INTERMEDIATE CODES

Syntax directed translations

- An extension to CFG
- Generates some form of intermediate codes

With each production some action is associated so when ever a reduction takes place, the action associated with it is executed

These actions perform some semantics as well- hence also called semantic action or semantic rule

For the production $E \rightarrow E + E$

the syntax directed translation (semantic action) would be

$E.VAL = E^{(1)}.VAL + E^{(2)}.VAL$

$S \rightarrow E \$$	Print E.VAL
$E \rightarrow E + E$	$E.VAL = E^{(1)}.VAL + E^{(2)}.VAL$
$E \rightarrow E * E$	$E.VAL = E^{(1)}.VAL * E^{(2)}.VAL$
$E \rightarrow (E)$	$E.VAL = E^{(1)}.VAL$
$E \rightarrow I$	E.VAL = I.VAL
$I \rightarrow I * digit$	$I.VAL = 10 * I^{(1)}.VAL + Lexval$
$I \rightarrow digit$	I.VAL = Lexval

FORMS OF INTERMEDIATE CODES

1. Polish notation

E → E OP E, E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || OP

E→ (E), E.CODE := $E^{(1)}$.CODE

E→ id, E.code := id

2. THREE ADDRESS CODES

A source statement is represented as a sequence of elementary operations. Each elementary operation is described in the form of a pseudo machine instruction

Triples – uses three fields

Quadruples- uses four fields

Operator	Operand1	Operand2

EXPRESSION TRANSLATION

Requires two fields (for the expression E)

E.PLACE : Name that will hold the value of the expression

E.CODE : A sequence of three address statements evaluating the expression

And a function NEWTEMP( ) to create new temporary names

$A \rightarrow id := E$	A.CODE := E.CODE ‖
	Id.PLACE‖ :=‖ E.PLACE
$E \rightarrow E^{(1)} + E^{(2)}$	T := NEWTEMP( );
	E.PLACE := T;
	E.CODE := $E^{(1)}$.CODE‖ $E^{(2)}$.CODE‖
	E.PLACE‖:=‖ $E^{(1)}$.PLACE ‖ + ‖ $E^{(2)}$.PLACE

$E \rightarrow E^{(1)} * E^{(2)}$	T := NEWTEMP( ); E.PLACE := T; E.CODE := $E^{(1)}$.CODE|| $E^{(2)}$.CODE|| E.PLACE||:=|| $E^{(1)}$.PLACE || * || $E^{(2)}$.PLACE
$E \rightarrow - E^{(1)}$	T := NEWTEMP( ); E.PLACE := T; E.CODE := $E^{(1)}$.CODE|| E.PLACE||:= - || $E^{(1)}$.PLACE
$E \rightarrow (E^{(1)})$	E.PLACE:= | $E^{(1)}$.PLACE; E.CODE := $E^{(1)}$.CODE
E→ id	E.PLACE := id.PLACE; E.CODE := null

Mixed Mode expressions

E.MODE : Mode of the expression E

T:=NEWTEMP( );

(1) If $E^{(1)}$.MODE = INTERGER and $E^{(2)}$.MODE = INTEGER then

begin

GEN(T:=$E^{(1)}$.PLACE int op $E^{(2)}$.PLACE);

E.MODE : = INTEGER

end

(2) If $E^{(1)}$.MODE = REAL and $E^{(2)}$.MODE = REAL then

---

(3) If $E^{(1)}$.MODE = INTEGER and $E^{(2)}$.MODE = REAL then

```
begin
    U:= NEWTEMP( );
    GEN( U := inttoreal E(1).PLACE );
    GEN( T := U real op E(2).PLACE );
     E.MODE :=REAL
end
```

(4) If $E^{(1)}$.MODE = REAL and $E^{(2)}$.MODE = INTEGER then

```
begin
    U:= NEWTEMP( );
    GEN( U := inttoreal E(2).PLACE );
    GEN( T := E(1).PLACE real op U);
     E.MODE :=REAL
end
```

Boolean Expressions

E→ E OR E | E AND E | NOT E |… | E relop E

- Numeric encoding- like arithmetic expressions

A OR B AND C

T1 = B AND C

T2 = A OR T1

A< B OR C

(1) If A < B goto (4)
(2) T1 : = 0;
(3) goto (5)
(4) T1 = 1;
(5) T2 := T1 OR C

E→ $E^{(1)}$ OR $E^{(2)}$

T := NEWTEMP( );

E.PLACE :=T;

GEN($T:=E^{(1)}$.PLACE OR $E^{(2)}$.PLACE)

E → id relop id

T := NEWTEMP( );

E.PLACE :=T;

GEN(if $id^{(1)}$.PLACE relop $id^{(2)}$.PLACE goto NEXTQUAD + 3);

GEN(T:=0);

GEN(goto NEXTQUAD + 2);

GEN (T=1);

- **CONTROL FLOW REPRESENTATION**

Boolean statements in context of conditional statements

Every Boolean expression has two exits

True exit to statement location TRUE

False exit to statement location FALSE

$E \rightarrow E^{(1)}$ op $E^{(2)}$

If $E^{(1)}$ is true, E is true

So, TRUE for E is TRUE for $E^{(1)}$

If $E^{(1)}$ is false, $E^{(2)}$ must be evaluated

So, true exit of $E^{(2)}$ is TRUE of E

and false exit of $E^{(2)}$ is FALSE of E

Backpatching

In generating quadruples for Boolean expressions, it is possible that we may not have generated quadruples to which the jumps are to be made at the time the jump statements are generated.

Each such quadruples will be on one or another list of quadruples to be filled in when proper location is found. This subsequent filling in of quadruples is *backpatching*

TRANSLATION REQUIRES THREE FUNCTIONS

MAKELIST(i) : creates a new list containing only I, an index into the array of quadruples being generated and returns a pointer to the list it has made

MERGE(p1,p2) : concatenates two lists pointed to by p1 and p2 respectively and returns a pointer to the concatenated list

BACKPATCH(p, i) : fills in up the target location as i for each of the quadruples on the list pointed by p

TRANSLATIONS

Two translation are required for every expression E

E.TRUE and E.FALSE

Each of these translations are the pointers to the list of already generated quadruples that must be filled in when the target location is found

To pick up next quadruple (NEXTQUAD) at appropriate time to be used as i in backpatch(p, i) a new production is defined as

M → e { M.QUAD : = NEXTQUAD }

$E \rightarrow E^{(1)}$ OR M $E^{(2)}$	BACKPATCH($E^{(1)}$.FALSE, M.QUAD); E.TRUE:= MERGE($E^{(1)}$.TRUE, $E^{(2)}$.TRUE); E.FALSE := $E^{(2)}$.FALSE
$E \rightarrow E^{(1)}$ AND M $E^{(2)}$	BACKPATCH($E^{(1)}$.TRUE, M.QUAD); E.FALSE:= MERGE($E^{(1)}$.FALSE, $E^{(2)}$.FALSE); E.TRUE := $E^{(2)}$.TRUE
$E \rightarrow$ NOT $E^{(1)}$	E.TRUE := $E^{(1)}$.FALSE; E.FALSE := $E^{(1)}$.TRUE
$E \rightarrow$ (E)	E.TRUE := $E^{(1)}$.TRUE; E.FALSE := $E^{(1)}$.FALSE

E → id	E.TRUE : = MAKELIST(NEXTQUAD); E.FALSE := MAKELIST(NEXTQUAD+1); GEN(if id.PLACE goto __); GEN(goto __)
E → $id^{(1)}$ relop $id^{(2)}$	E.TRUE : = MAKELIST(NEXTQUAD); E.FALSE := MAKELIST(NEXTQUAD+1); GEN(if $id^{(1)}$.PLACE relop $id^{(2)}$.PLACE goto __); GEN(goto __)
M →e	M.QUAD → NEXTQUAD

TRANSLATION OF ARRAY REFERENCES

Grammar for array references

A → L := E

E → E + E | (E) | L

L → elist]

L → id

elist → elist, E

elist → id[E

L is l-value of array

Generating three address code for array references

For a two dimensional array A[10*20]

and for a word organized memory

The array A will need 200 words (row major order)

if *a* be the location of the first word of the block, A[i,j] will be computed as

a + 20(i-1) + j-1

ie, (a-21) + 20*i + j

computed in index register

Offset computed in base register

T1 := i

T2 := 20*T1

T3 := j

T4 : =T2 + T3

T5 := a-21

T := T5[T4]

Translations needed

elist.ARRAY : denotes the symbol table entry for the ARRAY name

elist.NDIM : records the no. of dimensions(i.e. index expression E) in the elist

elist.PLACE : denotes symbol table entry for a name whose value is computed by the three address code generated for elist

LIMIT(ARRAY,i) : returns the upper limit along the i^{th} dimension

L.PLACE : Pointer to symbol table location for name in case L is not array

L.OFFSET : null in case L is not an array

A → L := E

```
{   If(L.OFFSET =null) then
    GEN( L.PLACE := E.PLACE);
    else
    GEN( L.PLACE[L.OFFSET] := E.PLACE) }
```

E → L

```
{    if (L.OFFSET = null) then
     E.PLACE = L.PLACE
  else  begin
          T := NEWTEMP( );
          GEN( T := L.PLACE[L.OFFSET]);
          E.PLACE := T
        end }
```

L → elist]

{ T := NEWTEMP( );
U := NEWTEMP( );
GEN(T := elist.ARRAY –C); C is a constant
GEN(U := bpw * elist.PLACE);
L.PLACE := T;
L.OFFSET := U; }

L → id

{ L.PLACE := id.PLACE
L.OFFSET := null }

elist → elist$^{(1)}$, E

{ T := NEWTEMP( );

GEN(T := elist.PLACE * LIMIT(elist$^{(1)}$.ARRAY, elist.NDIM+1);

GEN(T := T + E.PLACE);

elist.ARRAY := elist$^{(1)}$.ARRAY;

elist.PLACE := T;

elist.NDIM := elist$^{(1)}$.NDIM + 1 }

elist → id[E

elist.PLACE := E.PLACE;

elist.NDIM := 1;

elist.ARRAY := id.PLACE