expression denoting $L(r) \cup M(s)$
  - rs is a regular expression denoting $L(r)M(s)$
  - 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

- Naming convention for regular expressions:

  $d_1 \to r_1$
  $d_2 \to r_2$
  …
  $d_n \to r_n$

  where r_i is a regular expression over

  $\Sigma \cup \{d_1, d_2, \ldots, d_{i\text{-}1}\}$
- Each d_j in r_i is textually substituted in r_i

Specification of Patterns for Tokens: Regular Definitions

- Example:

  letter → **`A`** | **`B`** | … | **`Z`** | **`a`** | **`b`** | … | **`z`**
  digit → **`0`** | **`1`** | … | **`9`**
  id → **letter** (**letter** | **digit**)*
- Cannot use recursion, this is illegal:

  digits → **digit digits** | **digit**

Specification of Patterns for Tokens: Notational Shorthands

- We frequently use the following shorthands:

  $r^+ = rr^*$

  $r? = r \mid \varepsilon$

  `[a-z]` = `a` | `b` | `c` | … | `z`
- For example:

  digit → `[0-9]`

  num → **digit**$^+$ (**. digit**$^+$)? (**E** (**+**|**-**)? **digit**$^+$)?

Regular Definitions and Grammars

Grammar

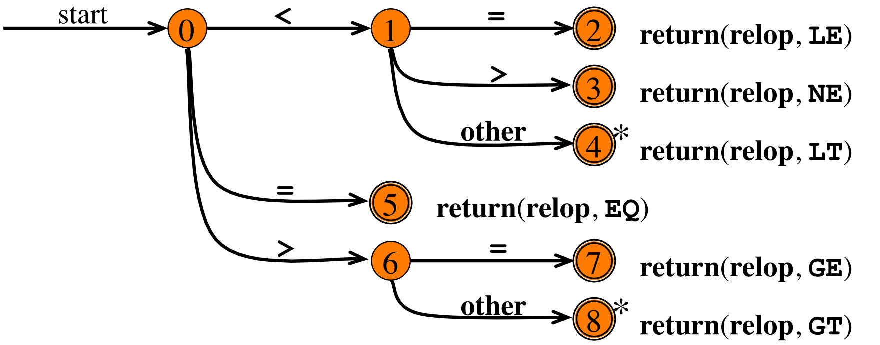
stmt → **if** *expr* **then** *stmt*
    | **if** *expr* **then** *stmt* **else** *stmt*
    | ε
expr → *term* **relop** *term*
    | *term*
term → **id**
    | **num**

Regular definitions

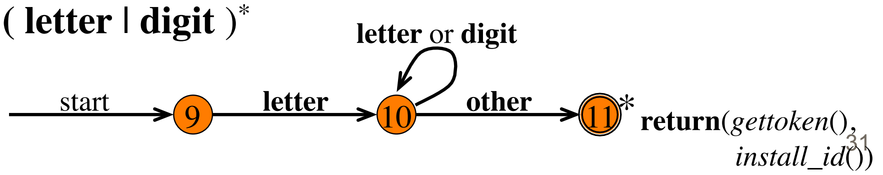
if → `if`
then → `then`
else → `else`
relop → **<** | **<=** | **<>** | **>** | **>=** | **=**
id → **letter** (**letter** | **digit**)*
num → **digit**$^+$ (**. digit**$^+$)? (**E** (**+**|**-**)? **digit**$^+$)?

Implementing a Scanner Using Transition Diagrams

relop → **<** | **<=** | **<>** | **>** | **>=** | **=**



id → **letter** (**letter** | **digit**)*



Implementing a Scanner Using 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 what
other start state
is applicable

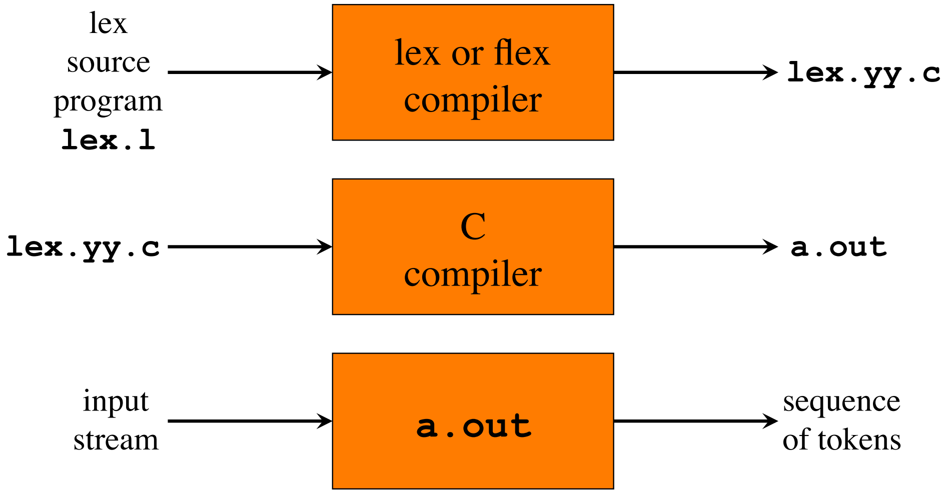
↓

```
int fail()
{ forward = token_beginning;
  swith (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;
}
```

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

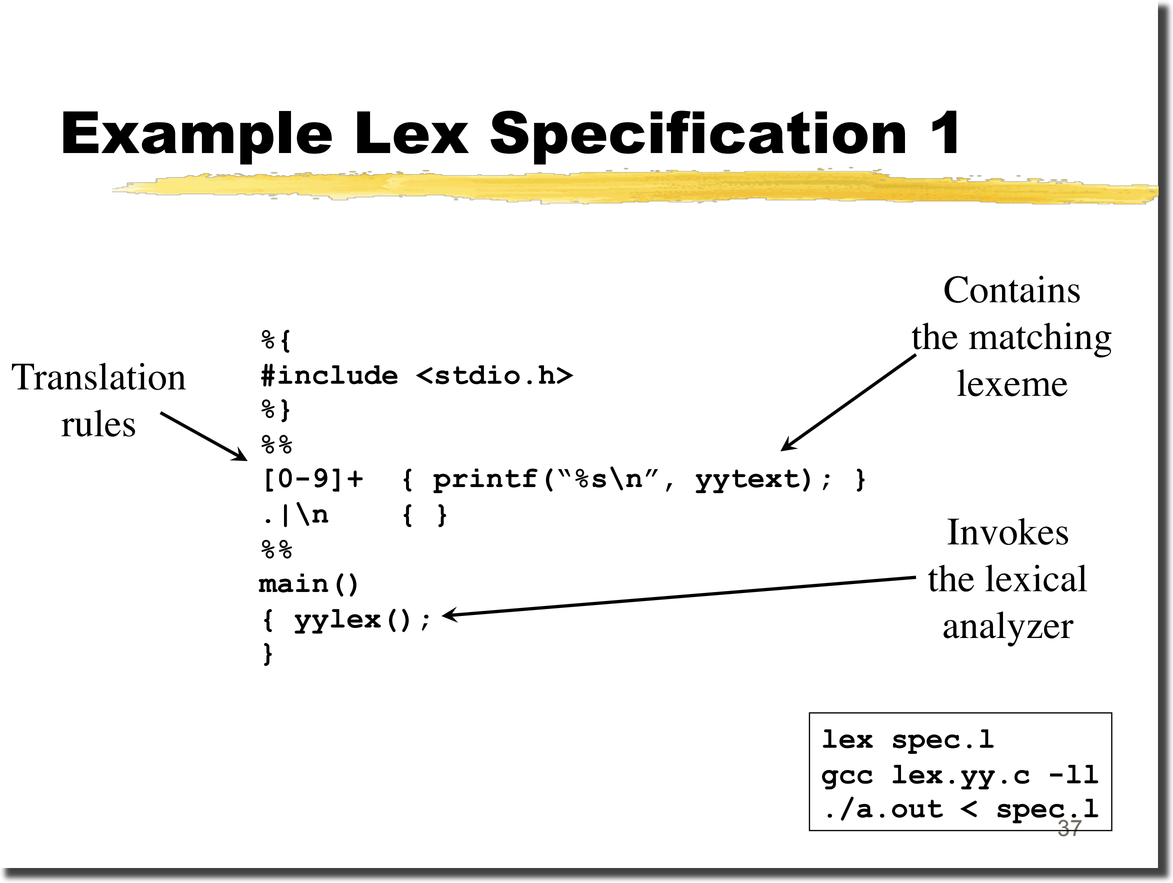


Lex Specification

- 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:
  p_1 { $action_1$ }
  p_2 { $action_2$ }
  …
  p_n { $action_n$ }

Regular Expressions in Lex

`x`	match the character **`x`**
`\.`	match the character `.`
"*string*"	match contents of string of characters
`.`	match any character except newline
`^`	match beginning of a line
`$`	match the end of a line
`[xyz]`	match one character **`x`**, **`y`**, or **`z`** (use `\` to escape `-`)
`[^xyz]`	match any character except **`x`**, **`y`**, and **`z`**
`[a-z]`	match one of **`a`** to **`z`**
r`*`	closure (match zero or more occurrences)
r`+`	positive closure (match one or more occurrences)
r`?`	optional (match zero or one occurrence)
r_1r_2	match r_1 then r_2 (concatenation)
$r_1 \mid r_2$	match r_1 or r_2 (union)
`(` r `)`	grouping
$r_1 \backslash r_2$	match r_1 when followed by r_2
`{`d`}`	match the regular expression defined by d



Example Lex Specification 1

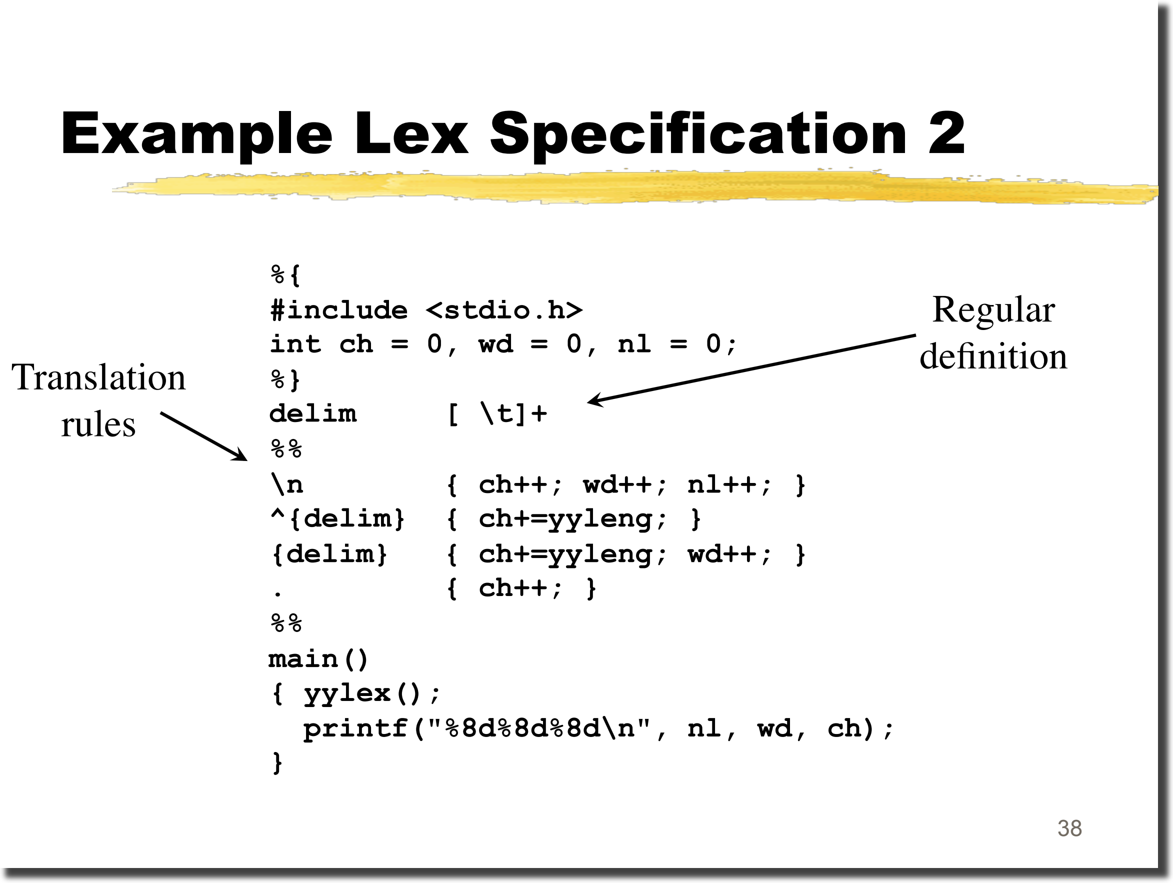
Translation rules

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

Contains the matching lexeme

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```



Example Lex Specification 2

Translation rules

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

Regular definition

Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#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();
}
```

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}+)?
%%
{ws}       { }
if         {return IF;}
then       {return THEN;}
else       {return ELSE;}
{id}       {yylval = install_id(); return ID;}
{number}   {yylval = install_num(); return NUMBER;}
"<"        {yylval = LT; return RELOP;}
"<="       {yylval = LE; return RELOP;}
"="        {yylval = EQ; return RELOP;}
"<>"       {yylval = NE; return RELOP;}
">"        {yylval = GT; return RELOP;}
">="       {yylval = GE; return RELOP;}
%%
int install_id()
…
```

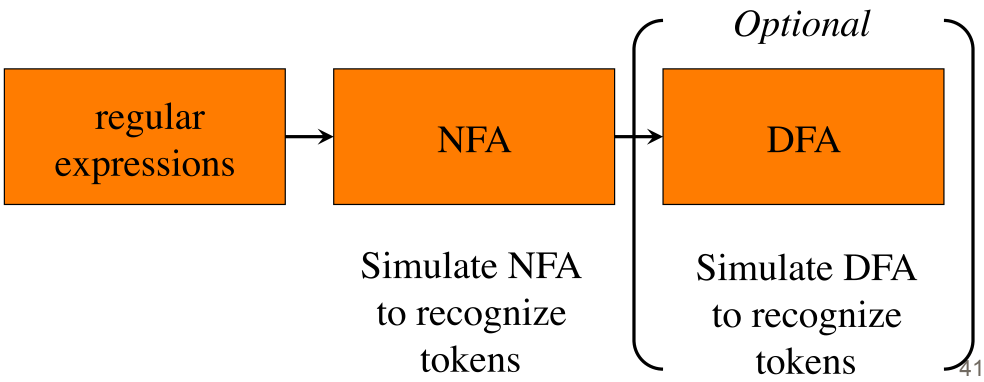
Return token to parser

Token attribute

Install **`yytext`** as identifier in symbol table

Design of a Lexical Analyzer Generator

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



Optional

regular expressions

NFA

DFA

Simulate NFA to recognize tokens

Simulate DFA to recognize tokens

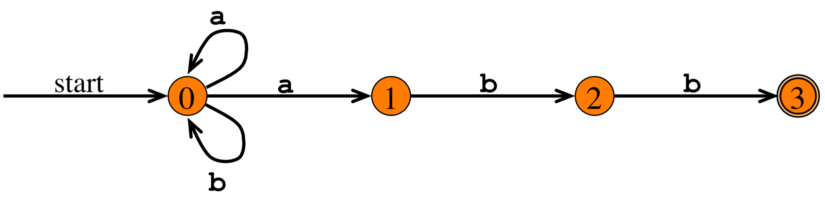
Nondeterministic Finite Automata

- Definition: an NFA is a 5-tuple (S,Σ,δ,s_0,F) where

  S is a finite set of *states*
  Σ is a finite set of *input symbol alphabet*
  δ 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*



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

Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

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

State	*Input* **a**	*Input* **b**
0	{0, 1}	{0}
1		{2}
2		{3}

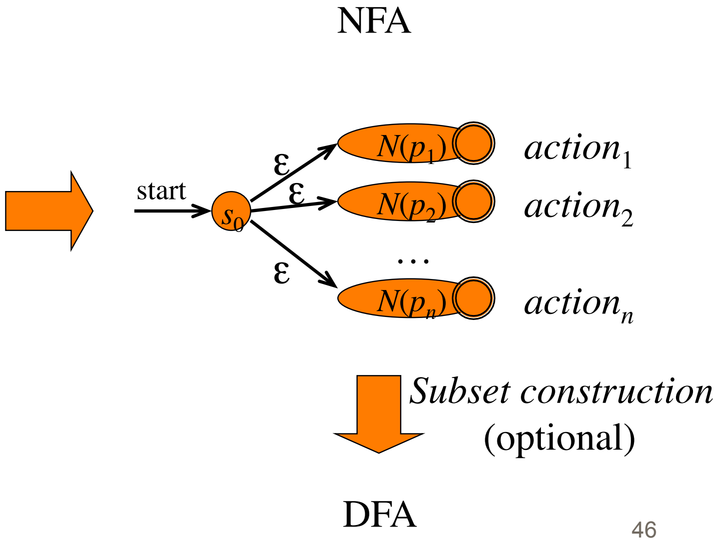
The Language Defined by an NFA

- An NFA *accepts* an input string *x* **iff** 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 (**`a`**|**`b`**)***`abb`** for the example NFA

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

Lex specification with regular expressions

p_1 { $action_1$ }
p_2 { $action_2$ }
…
p_n { $action_n$ }



From Regular Expression to NFA (Thompson's Construction)

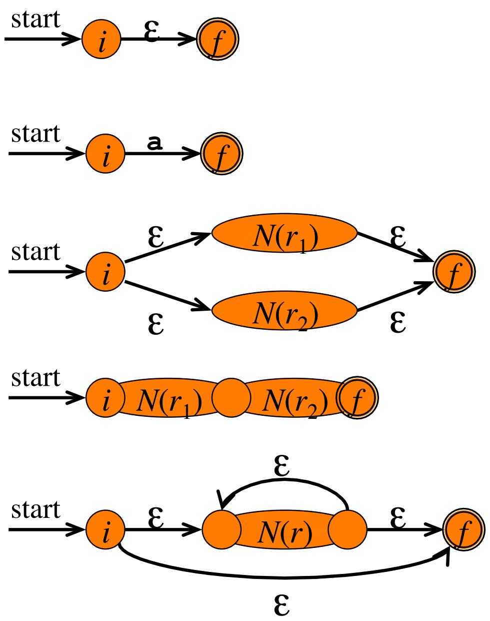
ε

a

$r_1 \mid r_2$

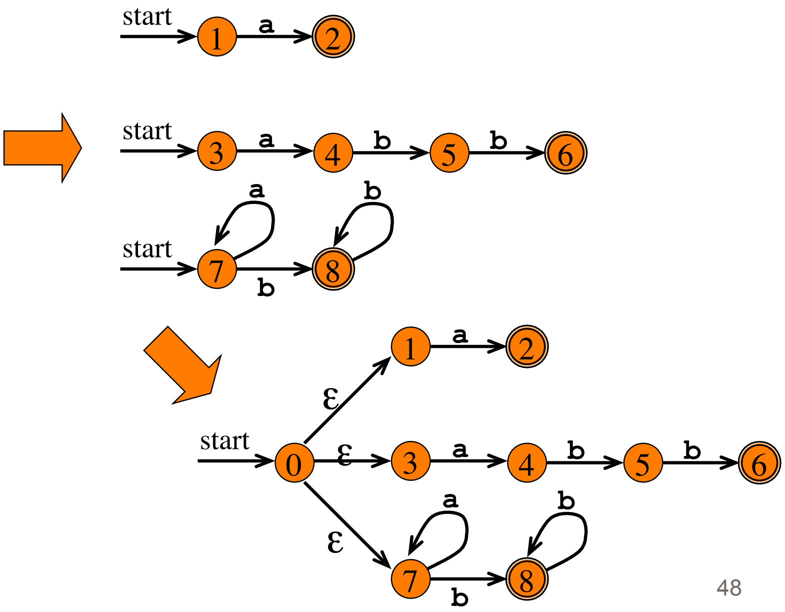
$r_1 r_2$

r^*

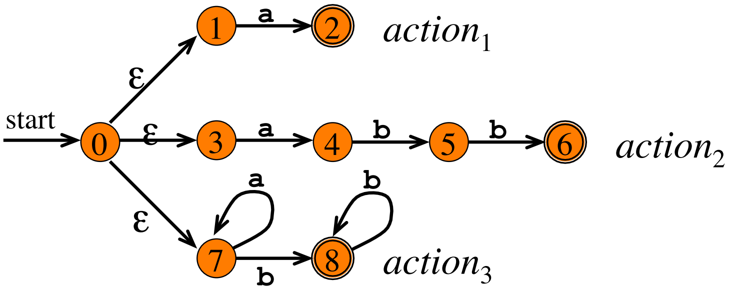


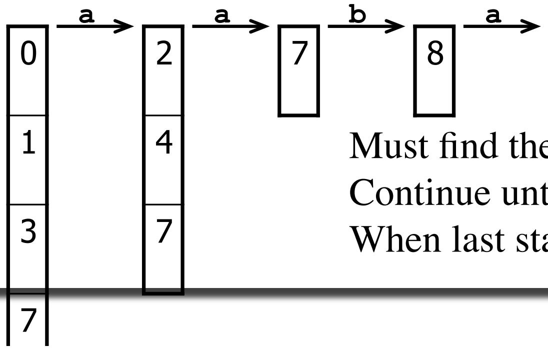
Combining the NFAs of a Set of Regular Expressions

a { *action*$_1$ }
abb { *action*$_2$ }
a*b+ { *action*$_3$ }



Simulating the Combined NFA Example 1

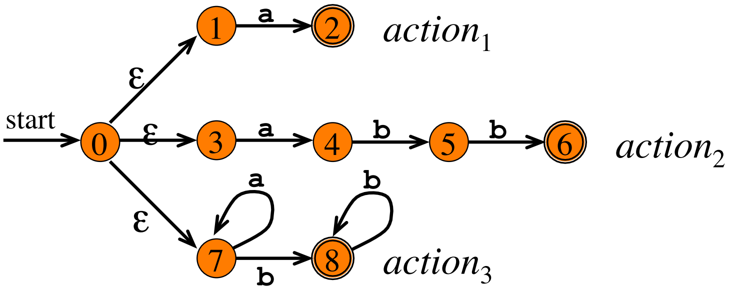


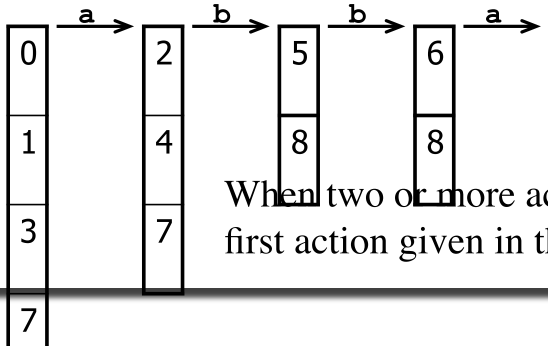


none
action$_3$

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





none
action$_2$
action$_3$

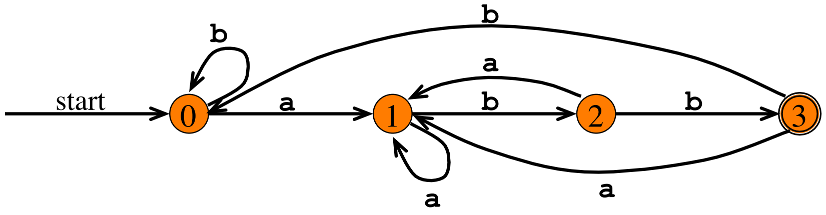
When two or more accepting states are reached, the first action given in the Lex specification is executed

Deterministic Finite Automata

- A *deterministic finite automaton* is a special case of an NFA
  - No state has an ε-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**|**b**)***abb**



Conversion of an NFA into a DFA

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

  $\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \rightarrow_{\varepsilon} \ldots \rightarrow_{\varepsilon} t\}$

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

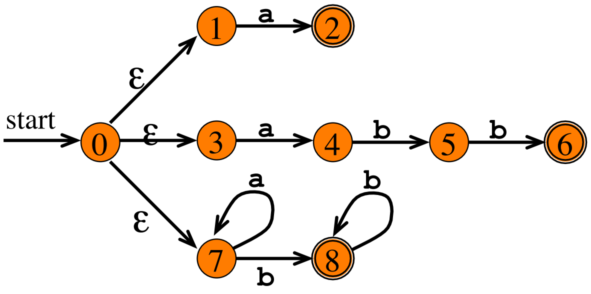
  $move(T,a) = \{t \mid s \rightarrow_{a} t \text{ and } s \in T\}$

- The algorithm produces:

  Dstates is the set of states of the new DFA consisting of sets of states of the NFA

  Dtran is the transition table of the new DFA

ε-*closure* and *move* Examples



$\varepsilon\text{-}closure(\{0\}) = \{0,1,3,7\}$

$move(\{0,1,3,7\},\mathbf{a}) = \{2,4,7\}$

$\varepsilon\text{-}closure(\{2,4,7\}) = \{2,4,7\}$

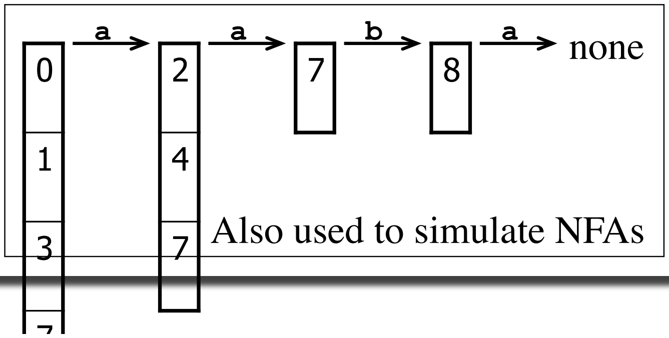
$move(\{2,4,7\},\mathbf{a}) = \{7\}$

$\varepsilon\text{-}closure(\{7\}) = \{7\}$

$move(\{7\},\mathbf{b}) = \{8\}$

$\varepsilon\text{-}closure(\{8\}) = \{8\}$

$move(\{8\},\mathbf{a}) = \varnothing$



Also used to simulate NFAs

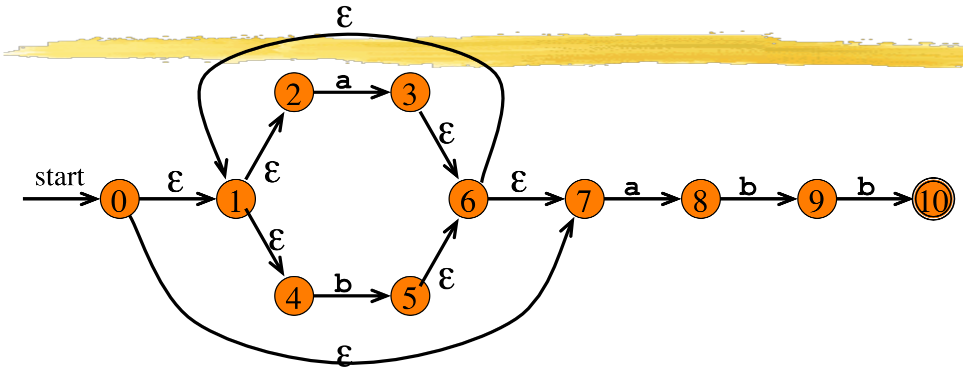
Simulating an NFA using ε-*closure* and *move*

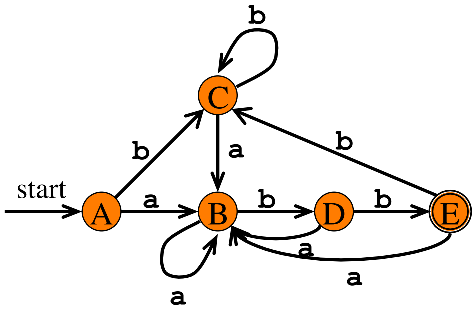
```
$S$ := ε-$closure$({$s_0$})
$S_{prev}$ := $\emptyset$
$a$ := $nextchar$()
while $S \neq \emptyset$ do
        $S_{prev}$ := $S$
        $S$ := ε-$closure$($move$($S$,$a$))
        $a$ := $nextchar$()
end do
if $S_{prev} \cap F \neq \emptyset$ then
        execute $action$ in $S_{prev}$
        return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

The Subset Construction Algorithm

```
Initially, ε-$closure$($s_0$) 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
                $U$ := ε-$closure$($move$($T$,$a$))
                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}

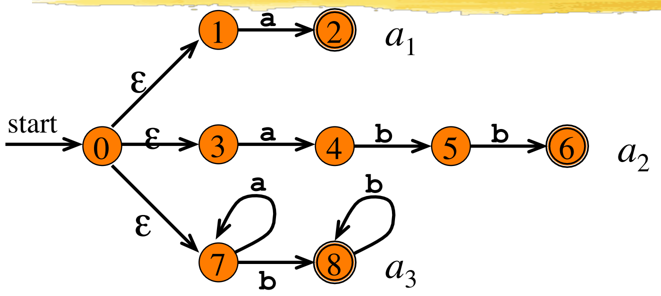
B = {1,2,3,4,6,7,8}

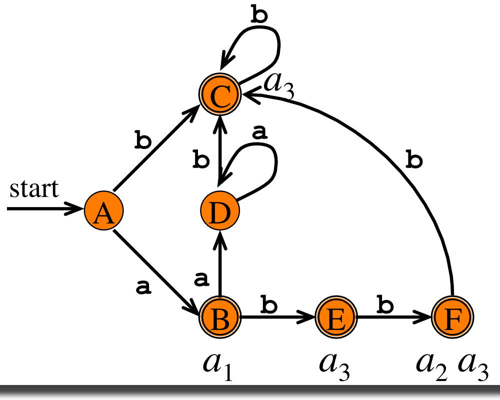
C = {1,2,4,5,6,7}

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}

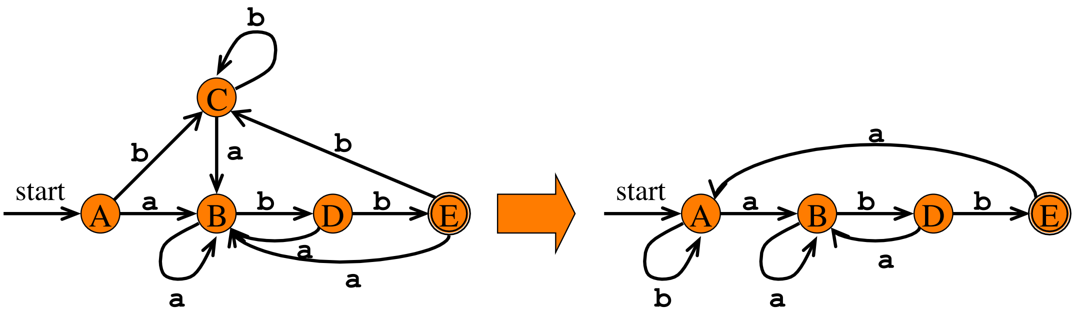
C = {8}

D = {7}

E = {5,8}

F = {6,8}

Minimizing the Number of States of a DFA



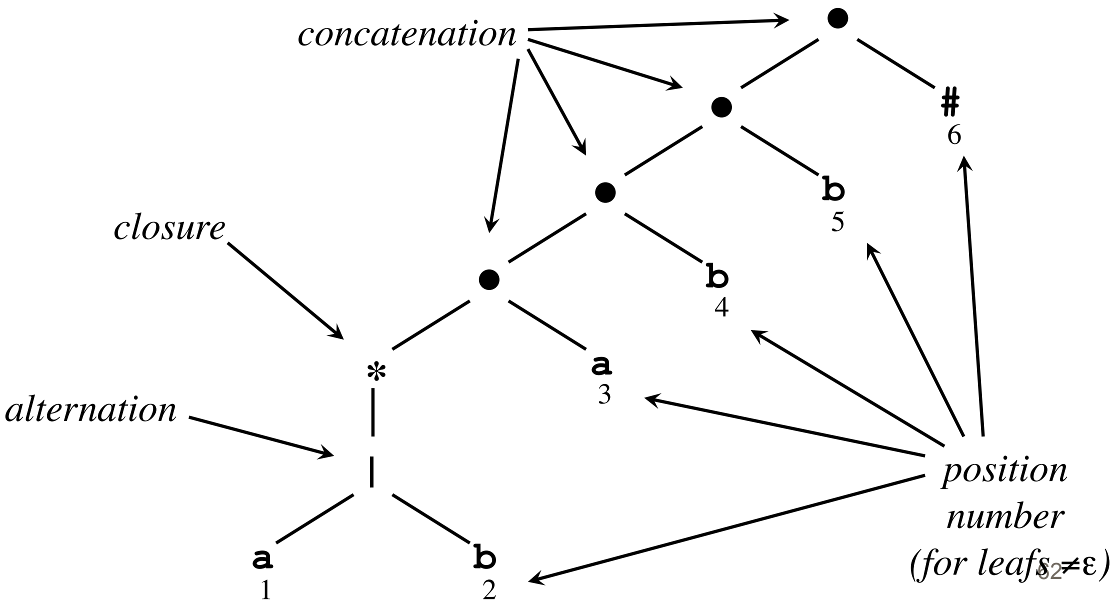
From Regular Expression to DFA Directly

- The *important states* of an NFA are those without an ε-transition, that is if *move*({*s*},*a*) ≠ ∅ for some *a* then *s* is an important state
- The subset construction algorithm uses only the important states when it determines ε-*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

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

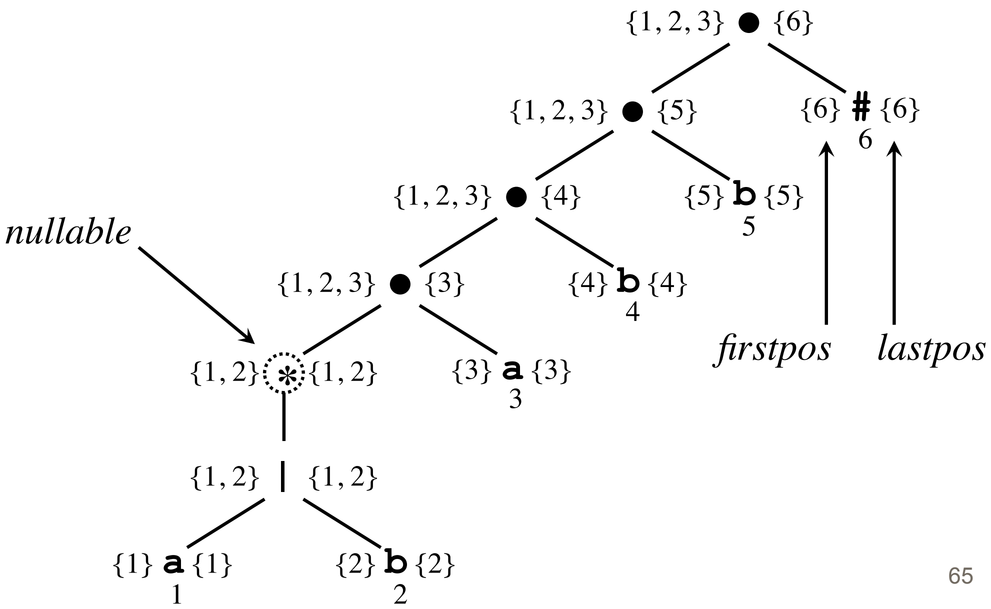
From Regular Expression to DFA Directly: Annotating the Tree

Node *n*	*nullable*(*n*)	*firstpos*(*n*)	*lastpos*(*n*)
Leaf ε	true	∅	∅
Leaf *i*	false	{*i*}	{*i*}
| / \\ c_1 c_2	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)	*lastpos*(c_1) ∪ *lastpos*(c_2)

•
/ \\
c_1 c_2
	nullable(c_1)
	and
	nullable(c_2)
		if *nullable*(c_1)
		then *firstpos*(c_1)
		∪ *firstpos*(c_2)
		else *firstpos*(c_1)
			if *nullable*(c_2)
			then *lastpos*(c_1)
			∪ *lastpos*(c_2)
			else *lastpos*(c_2)

* | c_1	true	*firstpos*(c_1)	*lastpos*(c_1)

From Regular Expression to DFA Directly: Syntax Tree of (a|b)*abb#



From Regular Expression to DFA Directly: *followpos*

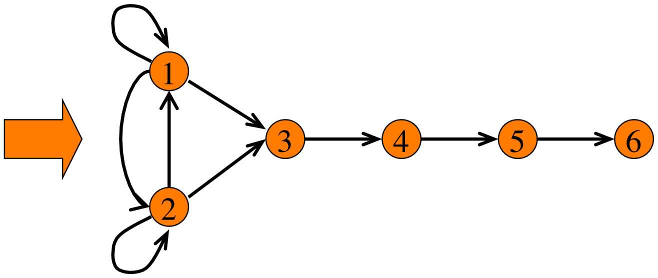
```
for each node $n$ in the tree do
        if $n$ is a cat-node with left child $c_1$ and right child $c_2$ then
                for each $i$ in $lastpos(c_1)$ do
                        $followpos(i) := followpos(i) \cup firstpos(c_2)$
                end do
        else if $n$ is a star-node
   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

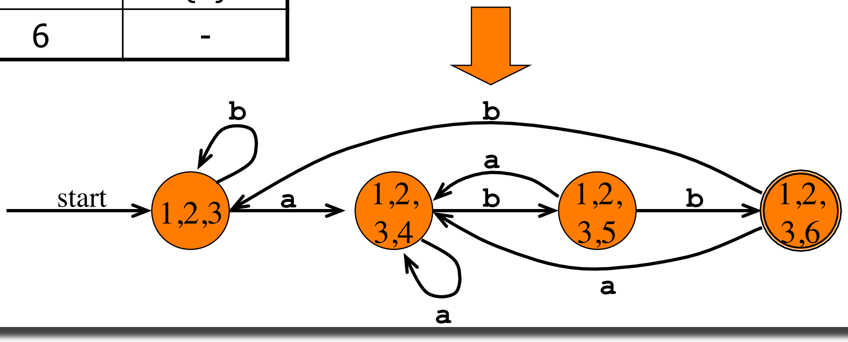
From Regular Expression to DFA Directly: Algorithm

```
$s_0$ := $firstpos(root)$ where $root$ is the root of the syntax tree
$Dstates$ := $\{s_0\}$ and is unmarked
while there is an unmarked state $T$ in $Dstates$ do
        mark $T$
        for each input symbol $a \in \Sigma$ do
                let $U$ be the set of positions that are in $followpos(p)$
                        for some position $p$ in $T$,
                        such that the symbol at position $p$ is $a$
                if $U$ is not empty and not in $Dstates$ then
                        add $U$ as an unmarked state to $Dstates$
                end if
                $Dtran[T,a]$ := $U$
        end do
end do
```

From Regular Expression to DFA Directly: Example

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-





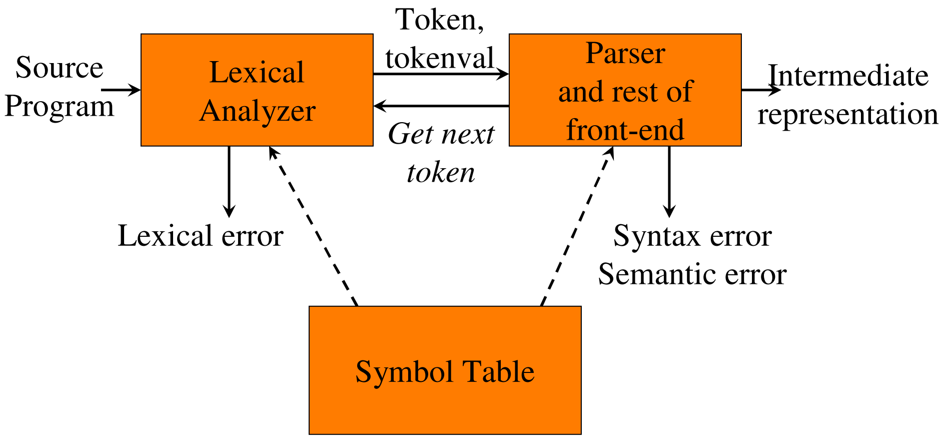
Time-Space Tradeoffs

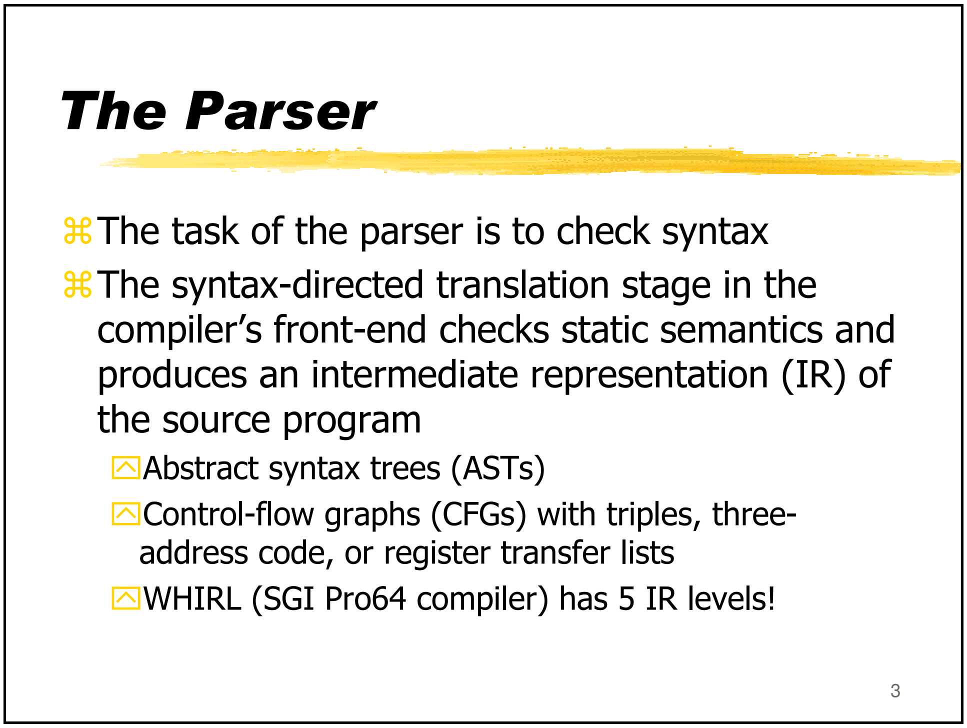
Automaton	*Space (worst case)*	*Time (worst case)*
NFA	$O(\lvert r \rvert)$	$O(\lvert r \rvert \times \lvert x \rvert)$
DFA	$O(2^{\lvert r \rvert})$	$O(\lvert x \rvert)$

Syntax Analysis (1)

Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS

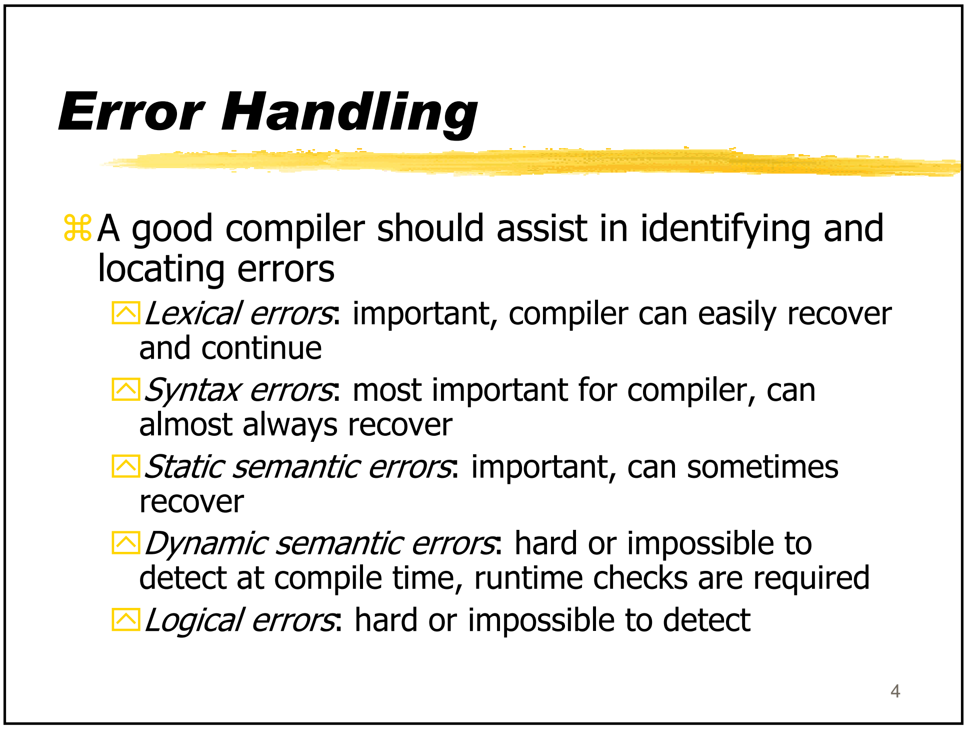
Position of a Parser in the Compiler Model





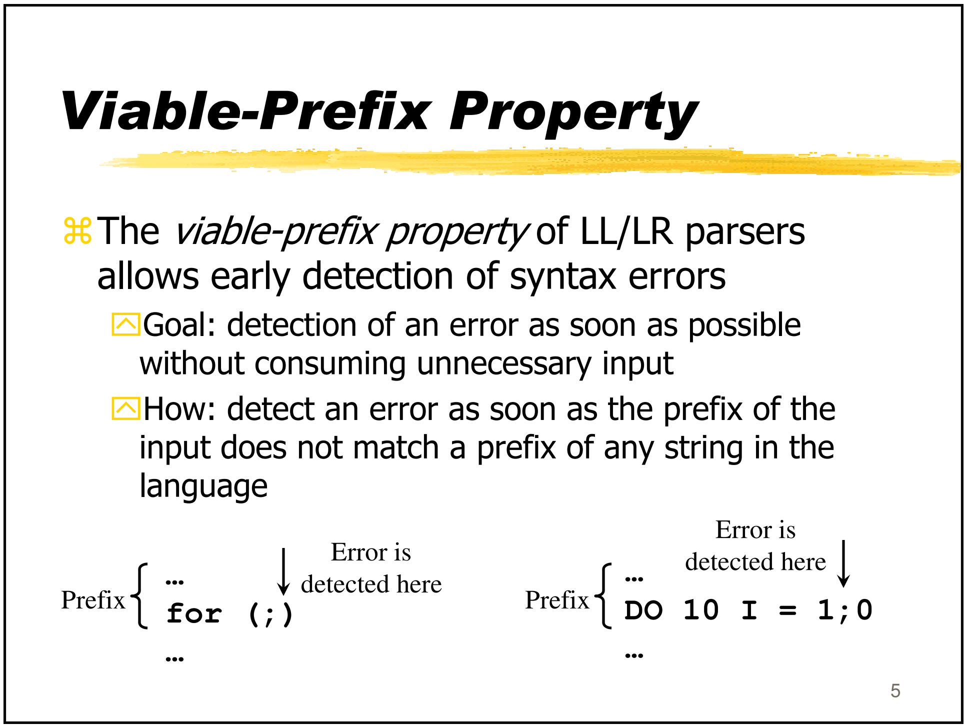
The Parser

- The task of the parser is to check syntax
- The syntax-directed translation stage in the compiler's front-end checks static semantics and produces an intermediate representation (IR) of the source program
  - Abstract syntax trees (ASTs)
  - Control-flow graphs (CFGs) with triples, three-address code, or register transfer lists
  - WHIRL (SGI Pro64 compiler) has 5 IR levels!



Error Handling

- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: important, compiler can easily recover and continue
  - *Syntax errors*: most important for compiler, can almost always recover
  - *Static semantic errors*: important, can sometimes recover
  - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required
  - *Logical errors*: hard or impossible to detect



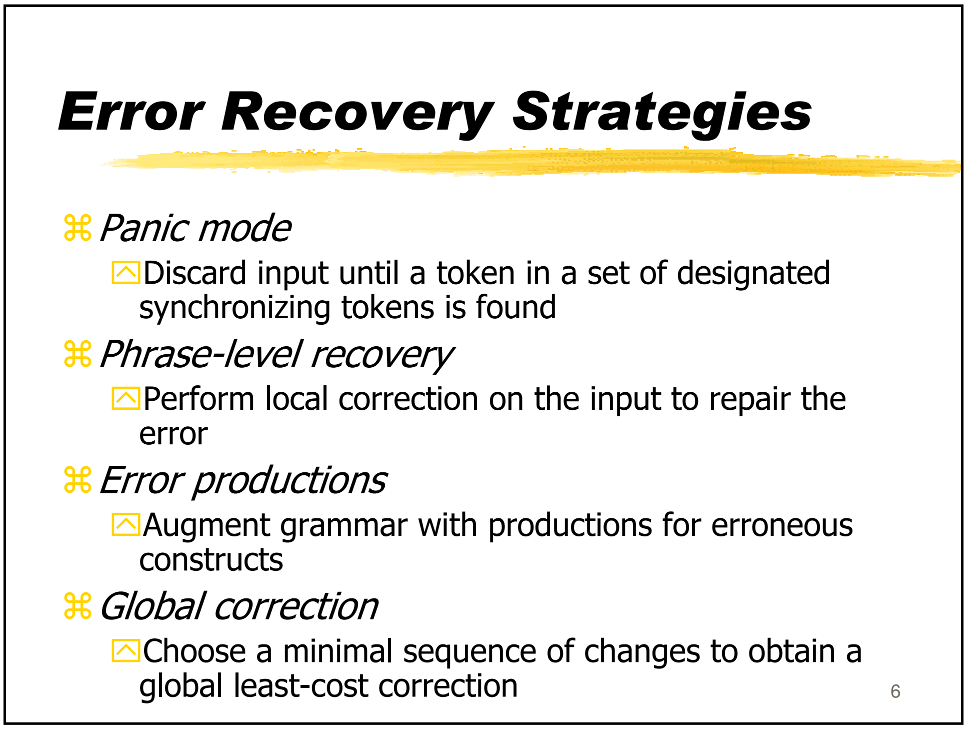
Viable-Prefix Property

- The *viable-prefix property* of LL/LR parsers allows early detection of syntax errors
  - Goal: detection of an error as soon as possible without consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Prefix
```
…
for (;)
…
```
Error is detected here

Prefix
```
…
DO 10 I = 1;0
…
```
Error is detected here

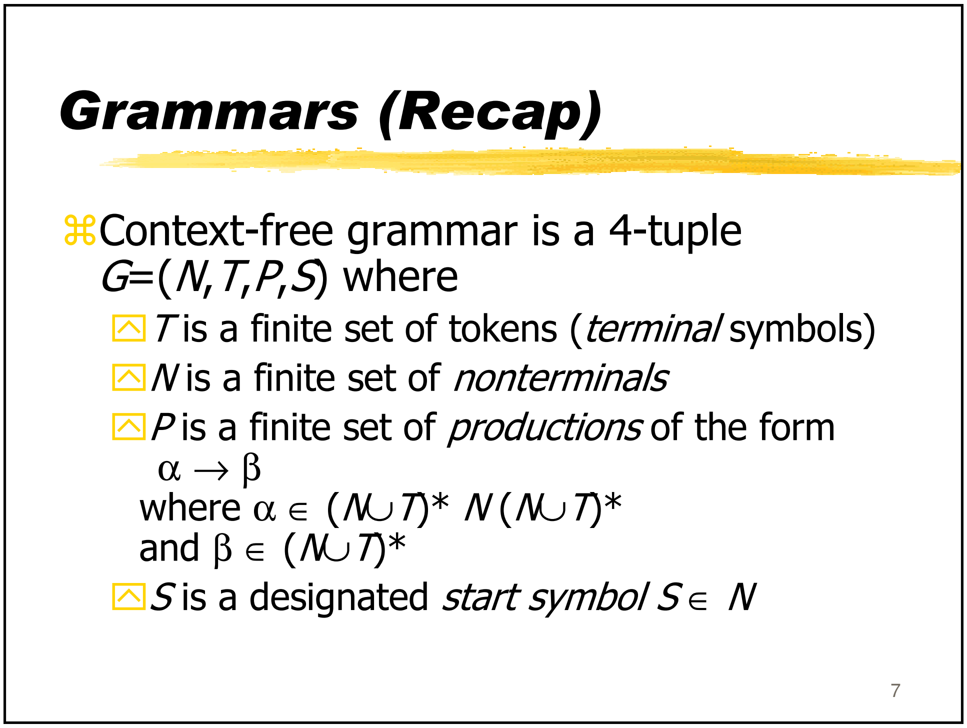
5



Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

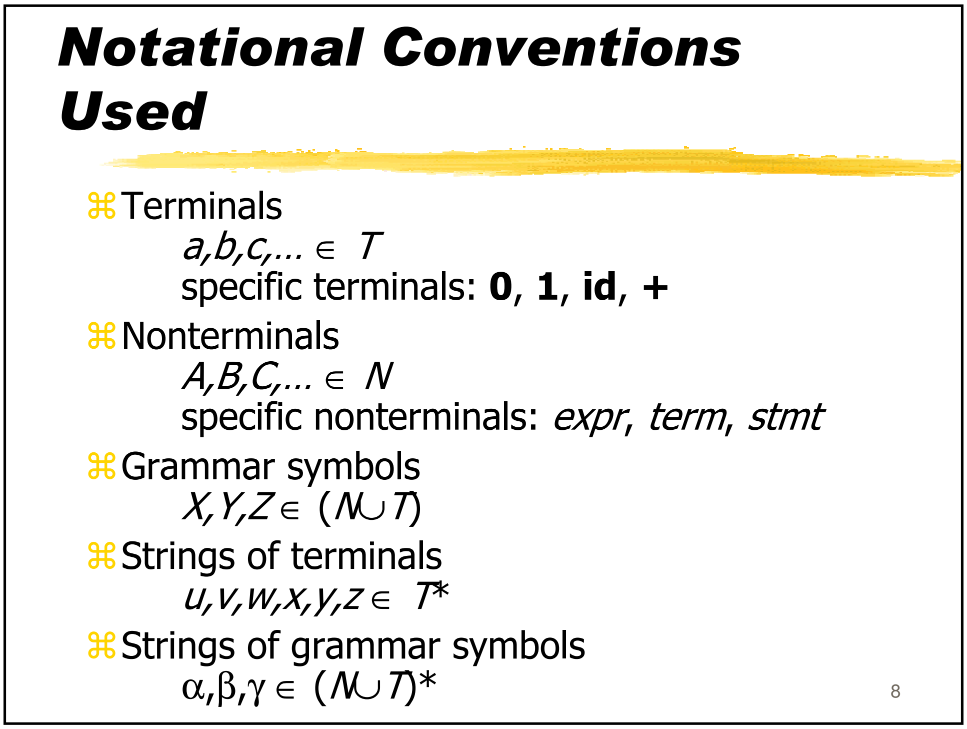
6



Grammars (Recap)

- Context-free grammar is a 4-tuple $G=(N,T,P,S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form
    $\alpha \rightarrow \beta$
    where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$
    and $\beta \in (N \cup T)^*$
  - S is a designated *start symbol* $S \in N$

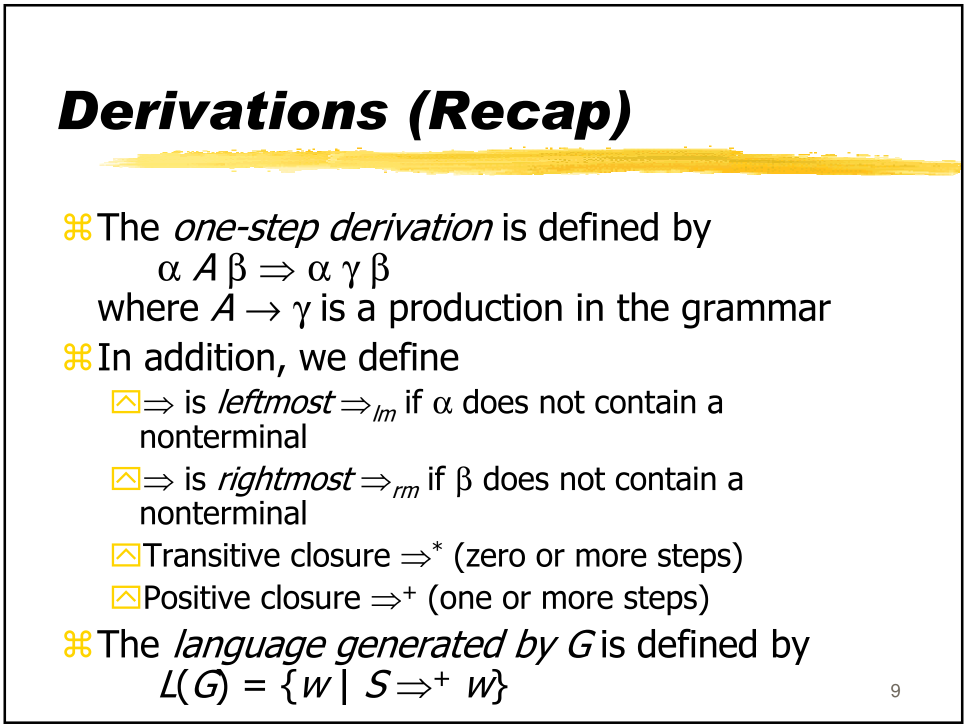
7



Notational Conventions Used

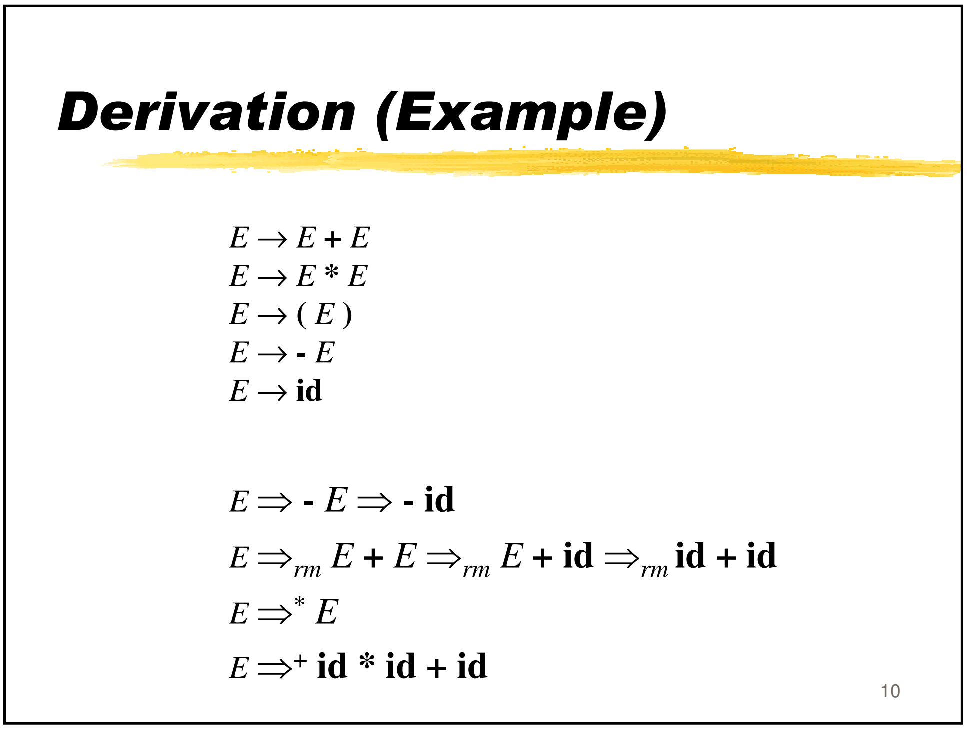
- Terminals
  $a,b,c,\ldots \in T$
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols
  $X,Y,Z \in (N \cup T)$
- Strings of terminals
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  $\alpha,\beta,\gamma \in (N \cup T)^*$

8



Derivations (Recap)

- The *one-step derivation* is defined by
  $\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$
  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- The *language generated by G* is defined by
  $L(G) = \{w \mid S \Rightarrow^+ w\}$



Derivation (Example)

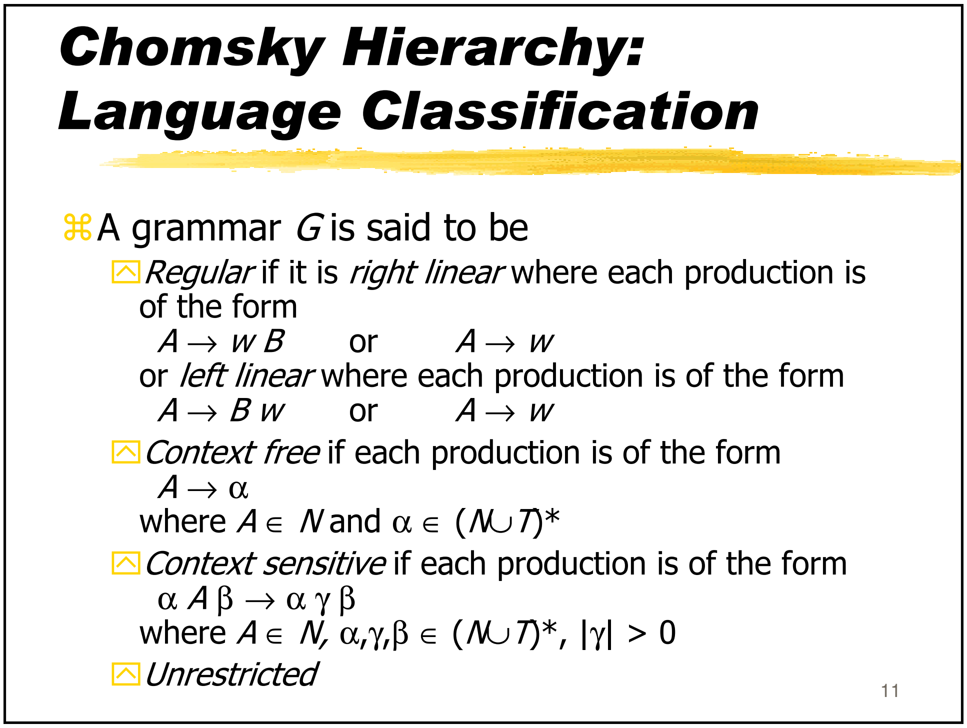
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (\ E\ )$
$E \rightarrow -\ E$
$E \rightarrow \mathbf{id}$

$E \Rightarrow -\ E \Rightarrow -\ \mathbf{id}$

$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$

$E \Rightarrow^* E$

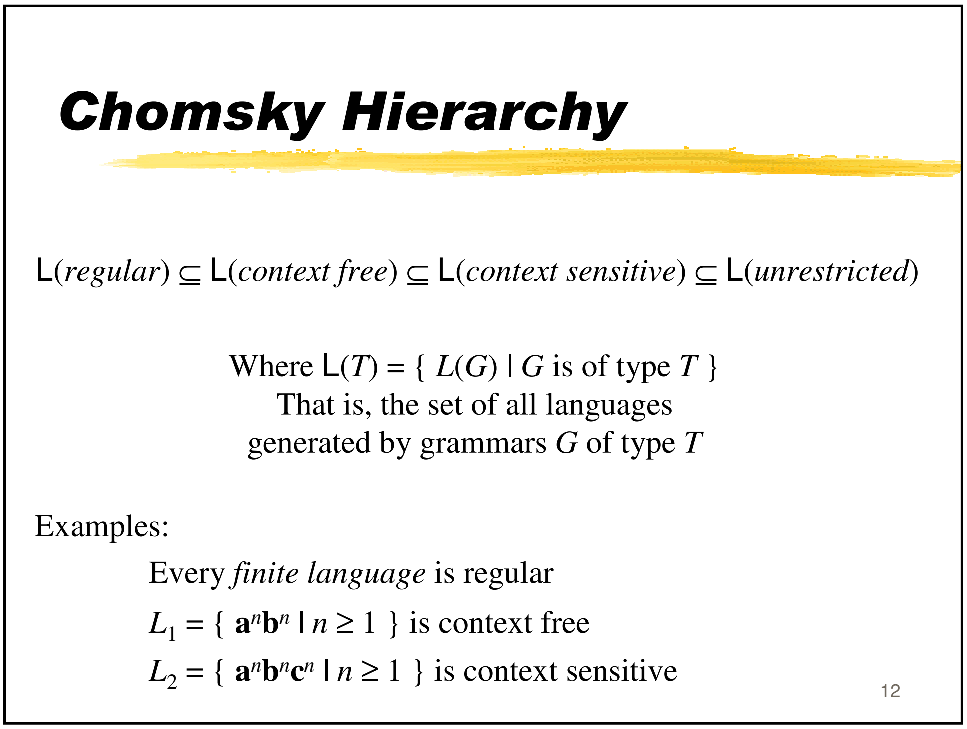
$E \Rightarrow^+ \mathbf{id} * \mathbf{id} + \mathbf{id}$



Chomsky Hierarchy: Language Classification

- A grammar G is said to be
  - *Regular* if it is *right linear* where each production is of the form
    $A \rightarrow w\,B$ or $A \rightarrow w$
    or *left linear* where each production is of the form
    $A \rightarrow B\,w$ or $A \rightarrow w$
  - *Context free* if each production is of the form
    $A \rightarrow \alpha$
    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form
    $\alpha\, A\, \beta \rightarrow \alpha\, \gamma\, \beta$
    where $A \in N$, $\alpha,\gamma,\beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

11



Chomsky Hierarchy

$\mathsf{L}(\textit{regular}) \subseteq \mathsf{L}(\textit{context free}) \subseteq \mathsf{L}(\textit{context sensitive}) \subseteq \mathsf{L}(\textit{unrestricted})$

Where $\mathsf{L}(T) = \{\ L(G) \mid G \text{ is of type } T\ \}$
That is, the set of all languages
generated by grammars G of type T

Examples:

Every *finite language* is regular

$L_1 = \{\ \mathbf{a}^n\mathbf{b}^n \mid n \geq 1\ \}$ is context free

$L_2 = \{\ \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1\ \}$ is context sensitive

12

Parsing

- *Universal* (any C-F grammar)
  - Cocke-Younger-Kasimi
  - Earley
- *Top-down* (C-F grammar with restrictions)
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
- *Bottom-up* (C-F grammar with restrictions)
  - Operator precedence parsing
  - LR (Left-to-right, Rightmost derivation) methods
    - SLR, canonical LR, LALR

13

Top-Down Parsing

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$E \rightarrow T + T$

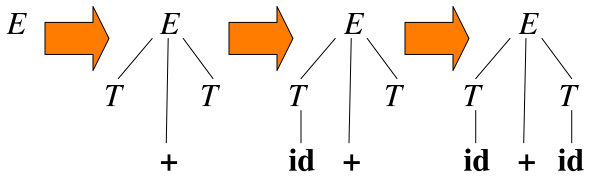
$T \rightarrow (E)$

$T \rightarrow - E$

$T \rightarrow \textbf{id}$

Leftmost derivation:

$$\begin{aligned} E &\Rightarrow_{lm} T + T \\ &\Rightarrow_{lm} \textbf{id} + T \\ &\Rightarrow_{lm} \textbf{id} + \textbf{id} \end{aligned}$$



14

Left Recursion (Recap)

- Productions of the form

  $A \to A\,\alpha$
  $/\ \beta$
  $|\ \gamma$

  are left recursive
- When one of the productions in a grammar is left recursive then a predictive parser may loop forever

General Left Recursion Elimination

```
Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for $i = 1, \ldots, n$ do
    for $j = 1, \ldots, i$-1 do
        replace each
            $A_i \to A_j\,\gamma$
        with
            $A_i \to \delta_1\,\gamma \mid \delta_2\,\gamma \mid \ldots \mid \delta_k\,\gamma$
        where
            $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
    enddo
    eliminate the immediate left recursion in $A_i$
enddo
```

Immediate Left-Recursion Elimination

Rewrite every left-recursive production

$$\begin{array}{ll} A \rightarrow & A\ \alpha \\ & \mid \beta \\ & \mid \gamma \\ & \mid A\ \delta \end{array}$$

into a right-recursive production:

$$\begin{array}{ll} A \rightarrow & \beta\ A_R \\ & \mid \gamma\ A_R \\ A_R \rightarrow & \alpha\ A_R \\ & \mid \delta\ A_R \\ & \mid \varepsilon \end{array}$$

Example Left Rec. Elimination

$$\left.\begin{array}{l} A \rightarrow B\ C \mid \mathbf{a} \\ B \rightarrow C\ A \mid A\ \mathbf{b} \\ C \rightarrow A\ B \mid C\ C \mid \mathbf{a} \end{array}\right\} \text{Choose arrangement: } A,\ B,\ C$$

$i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow C\ A \mid \underline{A}\ \mathbf{b}$

$\Rightarrow$ $B \rightarrow C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

$\Rightarrow_{\text{(imm)}}$ $B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

$i = 3, j = 1$: $C \rightarrow \underline{A}\ B \mid C\ C \mid \mathbf{a}$

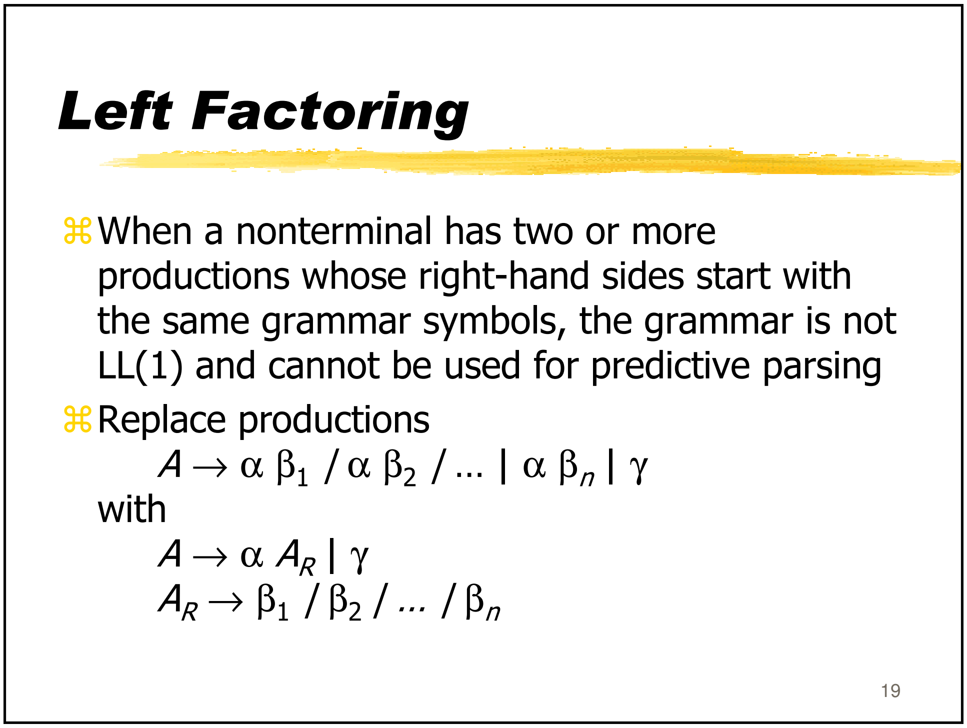
$\Rightarrow$ $C \rightarrow \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow$ $C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{\text{(imm)}}$ $C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$



Left Factoring

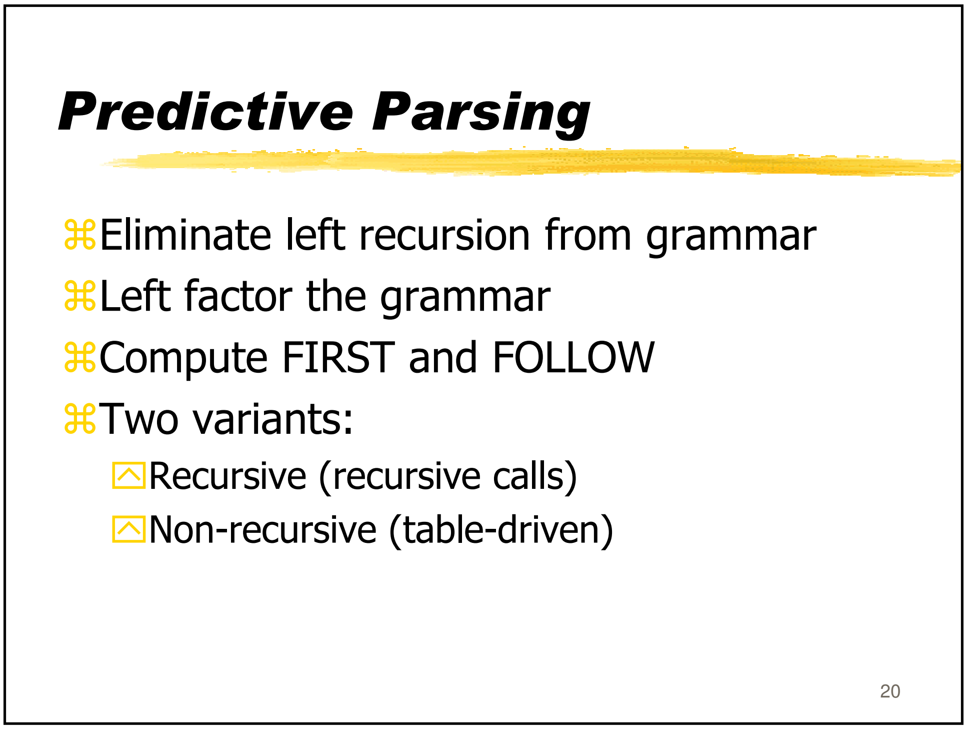
- When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing
- Replace productions

  $A \rightarrow \alpha\ \beta_1\ /\ \alpha\ \beta_2\ /\ \ldots\ |\ \alpha\ \beta_n\ |\ \gamma$

  with

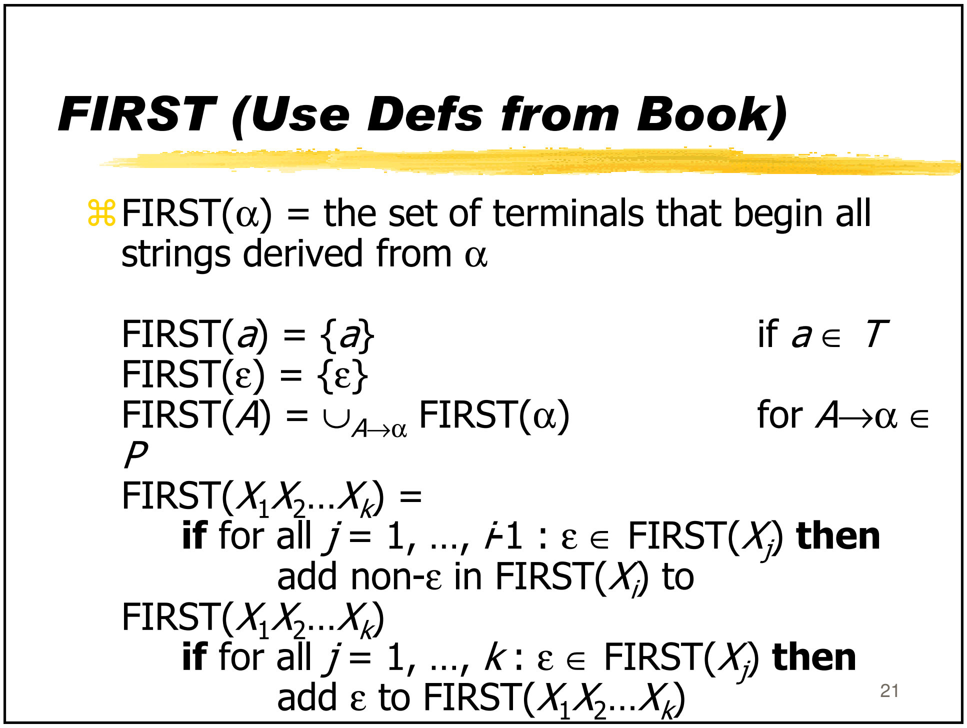
  $A \rightarrow \alpha\ A_R\ |\ \gamma$

  $A_R \rightarrow \beta_1\ /\ \beta_2\ /\ \ldots\ /\ \beta_n$



Predictive Parsing

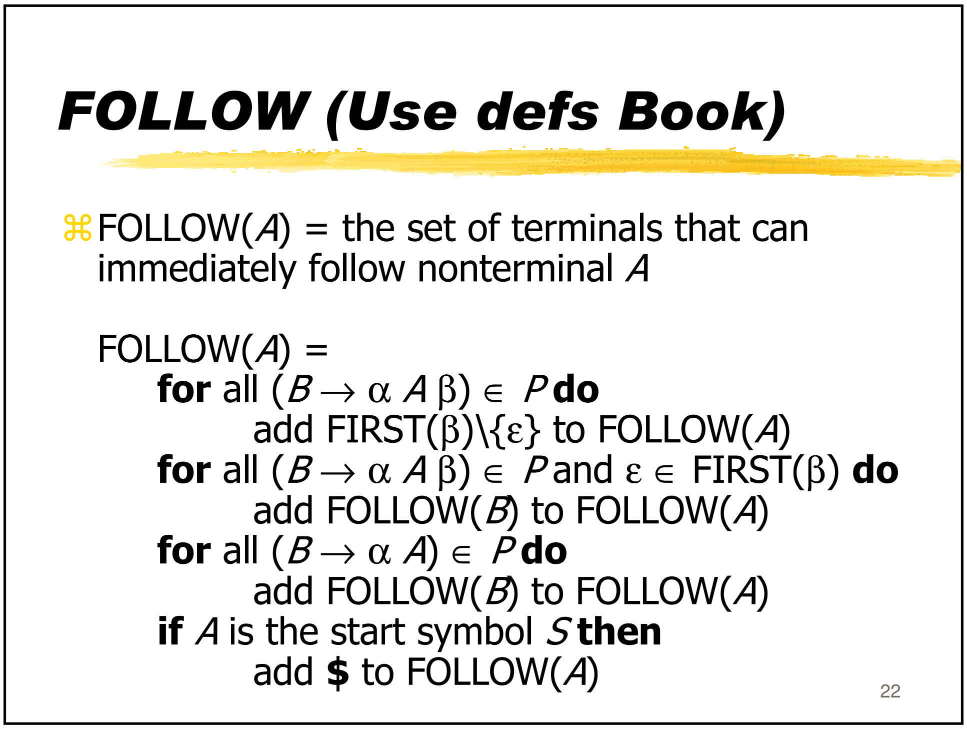
- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
  - Recursive (recursive calls)
  - Non-recursive (table-driven)



FIRST (Use Defs from Book)

- FIRST(α) = the set of terminals that begin all strings derived from α

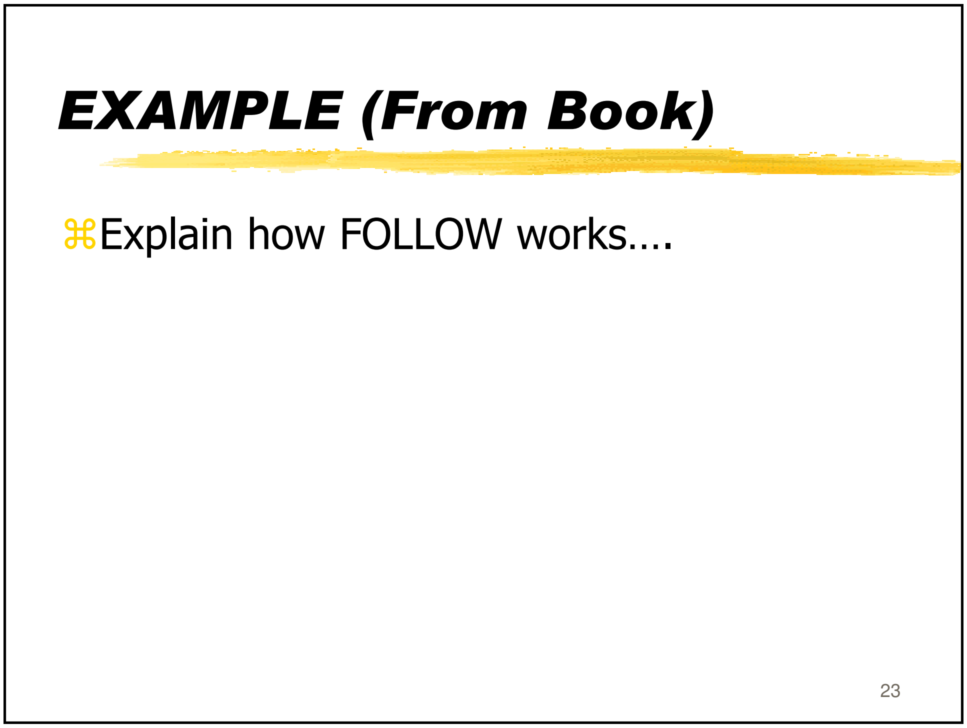
```
FIRST($a$) = {$a$}                      if $a \in T$
FIRST($\varepsilon$) = {$\varepsilon$}
FIRST($A$) = $\cup_{A\to\alpha}$ FIRST($\alpha$)          for $A\to\alpha \in P$
FIRST($X_1X_2\ldots X_k$) =
    if for all $j$ = 1, …, $i$-1 : $\varepsilon \in$ FIRST($X_j$) then
        add non-$\varepsilon$ in FIRST($X_i$) to FIRST($X_1X_2\ldots X_k$)
    if for all $j$ = 1, …, $k$ : $\varepsilon \in$ FIRST($X_j$) then
        add $\varepsilon$ to FIRST($X_1X_2\ldots X_k$)
```



FOLLOW (Use defs Book)

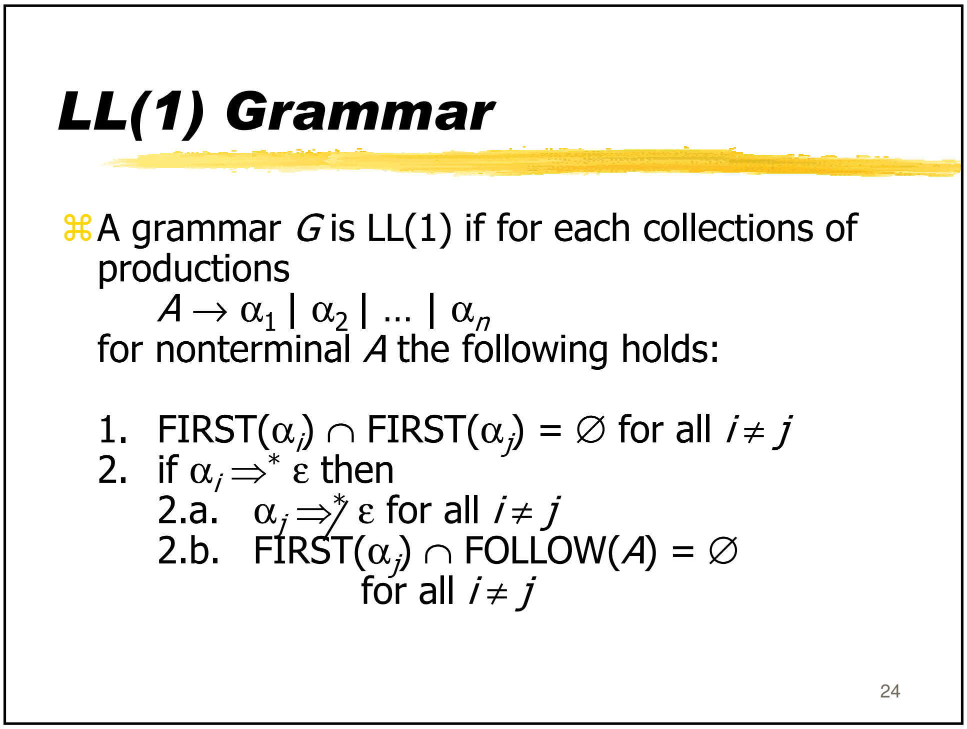
- FOLLOW(A) = the set of terminals that can immediately follow nonterminal A

```
FOLLOW($A$) =
    for all ($B \to \alpha\ A\ \beta$) $\in P$ do
        add FIRST($\beta$)\{$\varepsilon$} to FOLLOW($A$)
    for all ($B \to \alpha\ A\ \beta$) $\in P$ and $\varepsilon \in$ FIRST($\beta$) do
        add FOLLOW($B$) to FOLLOW($A$)
    for all ($B \to \alpha\ A$) $\in P$ do
        add FOLLOW($B$) to FOLLOW($A$)
    if $A$ is the start symbol $S$ then
        add $ to FOLLOW($A$)
```



EXAMPLE (From Book)

- Explain how FOLLOW works….



LL(1) Grammar

- A grammar G is LL(1) if for each collections of productions

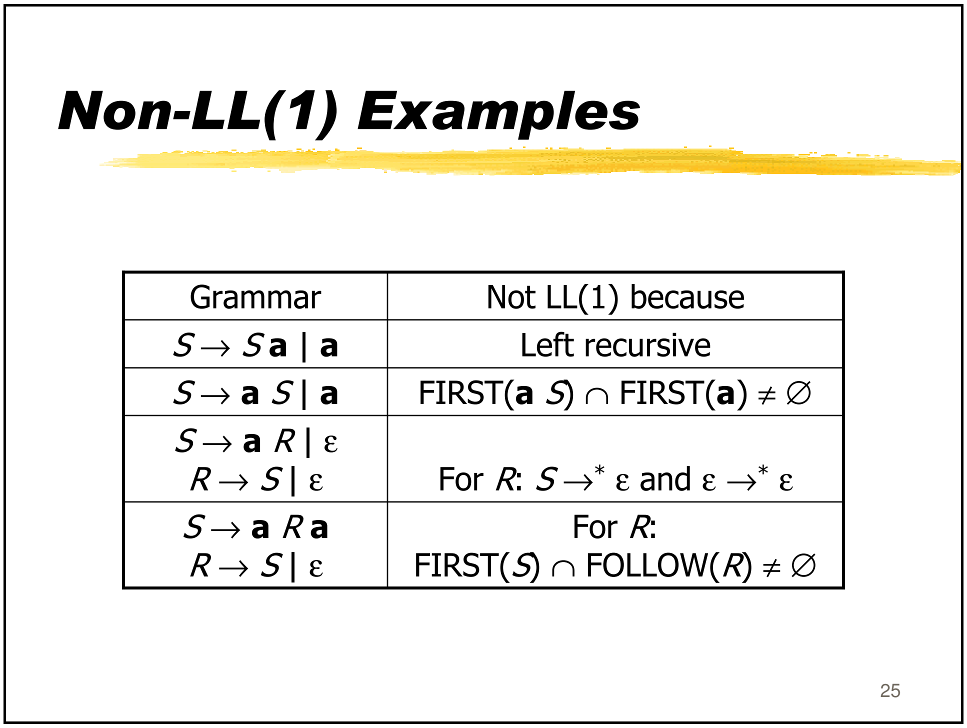
  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

  for nonterminal A the following holds:

  1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \varnothing$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then

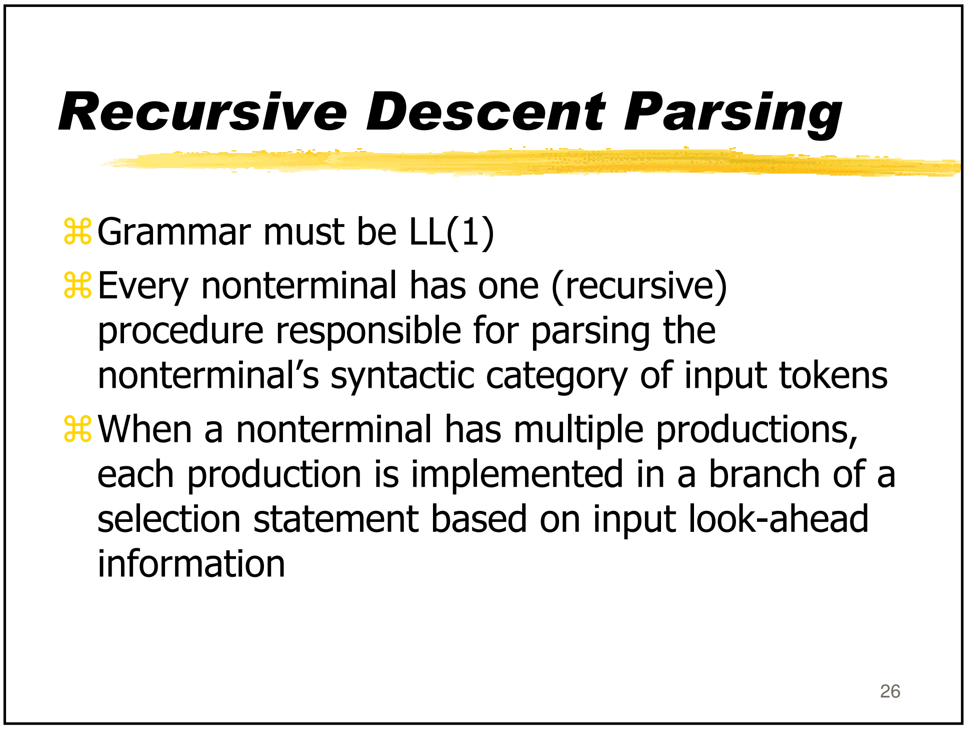
     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$

     2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \varnothing$ for all $i \neq j$



Non-LL(1) Examples

Grammar	Not LL(1) because
$S \rightarrow S\,\mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\ S \mid \mathbf{a}$	FIRST($\mathbf{a}\ S$) $\cap$ FIRST($\mathbf{a}$) $\neq \emptyset$
$S \rightarrow \mathbf{a}\ R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \rightarrow^{*} \varepsilon$ and $\varepsilon \rightarrow^{*} \varepsilon$
$S \rightarrow \mathbf{a}\ R\,\mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \emptyset$



Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal’s syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

Using FIRST and FOLLOW to Write a Rec. Descent Parser

expr → *term rest*
rest → + *term rest*
| - *term rest*
| ε
term → **id**

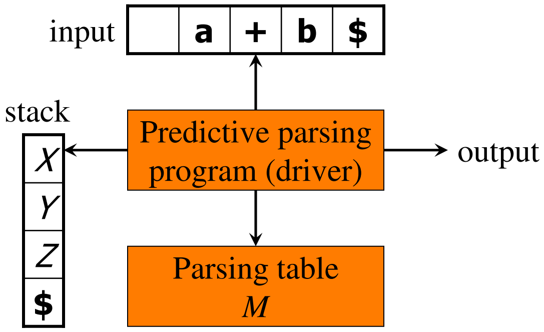
```
procedure rest();
begin
  if lookahead in FIRST(+ term rest) then
    match('+'); term(); rest()
  else if lookahead in FIRST(- term rest) then
    match('-'); term(); rest()
  else if lookahead in FOLLOW(rest) then
    return
  else error()
end;
```

FIRST(**+** *term rest*) = { **+** }
FIRST(**-** *term rest*) = { **-** }
FOLLOW(*rest*) = { **$** }

27

Non-Recursive Predictive Parsing

- Given an LL(1) grammar $G=(N,T,P,S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a driver program with a stack



28

Constructing a Predictive Parsing Table

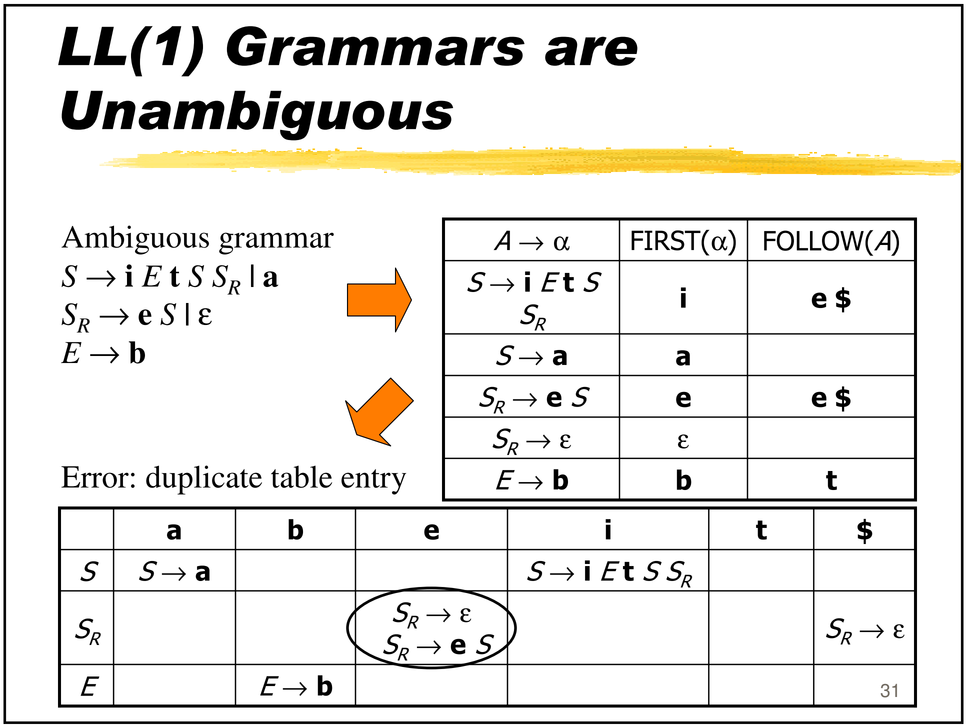
```
**for** each production $A \rightarrow \alpha$ **do**
    **for** each $a \in$ FIRST($\alpha$) **do**
        add $A \rightarrow \alpha$ to $M[A,a]$
    **enddo**
    **if** $\varepsilon \in$ FIRST($\alpha$) **then**
        **for** each $b \in$ FOLLOW($A$) **do**
            add $A \rightarrow \alpha$ to $M[A,b]$
        **enddo**
    **endif**
**enddo**
Mark each undefined entry in $M$ error
```

Example Table

$E \rightarrow T\,E_R$
$E_R \rightarrow \mathbf{+}\;T\,E_R \mid \varepsilon$
$T \rightarrow F\,T_R$
$T_R \rightarrow \mathbf{*}\;F\,T_R \mid \varepsilon$
$F \rightarrow (\;E\;) \mid \mathbf{id}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\,E_R$	**(id**	**$)**
$E_R \rightarrow \mathbf{+}\;T\,E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	
$T \rightarrow F\,T_R$	**(id**	**+ $)**
$T_R \rightarrow \mathbf{*}\;F\,T_R$	*****	**+ $)**
$T_R \rightarrow \varepsilon$	ε	
$F \rightarrow \mathbf{(}\;E\;\mathbf{)}$	**(**	*** + $)**
$F \rightarrow \mathbf{id}$	**id**	

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\,E_R$			$E \rightarrow T\,E_R$		
E_R		$E_R \rightarrow \mathbf{+}\;T\,E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\,T_R$			$T \rightarrow F\,T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \mathbf{*}\;F\,T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow \mathbf{(}\;E\;\mathbf{)}$		



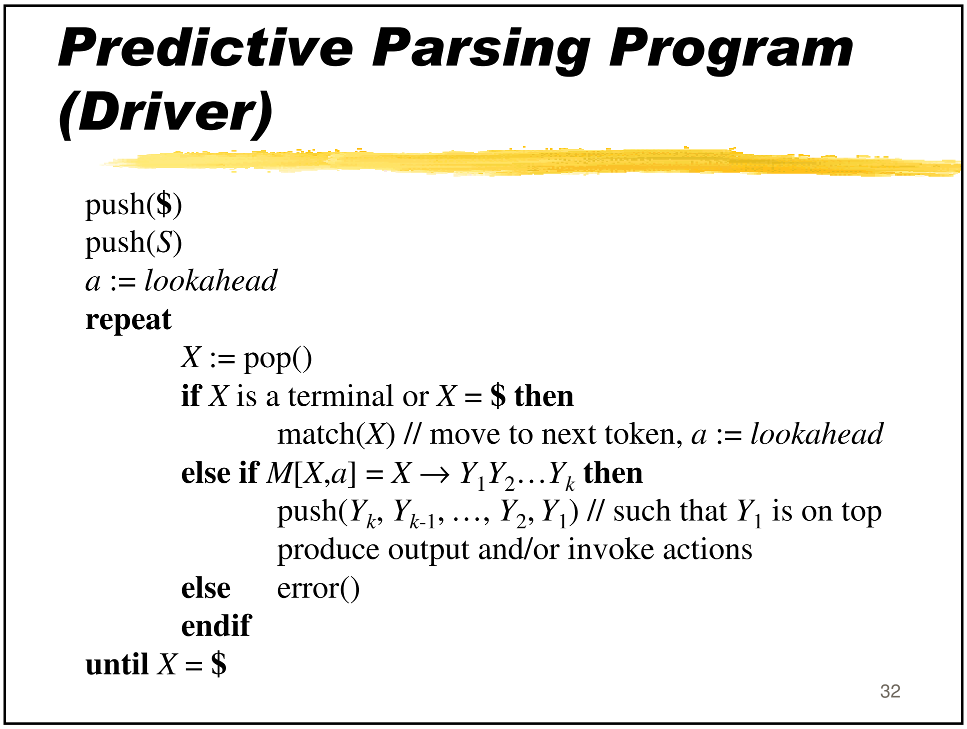
LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow$ **i** E **t** S S_R | **a**
$S_R \rightarrow$ **e** S | ε
$E \rightarrow$ **b**

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow$ **i** E **t** S S_R	**i**	**e $**
$S \rightarrow$ **a**	**a**	
$S_R \rightarrow$ **e** S	**e**	**e $**
$S_R \rightarrow \varepsilon$	ε	
$E \rightarrow$ **b**	**b**	**t**

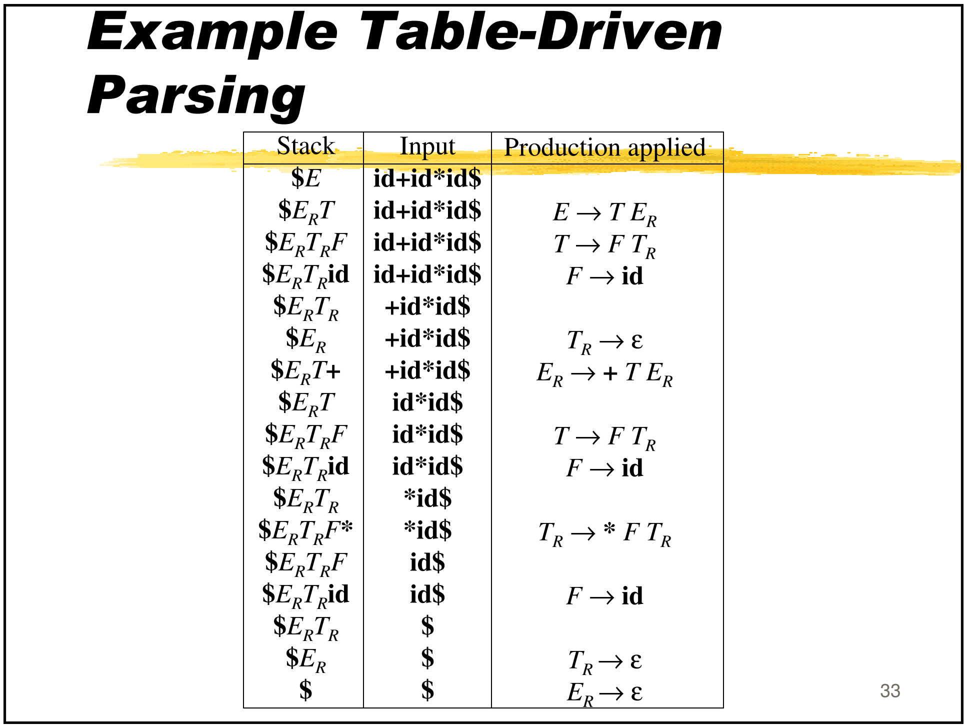
Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow$ **a**			$S \rightarrow$ **i** E **t** S S_R		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow$ **e** S			$S_R \rightarrow \varepsilon$
E		$E \rightarrow$ **b**				



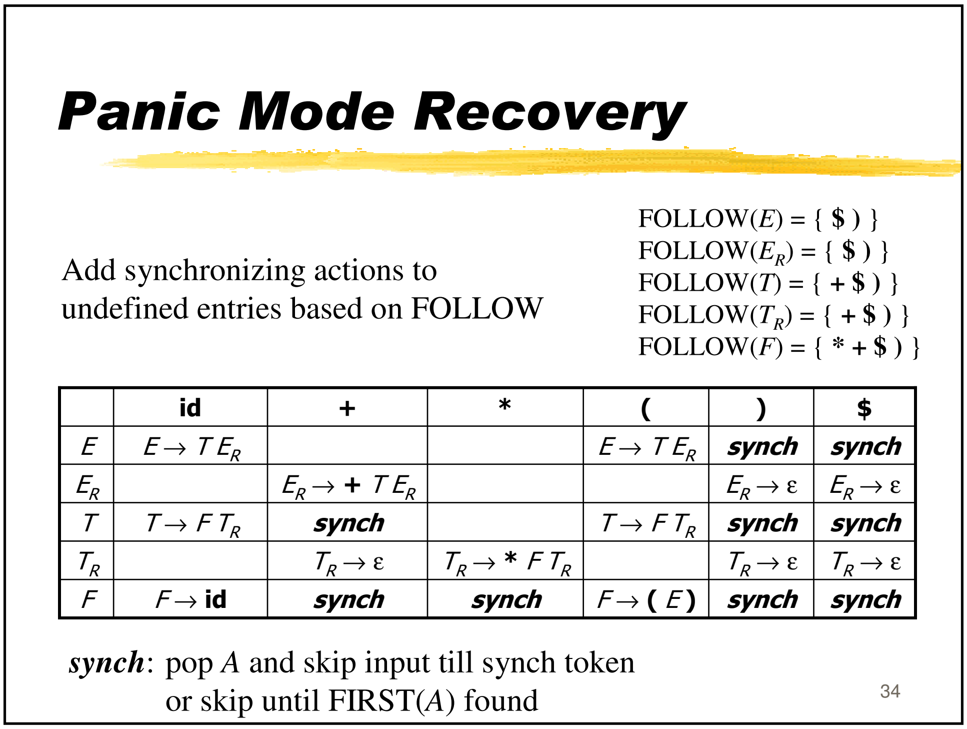
Predictive Parsing Program (Driver)

```
push($)
push($S$)
$a$ := lookahead
repeat
        $X$ := pop()
        if $X$ is a terminal or $X$ = $ then
                match($X$) // move to next token, $a$ := lookahead
        else if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ then
                push($Y_k, Y_{k-1}, \ldots, Y_2, Y_1$) // such that $Y_1$ is on top
                produce output and/or invoke actions
        else    error()
        endif
until $X$ = $
```



Example Table-Driven Parsing

Stack	Input	Production applied
$\$E$	**id+id*id$**	
$\$E_RT$	**id+id*id$**	$E \rightarrow T\,E_R$
$\$E_RT_RF$	**id+id*id$**	$T \rightarrow F\,T_R$
$\$E_RT_R$**id**	**id+id*id$**	$F \rightarrow$ **id**
$\$E_RT_R$	**+id*id$**	
$\$E_R$	**+id*id$**	$T_R \rightarrow \varepsilon$
$\$E_RT$**+**	**+id*id$**	$E_R \rightarrow$ **+** $T\,E_R$
$\$E_RT$	**id*id$**	
$\$E_RT_RF$	**id*id$**	$T \rightarrow F\,T_R$
$\$E_RT_R$**id**	**id*id$**	$F \rightarrow$ **id**
$\$E_RT_R$	***id$**	
$\$E_RT_RF$*****	***id$**	$T_R \rightarrow$ ***** $F\,T_R$
$\$E_RT_RF$	**id$**	
$\$E_RT_R$**id**	**id$**	$F \rightarrow$ **id**
$\$E_RT_R$	**$**	
$\$E_R$	**$**	$T_R \rightarrow \varepsilon$
$	**$**	$E_R \rightarrow \varepsilon$



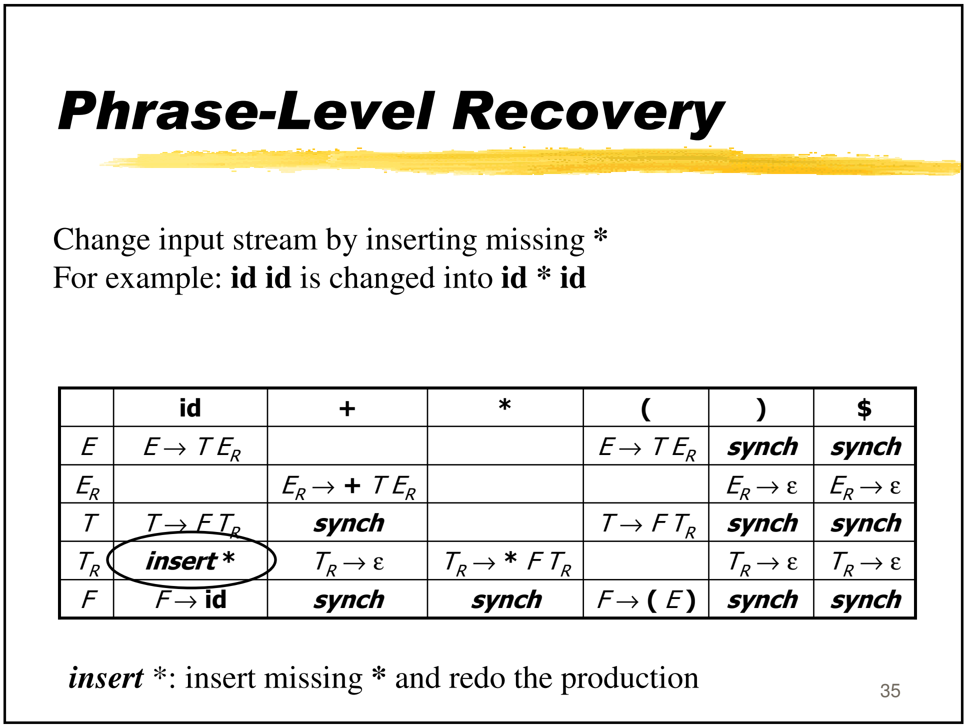
Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

FOLLOW($E$) = { **$** **)** }
FOLLOW(E_R) = { **$** **)** }
FOLLOW(T) = { **+** **$** **)** }
FOLLOW(T_R) = { **+** **$** **)** }
FOLLOW(F) = { ***** **+** **$** **)** }

	id	**+**	*****	**(**	**)**	**$**
$E$	$E \rightarrow T\,E_R$			$E \rightarrow T\,E_R$	***synch***	***synch***
E_R		$E_R \rightarrow$ **+** $T\,E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\,T_R$	***synch***		$T \rightarrow F\,T_R$	***synch***	***synch***
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow$ ***** $F\,T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow$ **(** $E$ **)**	***synch***	***synch***

synch: pop A and skip input till synch token
or skip until FIRST(A) found



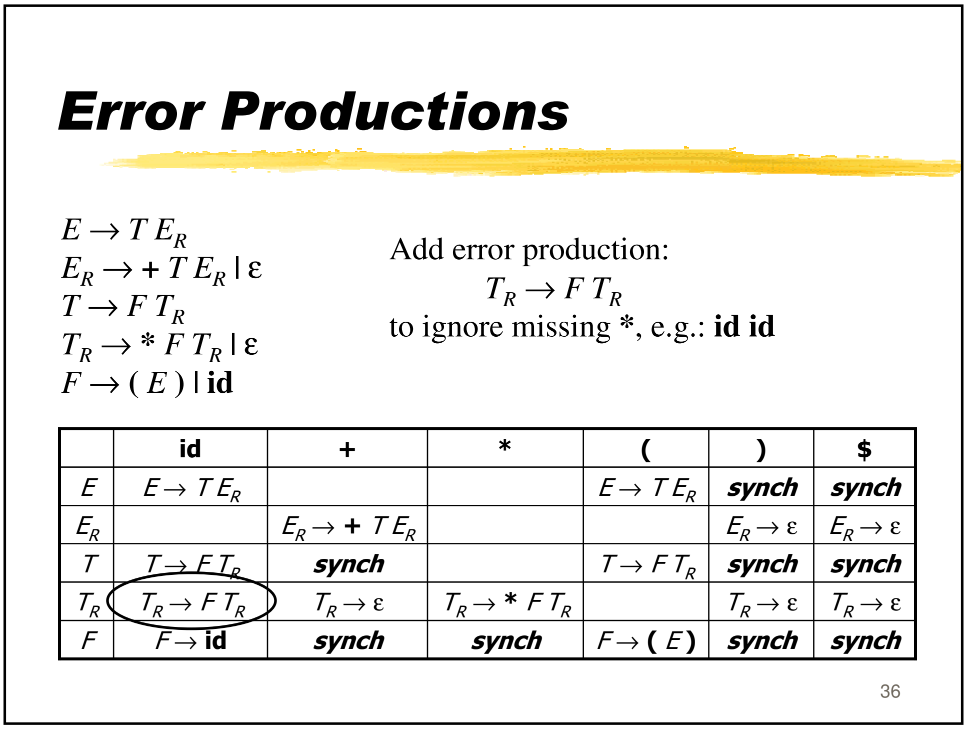
Phrase-Level Recovery

Change input stream by inserting missing *
For example: **id id** is changed into **id * id**

	id	+	*	(	)	$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	***synch***	***synch***
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$	***synch***		$T \rightarrow F T_R$	***synch***	***synch***
T_R	***insert ****	$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (E)$	***synch***	***synch***

insert *: insert missing * and redo the production

35



Error Productions

$E \rightarrow T E_R$
$E_R \rightarrow + T E_R \mid \varepsilon$
$T \rightarrow F T_R$
$T_R \rightarrow * F T_R \mid \varepsilon$
$F \rightarrow (E) \mid$ **id**

Add error production:
$T_R \rightarrow F T_R$
to ignore missing *, e.g.: **id id**

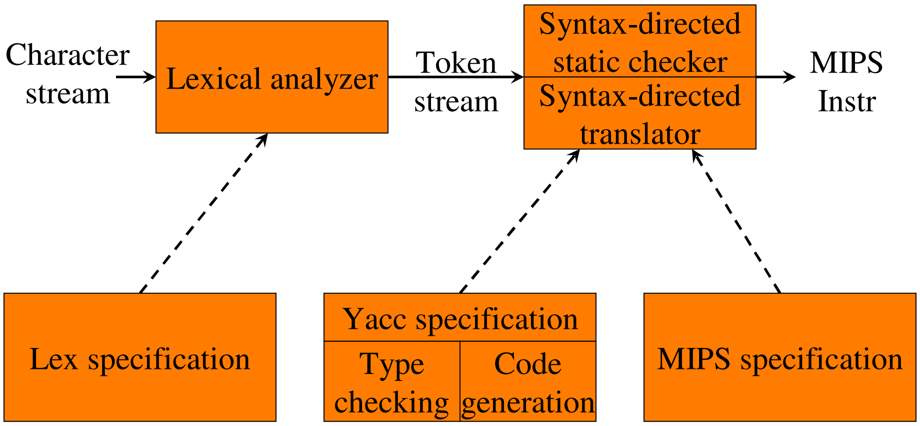
	id	+	*	(	)	$
E	$E \rightarrow T E_R$			$E \rightarrow T E_R$	***synch***	***synch***
E_R		$E_R \rightarrow + T E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F T_R$	***synch***		$T \rightarrow F T_R$	***synch***	***synch***
T_R	$T_R \rightarrow F T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow * F T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow (E)$	***synch***	***synch***

36

Static Checking and Type Systems

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

The Structure of our Compiler Revisited



Static versus Dynamic Checking

- *Static checking*: the compiler enforces programming language’s *static semantics*, which are checked at compile time
- *Runtime checking*: *dynamic semantics* are checked at run time by special code generated by the compiler

Static Checking

- Typical examples of static checking are
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks

Type Checks, Overloading, Coercion, and Polymorphism

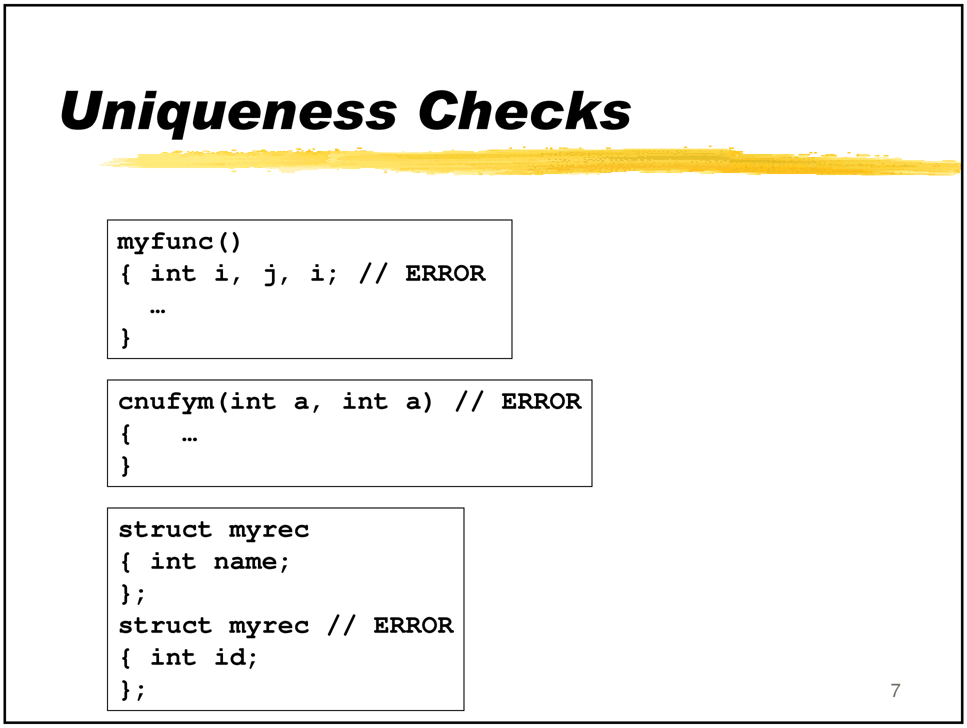
```
int op(int), op(float);
int f(float);
int a, c[10], d;

d = c+d;          // FAIL

*d = a;           // FAIL

a = op(d);        // OK: overloading (C++)

a = f(d);         // OK: coersion

vector<int> v;    // OK: template instantiation
```

Flow-of-Control Checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

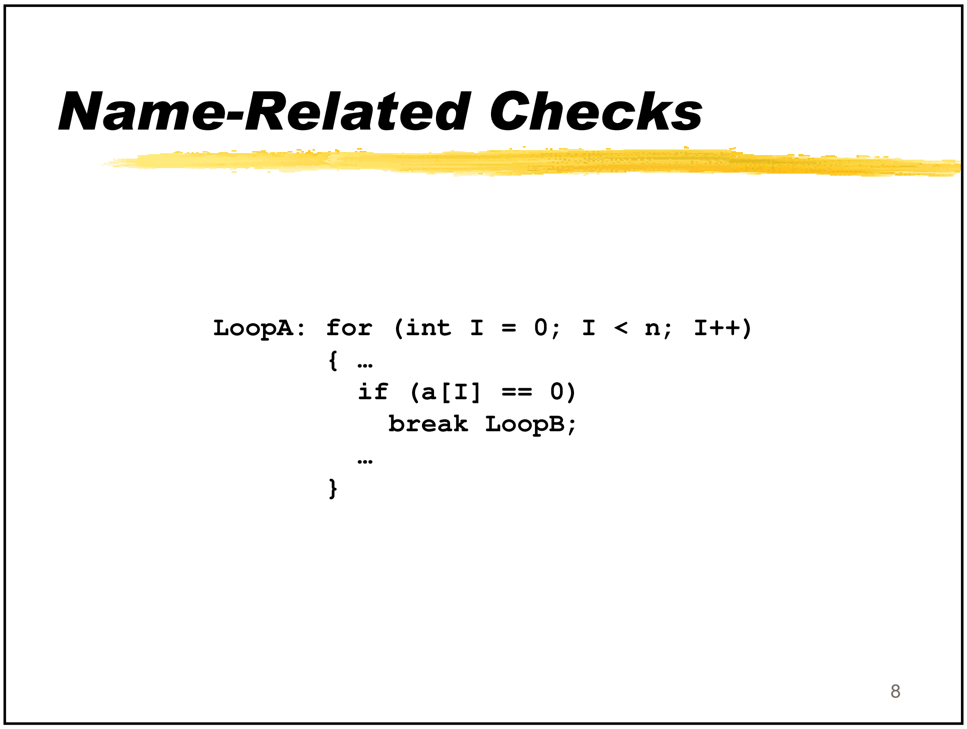


Uniqueness Checks

```
myfunc()
{ int i, j, i; // ERROR
  …
}
```

```
cnufym(int a, int a) // ERROR
{    …
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```



Name-Related Checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB;
         …
       }
```

One-Pass versus Multi-Pass Static Checking

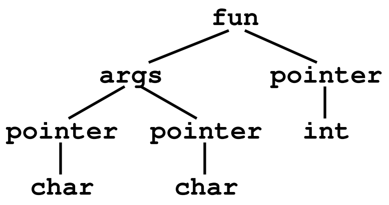
- *One-pass compiler*: static checking for C, Pascal, Fortran, and many other languages can be performed in one pass while at the same time intermediate code is generated
- *Multi-pass compiler*: static checking for Ada, Java, and C# is performed in a separate phase, sometimes requiring traversing the syntax tree multiple times

Type Expressions

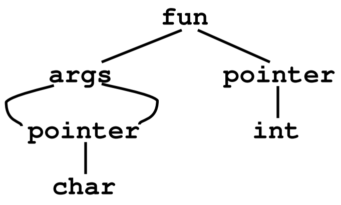
- *Type expressions* are used in declarations and type casts to define or refer to a type
  - *Primitive types*, such as **`int`** and **`bool`**
  - *Type constructors*, such as pointer-to, array-of, records and classes, templates, and functions
  - *Type names*, such as typedefs in C and named types in Pascal, refer to type expressions

Graph Representations for Type Expressions

```
int *fun(char*,char*)
```



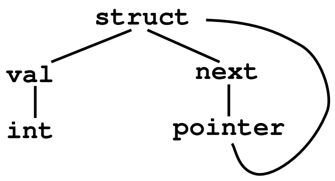
Tree forms



DAGs

Cyclic Graph Representations

```
struct Node
{ int val;
  struct Node *next;
};
```



Cyclic graph

Name Equivalence

- Each type name is a distinct type, even when the type expressions the names refer to are the same
- Types are identical only if names match
- Used by Pascal (inconsistently)

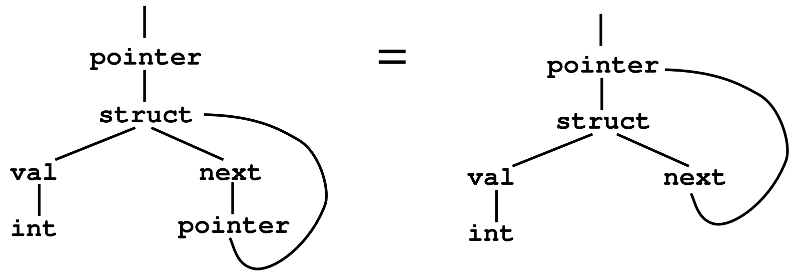
```
type link = ^node;
var next : link;
    last : link;
       p : ^node;
    q, r : ^node;
```

With name equivalence in Pascal:

```
 p ≠ next
 p ≠ last
p = q = r
next = last
```

Structural Equivalence of Type Expressions

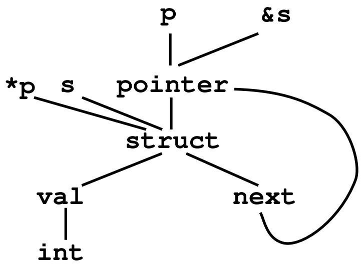
- Two types are the same if they are structurally identical
- Used in C, Java, C#



Structural Equivalence of Type Expressions (cont'd)

⌘Two structurally equivalent type expressions have the same pointer address when constructing graphs by sharing nodes

```
struct Node
{ int val;
  struct Node *next;
};
struct Node s, *p;

… p = &s; // OK
… *p = s; // OK
```



Type Systems

⌘A *type system* defines a set of types and rules to assign types to programming language constructs

⌘Informal type system rules, for example "*if both operands of addition are of type integer, then the result is of type integer*"

⌘Formal type system rules: Post system

A Simple Language Example

$P \rightarrow D\ ;\ S$
$D \rightarrow D\ ;\ D$
$\quad \mid \mathbf{id} : T$
$T \rightarrow \mathbf{boolean}$
$\quad \mid \mathbf{char}$
$\quad \mid \mathbf{integer}$
$\quad \mid \mathbf{array}\ [\ \mathbf{num}\ ]\ \mathbf{of}\ T$
$\quad \mid \hat{}\ T$
$S \rightarrow \mathbf{id} := E$
$\quad \mid \mathbf{if}\ E\ \mathbf{then}\ S$
$\quad \mid \mathbf{while}\ E\ \mathbf{do}\ S$
$\quad \mid S\ ;\ S$

$E \rightarrow \mathbf{true}$
$\quad \mid \mathbf{false}$
$\quad \mid \mathbf{literal}$
$\quad \mid \mathbf{num}$
$\quad \mid \mathbf{id}$
$\quad \mid E\ \mathbf{and}\ E$
$\quad \mid E\ \mathbf{mod}\ E$
$\quad \mid E\ [\ E\ ]$
$\quad \mid E\ \hat{}$

Simple Language Example: Declarations

$D \rightarrow \mathbf{id} : T$	$\{\ \mathit{addtype}(\mathbf{id}.\mathrm{entry}, T.\mathrm{type})\ \}$
$T \rightarrow \mathbf{boolean}$	$\{\ T.\mathrm{type} := \mathit{boolean}\ \}$
$T \rightarrow \mathbf{char}$	$\{\ T.\mathrm{type} := \mathit{char}\ \}$
$T \rightarrow \mathbf{integer}$	$\{\ T.\mathrm{type} := \mathit{integer}\ \}$
$T \rightarrow \mathbf{array}\ [\ \mathbf{num}\ ]\ \mathbf{of}\ T_1$	$\{\ T.\mathrm{type} := \mathit{array}(1..\mathbf{num}.\mathrm{val}, T_1.\mathrm{type})\ \}$
$T \rightarrow\ \hat{}\ T_1$	$\{\ T.\mathrm{type} := \mathit{pointer}(T_1)$

Simple Language Example: Statements

$S \rightarrow$ **id :=** E	{ S.type := **if id**.type = E.type **then** *void* **else** *type_error* }
$S \rightarrow$ **if** E **then** S_1	{ S.type := **if** E.type = *boolean* **then** S_1.type **else** *type_error* }
$S \rightarrow$ **while** E **do** S_1	{ S.type := **if** E.type = *boolean* **then** S_1.type **else** *type_error* }
$S \rightarrow S_1$ **;** S_2	{ S.type := **if** S_1.type = *void* **and** S_2.type = *void* **then** *void* **else** *type_error* }

Simple Language Example: Expressions

$E \rightarrow$ **true**	{ E.type = *boolean* }
$E \rightarrow$ **false**	{ E.type = *boolean* }
$E \rightarrow$ **literal**	{ E.type = *char* }
$E \rightarrow$ **num**	{ E.type = *integer* }
$E \rightarrow$ **id**	{ E.type = *lookup*(**id**.entry) }
…	

Simple Language Example: Expressions (cont'd)

$E \rightarrow E_1$ **and** E_2 { E.type := **if** E_1.type = *boolean* **and**
E_2.type = *boolean*
then *boolean* **else** *type_error* }

$E \rightarrow E_1$ **mod** E_2 { E.type := **if** E_1.type = *integer* **and**
E_2.type = *integer*
then *integer* **else** *type_error* }

$E \rightarrow E_1$ **[** E_2 **]** { E.type := **if** E_1.type = *array*(s, t) **and**
E_2.type = *integer*
then t **else** *type_error* }

$E \rightarrow E_1$ **^** { E.type := **if** E_1.type = *pointer*(t)
then t **else** *type_error* }

Simple Language Example: Adding Functions

$T \rightarrow T_1$ **->** T_2 { T.type := *function*(T_1.type, T_2.type) }

$E \rightarrow E_1$ **(** E_2 **)** { E.type := **if** E_1.type = *function*(s, t) **and**
E_2.type = s
then t **else** *type_error* }

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
  v := 1;
```

Constructing Type Graphs in Yacc

`Type *mkint()`	construct int node if not already constructed
`Type *mkarr(Type*,int)`	construct array-of-type node if not already constructed
`Type *mkptr(Type*)`	construct pointer-of-type node if not already constructed

Syntax-Directed Definitions for Constructing Type Graphs in Yacc

```
%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <int> NUM
%type <typ> type
%%
decl : type ID                { addtype($2, $1); }
     | type ID `[' NUM `]'    { addtype($2, mkarr($1, $4)); }
     ;
type : INT                    { $$ = mkint(); }
     | type `*'               { $$ = mkptr($1); }
     | /* empty */            { $$ = mkint(); }
     ;
```

Syntax-Directed Definitions for Type Checking in Yacc

```
%{
enum Types {Tint, Tfloat, Tpointer, Tarray, … };
typedef struct Type
{ enum Types type;
  struct Type *child;
} Type;
%}
%union
{ Type *typ;
}
%type <typ> expr
%%
expr : expr `+' expr { if ($1.type != Tint
                        || $3.type != Tint)
                         semerror("non-int operands in +");
                       $$ = mkint();
                       emit(iadd);
                     }
```

Type Conversion and Coercion

- *Consider x+i, where x:=real and i:=int*
  - *x i* ***inttoreal*** *real+*
- *Type conversion* is explicit, for example using type casts
- *Type coercion* is implicitly performed by the compiler
- Both require a type system to check and infer types for (sub)expressions

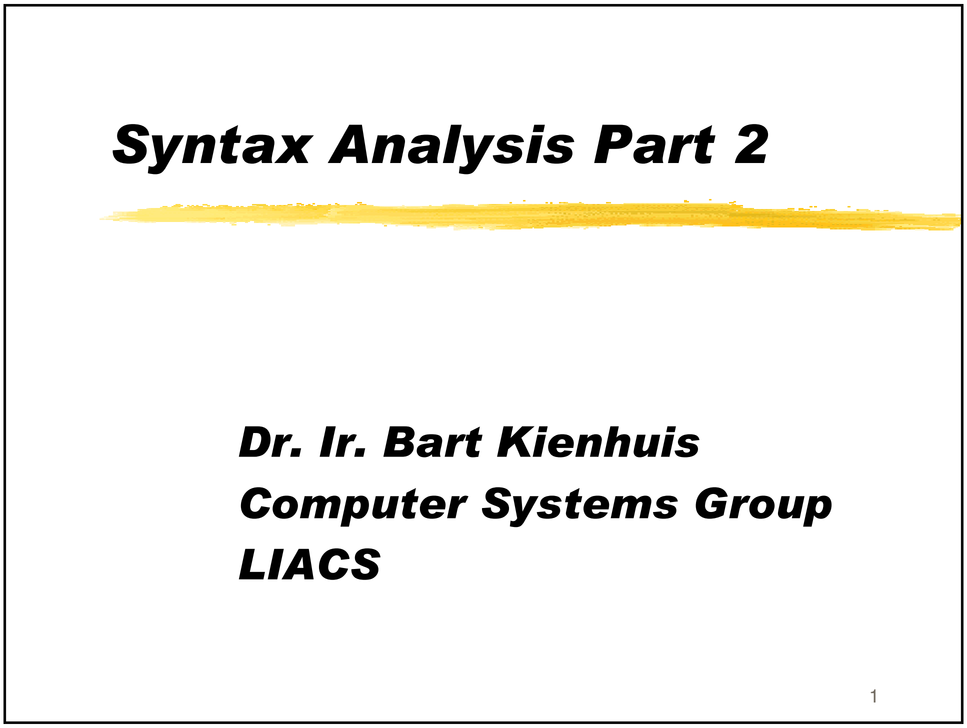
Syntax-Directed Definitions for Type Coercion in Yacc

```
%{ … %}
%%
expr : expr ‘+’ expr
     { if ($1.type == Tint && $3.type == Tint)
       { $$ = mkint(); emit(iadd);
       }
       else if ($1.type == Tfloat && $3.type == Tfloat)
       { $$ = mkfloat(); emit(fadd);
       }
       else if ($1.type == Tfloat && $3.type == Tint)
       { $$ = mkfloat(); emit(i2f); emit(fadd);
       }
       else if ($1.type == Tint && $3.type == Tfloat)
       { $$ = mkfloat(); emit(swap); emit(i2f); emit(fadd);
       }
       else semerror(“type error in +”);
         $$ = mkint();
     }
```

Syntax-Directed Definitions for L-Values and R-Values in Yacc

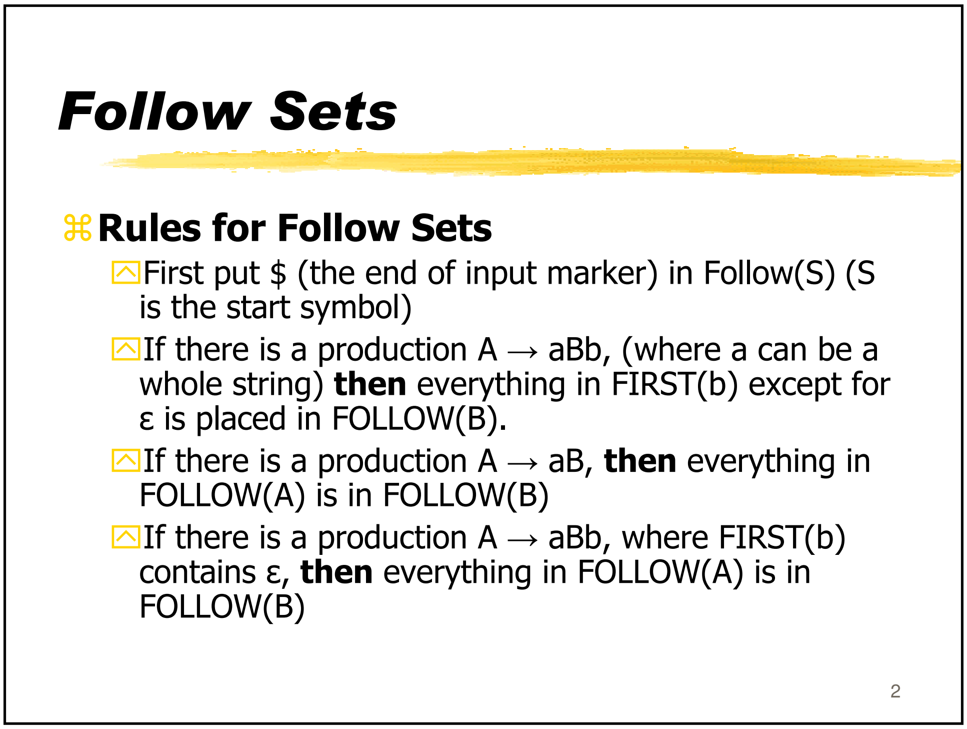
```
%{
typedef struct Node
{ Type *typ;
  int islval;
} Node;
%}
%union
{ Node *rec;
}
%type <rec> expr
%%
```

```
expr : expr ‘+’ expr
     { if ($1.typ->type != Tint
         || $3.typ->type != Tint)
          semerror(“non-int operands in +”);
       $$.typ = mkint();
       $$.islval = FALSE;
       emit(…);
     }
     | expr ‘=’ expr
     { if (!$1.islval || $1.typ != $3.typ)
          semerror(“invalid assignment”);
       $$.typ = $1.typ; $$.islval = FALSE;
       emit(…);
     }
     | ID
     { $$.typ = lookup($1);
       $$.islval = TRUE;
       emit(…);
     }
```



Syntax Analysis Part 2

Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS



Follow Sets

- **Rules for Follow Sets**
  - First put $ (the end of input marker) in Follow(S) (S is the start symbol)
  - If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
  - If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
  - If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Bottom-Up Parsing

- LR methods (Left-to-right, Reftmost derivation)
  - LR(0), SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

Operator-Precedence Parsing

- Special case of shift-reduce parsing
- We will not further discuss (you can skip textbook section 4.6)

Shift-Reduce Parsing

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b\ c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

These match production's right-hand sides

Reducing a sentence:

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b\ c\ d\ e}$

$\mathbf{a}\ \underline{A\ \mathbf{b\ c}}\ \mathbf{d\ e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

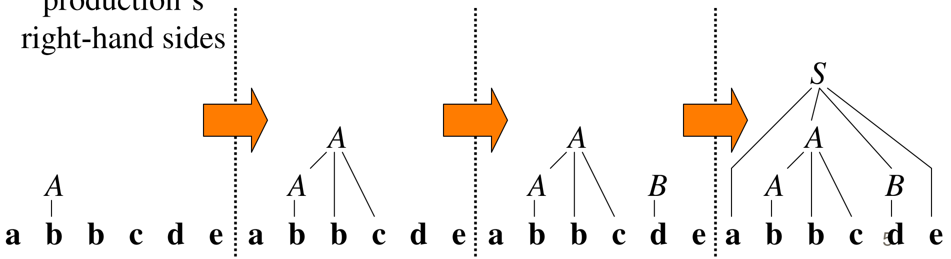
Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d\ e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b\ c\ d\ e}$

$\Rightarrow_{rm} \mathbf{a\ b\ b\ c\ d\ e}$



5

Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b\ c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b\ c\ d\ e}$

$\mathbf{a}\ \underline{A\ \mathbf{b\ c}}\ \mathbf{d\ e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b\ c\ d\ e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c\ d\ e}$

$\mathbf{a}\ A\ A\ \mathbf{e}$

… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

6

Stack Implementation of Shift-Reduce Parsing

Grammar:
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow \mathbf{id}$

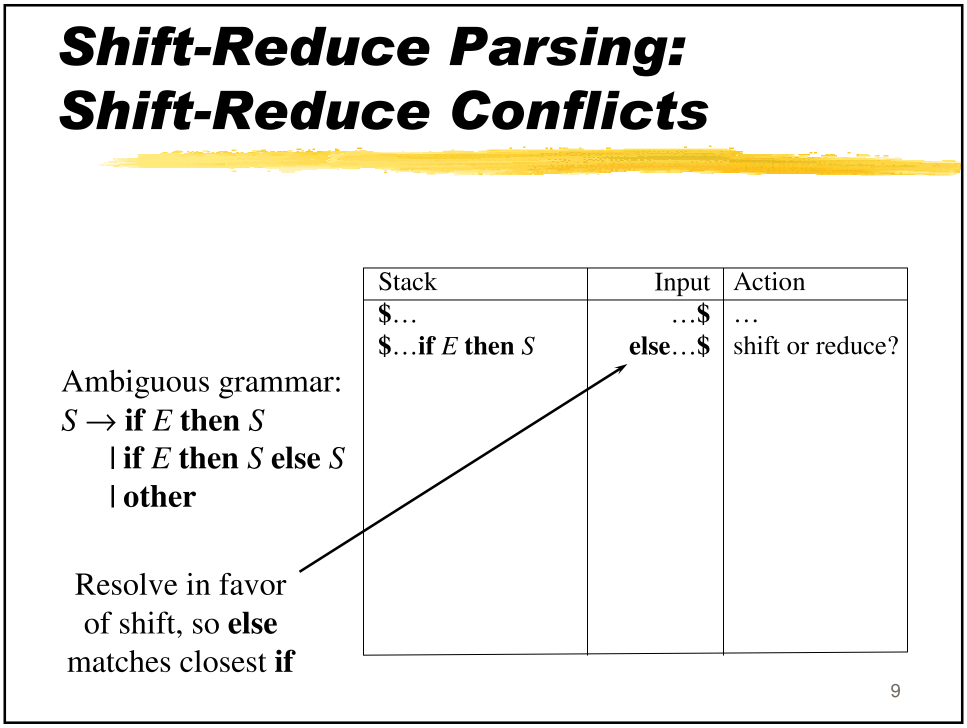
Stack	Input	Action
$\$$	**id+id*id$**	shift
$\$\underline{\mathbf{id}}$	**+id*id$**	reduce $E \rightarrow \mathbf{id}$
$\$E$	**+id*id$**	shift
$\$E\mathbf{+}$	**id*id$**	shift
$\$E\mathbf{+}\underline{\mathbf{id}}$	***id$**	reduce $E \rightarrow \mathbf{id}$
$\$E\mathbf{+}E$	***id$**	shift (or reduce?)
$\$E\mathbf{+}E\mathbf{*}$	**id$**	shift
$\$E\mathbf{+}E\mathbf{*}\underline{\mathbf{id}}$	**$**	reduce $E \rightarrow \mathbf{id}$
$\$E\mathbf{+}\underline{E\mathbf{*}E}$	**$**	reduce $E \rightarrow E * E$
$\$\underline{E\mathbf{+}E}$	**$**	reduce $E \rightarrow E + E$
$\$E$	**$**	accept

How to resolve conflicts?

Find handles to reduce

Conflicts

- Shift-reduce and reduce-reduce conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

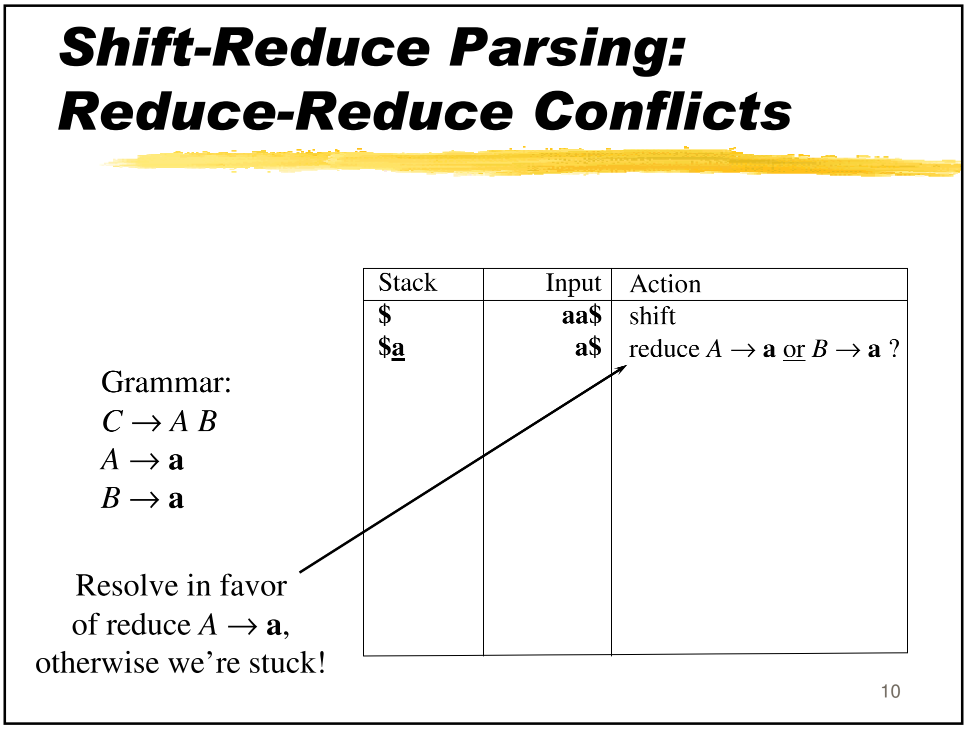


Shift-Reduce Parsing: Shift-Reduce Conflicts

Ambiguous grammar:
$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **other**

Stack	Input	Action
\$…	…\$	…
\$…**if** E **then** S	**else**…\$	shift or reduce?

Resolve in favor of shift, so **else** matches closest **if**



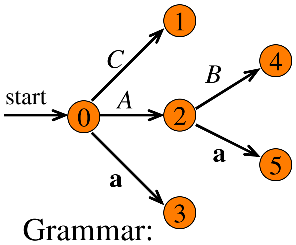
Shift-Reduce Parsing: Reduce-Reduce Conflicts

Grammar:
$C \rightarrow A\ B$
$A \rightarrow$ **a**
$B \rightarrow$ **a**

Stack	Input	Action
\$	**aa**\$	shift
\$<u>**a**</u>	**a**\$	reduce $A \rightarrow$ **a** <u>or</u> $B \rightarrow$ **a** ?

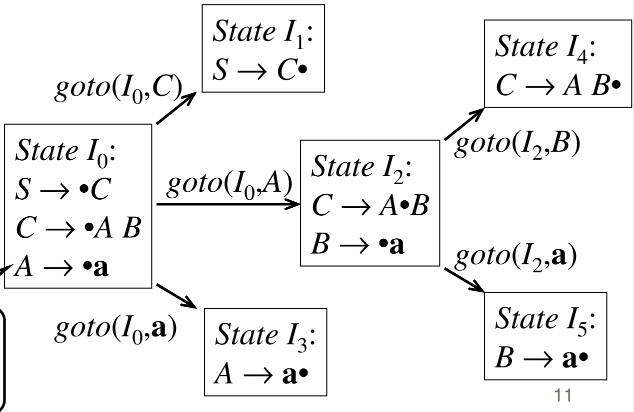
Resolve in favor of reduce $A \rightarrow$ **a**, otherwise we're stuck!

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions



Grammar:
$S \to C$
$C \to A\ B$
$A \to \mathbf{a}$
$B \to \mathbf{a}$

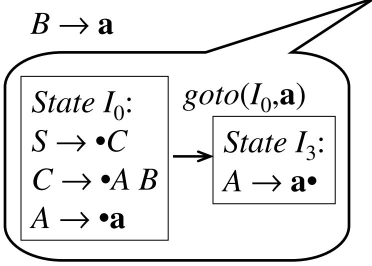
Can only reduce $A \to \mathbf{a}$ (not $B \to \mathbf{a}$)



DFA for Shift/Reduce Decisions

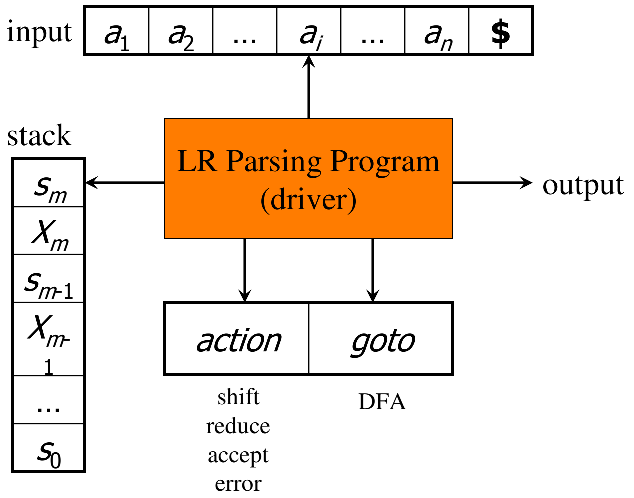
The states of the DFA are used to determine if a handle is on top of the stack

Grammar:
$S \to C$
$C \to A\ B$
$A \to \mathbf{a}$
$B \to \mathbf{a}$



Stack	Input	Action
$ 0	**aa$**	start in state 0
$ 0	**aa$**	shift (and goto state 3)
$ 0 **a** 3	**a$**	reduce *A* → **a** (goto 2)
$ 0 *A* 2	**a$**	shift (goto 5)
$ 0 *A* 2 **a** 5	$	reduce *B* → **a** (goto 4)
$ 0 *A* 2 *B* 4	$	reduce *C* → *AB* (goto 1)
$ 0 *C* 1	$	reduce *S* → *C*
$ 0 *S* 1	$	accept

Model of an LR Parser



LR Parsing

Configuration (= LR parser state):

$$\underbrace{(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m,}_{\text{stack}}\ \underbrace{a_i\, a_{i+1} \ldots a_n\, \$)}_{\text{input}}$$

If *action*$[s_m,a_i]$ = shift s, then push a_i, push s, and advance input:

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m\, a_i\, s,\ \ a_{i+1} \ldots a_n\, \$)$$

If *action*$[s_m,a_i]$ = reduce A → β and *goto*$[s_{m-r},A]$ = s with r=|β| then pop 2r symbols, push A, and push s:

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_{m-r}\, s_{m-r}\, A\, s,\ \ a_i\, a_{i+1} \ldots a_n\, \$)$$

If *action*$[s_m,a_i]$ = accept, then stop

If *action*$[s_m,a_i]$ = error, then attempt recovery

Example LR Parse Table

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**

	action						*goto*		
state	**id**	**+**	*****	**(**	**)**	**$**	*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			

Shift & goto 5

Reduce by production #1

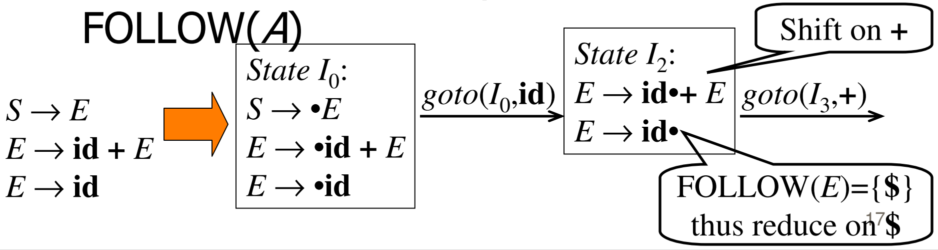
Example LR Parsing

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**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

SLR Grammars

- SLR (Simple LR): a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions *A*→α on symbols in FOLLOW(*A*)



$S \rightarrow E$
$E \rightarrow \textbf{id} + E$
$E \rightarrow \textbf{id}$
State I_0:
$S \rightarrow \bullet E$
$E \rightarrow \bullet\textbf{id} + E$
$E \rightarrow \bullet\textbf{id}$
$goto(I_0,\textbf{id})$
State I_2:
$E \rightarrow \textbf{id}\bullet + E$
$E \rightarrow \textbf{id}\bullet$
$goto(I_3,+)$
Shift on +
FOLLOW(E)={\$}
thus reduce on \$

SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \rightarrow E$
2. $E \rightarrow \textbf{id} + E$
3. $E \rightarrow \textbf{id}$

	id	**+**	**$**	*E*
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on +

FOLLOW(E)={\$}
thus reduce on \$

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - *Closure operation* to construct LR(0) items
  - *Goto operation* to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

LR(0) Items of a Grammar

- An *LR(0) item* of a grammar *G* is a production of *G* with a • at some position of the right-hand side
- Thus, a production

  $A \rightarrow X\ Y\ Z$

  has four items:

  $[A \rightarrow \bullet\ X\ Y\ Z]$

  $[A \rightarrow X \bullet\ Y\ Z]$

  $[A \rightarrow X\ Y \bullet\ Z]$

  $[A \rightarrow X\ Y\ Z \bullet]$
- Note that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

Constructing the set of LR(0) Items of a Grammar

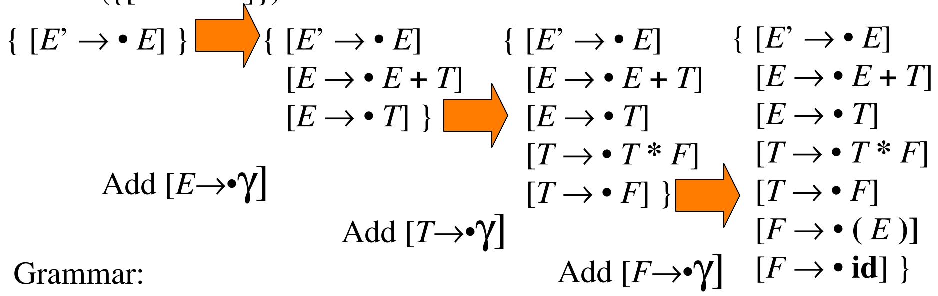
1. The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = closure(\{[S' \rightarrow \bullet S]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \emptyset$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

The Closure Operation for LR(0) Items

1. Start with $closure(I) = I$
2. If $[A \rightarrow \alpha \bullet B\beta] \in closure(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

The Closure Operation (Example)

$closure(\{[E' \to \bullet E]\}) =$



$\{\ [E' \to \bullet\, E]\ \}$

$\{\ [E' \to \bullet\, E]$
$[E \to \bullet\, E + T]$
$[E \to \bullet\, T]\ \}$

Add $[E \to \bullet\gamma]$

$\{\ [E' \to \bullet\, E]$
$[E \to \bullet\, E + T]$
$[E \to \bullet\, T]$
$[T \to \bullet\, T * F]$
$[T \to \bullet\, F]\ \}$

Add $[T \to \bullet\gamma]$

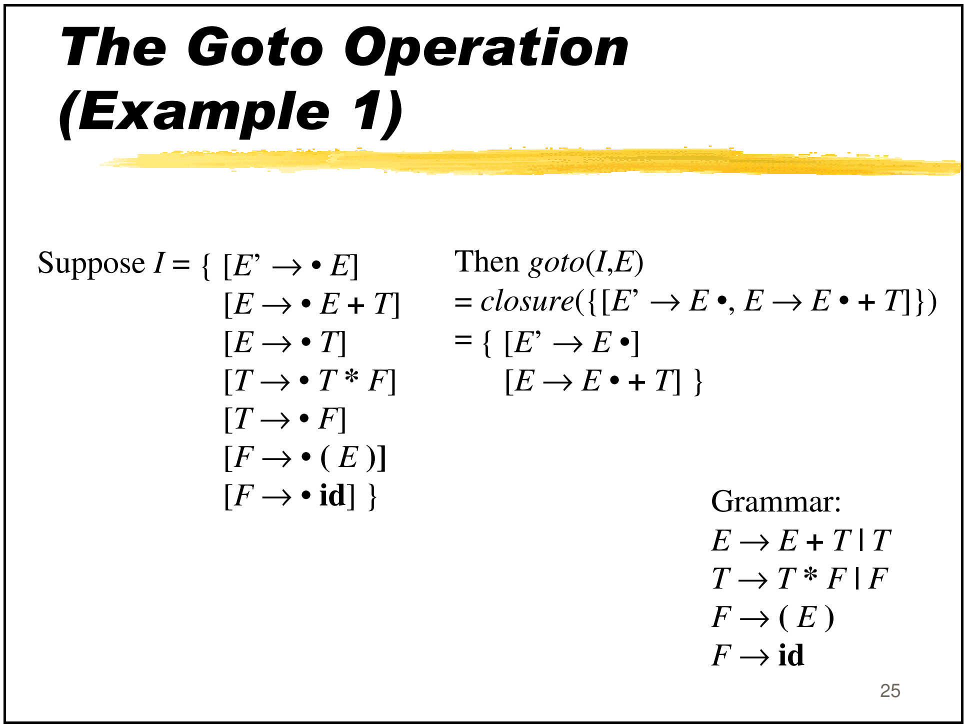
$\{\ [E' \to \bullet\, E]$
$[E \to \bullet\, E + T]$
$[E \to \bullet\, T]$
$[T \to \bullet\, T * F]$
$[T \to \bullet\, F]$
$[F \to \bullet\, (\ E\ )]$
$[F \to \bullet\, \mathbf{id}]\ \}$

Add $[F \to \bullet\gamma]$

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (\ E\ )$
$F \to \mathbf{id}$

The Goto Operation for LR(0) Items

1. For each item $[A \to \alpha \bullet X \beta] \in I$, add the set of items $closure(\{[A \to \alpha X \bullet \beta]\})$ to $goto(I,X)$ if not already there
2. Repeat step 1 until no more items can be added to $goto(I,X)$
3. Intuitively, $goto(I,X)$ is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

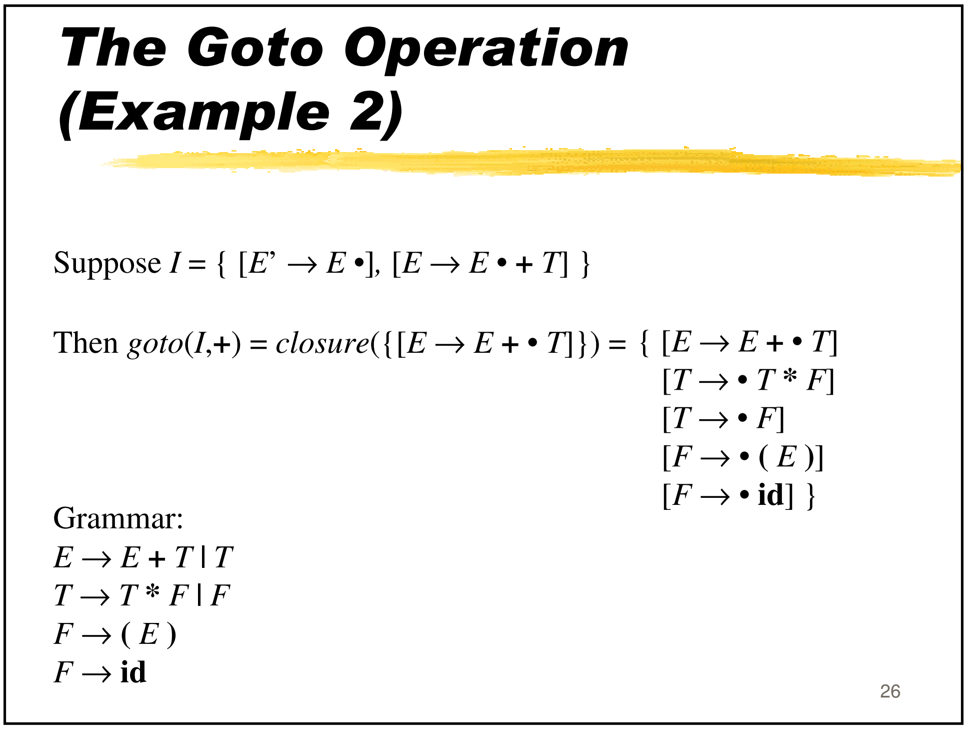


The Goto Operation (Example 1)

Suppose $I = \{\ [E' \rightarrow \bullet\, E]$
$[E \rightarrow \bullet\, E + T]$
$[E \rightarrow \bullet\, T]$
$[T \rightarrow \bullet\, T * F]$
$[T \rightarrow \bullet\, F]$
$[F \rightarrow \bullet\, (\ E\ )]$
$[F \rightarrow \bullet\, \mathbf{id}]\ \}$

Then $goto(I,E)$
$= closure(\{[E' \rightarrow E\, \bullet,\ E \rightarrow E \bullet + T]\})$
$= \{\ [E' \rightarrow E\, \bullet]$
$[E \rightarrow E \bullet + T]\ \}$

Grammar:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (\ E\ )$
$F \rightarrow \mathbf{id}$



The Goto Operation (Example 2)

Suppose $I = \{\ [E' \rightarrow E\, \bullet],\ [E \rightarrow E \bullet + T]\ \}$

Then $goto(I,+) = closure(\{[E \rightarrow E + \bullet\, T]\}) = \{\ [E \rightarrow E + \bullet\, T]$
$[T \rightarrow \bullet\, T * F]$
$[T \rightarrow \bullet\, F]$
$[F \rightarrow \bullet\, (\ E\ )]$
$[F \rightarrow \bullet\, \mathbf{id}]\ \}$

Grammar:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (\ E\ )$
$F \rightarrow \mathbf{id}$

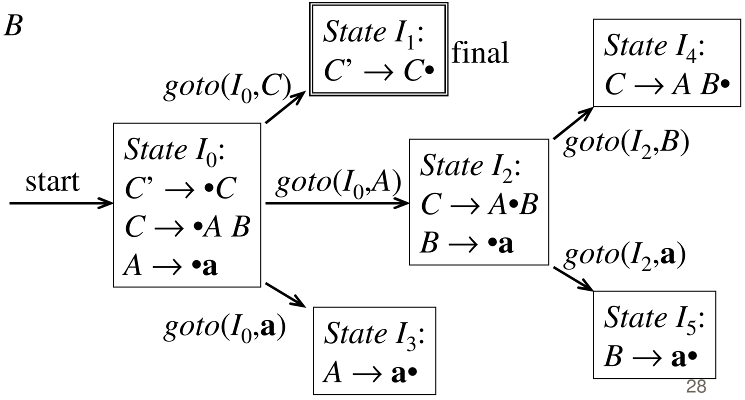
Constructing SLR Parsing Tables

1. Augment the grammar with $S' \to S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of *LR(0) items*
3. If $[A\to\alpha\bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A\to\alpha\bullet] \in I_i$ then set $action[i,a]$=reduce $A\to\alpha$ for all $a \in$ FOLLOW(A) (apply only if $A\neq S'$)
5. If $[S'\to S\bullet]$ is in I_i then set $action[i,\mathbf{\$}]$=accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S'\to\bullet S]$

Example SLR Grammar and LR(0) Items

Augmented grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

$I_0 = closure(\{[C' \to \bullet C]\})$
$I_1 = \text{goto}(I_0,C) = closure(\{[C' \to C\bullet]\})$
…


State I_1: $C' \to C\bullet$
final
State I_4: $C \to A\ B\bullet$
$goto(I_0,C)$
$goto(I_2,B)$
start
State I_0: $C' \to \bullet C$, $C \to \bullet A\ B$, $A \to \bullet\mathbf{a}$
$goto(I_0,A)$
State I_2: $C \to A\bullet B$, $B \to \bullet\mathbf{a}$
$goto(I_2,\mathbf{a})$
$goto(I_0,\mathbf{a})$
State I_3: $A \to \mathbf{a}\bullet$
State I_5: $B \to \mathbf{a}\bullet$

Example SLR Parsing Table

State I_0:
$C' \to \bullet C$
$C \to \bullet A\ B$
$A \to \bullet \mathbf{a}$

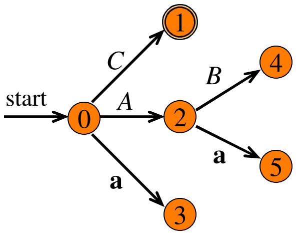
State I_1:
$C' \to C\bullet$

State I_2:
$C \to A\bullet B$
$B \to \bullet \mathbf{a}$

State I_3:
$A \to \mathbf{a}\bullet$

State I_4:
$C \to A\ B\bullet$

State I_5:
$B \to \mathbf{a}\bullet$



	a	**$**	*C*	*A*	*B*
0	s3		1	2	
1		acc			
2	s5				4
3	r3				
4		r2			
5		r4			

Grammar:
1. $C' \to C$
2. $C \to A\ B$
3. $A \to \mathbf{a}$
4. $B \to \mathbf{a}$

SLR and Ambiguity

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

$$S \to L = R \mid R$$
$$L \to * R \mid \mathbf{id}$$
$$R \to L$$

I_0:
$S' \to \bullet S$
$S \to \bullet L{=}R$
$S \to \bullet R$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$
$R \to \bullet L$

I_1:
$S' \to S\bullet$

I_2:
$S \to L\bullet{=}R$
$R \to L\bullet$

I_3:
$S \to R\bullet$

I_4:
$L \to {*}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_5:
$L \to \mathbf{id}\bullet$

I_6:
$S \to L{=}\bullet R$
$R \to \bullet L$
$L \to \bullet{*}R$
$L \to \bullet\mathbf{id}$

I_7:
$L \to {*}R\bullet$

I_8:
$R \to L\bullet$

I_9:
$S \to L{=}R\bullet$

action[2,=]=s6
action[2,=]=r5

no

Has no SLR parsing table

LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

LR(0) item:	LR(1) item:
$[A \to \alpha \bullet \beta]$	$[A \to \alpha \bullet \beta, a]$

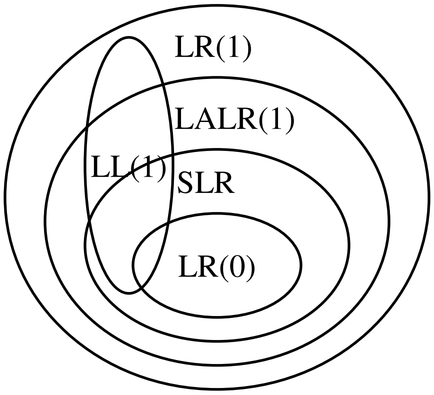
LALR(1) Grammars

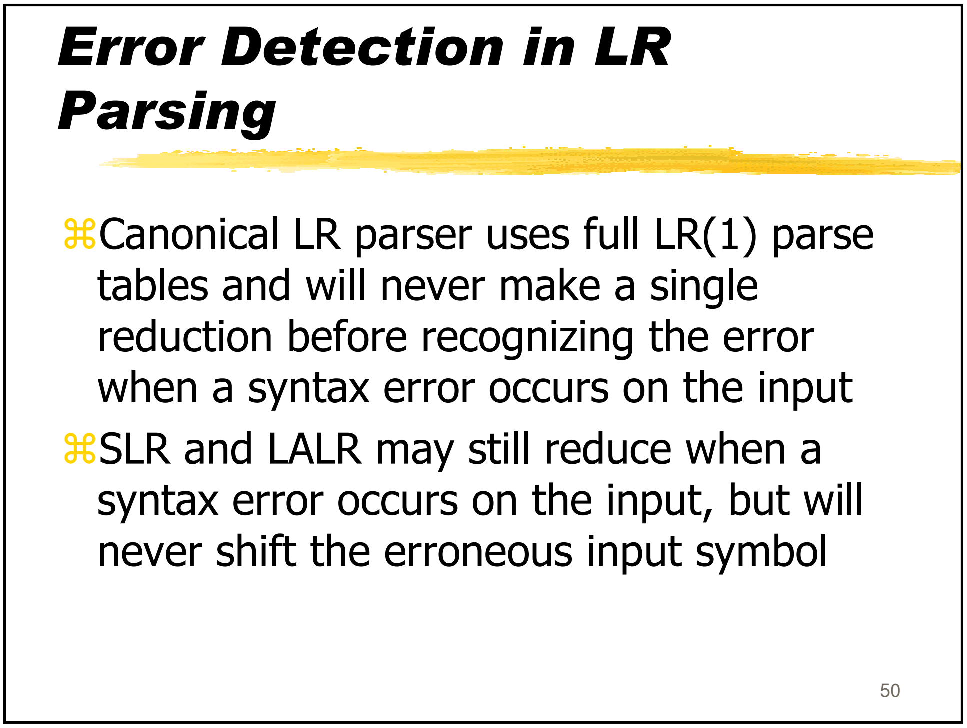
- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals × terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states × terminals → shift/reduce actions
  - LR states × terminals → goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

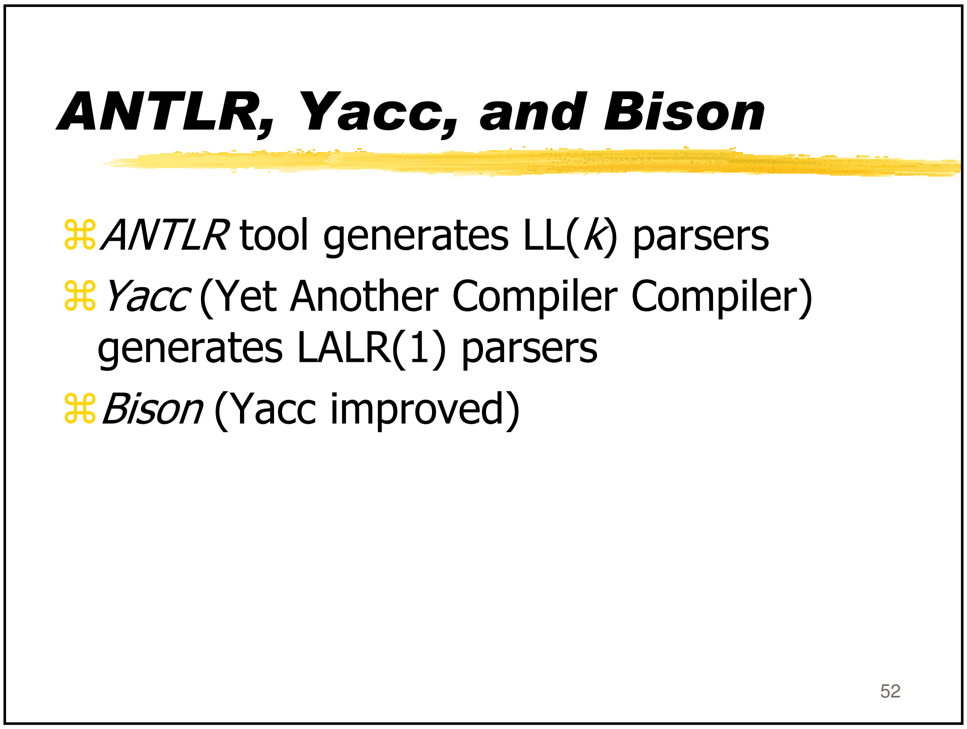
LL, SLR, LR, LALR Grammars





Error Detection in LR Parsing

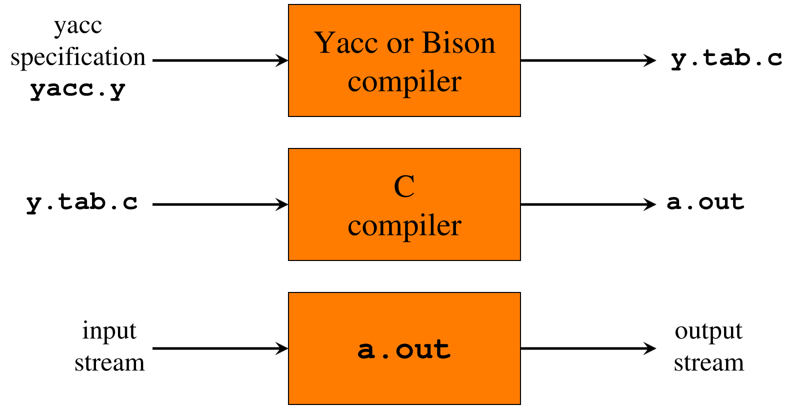
- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol



ANTLR, Yacc, and Bison

- *ANTLR* tool generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler) generates LALR(1) parsers
- *Bison* (Yacc improved)

Creating an LALR(1) Parser with Yacc/Bison



yacc specification **`yacc.y`** → Yacc or Bison compiler → **`y.tab.c`**

`y.tab.c` → C compiler → **`a.out`**

input stream → **`a.out`** → output stream

Yacc Specification

- A *yacc specification* consists of three parts:

  yacc declarations, and C declarations in `%{ %}`
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures

- *Translation rules* are grammar productions and actions:

  production$_1$ { *semantic action*$_1$ }
  production$_2$ { *semantic action*$_2$ }
  …
  production$_n$ { *semantic action*$_n$ }

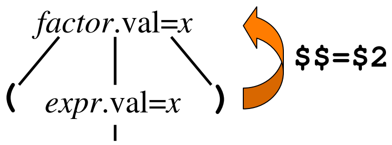
Writing a Grammar in Yacc

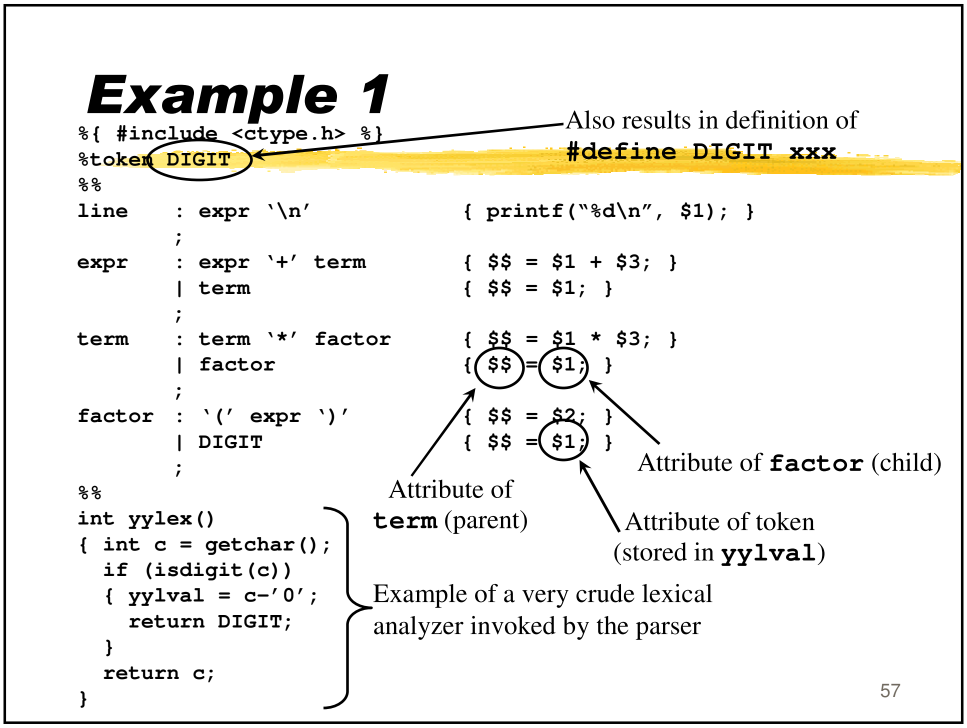
- Productions in Yacc are of the form
  Nonterminal : tokens/nonterminals { *action* }
  | tokens/nonterminals { *action* }
  …
  ;
- Tokens that are single characters can be used directly within productions, e.g. `'+'`
- Named tokens must be declared first in the declaration part using
  `%token` *TokenName*

Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:
  X : Y_1 Y_2 Y_3 … Y_n { *action* }
  - **`$$`** refers to the value of the attribute of *X*
  - **`$`***i* refers to the value of the attribute of Y_i
- For example

```
factor : '(' expr ')' { $$=$2; }
```





Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr `\n'             { printf("%d\n", $1); }
        ;
expr    : expr `+' term         { $$ = $1 + $3; }
        | term                  { $$ = $1; }
        ;
term    : term `*' factor       { $$ = $1 * $3; }
        | factor                { $$ = $1; }
        ;
factor  : `(' expr `)'          { $$ = $2; }
        | DIGIT                 { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of
`#define DIGIT xxx`

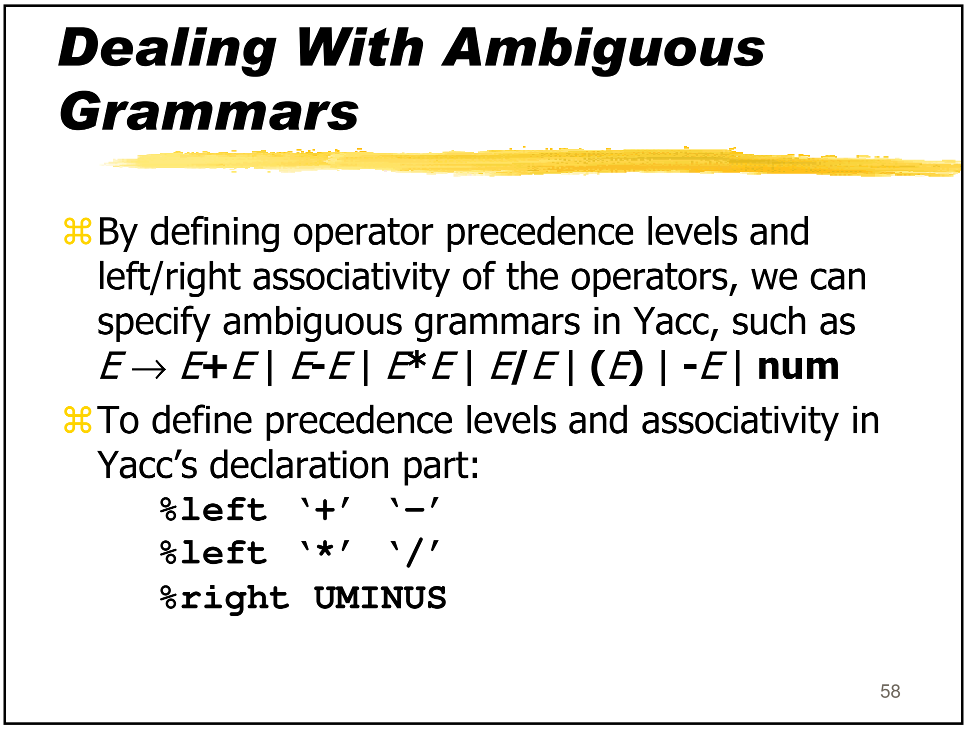
Attribute of **`factor`** (child)

Attribute of **`term`** (parent)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

57

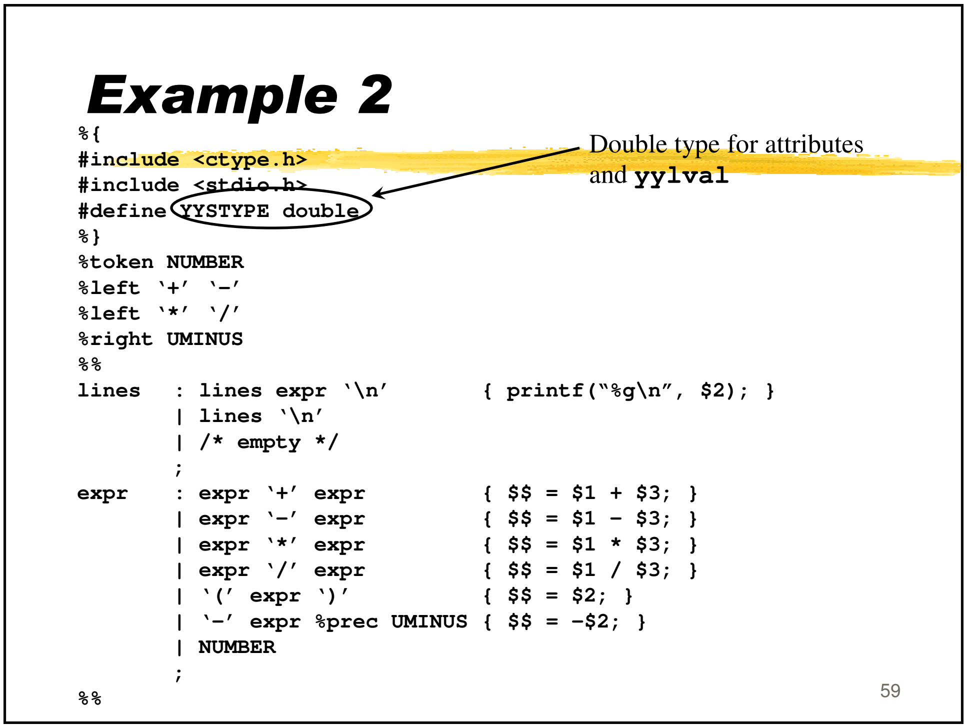


Dealing With Ambiguous Grammars

- By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as $E \rightarrow E\mathbf{+}E \mid E\mathbf{-}E \mid E\mathbf{*}E \mid E\mathbf{/}E \mid \mathbf{(}E\mathbf{)} \mid \mathbf{-}E \mid \textbf{num}$
- To define precedence levels and associativity in Yacc's declaration part:

```
%left  '+'  '-'
%left  '*'  '/'
%right UMINUS
```

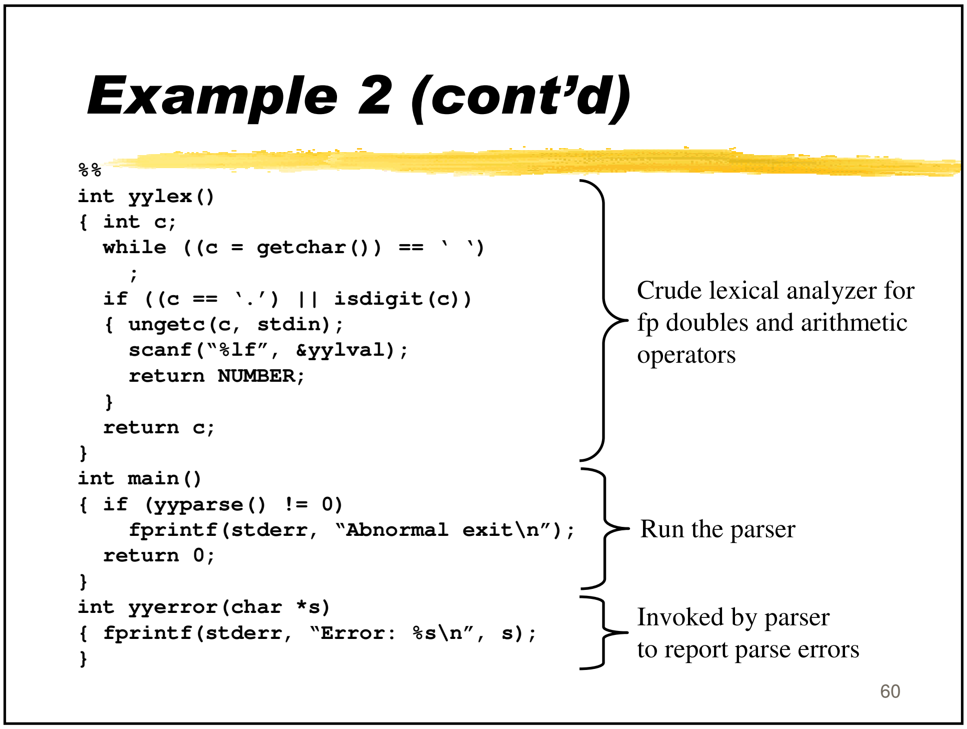
58



Example 2

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%right UMINUS
%%
lines   : lines expr ‘\n’       { printf(“%g\n”, $2); }
        | lines ‘\n’
        | /* empty */
        ;
expr    : expr ‘+’ expr         { $$ = $1 + $3; }
        | expr ‘-’ expr         { $$ = $1 - $3; }
        | expr ‘*’ expr         { $$ = $1 * $3; }
        | expr ‘/’ expr         { $$ = $1 / $3; }
        | ‘(’ expr ‘)’          { $$ = $2; }
        | ‘-’ expr %prec UMINUS { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **`yylval`**



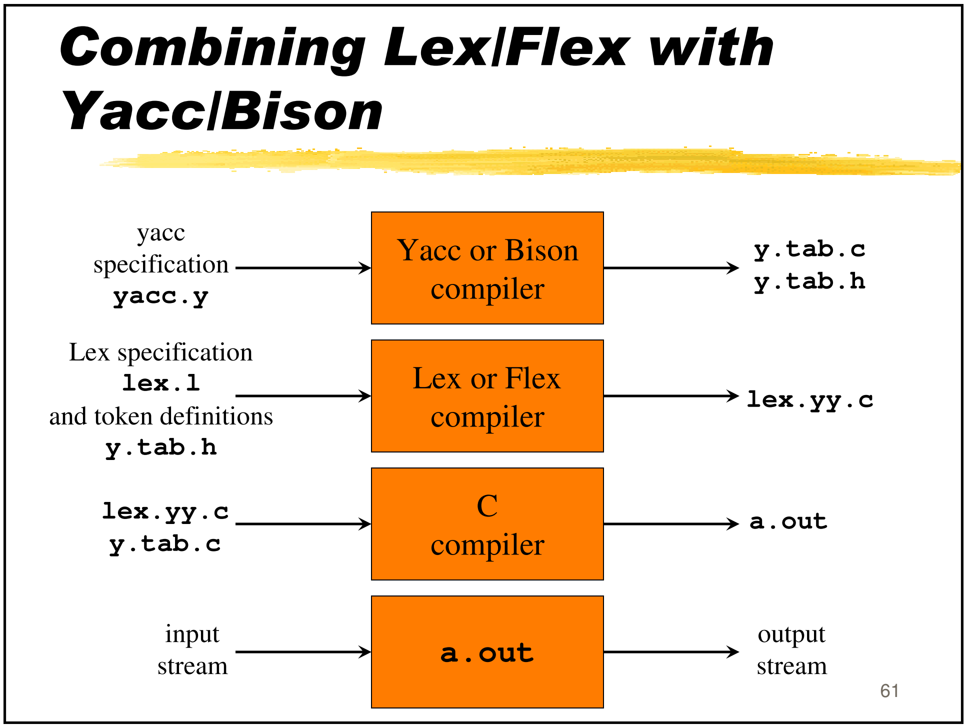
Example 2 (cont’d)

```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ‘ ’)
    ;
  if ((c == ‘.’) || isdigit(c))
  { ungetc(c, stdin);
    scanf(“%lf”, &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, “Abnormal exit\n”);
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, “Error: %s\n”, s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors



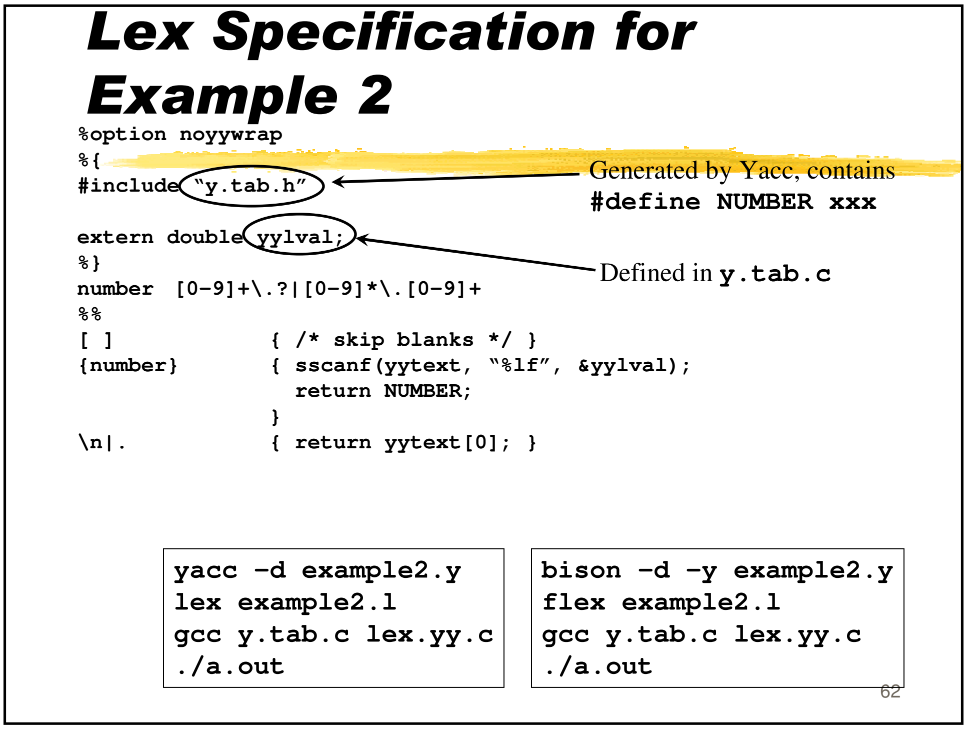
Combining Lex/Flex with Yacc/Bison

yacc specification **yacc.y** → Yacc or Bison compiler → **y.tab.c** **y.tab.h**

Lex specification **lex.l** and token definitions **y.tab.h** → Lex or Flex compiler → **lex.yy.c**

lex.yy.c **y.tab.c** → C compiler → **a.out**

input stream → **a.out** → output stream



Lex Specification for Example 2

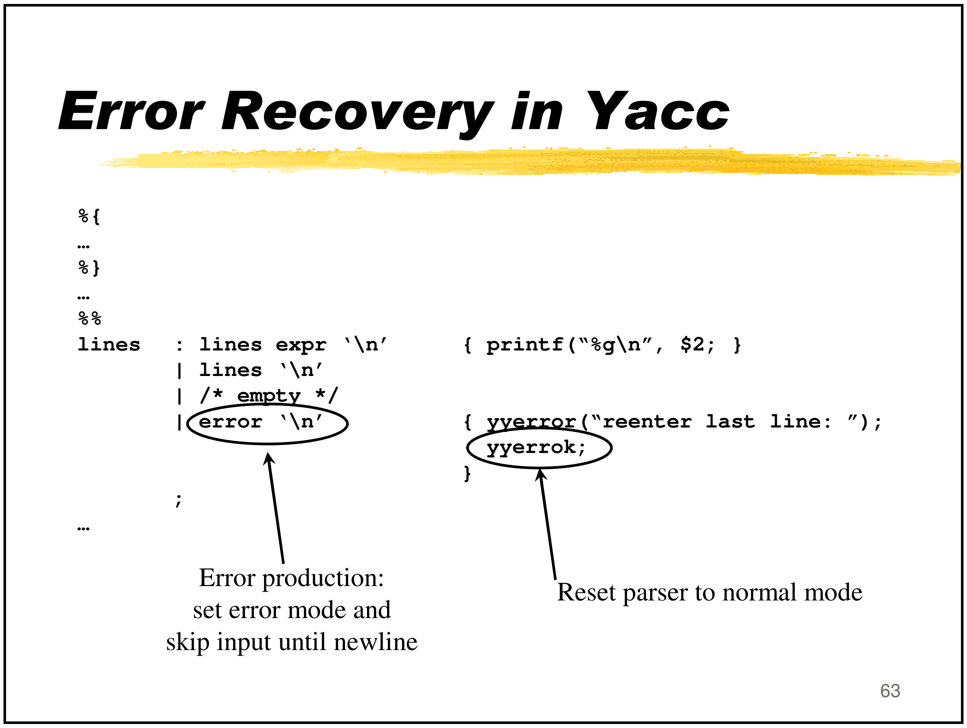
```
%option noyywrap
%{
#include “y.tab.h”

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]            { /* skip blanks */ }
{number}       { sscanf(yytext, “%lf”, &yylval);
                 return NUMBER;
               }
\n|.           { return yytext[0]; }
```

Generated by Yacc, contains **#define NUMBER xxx**

Defined in **y.tab.c**

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```



Error Recovery in Yacc

```
%{
…
%}
…
%%
lines   : lines expr ‘\n’       { printf(“%g\n”, $2; }
        | lines ‘\n’
        | /* empty */
        | error ‘\n’            { yyerror(“reenter last line: ”);
                                  yyerrok;
                                }
        ;
…
```

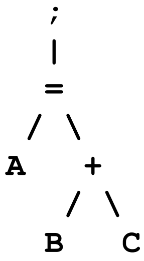
Error production:
set error mode and
skip input until newline

Reset parser to normal mode

Intermediate Code Generation

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

The Phases of a Compiler

Phase	Output	Sample
Programmer	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A', '=', 'B', '+', 'C', ';'` And *symbol table* for identifiers
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

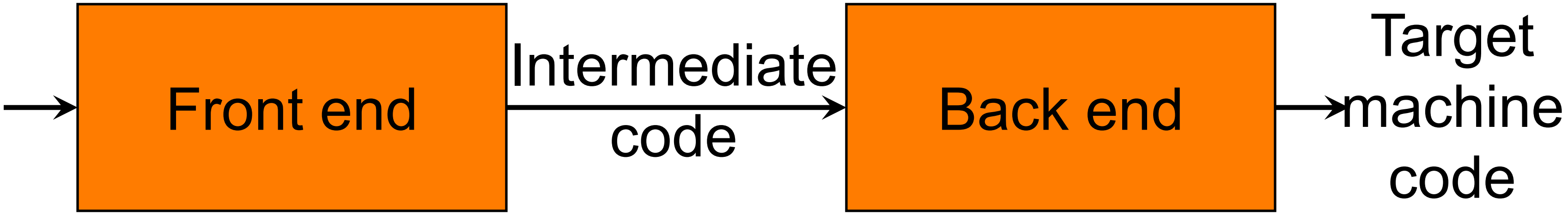
Optimizer	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 #2.3 A`

Code generator	Assembly code	`MOVF #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2`
		`MOVF r2,A`

Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end



- Enables machine-independent code optimization

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

  x := y op z

- *Two-address code*:

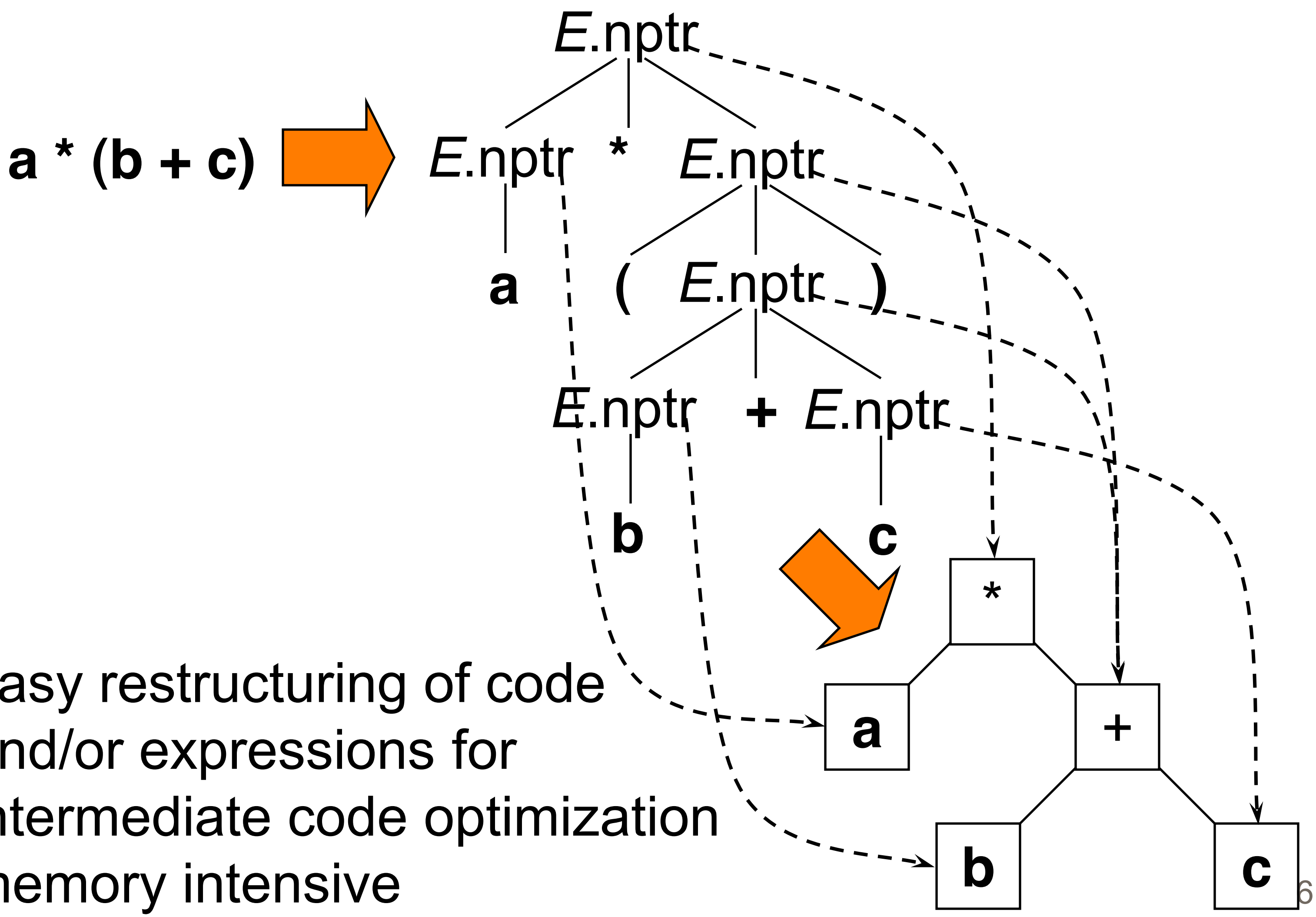
  x := op y

  which is the same as x := x op y

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \rightarrow \textbf{id} \mathbin{\textbf{:=}} E$	S.nptr := *mknode*(‘:=’, *mkleaf*(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1 \mathbin{\textbf{+}} E_2$	E.nptr := *mknode*(‘+’, E_1.nptr, E_2.nptr)
$E \rightarrow E_1 \mathbin{\textbf{*}} E_2$	E.nptr := *mknode*(‘*’, E_1.nptr, E_2.nptr)
$E \rightarrow \textbf{-}\ E_1$	E.nptr := *mknode*(‘uminus’, E_1.nptr)
$E \rightarrow \textbf{(}\ E_1\ \textbf{)}$	E.nptr := E_1.nptr
$E \rightarrow \textbf{id}$	E.nptr := *mkleaf*(**id**, **id**.entry)

Abstract Syntax Trees

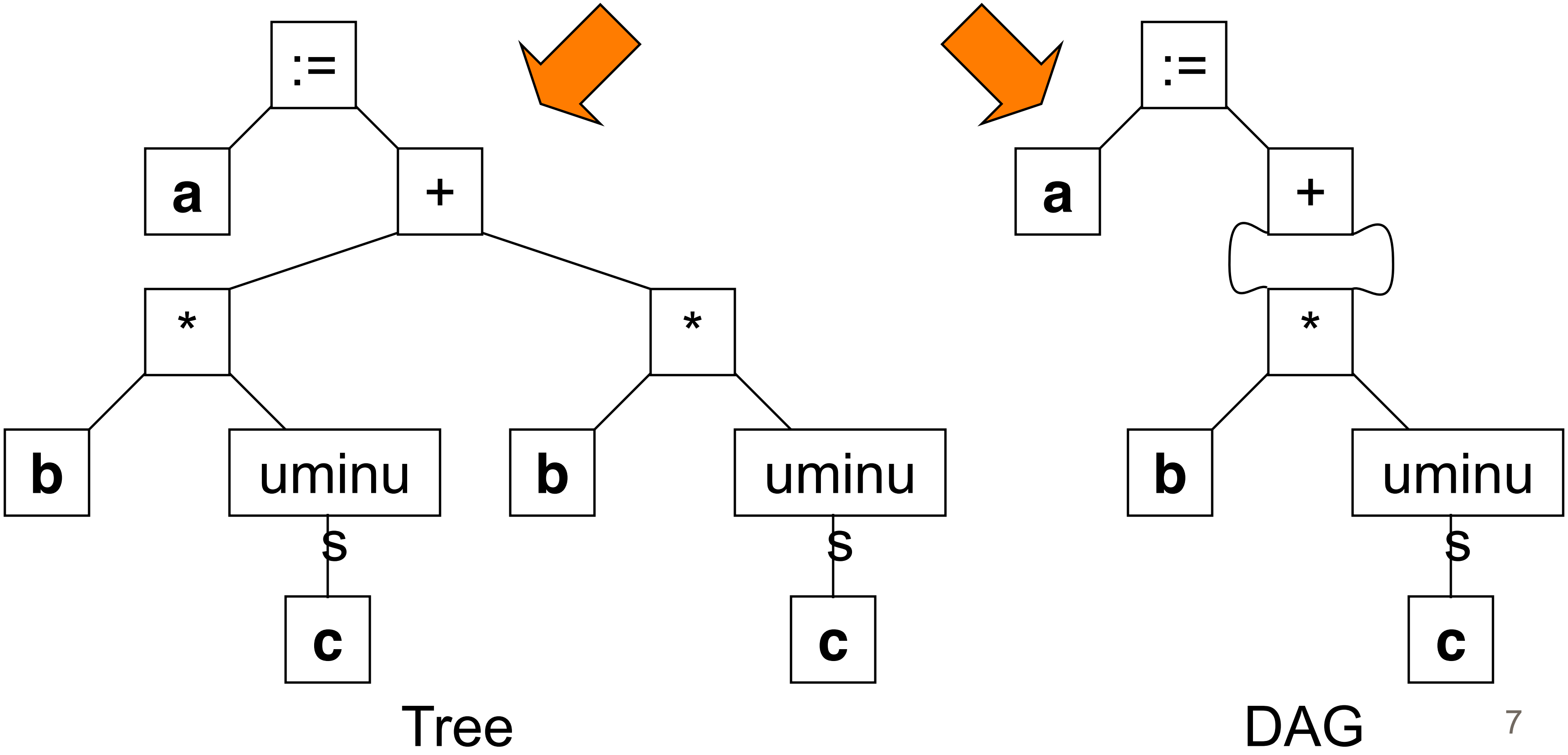


Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Abstract Syntax Trees versus DAGs

a := b * -c + b * -c



Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pro: easy to generate

Cons: stack operations are more difficult to optimize

Bytecode (for example)

```
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iadd           // +
istore 1       // store a
```

Three-Address Code

a := b * -c + b * -c

```
t1  := - c
t2  := b * t1
t3  := - c
t4  := b * t3
t5  := t2 + t4
a   := t5
```

Linearized representation of a syntax tree

```
t1  := - c
t2  := b * t1
t5  := t2 + t2
a   := t5
```

Linearized representation of a syntax DAG

Three-Address Statements

- Assignment statements: *x* := *y op z*, *x* := *op y*
- Indexed assignments: *x* := *y*[*i*], *x*[*i*] := *y*
- Pointer assignments: *x* := &*y*, *x* := **y*, **x* := *y*
- Copy statements: *x* := *y*
- Unconditional jumps: **goto** *lab*
- Conditional jumps: **if** *x relop y* **goto** *lab*
- Function calls: **param** *x*… **call** *p*, *n*
  return *y*

Syntax-Directed Translation into Three-Address Code

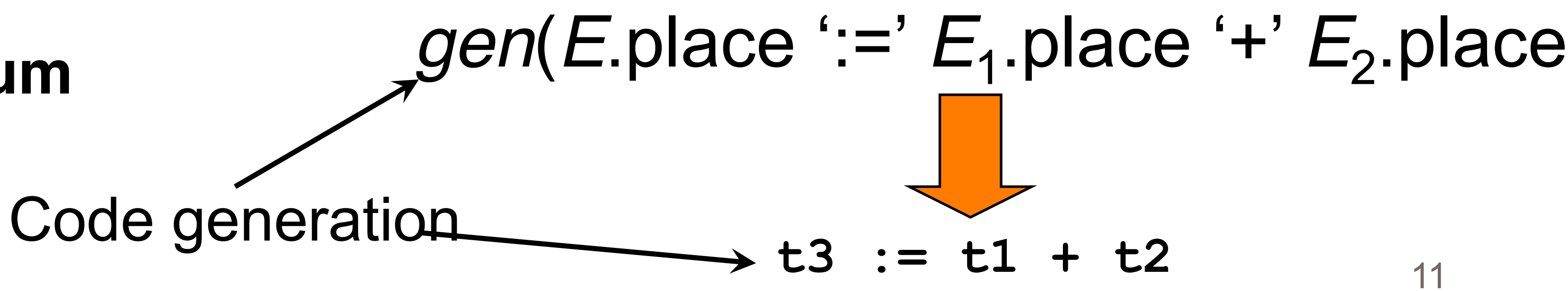
Productions

$S \rightarrow$ **id :=** E
  | **while** E **do** S

$E \rightarrow E$ **+** E
  | E ***** E
  | **-** E
  | **(** E **)**
  | **id**
  | **num**

Synthesized attributes:

S.code	three-address code for
S.begin	label to start of *S* or nil
S.after	label to end of *S* or nil
E.code	three-address code for
E.place	a name holding the val



gen(*E*.place ‘:=’ E_1.place ‘+’ E_2.place

Code generation

```
t3 := t1 + t2
```

Syntax-Directed Translation into Three-Address Code (cont'd)

Productions	Semantic rules
$S \rightarrow$ **id** **:=** E	*S*.code := *E*.code || *gen*(**id**.place ':=' *E*.place); *S*.begin := *S*.after :
$S \rightarrow$ **while** E **do** S_1	(*see next slide*)
$E \rightarrow E_1$ **+** E_2	*E*.place := *newtemp*(); *E*.code := E_1.code || E_2.code || *gen*(*E*.place ':=' E_1.place '+' E_2.pla
$E \rightarrow E_1$ ***** E_2	*E*.place := *newtemp*(); *E*.code := E_1.code || E_2.code || *gen*(*E*.place ':=' E_1.place '*' E_2.pla
$E \rightarrow$ **-** E_1	*E*.place := *newtemp*(); *E*.code := E_1.code || *gen*(*E*.place ':=' 'uminus' E_1.place)
$E \rightarrow$ **(** E_1 **)**	*E*.place := E_1.place *E*.code := E_1.code
$E \rightarrow$ **id**	*E*.place := **id**.name *E*.code := ''
$E \rightarrow$ **num**	*E*.place := *newtemp*(); *E*.code := *gen*(*E*.place ':=' **num**.value)

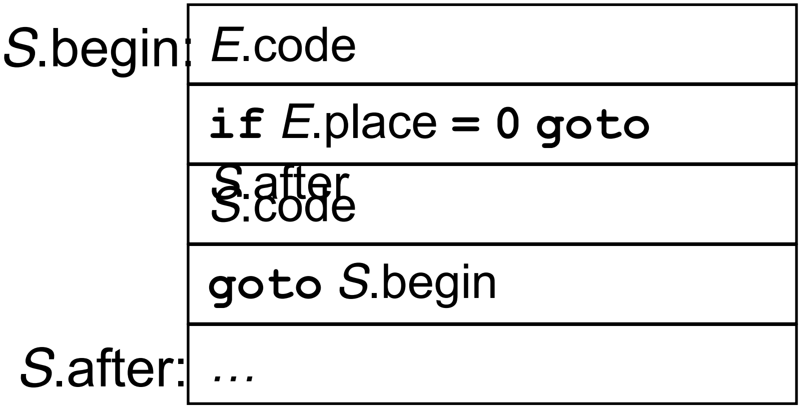
Syntax-Directed Translation into Three-Address Code (cont'd)

Production

$S \rightarrow$ **while** E **do** S_1

Semantic rule

S.begin := *newlabel*()
S.after := *newlabel*()
S.code := *gen*(*S*.begin ':') ||
    E.code ||
    gen('if' *E*.place '=' '0' 'goto' *S*.after) ||
    S_1.code ||
    gen('goto' *S*.begin) ||
    gen(*S*.after ':')



S.begin: *E*.code

`if` *E*.place = 0 `goto` *S*.after

S.code

`goto` *S*.begin

S.after: …

Example

```
i := 2 * n + k
while i do
   i := i - k
```

```
    t1  := 2
    t2  := t1  * n
    t3  := t2  + k
    i   := t3
L1: if i = 0 goto L2
    t4  := i - k
    i   := t4
    goto L1
L2:
```

Implementation of Three-Address Statements: Quads

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Pro: temporaries are implicit
Cons: difficult to rearrange code

Implementation of Three-Address Stmts: Indirect Triples

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

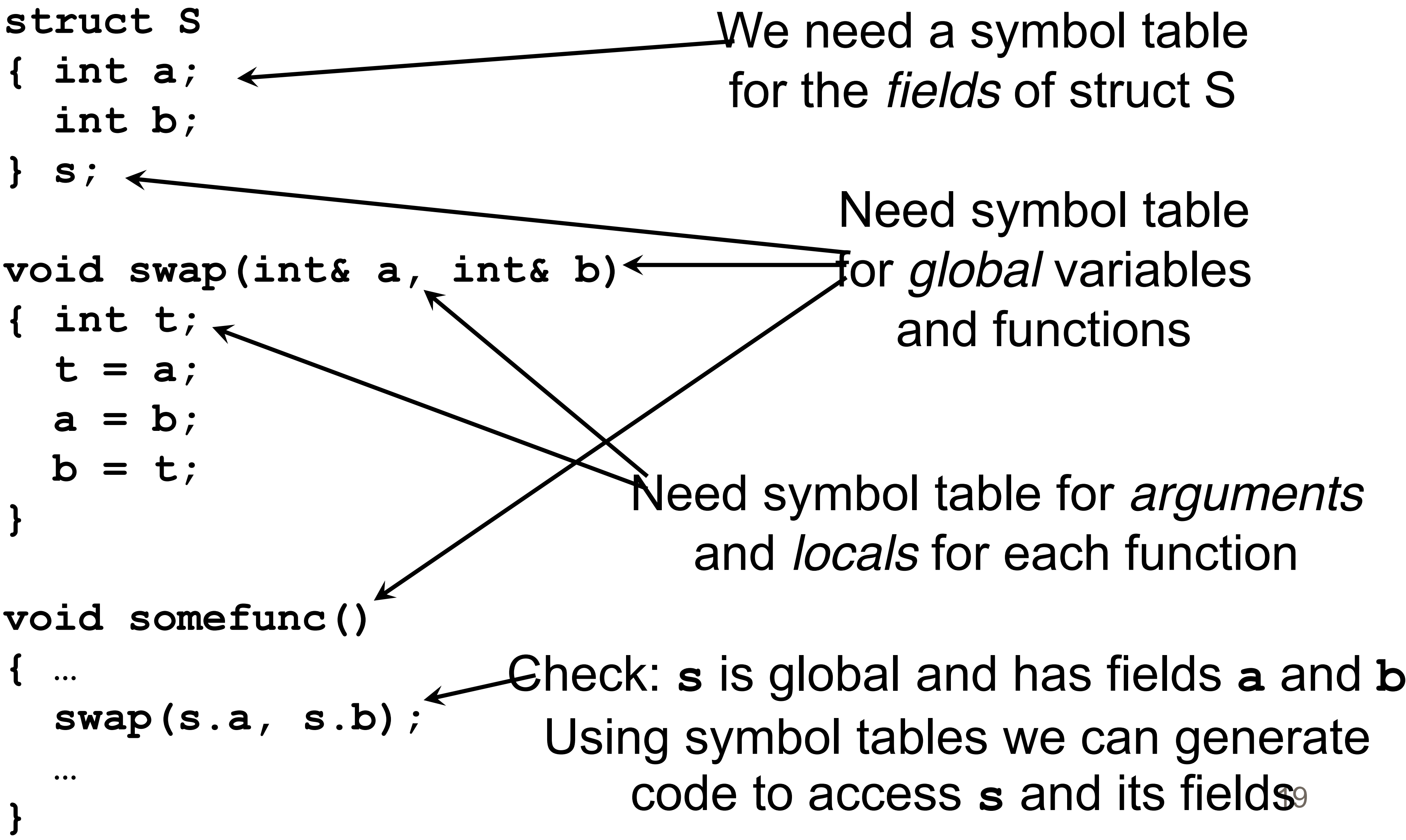
Triple container

Pro: temporaries are implicit & easier to rearrange code

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown on the previous slides is somewhat simplistic, because it assumes that the names of variables can be easily resolved by the back end in global or local variables
- We need local symbol tables to record global declarations as well as local declarations in procedures, blocks, and structs to resolve names

Symbol Tables for Scoping



```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

We need a symbol table for the *fields* of struct S

Need symbol table for *global* variables and functions

Need symbol table for *arguments* and *locals* for each function

Check: **s** is global and has fields **a** and **b**

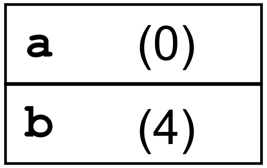
Using symbol tables we can generate code to access **s** and its fields

Offset and Width for Runtime Allocation

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

The fields `a` and `b` of struct S are located at *offsets* 0 and 4 from the start of S

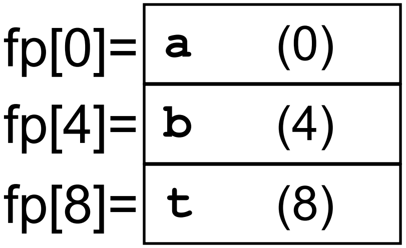
The *width* of S is 8



a	(0)
b	(4)

Subroutine frame holds arguments `a` and `b` and local `t` at *offsets* 0, 4, and 8

Subroutine frame

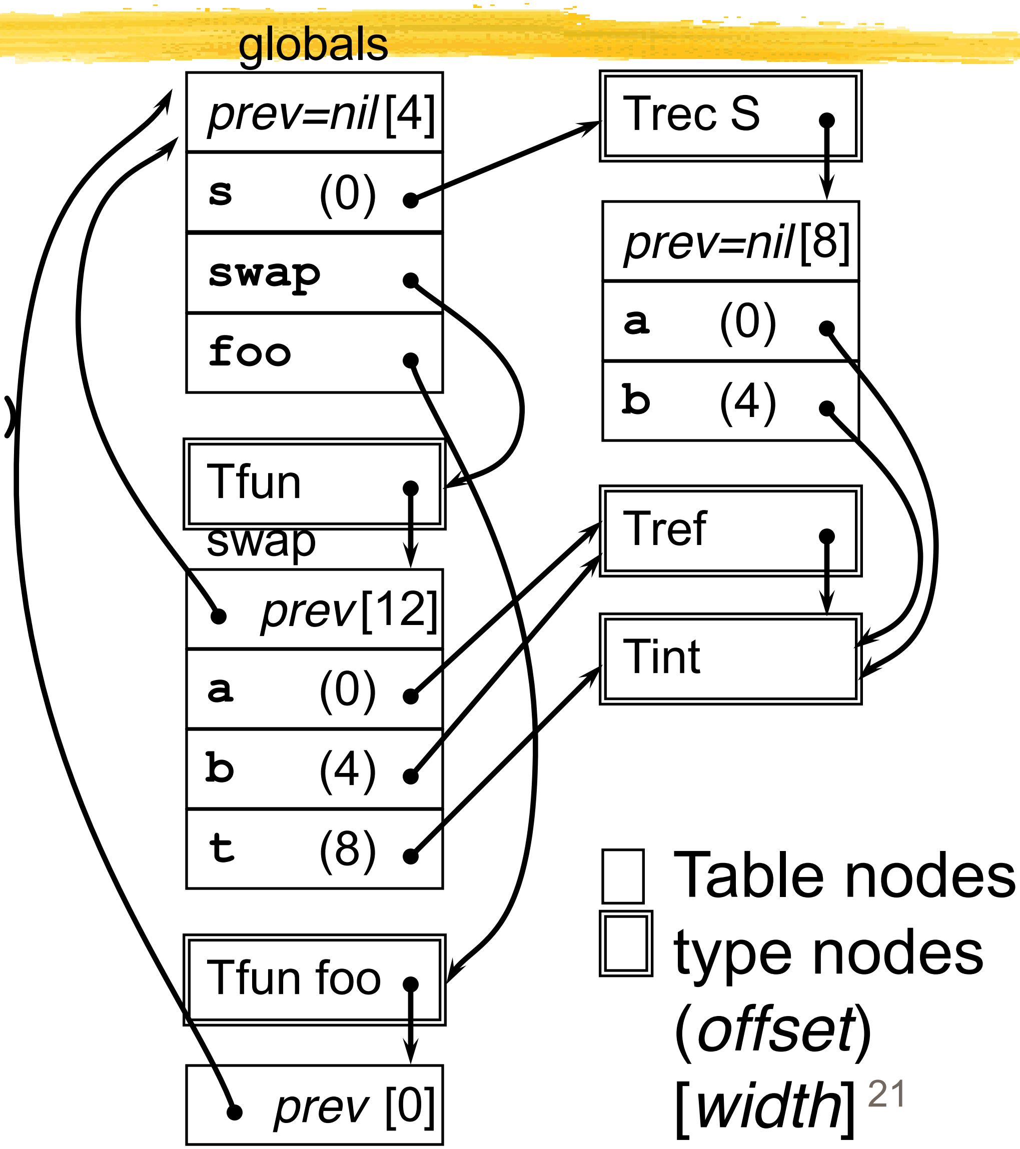


fp[0]=	a	(0)
fp[4]=	b	(4)
fp[8]=	t	(8)

The *width* of the frame is 12

Example

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ …
  swap(s.a, s.b);
  …
}
```



globals

prev=nil [4]

s (0)

swap

foo

Tfun swap

prev [12]

a (0)

b (4)

t (8)

Tfun foo

prev [0]

Trec S

prev=nil [8]

a (0)

b (4)

Tref

Tint

□ Table nodes

□ type nodes

(*offset*)

[*width*]

Hierarchical Symbol Table Operations

- *mktable*(*previous*) returns a pointer to a new table that is linked to a previous table in the outer scope
- *enter*(*table*, *name*, *type*, *offset*) creates a new entry in *table*
- *addwidth*(*table*, *width*) accumulates the total width of all entries in *table*
- *enterproc*(*table*, *name*, *newtable*) creates a new entry in *table* for procedure with local scope *newtable*
- *lookup*(*table*, *name*) returns a pointer to the entry in the table for *name* by following linked tables

Syntax-Directed Translation of Declarations in Scope

Productions

P → *D* **;** *S*
D → *D* **;** *D*
    | **id :** *T*
    | **proc id ;** *D* **;** *S*
T → **integer**
    | **real**
    | **array [num] of** *T*
    | **^** *T*
    | **record** *D* **end**
S → *S* **;** *S*
    | **id :=** *E*
    | **call id (** *A* **)**

Productions *(cont'd)*

E → *E* **+** *E*
    | *E* ***** *E*
    | **-** *E*
    | **(** *E* **)**
    | **id**
    | *E* **^**
    | **&** *E*
    | *E* **. id**
A → *A* **,** *E*
    | *E*

Synthesized attributes:

T.type	pointer to type
T.width	storage width of type (bytes)
E.place	name of temp holding value of

Global data to implement scoping:

tblptr	stack of pointers to tables
offset	stack of offset values

Syntax-Directed Translation of Declarations in Scope (cont'd)

```
P →   { t := mktable(nil); push(t, tblptr); push(0, offset) }
     D ; S
D → id : T
      { enter(top(tblptr), id.name, T.type, top(offset));
        top(offset) := top(offset) + T.width }
D → proc id ;
      { t := mktable(top(tblptr));  push(t, tblptr); push(0, offset
     D_1 ; S
      { t := top(tblptr); addwidth(t, top(offset));
        pop(tblptr); pop(offset);
        enterproc(top(tblptr), id.name, t) }
D → D_1 ; D_2
```

Syntax-Directed Translation of Declarations in Scope (cont’d)

T → **integer** { *T*.type := ‘*integer*’; *T*.width := 4 }

T → **real** { *T*.type := ‘*real*’; *T*.width := 8 }

T → **array [num] of** T_1

{ *T*.type := *array*(**num**.val, T_1.type);
T.width := **num**.val * T_1.width }

T → **^** T_1

{ *T*.type := *pointer*(T_1.type); *T*.width := 4 }

T → **record**

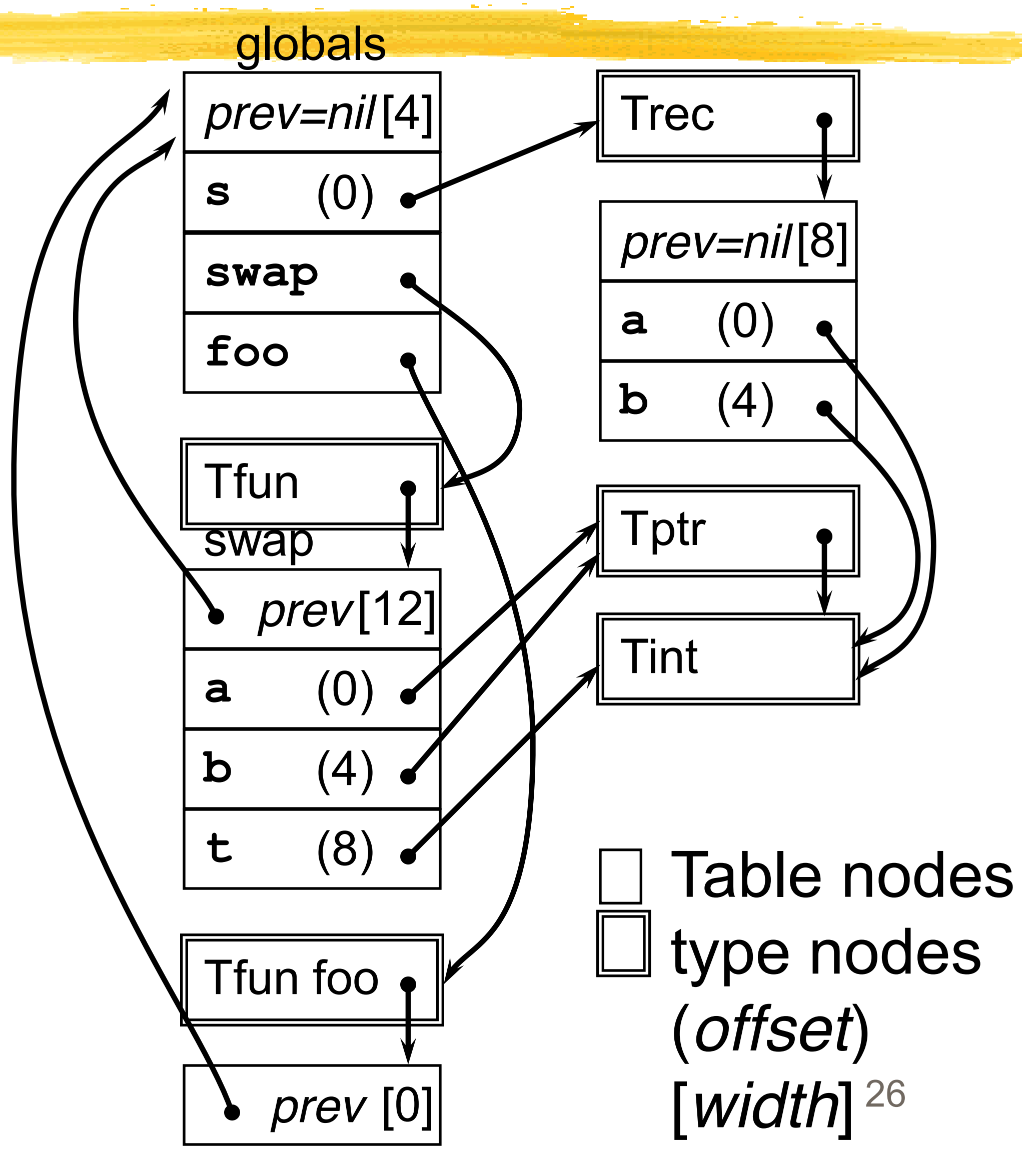
{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }

D **end**

{ *T*.type := *record*(*top*(*tblptr*)); *T*.width := *top*(*offset*);
addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*

Example

```
s: record
      a: integer;
      b: integer;
   end;

proc swap;
   a: ^integer;
   b: ^integer;
   t: integer;
   t := a^;
   a^ := b^;
   b^ := t;

proc foo;
   call swap(&s.a, &s.b);
```



globals

prev=nil [4]

s (0)

swap

foo

Tfun swap

prev [12]

a (0)

b (4)

t (8)

Tfun foo

prev [0]

Trec

prev=nil [8]

a (0)

b (4)

Tptr

Tint

□ Table nodes

□ type nodes

(*offset*)

[*width*]

Syntax-Directed Translation of Statements in Scope

```
S → S ; S
S → id := E
        { p := lookup(top(tblptr), id.name);
          if p = nil then
            error()
          else if p.level = 0 then // global variable
            emit(id.place ':=' E.place)
          else // local variable in subroutine frame
            emit(fp[p.offset] ':=' E.place) }
```

Globals

s	(0)
x	(8)
y	(12)

Subroutine frame

fp[0]=	a	(0)
fp[4]=	b	(4)
fp[8]=	t	(8)
…		

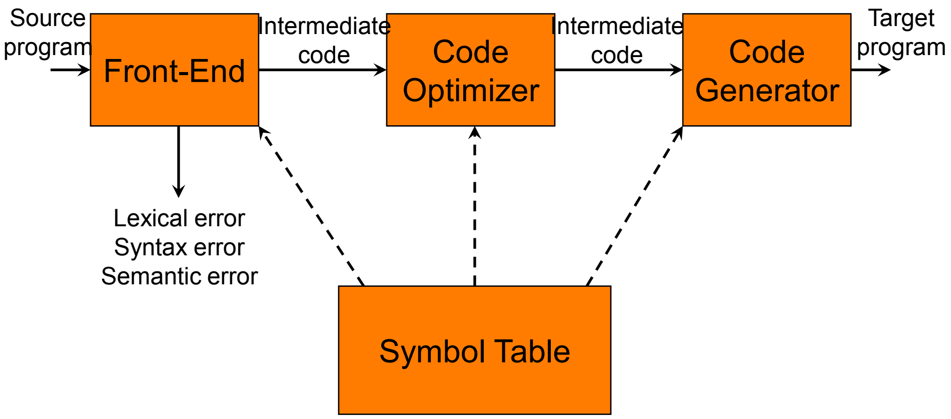
Syntax-Directed Translation of Expressions in Scope

```
$E \to E_1 + E_2$   { $E$.place := newtemp();
                        emit($E$.place ‘:=’ $E_1$.place ‘+’ $E_2$.place) }
$E \to E_1 * E_2$   { $E$.place := newtemp();
                        emit($E$.place ‘:=’ $E_1$.place ‘*’ $E_2$.place) }
$E \to - E_1$       { $E$.place := newtemp();
                        emit($E$.place ‘:=’ ‘uminus’ $E_1$.place) }
$E \to ( E_1 )$     { $E$.place := $E_1$.place }
$E \to$ id          { $p$ := lookup(top(tblptr), id.name);
                        if $p$ = nil then error()
                        else if $p$.level = 0 then // global variable
                          $E$.place := id.place
                        else // local variable in frame
                          $E$.place := fp[$p$.offset] }
```

Code Generation

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

Position of a Code Generator in the Compiler Model



Code Generation

- Code produced by compiler must be correct
  - Source to target program transformation is *semantics preserving*
- Code produced by compiler should be of high quality
  - Effective use of target machine resources
  - Heuristic techniques can generate good but suboptimal code, because generating optimal code is undecidable

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker)
  - Assembly language (facilitates debugging)
  - Byte code forms for interpreters (e.g. JVM)

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **`R0`**, **`R1`**, …, **`R`***n*-1
  - Two-address instructions of the form

    op source, *destination*

The Target Machine: Op-codes and Address Modes

- Op-codes (*op*), for example
  - **`MOV`** (move content of *source* to *destination*)
  - **`ADD`** (add content of *source* to *destination*)
  - **`SUB`** (subtract content of *source* from *dest.*)
- Address modes

Mode	Form	Address	Added Cost
Absolute	**`M`**	**`M`**	1
Register	**`R`**	**`R`**	0
Indexed	*c*(**`R`**)	*c+contents*(**`R`**)	1
Indirect register	**`*R`**	*contents*(**`R`**)	0
Indirect indexed	**`*`***c*(**`R`**)	*contents*(*c+contents*(**`R`**))	1
Literal	**`#`***c*	N/A	1

Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction = 1 + cost(*source*-mode) + cost(*destination*-mode)

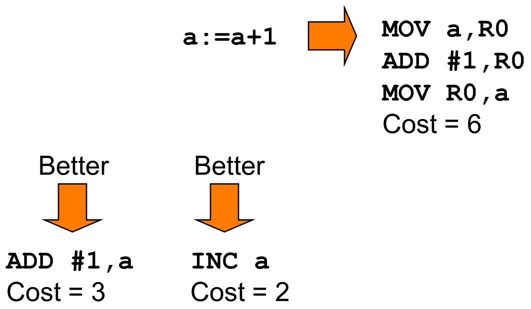
Examples

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(**`R0`**) into register **`R1`**	
`MOV R0,M`	Store *content*(**`R0`**) into memory location **`M`**	
`MOV M,R0`	Store *content*(**`M`**) into register **`R0`**	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(**`R0`**)) into **`M`**	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(**`R0`**))) into **`M`**	3
`MOV #1,R0`	Store 1 into **`R0`**	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(**`R0`**)) to *contents*(12+*contents*(**`R1`**))	3

Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

```
      x:=y+z
to: MOV y,R0
    ADD z,R0
    MOV R0,x
```



```
a:=a+1
```

```
MOV a,R0
ADD #1,R0
MOV R0,a
```
Cost = 6

Better

```
ADD #1,a
```
Cost = 3

Better

```
INC a
```
Cost = 2

Instruction Selection: Utilizing Addressing Modes

- Suppose we translate `a:=b+c` into

```
MOV b,R0
ADD c,R0
MOV R0,a
```

- Assuming addresses of `a`, `b`, and `c` are stored in `R0`, `R1`, and `R2`

```
MOV *R1,*R0
ADD *R2,*R0
```

- Assuming `R1` and `R2` contain values of `b` and `c`

```
ADD R2,R1
MOV R1,a
```

Need for Global Machine-Specific Code Optimizations

- Suppose we translate three-address code

```
    x:=y+z
to: MOV y,R0
    ADD z,R0
    MOV R0,x
```

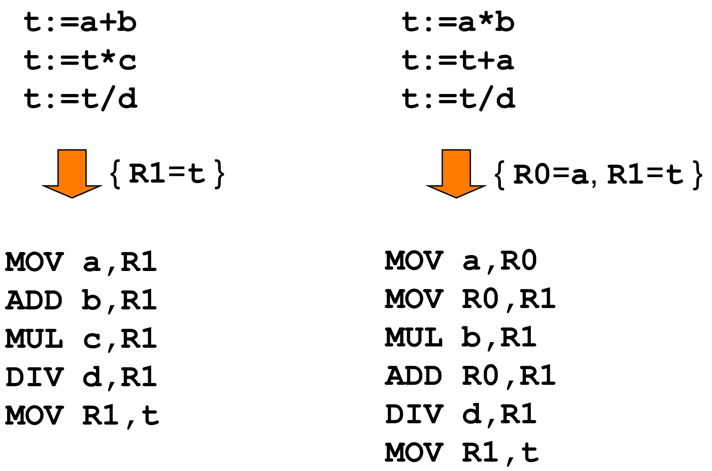
- Then, we translate

```
    a:=b+c
    d:=a+e
to: MOV a,R0
    ADD b,R0
    MOV R0,a
    MOV a,R0    <- Redundant
    ADD e,R0
    MOV R0,d
```

Register Allocation and Assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - *Register allocation* to select the set of variables that will reside in registers at a point in the code
  - *Register assignment* to pick the specific register that a variable will reside in
- Finding an optimal register assignment in general is NP-complete

Example



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

{R1=t}

```
MOV a,R1
ADD b,R1
MUL c,R1
DIV d,R1
MOV R1,t
```

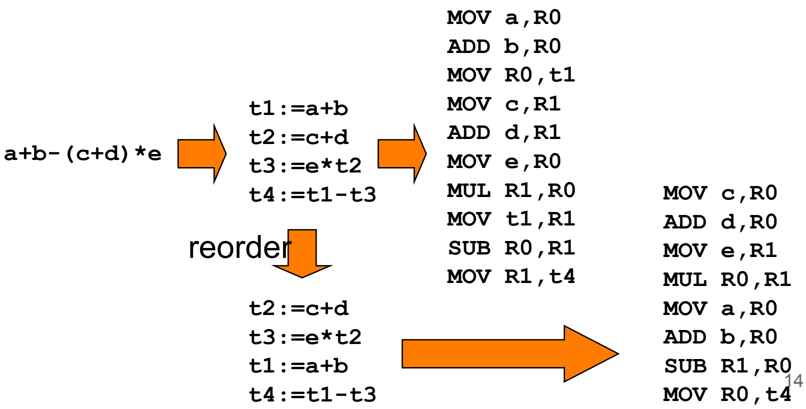
```
t:=a*b
t:=t+a
t:=t/d
```

{R0=a, R1=t}

```
MOV a,R0
MOV R0,R1
MUL b,R1
ADD R0,R1
DIV d,R1
MOV R1,t
```

Choice of Evaluation Order

- When instructions are independent, their evaluation order can be changed



```
a+b-(c+d)*e
```

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

```
MOV a,R0
ADD b,R0
MOV R0,t1
MOV c,R1
ADD d,R1
MOV e,R0
MUL R1,R0
MOV t1,R1
SUB R0,R1
MOV R1,t4
```

reorder

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

```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```

Generating Code for Stack Allocation of Activation Records

```
t1 := a + b        100: ADD #16,SP       Push frame
param t1           108: MOV a,R0
param c            116: ADD b,R0
t2 := call foo,2   124: MOV R0,4(SP)     Store a+b
…                  132: MOV c,8(SP)      Store c
                   140: MOV #156,*SP     Store return address
                   148: GOTO 500         Jump to foo
func foo           156: MOV 12(SP),R0    Get return value
…                  164: SUB #16,SP       Remove frame
return t1          172: …

                   500: …
                   564: MOV R0,12(SP)    Store return value
                   572: GOTO *SP         Return to caller
```

Note: Language and machine dependent
Here we assume C-like implementation with SP and no FP

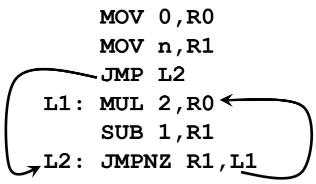
Code Generation Part 2

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

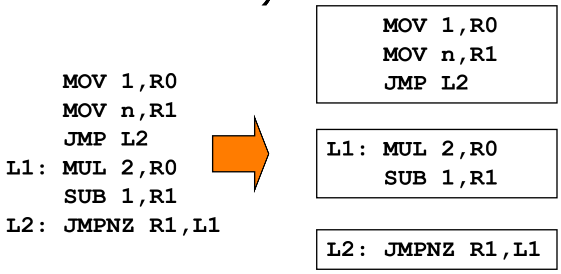
```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```



```
    MOV 0,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)



```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

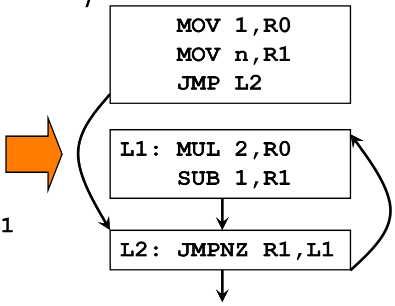
```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```



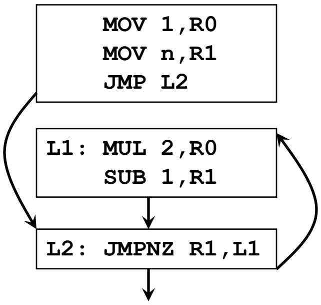
```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

Partition Algorithm for Basic Blocks

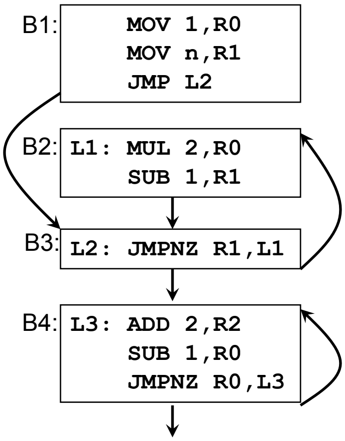
Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic bloc
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



```
B1:     MOV 1,R0
        MOV n,R1
        JMP L2

B2: L1: MUL 2,R0
        SUB 1,R1

B3: L2: JMPNZ R1,L1

B4: L3: ADD 2,R2
        SUB 1,R0
        JMPNZ R0,L3
```

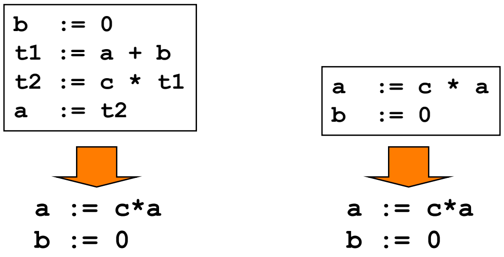
Strongly connected components:

SCC={ {B2,B3}, {B4} }

Entries: B3, B4

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions



```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

```
a  := c * a
b  := 0
```

```
a := c*a
b := 0
```

```
a := c*a
b := 0
```

ocks are equivalent, assuming **t1** and **t2** are *dead*: no longer used (no longer *liv*

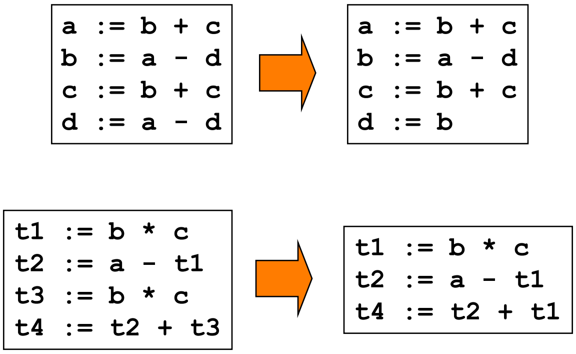
Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

25

Common-Subexpression Elimination

- Remove redundant computations



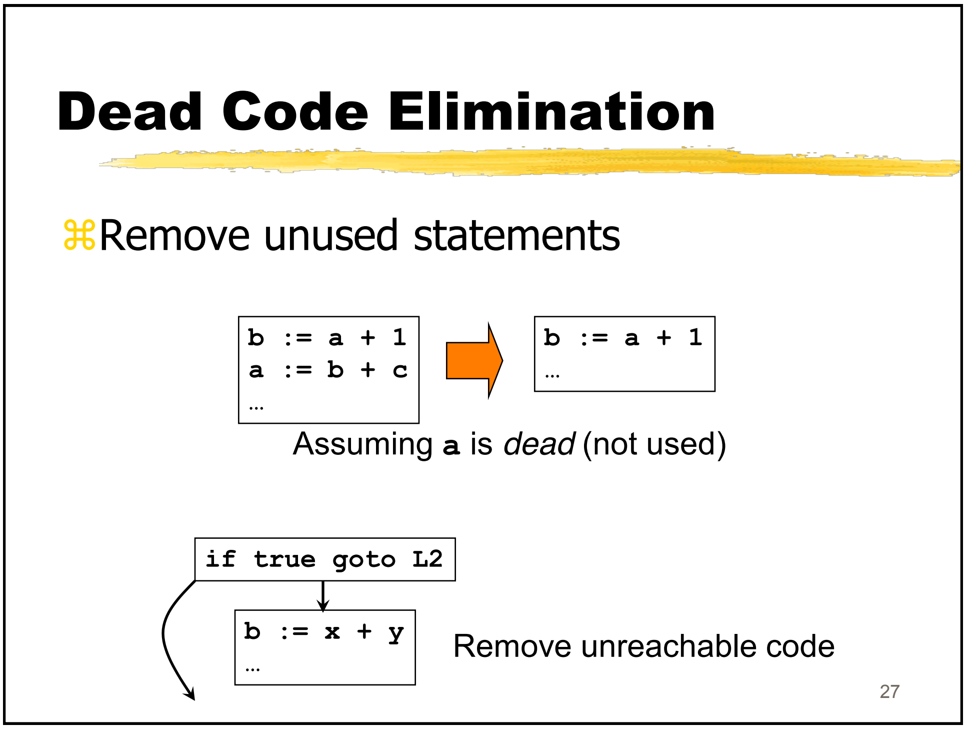
```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

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

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

```
t1  := b * c
t2  := a - t1
t4  := t2 + t1
```

26



Dead Code Elimination

- ⌘Remove unused statements

```
b := a + 1
a := b + c
…
```

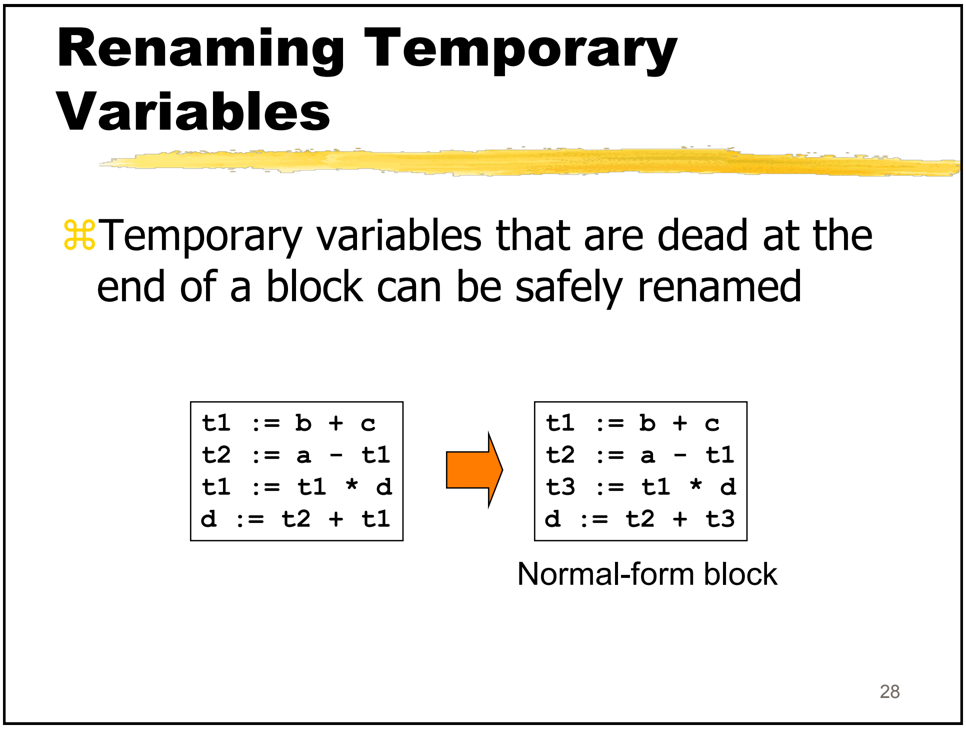
```
b := a + 1
…
```

Assuming **a** is *dead* (not used)

```
if true goto L2
```

```
b := x + y
…
```

Remove unreachable code



Renaming Temporary Variables

- ⌘Temporary variables that are dead at the end of a block can be safely renamed

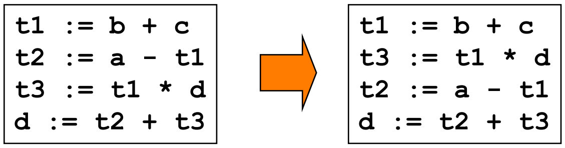
```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

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

Normal-form block

Interchange of Statements

- Independent statements can be reordered



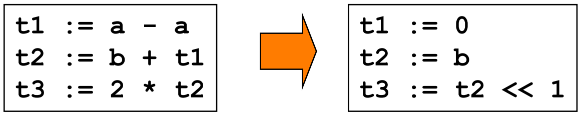
```
t1  := b + c
t2  := a - t1
t3  := t1 * d
d  := t2 + t3
```

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

Note that normal-form blocks permit all statement interchanges that are possible

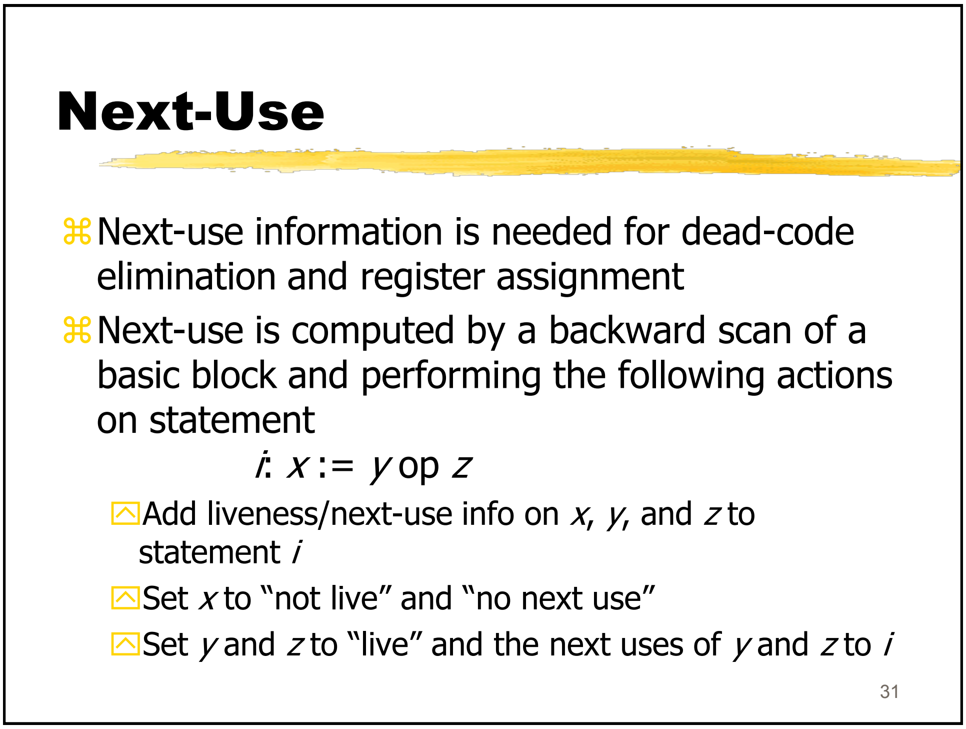
Algebraic Transformations

- Change arithmetic operations to transform blocks to algebraic equivalent forms



```
t1  := a - a
t2  := b + t1
t3  := 2 * t2
```

```
t1  := 0
t2  := b
t3  := t2 << 1
```

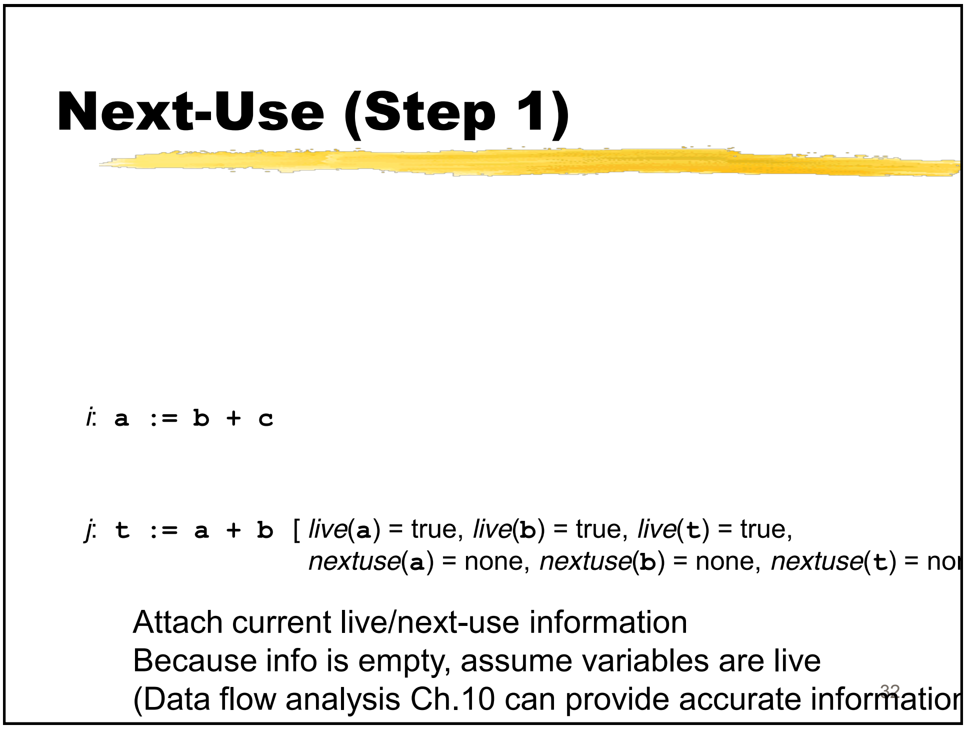


Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

  i: *x* := *y* op *z*

  - Add liveness/next-use info on *x*, *y*, and *z* to statement *i*
  - Set *x* to "not live" and "no next use"
  - Set *y* and *z* to "live" and the next uses of *y* and *z* to *i*



Next-Use (Step 1)

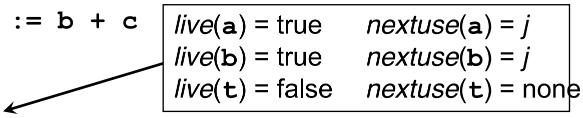
```
i: a := b + c


j: t := a + b
```

[*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = nor

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate informatior

Next-Use (Step 2)

i: `a := b + c`



live(**a**) = true *nextuse*(**a**) = *j*
live(**b**) = true *nextuse*(**b**) = *j*
live(**t**) = false *nextuse*(**t**) = none

j: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = nor

Compute live/next-use information at *j*

33

Next-Use (Step 3)

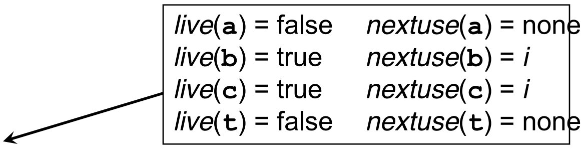
i: `a := b + c` [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: `t := a + b` [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = nor

Attach current live/next-use information to *i*

34

Next-Use (Step 4)



live(**a**) = false *nextuse*(**a**) = none
live(**b**) = true *nextuse*(**b**) = *i*
live(**c**) = true *nextuse*(**c**) = *i*
live(**t**) = false *nextuse*(**t**) = none

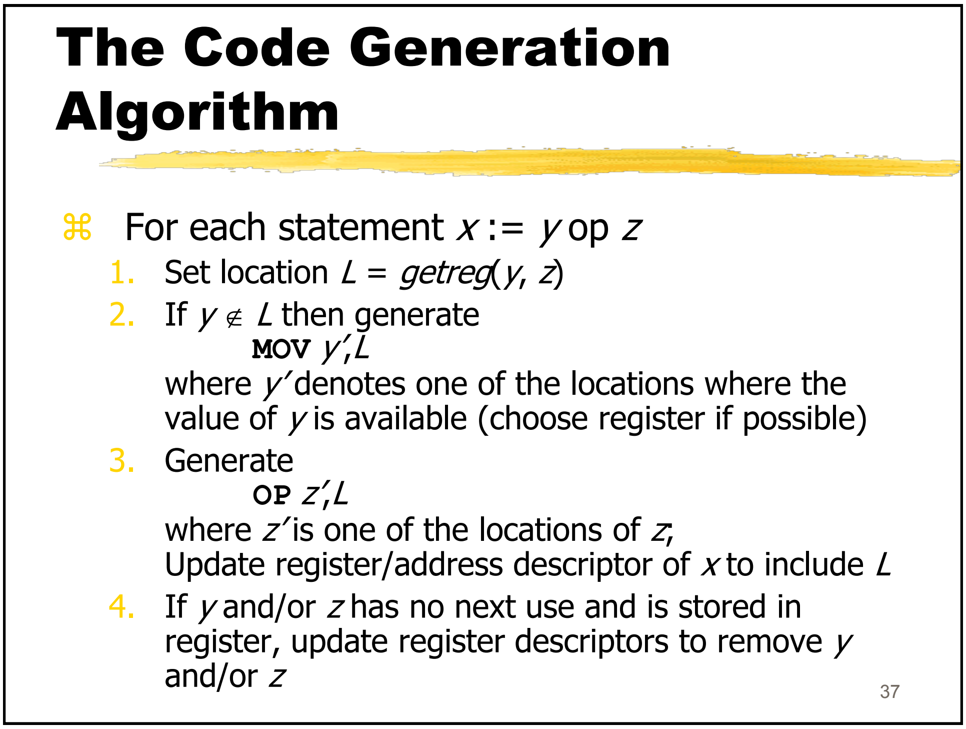
```
i: a := b + c
```
[*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

```
j: t := a + b
```
[*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = nor

Compute live/next-use information *i*

A Code Generator

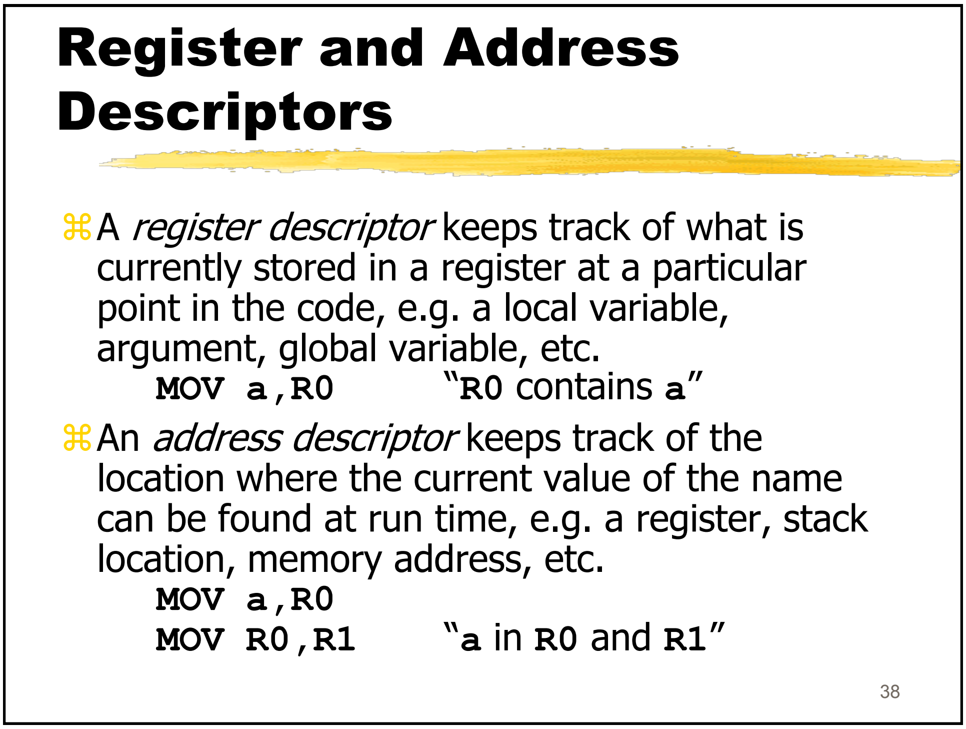
- Generates target code for a sequence of three-address statements using next-use information
- Uses new function *getreg* to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers



The Code Generation Algorithm

⌘ For each statement $x := y$ op z

1. Set location L = *getreg*(y, z)
2. If $y \notin L$ then generate
   MOV y',L
   where y' denotes one of the locations where the value of y is available (choose register if possible)
3. Generate
   OP z',L
   where z' is one of the locations of z;
   Update register/address descriptor of x to include L
4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z



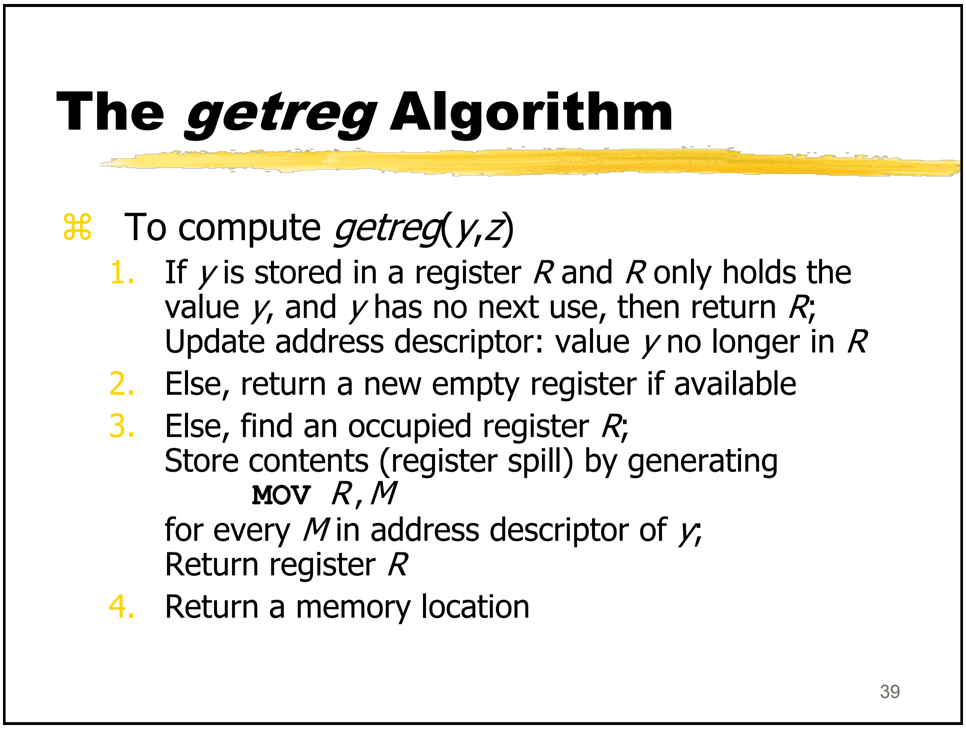
Register and Address Descriptors

⌘ A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

```
MOV a,R0        "R0 contains a"
```

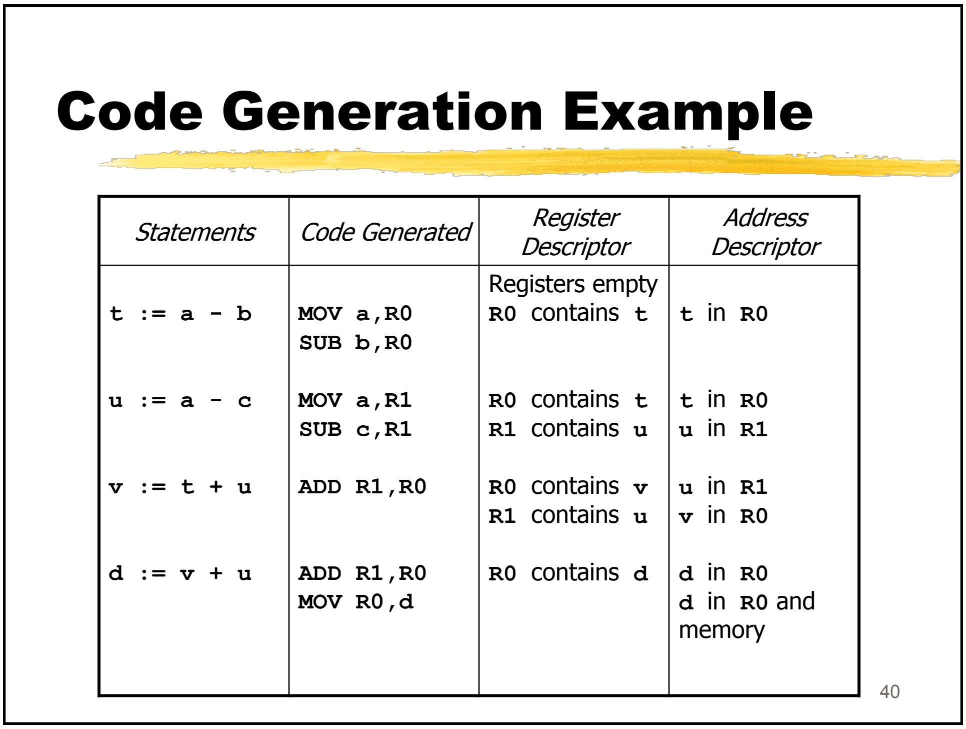
⌘ An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

```
MOV a,R0
MOV R0,R1       "a in R0 and R1"
```



The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
Update address descriptor: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
Store contents (register spill) by generating
`MOV` *R*,*M*
for every *M* in address descriptor of *y*;
Return register *R*
  4. Return a memory location



Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	

`t := a - b`	`MOV a,R0`
	`SUB b,R0`
		`R0` contains `t`	`t` in `R0`

`u := a - c`	`MOV a,R1`
	`SUB c,R1`
		`R0` contains `t`
		`R1` contains `u`
			`t` in `R0`
			`u` in `R1`

`v := t + u`	`ADD R1,R0`	`R0` contains `v`
		`R1` contains `u`
			`u` in `R1`
			`v` in `R0`

`d := v + u`	`ADD R1,R0`
	`MOV R0,d`
		`R0` contains `d`	`d` in `R0`
			`d` in `R0` and memory

Register Allocation and Assignment

- The *getreg* algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

Allocating Registers in Loops

- Suppose loading a variable x has a cost of 2
- Suppose storing a variable x has a cost of 2
- Benefit of allocating a register to a variable x within a loop L is

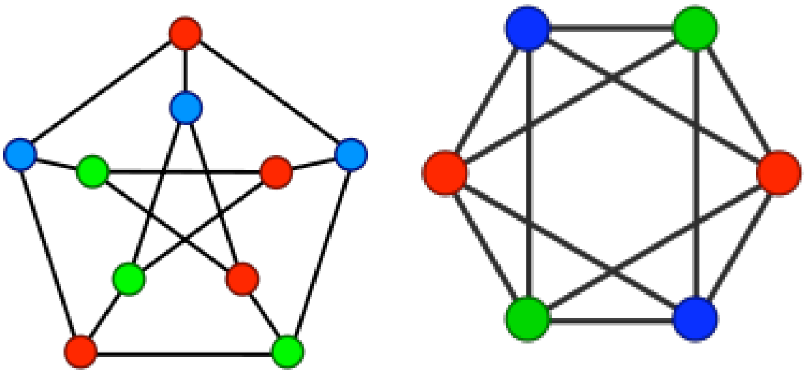
  $$\sum_{B \in L} \left(\mathit{use}(x, B) + 2\ \mathit{live}(x, B) \right)$$

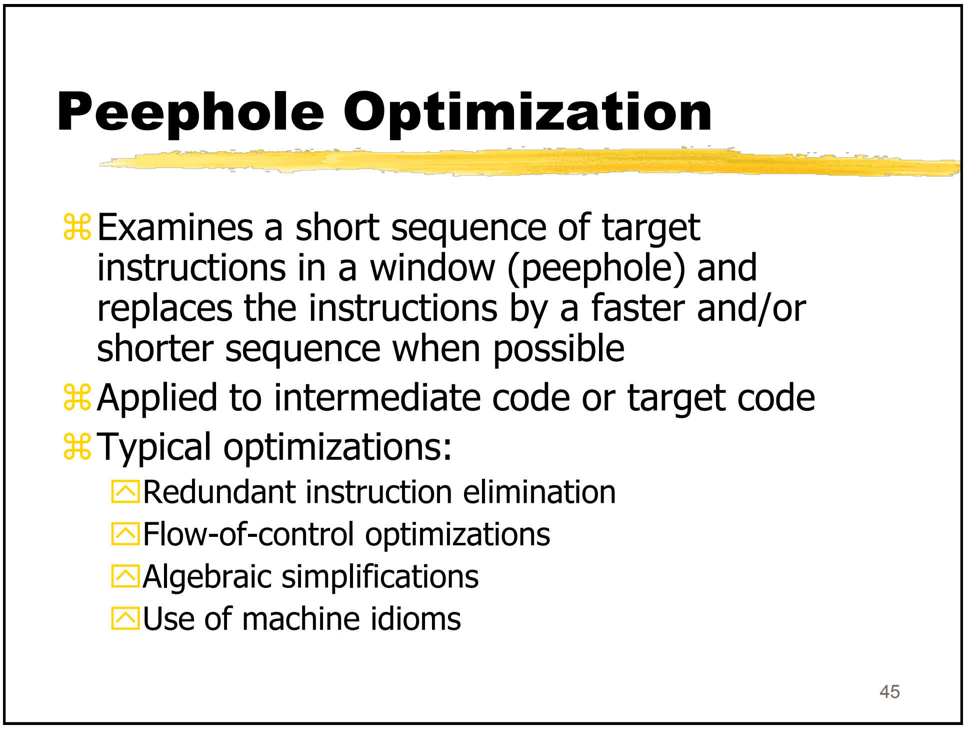
  where $\mathit{use}(x, B)$ is the number of times x is used in B and $\mathit{live}(x, B)$ = true if x is live on exit from B

Global Register Allocation Using Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers

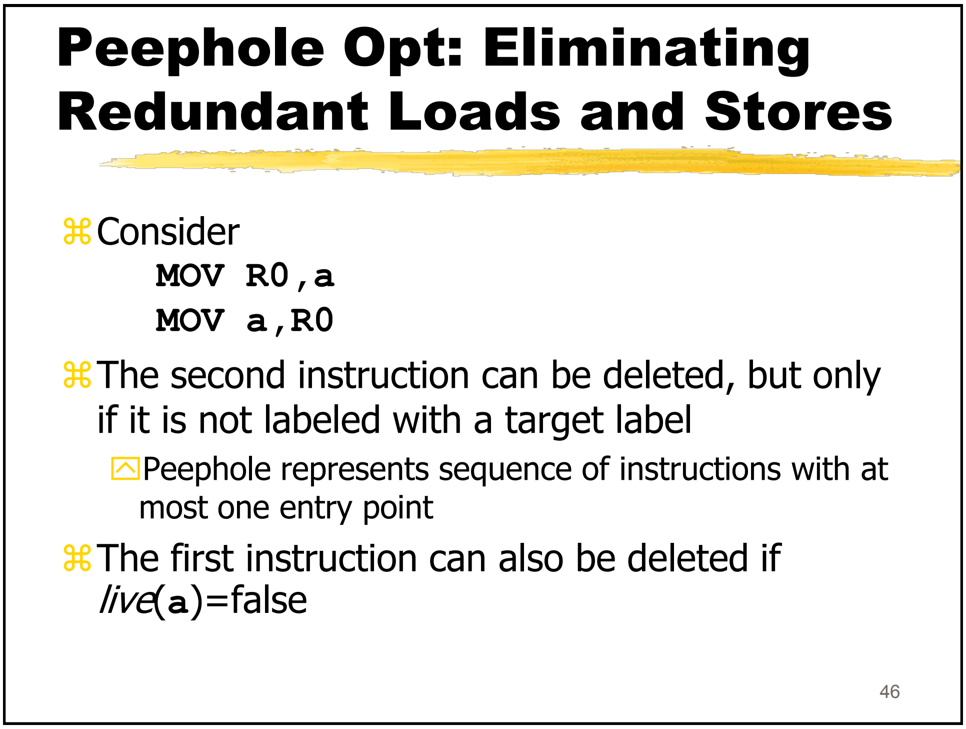
Graph Coloring Example





Peephole Optimization

- Examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence when possible
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms



Peephole Opt: Eliminating Redundant Loads and Stores

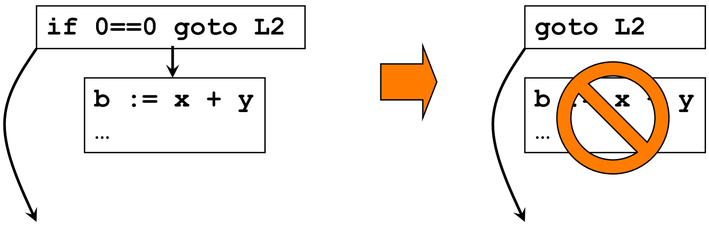
- Consider

```
    MOV R0,a
    MOV a,R0
```

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**)=false

Peephole Optimization: Deleting Unreachable Code

⌘Unlabeled blocks can be removed

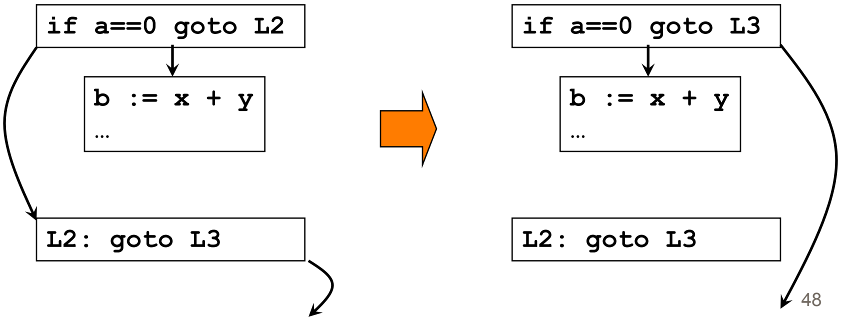


```
if 0==0 goto L2
b := x + y
…
```

```
goto L2
b := x + y
…
```

Peephole Optimization: Branch Chaining

⌘Shorten chain of branches by modifying target labels

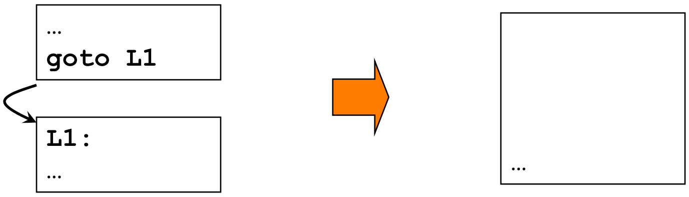


```
if a==0 goto L2
b := x + y
…
L2: goto L3
```

```
if a==0 goto L3
b := x + y
…
L2: goto L3
```

Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps



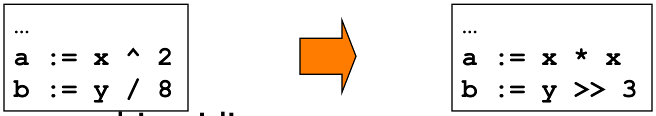
```
…
goto L1
```

```
L1:
…
```

```
…
```

Other Peephole Optimizations

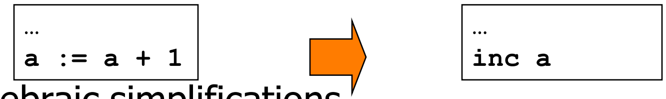
- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones



```
…
a := x ^ 2
b := y / 8
```

```
…
a := x * x
b := y >> 3
```

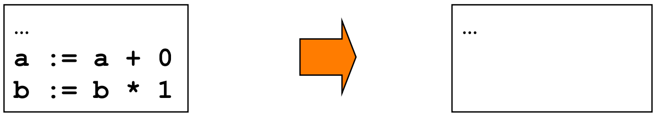
- Utilize machine idioms



```
…
a := a + 1
```

```
…
inc a
```

- Algebraic simplifications



```
…
a := a + 0
b := b * 1
```

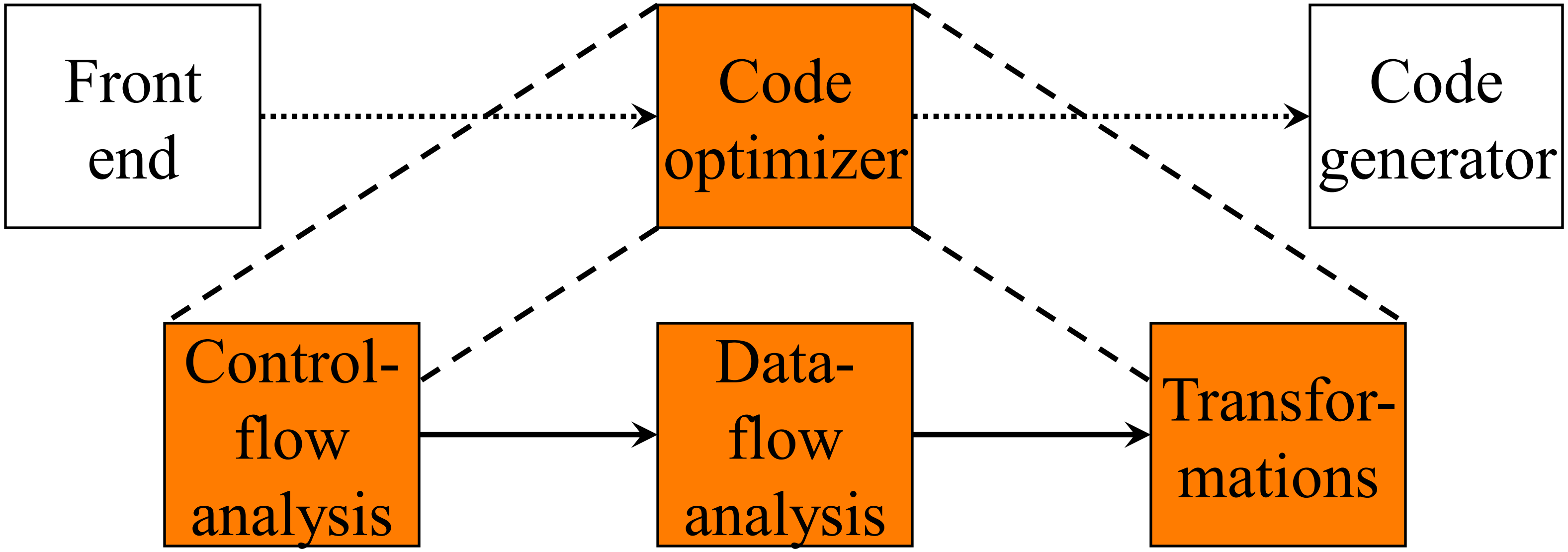
```
…
```

Code Optimization

Bart Kienhuis
Computer Systems Group
University Leiden (LIACS)

The Code Optimizer

- Control flow analysis: CFG (Ch. 9)
- Data-flow analysis
- Transformations



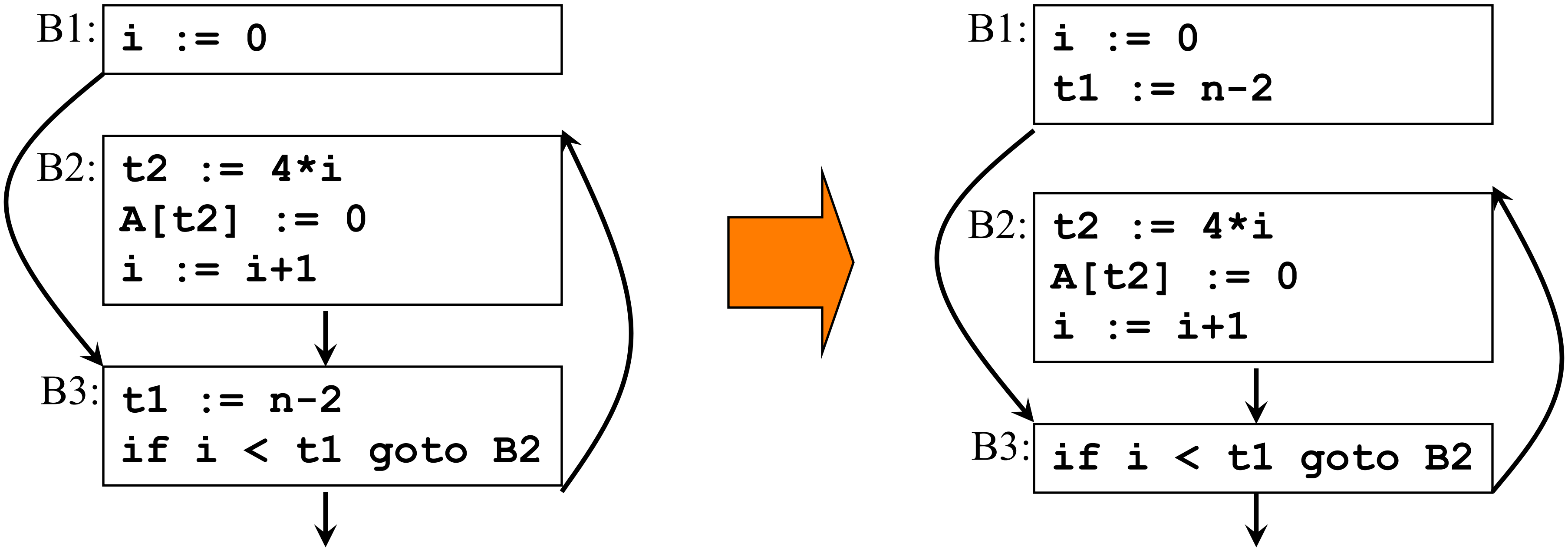
Code Optimizations

- Local/global common subexpression elimination
- Dead-code elimination
- Instruction reordering
- Constant folding
- Algebraic transformations
- Copy propagation
- *Loop optimizations*

Loop Optimizations

- Code motion
- Induction variable elimination
- Reduction in strength
- … lots more

Code Motion

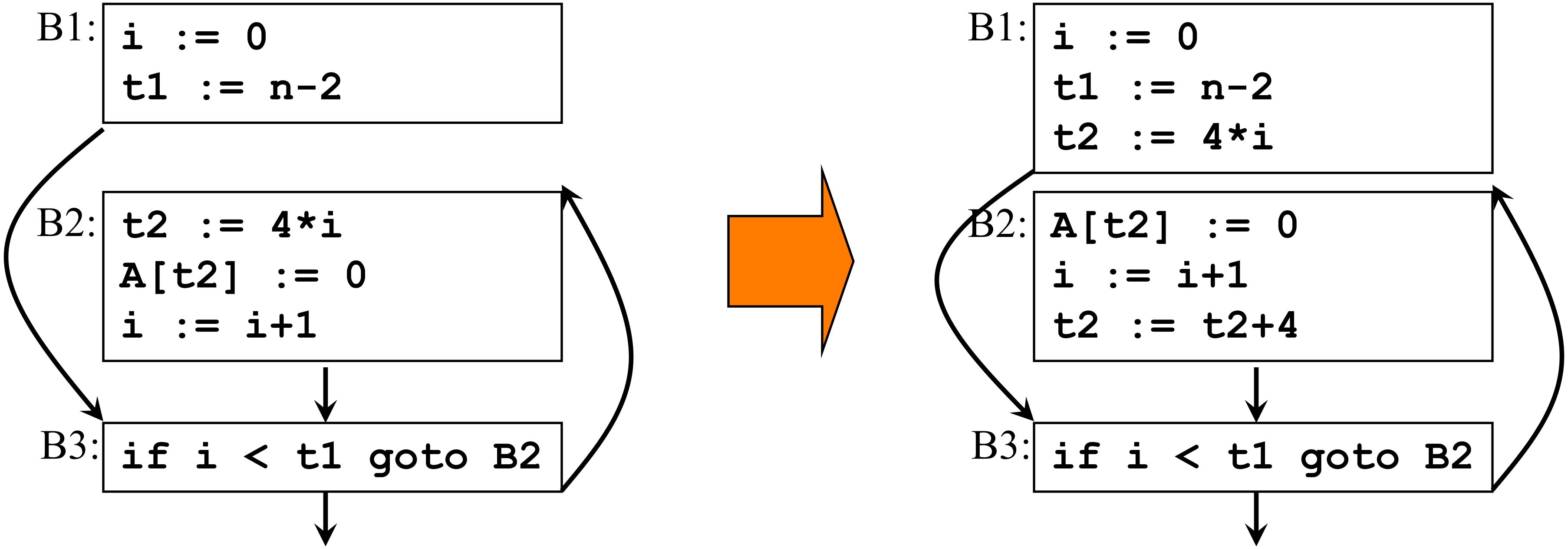


```
B1: i := 0

B2: t2 := 4*i
    A[t2] := 0
    i := i+1

B3: t1 := n-2
    if i < t1 goto B2
```

```
B1: i := 0
    t1 := n-2

B2: t2 := 4*i
    A[t2] := 0
    i := i+1

B3: if i < t1 goto B2
```

Move *loop-invariant computations* before the loop

Strength Reduction

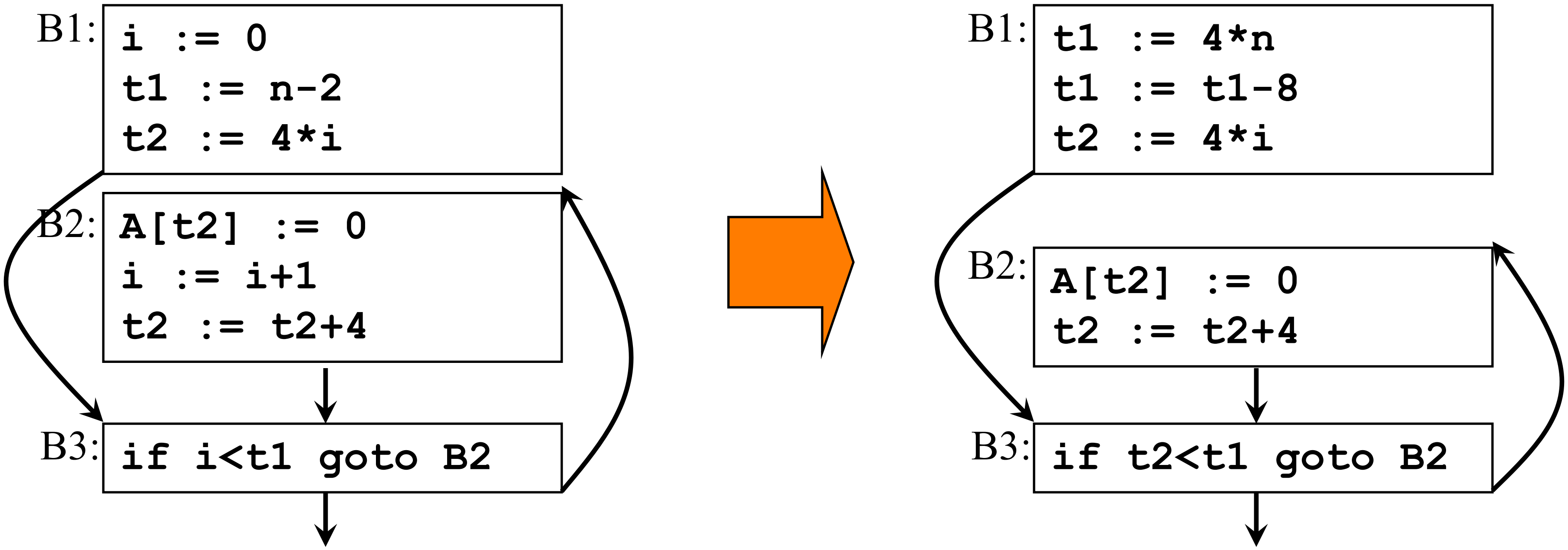


```
B1: i  := 0
    t1  := n-2

B2: t2  := 4*i
    A[t2]  := 0
    i  := i+1

B3: if i < t1 goto B2
```

```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i < t1 goto B2
```

Replace expensive computations with *induction variables*

Reduction Variable Elimination



```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i<t1 goto B2
```

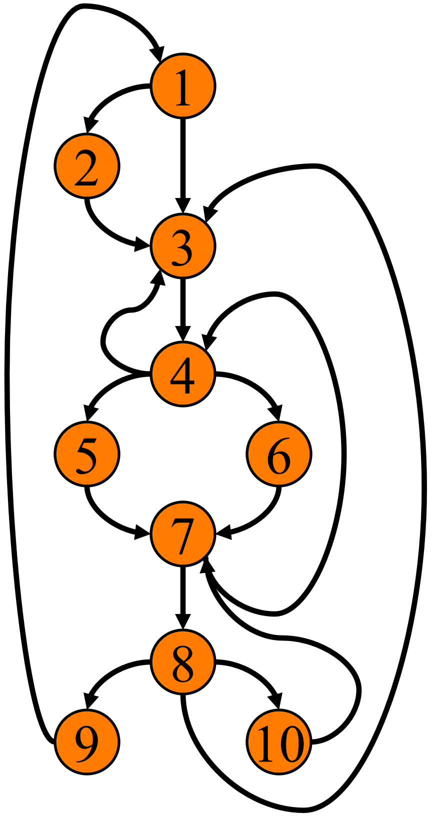
```
B1: t1  := 4*n
    t1  := t1-8
    t2  := 4*i

B2: A[t2]  := 0
    t2  := t2+4

B3: if t2<t1 goto B2
```

Replace induction variable in expressions with another

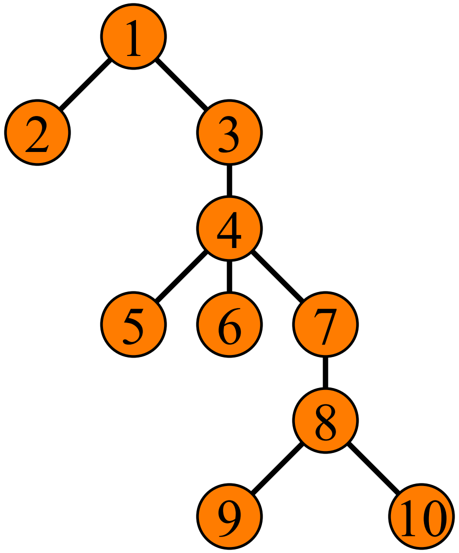
Determining Loops in Flow Graphs: Dominators

- Dominators: *d dom n*
  - Node *d* of a CFG *dominates* node *n* if *every* path from the initial node of the CFG to *n* goes through *d*
  - The loop entry dominates all nodes in the loop
- The *immediate dominator m* of a node *n* is the last dominator on the path from the initial node to *n*
  - If $d \neq n$ and *d dom n* then *d dom m*

Dominator Trees



CFG

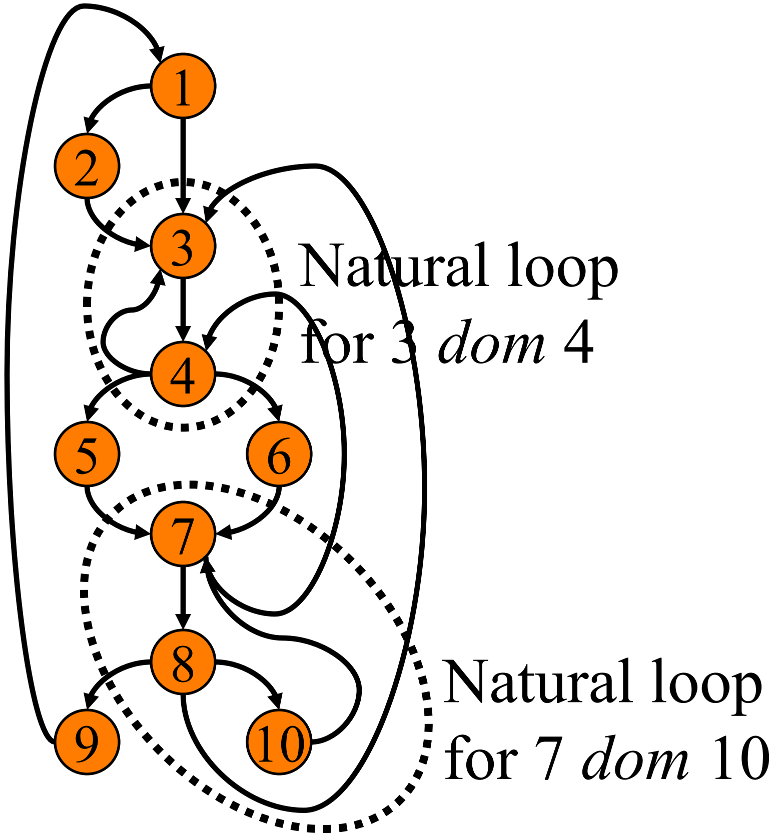


Dominator tree

Natural Loops

- A *back edge* is is an edge $a \to b$ whose head b dominates its tail a
- Given a back edge $n \to d$
  - The *natural loop* consists of d plus the nodes that can reach n without going through d
  - The *loop header* is node d
- Unless two loops have the same header, they are disjoint or one is nested within the other
  - A nested loop is an *inner loop* if it contains no other loops

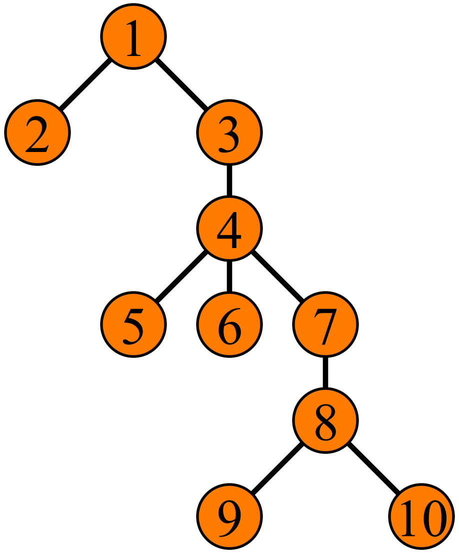
Natural (Inner) Loops Example



Natural loop for 3 *dom* 4

Natural loop for 7 *dom* 10

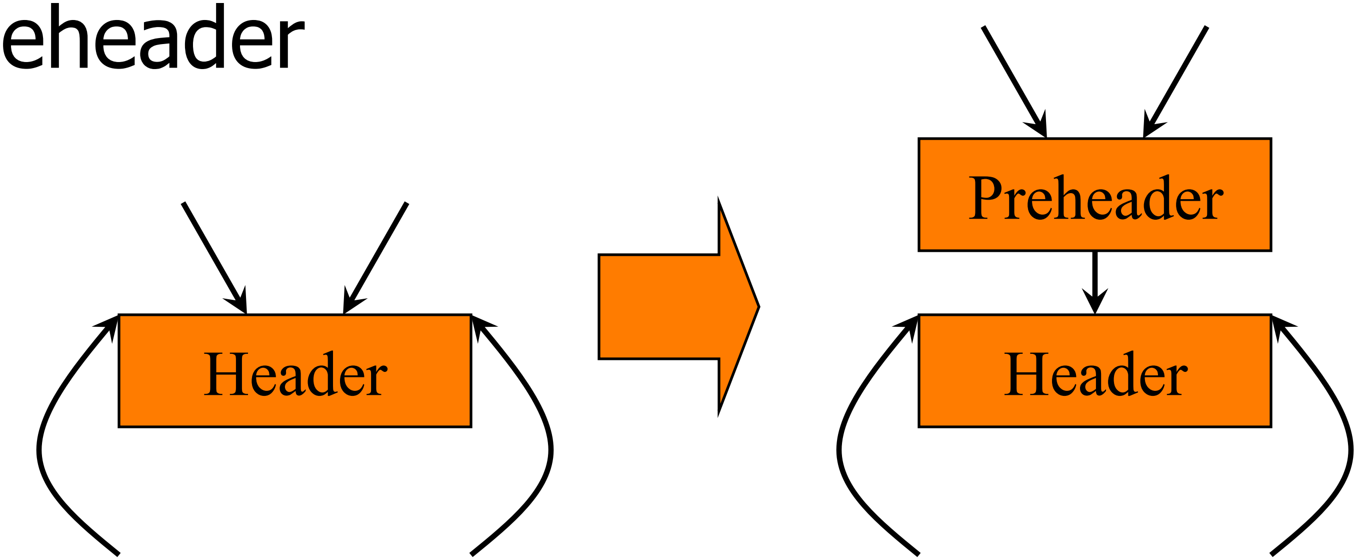
CFG



Dominator tree

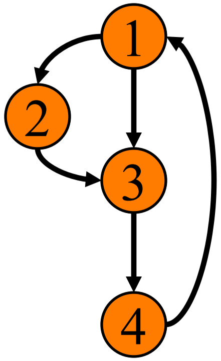
Pre-Headers

- To facilitate loop transformations, a compiler often adds a *preheader* to a loop
- Code motion, strength reduction, and other loop transformations populate the preheader

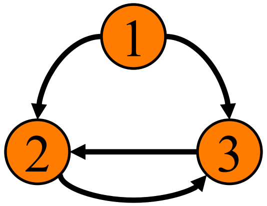


Reducible Flow Graphs

- *Reducible graph* = disjoint partition in forward and back edges such that the forward edges form an acyclic (sub)graph



Example of a
reducible CFG

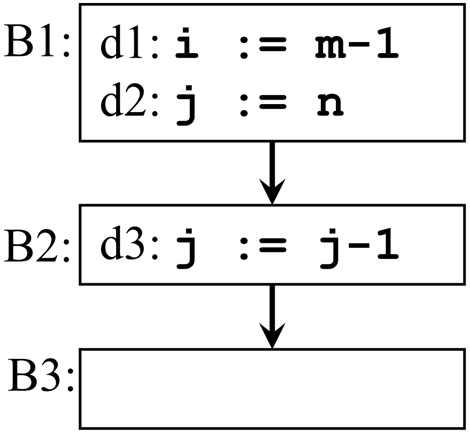


Example of a
nonreducible CFG

Global Data-Flow Analysis

- To apply global optimizations on basic blocks, *data-flow information* is collected by solving systems of *data-flow equations*
- Suppose we need to determine the *reaching definitions* for a sequence of statements *S*

$$out[S] = gen[S] \cup (in[S] - kill[S])$$



B1: d1: `i := m-1`
d2: `j := n`

B2: d3: `j := j-1`

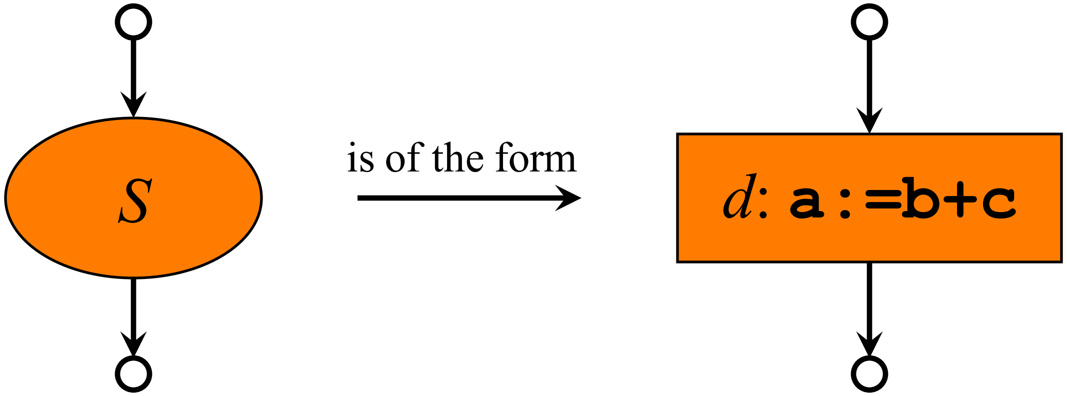
B3:

$out[\mathrm{B1}] = gen[\mathrm{B1}] = \{\mathrm{d1}, \mathrm{d2}\}$

$out[\mathrm{B2}] = gen[\mathrm{B2}] \cup \{\mathrm{d1}\} = \{\mathrm{d1}, \mathrm{d3}\}$

d1 reaches B2 and B3 and
d2 reaches B2, but not B3
because d2 is killed in B2

Reaching Definitions

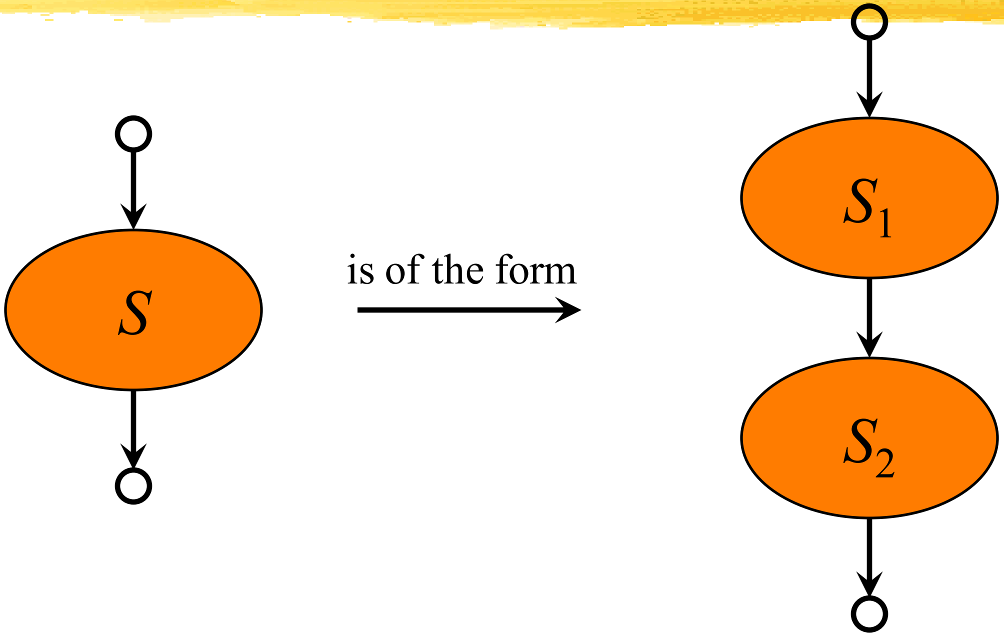


Then, the data-flow equations for S are:

$$gen[S] = \{d\}$$
$$kill[S] = D_{\mathtt{a}} - \{d\}$$
$$out[S] = gen[S] \cup (in[S] - kill[S])$$

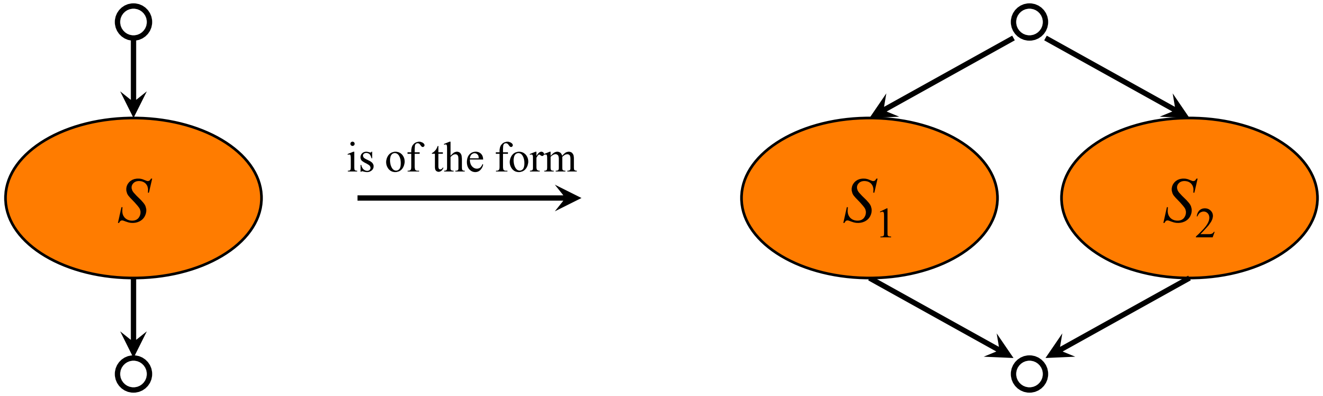
where $D_{\mathtt{a}}$ = all definitions of **`a`** in the region of code

Reaching Definitions



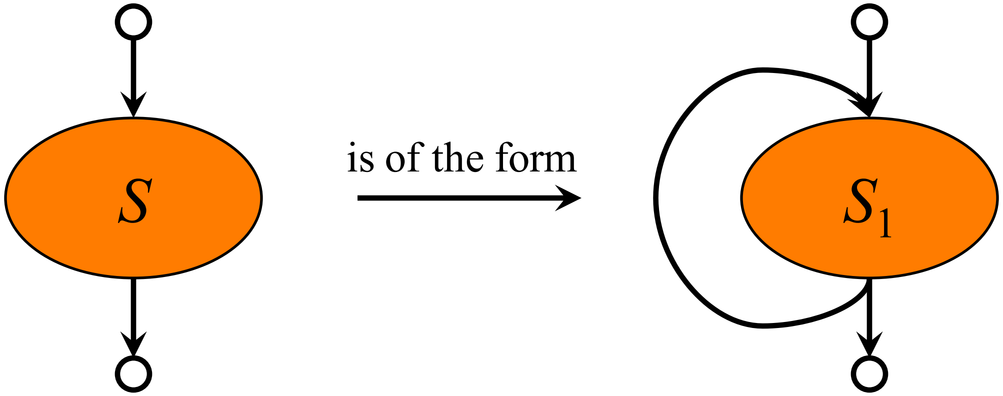
$$
\begin{aligned}
gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\
kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\
in[S_1] &= in[S] \\
in[S_2] &= out[S_1] \\
out[S] &= out[S_2]
\end{aligned}
$$

Reaching Definitions



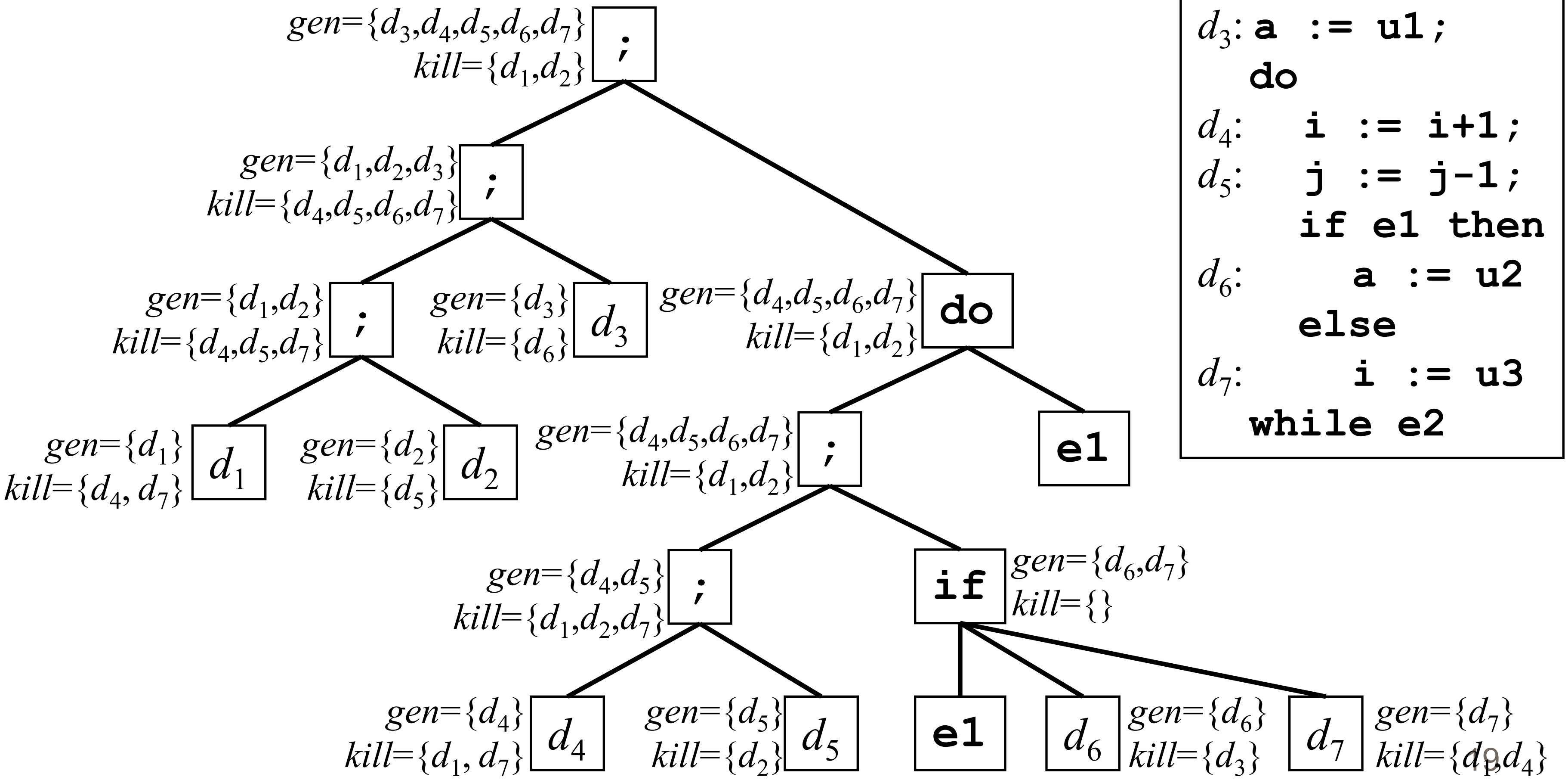
$$
\begin{aligned}
gen[S] &= gen[S_1] \cup gen[S_2] \\
kill[S] &= kill[S_1] \cap kill[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{aligned}
$$

Reaching Definitions



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

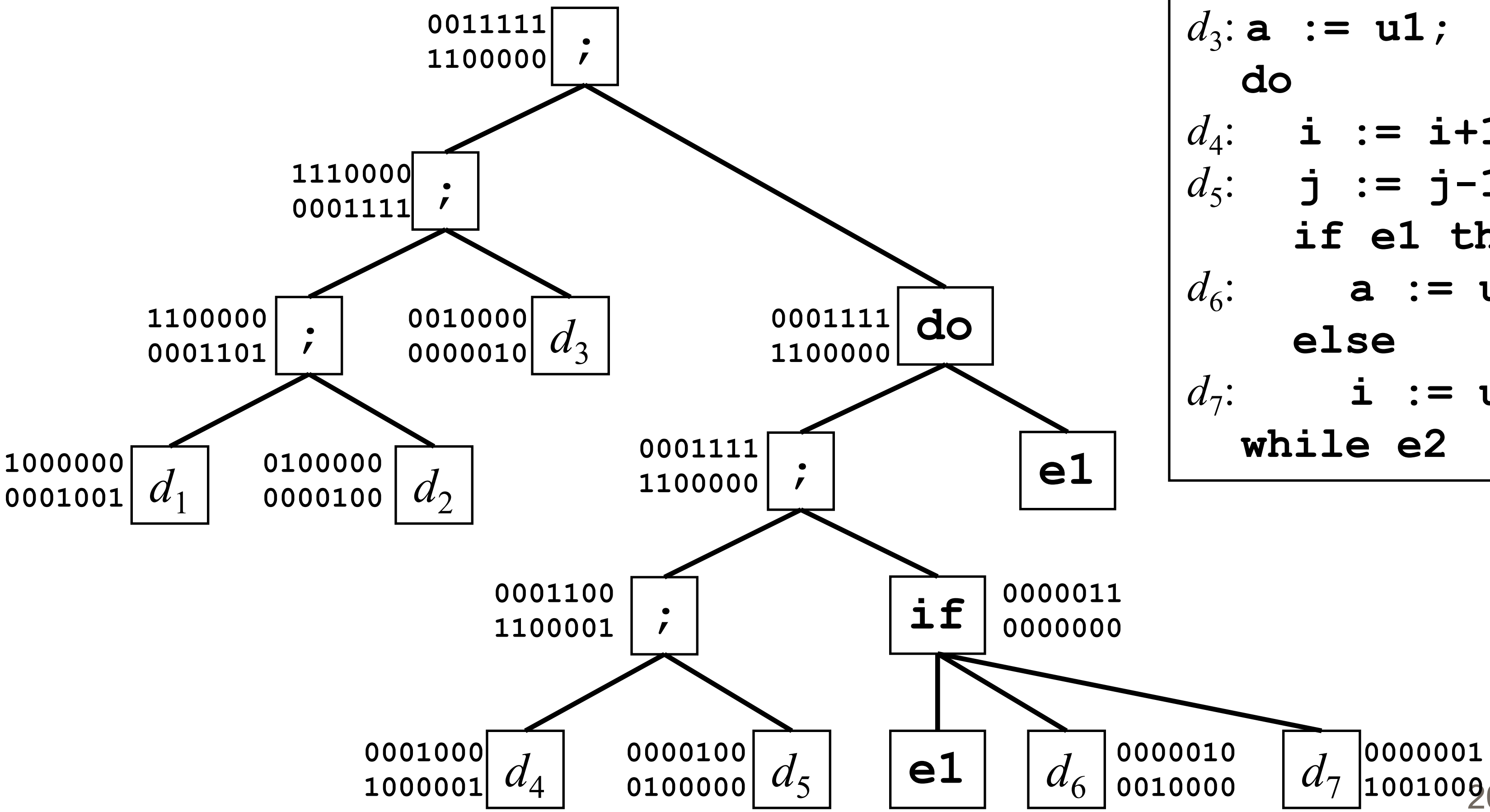
Example Reaching Definitions



```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:     i := i+1;
d5:     j := j-1;
        if e1 then
d6:         a := u2
        else
d7:         i := u3
    while e2
```

Using Bit-Vectors to Compute Reaching Definitions

```
d1: i := m-1;
d2: j := n;
d3: a := u1;
    do
d4:     i := i+1;
d5:     j := j-1;
        if e1 then
d6:         a := u2
        else
d7:         i := u3
    while e2
```



0011111
1100000
;

1110000
0001111
;

1100000
0001101
;

0010000
0000010
d_3

0001111
1100000
do

1000000
0001001
d_1

0100000
0000100
d_2

0001111
1100000
;

e1

0001100
1100001
;

if
0000011
0000000

0001000
1000001
d_4

0000100
0100000
d_5

e1

d_6
0000010
0010000

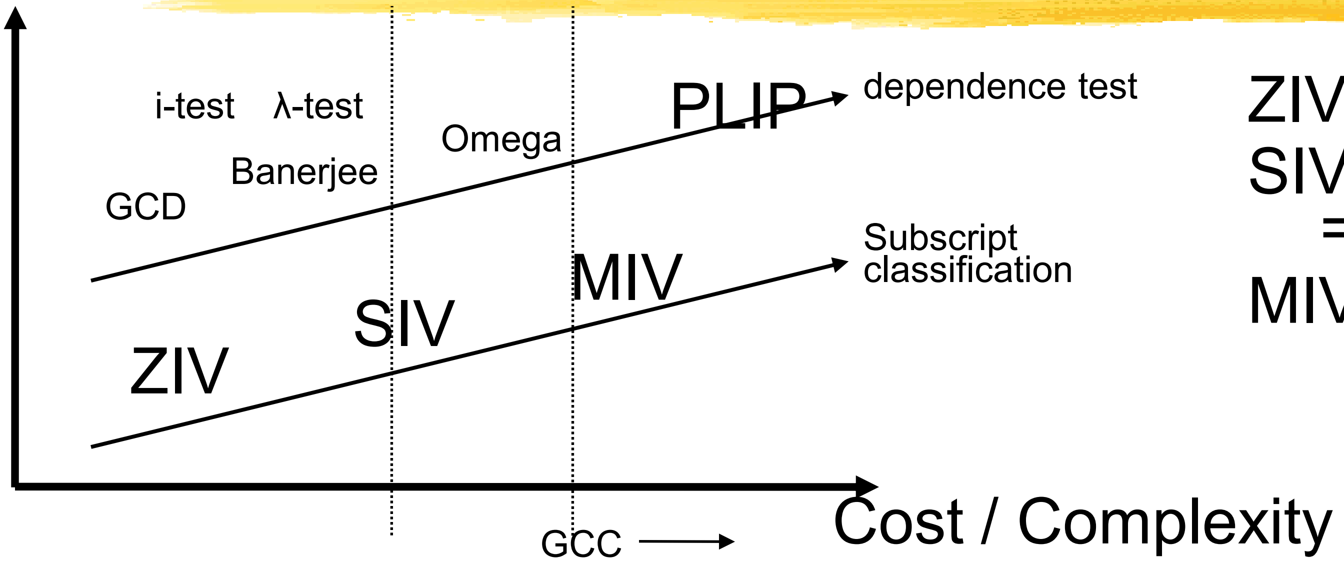
d_7
0000001
1001000

QR Algorithm – Smart Antenna

Matlab Code (QR Algorithm)

```
%parameter N 8 16;
%parameter K 100 1000;

for k = 1:1:K,
    for j = 1:1:N,
        [ r(j,j), x(k,j), t ]=Vectorize( r(j,j), x(k,j) );
        for i = j+1:1:N,
            [ r(j,i), x(k,i), t]=Rotate( r(j,i), x(k,i), t );
       end
    end
end
```

Data Dependence Tests



ZIV: a[3] = a[5]

SIV: a[i] = a[2], a[i] = a[i]

MIV: a[i][j] = a[j][i+j]

GCD, Banerjee, i-test, λ-test, cannot handle:

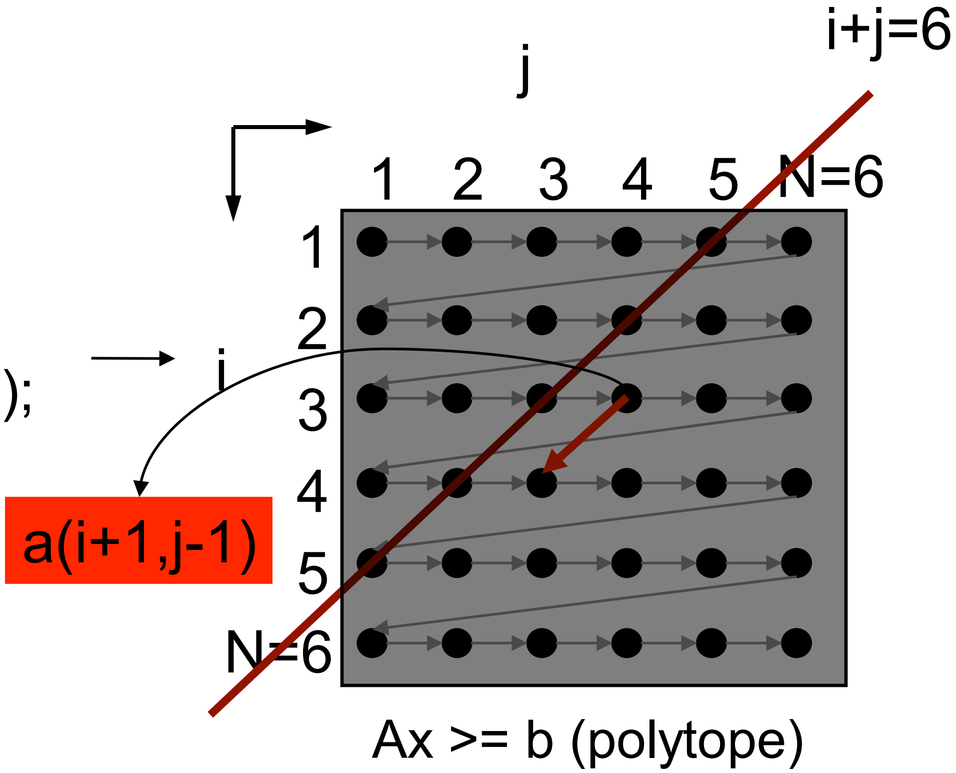
- if conditionals
- parametric loop bounds
- coupled subscripts
- parametric subscripts

Omega, PLIP:

- Exact data dependencies
- Omega: Fourier-Motzkin
- PLIP: dual-simplex method, more precise with parametric codes

Exact Dependency Analysis

```
for i= 1 : 1 : N,
    for j= 1 : 1 : N,
       [ a(i+j) ] = funcA( a(i+j) );
    end
end
```



The for-next loops define an Iteration Domain

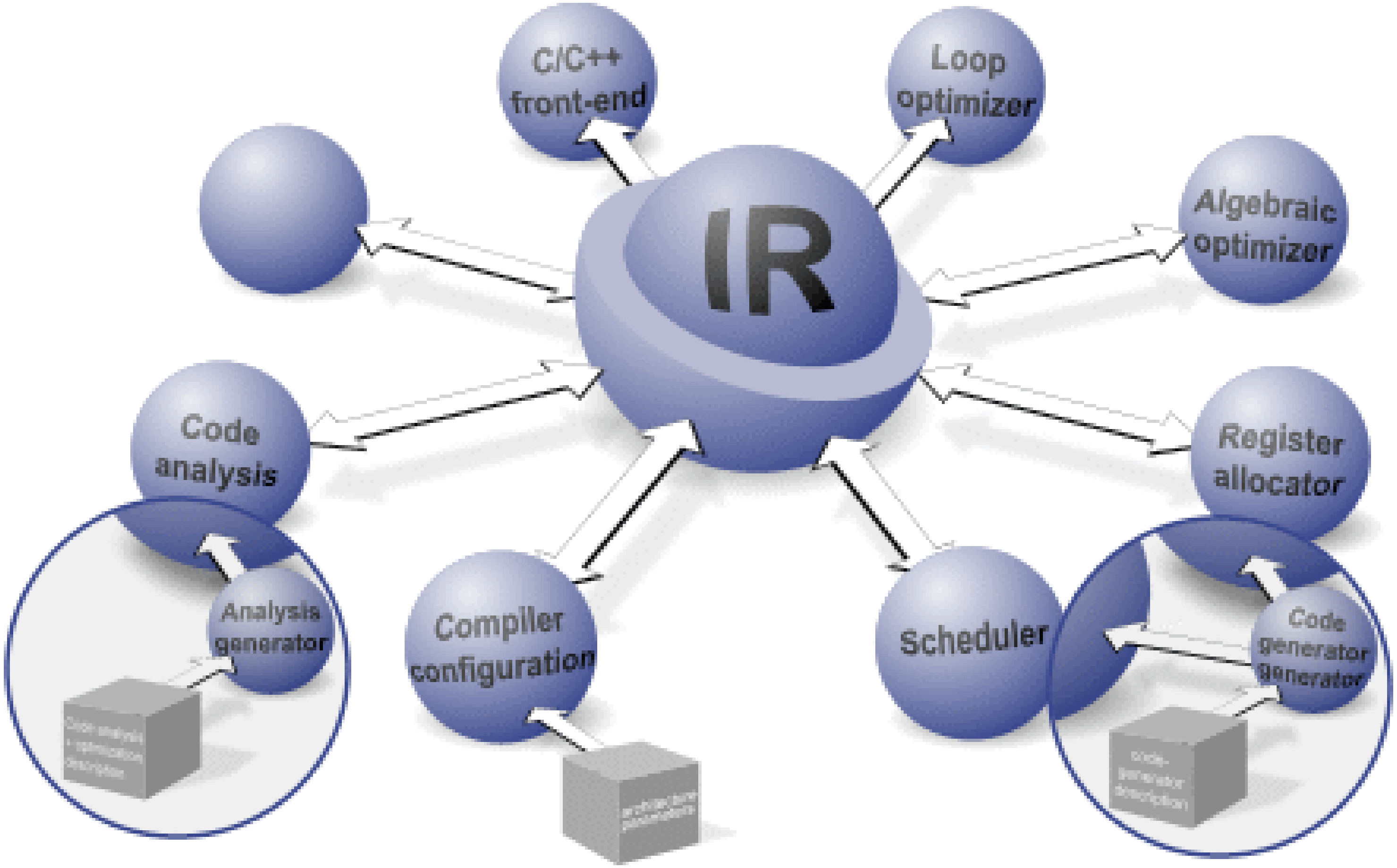
Many more optimizations

- Aliases analysis (pointers)
  - if two or more expressions denote the same memory address, the expressions are aliases of one another.

Compiler Frameworks

- Open Source
  - GCC
  - LLVM
  - Open64
  - SUIF
- Commercial
  - Target
  - Altrium
  - ACE
- In-house
  - Many

Compilers


C/C++ front-end
Loop optimizer
IR
Algebraic optimizer
Code analysis
Register allocator
Analysis generator
Code analysis + optimization description
Compiler configuration
architecture parameters
Scheduler
Code generator generator
Code-generator description