LANGUAGE PROCESSORS

UNIT 9: INTERMEDIATE CODE GENERATION

David Griol Barres
dgriol@inf.uc3m.es
Computer Science Department
Carlos III University of Madrid
Leganés (Spain)

OUTLINE

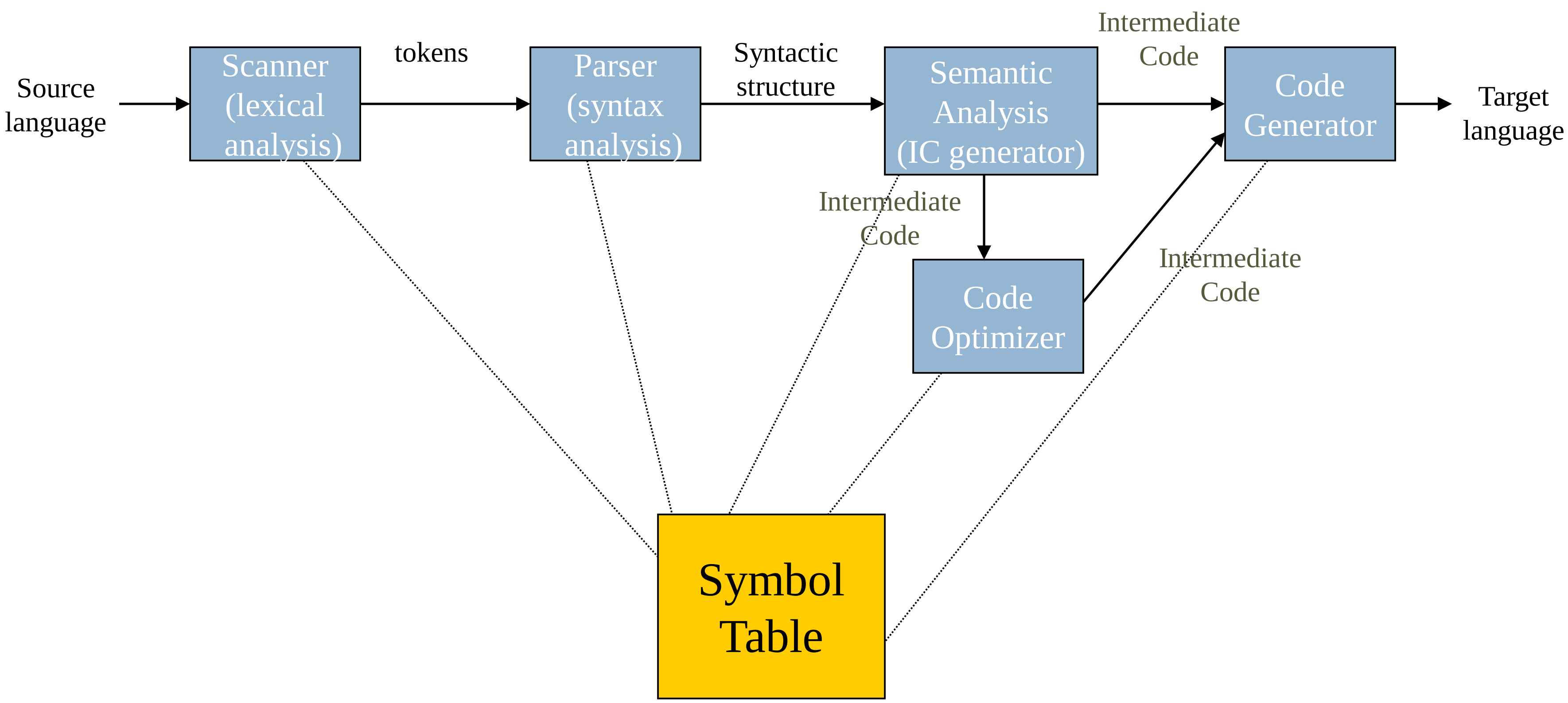
- Introduction
- Advantages and disadvantages
- Compiler architecture review
  - The Role of Intermediate Code
  - Analysis Phase
  - Synthesis Phase
- Intermediate Languages Types
  - Graphical IRs
  - Linearized IC

OUTLINE

- Graphical IRs
  - Abstract Syntax Trees
  - Directed Acyclic Graphs
  - Control Flow Graphs
- Linearized IC
  - Stack based
  - Three-Address Code
    - Triples and Quadruples
- Declarations

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Compiler Architecture



Intermediate Code

- **Advantages**:
  - Allows the analysis phase to be machine independent (language - machine independence).
  - Makes optimization easier (not machine-dependent).
  - The same analysis and/or optimizer can be used on the same intermediate code.
  - Facilitates the division in phases of a compiler project.

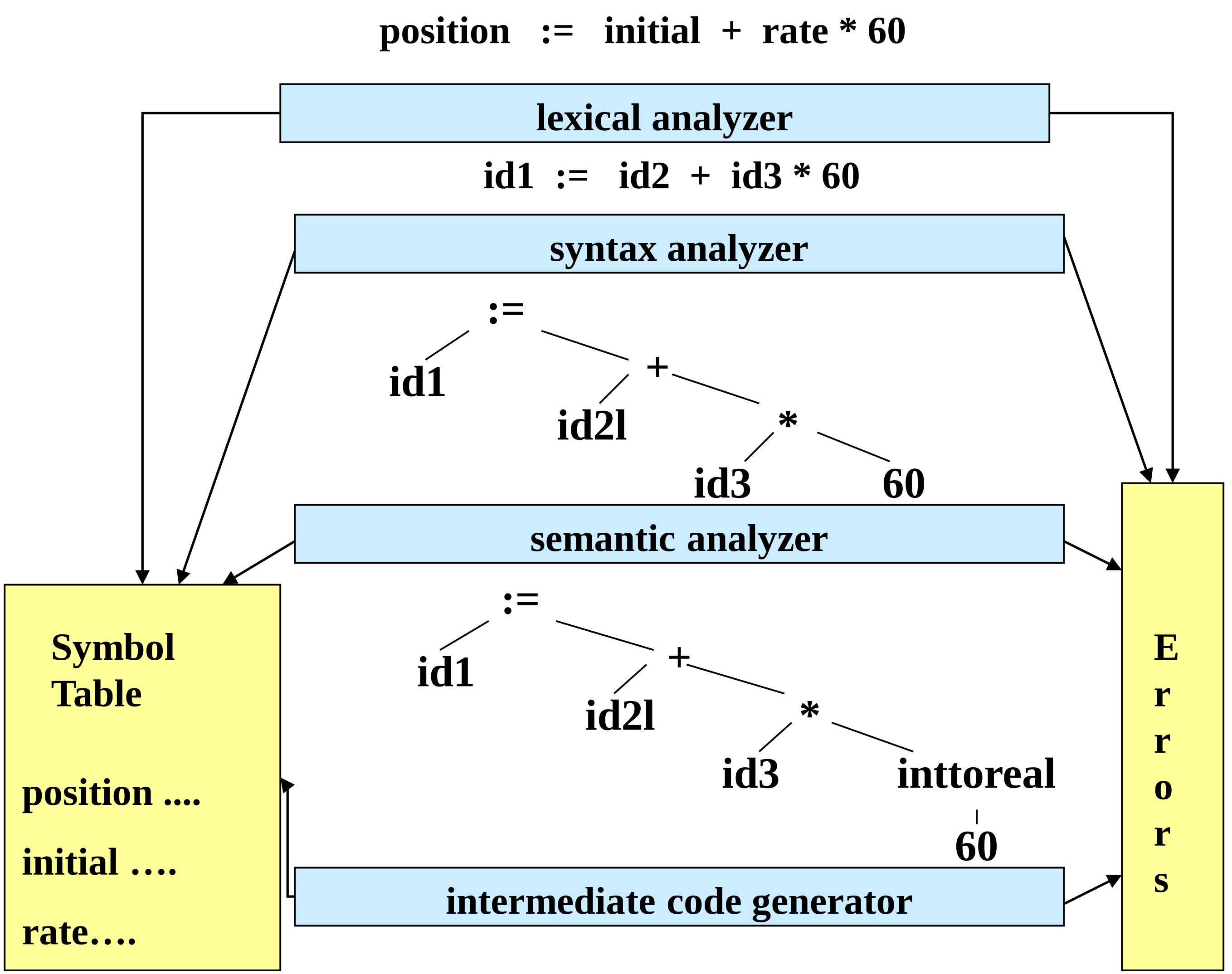
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Intermediate Code

- **Disadvantages**:
  - Loss of efficiency (not only one pass).
  - Introduces a new translation phase in the compiler.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Analysis Phase



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Synthesis Phase

Symbol Table

position ....

initial ….

rate….

intermediate code generator

temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

`3 address code`

code optimizer

temp1 := id3 * 60.0
id1 := id2 + temp1

final code generator

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R1, R2
MOVF R1, id1

Errors

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Intermediate Languages Types

Depends on the target program (often depends on the machine):

- **Graphical IRs:**
  - Abstract Syntax Trees (AST),
  - Direct Acyclic Graphs (DAGs),
  - Control Flow Graphs (CFG).
- **Linear IRs:**
  - Stack based (postfix).
  - Three address code (triples, indirect triples, quadruples).

Graphical IRs

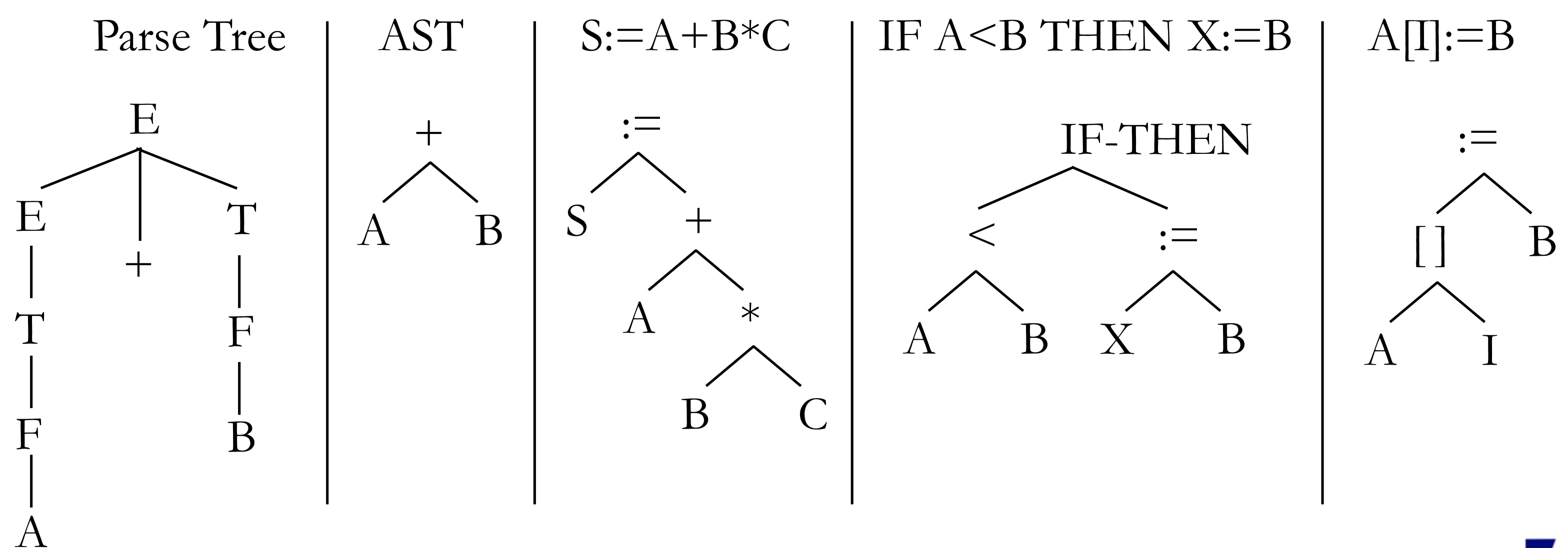
- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication.
- Control flow graphs (CFG) – explicitly model control flow.

Linearized IC

- **Postfix notation:**
  - Operators follow operands
    - S = A + B * C ⇨ → S A B C * + =
  - Benefits
    - Simple notation
  - Disadvantages
    - Difficult to understand code
    - not useful for optimizations
- **Three-address code:**
  - quadruples
  - triples
  - Indirect triples

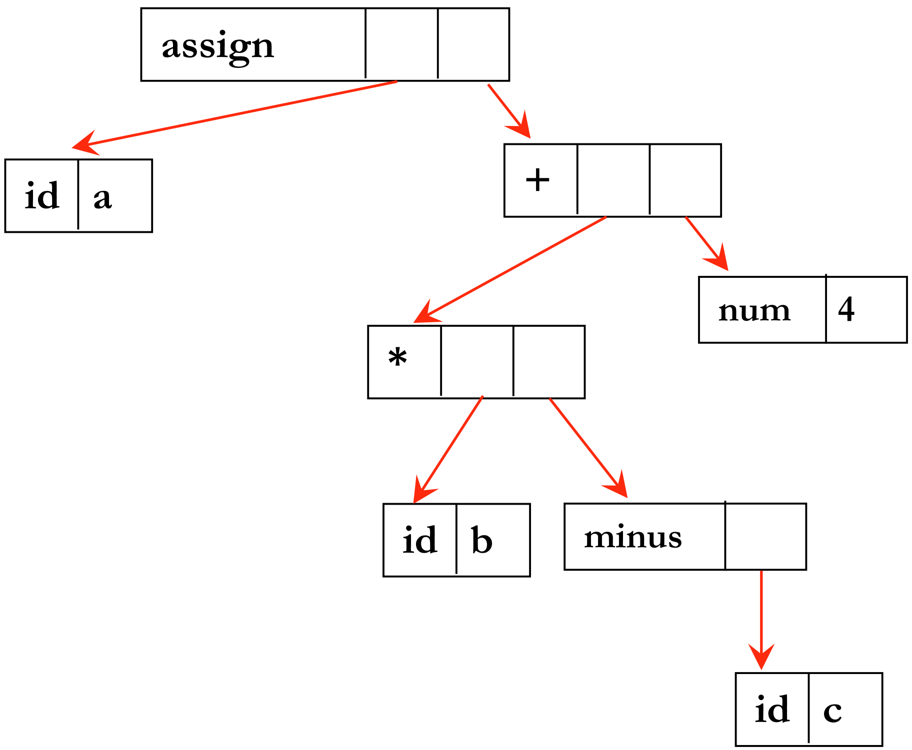
Abstract Syntax Tree

- Condensed form of a parse tree useful for representing language constructs.
- Keywords and operators appear as interior nodes.
- Examples:



Abstract Syntax Tree

- Include semantic information:
  - example: a=b*-c+4



Three-Address Code

- Sequence of statements of the form:
  - x = y op z
  - One operator and up to three addresses.
  - Compiler-generated temporary variables.
- Directly related to the evaluation of expressions:
  - Example: a = b*(c+d) is translated to:

    tmp1 = c+d

    a = b*tmp1

Three-address code

- Lineal representation of a AST

Example 1:
a=b*-c+4

3-address code

```
t1=-c
t2=b*t1
t3=4
t4=t3+t2
a=t4
```

Example 2:

```
if cond then then_statements
else
  else_statements;
end if;
```

3-address code

t1 = cond
if not t1 goto *else_label*
code for the "then_statements"
goto *endif_label*
else_label:
code for the "else_statements"
endif_label:

Types of 3-address statements

- Assignment: **x=y op z** (op is a binary arithmetic or logical operator)
- Assignment **x=op y** (unary operators)
- copy **x:=y**
- Unconditional jump: **goto L** (L label)
- Conditional **gotoc** x L (conditional jump **if x relop y goto L)**

Types of 3-address statements

- Calls to routines:

  param x1

  ...

  param xn

  call p,n
- Indexed assignments

  x:=y[i]

  x[i]=y
- Pointer assignments

  x:=&y

  x:=*y

  ***x:=y**

3-address code implementation

- Quadruples
  - Record structure with 4 fields: (op,y,z,x) to represent x=y op z
- Triples
  - The fourth value is a temporary value.
  - The index is used instead.

Example: a = b+(c*d)

[quadruples]	[triples]
1. (mul,c,d,t1)	1: (mul,c,d)
2. (add,b,t1,t2)	2: (add,b,(1))
3. (asn,a,t2,_)	3: (asn,a,(2))

3-address statements

- Quadruples
  - 4 values:

    (<OPERATOR>,<Operand$_1$>,<Operand$_2$>,<Result>)
  - Examples

Expression	Quadruples				
S:=A+B*C	*	B	C	T_1	(*, B, C, T_1)
	+	A	T_1	T_2	(+, A, T_1, T_2)
	:=	T_2		S	(:=, T_2, , S)
IF A<B THEN X:=B	<	A	B	E_1	(<, A, B, E_1)
	GOTOC	E_1		E_2	(GOTOC, E_1, , E_2)
	:=	B		X	(:=, B, , X)
	LABEL			E_2	(LABEL, , , E_2)

Triples

- Quadruples are more general but:
  - Need too much space.
  - Too many auxiliary variables to store intermediate results.
- Triples omitts the last operand, it is implicit and associated to the triple:

  (<OPERATOR>, <Operand$_1$>, <Operand$_2$>)
  - Equivalent to AST

Declarations

- Construction of the symbol table
  - Create an entry for each local name (within a procedure or a block) including information like the type and the relative address of the storage for that name
- Reserve space
  - *Offset:* a global variable that keeps track of the next available relative address
  - Synthesized attributes *type* and *width* to indicate the type and the number of memory bits taken by objects of that type