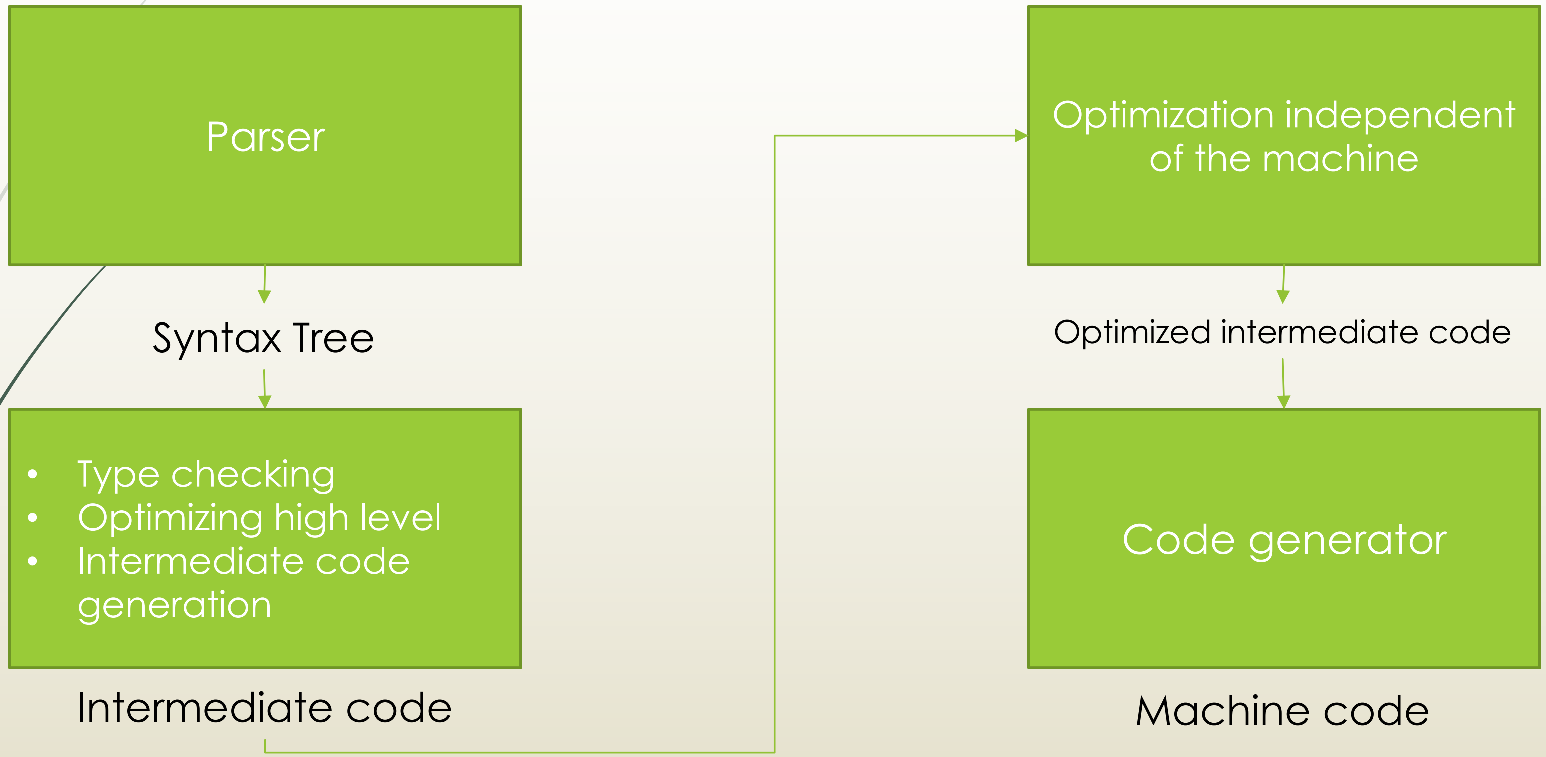
Item 3: Intermediate code generation

Mari Paz Guerrero Lebrero
Grado en Ingeniería Informática
Curso 2018/2019

Intermediate code generation

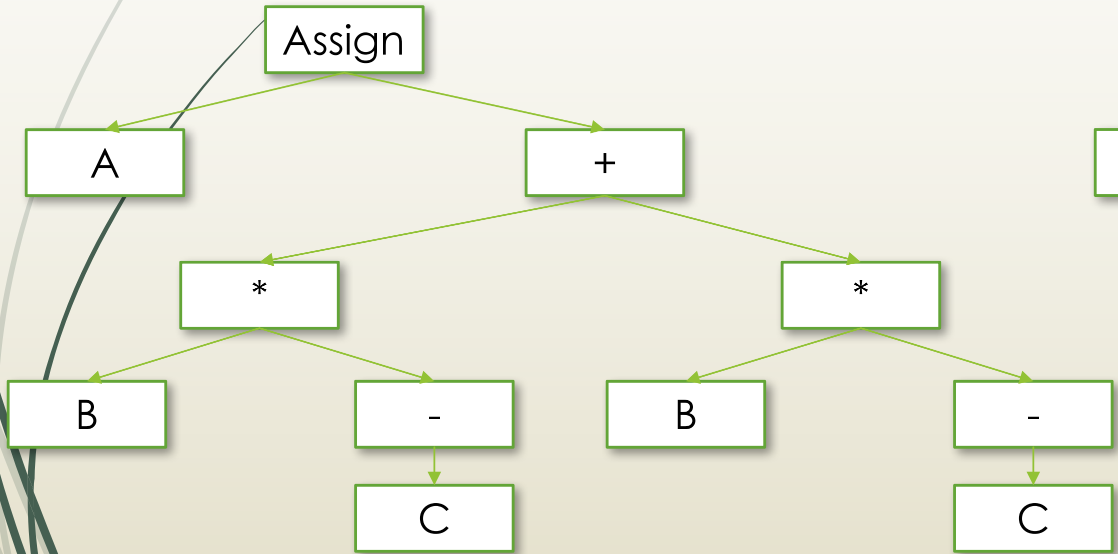


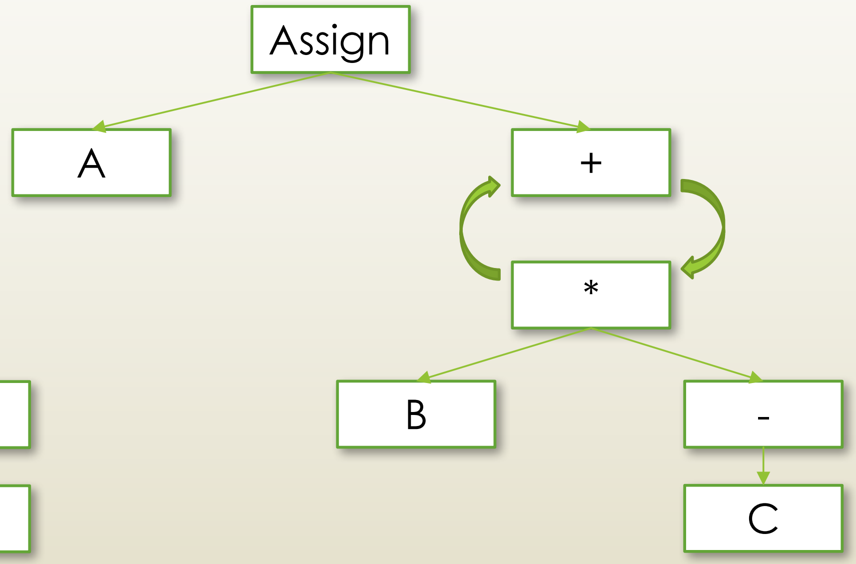
First intermediate representation (IR)

- Syntactic Trees (ST), also called **Abstract Syntax Tree (AST)** and its close relative: Acyclic directed graphs (ADG):
  - They are the 1st intermediate representations
  - Suitable to represent arithmetic and logical expressions, control statements and declarations
  - The type checking is usually done on this representation.

Syntax Trees

- Syntax Tree (ST) and Acyclic Directed Graphs (ADG) for de input:
  - **A = B * - C+ B * - C**

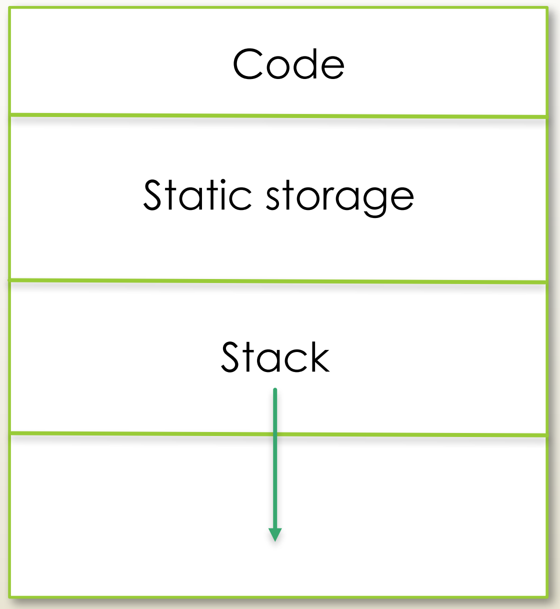




Syntax tree using static memory

0	ID	B	
1	ID	C	
2	MENOSU	1	
3	*	0	2
4	ID	B	
5	ID	C	
6	MENOSU	5	
7	*	4	6
8	+	3	7
9	ID	A	
10	ASIGNAR	9	
11			

It is the method used by Prolog



Computer based on a stack

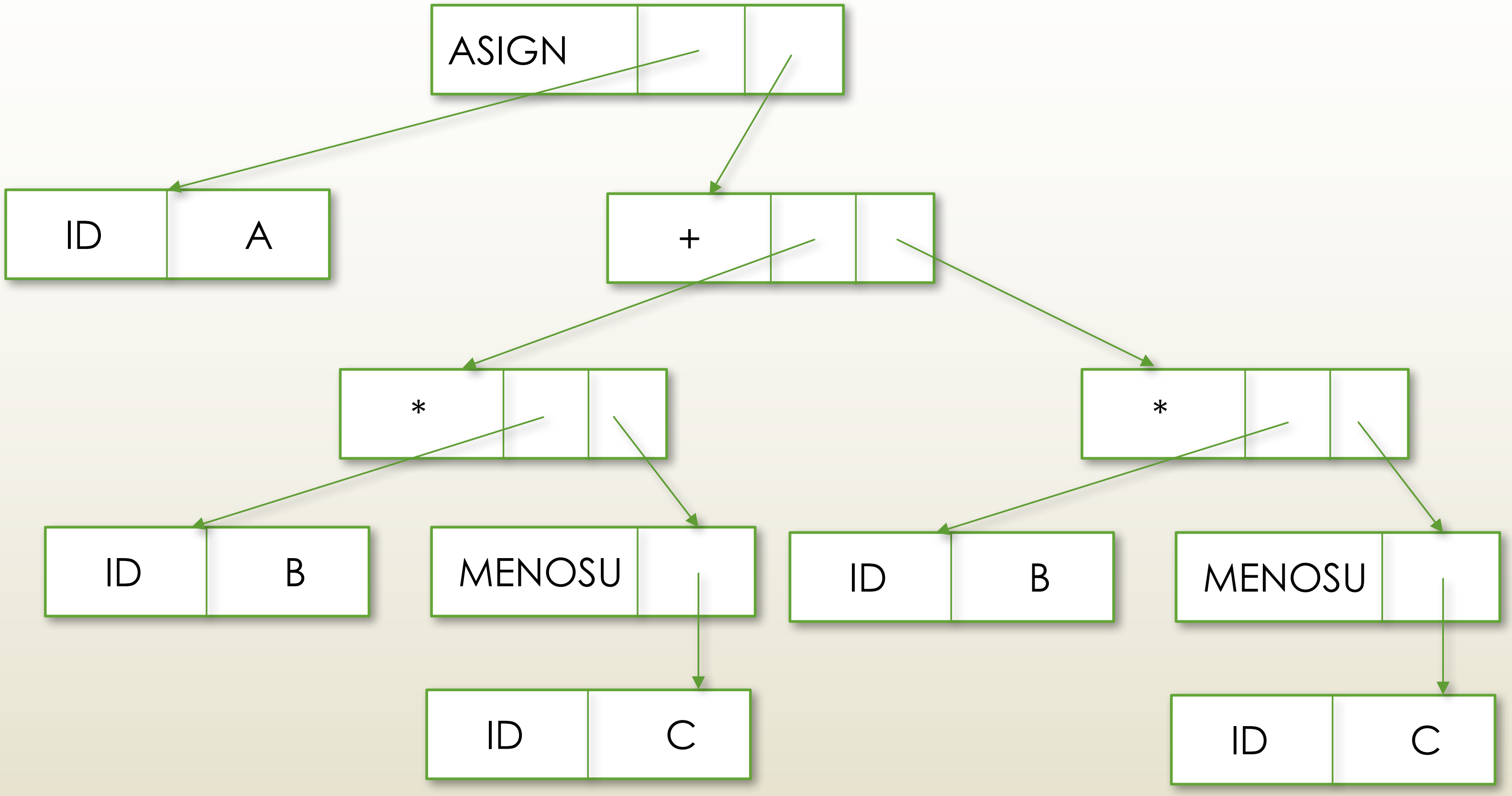
Translation scheme for ST construction

- Translation schemes for assignment statements:
  - If the ***HazNodo*** functions always create a new entry, the result is a syntax tree
  - If ***HazNodo*** functions check that entries are already created, in order to avoid duplications, the result will be an acyclic graph

Translation scheme to generate a ST or ADG using assignment statements

Grammar	Semantic rules
S→ID ':=' E	`S.ptr := hazNodoAsign(hazNodoId(ID.lexval), E.ptr)`
E→E1 + E2	`E.ptr := hazNodoSuma(E1.ptr, E2.ptr)`
E→E1 – E2	`E.Ptr := hazNodoResta(E1.ptr, E2.ptr)`
E→E1 * E2	`E.ptr := hazNodoProd(E1.ptr, E2.ptr)`
E→- E1	`E.ptr := hazNodoMenosU(E1.ptr)`
E→(E1)	`E.ptr := E1.ptr`
E→ID	`E.ptr := hazNodoId(ID.lexval)`
E→NUM	`E.ptr := hazNodoNum(Num.lexval)`

Resulting ST



Second IR: postfix Polish notation

- It is used to represent arithmetic and logical expressions.
- It is directly implementable on a machine with a stack.
- It is very close to the machine language of some processors.
  - Intel X87 has 8 floating point registers as a stack, so the assembly language for x87 expressions of shaped postfix notation Polish.

Machine base on a stack

- Intermediate code for an abstract computer based on a stack (postfix Polish notation). The instruction set of this computer will be composed of:
  - Arithmetic and logical instructions: take data from top of the stack and leave in place the result on top of the stack
    - +, -, *, / , mod
  - Management variables and constants:
    - RVALUE id, LVALUE id, ASIGNAR, PUSH, ctenum, POP, COPY

Stack code for an assignment

Example:
A : = B * -C + B * -C

```
LVALUE A
RVALUE B
RVALUE C
MENOSU
*
RVALUE B
RVALUE C
MENOSU
*
+
ASIGNAR
```

Stack scheme translation

Example generated code for the following example:
`dia := (1461*anyo) div 4 + (153 * mes + 2) div 5 + d`

```
LVALUE dia
PUSH 1461
RVALUE anyo
*
PUSH 4
/
PUSH 153
RVALUE mes
*
PUSH 2
+
PUSH 5
/
+
RVALUE d +
ASIGNAR
```

Generated postfix code

Productions with semantic actions for a computer base on stack

```
SentAsig -> ID := {emite(LVALUE, ID.lexval);}
            ExpA {emite(ASIGNAR);}
ExpA     -> ExpA '+' SumA {emite('+');}
ExpA     -> ExpA '-' SumA {emite('-');}
ExpA     -> SumA
SumA     -> SumA '*' FactA {emite('*');}
SumA     -> SumA '/' FactA {emite('/');}
SumA     -> FactA
FactA    -> '-' FactA {emite(MENOSU);}
FactA    -> '(' ExpA ')'
FactA    -> NUMERO {emite(PUSH, NUMERO.lexval);}
FactA    -> ID {emite(RVALUE, ID.lexval);}
```

Example floating point code

```c
#include <stdio.h>

float a, b, c;
int main() {
    a = 7.0;
    b = 2.0;
    c = 3.0;
    c = a + 3.0 * b * b / 5.0;
    printf("El resultado = %f\n", c);
    return 0;
}
```

```
    .file   "flotante02.c"
    .comm   a, 4, 4
    .comm   b, 4, 4
    .comm   c, 4, 4
    .section    .rodata
.LC3:
    .string "El resultado es = %f\n"
    .section    .rodata
.LC0:
    .float 7.0
.LC1
    .float 2.0
.LC2
    .float 3.0
.LC4
    .float 5.0
    .text
    .globl main
    .type main, @function
main:
    pushl %ebp          ##PRÓLOGO
    movl %esp, %ebp     ##PRÓLOGO
```

Example floating point code

```c
#include <stdio.h>

float a, b, c;
int main() {
    a = 7.0;
    b = 2.0;
    c = 3.0;
    c = a + 3.0 * b * b / 5.0;
    printf("El resultado = %f\n", c);
    return 0;
}
```

```
movl .LC0, %eax      # a = 7.0
movl %eax, a
movl .LC1, %eax      # b = 2.0
movl %eax, b
flds a       # pushFL a
flds .LC4    # pushFL 5.0 (el divisor primero)
flds .LC2    # pushFL 3.0
flds b       # pushFL b
fmulp        # pushFL(popFL(b) * popFL(3.0))
flds b       # pushFL b
fmulp        # pushFL(popFL(b) * popFL(3.0*b))
fdivp        # pushFL(popFL(3.0*b*b)/popFL(5.0))
faddp        # pushFL(popFL(3.0*b*b/5.0) + popFL(a))
fstps c          # almacena el resultado en variable c
subl $8, %esp    # reserva espacio en la pila
flds c           #convierte c a doble precisión
fstpl (%esp)     # lo almacena (empuja) a la pila
movl $.LC3, %eax    #empuja a la pila cadena
pushl %eax
call printf
movl $0, %eax       # return 0;
leave               # EPÍLOGO
ret                 # EPÍLOGO
```

Remember that X87 Intel has eight floating-point registers in a stack, so that the assembly language expressions x87 floating shaped Polish postfix

Third IR: Three-address code

- It is used to represent arithmetic and logical expressions.
- You may be considered a linear representation of the syntax tree, or GDA.
- Each node is labelled with a temporary variable
  - Result <- data1 operator data2

Three-address code for AST

- Three-address code corresponding to the syntax tree:

Example:
A : = B * -C + B * -C

```
T1:= -C
T2:= B*T1
T3:= -C
T4:= B*T3
T5:= T2+T4
A:= T5
```

Three-address code for GDA

- Three-address code corresponding to the directed acyclic graph:

Example:
A : = B * -C + B * -C

```
T1:= -C
T2:= B*T1
T5:= T2+T2
A:= T5
```

Instruction set for three-address code

- Intermediate code three addresses for **assignment statements**. Overall our instruction set will be as follows:
  1. Assignment statements of the form "X: = Y op Z", where op is a binary arithmetic or logical operator (+, -, *, /, OR, AND, ...)
  2. Assignment statements of the form "X: = op Y", where *op* is a unary operator
  3. Copy sentences: "X: = Y"
  4. Allocations pointers and addresses: "X: = & Y", "X: = * Y", "* X: = Y"
  5. In cases of medium and high level code , usually allows the use of assignments with indexes.
     - "X: = Y [i]", "X [i]: = Y". // Working directly with pointers In the case of low-level code.

Translation scheme for three-address code

- Translation scheme for three-address code for **assignment statements**

```
SentAsign -> ID ‘:=‘ E {emite(ID.lexval, ‘:=‘, E.tmp);}
E         -> E1 + S {E.tmp := nuevoTmp( ); emite(E.tmp, ‘:=‘, E1.tmp,’+’,S.tmp);}
E         -> E1 - S {E.tmp := nuevoTmp( ); emite(E.tmp, ‘:=‘, E1.tmp,’-’,S.tmp);}
E         -> S {E.tmp := S.tmp;}
S         -> S1 * F { S.tmp := nuevoTmp( ); emite(S.tmp,’:=‘,S1.tmp,’*’,F.tmp);}
S         -> S1 / F { S.tmp := nuevoTmp( ); emite(S.tmp,’:=‘,S1.tmp,’/’,F.tmp);}
S         -> F { S.tmp := F.tmp;}
F         -> - F1 {F.tmp := nuevoTmp( ); emite(F.tmp, “:=“,MENOSU,F1.tmp);}
F         -> ID {F.tmp : = ID.lexval;}
F         -> NUMERO {F.tmp := nuevoTmp( ); emite(F.tmp, ‘:=‘, NUMERO.lexval);}
```

Translation scheme for three-address code

- In the above translation scheme there are two things to consider:
  1. We have used a function in nuevoTmp () that every time it runs, we generate a new temporary variable. Your code in C ++ could be:

```
string nuevoTmp( void ) {
    static int cont = 0;
    ostringstream os << "t" << cont++;
    return os.str( );
} // fin de nuevo Tmp( )
```

  2. The induction hypothesis used is that the synthesized attributes E.tmp, S.tmp, and F.tmp are the variable names (maybe temporary) where the result is stored.