Semantics & Intermediate Representation

Statements

(Non-linear Flows)

Statements

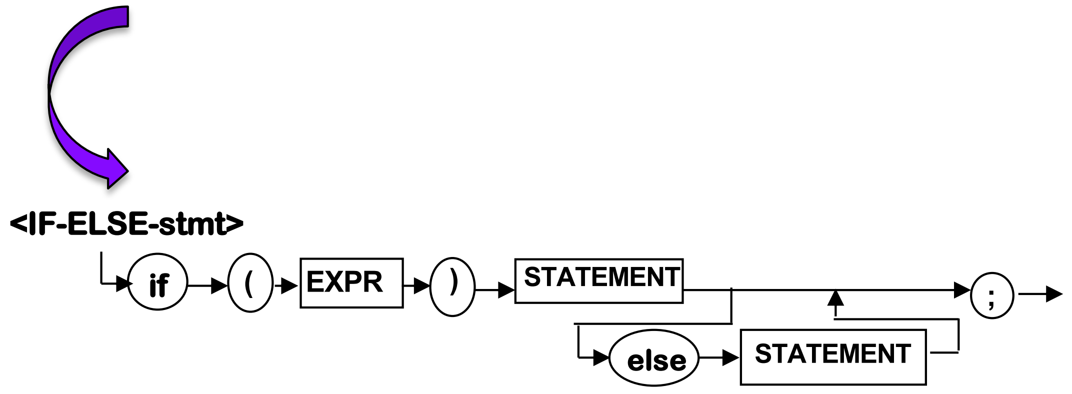
- ***In any programming language there is a group of statements that require a NON-LINEAR Execution, either because there is a condition that must be satisfied or because the instruction must be repeated multiple times in Run-time.***

- ***These statements are Conditions and Loops. Examples:***
  - ***if (EXPR) <statement>;***
  - ***if (EXPR) <statement> else <statement>;***
  - ***while (EXPR) <statement>***
  - ***...***

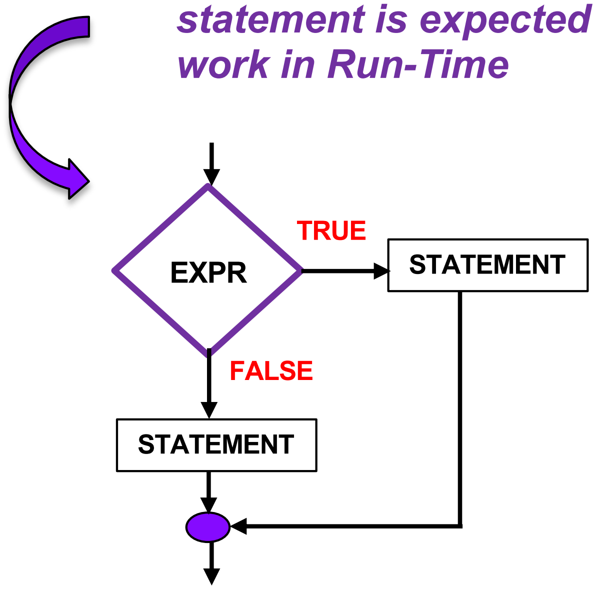
Non-linear Execution

- *This kind of statements MUST be analyzed from 2 different perspectives:*
  - *Their syntax flow*
  - *Their execution flow*

Syntax Flow: describe the sequence of tokens



Execution Flow: describe the way the statement is expected to work in Run-Time



Intermediate Representation for Statements

- ***When we have CONDITIONS in Compile-time we don’t know which statement is going to be executed because EXPR doesn’t have a VALUE. That means there’s no way we can find out if the TRUE statement or the FALSE statement will be executed.***

- ***We’ll have to compile EVERY statement and we’ll need to ADD some OPERATION-CODES that will provide a way to AVOID one statement or the other.***

- ***We’ll need JUMPS for Conditional execution.***

Intermediate Representation for Statements

- ***NON-LINEAR Operation CODES.***
    - **GOTOF , EXPR, , DESTINATION**
    - **GOTOV , EXPR, , DESTINATION**
    - **GOTO , , , DESTINATION**

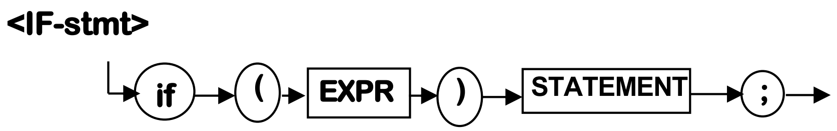
- ***GOTOF and GOTOV***
    - ***Conditional JUMPS that, depending on the value of EXPR (in RUN-Time, change the InstructionPointer to a specific line of code (DESTINATION)***

- ***GOTO***
    - ***Unconditional JUMP that changes the InstructionPointer to a specific line of code (DESTINATION)***
- ***An EXTRA Stack will be needed (PendingJumps Stack)***

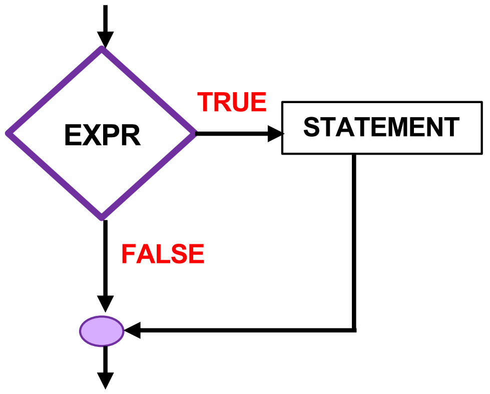
Intermediate Representation for Statements

- ***Simple Condition Statement***

Syntax Flow

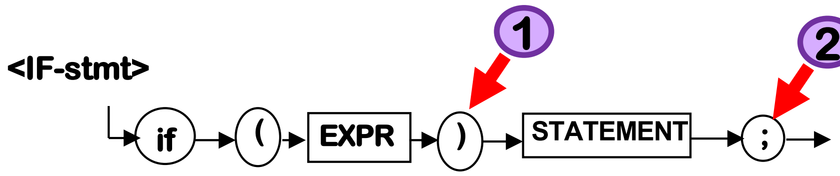


Execution Flow



Intermediate Representation for Statements

- ***Simple Condition Statement***



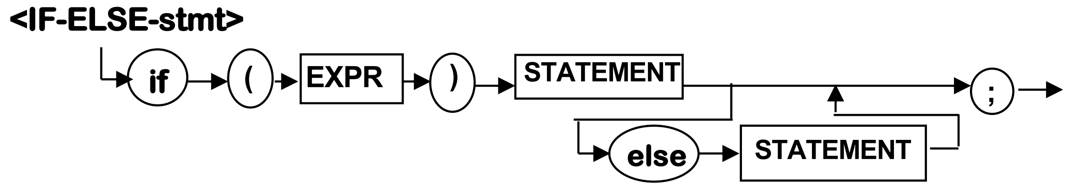
```
1.- exp_type = PTypes.Pop()
    if (exp_type != bool) ERROR(Type-mismatch)
    else
        result = PilaO.Pop()
        Generate quad:   GotoF, result , ,  ___
        PJumps.Push (cont-1)
2.- end=PJumps.Pop()
    FILL (end, cont)
```

Our Quad_Pointer is always pointing to the next quadruple to be generated

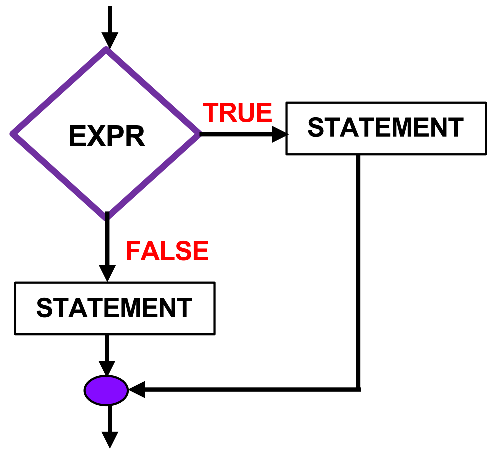
Intermediate Representation for Statements

- *TWO-Ways Condition Statement*

Syntax Flow

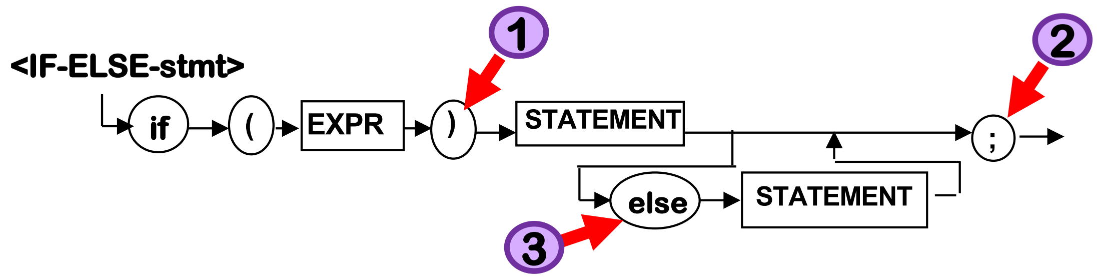


Execution Flow



Intermediate Representation for Statements

- ***Two-ways Condition Statement***



```
1.- exp_type = PTypes.Pop()
    if (exp_type != bool) ERROR(Type-mismatch)
    else
       result = PilaO.Pop()
       Generate quad:   GotoF, result , , ___
                        PJumps.Push (cont-1)
```

```
2.- end=PJumps.Pop()
    FILL (end, cont)
```

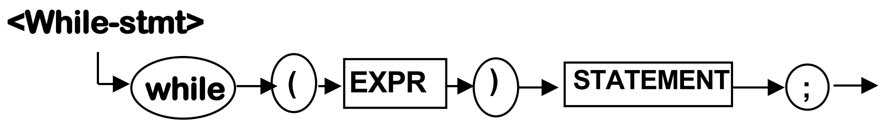
```
3.- Generate quad: GOTO ___
    false= PJumps.Pop()
    PJumps.Push(cont-1)
    FILL (false, cont)
```

Our Quad_Pointer is always pointing to the next quadruple to be generated

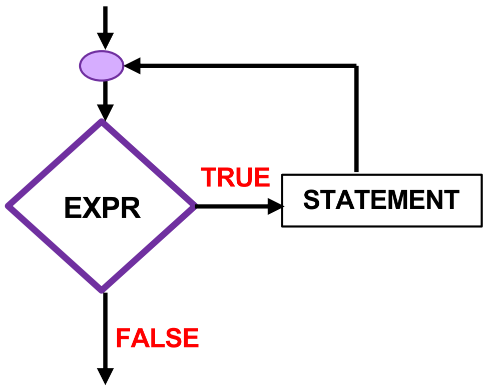
Intermediate Representation for Statements

- ***Simple PRE-Condition LOOP Statement***

Syntax Flow

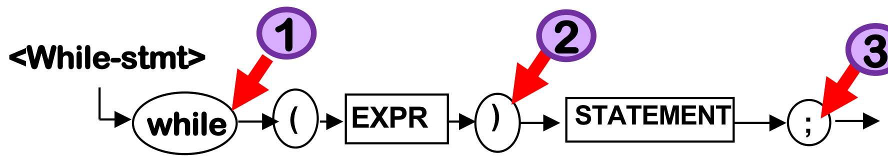


Execution Flow



Intermediate Representation for Statements

- ***Simple PRE-Condition LOOP statement***



```
1.- PJumps.Push (cont)
2.- exp_type = PTypes.Pop()
    if (exp_type != bool) ERROR(Type-mismatch)
    else
       result = PilaO.Pop()
       Generate quad:   GotoF,  result ,  ,  ___
       PJumps.Push (cont-1)
3.- end=PJumps.Pop()
    return=PJumps.Pop()
    Generate quad: GOTO return
              FILL (end, cont)
```

Our Quad_Pointer is always pointing to the next quadruple to be generated