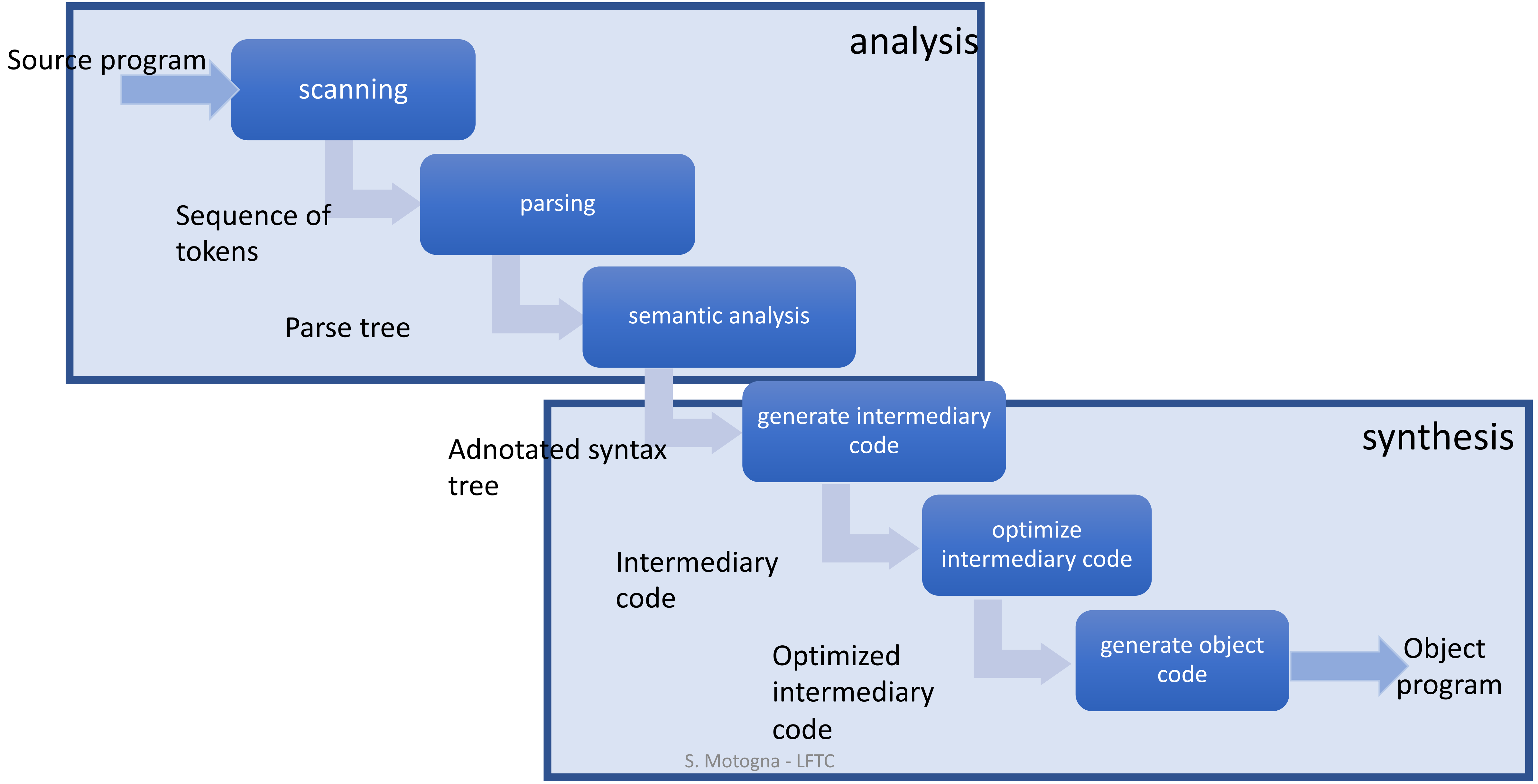
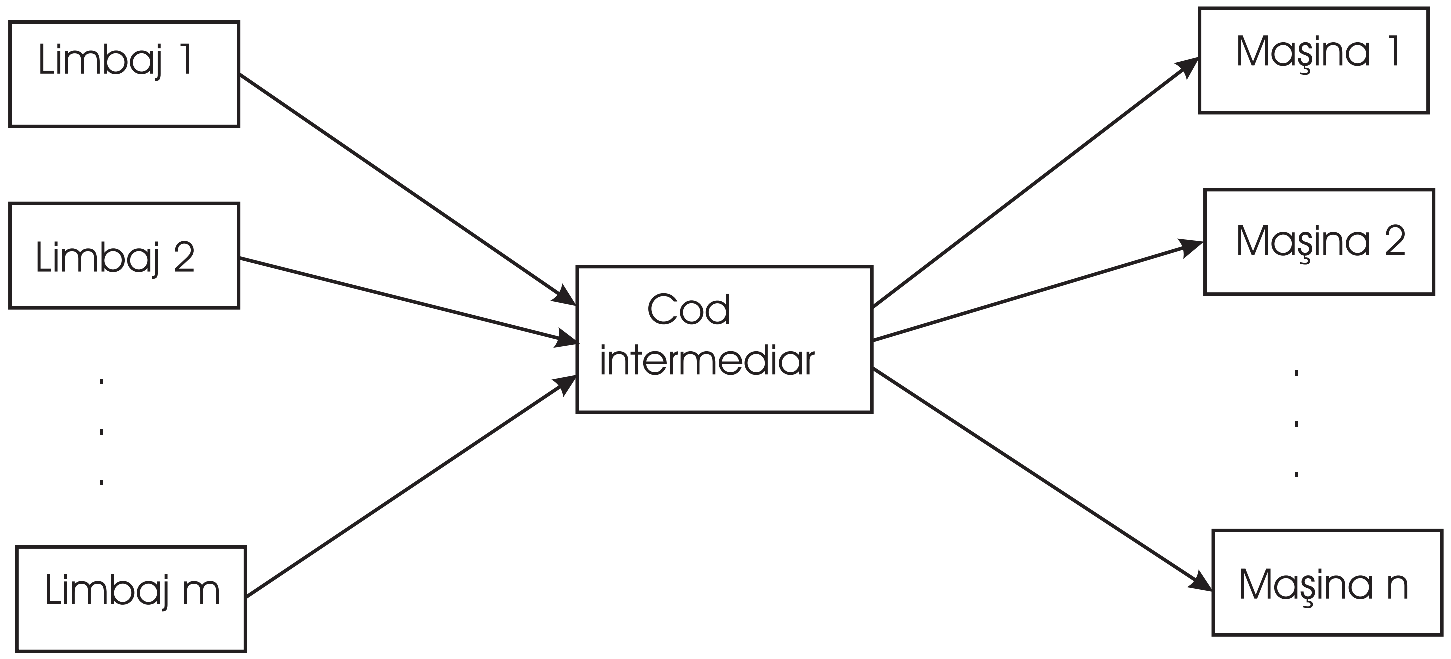
Course 12

Structure of compiler



Generate intermediary code



Forms of intermediary code

- Java bytecode – source language: Java
  – machine language (dif. platforms) JVM
- MSIL (Microsoft Intermediate Language)
  – source language: C#, VB, etc.
  – machine language (dif. platforms) Windows
- GNU RTL (Register Transfer Language)
  – source language: C, C++, Pascal, Fortran etc.
  – machine language (dif. platforms)

Representations of intermediary code

- Adnotated tree: intermediary code is generated in semantic analysis
- Polish postfix form:
  - No parenthesis
  - Operators appear in the order of execution
  - Ex.: MSIL

Exp = a + b * c	ppf = abc*+
Exp = a * b + c	ppf = ab*c+
Exp = a * (b + c)	ppf = abc+*

- 3 address code

3 address code

= sequence of simple format statements, close to object code, with the following general form:

< result >=< arg1 >< op >< arg2 >

Represented as:

- Quadruples
- Triples
- Indirected Triples

- Quadruples:

  < op > < arg1 > < arg2 > < result >

- Triples:

  < op > < arg1 > < arg2 >

(considered that the triple is storing the result)

Special cases:

1. Expressions with unary operator: **< result >=< op >< arg2 >**
2. Assignment of the form **a := b** => the 3 addresss code is **a = b** (no operatorand no 2nd argument)
3. Unconditional jump: statement is **goto L**, where L is the label of a 3 address code
4. Conditional jump: **if c goto L**: if **c** is evaluated to **true** then unconditional jump to statement labeled with L, else (if c is evaluated to false), execute the next statement
5. Function call p(x1, x2, ..., xn) – sequence of statements: **param x1, param x2 , param xn, call p, n**
6. Indexed variables: < arg1 >,< arg2 >,< result > can be array elements of the form **a[i]**
7. Pointer, references: **&x, ∗x**

Example: $b*b-4*a*c$

op	arg1	arg2	rez
*	b	b	t1
*	4	a	t2
*	t2	c	t3
-	t1	t3	t4

nr	op	arg1	arg2
(1)	*	b	b
(2)	*	4	a
(3)	*	(2)	c
(4)	-	(1)	(3)

Optimize intermediary code

- Local optimizations:
  - Perform computation at compile time – constant values
  - Eliminate redundant computations
  - Eliminate inaccessible code – if…then...else...

- Loop optimizations:
  - Factorization of loop invariants
  - Reduce the power of operations

Eliminate redundant computations

Example:

D:=D+C*B
A:=D+C*B
C:=D+C*B

(1)	*	C	B
(2)	+	D	(1)
(3)	:=	(2)	D
~~(4)~~	~~*~~	~~C~~	~~B~~
(5)	+	D	(4)
(6)	:=	(5)	A
~~(7)~~	~~*~~	~~C~~	~~B~~
~~(8)~~	~~+~~	~~D~~	~~(7)~~
(9)	:=	(8)	C

Factorization of loop invariants

```
for(i=0, i<=n,i++)
      { x=y+z;
      a[i]=i*x}
```

```
x=y+z;
for(i=0, i<=n,i++)
      { a[i]=i*x}
```

Reduce the power of operations

```
for(i=k, i<=n,i++)
        { t=i*v;
        . . .}
```

```
t1=k*v;
for(i=k, i<=n,i++)
        { t=t1;
        t1=t1+v;...}
```

Generate object code

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

- Depend on “machine”: architecture and OS

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

Computer with accumulator

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

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!)

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	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 ranges 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)