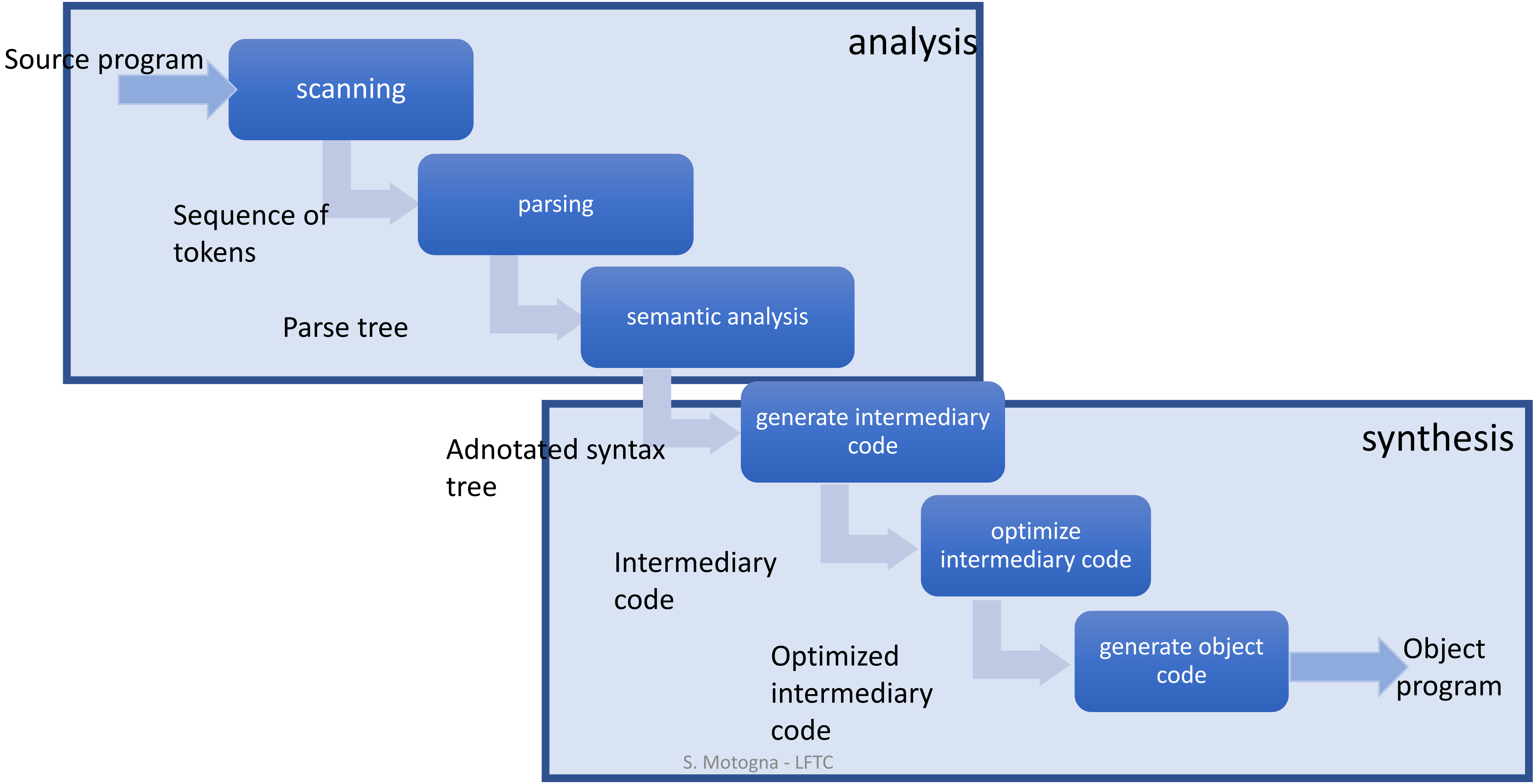
Course 11 & 12

Structure of compiler



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

Semantic analysis – Attribute grammars

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

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,$\boldsymbol{\Sigma}$,P,S) is a context free grammar
- A = {A(X) | X ∈N U $\boldsymbol{\Sigma}$} – is a finite set of attributes
- R = {R(p) | p ∈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
  - S-attribute grammars

Example 2 (L-attribute grammar)

Decl -> DeclTip ListId

ListId -> Id

ListId -> ListId, Id

ListId.type = DeclTip.type

Id.type = ListId.type

$\text{ListId}_2\text{.type} = \text{ListId}_1\text{.type}$

$\text{Id.type} = \text{ListId}_1\text{.type}$

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

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

$ListDecl.dim = Decl.dim$

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

$Type.dim = 4$

$Type.dim =8$

$ListId.nr = 1$

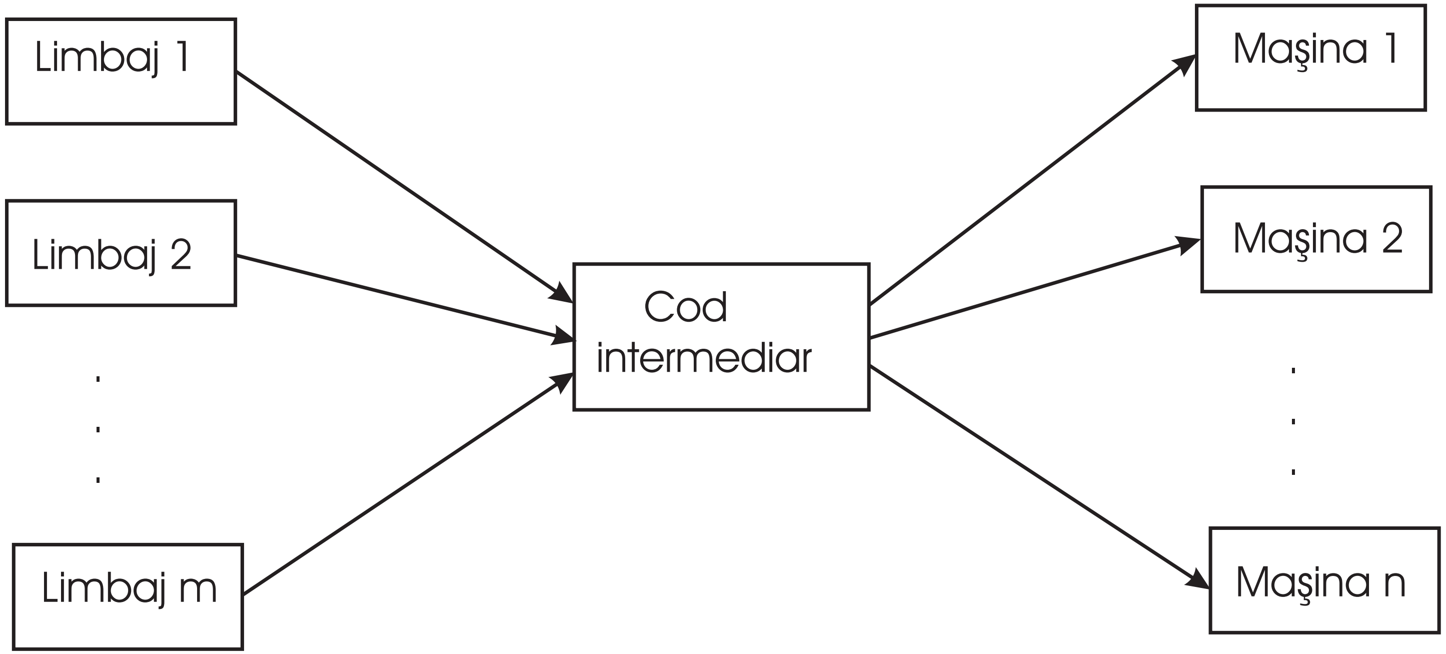
$ListId_1.nr = ListId_2.nr + 1$

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

Proposed problems (HW):

1) Define an attribute grammar for arithmetic expressions
2) Define an attribute grammar for logical expressions
3) Define an attribute grammar for if statement

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
- Triplets
- Indirected Triplets

- Quadruples:

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

- Triplets:

  < op > < arg1 > < arg2 >

(considered that the triplet 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

More examples at seminar

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;...}
```

Course 12

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}

Syntax oriented 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 → id := E	
$E \rightarrow E_1 + E_2$	
$E \rightarrow E_1 * E_2$	
$E \rightarrow (E_1)$	
E → id	
E → const	

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

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.urm) || gen(E.true ':') || S_1.code

Homework

- While statement
- Repeat statement
- For statement
- Condition