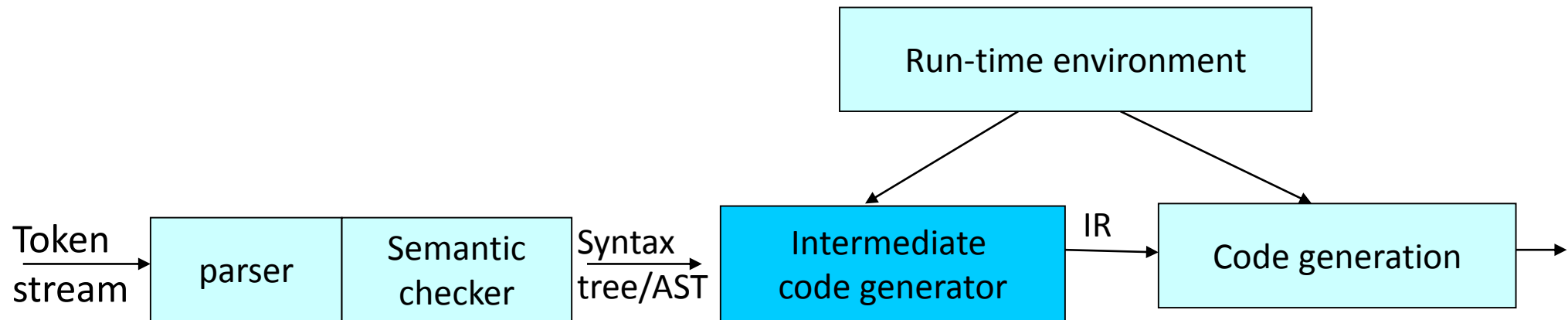


Intermediate Code Generation

- **Intermediate codes** are machine independent codes, but they are close to machine instructions
- The given program in a source language is converted to an **equivalent program** in an intermediate language by the intermediate code generator

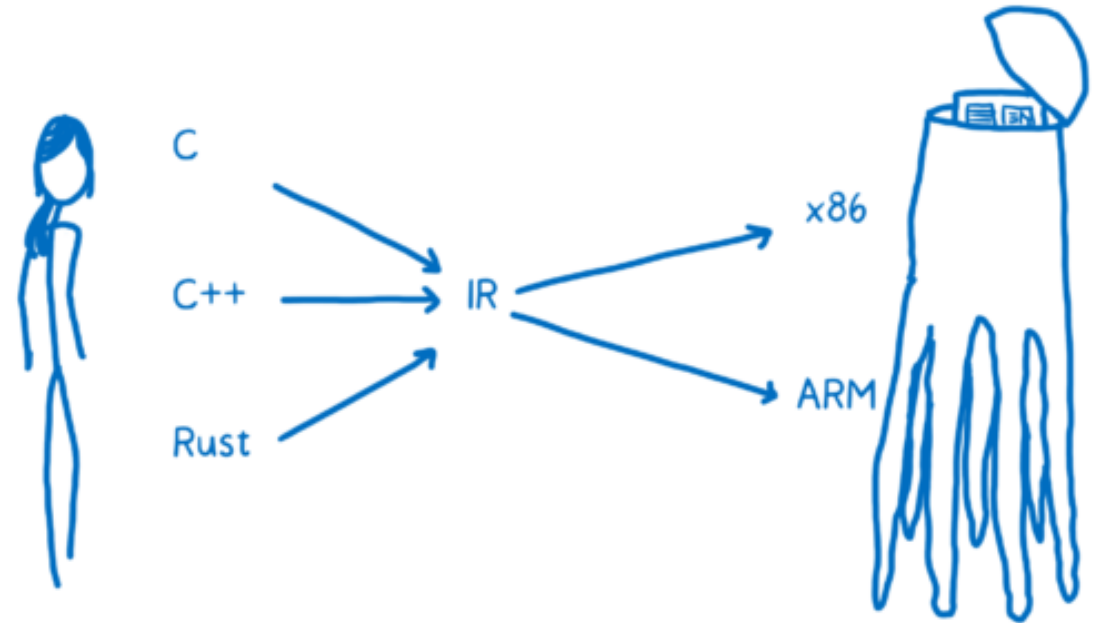


Basic Goals: Separation of Concerns

- Generate efficient code sequences for individual operations
- Keep it fast and simple: leave most optimizations to later phases
- Provide clean, easy-to-optimize code
- IR forms the basis for code optimization and target code generation

Intermediate language

- Goal: Translate AST to low-level machine-independent 3-address IR
- Two alternative ways:
 1. Bottom-up tree-walk on AST
 2. Syntax-Directed Translation



Three-Address Code (Quadraples)

- A quadraple is: $x := y \text{ op } z$
where x , y and z are names, constants or compiler-generated temporaries;
 op is any operator.
- But we may also use the following notation for quadraples (much better notation because it looks like a machine code instruction)
 $\text{op } x, y, z$
apply operator op to y and z , and store the result in x .
- We use the term “three-address code” because each statement usually contains three addresses (two for operands, one for the result).

Three-Address Statements

Binary Operator: `op result, y, z` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex: `add a, b, c`
 `addi a, b, c`
 `gt a, b, c`

Unary Operator: `op result, , y` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex: `uminus a, , c`
 `not a, , c`
 `inttoreal a, , c`

Three-Address Statements (cont.)

Move Operator: `mov result, , y` or `result := y`

where the content of `y` is copied into `result`.

Ex: `mov a, , c`
 `movi a, , c`
 `movr a, , c`

Unconditional Jumps: `jmp , , L` or `goto L`

We will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex: `jmp , , L1 // jump to L1`
 `jmp , , 7 // jump to the statement 7`

Three-Address Statements (cont.)

Conditional Jumps: `jmp`*relop* `y, z, L` or `if y` *relop* `z goto L`

We will jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex:

<code>jmpgt</code>	<code>y, z, L1</code>	// jump to L1 if <code>y > z</code>
<code>jmpge</code>	<code>y, z, L1</code>	// jump to L1 if <code>y >= z</code>
<code>jmpeq</code>	<code>y, z, L1</code>	// jump to L1 if <code>y == z</code>
<code>jmpne</code>	<code>y, z, L1</code>	// jump to L1 if <code>y != z</code>

Our relational operator can also be a unary operator.

<code>jmpnz</code>	<code>y, , L1</code>	// jump to L1 if <code>y</code> is not zero
<code>jmpz</code>	<code>y, , L1</code>	// jump to L1 if <code>y</code> is zero
<code>jmpt</code>	<code>y, , L1</code>	// jump to L1 if <code>y</code> is true
<code>jmpf</code>	<code>y, , L1</code>	// jump to L1 if <code>y</code> is false

Three-Address Statements (cont.)

Procedure Parameters: param $x, ,$ or param x

Procedure Calls: call $p, n,$ or call p, n

where x is an actual parameter, we invoke the procedure p with n parameters.

Ex: param $x_1, ,$
... $\rightarrow p(x_1, \dots, x_n)$

param $x_n, ,$

call $p, n,$

$f(x+1, y) \rightarrow$ add $t1, x, 1$

param $t1, ,$

param $y, ,$

call $f, 2,$

Three-Address Statements (cont.)

Indexed Assignments:

move x, , y[i] or x := y[i]

move y[i], , x or y[i] := x

Address and Pointer Assignments:

moveaddr x, , y or x := &y

movecont x, , y or x := *y

Declarations

- A symbol table entry is created for every declared name
- Information includes name, type, relative address of storage, etc.
- Relative address consists of an offset:
 - Offset is from the base of the static data area for global
 - Offset is from the field for local data in an activation record for locals to procedures
- Types are assigned attributes type and width (size)
- Becomes more complex if we need to deal with nested procedures or records

Declarations

$D \rightarrow T \text{ id} ; D \mid \varepsilon$

$T \rightarrow B C \mid \text{record } \{ D \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \varepsilon \mid [\text{num}] C$

SDT for Declarations

$D \rightarrow T \text{ id ; } D \mid \varepsilon$

$T \rightarrow B C \mid \text{record } \{ ' D ' \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \varepsilon \mid [\text{num}] C$

$P \rightarrow \{ \text{offset} = 0; \text{top} = \text{new ST}(); \} D$

$D \rightarrow T \text{ id ; } \{ \text{top.enter}(\text{id.name}, T.\text{type}, \text{offset}); \text{offset} = \text{offset} + T.\text{width}; \} D_1$

$D \rightarrow \varepsilon$

$T \rightarrow B \{ C.t = B.type ; C.w = B.width; \} C \{ T.type = C.type; T.width = C.width ; \}$

$B \rightarrow \text{int} \{ B.type = \text{integer}; B.width = 4; \}$

$B \rightarrow \text{float} \{ B.type = \text{float}; B.width = 8; \}$

$C \rightarrow \varepsilon \{ C.type = C.t ; C.width = C.w; \}$

$C \rightarrow [\text{num}] \{ C_1.t = C.t; C_1.w = C.w; \} C_1 \{ C.type = \text{array}(\text{num.val}, C_1.type); C.width = \text{num.val} * C_1.width; \}$

$T \rightarrow \text{record } \{ ' \{ \text{STStack.push}(\text{top}); \text{top} = \text{new ST}(\text{top}); \text{Stack.push}(\text{offset}); \text{offset} = 0 \}$

$D ' \} \{ T.type = \text{record}(\text{Top}); T.width = \text{offset}; \text{top.STStack.pop}(); \text{offset} = \text{Stack.pop}(); \}$

Review

- Three-address code
- Symbol table construction using SDT/SDD,
- Same to construct symbol table by AST traversal
 - ✓ Type, width, offset

Syntax-Directed Translation into Three-Address Code

- Temporary names are created for the interior nodes of a syntax tree
- The synthesized attribute **S.code** represents the code for the production S
- The nonterminal E has attributes:
 - **E.place** is the name that holds the value of E
 - **E.code** is a sequence of three-address statements evaluating E
- The function **newtemp()** returns a distinct name
- The function **newlabel()** returns a distinct label

Statements

$S \rightarrow \mathbf{id} := E$

$S \rightarrow \text{while } E \text{ do } S_1$

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

$S \rightarrow S_1 S_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \mathbf{id}$

Syntax-Directed Translation into Three-Address Code

```
S → id := E      { S.code = E.code || p = top.lookup(id.name);  
                  if p != NULL then gen('mov' p ',', E.place); else error; }  
  
E → E1 + E2   { E.place = newtemp();  
                  E.code = E1.code || E2.code || gen('add' E.place 'E1.place ' E2.place); }  
  
E → E1 * E2   { E.place = newtemp();  
                  E.code = E1.code || E2.code || gen('mult' E.place ' E1.place ' E2.place); }  
  
E → - E1      { E.place = newtemp();  
                  E.code = E1.code || gen('uminus' E.place ',', E1.place); }  
  
E → ( E1 )    { E.place = E1.place;  
                  E.code = E1.code; }  
  
E → id          { p = top.lookup(id.name);  
                  if p != NULL then E.place = id.place; else error;  
                  E.code = "" // null }
```

Syntax-Directed Definitions (cont.)

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

```
S.else = newlabel();
S.after = newlabel();
S.code = E.code ||
    gen('jmpf' E.place ',' S.else) || S1.code ||
    gen('jmp' ',' S.after) ||
    gen(S.else ':') || S2.code ||
    gen(S.after ':')
```

$S \rightarrow S_1 S_2$ $S_1.\text{code} || S_2.\text{code}$

Syntax-Directed Definitions (cont.)

$S \rightarrow \text{while } E \text{ do } S_1$

```
S.begin = newlabel();  
S.after = newlabel();  
S.code = gen(S.begin ":" ) || E.code ||  
           gen('jmpf' E.place ',' S.after) || S1.code ||  
           gen('jmp' ',' S.begin) ||  
           gen(S.after ':') }
```

Break and continue?

Syntax-Directed Definitions (cont.)

$S \rightarrow \text{while } E \text{ do } S_1$ $S_1.inbegin = \text{newlabel}(); S.begin = S_1.inbegin$
 $S_1.inafter = \text{newlabel}(); S.after = S_1.inafter$
 $S.code = \text{gen}(S.begin \text{ ":"}) \ || \ E.code \ ||$
 $\text{gen}(\text{'jmpf' } E.place \text{ ',' } S.after) \ || \ S_1.code \ ||$
 $\text{gen}(\text{'jmp' } \text{' ',' } S.begin) \ ||$
 $\text{gen}(S.after \text{ ':'}) \}$

$S \rightarrow S_1 S_2$ $S_1.inbegin=S_2.inbegin=S.inbegin;$
 $S_1.inafter=S_2.inafter=S.inafter;$
 $S_1.code \ || \ S_2.code$

$S \rightarrow \text{break}$ $\text{gen}(\text{'jmp' } S.inafter)$

$S \rightarrow \text{continue}$ $\text{gen}(\text{'jmp' } S.inbegin)$

Statements (cont.)

$D \rightarrow T \text{ id} ; D \mid \varepsilon$

$T \rightarrow B C \mid \text{record } \{ ' D ' \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \varepsilon \mid [\text{num}] C$

$S \rightarrow \mathbf{id} := E$

$S \rightarrow \text{while } E \text{ do } S_1$

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

$S \rightarrow S_1 S_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow \mathbf{id}$

$S \rightarrow \text{return } E$

$E \rightarrow \text{id (AP)}$

$AP \rightarrow \varepsilon \mid E, AP$

Function definitions and function calls

$D \rightarrow \text{fn } T \text{ id (FP) } \{ D ; S \}$

$FP \rightarrow \varepsilon \mid T \text{ id, FP}$

Syntax-Directed Translation (cont.)

$D \rightarrow \text{fn } T \text{ id}$

(FP) '{ begin=newlabel(); gen(begin' :');

{ **STStack.push(top); top =new ST(top); Stack.push(offset); offset=0** }

D {**top=STStack.pop(); offset=Stack.pop();** }

; S}' { top.enter(id.name,T.type, FP.types, begin)}

$FP \rightarrow \varepsilon \mid T \text{ id}$, FP construct a list of types from FP

$S \rightarrow \text{return } E ; \mid \text{others}$ // introduced in runtime organization

$E \rightarrow \text{id (AP) } \{p=\text{top.lookup(id.name)}; \text{AP.code} \mid \mid \text{gen('call' p,n)};\}$

$AP \rightarrow \varepsilon$

$AP \rightarrow E, AP_1 \{AP.code = E.code \mid \mid \text{gen('param' E.place)} \mid \mid AP_1.code\}$

$E \rightarrow \text{others}$ // as previous

Statements

$S \rightarrow \mathbf{id} := E$

$S \rightarrow L := E$

$S \rightarrow \text{while } E \text{ do } S$

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$E \rightarrow E * E$

$E \rightarrow E + E$

$E \rightarrow - E$

$E \rightarrow (E_1)$

$E \rightarrow \mathbf{id}$

$E \rightarrow L$

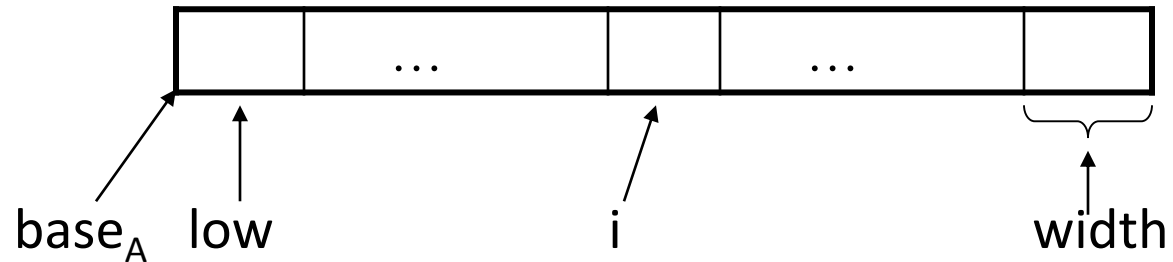
$L \rightarrow \mathbf{id} [E]$

$L \rightarrow L [E]$

Arrays

- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



base_A is the address of the first location of the array A,

width is the width of each array element.

low is the index of the first array element

location of A[i] → $\text{base}_A + (i - \text{low}) * \text{width}$

Arrays (cont.)

$\text{base}_A + (i - \text{low}) * \text{width}$

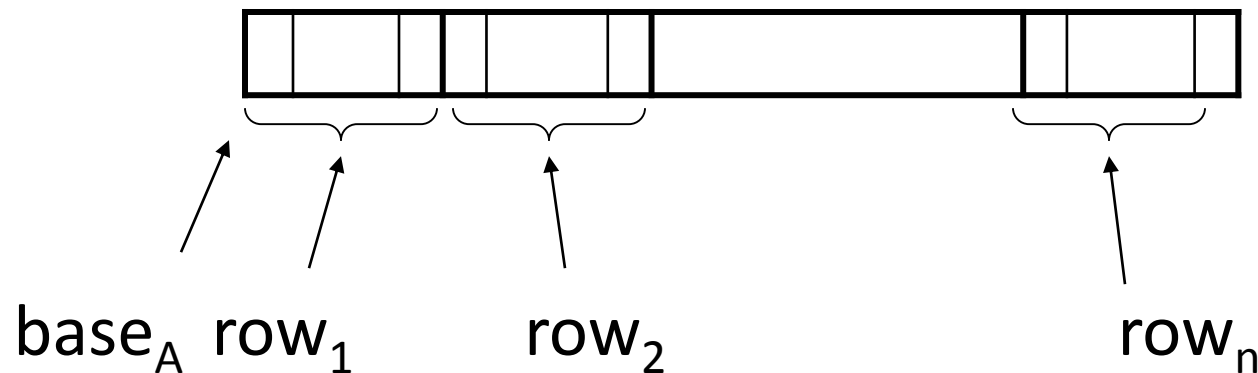
can be re-written as $i * \text{width} + (\text{base}_A - \text{low} * \text{width})$

should be computed at **run-time** can be computed at **compile-time**

- So, the location of $A[i]$ can be computed at the run-time by evaluating the formula $i * \text{width} + c$ where c is $(\text{base}_A - \text{low} * \text{width})$ which is evaluated at compile-time.
- Intermediate code generator should produce the code to evaluate this formula $i * \text{width} + c$ (one multiplication and one addition operation).

Two-Dimensional Arrays

- A two-dimensional array can be stored in
 - either **row-major** (*row-by-row*)
 - or **column-major** (*column-by-column*).
- Most of the programming languages use **row-major** method.
- Row-major representation of a two-dimensional array:



Two-Dimensional Arrays (cont.)

- The location of $A[i_1][i_2]$ is: $\text{base}_A + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * \text{width}$

base_A is the location of the array A.

low₁ is the index of the first row

low₂ is the index of the first column

n₂ is the number of elements in each row

width is the width of each array element

- Again, this formula can be re-written as

$$\underbrace{((i_1 * n_2) + i_2) * \text{width}}_{\text{run-time}} + \underbrace{(\text{base}_A - ((\text{low}_1 * n_1) + \text{low}_2) * \text{width})}_{\text{compile-time}}$$

should be computed at **run-time**

can be computed at **compile-time**

Multi-Dimensional Arrays

- In general, the location of $A[i_1][i_2] \dots [i_k]$ is

$$((\dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k) * \text{width} + (\text{base}_A - ((\dots((\text{low}_1 * n_1) + \text{low}_2) \dots) * n_k + \text{low}_k) * \text{width})$$

- So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of $A[i_1][i_2] \dots [i_k]$) :

$$((\dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k) * \text{width} + c$$

- To evaluate the $((\dots ((i_1 * n_2) + i_2) \dots) * n_k + i_k) * \text{width}$ portion of this formula, we can compute

$$i_1 * \text{width} * n_2 * \dots * n_k + \dots + i_j * \text{width} * n_{j+1} * \dots * n_k + \dots + i_k * \text{width}$$

Syntax-Directed Translation into Three-Address Code

```
S → id := E    { S.code = E.code || p = top.lookup(id.name);  
                  if p != NULL then gen('mov' p ',', E.place); else error ; }  
  
S → L := E     { S.code = E.code || gen('mov' L.array.base '[' L.place ']', , E.place); }  
  
E → E1 * E2 { E.place = newtemp();  
                  E.code = E1.code || E2.code || gen('mult' E.place ' ' E1.place ' ' E2.place); }  
  
E → L          { E.place = newtemp(); gen('mov' E.place, , L.array.base '[' L.place ''] ); }  
  
L → id [ E ]   { L.code = E.code || L.array = top.lookup(id.name); L.type = L.array.type.elem;  
                  L.place = newtemp(); gen('mult' L.place, E.place, L.type.width) ; }  
  
L → L1 [ E ]  { L.code = E.code || L.array = L1.array; L.type = L.array.type.elem;  
                  L.place = newtemp(); t= newtemp();  
                  gen('mult' t, E.place, L.type.width);  
                  gen('add' L.place, L1.place, t); }
```

Boolean Expressions

$E \rightarrow E_1 \text{ and } E_2$

```
{ E.code = E1.code || E2.code || E.place = newtemp(); gen('and' E.place ' ' E1.place ' ' E2.place; }
```

$E \rightarrow E_1 \text{ or } E_2$

```
{ E.code = E1.code || E2.code || E.place = newtemp(); gen('or' E.place ' ' E1.place ' ' E2.place }
```

$E \rightarrow \text{not } E_1$

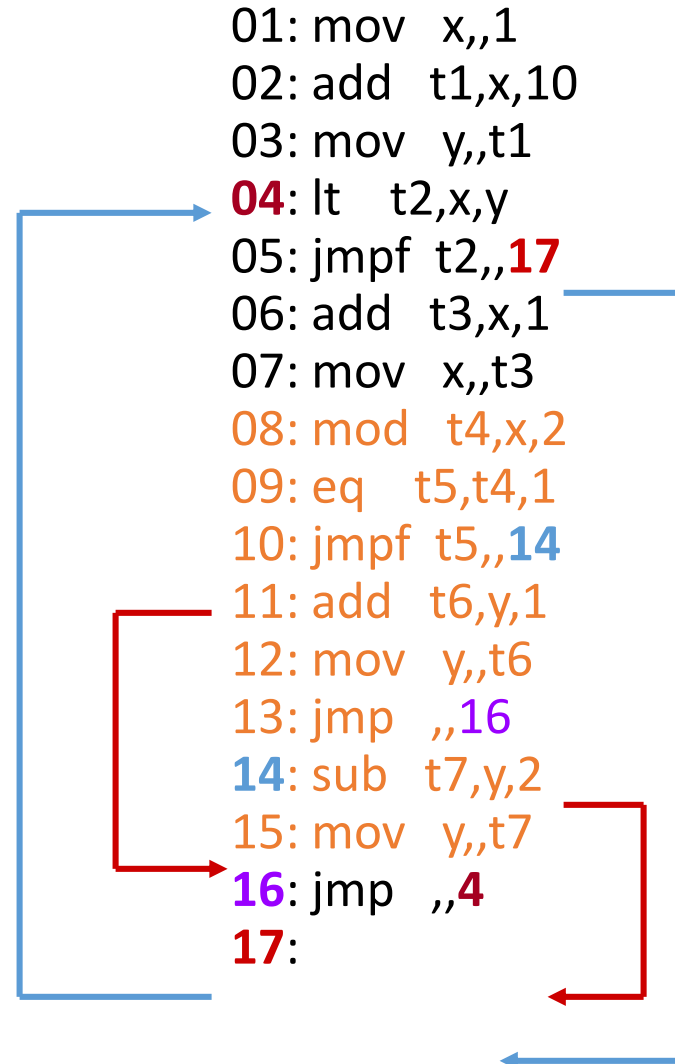
```
{ E.code = E1.code || E.place = newtemp(); gen('not' E.place ' ' E1.place) }
```

$E \rightarrow E_1 \text{ **relop** } E_2$

```
{ E.code = E1.code || E2.code || E.place = newtemp(); gen(relop.code E.place ' ' E1.place ' ' E2.place) }
```

Three Address Codes - Example

```
x:=1;  
y:=x+10;  
while (x<y) {  
  x:=x+1;  
  if (x%2==1) then y:=y+1; →  
  else y:=y-2;  
}
```



Classes

- Each class is regarded as a record
- All the **non-static attributes** are **fields** of the record
- All the **static attributes** are regarded as **global variables/functions**

```
Class C:{  
  int x;  
  fn T f(FP){  
    ...  
  }  
}  
f(AP)  
c.f(AP)  
o.x  
x
```

```
Record C {  
  int x;  
  fn T f'(C& this, FP){  
    ...  
  }  
}  
f'(this, AP)  
f'(c,AP)  
o.x  
this.x
```

Inheritance

- How to handle methods may inherited from this parent classes?
- Naive approach: each class has its own Implementation?
- **Better approach:**
 - For each class, construct a **method table** including all the functions (**pointers to entry points of functions**) defined in this class as well as functions inherited from this parent classes
 - **method table:**
 1. Copy inherited methods
 2. Overwrite overridden methods
 3. Append its own methods
 - The record of the class includes all the data attributes defined in this class as well as inherited data attributes, in addition with **a pointer to this method table**

Exercises

- Translating the following C codes into three address codes.

```
int fun_for()
{
    int i,s=0;
    int a[10]={0,1,2,3,4,5,6,7,8,9};
    for (i=0;i<10;i++)
        s=s+a[i];
    return s;
}
```

```
int fun_if()
{
    int i=10,j=12;
    if(i<j)
        j=j-i;
    else
        i=i-j;
    return i+j;
}
```


Quiz

Record {int x; float[3] y} z

$P \rightarrow \{\text{offset} = 0; \text{top} = \text{new ST}();\} D$

$D \rightarrow T \text{ id} ; \{\text{top.enter}(\text{id.name}, T.\text{type}, \text{offset}); \text{offset} = \text{offset} + T.\text{width};\} D_1$

$D \rightarrow \varepsilon$

$T \rightarrow B \{C.t = B.type ; C.w = B.width; \} C \{T.type = C.type; T.width = C.width ;\}$

$B \rightarrow \text{int} \{ B.type = \text{integer}; B.width = 4; \}$

$B \rightarrow \text{float} \{ B.type = \text{float}; B.width = 8; \}$

$C \rightarrow \varepsilon \{ C.type = C.t ; C.width = C.w; \}$

$C \rightarrow [\text{num}] \{C_1.t = C.t; C_1.w = C.w; \} C_1 \{ C.type = \text{array}(\text{num.val}, C_1.type); C.width = \text{num.val} * C_1.width; \}$

$T \rightarrow \text{record} \{ \{ \text{STStack.push}(\text{top}); \text{top} = \text{new ST}(\text{top}); \text{Stack.push}(\text{offset}); \text{offset} = 0 \}$

$D \{ \} \{ T.type = \text{record}(\text{Top}); T.width = \text{offset}; \text{top.STStack.pop}(); \text{offset} = \text{Stack.pop}(); \}$

$D \rightarrow T \text{ id} ; D \mid \varepsilon$

$T \rightarrow B C \mid \text{record} \{ \} D \{ \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \varepsilon \mid [\text{num}] C$