6.4 Translation of Expressions

- Study about Translation of expression into Three Address Code.

6.4.1 Operations within Expressions

- Little about Syntax-Directed Deffinition/Translation

> Grammar + Semantic Rules = Syntax-Directed Translation

- Here, every variable/Non-Terminal have attribute associated with it. Here, we have 2 attributes associated with two Non Terminals E and S, that are code and addres.

- Attributes S.code & E.code denote Three Address Code for S and E
- Attr. E.addr denotes address that will hold value of E

So, we have given a TAC for expression :-

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$;	S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr + E_2.addr)
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)

(E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$
id	$E.addr = top.get(id.lexeme)$ $E.code = ''$

Now, lets explain the above table ...

1. Conside $E \to id$. Semantic rules for this instance production define E.addr to point to symbol-table entry for this id

2. In, $E \to E_1$, translation of E is same as of subexpression E_1. Hence, E.addr equals $E_1.addr$ and E.code = $E_1.code$.

3. Operator '+' and '-' works similarly as operators in any language. E_1 is computed to $E_1.addr$ and E_2 to $E_2.addr$, then $E_1 + E_2$ to $t = E_1.addr + E_2.addr$ (t is temp variable) and E.addr is set to t.

Here, gen builds an instruction & returns it.

In prod. $E \to E_1 + E_2$, $E_1.code$ and $E_2.code$ concatenates, followed by instr. adding the values of E_1 and E_2, and at last put into E.addr.

4. $E \to -E_1$ is similar. Creation of new temporary for E and generate an instruction to perform unary minus.

5. $S \to id = E$; generates instruction that assign value of E to the id. Semantic Rule uses top.get to determine address of identifier represented by id.
S.code contains instructions to compute E into address given by E.addr, followed by assignment to address top.get(id.lexeme).

Eg :- SDT translates the statement $a = b - c$, into Three Address Code as :-

$t_1 = \text{minus } c$

$t_2 = b + t_1$

$a = t_2$

6.4.2 Incremental Translation

- Issue with above discussion, is that code attributes can be long strings, so they are usually generated incrementally.

- In this approach, gen not only constructs a Three-Address Instruction, it append the instruction to the seq. of instructions generated so far

- With Incremental approach code attr. is not used, since there is a single seq. of instructions created by successive calls to gen. Shown Below :-

$S \rightarrow id = E;$ { gen (top.get (id.lexeme) '=' E.addr) }

$E \rightarrow E_1 + E_2$ { E.addr = new Temp (); gen(E.addr '=' E_1.addr '+' E_2.addr) }

$| -E_1$ { E.addr = new Temp (); gen(E.addr '=' 'minus' E_1.addr) }

$| (E_1)$ { E_1.addr = E_1.addr }

| id { E.addr = top.get(id.lexeme); }

Page No.
Date / /

6.4.3 Addressing Array Elements :-

- Array elements can be accessed quickly if they are stored in a block of consecutive locations.
- To find the location of i^{th} element in the array
  $base + i \times w$ is used where (1-D array)
  base = relative address of the storage allocated for the array
  w = width of each array element
- For k dimensions, the formula is
  $$base + i_1 \times w_1 + i_2 \times w_2 \ldots\ldots + i_k \times w_k$$
  where for w_j, $1 \le j \le k$
- The relative array of an array reference can also be calculated in terms of the number of elements n_j along with dimension j of the array and the width $w = w_k$ of a single element of the array.
- In k-dimensions, the formula is
  $$base + ((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w$$
- More generally, array elements need not to be numbered starting at 0 as in 1-D array, the elements are numbered low, low+1 ... high and base = relative address of A[low].
- For 1-D array
  $$base + (i - low) \times w$$
  can also be written as $i \times w + c$
  where $c = base - low \times w$
- Compile-time precalculations can also be applied to address calculations for elements of multidimensional arrays but it is not applicable when the array's size is dynamic.
- A 2-D array is normally stored in one of the two forms, either row-major (row-by-row) or

column-major (column-by-column). The above address calculations are based on row-major layout.

- The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest.
- Column-major form generalizes to the opposite arrangement, with the left most subscripts varying fastest.

6.4.4 Translation of Array References :-

- The chief problem in generating code for array references is to relate the address calculation formulas to a grammar for array references.
- Let nonterminal L generates an array name followed by a sequence of index expressions

$$L \rightarrow L[E] \mid id[E]$$

- The below translation scheme generates three address code for expressions with array references.

$S \rightarrow id = E ; \{ gen(top.get(id.lexeme) '=' E.addr); \}$
$\mid L = E ; \{ gen(L.array.base '[' L.addr ']' '=' E.addr); \}$

$E \rightarrow E_1 + E_2 \{ E.addr = new\ Temp(); gen(E.addr '=' E_1.addr '+' E_2.addr); \}$
$\mid id \{ E.addr = top.get(id.lexeme); \}$
$\mid L \{ E.addr = new\ Temp(); gen(E.addr '=' L.array.base '[' L.addr ']'); \}$

```
L → id [E] { L.array = L1.array;
             L.type = L1.type.element;
             t = new Temp();
             L.addr = new Temp();
             gen(t '=' E.addr '*' L.type.width);
             gen(L.addr '=' L1.addr '+' t); }
```

Semantic actions for array references.

- L.addr denotes a temporary that is used while computing the offset for the array reference.
- L.array is a pointer to the symbol-table entry for the array name.
- L.array.base is used to determine the actual l-value of an array reference after all the index expressions are analyzed.
- L.type is the type of the subarray generated by L.
- S → id = E; represents an assignment to a nonarray variable, which is handled as usual.
- S → L = E; generates an indexed copy instruction to assign the value denoted by expression E to the location denoted by the array reference L.
- E → L generates code to copy the value from the location denoted by L into a new temporary.
- The code for the array reference places the r-value at the location designed by the base and offset into a new temporary denoted by E.addr.

6.5 Type Checking.

To do type checking, compiler needs to assign a type expression to each component of the source program. It must then determine that these expressions conform to a collection of logical rules that is called 'the type system' for the source language.

A sound type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.

Rules For Type Checking.

Type checking can take on two forms- synthesis & inference.

Synthesis:- Type synthesis builds up the type of an expression from the types of its subexpressions.

It requires names to be declared before they are used. for eg:- The type of $E_1 + E_2$ is defined in terms of the types of E_1 & E_2.

A typical rule for type synthesis has the form:-

if f has type $s \to t$ and x has type s.
then expression $f(x)$ has type t.

here -

→ f, x denote expression
→ $s \to t$ denotes a function from s to t.

This rule for function with 1 argument carries over to functions with several arguments.

The same can be adopted for $E_1 + E_2$ if it is viewed as a function application -

add (E_1, E_2)

Inference :- Type inference determines the type of a language construct from the way it is used. for eg: let null(x) be a function that test whether a list is empty or not.

So, the type of elements of 'x' is not known.

Variables representing type expressions allow us to talk about unknown types.
A typical rule for type inference has the form:

if f(x) is an expression,
then for some α and β, f has type $\alpha \to \beta$ and x has type α.

(Here Greek letters $\alpha, \beta \ldots$ are used for type variables in type expressions)

Type inference is needed for languages like ML, which checks types, but do not require names to be declared.

The rules for checking statements are similar to those for expressions.
for eg:- if (E) S; will be considered as if it were the application of a function 'if' to E and S. Let the special type 'void' denote the absence of value then function 'if' expects to be applied to a 'boolean' and 'a void' the result of the application is void.

TYPE CONVESION

Consider expression like x + i where
x is integer and
i is type float.
Since the representation of integers and floating point numbers are different within a computer, the compiler need to convert one of the operands of + to ensure both the operands are of same type

For ex:) Converting integer 2 to float

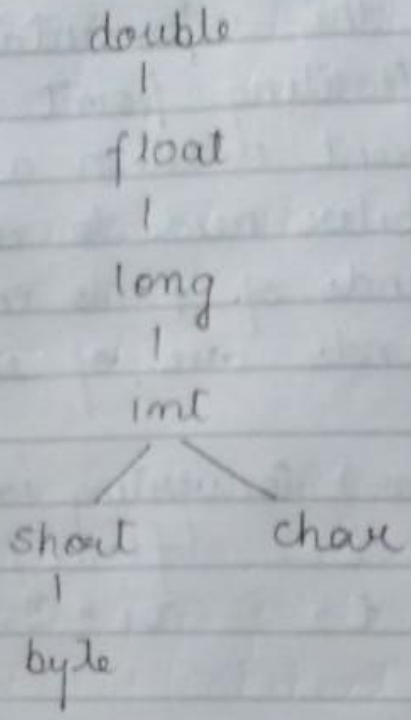
t_1 = (float) 2
t_2 = t_1 * 3.14

We introduce another attribute E.type whose value is either integer or float. The rule with ~~E → f~~
E → E_1 + E_2 builds on the pseudocode

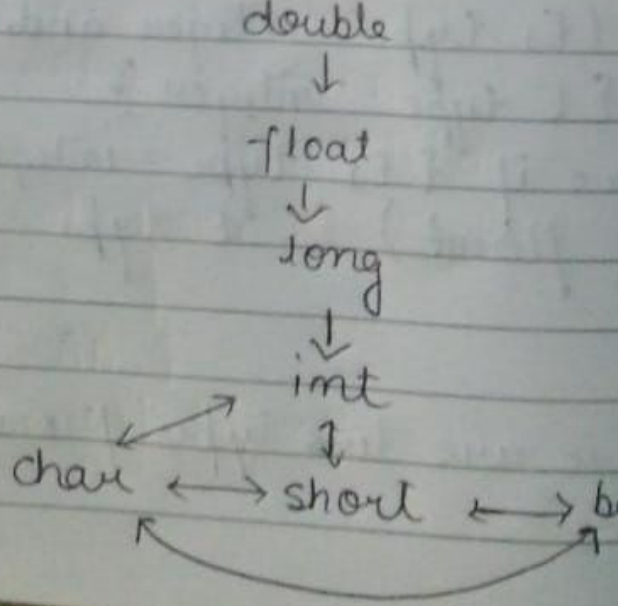
if (E_1.type = integer and E_2.type = integer)
{ E.type = integer }
else if { E_1.type = integer and E_2.type = float) { E.type = float }
:
:

There are two types of conversion:

1) Widening Conversion :) Can happen if both types are compatible and target type larger than source type. It preserves the information



2) Narrowing Conversion :) When we are assigning a larger type to a small type. It can lose information.



Implicit Type Conversion (Coercions)
It is done automatically by the compiler

The semantic action for checking $E_1 + E_2$ uses two function =>

1) Max(t_1, t_2) => takes two types t_1 and t_2 and returns maximum (or least upper bound). It declares error if t_1 and t_2 are not in hierarchy.

2) Widen(a, t, w) => generates type conversion if needed. It returns a itself if t and w are of same type otherwise returns an instruction to do the conversion and place result in temporary, which is returned as result.

```
Addr widen (Addr a, Type t, Type w)
    if (t = w) return a;
    else if ( t = int and w = float) {
        temp = new Temp();
        gen (temp '=' (float) 'a);
        return temp;
    }
    else error;
}
```