Type Checking

- A compiler must do semantic checks in addition to syntactic checks.
- *Type checking* is one of these static checking operations.
  - we may not do all type checking at compile-time.
  - Some systems also use dynamic type checking too.
- A programming language is *strongly-typed*, if every program if compiler accepts, will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
  - Ex: int x[100]; … x[i] ➔ most of the compilers cannot guarantee that i will be between 0 and 99

A Simple Type Checking System

$P \rightarrow D;E$

$D \rightarrow D;D$

$D \rightarrow \textbf{id}:T$ { addtype(id.entry,T.type) }

$T \rightarrow \text{char}$ { T.type=char }

$T \rightarrow \text{int}$ { T.type=int }

$T \rightarrow \text{real}$ { T.type=real }

$T \rightarrow \uparrow T_1$ { T.type=pointer(T_1.type) }

$T \rightarrow \text{array}[\textbf{intnum}] \text{ of } T_1$ { T.type=array(1..intnum.val,T_1.type) }

Type Checking of Expressions

$E \rightarrow$ **id** { E.type=lookup(id.entry) }

$E \rightarrow$ **charliteral** { E.type=char }

$E \rightarrow$ **intliteral** { E.type=int }

$E \rightarrow$ **realliteral** { E.type=real }

$E \rightarrow E_1 + E_2$ { if (E_1.type=int and E_2.type=int) then E.type=int
else if (E_1.type=int and E_2.type=real) then E.type=real
else if (E_1.type=real and E_2.type=int) then E.type=real
else if (E_1.type=real and E_2.type=real) then E.type=real
else E.type=type-error }

$E \rightarrow E_1\ [E_2]$ { if (E_2.type=int and E_1.type=array(s,t)) then E.type=t
else E.type=type-error }

$E \rightarrow E_1 \uparrow$ { if (E_1.type=pointer(t)) then E.type=t
else E.type=type-error }

Type Checking of Statements

$S \rightarrow$ **id** = E	{ if (id.type=E.type then S.type=void else S.type=type-error }
$S \rightarrow$ if E then S_1	{ if (E.type=boolean then S.type=S_1.type else S.type=type-error }
$S \rightarrow$ while E do S_1	{ if (E.type=boolean then S.type=S_1.type else S.type=type-error }

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.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

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 the following notation for quadraples (much better notation because it looks like a machine code instruction)

  `op  y,z,x`

  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 y,z,result` 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
gt   a,b,c
addi a,b,c
```

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

```
jmpgt    y,z,L1      // jump to L1 if y>z
jmpgte   y,z,L1      // jump to L1 if y>=z
jmpe     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 x1,,
param x2,,
                    → p(x1,...,xn)
param xn,,
call  p,n,
```

```
f(x+1,y)  →    add   x,1,t1
               param t1,,
               param y,,
               call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

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

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

Address and Pointer Assignments:

`moveaddr y,,x` or `x := &y`

`movecont y,,x` or `x := *y`

Syntax-Directed Translation into Three-Address Code

$S \rightarrow$ **id** := E	S.code = E.code || gen('mov' E.place ',,' id.place)
$E \rightarrow E_1 + E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('add' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow E_1 * E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('mult' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow - E_1$	E.place = newtemp();
	E.code = E_1.code || gen('uminus' E_1.place ',,' E.place)
$E \rightarrow (E_1)$	E.place = E_1.place;
	E.code = E_1.code
$E \rightarrow$ **id**	E.place = **id**.place;
	E.code = null

Syntax-Directed Translation (cont.)

$S \rightarrow$ 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 ':")

$S \rightarrow$ if E then S_1 else S_2

S.else = newlabel();
S.after = newlabel();
S.code = E.code ||
gen('jmpf' E.place ',,' S.else) || S_1.code ||
gen('jmp' ',,' S.after) ||
gen(S.else ':") || S_2.code ||
gen(S.after ':")

Translation Scheme to Produce Three-Address Code

$S \rightarrow$ **id** := E { p = lookup(id.name);
if (p is not nil) then emit('mov' E.place ',,' p)
else error("undefined-variable") }

$E \rightarrow E_1 + E_2$ { E.place = newtemp();
emit('add' E_1.place ',' E_2.place ',' E.place) }

$E \rightarrow E_1 * E_2$ { E.place = newtemp();
emit('mult' E_1.place ',' E_2.place ',' E.place) }

$E \rightarrow - E_1$ { E.place = newtemp();
emit('uminus' E_1.place ',,' E.place) }

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

$E \rightarrow$ **id** { p = lookup(id.name);
if (p is not nil) then E.place = **id**.place
else error("undefined-variable") }

a = b+ -c

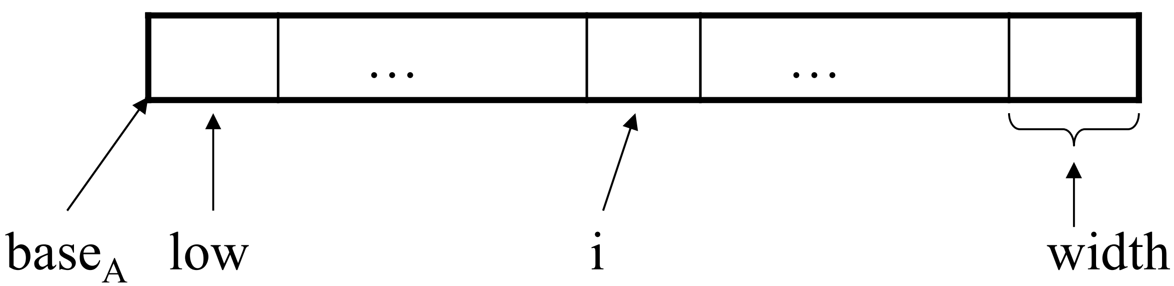
ICG will generate

uminus c,, t1
add b, t1, t2
mov t2,,a

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.)

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

can be re-written as $\underbrace{\texttt{i*width}}_{\text{should be computed at run-time}} + \underbrace{(\texttt{base}_A\texttt{-low*width})}_{\text{can be computed at compile-time}}$

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).

Translation Scheme for Arrays

- If we use the following grammar to calculate addresses of array elements, we need inherited attributes.

$$L \rightarrow \textbf{id} \quad | \quad \textbf{id} \; [\; Elist \;]$$

$$Elist \rightarrow Elist \; , \; E \quad | \quad E$$

- Instead of this grammar, we will use the following grammar to calculate addresses of array elements so that we do not need inherited attributes (we will use only synthesized attributes).

$$L \rightarrow \textbf{id} \quad | \quad Elist \;]$$

$$Elist \rightarrow Elist \; , \; E \quad | \; \textbf{id} \; [\; E$$

Translation Scheme for Arrays (cont.)

$S \rightarrow L := E$ { if (L.offset is null) emit('mov' E.place ',,' L.place)
else emit('mov' E.place ',,' L.place '[' L.offset ']') }

$E \rightarrow E_1 + E_2$ { E.place = newtemp();
emit('add' E_1.place ',' E_2.place ',' E.place) }

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

$E \rightarrow L$ { if (L.offset is null) E.place = L.place)
else { E.place = newtemp();
emit('mov' L.place '[' L.offset ']' ',,' E.place) } }

Translation Scheme for Arrays (cont.)

$L \rightarrow$ **id** { L.place = **id**.place; L.offset = null; }

$L \rightarrow$ Elist]

{ L.place = newtemp(); L.offset = newtemp();

emit('mov' c(Elist.array) ',,' L.place);

emit('mult' Elist.place ',' width(Elist.array) ',' L.offset) }

Elist $\rightarrow$ Elist_1 , E

{ Elist.array = Elist_1.array ; Elist.place = newtemp(); Elist.ndim = Elist_1.ndim + 1;

emit('mult' Elist_1.place ',' limit(Elist.array,Elist.ndim) ',' Elist.place);

emit('add' Elist.place ',' E.place ',' Elist.place); }

Elist $\rightarrow$ **id** [E

{Elist.array = id.place ; Elist.place = E.place; Elist.ndim = 1; }

Translation Scheme for Arrays – Example1

- A one-dimensional `double` array `A : 5..100`

  ➔ n_1=95 width=8 (double) low_1=5

- Intermediate codes corresponding to `x := A[y]`

```
mov    c,,t1              // where c=baseA-(5)*8
mult   y,8,t2
mov    t1[t2],,t3
mov    t3,,x
```

Declarations

$P \rightarrow M\ D$

$M \rightarrow €$ { offset=0 }

$D \rightarrow D\ ;\ D$

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

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

$T \rightarrow$ real { T.type=real; T.width=8 }

$T \rightarrow$ array[num] of T_1 { T.type=array(num.val,T_1.type);
T.width=num.val*T_1.width }

$T \rightarrow \uparrow T_1$ { T.type=pointer(T_1.type); T.width=4 }

where *enter* creates a symbol table entry with given values.