Concepts Introduced in Chapter 6

• types of intermediate code representations

• translation of
  – declarations
  – arithmetic expressions
  – boolean expressions
  – flow-of-control statements

• backpatching
Intermediate Code Generation Is Performed by the Front End

<table>
<thead>
<tr>
<th>Scanner</th>
<th>Parser</th>
<th>Static Checker</th>
<th>Intermediate Code Generation</th>
<th>Intermediate Code</th>
<th>Back End</th>
</tr>
</thead>
</table>

EECS 665 Compiler Construction
Intermediate Code Generation

- Intermediate code generation can be done in a separate pass (e.g. Ada requires complex semantic checks) or can be combined with parsing and static checking in a single pass (e.g. Pascal designed for one-pass compilation).

- Generating intermediate code rather than the target code directly
  - facilitates retargeting
  - allows a machine independent optimization pass to be applied to the intermediate representation
Types of Intermediate Representation

- Syntax trees and Directed Acyclic Graphs (DAG)
  - nodes represent language constructs
  - children represent components of the construct

- DAG
  - represents each *common subexpression* only once in the tree
  - helps compiler optimize the generated code
Types of Intermediate Representation

• Three-address code
  – general form: \( x = y \ op \ z \) (2 source, 1 destination)
  – widely used form of intermediate representation
  – Types of three-address code
    • quadruples, triples, static single assignment (SSA)
• Postfix
  – 0 operands (just an operator)
  – all operands are on a compiler-generated stack
Types of Intermediate Representation

• Two-address code
  - \( x := \text{op} \ y \)
  - where \( x := x \text{op} y \) is implied

• One-address code
  - \( \text{op} \ x \)
  - where \( ac := ac \text{op} x \) is implied and \( ac \) is an accumulator
Types of Three-Address Code

- Quadruples
  - has 4 fields, called $op$, $arg1$, $arg2$, and $result$
  - often used in compilers that perform global optimization on intermediate code.
  - easy to rearrange code since result names are explicit.

followed by Fig. 6.10
Types of Three-Address Code (cont...)

- **Triples**
  - similar to quadruples, but implicit results and temporary values
  - *result* of an operation is referred to by its position
  - triples avoid symbol table entries for temporaries, but complicate rearrangement of code.
  - indirect triples allow rearrangement of code since they reference a pointer to a triple instead.

followed by Fig. 6.11, 6.12
Types of Three-Address Code (cont...)

- Static Single Assignment (SSA) form
  - an increasing popular format in optimizing compilers
  - all assignments in SSA are to variables with a distinct name
  - see Figure 6.13

- $\phi$–function to combine multiple variable definitions

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

```
if (flag)
  x_1 = -1;
  x_2 = -1;
  x_3 = \phi(x_1, x_2);
  y = x_3 * a;
```

followed by Fig. 6.13
Three Address Stmts Used in the Text

- \( x := y \text{ op } z \)  \# binary operation
- \( x := \text{ op } y \)  \# unary operation
- \( x := y \)  \# copy or move
- \( \text{goto } L \)  \# unconditional jump
- \( \text{if } x \text{ relop } y \text{ goto } L \)  \# conditional jump
- \( \text{param } x \)  \# pass argument
- \( \text{call } p,n \)  \# call procedure \( p \) with \( n \) args
- \( \text{return } y \)  \# return (value is optional)
- \( x := y[i], x[i] := y \)  \# indexed assignments
- \( x := &y \)  \# address assignment
- \( x := *y, *x = y \)  \# pointer assignments
Postfix

- Having the operator after operand eliminates the need for parentheses.

\[(a+b) \times c \Rightarrow ab + c \times\]

\[a \times (b + c) \Rightarrow abc + \times\]

\[(a + b) \times (c + d) \Rightarrow ab + cd + \times\]

- Evaluate operands by pushing them on a stack.
- Evaluate operators by popping operands, pushing result.

\[A = B \times C + D \Rightarrow ABC \times D + =\]
Postfix (cont.)

<table>
<thead>
<tr>
<th>Activity</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push A</td>
<td>A</td>
</tr>
<tr>
<td>push B</td>
<td>AB</td>
</tr>
<tr>
<td>push C</td>
<td>ABC</td>
</tr>
<tr>
<td>*</td>
<td>Ar*</td>
</tr>
<tr>
<td>push D</td>
<td>Ar*D</td>
</tr>
<tr>
<td>+</td>
<td>Ar+</td>
</tr>
</tbody>
</table>

- Code generation of postfix code is trivial for several types of architectures.
Translation of Declarations

• Assign storage and data type to local variables.
• Using the declared data type
  – determine the amount of storage (integer – 4 bytes, float – 8 bytes, etc.)
  – assign each variable a relative offset from the start of the activation record for the procedure

followed by Fig. 6.17, 6.15, 6.16
Translation of Expressions

• Translate arithmetic expressions into three-address code.

• see Figure 6.19

• \( a = b + -c \) is translated into:

\[
\begin{align*}
  t_1 &= \text{minus } c \\
  t_2 &= b + t_1 \\
  a &= t_2
\end{align*}
\]
Translation of Boolean Expressions

- Boolean expressions are used in statements, such as if, while, to alter the flow of control.

- Boolean operators
  - ! – NOT (highest precedence)
  - && – AND (mid precedence, left associative)
  - || – OR (lowest precedence, left associative)
  - <, <=, >, >=, =, !=, are relational operators

- Short-circuit code
  - B1 || B2, if B1 true, then don't evaluate B2
  - B1 && B2, if B1 false, then don't evaluate B2

followed by Fig. 6.37
Translation of Control-flow Statements

- Control-flow statements include:
  - *if* statement
  - *if* statement *else* statement
  - *while* statement

followed by Fig. 6.35, 6.36
Control-Flow Translation of \textit{if}-Statement

• Consider statement:

\[
\text{if } (x < 100 \lor x > 200 \land x \neq y) \ x = 0;
\]

\[
\text{if } x < 100 \text{ goto } L_2 \\
goto L_3 \\
L_3 : \text{ if } x > 200 \text{ goto } L_4 \\
goto L_1 \\
L_4 : \text{ if } x \neq y \text{ goto } L_2 \\
goto L_1 \\
L_2 : x = 0 \\
L_1 : \]
Backpatching

- Allows code for boolean expressions and flow-of-control statements to be generated in a single pass.
- The targets of jumps will be filled in when the correct label is known.
Backpatching an ADA While Loop

- Example
  
  ```ada
  while a < b loop
    a := a + cost;
  end loop;
  ```

- `loop_stmt`: `WHILE m cexpr LOOP m seq_of_stmts n END LOOP m ';'`
  ```ada
  { dowhile ($2, $3, $5, $7, $10); }
  ```
Backpatching an Ada While Loop (cont.)

```ada
loop_stmt : WHILE m cexpr LOOP m seq_of_stmts n END LOOP m ';
  { dowhile ($2, $3, $5, $7, $10); } 

; 

void dowhile (int m1, struct sem_rec *e, int m2,
               struct sem_rec *n1, int m3) {
  backpatch(e→back.s_true, m2);
  backpatch(e→s_false, m3);
  backpatch(n1, m1);
  return(NULL);
}
```
Backpatching an Ada If Statement

• Examples:

```
if a < b then
  a := a + 1;
end if;
```

```
if a < b then
  a := a + 1;
else
  a := a + 2;
end if;
```

```
if a < b then
  a := a + 1;
elsif a < c then
  a := a + 2;
  ...
end if;
```
Backpatching an Ada If Statement (cont.)

\[
\text{if_stmt} : \quad \text{IF} \ \text{cexpr} \ \text{THEN} \ m \ \text{seq_of_stmts} \ n \ \text{elsif_list0} \\
\text{else_option} \ \text{END} \ \text{IF} \ m \ ';$'
\]
\[
\quad \{ \ \text{doif}(\$2, \$4, \$6, \$7, \$8, \$11); \ \}
\]
\[
\]
\[
\text{elsif_list0} : \quad \{ \\$\$ = (\text{struct sem_rec *}) \text{NULL}; \ \}
\| \ \text{elsif_list0} \ \text{ELSIF} \ m \ \text{cexpr} \ \text{THEN} \ m \ \text{seq_of_stmts} \ n
\]
\[
\quad \\{ \\$\$ = \text{doelsif}(\$1, \$3, \$4, \$6, \$8); \ \}
\]
\[
\]
\[
\text{else_option}: \quad \{ \ \$\$ = (\text{struct sem_rec *}) \text{NULL}; \ \}
\| \ \text{ELSE} \ m \ \text{seq_of_stmts} \quad \{ \ \$\$ = \$2; \ \}
\]
if_stmt : IF cexpr THEN m seq_of_stmts n elsif_list0 else_option END IF

    { doif($2, $4, $6, $7, $8, $11); }

void doif(struct sem_rec *e, int m1, struct sem_rec *n1,
        struct sem_rec *elsif, int elsopt, int m2) {
    backpatch(e->back.s_true, m1);
    backpatch(n1, m2);
    if (elsif != NULL) {
        backpatch(e->s_false, elsif->s_place);
        backpatch(elsif->back.s_link, m2);
        if (elsopt != 0)
            backpatch(elsif->s_false, elsopt);
        else
            backpatch(elsif->s_false, m2);
    } else if (elsopt != 0)
        backpatch(e->s_false, elsopt);
    else
        backpatch(e->s_false, m2);
}
Backpatching an Ada If Statement (cont.)

elsif_list0 : { $$ = (struct sem_rec *) NULL; }
    | elsif_list0 ELSIF m cexpr THEN m seq_ofStmts n
        { $$ = doelsif($1, $3, $4, $6, $8); }
    ;

struct sem_rec *doelsif (struct sem_rec *elsif, int m1, struct sem_rec *e,
                        int m2, struct sem_rec *n1) {
    backpatch (e->back.s_true, m2);
    if (elsif != NULL) {
        backpatch(elsif->s_false, m1);
        return (node(elsif->s_place, 0, merge(n1, elsif->back.s_link), e->s_false));
    }
    else
        return (node(m1, 0, n1, e->s_false));
}
Translating Record Declarations

Example:

```c
struct foo {int x; char y; float z;};
```

```plaintext
type :  CHAR { $$ = node(0,T_CHAR,1,0,0); }
       | FLOAT  { $$ = node(0,T_FLOAT,8,0,0); }
       | INT    { $$ = node(0,T_INT,4,0,0); }
       | STRUCT '{' fields '}' { $$ = node(0,T_STRUCT,$3→width,0,0); }

fields :  field ';;' { $$ = addfield($1,0); }
         | fields field ';;' { $$ = addfield($2,$1); }

field :  type ID { $$ = makefield($2,$1); }
        | field '[' CON ']' { $1→width = $1→width*$3; $$ = $1; }
```
Translating Record Declarations (cont.)

fields : field ';' { $$ = addfield($1, 0); } 
    | fields field ';' { $$ = addfield($2,$1); } 
| ;

struct sem_rec *addfield(struct id_entry *field, struct sem_rec *fields) {
    if (fields != NULL) {
        field->s_offset = fields->width;
        return (node(0,0,field->s_width+fields->width,0,0));
    }
    else {
        field->s_offset = 0;
        return (node(0,0,field->s_width,0,0));
    }
}
Translating Record Declarations
(cont.)

field : type ID
  {$$ = makefield($2,$1);}
| field [' CON ']
  {$1\rightarrow s\_width = $1\rightarrow s\_width*$$3; $$ = $1;}
;      

struct id_entry *makefield(char *id, struct sem_rec *type) {
  struct id_entry *p;

  if ((p = lookup(id, 0)) != NULL)
    fprintf( stderr, "duplicate field name\n");
else {
  p = install(id, 0);
  p\rightarrow s\_width = type\rightarrow width;
  p\rightarrow attributes = field_descriptor;
}
return (p);
Translating Switch Statements

switch (E) {
    case V1:   S1
    case V2:   S2
    ...
    case Vn-1:  Sn-1
    default:   Sn
}

Translating Large Switch Statements

switch (E) {
    case 1:       S1
    case 2:       S2
    ...
    case 1000:    S1000
    default:      S1001
}

Translating Large Switch Statements

goto test
L1: code for S1
L2: code for S2
...
L1000: code for S1000
LD: code for S1001
goto next
test: check if expr is in range
if not goto LD
t := m[jump_table_base + expr << 2];
goto t;
next:

followed by Fig. 6.49, 6.50
Addressing One Dimensional Arrays

- Assume $w$ is the width of each array element in array $A[]$ and $low$ is the first index value.
- The location of the $i$th element in $A$.
  
  $$ \text{base} + (i - \text{low}) \times w $$

- Example:

  INTEGER ARRAY $A[5:52]$;
  ...
  $N = A[I]$;
  - $\text{low}=5$, $\text{base}=\text{addr}(A[5])$, $\text{width}=4$
  $\text{address}(A[I])=\text{addr}(A[5])+(I-5)\times4$
Addressing One Dimensional Arrays Efficiently

• Can rewrite as:

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

\[ \text{address}(A[I]) = I \times 4 + \text{addr}(A[5]) - 5 \times 4 \]

\[ = I \times 4 + \text{addr}(A[5]) - 20 \]
Addressing Two Dimensional Arrays

• Assume row-major order, \( w \) is the width of each element, and \( n_2 \) is the number of values \( i_2 \) can take.

\[
\text{address} = \text{base} + ((i_1 - \text{low}_1)*n_2 + i_2 - \text{low}_2)*w
\]

• Example in Pascal:

\[
\text{var } a : \text{array[3..10, 4..8] of real;}
\]
\[
\text{addr(a[i][j]) = addr(a[3][4]) + ((i-3)*5 + j - 4)*8}
\]

• Can rewrite as

\[
\text{address} = ((i_1*n_2)+i_2)*w + (\text{base} - ((\text{low}_1*n_2)+\text{low}_2)*w)
\]
\[
\text{addr(a[i][j]) = ((i*5)+j)*8 + addr(a[3][4]) - ((3*5)+4)*8}
\]
\[
= ((i*5)+j)*8 + \text{addr(a[3][4])} - 152
\]
Addressing C Arrays

• Lower bound of each dimension of a C array is zero.

• 1 dimensional
  base + i*w

• 2 dimensional
  base + (i1*n2 + i2)*w

• 3 dimensional
  base + ((i1*n2 + i2)*n3 + i3)*w
Static Checking

1. Type Checks
   Ex: int a, c[10], d;
       a = c + d;

2. Flow-of-control Checks
   Ex: main {
       int i;
       i++; 
       break;
   }
Static Checking (cont.)

3. Uniqueness Checks

Ex:  program foo ( output );
     var  i, j : integer;
     a, i : real;

4. Name-related Checks

Ex:  LOOPA:
     LOOP
     EXIT WHEN I = N;
     I = I + 1;
     TERM := TERM / REAL ( I );
     END LOOP LOOPB;
Static and Dynamic Type Checking

- Static type checking is performed by the compiler.
- Dynamic type checking is performed when the target program is executing.
- Some checks can only be performed dynamically:
  
  ```
  var i : 0..255;
  ...
  i := i+1;
  ```
Why is Static Checking Preferable to Dynamic Checking?

- There is no guarantee that the dynamic check will be tested before the application is distributed.
- The cost of a static check is at compile time, where the cost of a dynamic check may occur every time the associated language construct is executed.
Basic Terms

• Atomic types - types that are predefined or known by the compiler
  – boolean, char, integer, real in Pascal

• Constructed types - types that one declares
  – arrays, records, pointers, classes

• Type expression - the type associated with a language construct

• Type system - a collection of rules for assigning type expressions to various parts of a program
Equivalence of Type Expressions

- Name equivalence - views each type name as a distinct type
- Structural equivalence - names are replaced by the type expressions they define

Ex: type link = \( \uparrow \text{cell} \);

```plaintext
var next : link;
last : link;
p : \( \uparrow \text{cell} \);
q, r : \( \uparrow \text{cell} \);
```
### Equivalence of Type Expressions (cont.)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>link</td>
</tr>
<tr>
<td>last</td>
<td>link</td>
</tr>
<tr>
<td>p</td>
<td>pointer (cell)</td>
</tr>
<tr>
<td>q</td>
<td>pointer (cell)</td>
</tr>
<tr>
<td>r</td>
<td>pointer (cell)</td>
</tr>
</tbody>
</table>

**structural equivalence** - all are equivalent

**name equivalence**  -  next == last, p == q == r  
but p != next
Type Checking

• Perform type checking
  – assign type expression to all source language components
  – determine conformance to the language type system

• A sound type system statically guarantees that type errors cannot occur at runtime.

• A language implementation is strongly typed if the compiler guarantees that the program it accepts will run without type errors.
Rules for Type Checking

• Type synthesis
  – build up type of expression from types of subexpressions

  \[
  \text{if } f \text{ has type } s \rightarrow \text{and } x \text{ has type } s, \\
  \text{then expression } f(x) \text{ has type } t
  \]

• Type inference
  – determine type of a construct from the way it is used

  \[
  \text{if } f(x) \text{ is an expression} \\
  \text{then for some } \alpha \text{ and } \beta, f \text{ has type } \alpha \rightarrow \beta \text{ and } x \text{ has type } \alpha
  \]
Example of a Simple Type Checker

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P → D; E</td>
<td></td>
</tr>
<tr>
<td>D → D; D</td>
<td></td>
</tr>
<tr>
<td>D → id : T</td>
<td>{ addtype(id.entry, T.type); }</td>
</tr>
<tr>
<td>T → char</td>
<td>{ T.type = char; }</td>
</tr>
<tr>
<td>T → integer</td>
<td>{ T.type = integer; }</td>
</tr>
<tr>
<td>T → ↑T₁</td>
<td>{ T.type = pointer (T₁.type); }</td>
</tr>
<tr>
<td>T → array[num]of T₁</td>
<td>{ T.type = array(num.val,T₁.type); }</td>
</tr>
<tr>
<td>E → literal</td>
<td>{ E.type = char; }</td>
</tr>
<tr>
<td>E → num</td>
<td>{ E.type = integer; }</td>
</tr>
</tbody>
</table>
Example of a Simple Type Check (cont.)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → id</td>
<td>{ E.type = lookup(id.entry); }</td>
</tr>
<tr>
<td>E → E₁ mod E₂</td>
<td>{ E.type = E₁.type == integer &amp;&amp; E₂.type == integer ? integer : type_error( ); }</td>
</tr>
<tr>
<td>E → E₁[E₂]</td>
<td>{ E.type = E₂.type == integer &amp;&amp; isarray(E₁.type, &amp;t) ? t : type_error( ); }</td>
</tr>
<tr>
<td>E → E₁↑</td>
<td>{ E.type = ispointer(E₁.type,&amp;t) ? t : type_error( ); }</td>
</tr>
</tbody>
</table>
Type Conversions - Coercions

• An implicit type conversion.
• In C or C++, some type conversions can be implicit
  - assignments
  - operands to arithmetic and logical operators
  - parameter passing
  - return values
Overloading in Java

• A function or operator can represent different operations in different contexts

• Example 1
  – operators '+', '-' etc., are overloaded to work with different data types

• Example 2
  – function overloading resolved by looking at the arguments of a function

```java
void err () { ... }
void err (String s) { ... }
```
Polymorphism

- The ability for a language construct to be executed with arguments of different types

- Example 1
  - function `length` can be called with different types of lists

  ```
  fun length (x) = 
      if null (x) then 0 else length (tail(x)) + 1
  ```

- Example 2
  - templates in C++

- Example 3
  - using the `object` class in Java