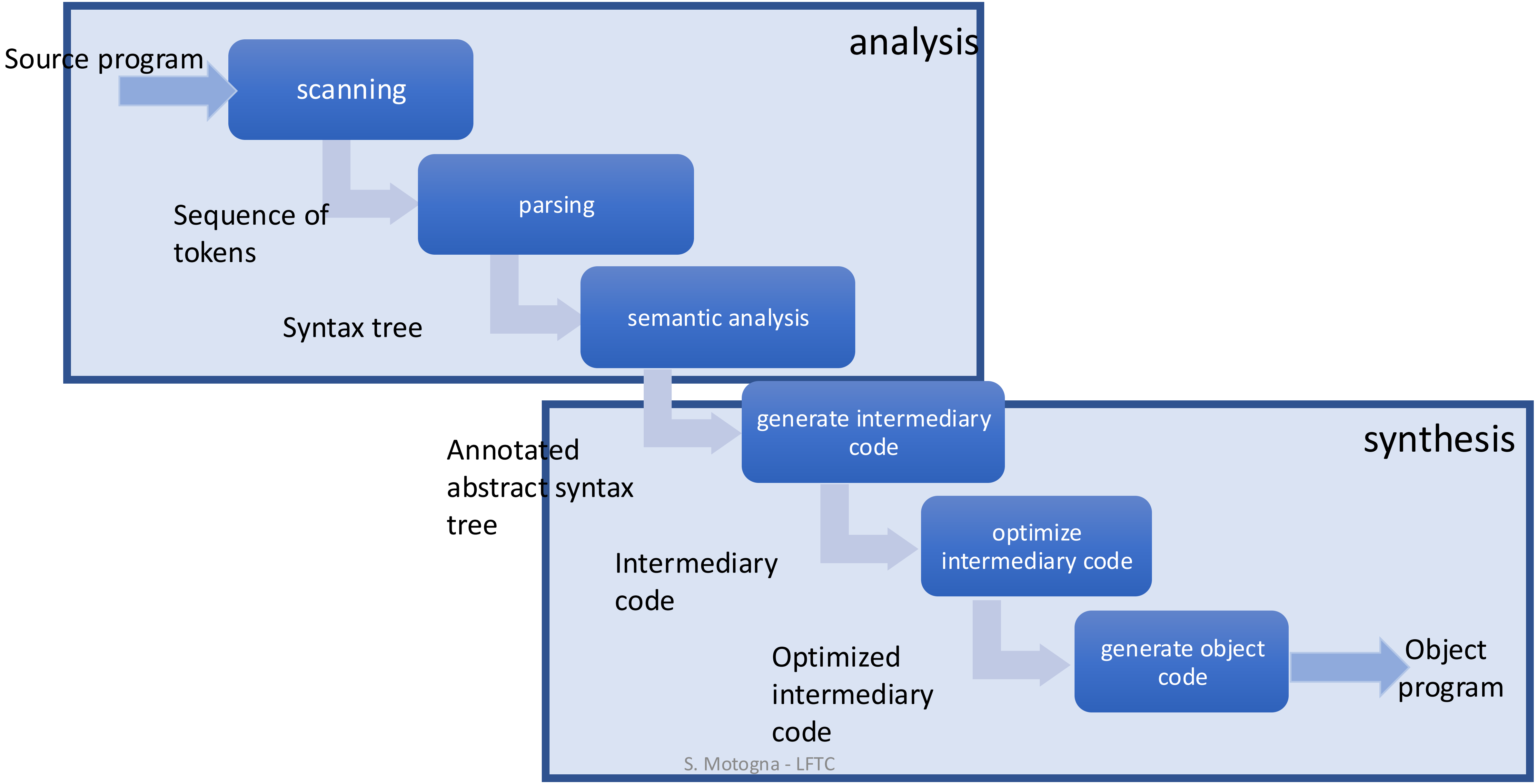
Course 11

Structure of compiler



Semantic analysis

- Parsing – result: syntax tree (ST)
- Simplification: abstract syntax tree (AST)
- Annotated abstract syntax tree (AAST)
  - Attach semantic info in tree nodes

Example

Semantic analysis

- Attach meanings to syntactical constructions of a program
- What:
  - Identifiers -> values / how to be evaluated
  - Statements -> how to be executed
  - Declaration -> determine space to be allocated and location to be stored
- Examples:
  - Type checkings
  - Verify properties
- How:
  - **Attribute grammars**
  - Manual methods

Attribute grammar

- Syntactical constructions (nonterminals) – attributes

$$\forall X \ \in N \ \cup \ \Sigma : A(X)$$

- Productions – rules to compute/ evaluate attributes

$$\forall p \ \in P : R(p)$$

Definition

AG = (G,A,R) is called ***attribute grammar*** where:

- G = (N,Σ,P,S) is a context free grammar
- A = {A(X) | X $\in$N U Σ} – is a finite set of attributes
- R = {R(p) | p $\in$P} – is a finite set of rules to compute/evaluate attributes

Example 1

- G = ({N,B},{0,1}, P, N}

P: N -> NB

N -> B

B -> 0

B -> 1

$N_1.v = 2 * N_2.v + B.v$

$N.v = B.v$

$B.v = 0$

$B.v = 1$

Attribute – value of number = **v**

- **Synthetized attribute: A(lhp) depends on rhp**
- **Inherited attribute: A(rhp) depends on lhp**

Evaluate attributes

- Traverse the tree: can be an infinite cycle

- Special classes of AG:
  - L-attribute grammars: for any node the depending attributes are on the “*left*”;
    - can be evaluated in one left-to-right traversal of syntax tree
    - Incorporated in top-down parser (LL(1))
  - S-attribute grammars: synthetized attributes
    - Incorporated in bottom-up parser (LR)

Steps

- What? - decide what you want to compute (type, value, etc.)
- Decide attributes:
  - How many
  - Which attribute is defined for which symbol
- Attach evaluation rules:
  - For each production – which rule/rules

Example 2 (L-attribute grammar)

Decl -> DeclTip ListId

ListId -> Id

ListId -> ListId, Id

ListId.type = DeclTip.type

Id.type = ListId.type

$ListId_2$.type = $ListId_1$.type

Id.type = $ListId_1$.type

Attribute – type

int i,j

Example 3 (S-attribute grammar)

ListDecl -> ListDecl; Decl

ListDecl -> Decl

Decl -> Type ListId

Type -> int

Type -> long

ListId -> Id

ListId -> ListId, Id

$ListDecl_1.dim = ListDecl_2.dim + Decl.dim$

$ListDecl.dim = Decl.dim$

$Decl.dim = Type.dim * ListId.no$

$Type.dim = 4$

$Type.dim = 8$

$ListId.no = 1$

$ListId_1.no = ListId_2.no + 1$

Attributes – dim + no – **for which symbols**

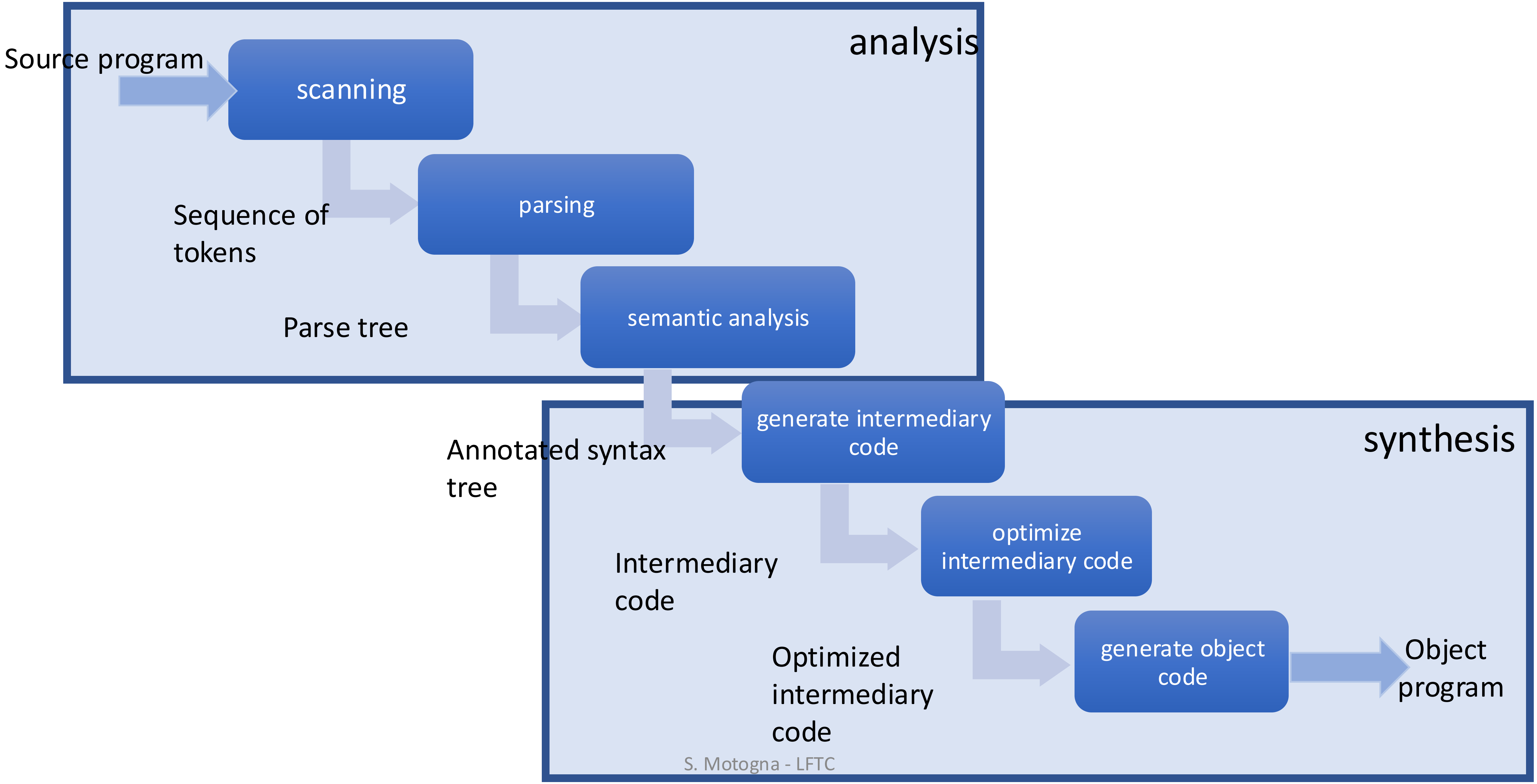
int i,j; long k

Manual methods

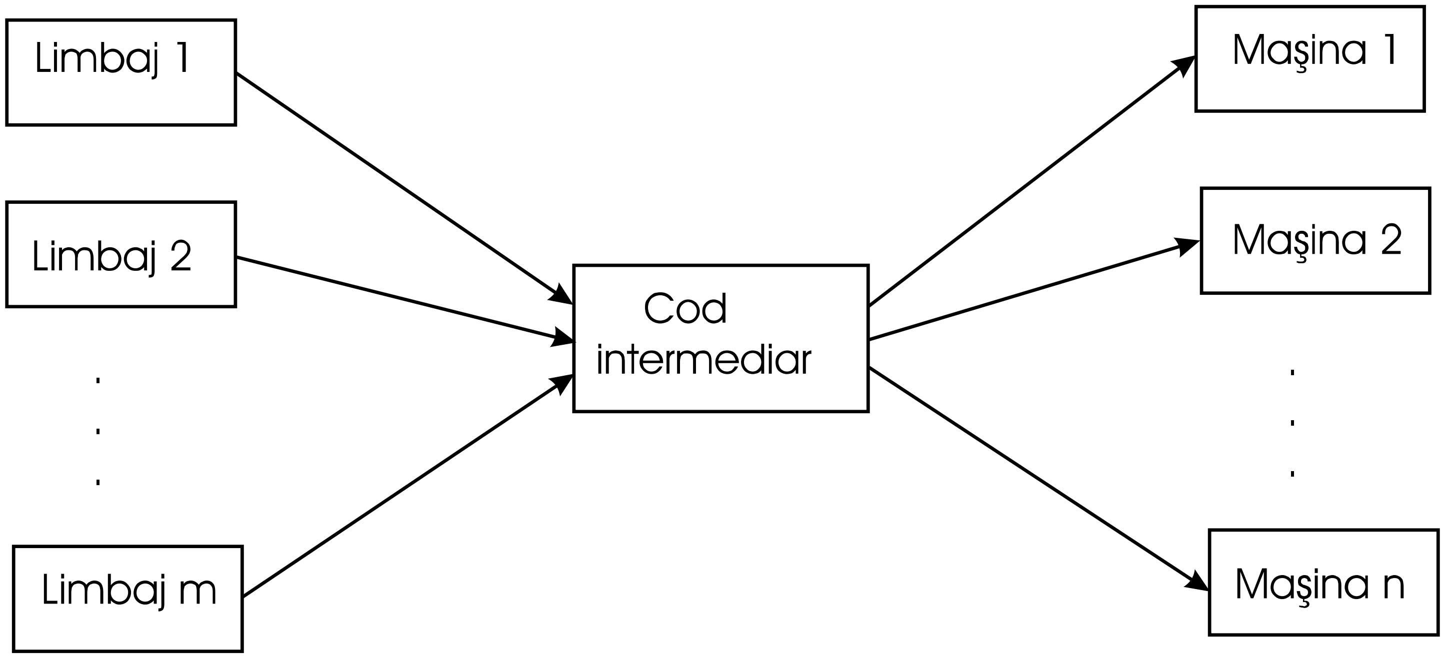
- Symbolic execution
  - Using control flow graph, simulate on stack how the program will behave
  - [Grune – Modern Compiler Design]

- Data flow equations
  - Data flow – associate equations based on data consumed in each node (statement) of the control flow graph: In, Out, Generated, Killed
  - [Grune – Modern Compiler Design], [Kildall], [course]

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

- Annotated 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)

Example 2

If (a<2) then a=b else a=b*b

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

Determine redundant operations

- Operation (j) is redudant to operation (i) with i<j if the 2 operations are identical and if the operands in (j) did not change in any operation between (i+1) and (j-1)
- Algorithm [Aho]

Factorization of loop invariants

What is a loop invariant?

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

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

Challenge

Consider n, and a[i] i=0,n the coefficients of a polynomial P.
Given v, write an algorithm that computes the value of P(v)

3 solutions

P(x) = a[n]*x^n+ … + a[1]*x + a[0] = (a[n]*x^(n-1)+ … + a[1])*x + a[0]

```
V1:
P = a[0]
For i=1 to n
  P = P + a[i]*v^i
```

```
V2:
P = a[0]
Q=v
For i=1 to n
  P = P + a[i]*Q
  Q = Q*v
```

```
V3
P=a[n]
For i=1 to n
  P = P*v + a[n-i]
```

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;...}
```