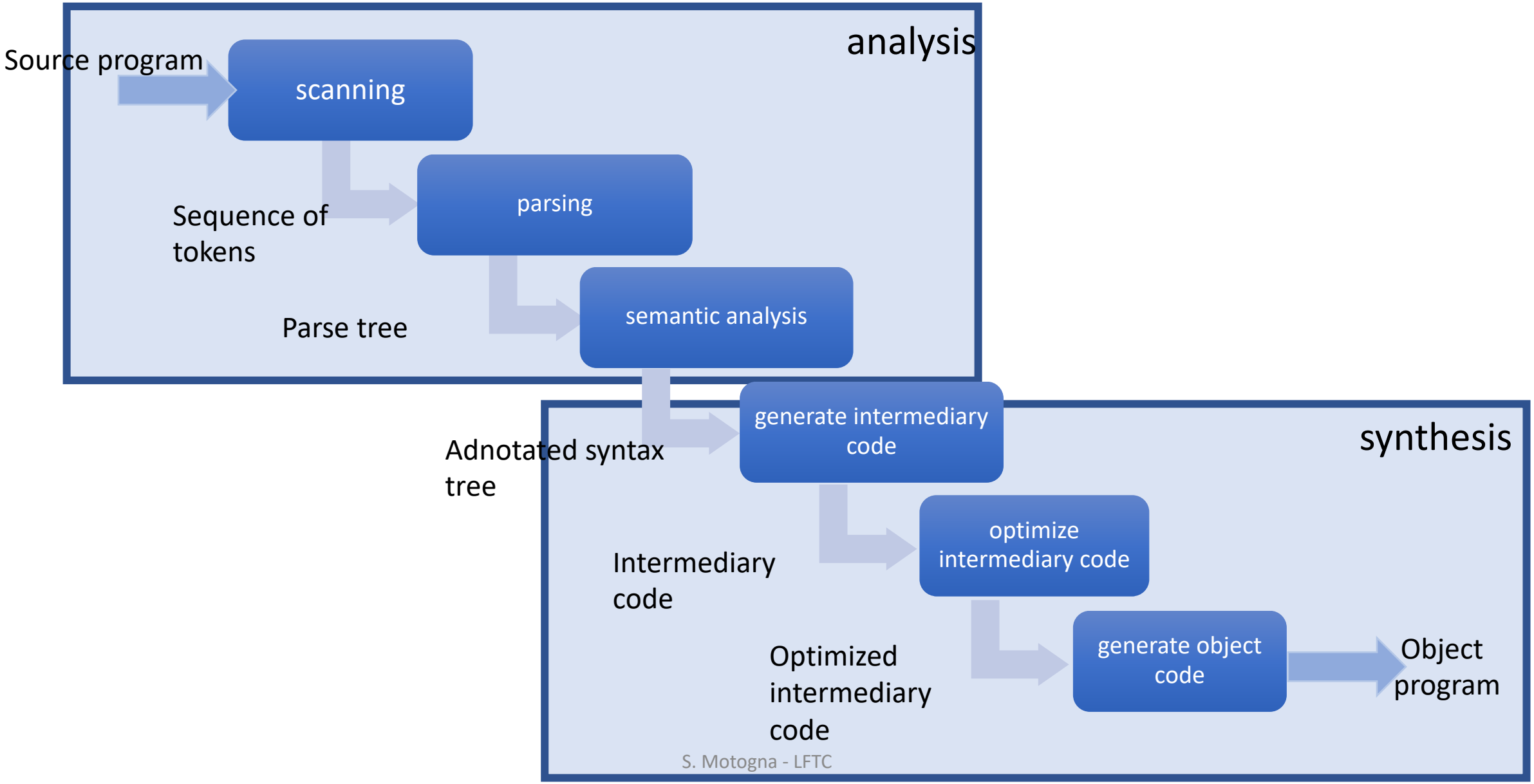
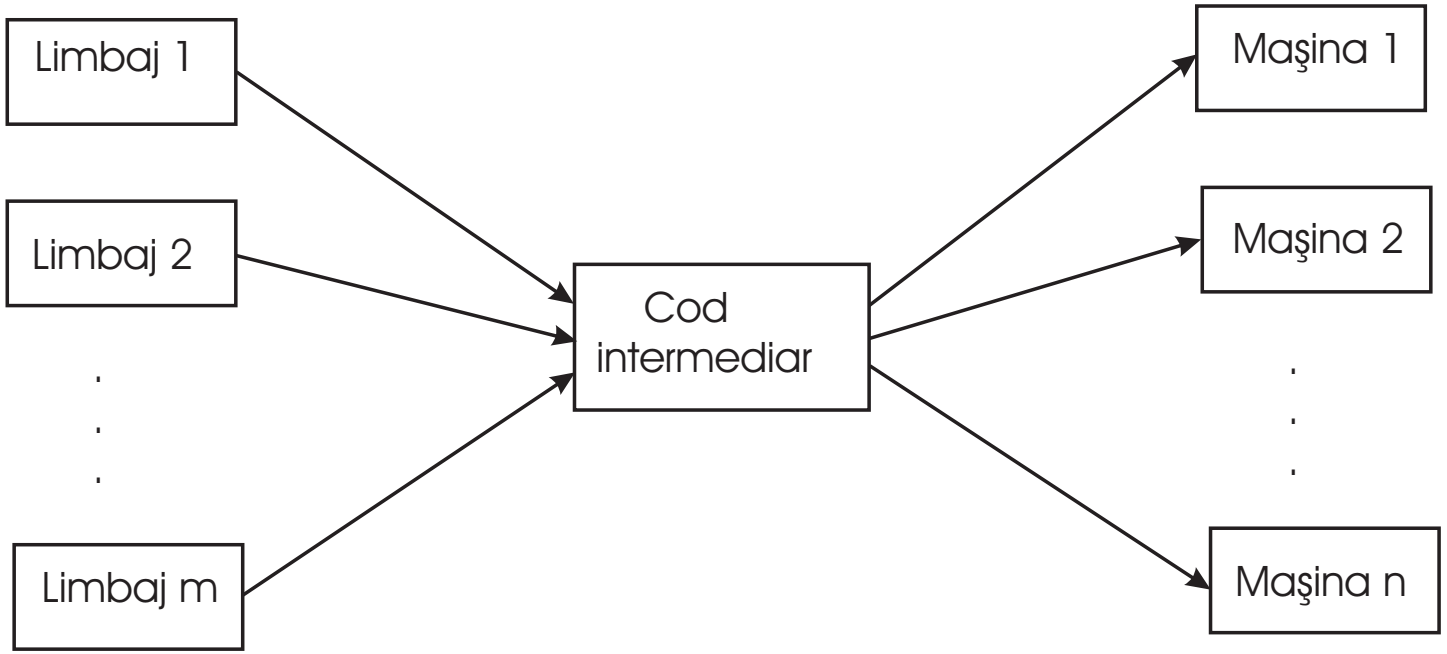
Course 13

Structure of compiler



Generate intermediary code



Generate object code

= translate intermediary code statements into statements of object code (machine language)

- Depend on “machine”: architecture and OS

Computer with accumulator

- A **stack machine** consists of:
- a stack for storing and manipulating values (store subexpressions and results)
- Accumulator – to execute operation
- 2 types of statements:
  - move and copy values in and from head of stack to accumulator
  - Operations on stack head, functioning as follows: operands are popped from stack, execute operation and then put the result in stack

Example: 4 * (5+1)

Code	acc	stack
acc ← 4	4	<>
push acc	4	<4>
acc ← 5	5	<4>
push acc	5	<5,4>
acc ← 1	1	<5,4>
acc ← acc + head	6	<5,4>
pop	6	<4>
acc ← acc * head	24	<4>
pop	24	<>

Computer with registers

- Registers +
- Memory

- Instructions:
  - LOAD v,R – load value **v** in register **R**
  - STORE R,v – put value **v** from register **R** in memory
  - ADD R1,R2 – add to the value from register **R1**, value from register **R2** and store the result in **R1** (initial value is lost!)

2 aspects:

- Register allocation – way in which variable are stored and manipulated;

- Instruction selection – way and order in which the intermediary code statements are mapped to machine instructions

Remarks:

1. A register can be available or occupied =>

   VAR(R) = set of variables whose values are stored in register R

2. For every variable, the place (register, stack or memory) in which the current value of the value exists=>

   MEM(x)= set of locations in which the value of variable x exists (will be stored in Symbol Table)

Example: F := A * B – (C + B) * (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B			
(2) T2 = C + B			
(3) T3 = T2 * T1			
(4) F:= T1 – T3			

Example: F := A * B – (C + B) * (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B	LOAD A, R0 MUL R0, B	VAR(R0) = {A} VAR(R0) = {T1}	MEM(T1) = {R0}
(2) T2 = C + B			
(3) T3 = T2 * T1			
(4) F:= T1 – T3			

Example: F := A * B – (C + B) * (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) T2 = C + B	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) T3 = T2 * T1			
(4) F:= T1 – T3			

Example: F := A * B – (C + B) * (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) T2 = C + B	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) T3 = T2 * T1	MUL R1,R0	VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) F:= T1 – T3			

Example: F := A * B – (C + B) * (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) T2 = C + B	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) T3 = T2 * T1	MUL R1,R0	VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) F:= T1 – T3	SUB R0,R1 STORE RO, F	VAR(R0) = {F} VAR(R1) = {}	MEM(T1) = {} MEM(F) = {R0, F}

More about Register Allocation

- Registers – **limited resource**
- Registers – perform operations / computations
- Variables **much more** than registers

IDEA: *assigning a large number of variables to a reduced number of registers*

Live variables

- Determine the number of variables that are live (used)

Example:

a = b + c

d = a + e

e = a + c

	op	op1	op2	rez
1	+	b	c	a
2	+	a	e	d
3	+	a	c	e

	1	2	3
a	x	x	x
b	x		
c	x	x	x
d		x	
e		x	x

Graph coloring allocation (Chaitin a.o. 1982)

- Graph:
  - nodes = live variables that should be allocated to registers
  - edges = live ranges simultaneously live

Register allocation = graph coloring: colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color

Disadvantage:

- NP complete problem

Linear scan allocation (Poletto a.o., 1999)

- determine all live range, represented as an interval
- intervals are traversed chronologically
- greedy algorithm

Advantage: speed – code is generated faster (speed in code generation)

Disadvantage: generated code is slower (NO speed in code execution)

Instruction selection

Example: F := A ∗ B – (C + B) ∗ (A * B)

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) T1 = A * B	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) T2 = C + B	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) T3 = T2 * T1	MUL R1,R0	VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) F:= T1 – T3			

STORE R0,T1

MUL R0,R1

LOAD T1,R1

Decide which register to use for an instruction

Syntax directed translation

Syntax oriented translation

- The actions are decided based on grammar rules

- Applied to generate intermediary code

Preliminaries

Functions

- *gen* – generate intermediary code
- *new_temp* – return a new name for a temporary variable

Attributes

- *E.loc* = location for value of E
- *E.code* = sequence of 3 address code to evaluate E

Example

Production
$S \rightarrow id := E$
$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow (E_1)$
$E \rightarrow id$
$E \rightarrow const$

Example

Production	Translation rule
S → id := E	S.code = E.code || gen(id.loc ‘:=‘ E.loc)
E → E_1 + E_2	E.loc = new_temp E.code = E_1.code ||E_2.code ||gen(E.loc ‘=‘ E_1.loc ‘+’ E_2.loc)
E → E_1 * E_2	E.loc = new_temp E.code = E_1.code ||E_2.code ||gen(E.loc ‘=‘ E_1.loc ‘*’ E_2.loc)
E → (E_1)	E.loc = E_1.loc E.code = E_1.code
E → id	E.loc = id.loc E.code = ‘’
E → const	E.loc = const.loc E.code = ‘’

i := a + b * c

Prod	**Location**	**Code**
S → id := E		E.code i := E.loc
E → E_1 + E_2	E.loc = T1	E_1.code E_2.code T1 = E_1.loc + E_2.loc i := T1
E → id	E_1.loc = id.loc	E_2.code T1 = a + E_2.loc i := T1

i := a + b * c

Prod	**Location**	**Code**
$E_2 \rightarrow E_{21} * E_{22}$	E2.loc = T2	E_{21}.code E_{22}.code T2 = E_{21}.loc * E_{22}.loc T1 = a + T2 i := T1
$E_{21} \rightarrow$ id	E_{21}.loc = id.loc	E_{22}.code T2 = b * E_{22}.loc T1 = a + T2 i := T1
$E_{22} \rightarrow$ id	E_{22}.loc = id.loc	T2 = b * c T1 = a + T2 i := T1

$S \rightarrow$ if E then S_1 else S_2

Conditional statement

Production	Translation rule
$S \rightarrow$ if E then S_1 else S_2	E.false = new_label E.true = new_label S_1.next = S.next S_2.next = S.next S.code = E.code || gen(E.false ':') || S_2.code || gen('goto' S.next) || gen(E.true ':') || S_1.code

Homework

- While statement
- Repeat statement
- For statement

Production	Translation rule
$S \rightarrow$ while E do S_1	E.false = new_label E.true = new_label S.code = E.code || gen(E.false':') ||gen('goto' S.next) || gen(E.true':')||S_1.code || gen('goto' S.begin)