Intermediate Code Generation
Part I
Chapter 6

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

Compiler Front End

- Token stream
  - Parser
  - Static Checker
  - Intermediate Code Generator
  - IR

Lex specification
Yacc specification with semantic rules
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.\text{node} = \text{new Node}(\text{&quot;=&quot;}, \text{Leaf(id, id.entry)}, E.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 + E_2 )</td>
<td>( E.\text{node} = \text{new Node}(\text{&quot;+&quot;}, E_1.\text{node}, E_2.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 \ast E_2 )</td>
<td>( E.\text{node} = \text{new Node}(\text{&quot;\times&quot;}, E_1.\text{node}, E_2.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow - E_1 )</td>
<td>( E.\text{node} = \text{new Node}(\text{&quot;uminus&quot;}, E_1.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow ( E_1 ) )</td>
<td>( E.\text{node} = E_1.\text{node} )</td>
</tr>
<tr>
<td>( E \rightarrow \text{id} )</td>
<td>( E.\text{Node} = \text{new Leaf(id, id.entry)} )</td>
</tr>
</tbody>
</table>
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

\[ a = b^* - c + b^* - c \]

Tree

DAG

Directed Acyclic Graphs for Abstract Syntax Trees

\[ a + a * (b - c) + (b - c) * d \]
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>(<code>+</code>, $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T$</td>
<td>$E$.node = <code>new Node</code>(<code>-</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>$T \rightarrow \text{id}$</td>
<td>$T$.node = <code>new Leaf</code>($\text{id}$, $\text{id}$.entry)</td>
</tr>
<tr>
<td>$T \rightarrow \text{num}$</td>
<td>$T$.node = <code>new Leaf</code>($\text{num}$, $\text{num}$.val)</td>
</tr>
</tbody>
</table>

Value-Number Representation for DAGs

```
i = i + 10
```
**Value-Number Construction Method**

- Use signature \( <op, lc, rc> \) 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

\[
\begin{align*}
  x + y * z & \quad \Rightarrow \quad t_1 = y * z \\
  t_2 = x + t_1
\end{align*}
\]
Three-Address Code

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

\[
\begin{align*}
  t_1 &= -c \\
  t_2 &= b \cdot t1 \\
  t_3 &= -c \\
  t_4 &= b \cdot t3 \\
  t_5 &= t2 + t4 \\
  a &= t5
\end{align*}
\]

From abstract syntax tree  \hspace{1cm} From DAG representation of abstract syntax tree

Addresses and Labels

- In 3AC an \textit{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 flow of control
Three-Address Statements

- Assignment (binary): \( x = y \text{ op } z \)
- Assignment (unary): \( x = \text{ op } y \)
- Copy statement: \( x = y \)
- Unconditional jump: \textit{goto} lab

Three-Address Statements

- Conditional jump: \texttt{if} \( x \text{ goto} lab \), \texttt{ifFalse} \( x \text{ goto} lab \)
- Conditional jump: \texttt{if} \( x \text{ relop } y \text{ goto} lab \)
- Indexed assignment: \( x = y[i], x[i] = y \)
- Pointer assignment: \( x = \& y, x = * y, * x = y \)
Three-Address Statements

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

• Return: \text{return} \ x

• 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

<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_2</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

Quads (quadruples)

Pro: 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

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

<table>
<thead>
<tr>
<th>#</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>*</td>
<td>b</td>
<td>(0)</td>
</tr>
<tr>
<td>(2)</td>
<td>uminus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>*</td>
<td>b</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>+</td>
<td>(1)</td>
<td>(3)</td>
</tr>
<tr>
<td>(5)</td>
<td>=</td>
<td>a</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Pro: 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>uminus</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>uminus</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>

Pro: temporaries are implicit & easier to rearrange code

Type Expressions

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

```
int[2][3]
```
Graph Representations for Type Expressions

\[ \text{int } *f(\text{char*}, \text{char*}) \]

Tree forms

DAGs

Cyclic Graph Representations

```
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 by Pascal (inconsistently)

```plaintext
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.

```c
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

```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 runtime

\[
D \rightarrow T \text{id} ; D \mid \epsilon \\
T \rightarrow B \ C \mid \text{record } \{ D \} \\
B \rightarrow \text{int} \mid \text{float} \\
C \rightarrow \epsilon \mid \text{[ num ] } C
\]
Type Expression
and Storage Allocation

• Synthesized attribute \( type \) : type expression
• Synthesized attribute \( width \) : number of
  storage units needed for the type
• Inherited attributes \( t \) and \( w \) : used to pass
  type and width information to type
  constructors

\[
T \rightarrow B \quad \{ \ C.t = B.type; \quad C.w = B.width; \ \}
\]
\[
C \quad \{ \quad T.type = C.type; \quad T.width = C.width; \ \}
\]
\[
B \rightarrow \text{int} \quad \{ \quad B.type = \text{integer}; \quad B.width = 4; \ \}
\]
\[
B \rightarrow \text{float} \quad \{ \quad B.type = \text{float}; \quad B.width = 8; \ \}
\]
\[
C \rightarrow \varepsilon \quad \{ \quad C.type = C.t; \quad C.width = C.w; \ \}
\]
\[
C \rightarrow \left[ \ 	ext{num} \ \right] \quad \{ \quad C_1.t = C.t; \quad C_1.w = C.w; \ \}
C_1 \quad \{ \quad C.type = \text{array}(\text{num.value}, C_1.type);\quad
C.width = \text{num.value} \times C_1.width; \ \}
\]
Type Expression and Storage Allocation

Sequences of Declarations

- Augment previous grammar by accounting for sequences of declarations, distributed within someblock
- Use nonterminal $T$ as in previous SDT
- Use global variable $offset$ to keep track of next available relative address
Sequences of Declarations

\[ P \rightarrow \{ \text{offset} = 0; \} \quad D \]

\[ D \rightarrow T \text{ id} ; \quad \{ \top.put(\text{id}.\text{lexeme}, T.\text{type}, \text{offset}); \quad \text{offset} = \text{offset} + T.\text{width}; \} \quad D_1 \]

\[ D \rightarrow \epsilon \]

Fields in Records / Classes

• ...

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
  – *new Temp()* generates a fresh temporary name
## Translation of Expressions

### Productions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow \text{id} = E ; )</td>
<td>( S).code = ( E).code ( \parallel ) gen(top.get(id.lexeme) ‘=’ ( E).addr)</td>
</tr>
<tr>
<td>( E \rightarrow E_1 + E_2 )</td>
<td>( E).addr = \text{new Temp}(); ( E).code = ( E_1).code ( \parallel ) ( E_2).code ( \parallel ) gen(E.addr ‘=’ ( E_1).addr ‘+’ ( E_2).addr)</td>
</tr>
<tr>
<td>( E \rightarrow E_1 \times E_2 )</td>
<td>( E).addr = \text{new Temp}(); ( E).code = ( E_1).code ( \parallel ) ( E_2).code ( \parallel ) gen(E.addr ‘=’ ( E_1).addr ‘*’ ( E_2).addr)</td>
</tr>
<tr>
<td>( E \rightarrow - E_1 )</td>
<td>( E).addr = \text{new Temp}(); ( E).code = ( E_1).code ( \parallel ) gen(E.addr ‘=’ ( E_1).addr ‘-’ ( E).addr)</td>
</tr>
<tr>
<td>( E \rightarrow ( E_1 ) )</td>
<td>( E).addr = ( E_1).addr ( E).code = ( E_1).code</td>
</tr>
<tr>
<td>( E \rightarrow \text{id} )</td>
<td>( E).addr = top.get(id.lexeme) ( E).code = ‘’</td>
</tr>
</tbody>
</table>

### Example:

Translation of Expressions

\[
a = b + - c ;
\]

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

```
Example:
Translation of Expressions

addr = c
code = ''

addr = t1
code = 't1 = minus c
t2 = b + t1
a = t2'

addr = t2
code = 't1 = minus c
t2 = b + t1'

addr = t1
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>$S \rightarrow id = E;$</td>
<td>$gen(top.get(id.lexeme) '=' E.addr)$</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 + E_2$ | $E.addr = \text{new Temp}();$  
| | $gen(E.addr '==' E_1.addr ' +' E_2.addr)$ |
| $E \rightarrow E_1 * E_2$ | $E.addr = \text{new Temp}();$  
| | $gen(E.addr '==' E_1.addr '* E_2.addr)$ |
| $E \rightarrow - E_1$ | $E.addr = \text{new Temp}();$  
| | $gen(E.addr '==' 'minus' E_1.addr)$ |
| $E \rightarrow ( E_1 )$ | $E.addr = E_1.addr$ |
| $E \rightarrow id$ | $E.addr = top.get(id.lexeme)$ |

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

\[ a[3][5]: \]

array base address \( c(a) \)

row 0  row 1  row 2

\[ a[1][0] \]

Addressing Array Elements

• Row-major order vs. column-major order

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

\begin{array}{|c|c|}
\hline
\text{c(a)} & \text{c(a)} \\
\hline
a[0][0] & a[0][0] \\
a[0][1] & a[1][0] \\
a[0][2] & a[0][1] \\
a[1][0] & a[1][1] \\
a[1][1] & a[0][2] \\
a[1][2] & a[1][2] \\
\hline
\end{array}

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
  – $a_{[i_1, \ldots, i_k]}$ located at
    \[
    c(a) + \\
    (( \ldots (i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots ) \times n_k + i_k \times w
    \]

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

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

• Other addressing conventions (two dimensions)

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

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

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 * d1
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

```
S \rightarrow L = E  \quad L \rightarrow Elist [ ]
E \rightarrow E + E  \quad L \rightarrow id
E \rightarrow ( E )  \quad Elist \rightarrow Elist , E
E \rightarrow L       \quad Elist \rightarrow id [ E
```
Addressing Array Elements

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

**Productions**

<table>
<thead>
<tr>
<th>$S \rightarrow L = E$</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + E_2$</td>
<td>if ($L.offset == null$)</td>
</tr>
<tr>
<td></td>
<td>\hspace{1em} gen($L.place \leftarrow E.place$);</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>\hspace{1em} gen($L.place \leftarrow L.offset \leftarrow E.place$);</td>
</tr>
<tr>
<td>$E \rightarrow (E)$</td>
<td>$E.place = newlabel();$</td>
</tr>
<tr>
<td></td>
<td>gen($E.place \leftarrow E_1.place + E_2.place$);</td>
</tr>
</tbody>
</table>

**Productions**

<table>
<thead>
<tr>
<th>$E \rightarrow L$</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow Elist$</td>
<td>if ($L.offset == null$)</td>
</tr>
<tr>
<td></td>
<td>\hspace{1em} $E.place = L.place$;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>\hspace{1em} { $E.place = newlabel()$;</td>
</tr>
<tr>
<td></td>
<td>\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} gen($E.place \leftarrow L.place$); }</td>
</tr>
<tr>
<td>$L \rightarrow Elist$</td>
<td>$L.place = newlabel();$</td>
</tr>
<tr>
<td></td>
<td>$L.offset = newlabel();$</td>
</tr>
<tr>
<td></td>
<td>gen($L.place \leftarrow c(Elist.array)$);</td>
</tr>
<tr>
<td></td>
<td>gen($L.offset \leftarrow Elist.place \leftarrow width(Elist.array)$);</td>
</tr>
</tbody>
</table>
## Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$L.place = \text{id}.place$; $L.offset = \text{null}$; $t = \text{newlabel}()$; $m = Elist_1.dim + 1$; $\text{gen}(t \text{ <code>=' } Elist_1\.place \text{ </code>*<code>limit(Elist}_1\.array, m))$; $\text{gen}(t \text{</code>=' } t \text{ `+' } E.place)$; $Elist.array = Elist_1.array$; $Elist.place = t$; $Elist.dim = m$;</td>
</tr>
<tr>
<td>$Elist \rightarrow Elist_1, E$</td>
<td></td>
</tr>
</tbody>
</table>

---

## Addressing Array Elements

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Elist \rightarrow \text{id} [ E$</td>
<td>$Elist.array = \text{id}.place$; $Elist.place = E.place$; $Elist.dim = 1$;</td>
</tr>
</tbody>
</table>

---
Addressing Array Elements

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

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

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

\[ L \to L_1[E] \]
\[
\{ \text{L.array} = L_1.array; \\
\text{L.type} = L_1.type.elem; \\
\text{t} = \text{new Temp}(); \\
\text{L.addr} = \text{new Temp}(); \\
\text{gen(t} \times E.addr \times L.type.width); \\
\text{gen(L.addr} \times L_1.addr + t); } \}
\]

Addressing Array Elements

\[
c + a[i][j];
\]
\[
t_1 = i \times 12 \\
t_2 = j \times 4 \\
t_3 = t_1 + t_2 \\
t_4 = a[t_3] \\
t_5 = c + t_4
\]
\[ E.addr = \text{t5} \]
\[ E.addr = \text{c} \]
\[ E.addr = \text{t4} \]
\[ L.array = \text{a} \]
\[ L.type = \text{integer} \]
\[ L.addr = \text{t3} \]
\[ E.addr = \text{j} \]
\[ L.array = \text{a} \]
\[ L.type = \text{array(3, integer)} \]
\[ L.addr = \text{t1} \]
\[ E.addr = \text{i} \]
\[ L.type = \text{array(2, array(3, integer))} \]