G22.2130-001
Compiler Construction
Lecture 9:
Intermediate-Code Generation

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
Back-end and Front-end of A Compiler
Back-end and Front-end of A Compiler

Close to source language (e.g. syntax tree)               Close to machine language

m x n compilers can be built by writing just m front ends and n back ends
Back-end and Front-end of A Compiler

Includes:

- Type checking
- Any syntactic checks that remain after parsing (e.g. ensure `break` statement is enclosed within `while`, `for`, or `switch` statements).
Syntax Tree

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E\cdot node = \text{new Node}('+$, $E_1\cdot node, T\cdot node)$</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E\cdot node = \text{new Node}('-$', $E_1\cdot node, T\cdot node)$</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E\cdot node = T\cdot node$</td>
</tr>
<tr>
<td>4) $T \rightarrow (E)$</td>
<td>$T\cdot node = E\cdot node$</td>
</tr>
<tr>
<td>5) $T \rightarrow \text{id}$</td>
<td>$T\cdot node = \text{new Leaf}($id, id.entry)$</td>
</tr>
<tr>
<td>6) $T \rightarrow \text{num}$</td>
<td>$T\cdot node = \text{new Leaf}(\text{num, num.val})$</td>
</tr>
</tbody>
</table>

$$a + a \cdot (b - c) + (b - c) \cdot d$$
## Syntax Tree

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E$.node = new Node('+', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E$.node = new Node('-', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>4) $T \rightarrow ( E )$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>5) $T \rightarrow \text{id}$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>6) $T \rightarrow \text{num}$</td>
<td>$T$.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>

\[
a + a * (b - c) + (b - c) * d
\]
Syntax Tree

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E$.node = new Node('+', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E$.node = new Node('-', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>4) $T \rightarrow ( E )$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>5) $T \rightarrow \text{id}$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>6) $T \rightarrow \text{num}$</td>
<td>$T$.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>

$a + a * (b - c) + (b - c) * d$
Syntax Tree

Directed Acyclic Graph (DAG):
- More compact representation
- Gives clues regarding generation of efficient code

Node can have more than one parent
Example

Construct the DAG for:

\[((x + y) - ((x + y) \times (x - y))) + ((x + y) \times (x - y))\]
How to Generate DAG from Syntax-Directed Definition?

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E$.node = new Node('+', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E$.node = new Node('-', $E_1$.node, $T$.node)</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E$.node = $T$.node</td>
</tr>
<tr>
<td>4) $T \rightarrow (E)$</td>
<td>$T$.node = $E$.node</td>
</tr>
<tr>
<td>5) $T \rightarrow id$</td>
<td>$T$.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>6) $T \rightarrow num$</td>
<td>$T$.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>

All what is needed is that functions such as `Node` and `Leaf` above check whether a node already exists. If such a node exists, a pointer is returned to that node.
How to Generate DAG from Syntax-Directed Definition?

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $E \rightarrow E_1 + T$</td>
<td>$E.n\text{ode} = \text{new } \text{Node}(+', E_1.n\text{ode}, T.n\text{ode})$</td>
</tr>
<tr>
<td>2) $E \rightarrow E_1 - T$</td>
<td>$E.n\text{ode} = \text{new } \text{Node}(-', E_1.n\text{ode}, T.n\text{ode})$</td>
</tr>
<tr>
<td>3) $E \rightarrow T$</td>
<td>$E.n\text{ode} = T.n\text{ode}$</td>
</tr>
<tr>
<td>4) $T \rightarrow (\ E\ )$</td>
<td>$T.n\text{ode} = E.n\text{ode}$</td>
</tr>
<tr>
<td>5) $T \rightarrow \text{id}$</td>
<td>$T.n\text{ode} = \text{new } \text{Leaf(id, id.entry)}$</td>
</tr>
<tr>
<td>6) $T \rightarrow \text{num}$</td>
<td>$T.n\text{ode} = \text{new } \text{Leaf(num, num.val)}$</td>
</tr>
</tbody>
</table>

1) $p_1 = \text{Leaf(id, entry-a)}$
2) $p_2 = \text{Leaf(id, entry-a)} = p_1$
3) $p_3 = \text{Leaf(id, entry-b)}$
4) $p_4 = \text{Leaf(id, entry-c)}$
5) $p_5 = \text{Node}(-', p_3, p_4)$
6) $p_6 = \text{Node}(\ast', p_1, p_5)$
7) $p_7 = \text{Node}(+', p_1, p_6)$
8) $p_8 = \text{Leaf(id, entry-b)} = p_3$
9) $p_9 = \text{Leaf(id, entry-c)} = p_4$
10) $p_{10} = \text{Node}(-', p_3, p_4) = p_5$
11) $p_{11} = \text{Leaf(id, entry-d)}$
12) $p_{12} = \text{Node}(\ast', p_5, p_{11})$
13) $p_{13} = \text{Node}(+', p_7, p_{12})$

$$a + a \ast (b - c) + (b - c) \ast d$$
Data Structure: Array

```
 id | num | 10 |
---+-----+----|
 i  | +   | 10 |
```

to entry for i

```
i 12  
10 |
```
Data Structure: Array

Scanning the array each time a new node is needed, is not an efficient thing to do.
Data Structure: Hash Table

Hash function = h(op, L, R)
Three-Address Code

- Another option for intermediate presentation
- Built from two concepts:
  - addresses and instructions
- At most one operator

```
\begin{align*}
  t_1 &= b - c \\
  t_2 &= a \times t_1 \\
  t_3 &= a + t_2 \\
  t_4 &= t_1 \times d \\
  t_5 &= t_3 + t_4
\end{align*}
```
Address

Can be one of the following:
• A name: source program name
• A constant
• Compiler-generated temporary
Instructions

Assignment instructions of the form \( x = y \ op \ z \)

Assignments of the form \( x = \ op \ y \)

Copy instructions of the form \( x = y \)

An unconditional jump goto \( L \)

Conditional jumps of the form if \( x \) goto \( L \) and ifFalse \( x \) goto \( L \)

Conditional jumps such as if \( x \ \text{relop} \ y \) goto \( L \)

Procedure call such as p(x1, x2, ..., xn) is implemented as:

\[
\begin{align*}
\text{param } x_1 \\
\text{param } x_2 \\
\text{...} \\
\text{param } x_n \\
\text{call } p, n
\end{align*}
\]

Indexed copy instructions of the form \( x=y[i] \) and \( x[i]=y \).

Address and pointer assignments of the form \( x = \& y \), \( x = *y \), and \( *x = y \)
Example

do i = i+1; while (a[i] < v);

L: \[ t_1 = i + 1 \]
   \[ i = t_1 \]
   \[ t_2 = i \times 8 \]
   \[ t_3 = a[t_2] \]
   if \( t_3 < v \) goto L

100: \[ t_1 = i + 1 \]
101: \[ i = t_1 \]
102: \[ t_2 = i \times 8 \]
103: \[ t_3 = a[t_2] \]
104: if \( t_3 < v \) goto 100
Choice of Operator Set

• Rich enough to implement the operations of the source language
• Close enough to machine instructions to simplify code generation
Data Structure

How to present these instructions in a data structure?

– Quadruples
– Triples
– Indirect triples
Data Structure: Quadruples

- Has four fields: op, arg1, arg2, result
- Exceptions:
  - Unary operators: no arg2
  - Operators like param: no arg2, no result
  - (Un)conditional jumps: target label is the result
\[ t_1 = \text{minus } c \]
\[ t_2 = b \times t_1 \]
\[ t_3 = \text{minus } c \]
\[ t_4 = b \times t_3 \]
\[ t_5 = t_2 + t_4 \]
\[ a = t_5 \]

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>c</td>
<td>t_1</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>b</td>
<td>t_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t_2</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>c</td>
<td>t_3</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>b</td>
<td>t_3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t_4</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>t_2</td>
<td>t_4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>t_5</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>t_5</td>
<td>a</td>
</tr>
</tbody>
</table>

...
Data Structure: Triples

- Only three fields: no result field
- Results referred to by its position

(a) Syntax tree
(b) Triples

Representations of $a + a \times (b - c) + (b - c) \times d$
Data Structure: Indirect Triples

- When instructions are moving around during optimizations: quadruples are better than triples.
- Indirect triples solve this problem

<table>
<thead>
<tr>
<th>instruction</th>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>minus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List of pointers to triples

Optimizing compiler can reorder instruction list, instead of affecting the triples themselves
Single-Static-Assignment (SSA)

- Is an intermediate presentation
- Facilitates certain code optimizations
- All assignments are to variables with distinct names

\[
\begin{align*}
\text{(a) Three-address code.} & \quad \text{(b) Static single-assignment form.} \\
\text{p} &= a + b \\
\text{q} &= p - c \\
\text{p} &= q \times d \\
\text{p} &= e - p \\
\text{q} &= p + q \\
\text{p}_1 &= a + b \\
\text{q}_1 &= p_1 - c \\
\text{p}_2 &= q_1 \times d \\
\text{p}_3 &= e - p_2 \\
\text{q}_2 &= p_3 + q_1
\end{align*}
\]
Single-Static-Assignment (SSA)

Example:

```java
if ( flag ) x = -1; else x = 1;
y = x * a;
```

If we use different names for X in true part and false part, then which name shall we use in the assignment of \( y = x \times a \) ?

The answer is: \( \emptyset \)-function

```java
if ( flag ) x₁ = -1; else x₂ = 1;
x₃ = \( \emptyset \)(x₁, x₂);
```

Returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the \( \emptyset \)-function
Example

Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.
b) Quadruples.
c) Triples.
d) Indirect triples.
Types and Declarations

• Type checking: to ensure that types of operands match the type expected by operator
• Determine the storage needed
• Calculate the address of an array reference
• Insert explicit type conversion
• Choose the right version of an operator
• ...
Storage Layout

- From the type, we can determine amount of storage at run time.
- At compile time, we will use this amount to assign its name a relative address.
- Type and relative address are saved in the symbol table entry of the name.
- Data with length determined only at run time saves a pointer in the symbol table.
• Multibyte objects are stored in consecutive bytes and given the address of the first byte
• Storage for aggregates (e.g. arrays and classes) is allocated in one contiguous block of bytes.
\[
T \rightarrow B \\
\quad C \\
B \rightarrow \text{int} \\
\quad \{ B.\text{type} = \text{integer}; B.\text{width} = 4; \} \\
B \rightarrow \text{float} \\
\quad \{ B.\text{type} = \text{float}; B.\text{width} = 8; \} \\
C \rightarrow e \\
\quad \{ C.\text{type} = t; C.\text{width} = w; \} \\
C \rightarrow [\text{num}]C_1 \\
\quad \{ \text{array}(\text{num}.\text{value}, C_1.\text{type}); C.\text{width} = \text{num}.\text{value} \times C_1.\text{width}; \} \\
\]

\text{int}[2][3]
\[ P \rightarrow D \{ \text{offset} = 0; \} \]

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

\[ D_1 \]

\[ D \rightarrow \epsilon \]
To keep track of the next available relative address

Create a symbol table entry

\[
P \rightarrow \ D
\]

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

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

\[
D \rightarrow \epsilon
\]
Translations of Statements and Expressions

Syntax-Directed Definition (SDD)  Syntax-Directed Translation (SDT)
<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 )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 + E_2 )</td>
<td>( E.addr = \text{new Temp}() )</td>
</tr>
<tr>
<td></td>
<td>( E.code = E_1.code )</td>
</tr>
<tr>
<td>| ( - E_1 )</td>
<td>( E.addr = \text{new Temp}() )</td>
</tr>
<tr>
<td></td>
<td>( E.code = E_1.code )</td>
</tr>
<tr>
<td>| ( (E_1) )</td>
<td>( E.addr = E_1.addr )</td>
</tr>
<tr>
<td></td>
<td>( E.code = E_1.code )</td>
</tr>
<tr>
<td>| ( \text{id} )</td>
<td>( E.addr = \text{top.get(id.lexeme)} )</td>
</tr>
<tr>
<td></td>
<td>( E.code = '' )</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>SEMANTIC RULES</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>S → id = E ;</td>
<td>S.code = E.code</td>
</tr>
</tbody>
</table>
| E → E₁ + E₂ | E.addr = new Temp ()
| | E.code = E₁.code || E₂.code || gen(E.addr '=' E₁.addr +' ' E₂.addr) |
| | - E₁ | E.addr = new Temp ()
| | E.code = E₁.code ||
| | | gen(E.addr ']=' 'minus' E₁.addr) |
| | ( E₁ ) | E.addr = E₁.addr
| | E.code = E₁.code |
| | id | E.addr = top.get(id.lexeme)
<p>| | E.code = '' |</p>
<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| $S \rightarrow \text{id} = E$ | $\text{S.code} = \text{E.code} \mid$
  \hspace{2em} \text{gen(top.get(id.lexeme)} \text{'}=\text{' E.addr}$ |
| $E \rightarrow E_1 + E_2$ | $\text{E.addr} = \text{new Temp}()$
  \hspace{2em} $\text{E.code} = \text{E}_1.\text{code} \mid \text{E}_2.\text{code} \mid$
  \hspace{4em} $\text{gen(E.addr} \text{'}=\text{' E}_1.\text{addr} \text{'}+\text{' E}_2.\text{addr}$ |
| $E \rightarrow -E_1$ | $\text{E.addr} = \text{new Temp}()$
  \hspace{2em} $\text{E.code} = \text{E}_1.\text{code} \mid$
  \hspace{4em} $\text{gen(E.addr} \text{'}=\text{' minus' E}_1.\text{addr}$ |
| $E \rightarrow (E_1)$ | $\text{E.addr} = \text{E}_1.\text{addr}$
  \hspace{2em} $\text{E.code} = \text{E}_1.\text{code}$ |
| $E \rightarrow \text{id}$ | $\text{E.addr} = \text{top.get(id.lexeme)}$
  \hspace{2em} $\text{E.code} = \text{'}$ |

\[ a = b + -c \]

\[ t_1 = \text{minus c} \]
\[ t_2 = b + t_1 \]
\[ a = t_2 \]
### Producing three-address code incrementally to avoid long string manipulations

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| \( S \to \text{id} = E \) | \( S.code = E.code \mid \)  
  \( \quad \text{gen}(\text{top.get(id.lexeme)} = E.addr) \) |
| \( E \to E_1 + E_2 \) | \( E.addr = \text{new Temp}() \)  
  \( E.code = E_1.code || E_2.code \mid \)  
  \( \quad \text{gen}(E.addr = E_1.addr + E_2.addr) \) |
| \( - E_1 \) | \( E.addr = \text{new Temp}() \)  
  \( E.code = E_1.code \mid \)  
  \( \quad \text{gen}(E.addr = \text{`minus' } E_1.addr) \) |
| \( ( E_1 ) \) | \( E.addr = E_1.addr \)  
  \( E.code = E_1.code \) |
| \( \text{id} \) | \( E.addr = \text{top.get(id.lexeme)} \)  
  \( E.code = \'' \) |

---

**gen()** does two things:
- generate three-address instruction
- append it to the sequence of instructions generated so far

\[
\begin{align*}
S & \to \text{id} = E ; & \{ \text{gen}(\text{top.get(id.lexeme)} = E.addr) ; \} \\
E & \to E_1 + E_2 & \{ E.addr = \text{new Temp}() ; \}
  & \text{gen}(E.addr = E_1.addr + E_2.addr) ; \}
  | - E_1 & \{ E.addr = \text{new Temp}() ; \}
  & \text{gen}(E.addr = \text{`minus' } E_1.addr) ; \}
  | ( E_1 ) & \{ E.addr = E_1.addr ; \}
  | \text{id} & \{ E.addr = \text{top.get(id.lexeme)} ; \}
\end{align*}
\]
Arrays

- Elements of the same type
- Stored consecutively in memory
- In languages like C or Java elements are: 0, 1, ..., n-1
- In some other languages: low, low+1, ..., high
Arrays

If elements start with 0, and element width is \( w \), then \( a[i] \) address is: 

\[
\text{base} + i \times w
\]

base is address of \( A[0] \)

Generalizing to two-dimensions \( a[i1][i2] \):

\( w_1 \) is width of a row and \( w_2 \) the width of an element 

\[
\text{base} + i_1 \times w_1 + i_2 \times w_2
\]

or

\[
\text{base} + (i_1 \times n_2 + i_2) \times w
\]

w is width of an element, \( n_2 \) is number of elements per row

Generalizing to k-dimensions:

\[
\text{base} + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k
\]

or

\[
\text{base} + ((\cdots (i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w
\]
\[ S \to \text{id} = E ; \quad \{ \text{gen}(\text{top.get(id.lexeme)} \,'=\,' \ E.\text{addr}); \} \]

| \[ L = E ; \quad \{ \text{gen}(L.\text{addr}.\text{base} \,[\,' L.\text{addr} \,]' \,'=\,' \ E.\text{addr}); \} \]

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

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

| \[ L \quad \{ E.\text{addr} = \text{new} \ \text{Temp}(); \]
\[ \quad \text{gen}(E.\text{addr} \,'=\,' \ L.\text{array}.\text{base} \,[\,' L.\text{addr} \,]'); \} \]

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

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

**Temporary used while computing the offset**

**Pointer to symbol table entry**
A is a 2x3 array of integers

t_1 = i * 12

t_2 = j * 4

t_3 = t_1 + t_2

t_4 = a [ t_3 ]

t_5 = c + t_4
So...

- Skim: 6.3.1, 6.3.2
- Read: Beginning of chapter 6 -> 6.4