Intermediate Code Generation
Part I

Chapter 6: Intermediate Representations

Slides adapted from:
© Robert van Engelen, Florida State University
Intermediate Representations

- **Graphical representations** (e.g. abstract syntax tree, DAGs)
- **Three-address code** (e.g. *triples* and *quads*):
  \[ x = y \text{ op } z \]
- **Two-address code**:
  \[ x = \text{ op } y \]
  which is the same as \[ x = x \text{ op } y \]
- **Postfix notation**: operations on values stored on operand stack (similar to JVM bytecode)
S-Attributed SDD for Generating Abstract Syntax Trees

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E$</td>
<td>$S$.node = new Node(‘=’, Leaf(id, id.entry), E.node)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E$.node = new Node(‘+’, $E_1$.node, $E_2$.node)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 * E_2$</td>
<td>$E$.node = new Node(‘*’, $E_1$.node, $E_2$.node)</td>
</tr>
<tr>
<td>$E \rightarrow - E_1$</td>
<td>$E$.node = new Node(‘uminus’, $E_1$.node)</td>
</tr>
<tr>
<td>$E \rightarrow (E_1)$</td>
<td>$E$.node = $E_1$.node</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E$.Node = new Leaf(id, id.entry)</td>
</tr>
</tbody>
</table>
Pros: easy restructuring of code and/or expressions for intermediate code optimization
Cons: memory intensive
Directed Acyclic Graphs

- Directed acyclic graphs (DAGs) identify and uniquely represent common sub-expressions of an abstract syntax tree
- Used to generate efficient code
Directed Acyclic Graphs for Abstract Syntax Trees

\[ a = b \times -c + b \times -c \]
Directed Acyclic Graphs
for Abstract Syntax Trees

\[ a + a \ast (b - c) + (b - c) \ast d \]
S-Attributed SDD for Generating DAGs

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E$.node = <code>new Node</code>('+', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T$</td>
<td>$E$.node = <code>new Node</code>('-', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>$T \rightarrow (E)$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>$T \rightarrow id$</td>
<td>$T$.node = <code>new Leaf</code>(<code>id</code>, <code>id</code>.entry)</td>
</tr>
<tr>
<td>$T \rightarrow num$</td>
<td>$T$.node = <code>new Leaf</code>(<code>num</code>, <code>num</code>.val)</td>
</tr>
</tbody>
</table>
Value-Number Representation for DAGs

\[ i = i + 10 \]

Value number of children

To symbol table entry for \( i \)
Value-Number Construction Method

• Use **signature** \(\langle op, lc, rc \rangle\) for nodes, where
  – \(op\) is a label
  – \(lc, rc\) are value numbers
  – \(rc\) is null for leaf nodes

• To construct a new node
  – Use signature to search the array and return the **value number** if found
  – Create new entry otherwise
Three-Address Code

- In three-address code (TAC or 3AC) instructions have at most one operator in the right hand side.
- When translating, compiler needs to generate temporary names.

\[
x + y * z \quad \Rightarrow \quad t_1 = y * z \quad \text{and} \quad t_2 = x + t_1
\]
Three-Address Code

\[ a = b \cdot -c + b \cdot -c \]

\[
\begin{align*}
  t_1 &= -c \\
  t_2 &= b \cdot t_1 \\
  t_3 &= -c \\
  t_4 &= b \cdot t_3 \\
  t_5 &= t_2 + t_4 \\
  a &= t_5
\end{align*}
\]

From abstract syntax tree

\[
\begin{align*}
  t_1 &= -c \\
  t_2 &= b \cdot t_1 \\
  t_3 &= -c \\
  t_5 &= t_2 + t_2 \\
  a &= t_5
\end{align*}
\]

From DAG representation of abstract syntax tree
Addresses and Labels

• In 3AC an **address** can be
  – A name: an identifier from source program, or else pointer to its table entry
  – A constant
  – A compiler-generated temporary

• Symbolic **labels** (index of instructions) used to alter control flow
Three-Address Statements

- Assignment (binary): \( x = y \ op \ z \)
- Assignment (unary): \( x = op \ y \)
- Copy statement: \( x = y \)
Three-Address Statements

• Unconditional jump: \texttt{goto lab}

• Conditional jump: \texttt{if x goto lab, ifFalse x goto lab}

• Conditional jump: \texttt{if x relop y goto lab}

• Indexed assignment: \( x = y[i], x[i] = y \)

• Pointer assignment: \( x = &y, x = *y, *x = y \)
Three-Address Statements

- Procedure call:
  \[
  \text{param } x_1 \\
  \text{param } x_2 \\
  \ldots \\
  \text{param } x_n \\
  \text{call } p, n
  \]

- Assignment + call: \[ y = \text{call } p, n \]
Implementation of 3AC: Quadruples

- 3AC instructions represented as data structures called quadruples
- Quadruples have four fields
  - \( op, \ arg_1, \ arg_2, \ result \)
- Some instructions use a proper subset of these fields
Example

\[ a = b(-c) + b(-c) \]

\[
\begin{align*}
    t_1 &= \text{minus } c \\
    t_2 &= b \times t_1 \\
    t_3 &= \text{minus } c \\
    t_4 &= b \times t_3 \\
    t_5 &= t_2 + t_4 \\
    a &= t_5
\end{align*}
\]
Example

\[ t_1 = \text{minus } c \]
\[ t_2 = b \times t_1 \]
\[ t_3 = \text{minus } c \]
\[ t_4 = b \times t_3 \]
\[ t_5 = t_2 + t_4 \]
\[ a = t_5 \]

Quads (quadruples)

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
<td>( t_1 )</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
<td>( t_3 )</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>( t_3 )</td>
<td>( t_4 )</td>
</tr>
<tr>
<td>+</td>
<td>( t_2 )</td>
<td>( t_4 )</td>
<td>( t_5 )</td>
</tr>
<tr>
<td>=</td>
<td>( t_5 )</td>
<td></td>
<td>( a )</td>
</tr>
</tbody>
</table>

Pros: easy to rearrange code for global optimization
Cons: lots of temporaries
Implementation of 3AC: Triples

• Triples are an alternative representation for 3AC instructions using three fields
  – $op, arg_1, arg_2$

• We refer to the result of an operation by its **position**, rather than by an explicit temporary name
Example

$t_1 = \text{minus } c$
$t_2 = b \times t_1$
$t_3 = \text{minus } c$
$t_4 = b \times t_3$
$t_5 = t_2 + t_4$
$a = t_5$

Pros: temporaries are implicit
Cons: difficult to rearrange code
## Implementation of 3AC:

### Indirect Triples

<table>
<thead>
<tr>
<th>#</th>
<th>Stmt</th>
<th>Alias</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(14)</td>
<td>(14)</td>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>(15)</td>
<td>(15)</td>
<td>*</td>
<td>b</td>
<td>(14)</td>
</tr>
<tr>
<td>(2)</td>
<td>(16)</td>
<td>(16)</td>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>(17)</td>
<td>(17)</td>
<td>*</td>
<td>b</td>
<td>(16)</td>
</tr>
<tr>
<td>(4)</td>
<td>(18)</td>
<td>(18)</td>
<td>+</td>
<td>(15)</td>
<td>(17)</td>
</tr>
<tr>
<td>(5)</td>
<td>(19)</td>
<td>(19)</td>
<td>=</td>
<td>a</td>
<td>(18)</td>
</tr>
</tbody>
</table>

**Instructions**

**Triple container**

**Pro:** temporaries are implicit & easier to rearrange code
Type Expressions

- Types have internal structures, which are represented by *type expressions*

\[
\text{int}[2][3]
\]

```
array
  2
  array
  3
    int
```
Graph Representations for Type Expressions

int *f(char*,char*)

Tree form

DAG
Cyclic Graph Representations

```c
struct Node
{
    int val;
    struct Node *next;
};
```

Cyclic graph
Recursive definition for Type Expressions

- *Basic types*, such as `int`, `boolean`, and `void`
- *Type names*, such as typedefs in C and named types in Pascal
- *Array* type constructors applied to an integer number and a type expression
- *Record* type constructors applied to the field names and their types
Recursive definition for Type Expressions

- *Pointer* type constructor, applied to a type expression
- $s \rightarrow t$ denotes function from type $s$ to type $t$
- Cartesian product $s \times t$ of type expressions, used to represent list or tuple of types (function parameters)
- Variables whose values are type expressions
Name Equivalence

• Each *type name* is a distinct type, even when the associated type expressions are the same
• Types are identical only if names match
• Used in Pascal (inconsistently)

```
type link = ^node;
var next : link;
  last : link;
  p : ^node;
  q, r : ^node;
```

With name equivalence in Pascal:
```
p ≠ next
p ≠ last
p = q = r
next = last
```
Structural Equivalence of Type Expressions

- Two types are the same if they are *structurally equivalent*
- Used in C, Java, C#
Structural Equivalence of Type Expressions

- Two structurally equivalent types receive the same pointer address when constructing graphs by sharing nodes

```
struct Node
{
    int val;
    struct Node *next;
};
struct Node s, *p;

... p = &s; // OK
... *p = s; // OK
```
Constructing Type Graphs in Bison

Type *mkint()  
construct int node if not already constructed

Type *mkarr(Type*,int)  
construct array-of-type node if not already constructed

Type *mkptr(Type*)  
construct pointer-of-type node if not already constructed
Constructing Type Graphs in Bison

%union
{ Symbol *sym;
  int num;
  Type *typ;
}
%token INT
%token <sym> ID
%token <num> NUM
%type <typ> type
%
decl : type ID             { addtype($2, $1); }       
   | type ID ‘[’ NUM ‘]’    { addtype($2, mkarr($1, $4)); }  
   ;
type : INT                 { $$ = mkint(); }         
   | type ‘*’                { $$ = mkptr($1); }         
   ;
Type Expression and Storage Allocation

- Apply type expression and SDT to determine amount of storage needed at run time

\[
\begin{align*}
D & \to T \text{id} \; D \mid \varepsilon \\
T & \to B \; C \mid \text{record} \; \{ \; D \; \} \\
B & \to \text{int} \mid \text{float} \\
C & \to \varepsilon \mid [ \text{num} ] \; C
\end{align*}
\]
Type Expression and Storage Allocation

• Synthesized attribute \textit{type} : type expression
• Synthesized attribute \textit{width} : number of storage units needed for the type
• Inherited attributes \textit{t} and \textit{w} : used to pass type and width information to type constructors
Type Expression
and Storage Allocation

\[
T \rightarrow B \quad \{ \ C.t = B.type; \ C.w = B.width; \ \}
\]

\[
C \quad \{ \ T.type = C.type; \ T.width = C.width; \ \}
\]

\[
B \rightarrow \text{int} \quad \{ \ B.type = \text{integer}; \ B.width = 4; \ \}
\]

\[
B \rightarrow \text{float} \quad \{ \ B.type = \text{float}; \ B.width = 8; \ \}
\]

\[
C \rightarrow \varepsilon \quad \{ \ C.type = C.t; \ C.width = C.w; \ \}
\]

\[
C \rightarrow [ \text{num} ] \quad \{ \ C_1.t = C.t; \ C_1.w = C.w; \ \}
\]

\[
C_1 \quad \{ \ C.type = \text{array}(\text{num}.value, C_1.type); \ C.width = \text{num}.value \times C_1.width; \ \}
\]
Type Expression and Storage Allocation

$\text{type} = \text{integer}$
$\text{width} = 4$

$\text{int}$

$\text{array}(2, \text{array}(3, \text{integer}))$
$\text{width} = 24$

$\text{array}(3, \text{integer})$
$\text{width} = 12$

$\text{integer}$
$\text{width} = 4$

$t = \text{integer}$
$w = 4$
Sequences of Declarations

• Augment previous grammar by accounting for sequences of declarations, distributed within some block

• Use nonterminal $T$ as in previous SDT

• Use global variable $offset$ to keep track of next available relative address in central memory
Sequences of Declarations

\[ P \rightarrow \{ \text{offset} = 0; \} \; D \]
\[ D \rightarrow T \; \text{id} \; ; \{ \text{top.put(id.lexeme, T.type, offset);} \]
\[ \quad \text{offset} = \text{offset} + T.width; \; \} \; D_1 \]
\[ D \rightarrow \varepsilon \]
Fields in Records / Classes

• Use the offset technique also for records and classes (class methods do not affect space allocation)
• Field records must be distinct within the same structure
• Offset is computed relative to the area record
Fields in Records / Classes

• Example:

```plaintext
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
...

x = p.x + q.x;
```
Fields in Records / Classes

• Augment previous declaration grammar with rule \( T \rightarrow \text{record} \ {D} \)

• Use SDT

\[
T \rightarrow \text{record} \ {'} \quad \{ \begin{array}{l}
\text{Env.push(top); top = new Env();} \\
\text{Stack.push(offset); offset = 0;}
\end{array} \} \\
D \ {'} \quad \{ \begin{array}{l}
T.type = \text{record}(top); \\
T.width = offset; \quad \text{top = Env.pop();} \\
offset = \text{Stack.pop();}
\end{array} \}
\]
Translation of Expressions

- Translation of assignment statements involving arithmetic expressions in 3AC
- Used attributes (all synthesized)
  - `code` to store 3AC
  - `addr` to denote the memory address (symbol table entry or temporary) that will hold the computed value for a sub-expression
  - `lexeme` to denote name of identifier token
Translation of Expressions

• Used operators
  – *top* is current symbol table
  – *get()* retrieves symbol table entry
  – *gen()* generates 3AC instructions
  – **new Temp()** generates a fresh temporary name
Translation of Expressions

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E;$</td>
<td>$S\text{.code} = E\text{.code} \parallel \text{gen(top.get(id.lexeme) ‘=’ E.addr)}$</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 + E_2$ | $E\text{.addr} = \text{new Temp}();$
| | $E\text{.code} = E_1\text{.code} \parallel E_2\text{.code} \parallel$
| | $\quad \text{gen}(E\text{.addr ‘=’ E}_1\text{.addr ‘+’ E}_2\text{.addr})$ |
| $E \rightarrow E_1 * E_2$ | $E\text{.addr} = \text{new Temp}();$
| | $E\text{.code} = E_1\text{.code} \parallel E_2\text{.code} \parallel$
| | $\quad \text{gen}(E\text{.addr ‘=’ E}_1\text{.addr ‘*’ E}_2\text{.addr})$ |
| $E \rightarrow - E_1$ | $E\text{.addr} = \text{new Temp}();$
| | $E\text{.code} = E_1\text{.code} \parallel \text{gen}(E\text{.addr ‘=’ ‘minus’ E}_1\text{.addr})$ |
| $E \rightarrow ( E_1 )$ | $E\text{.addr} = E_1\text{.addr}$
| | $E\text{.code} = E_1\text{.code}$ |
| $E \rightarrow \text{id}$ | $E\text{.addr} = \text{top.get(id.lexeme)}$
| | $E\text{.code} = ‘’$ |
Translation of Expressions

\[ a = b + -c ; \]

\[ t1 = \text{minus } c \]
\[ t2 = b + t1 \]
\[ a = t2 \]
Translation of Expressions

\[ S \rightarrow id \ E \ E + E ; \]

- \( id \rightarrow addr = b \)  
  \( code = "\)  

- \( id \rightarrow addr = t1 \)  
  \( code = 't1 = minus c' \)  
- \( addr = t2 \)  
  \( code = 't2 = b + t1' \)  
- \( addr = a \)  
  \( code = 't2 = b + t1' \)  
- \( addr = a \)  
  \( code = 't1 = minus c' \)  
- \( addr = c \)  
  \( code = "\)
Incremental Translation

- Attribute *code* not used: instructions generated as a stream by recursive calls to `gen()

  - Generate only the new 3AC instructions
  - Append to the sequence of instructions generated so far
Incremental Translation

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow \text{id} = E ;$</td>
<td>$\text{gen(top.get(id.lexeme) } \text{‘=} \text{ E.addr)}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>$E.addr = \text{new Temp();}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 * E_2$</td>
<td>$\text{gen(E.addr } \text{‘=} \text{ E_1.addr } \text{‘+’ E_2.addr)}$</td>
</tr>
<tr>
<td>$E \rightarrow - E_1$</td>
<td>$E.addr = \text{new Temp();}$</td>
</tr>
<tr>
<td>$E \rightarrow ( E_1 )$</td>
<td>$\text{gen(E.addr } \text{‘=} \text{ ‘minus’ E_1.addr)}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E.addr = E_1.addr$</td>
</tr>
<tr>
<td></td>
<td>$E.addr = \text{top.get(id.lexeme)}$</td>
</tr>
</tbody>
</table>
Addressing Array Elements

- Compile-time addressing of arrays can be applied only when array size is known
- When array size is dynamic, addressing must be evaluated as program executes
- We use translation into 3AC to evaluate addressing at run-time
Addressing Array Elements

- Row-major order for storing multi-dimensional arrays; zero-addressing

![Diagram showing array indexing and row-major order]

- `a[1][0]`: Array element at row 1, column 0.
- `c(a)`: Base address of the array `a`.
- Rows 0, 1, 2 labeled for demonstration.
Addressing Array Elements

- Row-major order vs. column-major order

\[ a[2][3] : \]

<table>
<thead>
<tr>
<th>( c(a) )</th>
<th>( c(a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[0][0] )</td>
<td>( a[0][0] )</td>
</tr>
<tr>
<td>( a[0][1] )</td>
<td>( a[1][0] )</td>
</tr>
<tr>
<td>( a[0][2] )</td>
<td>( a[0][1] )</td>
</tr>
<tr>
<td>( a[1][0] )</td>
<td>( a[1][1] )</td>
</tr>
<tr>
<td>( a[1][1] )</td>
<td>( a[0][2] )</td>
</tr>
<tr>
<td>( a[1][2] )</td>
<td>( a[1][2] )</td>
</tr>
</tbody>
</table>

Row-major         Column-major
Addressing Array Elements

• Array $a$ : one dimension
  – Base address: $c(a)$
  – Type width: $w$

• Address polynomial
  – $a[i]$ located at $c(a) + i \times w$
Addressing Array Elements

• Array $a$ : two dimensions
  – Row width: $w_1$
  – Element width : $w_2$

• Address polynomial
  – $a[i_1,i_2]$ located at
    $c(a) + i_1 \times w_1 + i_2 \times w_2$
Addressing Array Elements

- Array $a : k$ dimensions
  - Generalized widths (lower dim components):
    $w_j, 1 \leq j \leq k$

- Address polynomial
  - $a[i_1, \ldots, i_k]$ located at
    $c(a) + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$
Addressing Array Elements

- **Array** $a$: two dimensions
  - Number of elements in dimension 1: $n_1$ (column)
  - Number of elements in dimension 2: $n_2$ (row)
  - Element width: $w$

- **Address polynomial**
  - $a[i_1, i_2]$ located at
    
    $$c(a) + (i_1 \times n_2 + i_2) \times w$$
Addressing Array Elements

• Array $a : k$ dimensions
  – Number of elements in dimension $j$: $n_j$
  – Element width: $w$

• Address polynomial
  – $a[i_1, \ldots, i_k] \text{ located at }$
    
    $c(a) +$
    
    $(\ldots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots ) \times n_k + i_k) \times w$
Addressing Array Elements

- Other addressing conventions (Pascal)

\[
A : \text{array } [10..20] \text{ of integer};
\]
\[
\ldots = A[i] = base_A + (i - low) \times w = i \times w + c
\]
where \(c = base_A - low \times w\)
with \(low = 10, w = 4\)

\[
t1 = c // c = base_A - 10 \times 4
\]
\[
t2 = i \times 4
\]
\[
t3 = t1[t2]
\]
\[
\ldots = t3
\]
Addressing Array Elements

- Other addressing conventions (two dimensions)

\[
A : \text{array } [1..2,1..3] \text{ of integer; (Row-major)}
\]
\[
... = A[i,j] = base_A + ((i - low_1) * n_2 + j - low_2) * w
\]
\[
= ((i * n_2) + j) * w + c
\]
where \( c = base_A - ((low_1 * n_2) + low_2) * w \)
with \( low_1 = 1, low_2 = 1, n_2 = 3, w = 4 \)

\[
t1 := i * 3
\]
\[
t1 := t1 + j
\]
\[
t2 := c \quad // c = base_A - (1 * 3 + 1) * 4
\]
\[
t3 := t1 * 4
\]
\[
t4 := t2[t3]
\]
\[
... := t4
\]
Addressing Array Elements

• Assume arrays
  – $X$ with dimension $d_1 \times d_2$
  – $Y$ with dimension $d_3 \times d_4$

• Consider the assignment statement

\[
X[i, j] = Y[i + j, k] + z
\]
Addressing Array Elements

\[ X[i,j] = Y[i+j,k] + z \]

t1 = i * d2
t2 = t1 + j
t3 = c(X)
t4 = t2 * width(X)
t5 = i + j
t6 = t5 * d4
t7 = t6 + k
t8 = c(Y)
t9 = t7 * width(Y)
t10 = t8[t9]
t11 = t10 + z
t3[t4] = t11
Addressing Array Elements

- Apply type expression and SDT to determine offsets for array indexing

\[
\begin{align*}
S & \rightarrow L = E \\
E & \rightarrow E + E \\
E & \rightarrow ( E ) \\
E & \rightarrow L \\
L & \rightarrow \text{Elist} \\
L & \rightarrow \text{id} \\
\text{Elist} & \rightarrow \text{Elist} , E \\
\text{Elist} & \rightarrow \text{id} [ E
\end{align*}
\]
\[ X[i, j] = Y[i + j, k] + z \]
Addressing Array Elements

- **E.place**: temp or symbol entry holding value obtained in computation of expression
- **L.place**: temp holding base address of array
- **L.offset**: temp holding array offset
- **Elist.array**: symbol table entry for array
- **Elist.place**: temporary for offset computation
- **Elist.dim**: dimension index
# Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
</table>
| $S \rightarrow L = E$ | if ( $L.offset == null$ )
| | \hspace{1cm} gen($L.place \ '==' \ E.place$); |
| | else \hspace{1cm} gen($L.place \ '[' \ L.offset \ ']' \ '==' \ E.place$); |
| $E \rightarrow E_1 + E_2$ | $E.place = newlabel();$
| | gen($E.place \ '==' \ E_1.place \ '+' \ E_2.place$); |
| $E \rightarrow ( E_1 )$ | $E.place = E_1.place$; |
### Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
</table>
| \[ E \rightarrow L \] | if \( (L\.offset == null) \)  
\[ E\.place = L\.place; \]  
else  
\{  
  \[ E\.place = newlabel(); \]  
  gen\( (E\.place \text{ '=}\)  
    \[ L\.place [' L\.offset '] \); \}  
| \[ L \rightarrow id \] | \[ L\.place = id\.place; \]  
\[ L\.offset = null; \] |
# Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
</table>
| $L \rightarrow Elist$ | $L.place = newlabel();$
|              | $L.offset = newlabel();$
|              | $\text{gen}(L.place \ '=$ \ c(Elist.array));$
|              | $\text{gen}(L.offset \ '=$$
|              | $Elist.place \ '*' \ \text{width}(Elist.array);$|
| $Elist \rightarrow Elist_1, E$ | $t = newlabel();$
|              | $m = Elist_1.dim + 1;$
|              | $\text{gen}(t \ '=$ \ Elist_1.place \ '*' \ \text{limit}(Elist_1.array, m));$
|              | $\text{gen}(t \ '=$ \ t \ '+' \ E.place;$
|              | $Elist.array = Elist_1.array;$
|              | $Elist.place = t;$
|              | $Elist.dim = m;$ |
## Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Elist \rightarrow \textbf{id} \ [ \ E$</td>
<td>$Elist.array = \textbf{id}.place$;</td>
</tr>
<tr>
<td></td>
<td>$Elist.place = E.place$;</td>
</tr>
<tr>
<td></td>
<td>$Elist.dim = 1$;</td>
</tr>
</tbody>
</table>
\[ X[i, j] = Y[i + j, k] + z \]
Addressing Array Elements

\[
S \rightarrow \text{id} = E ; \\
| \ L = E ; \\
E \rightarrow E + E \\
| \ \text{id} \\
| \ L \\
L \rightarrow \text{id} [ E ] \\
| \ L [ E ]
\]
Addressing Array Elements

- $E.addr$: temp holding value of $E$
- $L.addr$: temp holding value obtained in computation of offset
- $L.array$: pointer to symbol table entry for array
  - $L.array.base$: base address of array
  - $L.array.type.width$: size of array elements
  - $L.array.type.elem$: type of array elements
- $L.type$: type of subarray generated by $L$
Addressing Array Elements

\[
S \rightarrow \text{id} = E ; \\
\{ \text{gen}(\text{top.get(id.lexeme)} \ '=?' \ E.\text{addr}); \}
\]

\[
S \rightarrow L = E ; \\
\{ \text{gen}(L.\text{array.base} \ '[' L.\text{addr} ']' \ '=?' \ E.\text{addr}); \}
\]

\[
E \rightarrow E_1 + E_2 \\
\{ E.\text{addr} = \text{new Temp}(); \\
\text{gen}(E.\text{addr} \ '=?' \ E_1.\text{addr} \ '+' \ E_2.\text{addr}); \}
\]

\[
E \rightarrow \text{id} \\
\{ E.\text{addr} = \text{top.get(id.lexeme)}; \}
\]

\[
E \rightarrow L \\
\{ E.\text{addr} = \text{new Temp}(); \\
\text{gen}(E.\text{addr} \ '=?' \ L.\text{array.base} \ '[' L.\text{addr} ']' ); \}
\]
Addressing Array Elements

\[ L \to \text{id} \ [ \ E \ ] \quad \text{\{ } \quad L.\text{array} = \text{top.get(id.lexeme)}; \]
\[ \text{L.type} = \text{L.array.type.elem}; \]
\[ L.\text{addr} = \text{new Temp}(); \]
\[ \text{gen}(L.\text{addr} \ ']=' \ E.\text{addr} \ '*' \]
\[ L.\text{type.width}); \ \text{\}} \]

\[ L \to L_1 \ [ \ E \ ] \quad \text{\{ } \quad L.\text{array} = L_1.\text{array}; \]
\[ L.\text{type} = L_1.\text{type.elem}; \]
\[ t = \text{new Temp}(); \]
\[ L.\text{addr} = \text{new Temp}(); \]
\[ \text{gen}(t \ ']=' \ E.\text{addr} \ '*' \ L.\text{type.width}); \]
\[ \text{gen}(L.\text{addr} \ ']=' \ L_1.\text{addr} \ '+' \ t); \ \text{\}} \]
Addressing Array Elements

c + a[i][j] ;

\[
\begin{align*}
t1 &= i \times 12 \\
t2 &= j \times 4 \\
t3 &= t1 + t2 \\
t4 &= a[t3] \\
t5 &= c + t4
\end{align*}
\]
\[ E.addr = t5 \]

\[ E.addr = c + E.addr = t4 \]

\[ L.array = a \]
\[ L.type = integer \]
\[ L.addr = t3 \]

\[ L.array = a \]
\[ L.type = array(3, integer) \]
\[ L.addr = t1 \]

\[ L.type = array(2, \]
\[ array(3, integer) \]

\[ E.addr = j \]
\[ j \]

\[ E.addr = i \]
\[ i \]