Intermediate Code Generation

Basic Approach and Application to Assignment and Expressions

Array Expressions

Boolean Expressions

Copyright 2016, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in Code Generation and Optimization
- For super-scalars, it's Allocation & Scheduling that counts
Intermediate Code Generation

- **Direct Translation**
  - Using SDT scheme
  - Parse Tree to Three-Address Instructions
  - Can be done while Parsing in a Single Pass
  - Needs to be able to deal with Syntactic Errors and Recovery

- **Indirect Translation**
  - First validate parsing constructing of AST
  - Uses SDT scheme to build AST
  - Traverse the AST and generate Three Address Instructions

\[
\text{Parse tree} \xrightarrow{\text{Direct Translation}} \text{Intermediate Code Generation} \xrightarrow{\text{Indirect Translation}} \text{Three-Address Instructions}
\]

\[O(n)\]
Three-Address Instructions IR

• High-level Constructs mapped to Three-Address Instructions
  – Register-based IR for Expression Evaluation
  – Infinite Number of Virtual Registers
  – Still Independent of Target Architecture
  – Parameter Passing Discipline either on Stack or via Registers

• Addresses and Instructions
  – Symbolic Names are addresses of the corresponding source-level variable.
  – Various constants, such as numeric and offsets (known at compile time)

• Generic Instruction Format:
  Label: \( x = y \) op \( z \) or \( \text{if exp goto L} \)
  – Statements can have Symbolic Labels
  – Compiler inserts Temporary Variables (any variable with \( t \) prefix)
  – Type and Conversions dealt in other Phases of the Code Generation
Three-Address Instructions

• Assignments:
  - \( x = y \text{ op } z \) (binary operator)
  - \( x = \text{ op } y \) (unary)
  - \( x = y \) (copy)
  - \( x = y[i] \text{ and } x[i] = y \) (array indexing assignments)
  - \( x = \phi y z \) (Static Single Assignment instruction)

• Memory Operations:
  - \( x = \& y; x = *y \text{ and } *x = y \); for assignments via pointer variables.
Three-Address Instructions

• Control Transfer and Function Calls:
  - \texttt{goto L} (unconditional);
  - \texttt{if (a relop b) goto L} (conditional) where \texttt{relop} is a relational operator consistent with the type of the variables \texttt{a} and \texttt{b};
  - \texttt{y = call p, n} for a function or procedure call instruction to the name or variable \texttt{p}
    • \texttt{p} might be a variable holding a set of possible symbolic names (a function pointer)
    • the value \texttt{n} specifies that before this call there were \texttt{n} \texttt{putparam} instructions to load the values of the arguments.
    • the \texttt{param x} instruction specifies a specific value in reverse order (i.e, the \texttt{param} instruction closest to the call is the first argument value.
    • Later we will talk about parameter passing disciplines (Run-Time Env.)
Function Call Example

Source Code

```
y = p(a, b+1)

int p(x,z){
    return x+z;
}
```

Three Address Instructions

```
t1 = a

putparam t1

putparam t2

y = call p, 2

p: getparam z

getparam x

t3 = x + z

return t3
```
Function Call Example

Source Code

```c
int p(x, z) {
    return x + z;
}

y = p(a, b+1)
```

Three Address Instructions

```c
y = call p, 2
p: getparam z
getparam x
t3 = x + z
return t3
```

t1 = a

\{
  t2 = b + 1
  putparam t1
  putparam t2
  y = call p, 2
  p: getparam z
  getparam x
t3 = x + z
  return t3
\}

argument evaluation

passing args on the stack

getting values from the stack
Loop Example

Source Code

\[
\text{do} \\
\text{\hspace{1em}} i = i + 1; \\
\text{while} \ (a[i] < v); 
\]

Three Address Instructions

\[
L: \ t1 = i + 1 \\
\text{\hspace{1em}} i = t1 \\
\text{\hspace{1em}} t2 = i \times 8 \\
\text{\hspace{1em}} t3 = a[t2] \\
\text{\hspace{1em}} \text{if} \ t3 < v \ \text{goto} \ L 
\]
Loop Example

Source Code

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

Three Address Instructions

```c
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v goto L
```

Where did this come from?
SDT for Three Address Code Generation

• Attributes for the Non-Terminals, E and S
  – Location (in terms of temporary variable) of the value of an expression: E.place. If E.place is t1 it means that the value of E is saved in t1.
  – The Code that Evaluates the Expressions or Statement: E.code
  – Markers for beginning and end of sections of the code S.begin, S.end
    • For simplicity these are symbolic labels.

• Semantic Actions in Productions of the Grammar
  – Functions to create temporaries newtemp, and labels newlabel
  – Auxiliary functions to enter symbols and lookup types corresponding to declarations in a symbol table.
  – To generate the code we use the function gen which creates a list of instructions to be emitted later and can generate symbolic labels corresponding to next instruction of a list.
  – Use of append function on lists of instructions.
  – Make use of Synthesized and Inherited Attributes
Assignment Statements

S → id = E  { p = lookup(id.name);
       if (p != NULL){
           S.code = gen(p ‘=’ E.place);
       } else }
       error;
       S.code = nulllist;
    }

E → E₁ + E₂  { E.place = newtemp();
           E.code = append(E₁.code,E₂.code,
                           gen(E.place‘=’ E₁.place ‘+’ E₂.place));
         }

E → E₁ * E₂  { E.place = newtemp();
           E.code = append(E₁.code,E₂.code,
                           gen(E.place ‘=’ E₁.place ‘*’ E₂.place));
         }

E → - E₁     { E.place = newtemp();
           E.code = append(E₁.code,gen(E.place ‘=’ ‘-’ E₁.place)); }

E → (E₁)     { E.place = E₁.place; E.code = E₁.code; }

E → id       { p = lookup(id.name);
              if (p != NULL)
                  E.place = p;
              else
                  error;
              E.code = nulllist;
           }
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[ E \rightarrow \text{id} \quad \{ \text{p} = \text{lookup(id.name)}; \]
\[ \text{if} \ (p \neq \text{NULL}) \]
\[ \text{E.place} = p; \]
\[ \text{else} \]
\[ \text{error}; \]
\[ \text{E.code} = \text{null list}; \}\]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

```
Production:

E → id { p = lookup(id.name); if (p != NULL) E.place = p; else error; E.code = null list; }
   | place = loc(f)
   | code = null
   | place = loc(e)
   | code = null

Assignment: Example
```
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[ E \rightarrow E_1 \times E_2 \{ E\.place = \text{newtemp}(); \]
\[ \quad E\.code = \text{gen}(E\.place = 'E_1\.place ' \times ' E_2\.place); \} \]

\[ E \rightarrow E + E \]
\[ \quad place = \text{loc}(t1) \]
\[ \quad code = \{ t1 = e \times f; \} \]

\[ E \rightarrow E - E \]
\[ \quad place = \text{loc}(t1) \]
\[ \quad code = \{ t1 = e \times f; \} \]

\[ x = \text{id} = E \times \text{id} + E \times \text{id} \]
\[ \quad place = \text{loc}(t1) \]
\[ \quad code = \{ t1 = e \times f; \} \]

\[ x = \text{id} = \text{id} \times \text{id} - \text{id} \times \text{id} \]
\[ \quad place = \text{loc}(t1) \]
\[ \quad code = \{ t1 = e \times f; \} \]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

Production:

\[ E \rightarrow E_1 \times E_2 \quad \{ E.\text{place} = \text{newtemp}(); \} \]
\[ \quad E.\text{code} = \text{gen}(E.\text{place} = \text{'='} \ E_1.\text{place} \ \text{'\times'} \ E_2.\text{place}); \]
Assignment: Example

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]

\[
x = a \ast b + c \ast d - e \ast f;
\]

S

\[
\text{code} = \{t1 = e \ast f; t2 = c \ast d; t3 = t2 - t1; t4 = a \ast b; t5 = t4 + t3; x = t5; \}
\]

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]

place = loc(x)

code = null

\[
x \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E}
\]

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]

place = loc(5)

\[
\text{code} = \{t1 = e \ast f; t2 = c \ast d; t3 = t2 - t1; t4 = a \ast b; t5 = t4 + t3; x = t5; \}
\]

place = loc(4)

\[
\text{E} \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E}
\]

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]

place = loc(3)

\[
\text{E} \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E}
\]

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]

place = loc(2)

\[
\text{E} \quad \text{E} \quad \text{E} \quad \text{E} \quad \text{E}
\]

Production:

\[
S \rightarrow id = E \quad \{ \ p = \text{lookup}(id.\text{name}); \\
\text{if} \ (p \neq \text{NULL}) \\
\quad E.\text{code} = \text{append}(E.\text{code}, \\
\quad \text{gen}(p.‘=‘ E.\text{place})); \\
\text{else} \\
\quad \text{error}; \\
\}\}
\]
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

\[ t1 = e \times f; \]
\[ t2 = c \times d; \]
\[ t3 = t2 - t1; \]
\[ t4 = a \times b; \]
\[ t5 = t4 + t3; \]
\[ x = t5; \]
Reusing Temporary Variables

- Temporary Variables
  - Short lived
  - Used for Evaluation of Expressions
  - Clutter the Symbol Table

- Change the `newtemp` Function
  - Keep track of when a value created in a temporary is used
  - Use a counter to keep track of the number of active temporaries
  - When a temporary is used in an expression decrement counter
  - When a temporary is generated by `newtemp` increment counter
  - Initialize counter to zero (0)

- Alternatively, can be done as a post-processing pass…
Assignment: Example

\[ x = a \times b + c \times d - e \times f; \]

- Only 2 Temporary Variables and hence only 2 Registers are Needed

\[
\begin{align*}
& t1 = e \times f; \quad // c = 0 \\
& t2 = c \times d; \quad // c = 1 \\
& t1 = t2 - t1; \quad // c = 1 \\
& t2 = a \times b; \quad // c = 2 \\
& t1 = t2 + t1; \quad // c = 1 \\
& x = t1; \quad // c = 0
\end{align*}
\]
Code Generation for Array Accesses

- **Questions:**
  - What is the Base Type of A?
  - What are the Dimensions of A?
  - What is the “layout” of A?

- **Where to Get Answers?**
  - Use Symbol Table
  - Check Array Layout

```plaintext
S = E
L = E
id
x
Elist
id
E
A
L
id
y

???
```

```plaintext

t1 = y * 20

t1 = t1 * z

t2 = baseA - 84

t3 = 4 * t1

t4 = t2[t3]

X = t4
```
How does the Compiler Handle \( A[i,j] \)?

First, we must agree on a Storage Scheme or Layout

**Row-major order**

- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- \( A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3] \)

**Column-major order**

- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- \( A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3] \)

**Indirection vectors**

- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis
Laying Out Arrays

The Concept

Row-major order

Column-major order

Indirection vectors

These have distinct & different cache behavior
Accessing an Array Element

How to “access” A[ i ]?

- Computing the Address of a 1-D Array Element:
  - baseA + (i - low) x sizeof(baseType(A))

- In General:
  - base(A) + (i - low) x sizeof(baseType(A))
Computing an Array Address

Computing the Address of $A[i]$: 

- **Computing the Address of a 1-D Array Element:** 
  $$\text{baseA} + (i - \text{low}) \times \text{sizeof(baseType(A))}$$

- **In General:** 
  $$\text{base}(A) + (i - \text{low}) \times \text{sizeof(baseType(A))}$$

Almost always a power of 2, known at compile-time and use a shift instruction for speed

int $A[1:10]$ where low is 1; Make low 0 for faster access (saves a $-$) as in the C language

Defined by the storage class of the array.
Computing an Array Address

A[i]
• baseA + (i – low) x sizeof(baseType(A))
• In general: base(A) + (i – low) x sizeof(baseType(A))

What about A[i_1,i_2]?

Row-major order, two dimensions

baseA + ((i_1 – low_1) x (high_2 – low_2 + 1) + i_2 – low_2) x sizeof(baseType(A))

Column-major order, two dimensions

baseA + ((i_2 – low_2) x (high_1 – low_1 + 1) + i_1 – low_1) x sizeof(baseType(A))

Indirection vectors, two dimensions

*(A[i_1])[i_2] — where A[i_1] is, itself, a 1-d array reference

This stuff looks expensive! Lots of implicit +, -, x ops
Optimizing Address Calculation for A[i,j]

In row-major order

\[ \text{baseA} + (i - \text{low}_1)(\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w \]

Which can be factored into

\[ \text{baseA} + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w \]
\[ - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w) \]

If \( \text{low}_i, \text{high}_i, \) and \( w \) are known, the last term is a constant

Define \( \text{baseA}_0 \) as

\[ \text{baseA} - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1)) \times w + \text{low}_2 \times w \]

and \( \text{len}_2 \) as \( (\text{high}_2 - \text{low}_2 + 1) \)

Then, the address expression becomes

\[ \text{baseA}_0 + (i \times \text{len}_2 + j) \times w \]
Address Calculation for $A[i_1, i_2, \ldots , i_k]$

- $A[i_1, i_2, \ldots , i_k]$

Addressing generalizes to

$$((\ldots((i_1 n_2 + i_2) n_3 + i_3)\ldots)n_k + i_k) \times w +$$

$$+ \text{baseA} - ((\ldots((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3)\ldots)m_k + \text{low}_k) \times w$$

where $n_j = \text{high}_j - \text{low}_j + 1$ and $w = \text{sizeof}(\text{baseType}(A))$

First term can be computed using the recurrence

$$e_1 = i_1$$

$$e_m = e_{m-1} \times n_m + i_m$$

at the end multiply by $w$ and add compile-time constant term
SDT for Addressing Arrays Elements

• Three Attributes
  – place: just the name or base address of the array
  – offset: the index value into the array
  – ndim: the number of dimensions

• Use the Recurrence to Compute Offset
  
  \[ \text{offset}_1 = i_1 \]
  
  \[ \text{offset}_m = \text{offset}_{m-1} \times \text{n}_m + i_m \]

  – At the end multiply by \( w = \text{sizeof(baseType(A))} \)
  – Add the compile-time constant term
  – Keep track of which dimension at each level
  – Use the auxiliary function \( n_m = \text{numElem(A,m)} \) as the number of elements along the \( m^{th} \) dimension of \( A \)
SDT for Addressing Arrays Elements

\[ L \rightarrow \text{Elist} \]
\[
\begin{align*}
L\text{.place} &= \text{newtemp}(); \\
L\text{.offset} &= \text{newtemp}(); \\
\text{code1} &= \text{gen}(L\text{.place} \ '==' \ \text{constTerm}(\text{Elist.array})); \\
\text{code2} &= \text{gen}(L\text{.offset} \ '==' \ \text{Elist.place} \ \times \ \text{sizeof}(\text{baseType}(\text{Elist.array}))); \\
L\text{.code} &= \text{append}(\text{Elist.code}, \text{code1}, \text{code2});
\end{align*}
\]

\[ x = A[y, z]; \]

\[ \text{Elist} \rightarrow \text{Elist}_1, E \]
\[
\begin{align*}
t &= \text{newtemp}(); \\
m &= \text{Elist}_1\text{.ndim} + 1; \\
\text{code1} &= \text{gen}(t = \text{Elist}_1\text{.place} \ \times \ \text{numElem}(\text{Elist}_1\text{.array}, m)); \\
\text{code2} &= \text{gen}(t = t + E\text{.place}); \\
\text{Elist.array} &= \text{Elist}_1\text{.array}; \\
\text{Elist.place} &= t; \\
\text{Elist.ndim} &= m; \\
\text{Elist.code} &= \text{append}(\text{Elist}_1\text{.code}, E\text{.code}, \text{code1}, \text{code2});
\end{align*}
\]

\[ \text{Elist} \rightarrow \text{id} [ E \}
\[
\begin{align*}
\text{Elist.array} &= \text{id}\text{.place}; \\
\text{Elist.place} &= E\text{.place}; \\
\text{Elist.ndim} &= 1; \\
\text{Elist.code} &= E\text{.code};
\end{align*}
\]
SDT for Addressing Arrays Elements

\[
E \rightarrow L \quad \begin{cases} 
\text{if } (L.\text{offset} = \text{NULL}) \text{ then} \\
E.\text{place} = L.\text{place}; \\
\text{else} \\
E.\text{place} = \text{newtemp}; \\
E.\text{code} = \text{gen}(E.\text{place} = L.\text{place}[L.\text{offset}]); 
\end{cases}
\]

\[
S \rightarrow L = E \quad \begin{cases} 
\text{if } L.\text{offset} = \text{NULL} \text{ then} \\
E.\text{code} = \text{gen}(L.\text{place} = E.\text{place}); \\
\text{else} \\
S.\text{code} = \text{append}(E.\text{code}, \text{gen}(L.\text{place}[L.\text{offset}] = E.\text{place}); 
\end{cases}
\]

\[
L \rightarrow id \quad \begin{cases} 
L.\text{place} = id.\text{place}; \\
L.\text{offset} = \text{null}; 
\end{cases}
\]
SDT for Addressing Arrays Elements

\[ x = A[y,z]; \]

A is 10 x 20 array with low_1 = low_2 = 1

\[ \text{sizeof(baseType(A))} = \text{sizeof(int)} = 4 \text{ bytes} \]
\[ x = A[y, z]; \]

A is 10 x 20 array with low_1 = low_2 = 1

\[
\text{sizeof(baseType}(A)) = \text{sizeof}(\text{int}) = 4 \text{ bytes}
\]

\[ \begin{align*}
  & \text{t1} = y \times 20 \quad \text{// numElem}(A, 2) = 20 \\
  & \text{t1} = \text{t1} + z \quad \text{// E.place is z} \\
  & \text{t2} = c \quad \text{// c = base}(A) - (20+1) \times 4 \\
  & \text{t3} = \text{t1} \times 4 \quad \text{// sizeof}(\text{int}) = 4 \\
  & \text{t4} = \text{t2}[t3] \\
  & x = \text{t4}
\end{align*} \]
Array References

What about Arrays as Actual Parameters?

Whole arrays, as call-by-reference parameters
- Need dimension information ⇒ build a dope vector
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address at each reference

Some improvement is possible
- Save len_i and low_i rather than low_i and high_i
- Pre-compute the fixed terms in prologue sequence
Array References

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it’s easy
• Pass the address or value, as needed
• Must know about both formal & actual parameter
• Language definition must force this interpretation

What is corresponding parameter is an array?
• Must know about both formal & actual parameter
• Meaning must be well-defined and understood
• Cross-procedural checking of conformability
Array References

What about Variable-Sized Arrays?

Local arrays dimensioned by actual parameters

• Same set of problems as parameter arrays
• Requires dope vectors (or equivalent)
  – dope vector at fixed offset in activation record

⇒ Different access costs for textually similar references

This presents a lot of opportunity for a good optimizer

• Common sub-expressions in the address
• Contents of dope vector are fixed during each activation

⇒ Handle them like parameter arrays
Array Address Calculations in a Loop

\[
\text{DO } J = 1, N \\
\text{END DO}
\]

• Naïve: Perform the address calculation twice

\[
\text{DO } J = 1, N \\
R1 = \text{baseA}_0 + (J \times \text{len}_1 + I) \times \text{floatsize} \\
R2 = \text{baseB}_0 + (J \times \text{len}_1 + I) \times \text{floatsize} \\
\text{MEM}(R1) = \text{MEM}(R1) + \text{MEM}(R2) \\
\text{END DO}
\]
Array Address Calculations in a Loop

DO J = 1, N
END DO

• Sophisticated: Move common calculations out of loop

    R1 = I x floatsize
    c = len1 x floatsize ! Compile-time constant
    R2 = baseA0 + R1
    R3 = baseB0 + R1
    DO J = 1, N
        a = J x c
        R4 = R2 + a
        R5 = R3 + a
        MEM(R4) = MEM(R4) + MEM(R5)
    END DO
Array Address Calculations in a Loop

DO J = 1, N
END DO

- Very sophisticated: Convert multiply to add (Operator Strength Reduction)

    R1 = I \times \text{floatsize}
    c = \text{len}_1 \times \text{floatsize} \quad ! \text{Compile-time constant}
    R2 = \text{baseA}_0 + R1; \quad R3 = \text{baseB}_0 + R1
    DO J = 1, N
        R2 = R2 + c
        R3 = R3 + c
        \text{MEM}(R2) = \text{MEM}(R2) + \text{MEM}(R3)
    END DO
SDT Scheme for Boolean Expressions

• Two Basic Code Generation Flavors
  – Use boolean and, or and not instructions (like arithmetic).
  – Control-flow (or positional code) defines true or false of predicate.

• Arithmetic Evaluation
  – Simpler to generate code as just eagerly evaluate the expression.
  – Associate ‘1’ or ‘0’ with outcome of predicates and combine with logic instructions.
  – Use the same SDT scheme explained for arithmetic operations.

• Control Flow Evaluation (short circuit evaluation)
  – More efficient in many cases.
  – Complications:
    • Need to Know Address to Jump To in Some Cases
    • Solution: Two Additional Attributes
      – nextstat (Inherited) Indicates the next symbolic location to be generated
      – laststat (Synthesized) Indicates the last location filled
      – As code is generated down and up the tree attributes are filled with the correct values
Arithmetic Scheme: Grammar and Actions

E → false  \[\|\] E.place = newtemp()
E.code = {gen(E.place = 0)}
E.laststat = E.nextstat + 1

E → true  \[\|\] E.place = newtemp()
E.code = {gen(E.place = 1)}
E.laststat = E.nextstat + 1

E → (E₁) \[\|\] E.place = E₁.place;
E.code = E₁.code;
E₁.nextstat = E.nextstat
E.laststat = E₁.laststat

E → not E₁ \[\|\] E.place = newtemp()
E.code = append(E₁.code, gen(E.place = not E₁.place))
E₁.nextstat = E.nextstat
E.laststat = E₁.laststat + 1
Arithmetic Scheme: Grammar and Actions

\[ E \rightarrow E_1 \text{ or } E_2 \mid E.place = \text{newtemp() } \]
\[ E.code = \text{append}(E_1.code, E_2.code, \text{gen}(E.place = E_1.place \text{ or } E_2.place)) \]
\[ E_1.nextstat = E.nexstat \]
\[ E_2.nextstat = E_1.laststat \]
\[ E.laststat = E_2.laststat + 1 \]

\[ E \rightarrow E_1 \text{ and } E_2 \mid E.place = \text{newtemp() } \]
\[ E_2.place) \]
\[ E.code = \text{append}(E_1.code, E_2.code, \text{gen}(E.place = E_1.place \text{ and } E_2.place)) \]
\[ E_1.nextstat = E.nexstat \]
\[ E_2.nextstat = E_1.laststat \]
\[ E.laststat = E_2.laststat + 1 \]

\[ E \rightarrow \text{id}_1 \text{ relop } \text{id}_2 \mid E.place = \text{newtemp() } \]
\[ E.code = \text{gen(if id1.place relop id2.place goto E.nextstat+3)} \]
\[ E.code = \text{append}(E.code, \text{gen}(E.place = 0)) \]
\[ E.code = \text{append}(E.code, \text{gen(goto E.nexstat+2})) \]
\[ E.code = \text{append}(E.code, \text{gen(E.place = 1})) \]
\[ E.laststat = E.nextstat + 4 \]
**Boolean Expressions: Example**

\[
a < b \ or \ c < d \ and \ e < f
\]

```
00: if a < b goto 03
01: t1 = 0
02: goto 04
03: t1 = 1
04: if c < d goto 07
05: t2 = 0
06: goto 08
07: t2 = 1
08: if e < f goto 11
09: t3 = 0
10: goto 12
11: t3 = 1
12: t4 = t2 and t3
13: t5 = t1 or t4
```
Summary

• Intermediate Code Generation
  – Using Syntax-Directed Translation Schemes
  – Expressions and Assignments
  – Array Expressions and Array References
  – Boolean Expressions (Arithmetic Scheme)