Compilation Summary Class

Mooly Sagiv

Advanced Topics (if interested please send email)

- Just in time compilation
- Compiler correctness

Topics

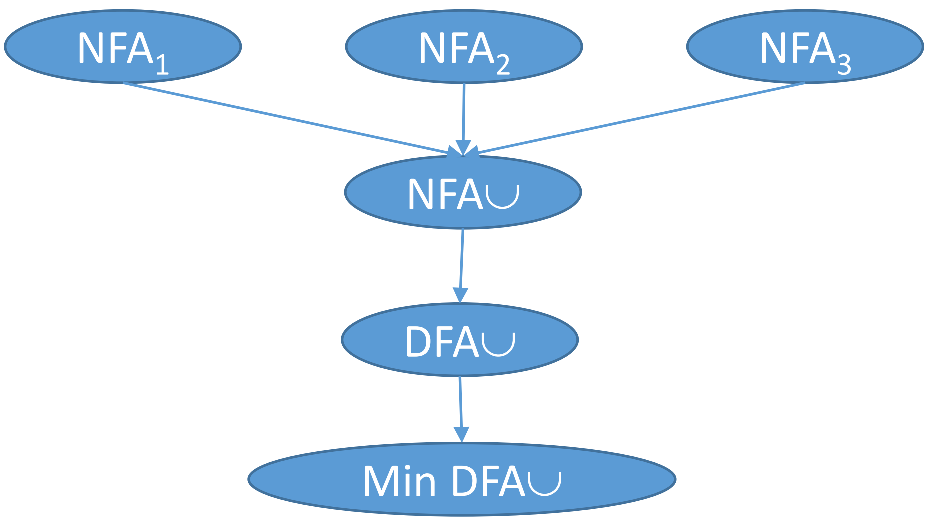
- Lexical Analysis
- Syntax Analysis
  - LL(1)
  - SLR(1)
- Semantic Analysis
- Intermediate Code Generation
- Register Allocation
- Machine code generation
- Assembler/Linker/Loader
- Memory allocation and Garbage Collection

Lexical Analysis (Scanning)

- input
  - program text (file)
- output
  - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- [Produce symbol table]

Generating Lexical Analysis

Token definition (Regular Expression)	Java Code for match
if	return(IF);
[a-z][a-z0-9]*	return(ID);
[0-9]*	return(NUM);



Maximal Match Scanner using DFA

```
Token nextToken()
{
lastFinal = 0;
currentState = 1 ;
inputPositionAtLastFinal = input;
currentPosition = input;
while (not(isDead(currentState)))  {
        nextState = edges[currentState][*currentPosition];
        if  (isFinal(nextState)) {
                lastFinal = nextState ;
                inputPositionAtLastFinal = currentPosition; }
        currentState = nextState;
        advance currentPosition;
        }
input = inputPositionAtLastFinal ;
return action[lastFinal];
}
```

Summary Lexical Analysis

- Based on regular expressions and automata theory
- Excellent open source tools LEX/FLEX/JLEX/…
- Some tricky implementation tricks
  - Input buffering
  - Maintain state in program counter

LL(1) Syntax Analysis

- Work for a restricted class of programs
- Identifies the left most derivation or syntax error
- The simplest implementation includes one (recursive) function for non-terminal
- The function for non-terminal A reads one token and **uniquely decides** on which of the productions of A to apply
  - $A \rightarrow \alpha$
  - $A \rightarrow \beta$
- A rule $A \rightarrow \alpha$ is applied for tokens $t \in \text{select}(A \rightarrow \alpha)$

Nullable Nonterminals

- A non-terminal is **nullable** if it can derive the empty word
- $A \Rightarrow^* \varepsilon$
- Computed iteratively
- Extended to right hand side of derivation

```
while changes do {
    for every rule of the form A ➔ X1 X2 … Xn
        where all Xi is nullable make X nullable
}
```

The set of terminals in which a grammar symbols may begin

- First(X) ={ t | X $\Rightarrow^*$ t α }
- Computed iteratively
- Extended to right hand sides of rules

Computing First Iteratively

For each token t, First(t) := {t}
For each non-terminal <A>, First(<A>) = {}

while changes occur do
    if there exists a non-terminal <A> and
        a rule <A> → $V_1 V_2 \ldots V_n \alpha$ and
        $V_1, V_2, \ldots, V_{n-1}$ are nullable
        a token t $\in$ First(V_n)
    add t to First(<A>)

The set of terminals first(α) for a string of grammar symbols α

- Tokens in which α **may** begin
- First(α) ={ t | $\alpha \Rightarrow^* \beta$}
- Computed iteratively

Computing First Iteratively

For the empty string, first(α) := {} for every rule A → α

while changes occur do
    if there exists a non-terminal <A> and
        a rule <A> → $V_1 V_2 \ldots V_n \alpha$ and
        $V_1, V_2, \ldots, V_{n-1}$ are nullable
        a token t $\in$ First(V_n)
    add t to First($V_1 V_2 \ldots V_n \alpha$)

The set of terminals which may follow a non-terminal

- For a non-terminal <A> define
  - Follow(<A>) = { t | $\exists\alpha, \beta$: <S> $\rightarrow^*$ α<A>tβ }
- For a rule <A> $\rightarrow$ α
  - If α is nullable then select(<A>$\rightarrow$ α) = First(α) $\cup$ Follow(<A>)
    - Otherwise select(<A>$\rightarrow$ α) = First(α)
- The grammar is LL(1) if for every two grammar rules <A> $\rightarrow$ α and <A> $\rightarrow$ β
  - Select(A $\rightarrow$ α) $\cap$ Select(A $\rightarrow$ β) = $\emptyset$

Computing Follow Iteratively

For each non-terminal A, Follow(A) := {}
Follow(S) = {\$}
while changes occur do
    for each rule V → α A β
        for each token t ∈ First(β) and
            add t to Follow(A)
        if β is nullable
            for each token t ∈ Follow(V)
            add t to Follow(A)

Predictive Parser

<S>→ **id** := <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → **id** <EP> | **(**<E>**)**
<EP> → ε | **+** <E> <EP>

```
def parse_S():
  if id(input):
    match(input, id)
    match(input, assign)
    parse_E()
  elif if_tok(input):
    match(input, if_tok)
    match(input, lp)
    parse_E()
    match(input, rp)
    parse_S()
    match(input, else_tok)
    parse_S()
  else:
    syntax_error()
```

```
def parse_E():
  if id(input):
    match(input, id)
    parse_EP()
  elif lp(input):
    match(input, lp)
    parse_E()
    match(input, rp)
  else:
    syntax_error()
```

```
def parse_EP():
  if plus(input):
    match(input, plus)
    parse_E()
    parse_EP()
  elif
    rp(input) or
    else_tok(input) or
    eof(input):
    return // ε
  else:
    syntax_error()
```

1:<S>→ **id :=** <E>
2:<S> → **if (** <E>**)** <S> **else** <S>
3: <E> → **id** <EP>
4: <E> → **(**<E>**)**
5: <EP> → ε
6: <EP> → **+** <E> <EP>

First

S	E	EP
id, if	id, (	+

Follow

S	E	EP
$, else	$,) , else	$,) , else

Select

rule	α	First(α)	Select(α)
1	**id** := <E>	id	id
2	**if (**<E>**)** <S> **else** <S>	**if**	**if**
3	**id** <EP>	**id**	**id**
4	**(**<E>**)**	**(**	**(**
5	ε		$,) , else

Predictive Parser

```
<S> → id := <E>
<S> → if (<E>) <S> else <S>
<E> → id <EP> | (<E>)
<EP> → ε | + <E> <EP>
```

```
def parse_S():
  if id(input):
    match(input, id)
    match(input, assign)
    parse_E()
  elif if_tok(input):
    match(input, if_tok)
    match(input, lp)
    parse_E()
    match(input, rp)
    parse_S()
    match(input, else_tok)
    parse_S()
  else:
    syntax_error()
```

```
def parse_E():
  if id(input):
    match(input, id)
    parse_EP()
  elif lp(input):
    match(input, lp)
    parse_E()
    match(input, rp)
  else:
    syntax_error()
```

```
def parse_EP():
  if plus(input):
    match(input, plus)
    parse_E()
    parse_EP()
  elif
    rp(input) or
    else_tok(input) or
    eof(input):
    return // ε
  else:
    syntax_error()
```

Bottom-Up Syntax Analysis

- Input
  - A context free grammar
  - A stream of tokens
- Output
  - A syntax tree or error
- Method
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in (reversed order)
  - For every potential right hand side and token decide when a production is found
  - Report an error as soon as the input is not a prefix of valid program

Constructing LR(0) parsing table

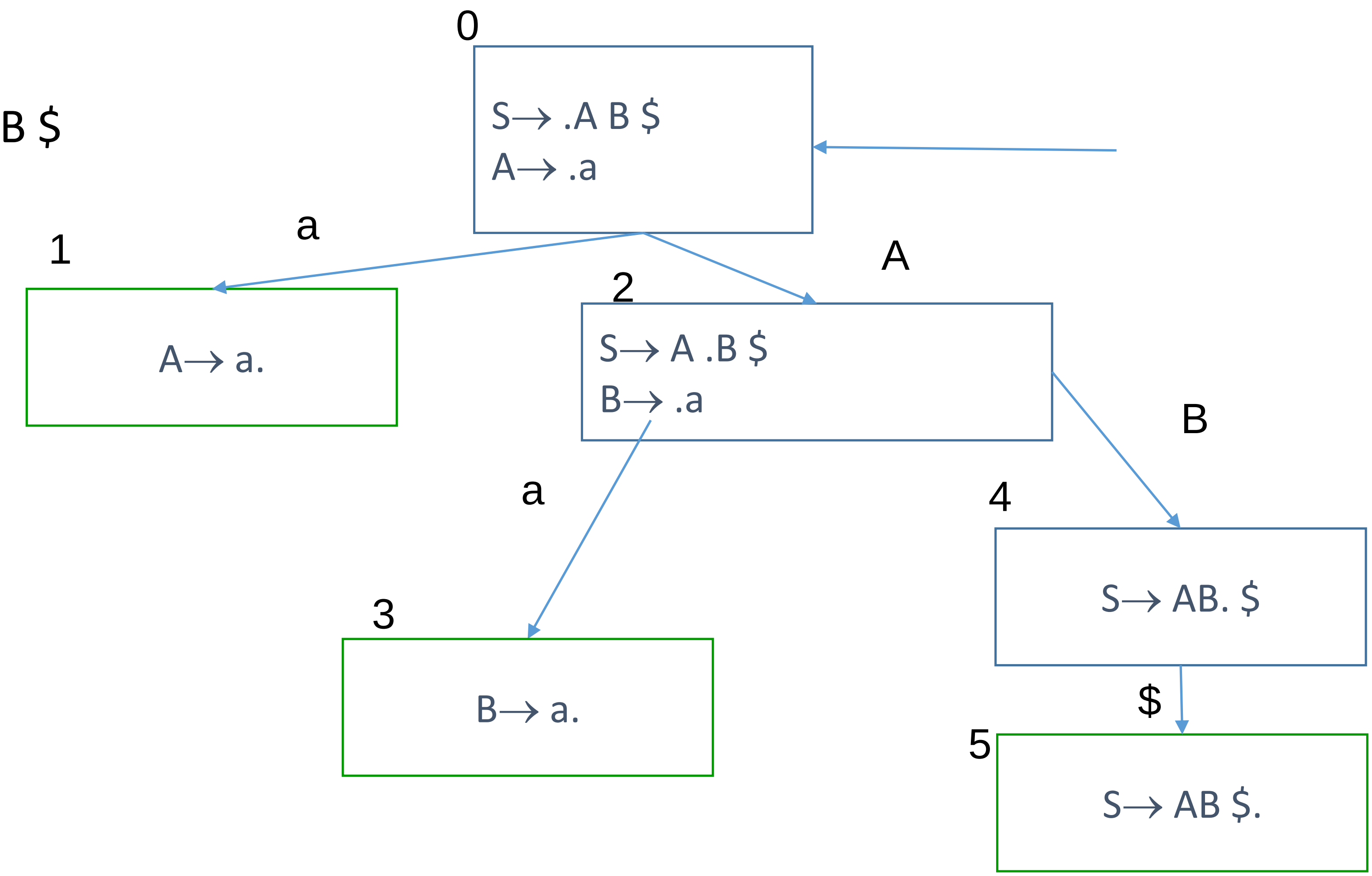
- Add a production $S' \rightarrow S\$$
- Construct a deterministic finite automaton accepting "valid stack symbols"
  - States are set of items $A \rightarrow \alpha \bullet \beta$
    - The initial state includes $S' \rightarrow \bullet S$ and its epsilon closure
    - All the states are accepting (excluding the sink which not shown)
    - There is a transition labeled by X between states include $A \rightarrow \alpha \bullet X \beta$ and the epsilon closure of $A \rightarrow \alpha X \bullet \beta$
- Fill the parsing table
  - $A \rightarrow \alpha \bullet$ ➔ reduce $A \rightarrow \alpha$ on all terminals
  - $A \rightarrow \alpha \bullet t \beta$ ➔ shift the appropriate state on t
  - $A \rightarrow \alpha \bullet X \beta$ ➔ goto the appropriate state on X (happens after reduce on the remaining stack)

Constructing SLR(1) parsing table

- Add a production $S' \rightarrow S\$$
- Construct a deterministic finite automaton accepting "valid stack symbols"
  - States are set of items $A \rightarrow \alpha \bullet \beta$
    - The initial state includes $S' \rightarrow \bullet S$ and its epsilon closure
    - All the states are accepting (excluding the sink which not shown)
    - There is a transition labeled by X between states include $A \rightarrow \alpha \bullet X \beta$ and the epsilon closure of $A \rightarrow \alpha X \bullet \beta$
- Fill the parsing table
  - $A \rightarrow \alpha \bullet$ ➔ reduce $A \rightarrow \alpha$ on terminals in Follow(A)
  - $A \rightarrow \alpha \bullet t \beta$ ➔ shift the appropriate state on t
  - $A \rightarrow \alpha \bullet X \beta$ ➔ goto the appropriate state on X (happens after reduce on the remaining stack)

A Trivial Example

- S → A B $
- A → a
- B → a



LR(0) Control Table Trivial Example

state	terminal			nonterminal		
	a	$	other	S	A	B
0 S→ .A B $ A→ .a	shift 1	err	err		2	
1 A→ a.	reduce A→a					
2 S→ A .B $ B→ .a	shift 3	err	err			4
3 B→ a.	reduce B→a					
4 S→ AB. $	err	shift 5	err			
5 S→ AB $.	accept					

SLR(1) Control Table Trivial Example

state	terminal			nonterminal		
	a	$	other	S	A	B
0 S→ .A B $ A→ .a	shift 1	err	err		2	
1 A→ a.	reduce A→a	err	err			
2 S→ A .B $ B→ .a	shift 3	err	err			4
3 B→ a.	err	reduce B→a	err			
4 S→ AB. $	err	shift 5	err			
5 S→ AB $.	accept					

Parsing aa

state	terminal			nonterminal		
	a	$	other	S	A	B
0	s1	err	err		2	
1	reduce A→a					
2	s3	err	err			4
3	reduce B→a					
4	err	s5	err			
5	accept					

stack	**input**	**action**
0($)	**aa $**	**shift 1**
1(a) **0($)**	**a $**	**reduce** **A→a**
0($)	**a $**	**A**
2(A) **0($)**	**a $**	**shift 3**

Parsing aa (Cont)

state	terminal			Nonterminal		
	a	$	other	S	A	B
0	s1	err	err		2	
1	reduce A→a					
2	s3	err	err			4
3	reduce B→a					
4	err	s5	err			
5	accept					

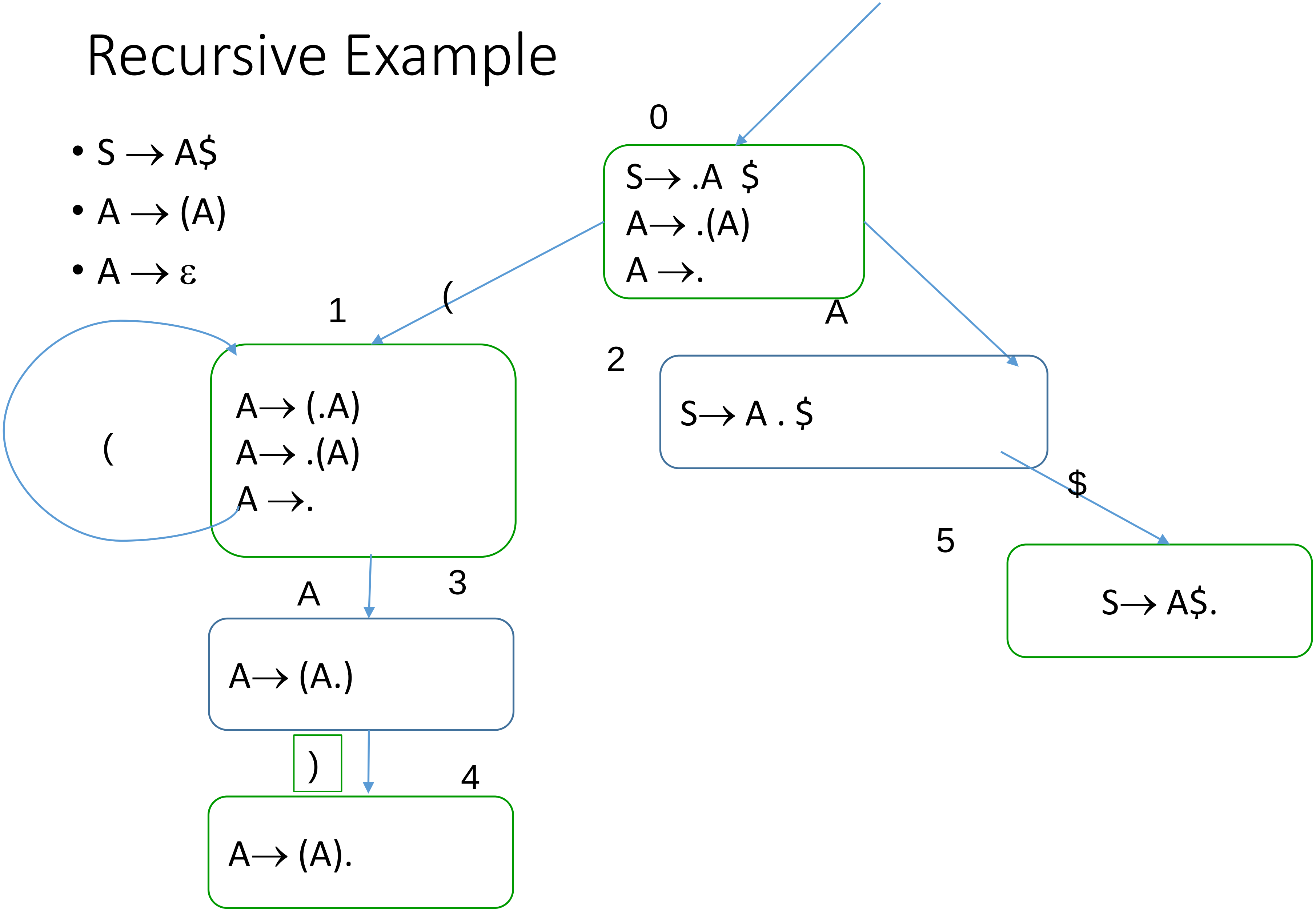
stack	input	action
3(a) 2(A) 0($)	$	reduce B→a
4(B) 2(A) 0($)	$	shift 5
5($) 4(B) 2(A) 0($)		accept

The Rightmost Derivation in Reverse Order

1. S → BA$
2. B→a
3. A→a

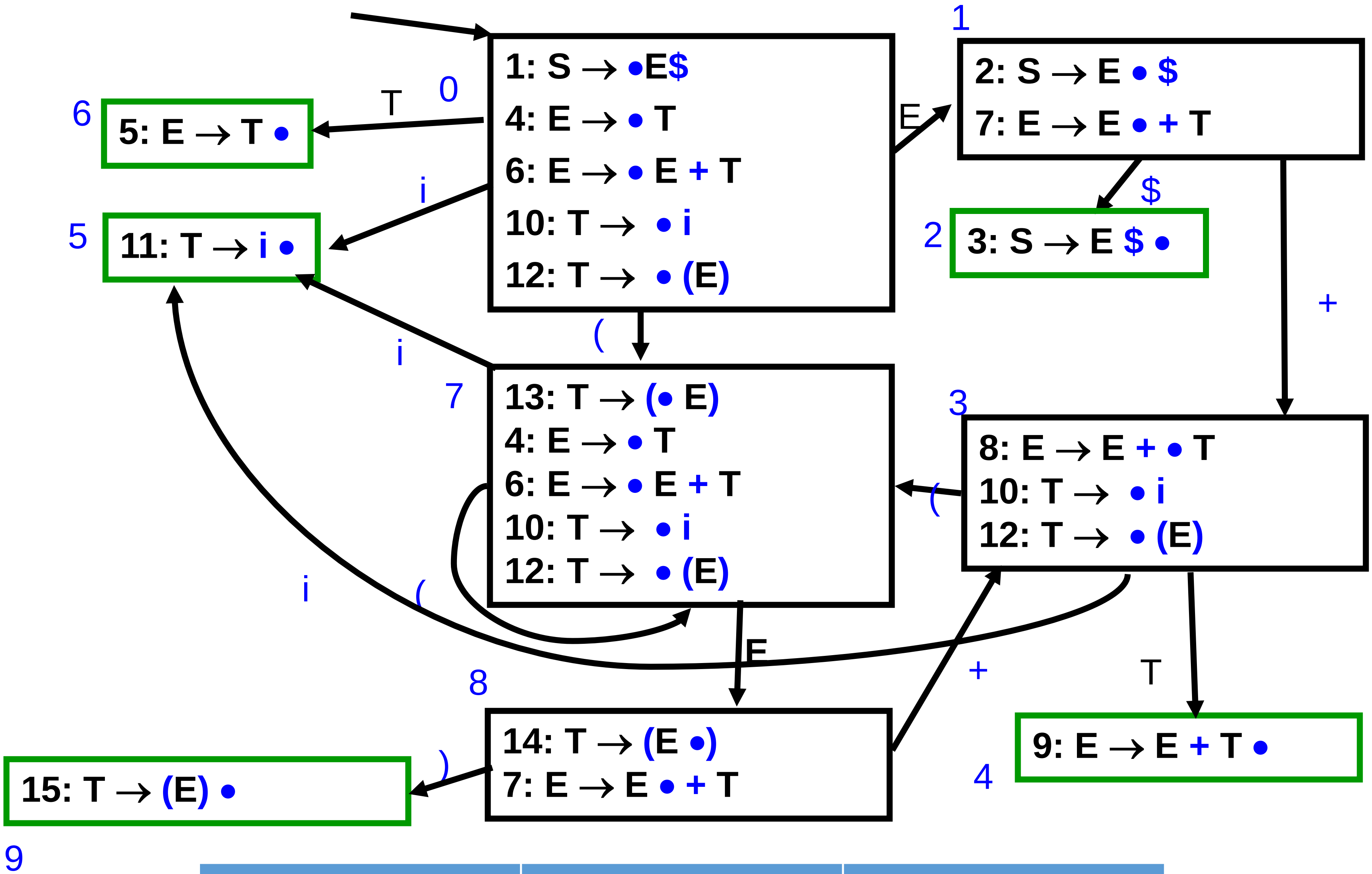
Recursive Example

- $S \rightarrow A\$$
- $A \rightarrow (A)$
- $A \rightarrow \varepsilon$



Control Table Recursive Example

state	terminal				nonterminal	
	(	)	$	other	S	A
0 S→ .A$ A→ .(A) A→.	shift 1	err	err	err		2
1 A→ (.A) A→ .(A) A →.	shift 1	reduce A→				
2 S→ A .$	shift 5	err				
3 A→ (A.)	err	shift 4				
4 A→ (A).	err	err	shift 5			
5 S→ A$.	accept					


0
1: S → •E$
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
1
2: S → E • $
7: E → E • + T
2
3: S → E $ •
3
8: E → E + • T
10: T → • i
12: T → • (E)
4
9: E → E + T •
5
11: T → i •
6
5: E → T •
7
13: T → (• E)
4: E → • T
6: E → • E + T
10: T → • i
12: T → • (E)
8
14: T → (E •)
7: E → E • + T
9
15: T → (E) •
T
E
$
+
i
(
)

	S	E	T
Follow		+,)	+,)

Example Control Table

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce $E \rightarrow E+T$						
5	reduce $T \rightarrow i$						
6	reduce $E \rightarrow T$						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce $T \rightarrow (E)$						

Example Control Table

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

0($)

input

i + i $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)
0 ($)

input

+ i $

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
0 ($)

input

+ i $

reduce E → T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

1(E)

0 ($)

input

+ i $

shift 3

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

3 (+)
1(E)
0 ($)

input

i $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)
3 (+)
1(E)
0($)

input

$

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

4 (T)
3 (+)
1(E)
0($)

input

$

reduce E → E + T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack	input	
1 (E) 0 ($)	$	**shift 2**

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

2 ($)
1 (E)
0 ($)

input

accept

The Rightmost Derivation in Reverse Order

1. S → E$
2. E → E + T
3. T→ I
4. E → T
5. T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack	input
0($)	**((i) $**

shift 7

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

7(()

0($)

input

(i) $

shift 7

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce $E \to E+T$						
5	reduce $T \to i$						
6	reduce $E \to T$						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce $T \to (E)$						

stack

7 (()
7(()
0($)

input

i) $

shift 5

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

5 (i)
7 (()
7(()
0($)

input

) $

reduce T → i

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
7 (()
7(()
0($)

input

) $

reduce E →T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

8 (E)
7 (()
7(()
0($)

input

) $

shift 9

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

9 ())
8 (E)
7 (()
7(()
0($)

input

$

reduce T → (E)

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

6 (T)
7(()
0($)

input

$

reduce E → T

	i	+	(	)	$	E	T
0	s5	err	s7	err	err	1	6
1	err	s3	err	err	s2		
2	acc						
3	s5	err	s7	err	err		4
4	reduce E→E+T						
5	reduce T → i						
6	reduce E → T						
7	s5	err	s7	err	err	8	6
8	err	s3	err	s9	err		
9	reduce T→(E)						

stack

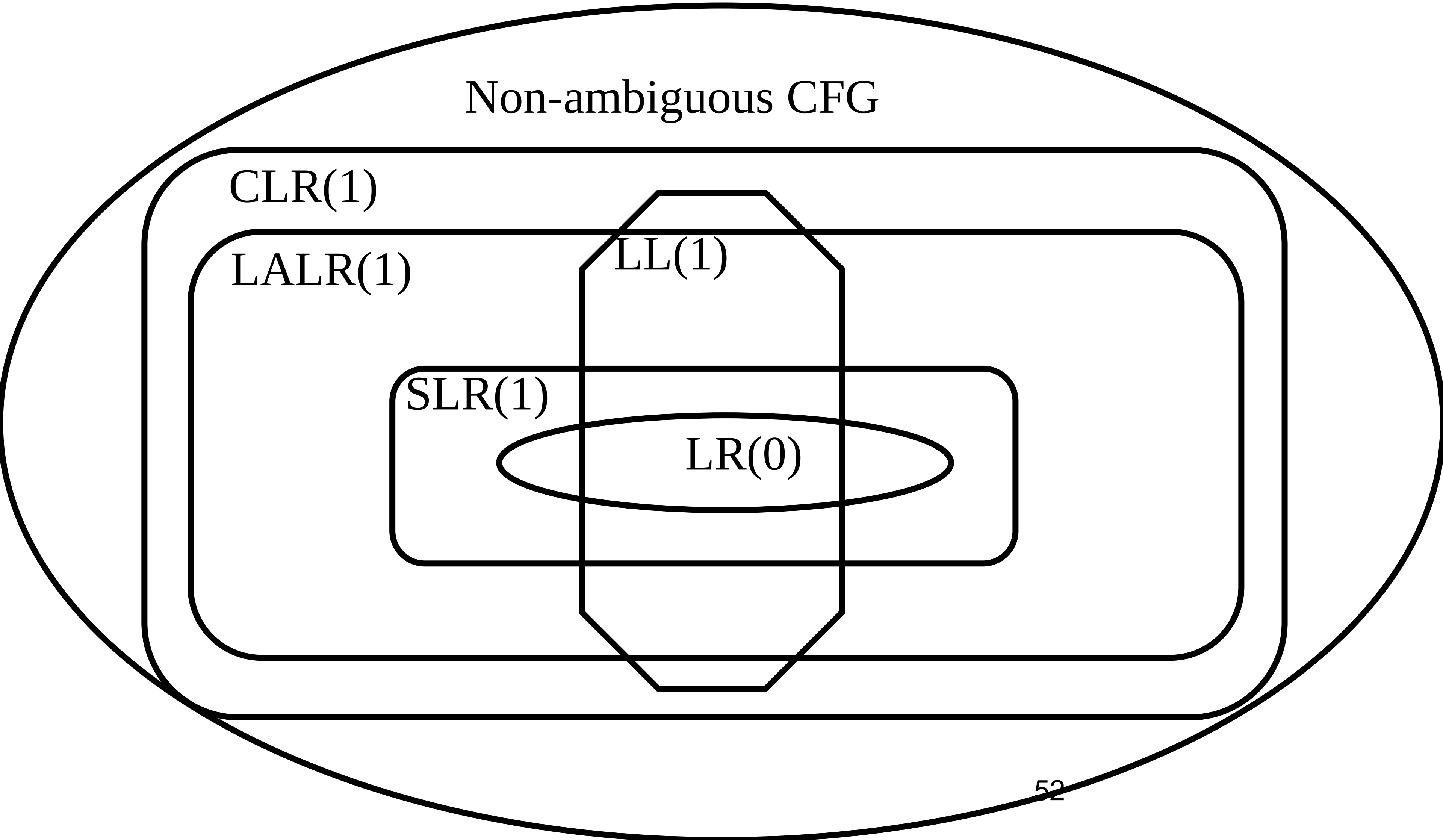
8 (E)
7(()
0($)

input

$

err

Grammar Hierarchy



Interesting Non LR(1) Grammars

- Ambiguous
  - Arithmetic expressions
  - Dangling-else
- Common derived prefix
  - $A \rightarrow B_1$ a b | B_2 a c
  - $B_1 \rightarrow \varepsilon$
  - $B_2 \rightarrow \varepsilon$
- Optional non-terminals
  - St $\rightarrow$ OptLab Ass
  - OptLab $\rightarrow$ id : | ε
  - Ass $\rightarrow$ id := Exp

St $\rightarrow$ id: Ass | Ass

Interesting Non LR(1) Grammars

- Ambiguous
  - Arithmetic expressions
  - Dangling-else
- Common derived prefix
  - $A \rightarrow B_1$ a b | B_2 a c
  - $B_1 \rightarrow \varepsilon$
  - $B_2 \rightarrow \varepsilon$
- Optional non-terminals
  - St $\rightarrow$ OptLab Ass
  - OptLab $\rightarrow$ id : | ε
  - Ass $\rightarrow$ id := Exp

Summary

- LR is a powerful technique
- Generates efficient parsers
- Generation tools exit LALR(1)
  - Bison, yacc, CUP, ply
- But some grammars need to be tuned
  - Shift/Reduce conflicts
  - Reduce/Reduce conflicts
  - Efficiency of the generated parser
- There exist more general methods
  - GLR
  - Arbitrary grammars in n^3
    - Early parsers
    - CYK algorithms

Code Generation

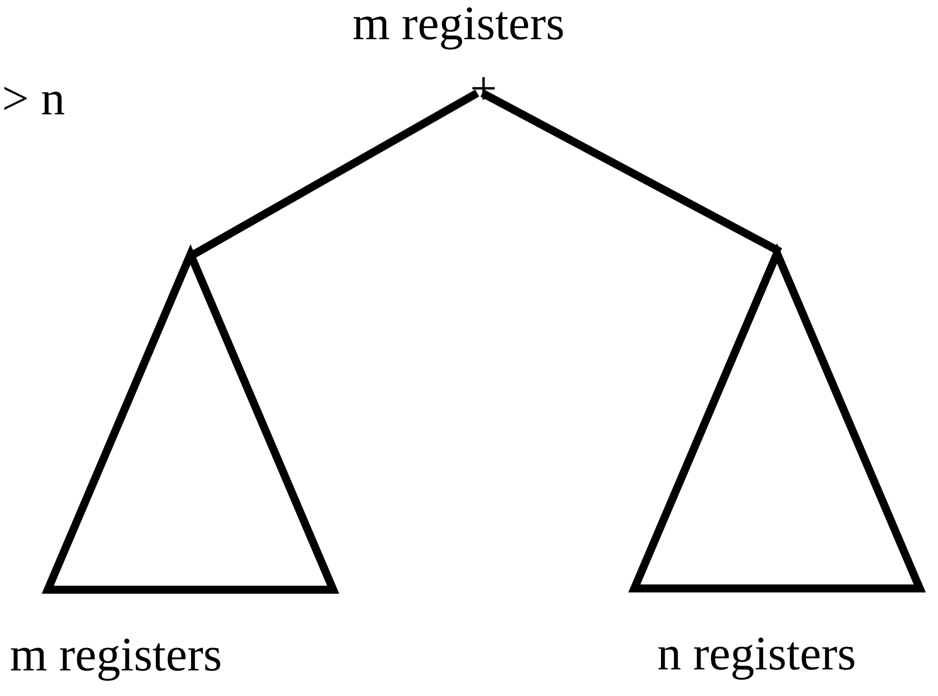
- Instruction Selection
- Register allocation
  - Caller save vs. Calle Save registers
  - Local vs. Global Register Allocation

Two Phase Solution
Dynamic Programming
Sethi & Ullman

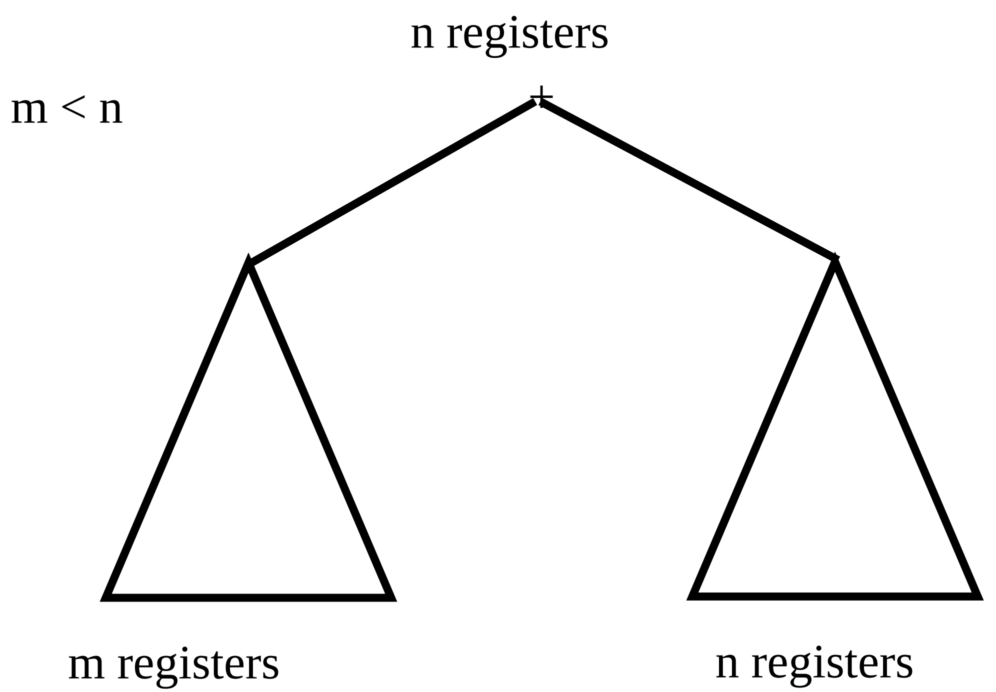
- Bottom-up (labeling)
  - Compute for every subtree
    - The minimal number of registers needed
    - Weight
- Top-Down
  - Generate the code using labeling by preferring “heavier” subtrees (larger labeling)
  - Can integrate spilling

The Labeling Principle

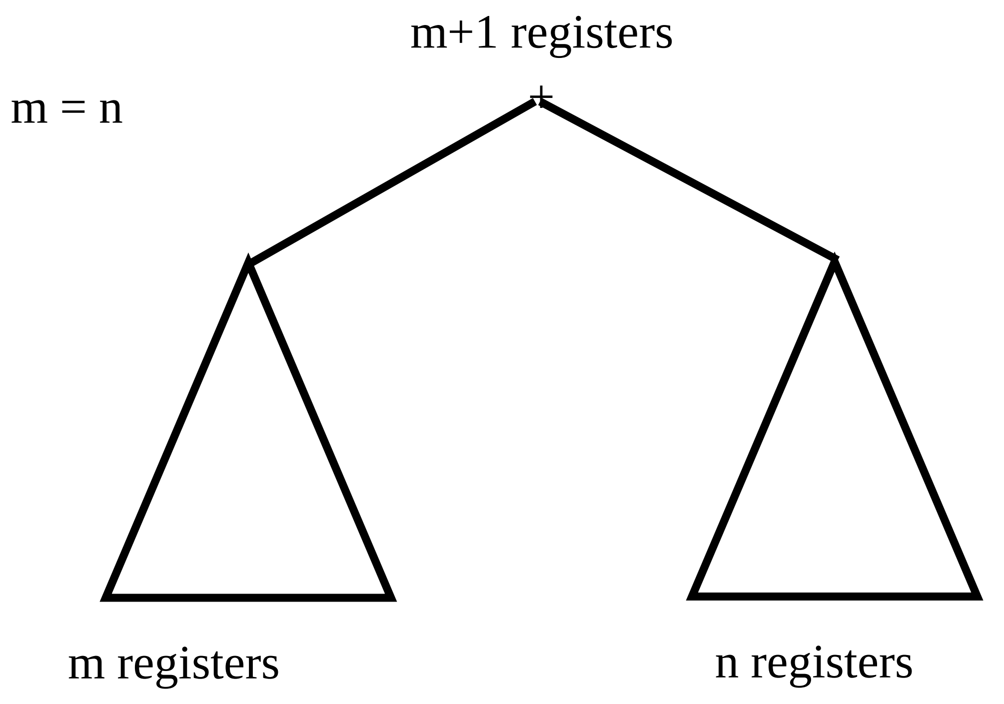
m > n



The Labeling Principle



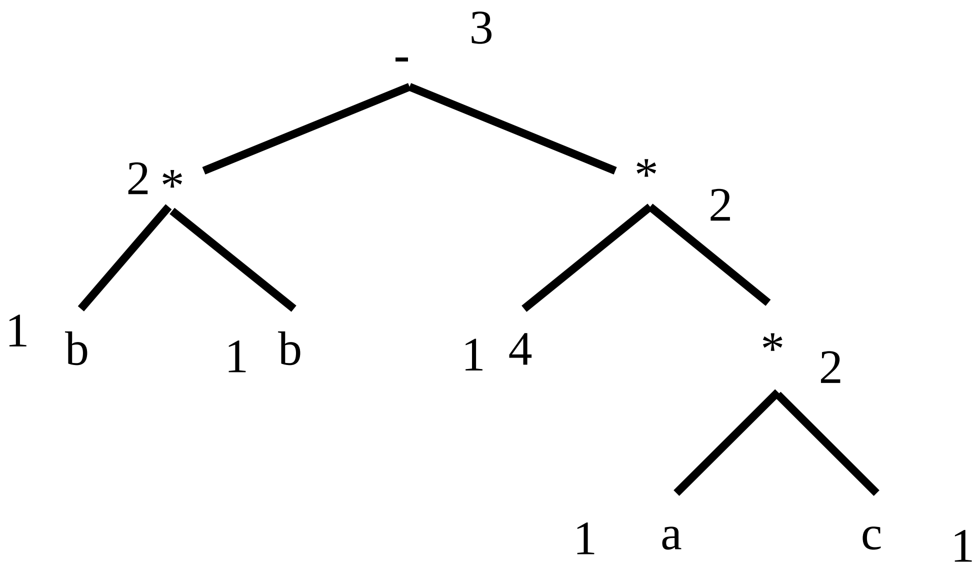
The Labeling Principle



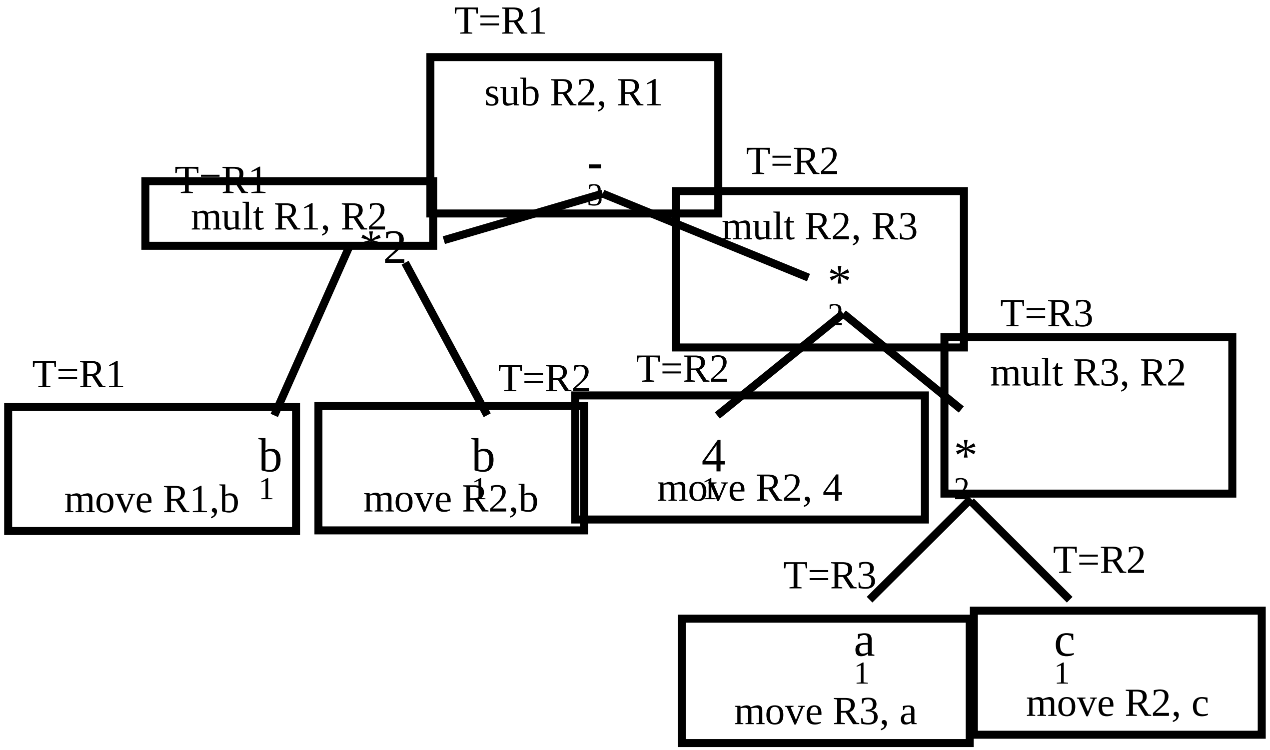
The Labeling Algorithm

```
weight(Node: expression): integer {
switch node: {
   case number(n: integer): return 1;
   case localVariable(v: symbol) return 1;
   case e1: Node + e2: Node {
       let lw: integer = weight(e1);
       let rw: integer = weight(e2);
       if (lw < rw) return rw ;
         else if (lw > rw) return lw;
         else return lw + 1 ;
}
…
}
```

Labeling the example (weight)



Top-Down


T=R1
sub R2, R1
-
3
T=R1
mult R1, R2
*2
T=R2
mult R2, R3
*
2
T=R3
mult R3, R2
*
2
T=R1
b
1
move R1,b
T=R2
b
1
move R2,b
T=R2
4
1
move R2, 4
T=R3
a
1
move R3, a
T=R2
c
1
move R2, c

The need for global register allocation

```
int foo() {
   int x = 1 ;
   x = x + 1;
   x = x + 1;
  …
  printf(“%d”, x);
}
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 1
        add     DWORD PTR [rbp-4], 1
        add     DWORD PTR [rbp-4], 1
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

```
foo():
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, 1
        add     eax, 1
        add     eax, 1
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

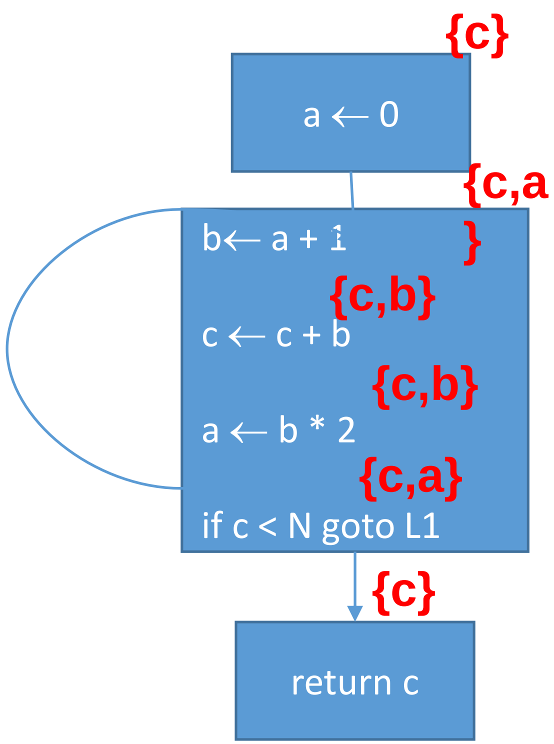
Caller-Save and Callee-Save Registers

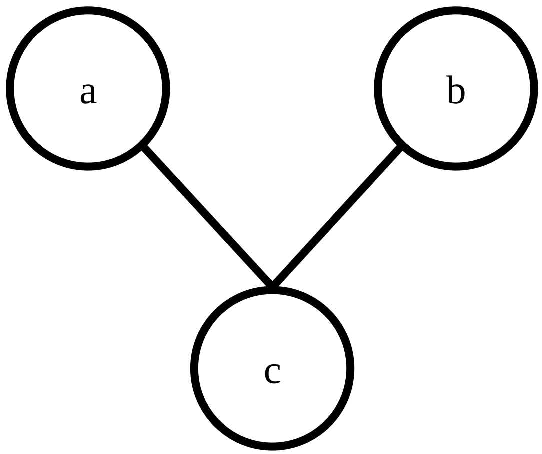
- callee-save-registers (MIPS 16-23, X86 r12-15, rbp, rsp)
  - Saved by the callee when modified
  - Values are automatically preserved across calls
- caller-save-registers
  - Saved by the caller when needed
  - Values are not automatically preserved
- Usually the architecture defines caller-save and callee-save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/languages
- But compilers can decide when to use calller/callee registers

X86lite Registers: 16 64-bit registers

register	usage	Callee save
rax	general purpose accumulator	N
rbx	base register, pointer to data	N
rcx	counter register for strings & loops	N
rdx	data register for I/O	N
rsi	pointer register, string source register	N
rdi	pointer register, string destination register	N
rbp	base pointer, points to the stack frame	Y
rsp	stack pointer, points to the top of the stack	Y
r08-r11	General purpose registers	N
r12-15	General purpose registers	Y

Using Liveness Information

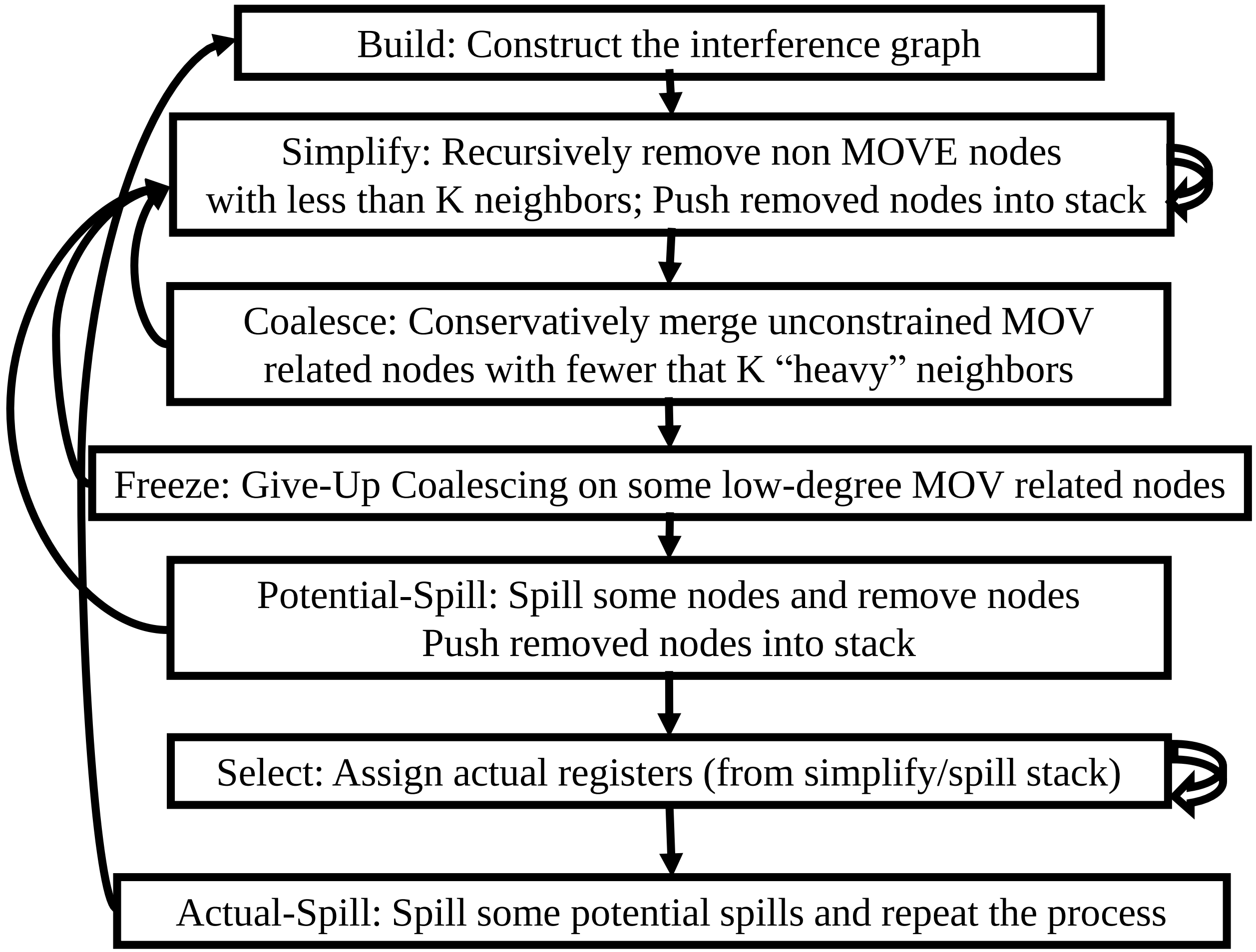




A Complete Example (Andrew Appel) https://www.cs.princeton.edu/~appel/

```
enter:
        c := r3          r1, r2   caller save
        a := r1          r3        callee-save
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```

Graph Coloring with Coalescing



A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

e := e-1

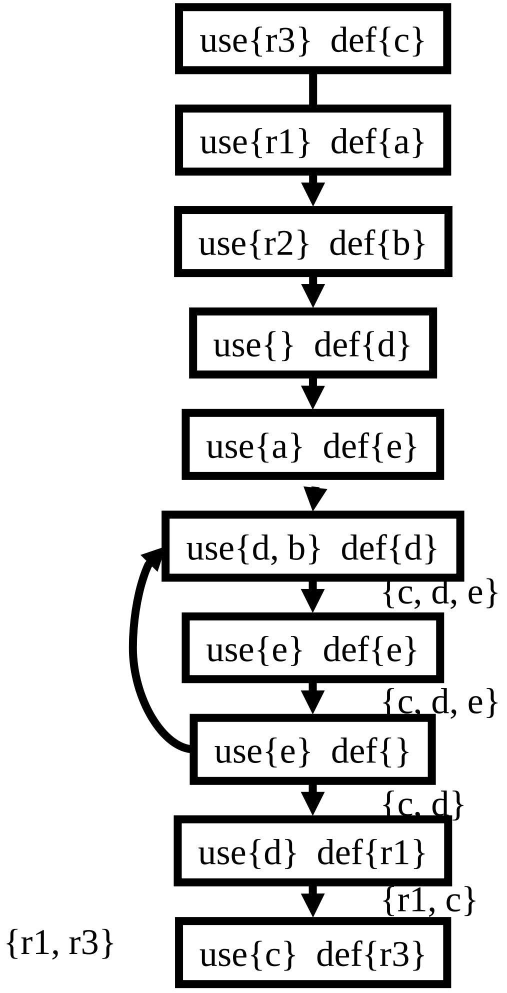
if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */

r1, r2 caller save
r3 callee-save



A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

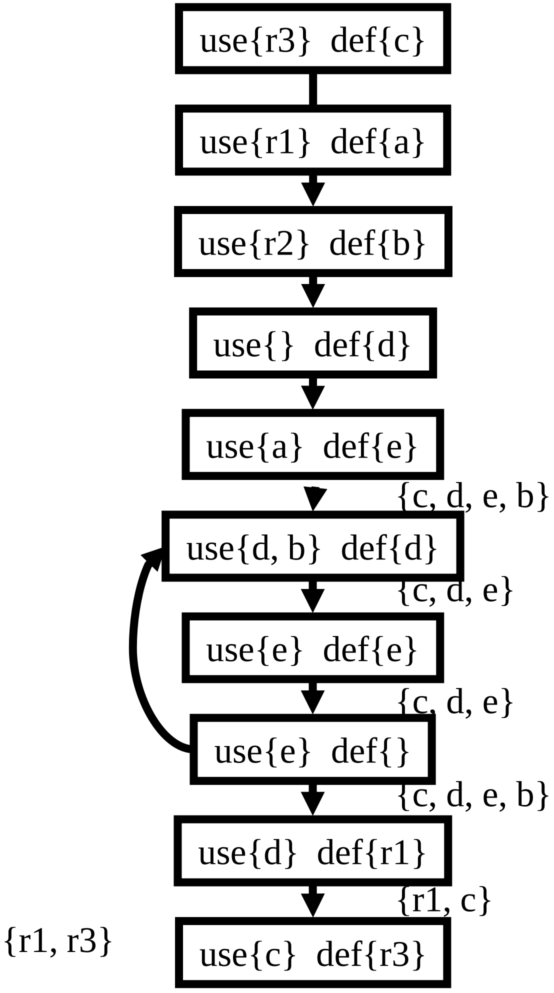
e := e-1

if e>0 goto loop

r1 := d

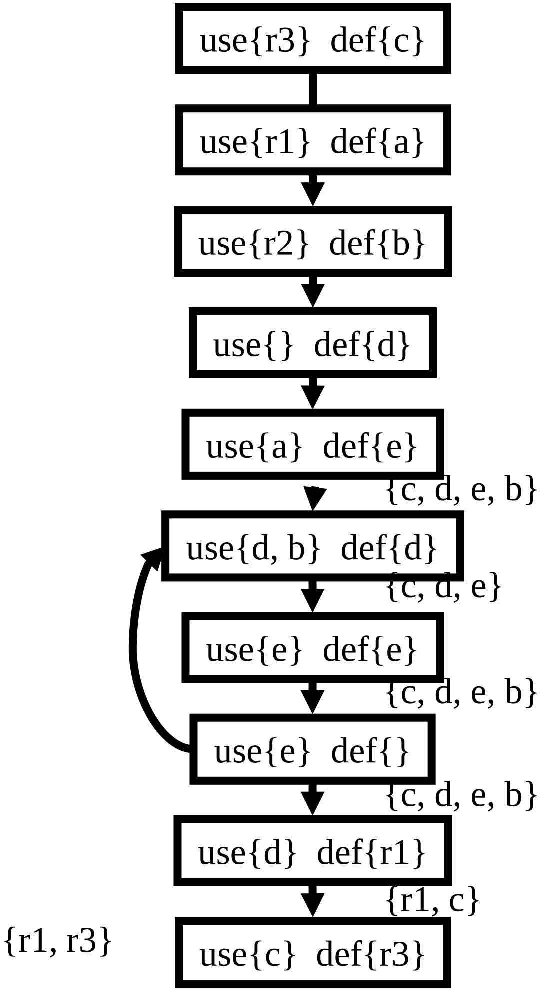
r3 := c

return /* r1,r3 */



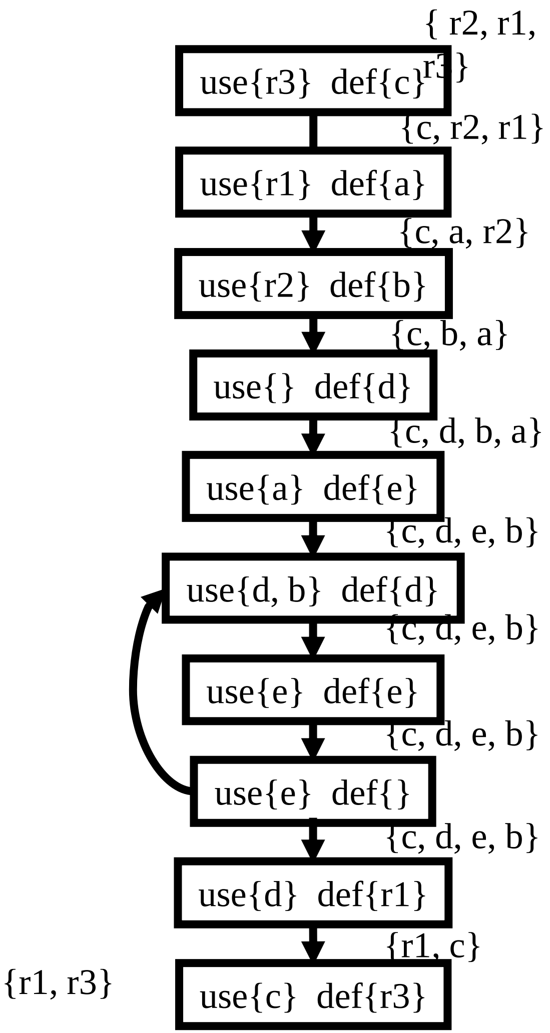
A Complete Example

```
enter:
        c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```



A Complete Example

```
enter:
        c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```

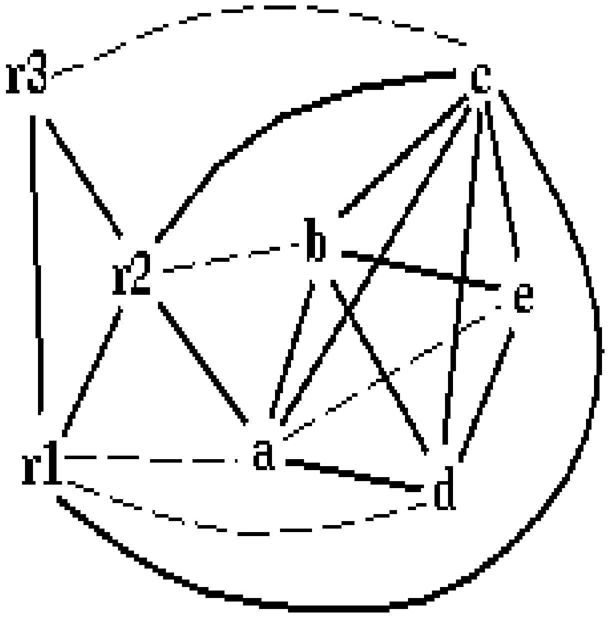


Live Variables Results

```
enter:
        c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```

```
enter:            /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   /* e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1, r3 */
```

```
enter              /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   /* e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

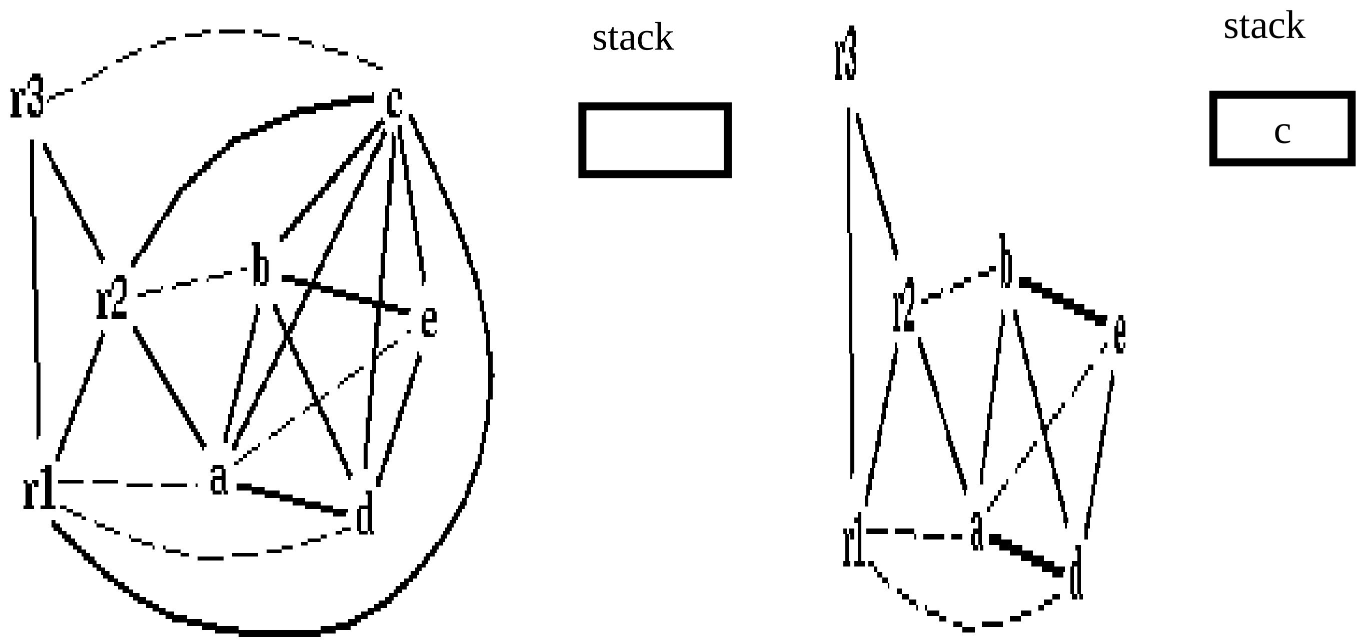


spill priority = (uo + 10 ui)/deg

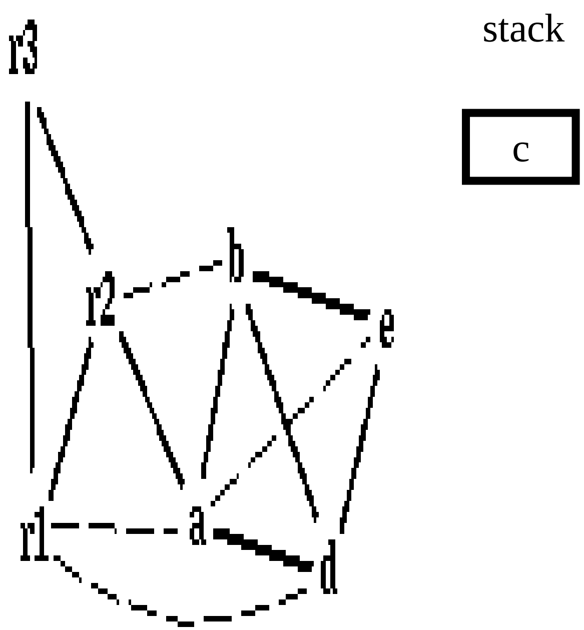
```
enter:                /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   / * e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

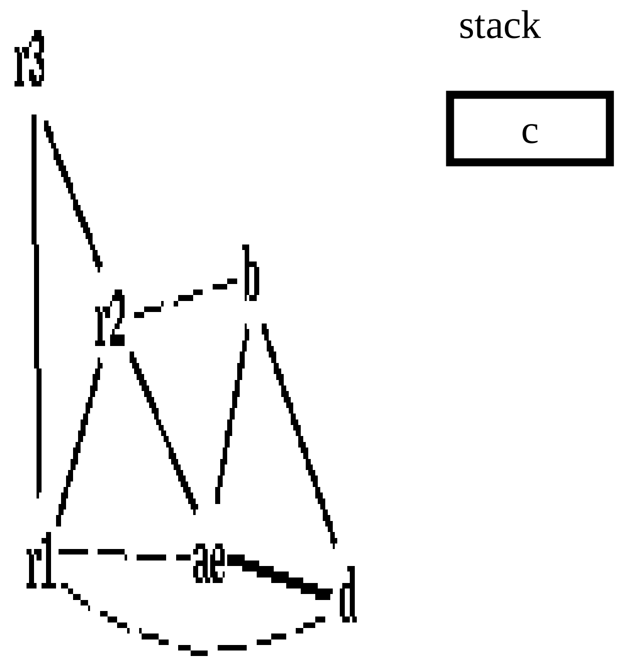
	use+ def outside loop	use+ def within loop	deg	spill priority
a	2	0	4	0.5
b	1	1	4	2.75
c	2	0	6	0.33
d	2	2	4	5.5
e	1	3	3	10.3

Spill C

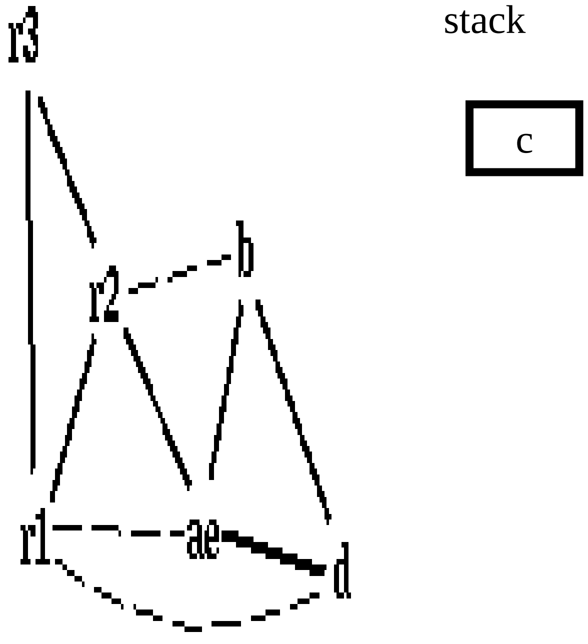


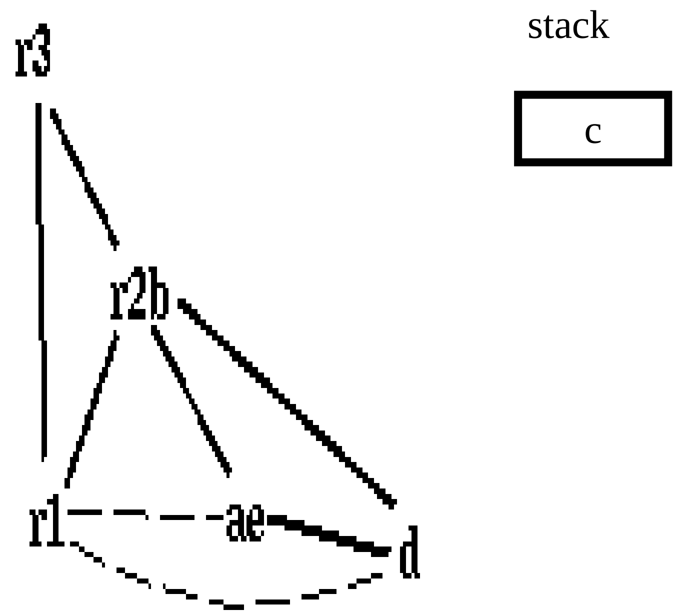
Coalescing *a*+*e*



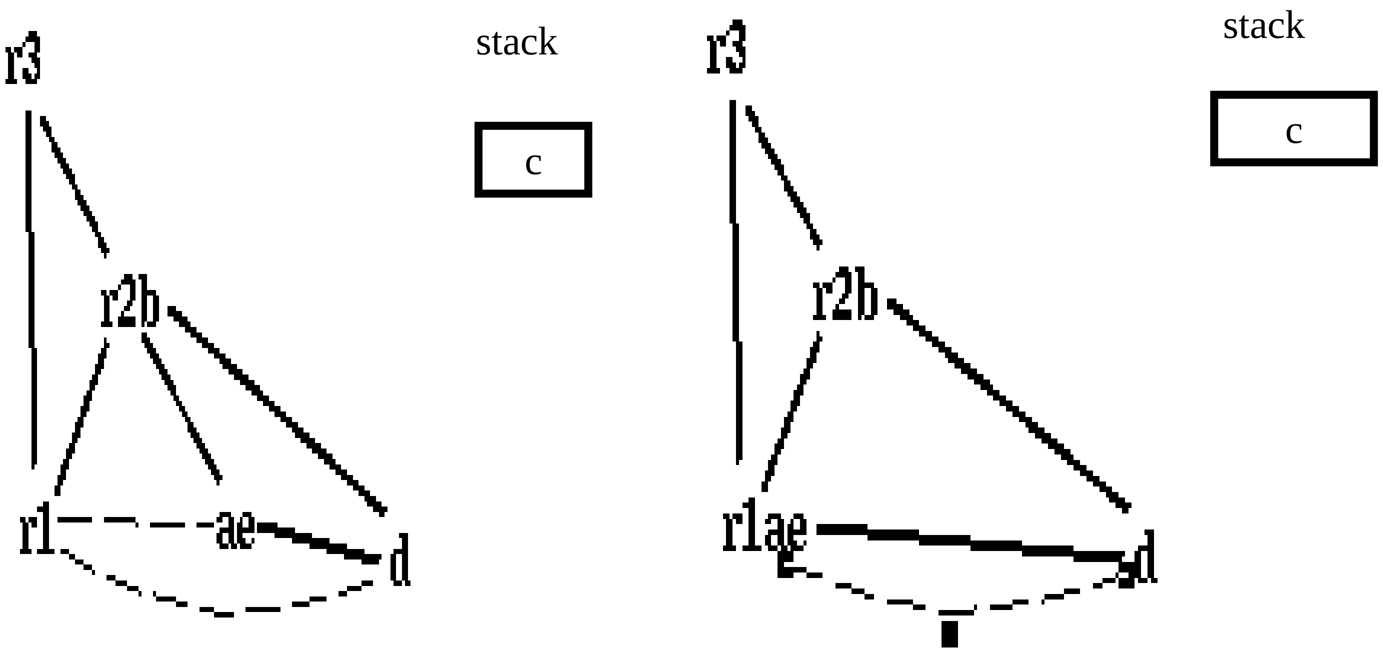


Coalescing *b+r2*



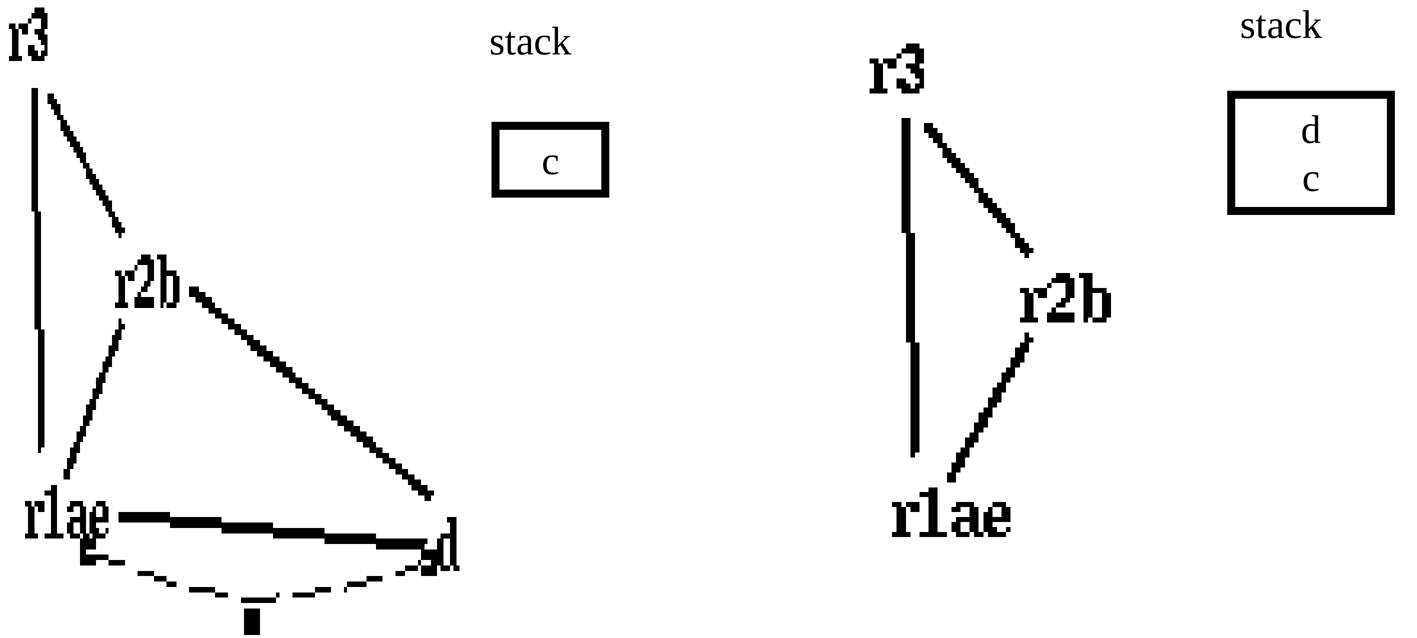


Coalescing *ae+r1*

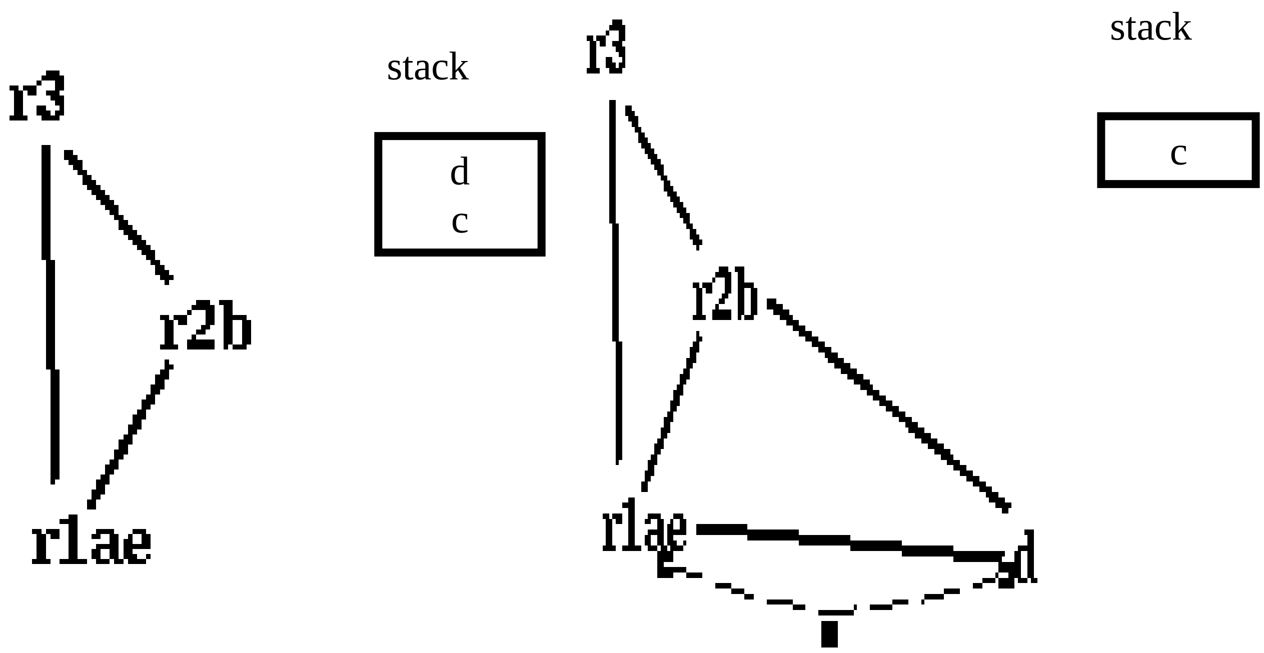


r1ae and d are constrained

Simplifying *d*

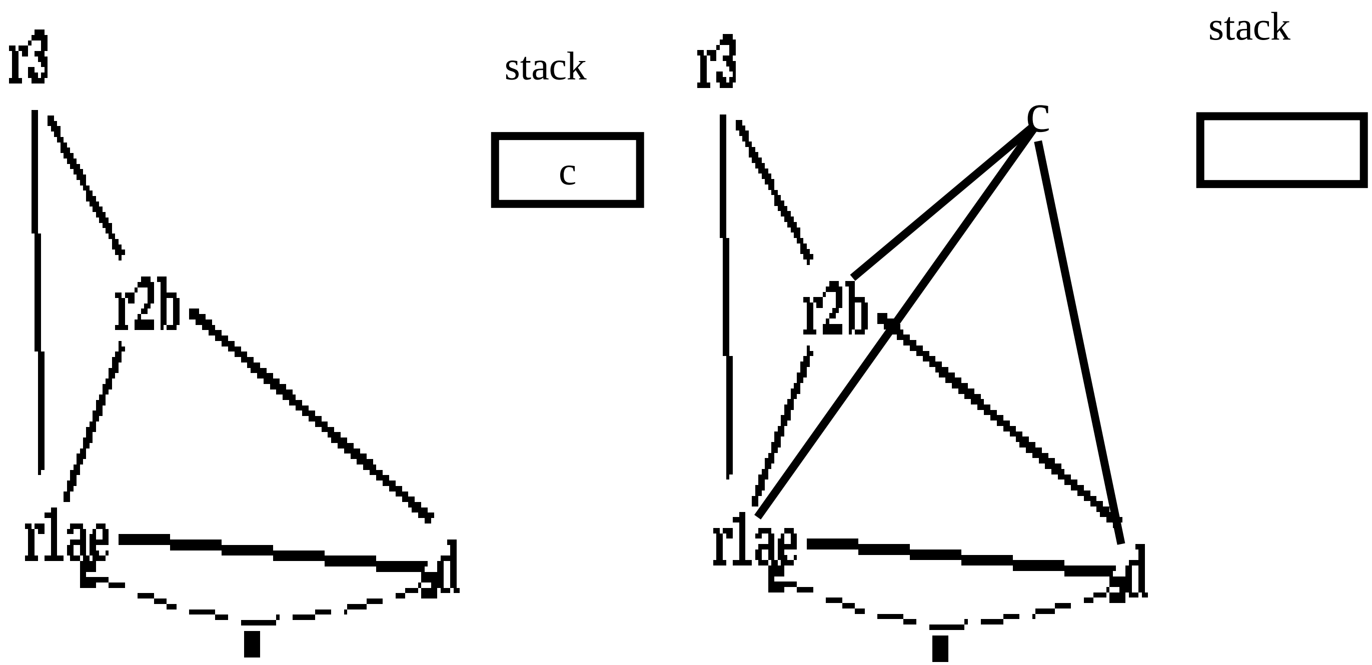


Pop *d*



d is assigned to r3

Pop *c*

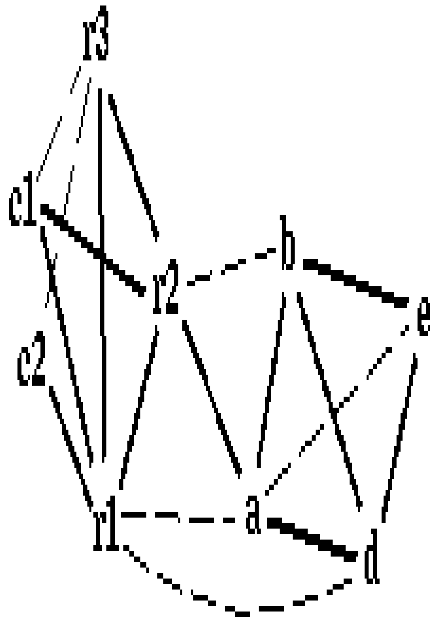


actual spill!

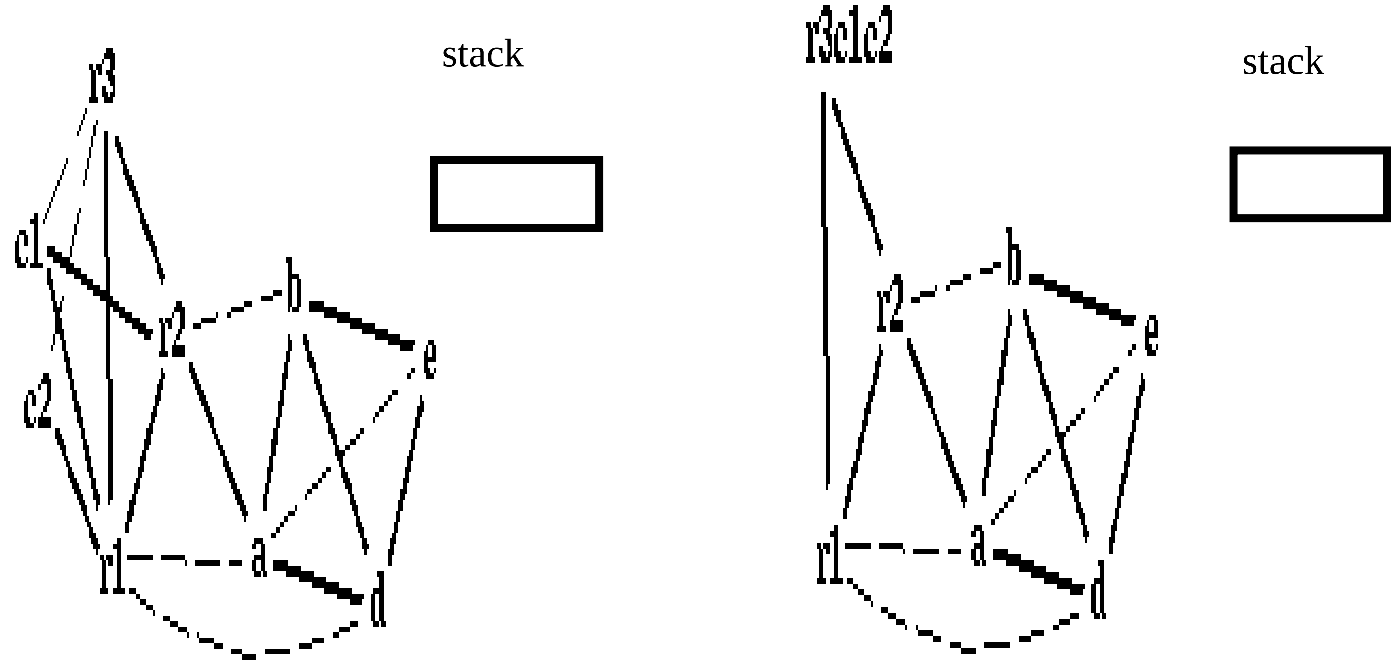
```
enter:              /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   / * e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

```
enter:              /* r2, r1, r3 */
        c1 := r3  /* c1, r2, r1 */
        M[c_loc] := c1 /* r2 */
        a := r1  /* a, r2 */
        b := r2  /* a, b */
        d := 0   /* a, b, d */
        e := a   / * e, b, d */
loop:
        d := d+b  /* e, b, d */
        e := e-1 /* e, b, d */
        if e>0 goto loop /* d */
        r1 := d  /* r1 */
        c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
  return  /* r1,r3 */
```

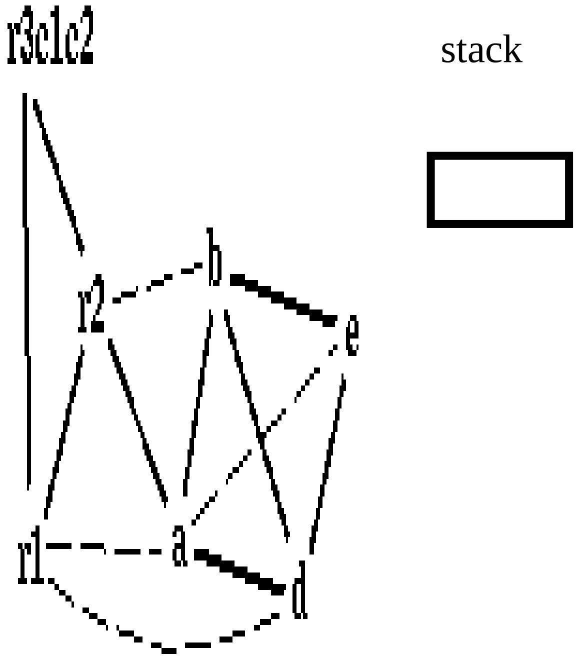
```
enter:              /* r2, r1, r3 */
        c1 := r3  /* c1, r2, r1 */
        M[c_loc] := c1 /* r2 */
        a := r1  /* a, r2 */
        b := r2  /* a, b */
        d := 0   /* a, b, d */
        e := a   / * e, b, d */
loop:
        d := d+b  /* e, b, d */
        e := e-1 /* e, b, d */
        if e>0 goto loop /* d */
        r1 := d  /* r1 */
        c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
  return  /* r1,r3 */
```

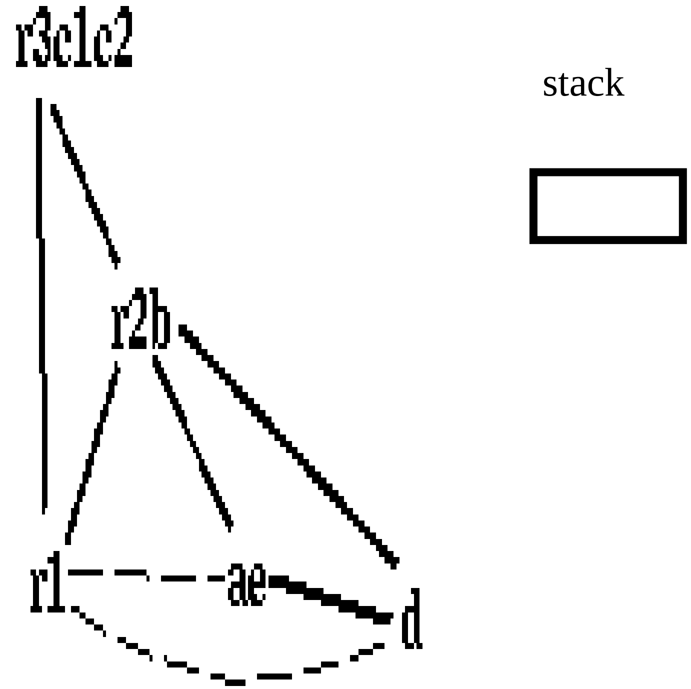


Coalescing c1+r3; c2+c1r3

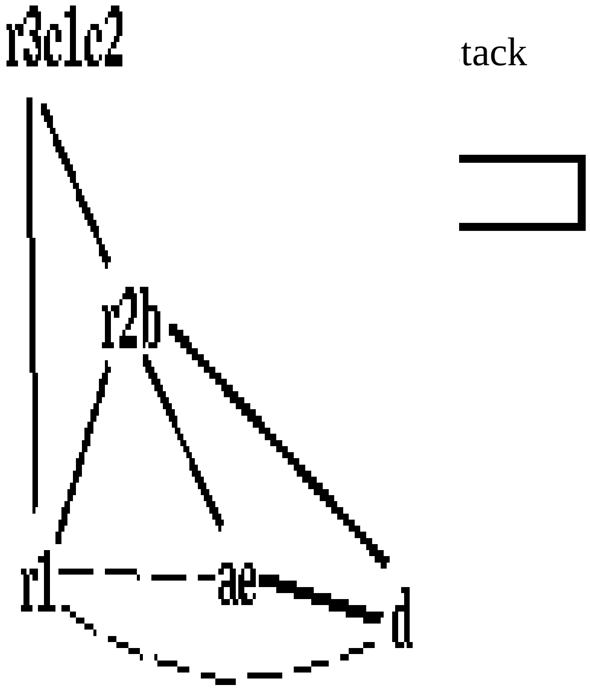


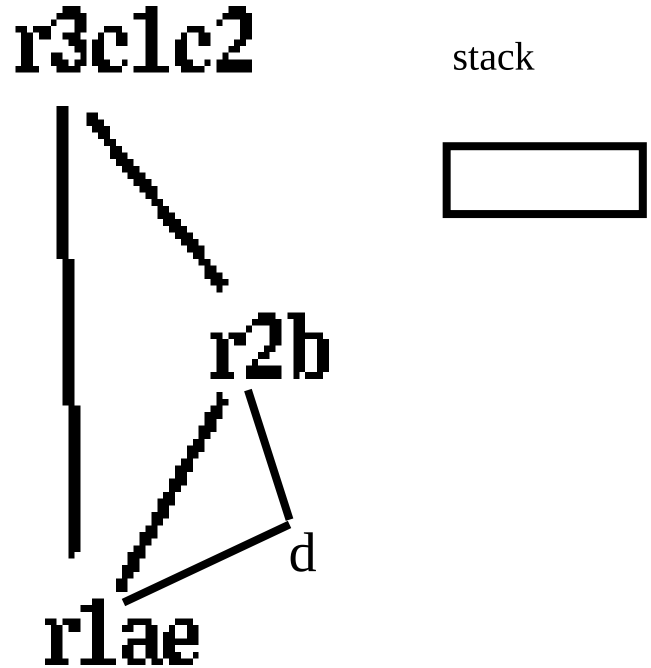
Coalescing a+e; b+r2





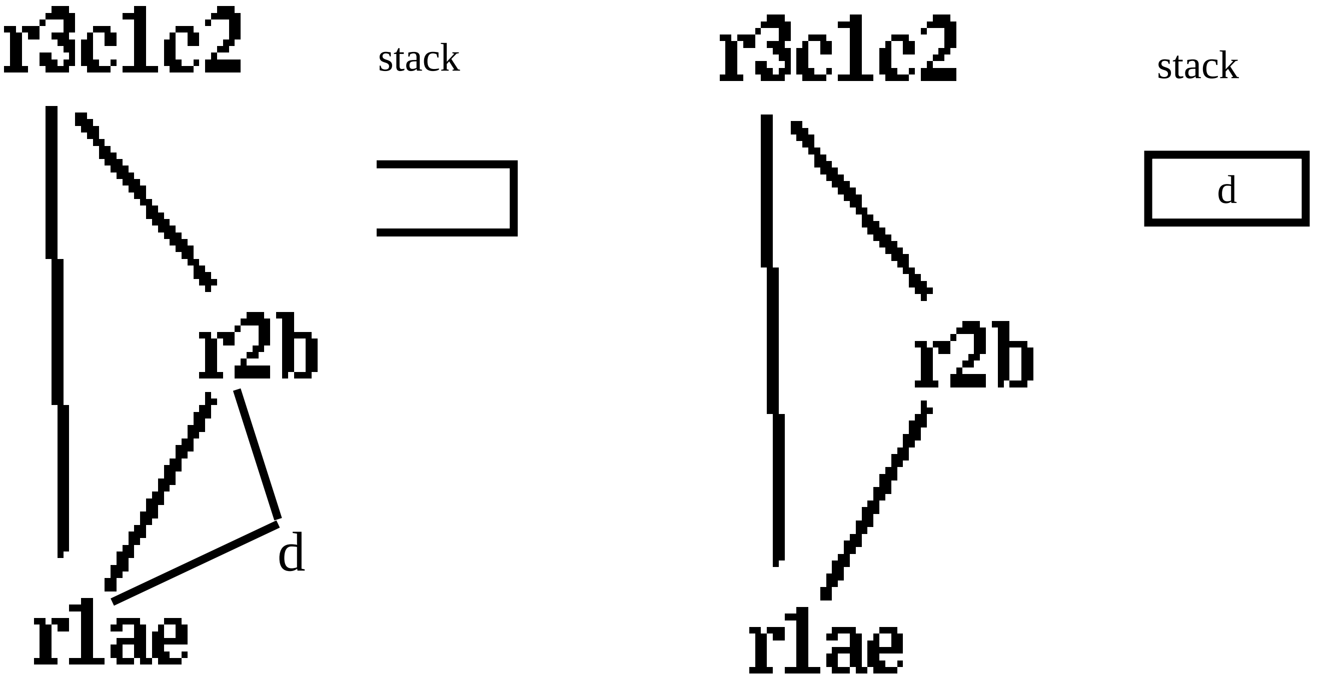
Coalescing ae+r1



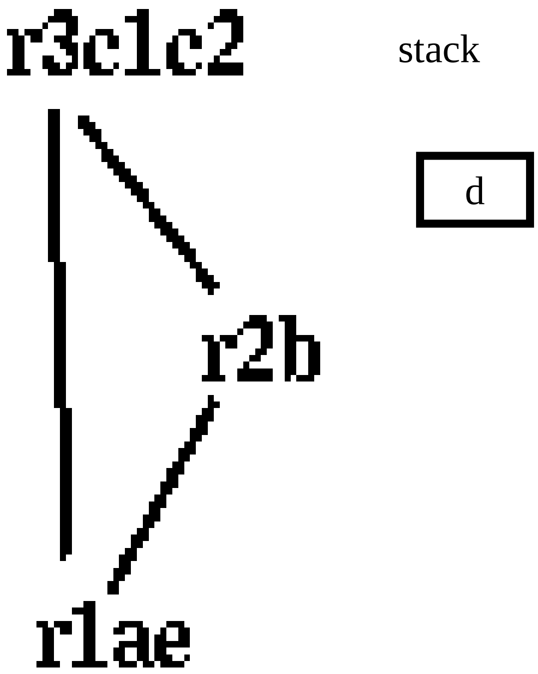


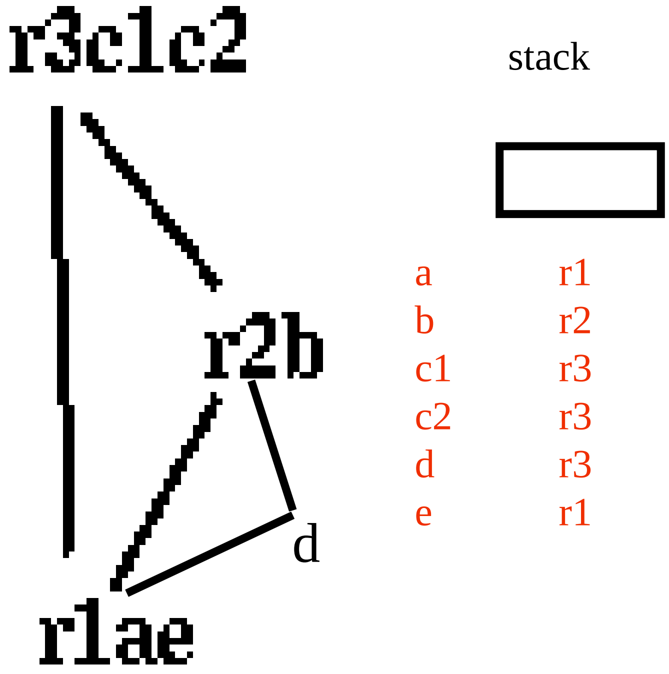
r1ae and d are constrained

Simplify d



Pop d


r3c1c2
stack
d
r2b
r1ae


r3c1c2
stack
r2b
d
r1ae
a r1
b r2
c1 r3
c2 r3
d r3
e r1

```
enter:
        c1 := r3
        M[c_loc] := c1
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        c2 := M[c_loc]
        r3 := c2
  return  /* r1,r3 */
```

a	r1
b	r2
c1	r3
c2	r3
d	r3
e	r1

```
enter:
        r3 := r3
        M[c_loc] := r3
        r1 := r1
        r2 := r2
        r3 := 0
        r1 := r1
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto
loop
        r1 := r3
        r3 := M[c_loc]
        r3 := r3
  return  /* r1,r3 */
```

```
enter:
        r3 := r3
        M[c_loc] := r3
        r1 := r1
        r2 := r2
        r3 := 0
        r1 := r1
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto
loop
        r1 := r3
        r3 := M[c_loc]
        r3 := r3
  return  /* r1,r3 */
```

```
enter:
        M[c_loc] := r3
        r3 := 0
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto
loop
        r1 := r3
        r3 := M[c_loc]
  return  /* r1,r3 */
```

Garbage Collection Techniques

- Reference counting
- Mark and sweep
- Copying
- Generational
- Incremental
- [Parallel]

Assembler/Linker/Loader

- Assembler convert symbolic instructions into binary format
  - Resolve local labels and generate relocation tables
- Linker merge several files into a single executable
  - Relocate instructions
  - Resolve external calls
- Loader build a runtime state from executable

Course Summary

- Many useful techniques
- Some implementation details
- Become a better programmer