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. AST)
- *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{<code>=</code>}, \text{Leaf(\text{id}, \text{id}.entry)}, E.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 + E_2 )</td>
<td>( E.\text{node} = \text{new Node}(\text{`+’}, E_1.\text{node}, E_2.\text{node}) )</td>
</tr>
<tr>
<td>( E \rightarrow E_1 * E_2 )</td>
<td>( E.\text{node} = \text{new Node}(\text{`*’}, 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{`uminus’}, 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(\text{id}, \text{id}.entry}) )</td>
</tr>
</tbody>
</table>
Pro: 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

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

\[ a + a \times (b - c) + (b - c) \times d \]
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>$E.\text{node} = \text{new Node}(\text{`+'}, E_1.\text{node}, T.\text{node})$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T$</td>
<td>$E.\text{node} = \text{new Node}(\text{`-'}, E_1.\text{node}, T.\text{node})$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.\text{node} = T.\text{node}$</td>
</tr>
<tr>
<td>$T \rightarrow (E)$</td>
<td>$T.\text{node} = E.\text{node}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T.\text{node} = \text{new Leaf(id, id.entry)}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{num}$</td>
<td>$T.\text{node} = \text{new Leaf(num, num.val)}$</td>
</tr>
</tbody>
</table>

Value-Number Representation for DAGs

$$i = i + 10$$

Value number of children
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 (3AC) instructions have at most one operator in the right hand side

• When translating, compiler needs to generate temporary names

\[
x + y \times z \quad \rightarrow \quad t_1 = y \times z \quad t_2 = x + t_1
\]
Three-Address Code

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

\begin{align*}
  t_1 &= -c \\
  t_2 &= b \ast t_1 \\
  t_3 &= -c \\
  t_4 &= b \ast t_3 \\
  t_5 &= t_2 + t_4 \\
  a &= t_5
\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 statements: \( x = y \odot z, \ x = \odot y \)
- Indexed assignments: \( x = y[i], x[i] = y \)
- Pointer assignments: \( x = \& y, x = *y, *x = y \)
- Copy statements: \( x = y \)
- Unconditional jumps: \( \text{goto lab} \)
- Conditional jumps: \( \text{if} \ x \ \text{relop} \ y \ \text{goto lab}, \text{iffalse} \ x \ \text{relop} \ y \ \text{goto lab} \)
- Function calls: \( \text{param} x \ \ldots \ \text{call} \ p, n, \) function exit: \( \text{return} y \)

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

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

Example

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

Quads (quadruples)

<table>
<thead>
<tr>
<th>#</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>minus</td>
<td>c</td>
<td></td>
<td>t_1</td>
</tr>
<tr>
<td>1</td>
<td>\times</td>
<td>b</td>
<td>t_1</td>
<td>t_2</td>
</tr>
<tr>
<td>2</td>
<td>minus</td>
<td>c</td>
<td></td>
<td>t_3</td>
</tr>
<tr>
<td>3</td>
<td>\times</td>
<td>b</td>
<td>t_3</td>
<td>t_4</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>t_2</td>
<td>t_4</td>
<td>t_5</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>t_5</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>#</th>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>minus</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>minus</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>#</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>

Program

Triple container

Pro: temporaries are implicit & easier to rearrange code

Type Expressions

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

```
int[2][3]
```

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

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

Tree forms

DAGs

Cyclic Graph Representations

\[
\text{struct Node} \\
\{ \text{int val;} \\
\quad \text{struct Node } \ast \text{next;} \\
\};
\]

Cyclic graph
Recursive definition for Type Expressions

• Basic types, such as \texttt{int}, \texttt{boolean} and \texttt{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
• 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#

```
struct
  val int
  next pointer

=

struct
  val int
  next pointer
```
Structural Equivalence of Type Expressions (cont’ d)

- 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

%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

\[
D \rightarrow T id ; D \mid \varepsilon \\
T \rightarrow B C \mid \text{record } \{ D \} \\
B \rightarrow \text{int} \mid \text{float} \\
C \rightarrow \varepsilon \mid [\text{num}] C
\]
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

```plaintext
T → B { C.t = B.type; C.w = B.width; }
C { T.type = C.type; T.width = C.width; }
B → int { B.type = integer; B.width = 4; }
B → float { B.type = float; B.width = 8; }
C → ε { C.type = C.t; C.width = C.w; }
C → [ num ] { C₁.t = C.t; C₁.w = C.w; }
C₁ { C₁.type = array(num.value, C₁.type);
    C₁.width = num.value × C₁.width; }
```
Type Expression and Storage Allocation

Sequences of Declarations

• ...

\[
\begin{align*}
t & = \text{integer} \\
w & = 4 \\
B & \quad \text{int} \\
\text{width} & = 4 \\
\end{align*}
\]
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

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

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

\[ t_1 = \text{minus } c \]
\[ t_2 = b + t_1 \]
\[ a = t_2 \]
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

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \to \text{id} = E ;$</td>
<td>$\text{gen(top.get(id.lexeme)} \ '=' \ E.\text{addr})$</td>
</tr>
</tbody>
</table>
| $E \to E_1 + E_2$ | $E.\text{addr} = \text{new Temp}();$  
| | $\text{gen(E.addr} \ '=' \ E_1.\text{addr} \ ‘+’ \ E_2.\text{addr})$ |
| $E \to E_1 * E_2$ | $E.\text{addr} = \text{new Temp}();$  
| | $\text{gen(E.addr} \ '=' \ E_1.\text{addr} \ ‘*’ \ E_2.\text{addr})$ |
| $E \to - E_1$ | $E.\text{addr} = \text{new Temp}();$  
| | $\text{gen(E.addr} \ '=' \ ‘\text{minus}’ \ E_1.\text{addr})$ |
| $E \to ( E_1 )$ | $E.\text{addr} = E_1.\text{addr}$ |
| $E \to \text{id}$ | $E.\text{addr} = \text{top.get(id.lexeme)}$ |
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: One-Dimensional Arrays

\[
A : \text{array} [10..20] \text{ of integer;}
\]

\[
... = A[i] = base_A + (i - \text{low}) \times w \\
= i \times w + c
\]

where \(c = base_A - \text{low} \times w\)

with \(\text{low} = 10; w = 4\)

\[
t1 = c \quad // \quad c = base_A - 10 \times 4 \\
t2 = i \times 4 \\
t3 = t1[t2] \\
... = t3
\]
Addressing Array Elements: Multi-Dimensional Arrays

\( A : \text{array [1..2,1..3] of integer; } \)

\( low_1 = 1, \; low_2 = 1, \; n_1 = 2, \; n_2 = 3, \; w = 4 \)

**base**\(_A\)  | **base**\(_A\)  
---|---
\( A[1,1] \) | \( A[1,1] \)  
\( A[1,2] \) | \( A[1,2] \)  
\( A[1,3] \) | \( A[1,3] \)  
Row-major | Column-major

---

Addressing Array Elements: Multi-Dimensional Arrays

\( A : \text{array [1..2,1..3] of integer}; \) (Row-major)

\[ ... = A[i,j] = \text{base}_A + ((i - low_1) \times n_2 + j - low_2) \times w \]

\[ = ((i \times n_2) + j) \times w + c \]

where \( c = \text{base}_A - ((low_1 \times n_2) + low_2) \times w \)

with \( low_1 = 1; \; low_2 = 1; \; n_2 = 3; \; w = 4 \)

\[ t1 := i \times 3 \]
\[ t1 := t1 + j \]
\[ t2 := c \quad // \quad c = \text{base}_A - (1 \times 3 + 1) \times 4 \]
\[ t3 := t1 \times 4 \]
\[ t4 := t2[t3] \]
\[ ... := t4 \]
Addressing Array Elements: Grammar

\[ S \rightarrow \text{id} = E ; \]
\[ E \rightarrow E + E \]
\[ L \rightarrow \text{id} \]
\[ L \rightarrow \text{id} [ E ] \]
\[ L \rightarrow L [ E ] \]

Addressing Array Elements: Indexing

- Assume 0-indexing
- Let \( w_j \) be the width of an element in the \( j \)-th dimension
- Use formula

\[
\text{base}_A + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k
\]
Addressing Array Elements:
Attributes

- \( E.addr \): 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:
Postfix SDT

\[
\begin{align*}
S & \rightarrow \text{id} = E ; & \{ \text{gen(top.get(id.lexeme) '=' E.addr); } \} \\
S & \rightarrow L = E ; & \{ \text{gen(L.array.base '[' L.addr ']'} = E.addr; } \} \\
E & \rightarrow E_1 + E_2 & \{ E.addr = \text{new Temp();} \\
 & & \text{gen(E.addr '=' E_1.addr '+' E_2.addr); } \} \\
E & \rightarrow \text{id} & \{ E.addr = \text{top.get(id.lexeme); } \} \\
E & \rightarrow L & \{ E.addr = \text{new Temp();} \\
 & & \text{gen(E.addr '=' L.array.base} \\
 & & \text{['L.addr ']'); } \} \\
\end{align*}
\]
Addressing Array Elements:
Postfix SDT (cont’d)

\[ L \rightarrow \text{id} [ E ] \]
\{
L.array = top.get(id.lexeme);
L.type = L.array.type.elem;
L.addr = \textbf{new} \ Temp();
gen(L.addr ‘=’ \ E.addr ‘*’
L.type.width); \}

\[ L \rightarrow L_1 [ E ] \]
\{
L.array = L_1.array;
L.type = L_1.type.elem;
t = \textbf{new} \ Temp();
L.addr = \textbf{new} \ Temp();
gen(t ‘=’ \ E.addr ‘*’ \ L.type.width);
gen(L.addr ‘=’ L_1.addr ‘+’ t); \}

Example:
Addressing Array Elements

\[ c + a[i][j] ; \]

\[
\begin{align*}
t_1 &= i \times 12 \\
t_2 &= j \times 4 \\
t_3 &= t_1 + t_2 \\
t_4 &= a \ [ t_3 ] \\
t_5 &= c + t_4
\end{align*}
\]
\[
E.addr = t5
\]

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

\[
\begin{aligned}
E.addr &= c \\
L.array &= a \\
L.type &= integer \\
L.addr &= t3 \\
\end{aligned}
\]

\[
\begin{aligned}
L.array &= a \\
L.type &= array(3, integer) \\
L.addr &= t1 \\
E.addr &= j \\
i
\end{aligned}
\]

\[
\begin{aligned}
L.type &= array(2, \\
array(3, integer) \\
E.addr &= i \\
\end{aligned}
\]