Intermediate Code Generation
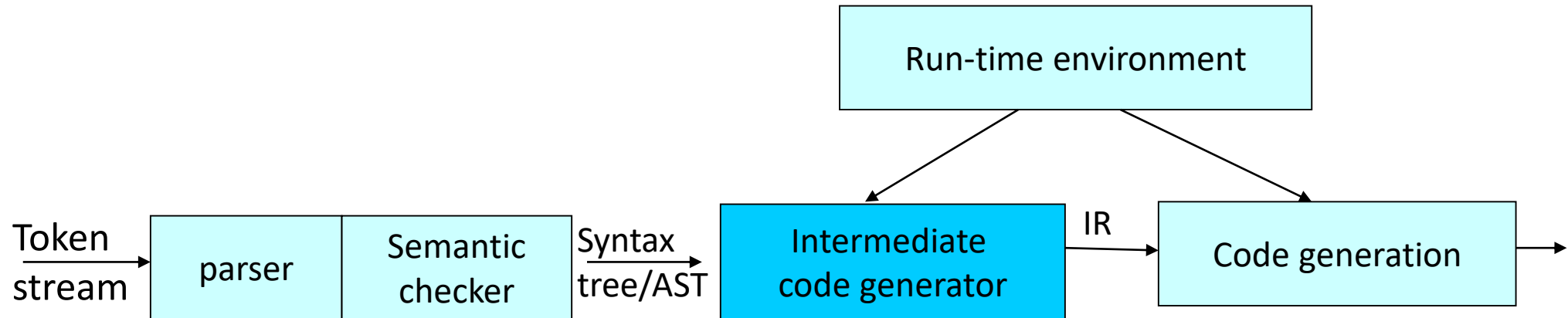
- *Intermediate codes* are machine independent codes, but they are close to machine instructions

- The given program in a source language is converted to an **equivalent program** in an intermediate language by the intermediate code generator

```
                                    ┌─────────────────────────┐
                                    │   Run-time environment   │
                                    └─────────────────────────┘
                                         ↙            ↘
Token        ┌────────┬──────────┐ Syntax  ┌──────────────┐  IR  ┌──────────────┐
─────→       │ parser │ Semantic │ ───────→│ Intermediate │ ────→│ Code generation │ ──→
stream       │        │ checker  │ tree/AST│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
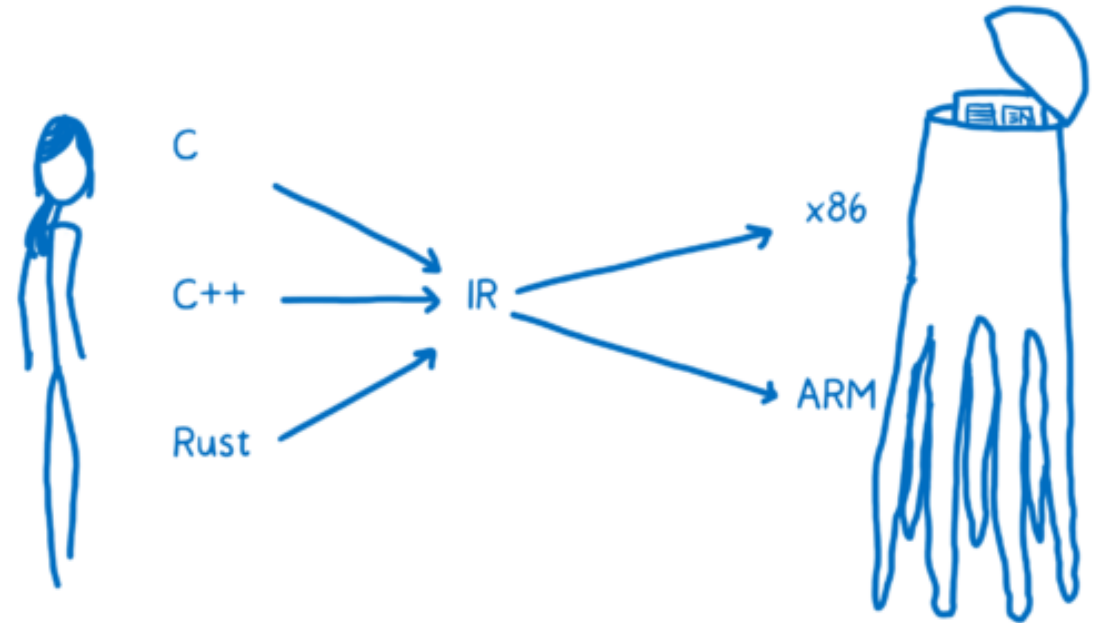
Three-Address Code (Quadraples)

- A quadraple is: `x := y 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)

      `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: `jmprelop 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:
```
jmpgt   y,z,L1    // jump to L1 if y>z
jmpge   y,z,L1    // jump to L1 if y>=z
jmpeq   y,z,L1    // jump to L1 if y==z
jmpne   y,z,L1    // jump to L1 if y!=z
```

Our relational operator can also be a unary operator.
```
jmpnz   y,,L1     // jump to L1 if y is not zero
jmpz    y,,L1     // jump to L1 if y is zero
jmpt    y,,L1     // jump to L1 if y is true
jmpf    y,,L1     // jump to L1 if y 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`$_1$`,,`

                      ➔ `p(x`$_1$`,...,x`$_n$`)`

           `param x`$_n$`,,`

           `call  p,n,`

`f(x+1,y)` ➔   `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
```

Bottom-up tree-walk on AST

```
expr( node )
    int result, t1, t2, t3;
    switch( type of node )
    {
    case TIMES:
        t1 = expr( left child of node );
        t2 = expr( right child of node );
        result = new_name();
        emit( mov, result, t1, t2 );
        break;
    case PLUS:
        t1 = expr( left child of node );
        t2 = expr( right child of node );
        result = new_name();
        emit( add, result, t1, t2);
        break;

    case ID:
        result = id.place;
        emit("")
        break;
    case NUM:
        result = node.val
        emit("")
        break;
    }
    return result;
```

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→ T id ; D | ε

T→ B C | record '{' D '}'

B→ int | float

C→ ε | [num] C

How to compute types via SDT?

SDT for Declarations

D→ T id ; D | ε

T→ B C | record '{' D '}'

B→ int | float

C→ ε | [num] C

P→ {offset = 0; top =new ST();} D

D→ T id ; {top.enter(id.name, T.type, offset); offset = offset + T.width;} D_1

D→ ε

T→ B {C.t = B.type ; C.w = B.width; } C {T.type = C.type; T.width = C.width ;}

B→ int { B.type = integer; B.width = 4; }

B→ float { B.type = float; B.width = 8; }

C→ ε { C.type = C.t ; C.width = C.w;   }

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

T→ record '{' { STStack.push(top); top =new ST(top); Stack.push(offset); offset=0 }

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

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

$S \rightarrow$ **id** := E

$S \rightarrow$ while E do S_1

$S \rightarrow$ if E then S_1 else S_2

$S \rightarrow S_1 S_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow$ **id**

Syntax-Directed Translation into Three-Address Code

$S \rightarrow$ **id** := E   { S.code = E.code || p = top.lookup(id.name);

                       if p != NULL then gen('mov' p ',' E.place); else error ;}

$E \rightarrow E_1 + E_2$ { E.place = newtemp();

         E.code = E_1.code || E_2.code || gen('add' E.place ',' E_1.place ',' E_2.place) ; }

$E \rightarrow E_1 * E_2$ { E.place = newtemp();

         E.code = E_1.code || E_2.code || gen('mult' E.place ',' E_1.place ',' E_2.place); }

$E \rightarrow - E_1$ { E.place = newtemp();

         E.code = E_1.code || gen('uminus' E.place ',' E_1.place); }

$E \rightarrow (E_1)$ { E.place = E_1.place;

         E.code = E_1.code; }

$E \rightarrow$ **id**   { p = top.lookup(id.name);

       if p != NULL then E.place = **id**.place; else error;

       E.code = "" // null }

Syntax-Directed Definitions (cont.)

S → if E then S₁ else S₂   S.else = newlabel();

$\qquad\qquad\qquad\qquad\qquad\qquad$ S.after = newlabel();

$\qquad\qquad\qquad\qquad\qquad\qquad$ S.code = E.code ||

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ gen('jmpf' E.place ',' S.else) || S₁.code ||

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ gen('jmp' ',' S.after) ||

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ gen(S.else ':") || S₂.code ||

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ gen(S.after ':")

S → S₁ S₂ $\qquad$ S₁.code || S₂.code

Syntax-Directed Definitions (cont.)

S → while E do S$_1$ S.begin = newlabel();

S.after = newlabel();

S.code = gen(S.begin ":") || E.code ||

gen('jmpf' E.place ',' S.after) || S$_1$.code ||

gen('jmp' ',' S.begin) ||

gen(S.after ':") }

Break and continue?

Syntax-Directed Definitions (cont.)

S $\rightarrow$ while E do S_1 $\qquad$ S_1.inbegin = newlabel(); S.begin = S_1.inbegin

$\qquad\qquad\qquad\qquad\qquad$ S_1.inafter = newlabel(); S.after = S_1.inafter

$\qquad\qquad\qquad\qquad\qquad$ S.code = gen(S.begin ":") || E.code ||

$\qquad\qquad\qquad\qquad\qquad\qquad$ gen('jmpf' E.place ',' S.after) || S_1.code ||

$\qquad\qquad\qquad\qquad\qquad\qquad$ gen('jmp' ',' S.begin) ||

$\qquad\qquad\qquad\qquad\qquad\qquad$ gen(S.after ":") }

S $\rightarrow$ S_1 S_2 $\qquad\qquad$ S_1.inbegin=S_2.inbegin=S.inbegin;

$\qquad\qquad\qquad\qquad$ S_1.inafter=S_2.inafter=S.inafter;

$\qquad\qquad\qquad\qquad$ S_1.code || S_2.code

S $\rightarrow$ break $\qquad\qquad$ gen('jmp' S.inafter)

S $\rightarrow$ continue $\qquad$ gen('jmp' S.inbegin)

Statements (cont.)

$D \rightarrow$ T id ; D | ε

$T \rightarrow$ B C | record '{' D '}'

$B \rightarrow$ int | float

$C \rightarrow$ ε | [num] C

$S \rightarrow$ **id** := E

$S \rightarrow$ while E do S_1

$S \rightarrow$ if E then S_1 else S_2

$S \rightarrow S_1 S_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow E_1 + E_2$

$E \rightarrow - E_1$

$E \rightarrow (E_1)$

$E \rightarrow$ **id**

$S \rightarrow$ return E

$E \rightarrow$ id (AP)

$AP \rightarrow \varepsilon$ | E, AP

Function definitions and function calls

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

$FP \rightarrow \varepsilon$ | T id, FP

20

Syntax-Directed Translation (cont.)

D → fn T id

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

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

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

         top.enter(id.name,T.type, FP.types, begin)}

FP → ε | T id, FP   construct a list of types from FP

S → return E   // introduced in runtime organization

E → id (AP)   {p=top.lookup(id.name); AP.code||gen('call' p,n);}

AP → ε

AP → E, AP_1 {AP.code = E.code|| gen('param' E.place)|| AP_1.code}

Statements

S → **id** := E

S → while E do S

S → if E then S else S

E → E * E

E → E + E

E → - E

E → (E_1)

E → **id**

S → L := E

E → L
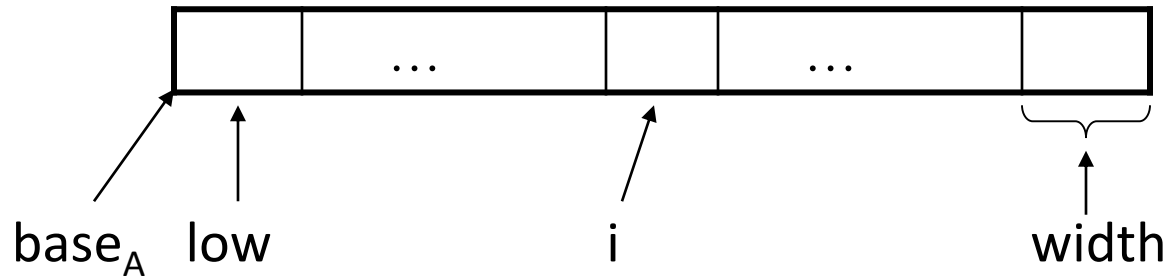
L → id [E]

L → 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] ➔ $base_A+(i-low)*width$

Arrays (cont.)

`base`$_A$`+(i-low)*width`

can be re-written as   `i*width` + (`base`$_A$`-low*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*width + c where c is (base$_A$-low*width) which is evaluated at compile-time.

- Intermediate code generator should produce the code to evaluate this formula i*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:

$base_A$ row_1 row_2 row_n
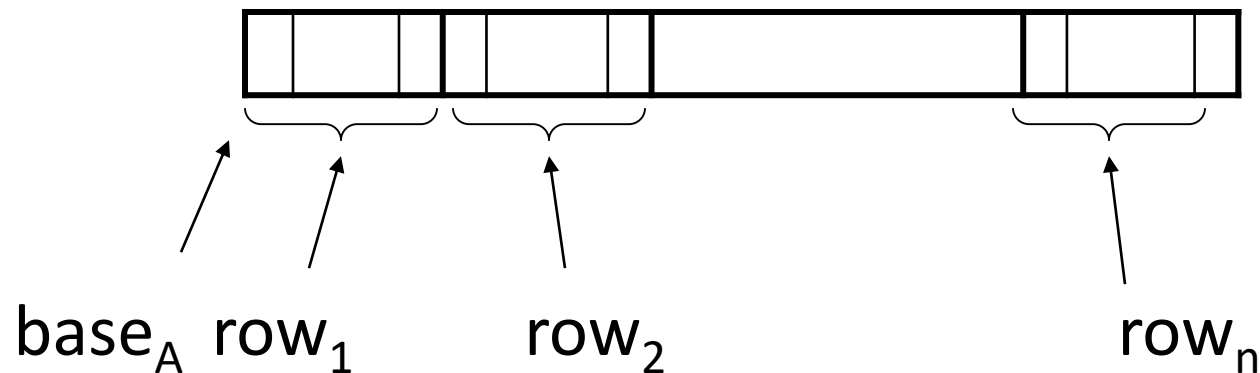
Two-Dimensional Arrays (cont.)

- The location of $A[i_1][i_2]$ is: $\quad$ $base_A + ((i_1 - low_1) * n_2 + i_2 - low_2) * width$

    $base_A$ is the location of the array A.
    low_1 is the index of the first row
    low_2 is the index of the first column
    n_2 is the number of elements in each row
    width is the width of each array element

- Again, this formula can be re-written as

$$\underbrace{((i_1 * n_2) + i_2) * width}\ + \ \underbrace{(base_A - ((low_1 * n_2) + low_2) * width)}$$

should be computed at run-time $\qquad$ can be computed at compile-time

Multi-Dimensional Arrays

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

    $((... ((i_1*n_2)+i_2) ...)*n_k+i_k)*width + (base_A-((...((low_1*n_2)+low_2)...)*n_k+low_k)*width)$

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

    $((... ((i_1*n_2)+i_2) ...)*n_k+i_k)*width + c$

- To evaluate the $((... ((i_1*n_2)+i_2) ...)*n_k+i_k) *width$ portion of this formula, we can compute

    $i_1 * width * n_2*...*n_k + ...+ i_{j\,*} width * n_{j+1}*...*n_k + ...+ i_k*width$

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 ; }

E → E_1 * E_2 { E.place = newtemp();

               E.code = E_1.code || E_2.code || gen('mult' E.place ',' E_1.place ',' E_2.place); }

S → L := E   { S.code =E.code || gen('mov' L.array.base '[' L.place ']', , E.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 → L_1 [E]   { L.code = E.code || L.array = L_1.array; L.type = L.array.type.elem;

                          L.place = newtemp(); t= newtemp();

                          gen('mult' t, E.place, L.type.width);

                          gen('add' L.place, L_1.place, t); }

Boolean Expressions

E $\rightarrow$ E$_1$ and E$_2$

  { E.code =E$_1$.code||E$_2$.code||E.place = newtemp(); gen('and' E.place ',' E$_1$.place ',' E$_2$.place; }

E $\rightarrow$ E$_1$ or E$_2$

  {E.code =E$_1$.code ||E$_2$.code|| E.place = newtemp(); gen('or' E.place ',' E$_1$.place ',' E$_2$.place}

E $\rightarrow$ not E$_1$

  {E.code =E$_1$.code || E.place = newtemp(); gen('not' E.place ',,' E$_1$.place) }

E $\rightarrow$ E$_1$ **relop** E$_2$

{E.code =E$_1$.code ||E$_2$.code|| E.place = newtemp(); gen(**relop**.code E.place ',' E$_1$.place ','
  E$_2$.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;
}
```

```
01: mov   x,,1
02: add   t1,x,10
03: mov   y,,t1
04: lt    t2,x,y
05: jmpf  t2,,17
06: add   t3,x,1
07: mov   x,,t3
08: mod   t4,x,2
09: eq    t5,t4,1
10: jmpf  t5,,14
11: add   t6,y,1
12: mov   y,,t6
13: jmp   ,,16
14: sub   t7,y,2
15: mov   y,,t7
16: jmp   ,,4
17:
```

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 {                Record C {
  int x;                   int x;
  fn T f(FP){              fn T f'(C& this, FP){

  …                        …
  }                        }
}                        }
f(AP)                    f'(this, AP)
c.f(AP)                  f'(c,AP)
o.x                      o.x
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

Exercise

$D \rightarrow T$ id ; $D \mid \varepsilon$

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

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

$B \rightarrow$ int | float

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

$C \rightarrow \varepsilon \mid$ [num] C

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

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

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

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

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

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

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