Concepts Introduced in Chapter 6

- types of intermediate code representations
- translation of
  - declarations
  - arithmetic expressions
  - boolean expressions
  - flow-of-control statements
- backpatching
- type checking

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>

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 Representations

- syntax tree or DAG
  - see Figure 6.3 for an example DAG
- postfix
  - 0 operands (just an operator)
  - all operands are on a compiler-generated stack
- three-address code
  - general form
    - $x := y \text{ op } z$
  - quadruples, triples, indirect triples
Types of Intermediate Representations (cont.)

- **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

Directed Acyclic Graphs for Expressions

- Directed acyclic graphs (dags) are like a syntax tree, except that a node in the dag can have more than one parent.
- Dags can be used to recognize common subexpressions in an expression. The routines that make a node can check if an identical node has already been constructed.

Postfix

- Having the operator after operand eliminates the need for parentheses.
  \[
  (a+b)*c \rightarrow ab+c* \\
  a*(b+c) \rightarrow abc* \\
  (a+b)*(c+d) \rightarrow ab+cd+*
  \]
- Evaluate operands by pushing them on a stack.
- Evaluate operators by popping operands, pushing result.

  \[
  A=B*C+D \rightarrow ABC*D+=
  \]

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>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>

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

- a record structure with four fields
  - operator
  - source argument 1
  - source argument 2
  - Result

Example:  \( A = B \cdot (C + D) \)

<table>
<thead>
<tr>
<th></th>
<th>Op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( T_1 \leftarrow B )</td>
<td>neg</td>
<td>B</td>
<td>( T_1 )</td>
</tr>
<tr>
<td>2.</td>
<td>( T_2 \leftarrow C + D )</td>
<td>int add</td>
<td>C</td>
<td>( D )</td>
</tr>
<tr>
<td>3.</td>
<td>( A \leftarrow T_1 \cdot T_2 )</td>
<td>int mul</td>
<td>( T_1 )</td>
<td>( T_2 )</td>
</tr>
</tbody>
</table>

Quadruples (cont.)

- Often used in compilers that perform global optimization on intermediate code.
- Easy to rearrange code since result names are explicit.

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

Triples

- Triples - like quadruples, but implicit results and temporary values

<table>
<thead>
<tr>
<th></th>
<th>( A = -B \cdot (C + D) )</th>
<th>( A[i] = B )</th>
<th>( A = B[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>neg ( B )</td>
<td>( [ ] = A \ i )</td>
<td>( [ ] = B \ i )</td>
</tr>
<tr>
<td>1.</td>
<td>( +i \ C \ D )</td>
<td>( =i (0) B )</td>
<td>( =i A (0) )</td>
</tr>
<tr>
<td>2.</td>
<td>( *i (0) (1) )</td>
<td>( =i A (2) )</td>
<td></td>
</tr>
</tbody>
</table>
Triples (cont.)

- 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.

Type Checking

- Static and dynamic checking
- Type systems
- Coercion, overloading, and polymorphism
- Checking equivalence of types

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: main() {
       int i, j;
       float a, i;
       ... 

4. Name-related Checks
   Ex: LOOPA:
       LOOP 
       EXIT WHEN I =N;
       I = I + 1;
       TERM := TERM / REAL ( I );
       END LOOP LOOPB;
Basic Terms

- Basic types - types that are predefined or known by the compiler
  - char, int, float, void in C
- 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

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:

\[
\text{var } i : 0..255;
\]
\[
\ldots
\]
\[
i := i+1;
\]

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 everytime the associated language construct is executed.

Grammar for a Simple Language

\[
P \rightarrow D ; E
\]
\[
D \rightarrow D ; D \mid \text{id} : T
\]
\[
T \rightarrow \text{char} \mid \text{integer} \mid \text{array [num] of } T \mid T
\]
\[
E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \mod E \mid E[E] \mid E^
\]
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 Checker (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>

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 = ↑cell;
var next : link;
    last : link;
p : ↑cell;
q, r : ↑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
Using Different Types

- Coercion - an implicit type conversion
- Overloading - a function or operator can represent different operations in different contexts
- Polymorphism - the ability for a language construct to be executed with arguments of different types

Coercions

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

Overloading in C++

```cpp
void swap(int &x, int &y);
void swap(double &x, double &y);

matrix operator*(matrix &r, matrix &s);
matrix operator*(vector &r, vector &s);
```

Polymorphism through Ada Generics

```ada
generic type ELEM is private;
procedure EXCHANGE(U, V: in out ELEM);

procedure EXCHANGE(U, V: in out ELEM) is
  T: ELEM;
begin
  T := U;  U := V; V := T;
end EXCHANGE;

procedure SWAP is new EXCHANGE(INTEGER);
```
Boolean Expressions

- Boolean expressions are used in flow of control statements and for computing logical values.
- In C and most other languages, boolean operators | |, &&, and ! are translated into code that uses transfers of control.

\[
 B \rightarrow B \mid B | B \&\& B \mid !B | (B) | E \text{ rel } E | \text{true} | \text{false}
\]

Example of Short-Circuit Code

if \((x < 100 \mid x > 200 \&\& x != y)\) \(x = 0\);

can translate into:

\[
\text{if } x < 100 \text{ goto L2}
\]
\[
\text{ifFalse } x > 200 \text{ goto L1}
\]
\[
\text{ifFalse } x != y \text{ goto L1}
\]

Flow of Control Statements

- Consider the translation of boolean expressions in the context of flow of control statements.

\[
S \rightarrow \text{if} (\ B \ ) S_1
\]
\[
S \rightarrow \text{if} (\ B \ ) S_1 \text{ else } S_1
\]
\[
S \rightarrow \text{while} (\ B \ ) S_1
\]

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
  while a < b loop
    a := a + cost;
  end loop;

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


Backpatching an Ada If Statement

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

Back Patching an Ada While Loop (cont.)

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);
}

Backpatching an Ada If Statement (cont.)

if_stmt : IF cexpr THEN m seq_of_stmts n elsif_list0 else_option END IF m ';
  { doif($2, $4, $6, $7, $9, $11); }

elsif_list0: { $$ = (struct sem_rec *) NULL; }
  | elsif_list0 ELSIF m cexpr THEN m seq_of_stmts n
  { $$ = doelsif($1, $3, $4, $6, $8); }
  |
else_option: { $$ = (struct sem_rec *) NULL; }
  | ELSE m seq_of_stmts
  { $$ = $2; }
  |
if_stmt : IF cexpr THEN m seq_of_stmts n elsif_list0
  else_option END IF m
  { 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);
  }

Translation of Record Declarations

Example:

struct foo { int x; char y; double z; };

type : CHAR
| DOUBLE
| INT
| STRUCT '{' fields '}'

fields : field ';
| fields field ';

field : type ID
| field '[' CON ']

Backpatching an Ada If Statement

elsif_list0 : { $$ = (struct sem_rec *) NULL; }
  | elsif_list0 ELSIF m cexpr THEN m seq_of_stmts n
  { $$ = 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, elsif-s_place);
    return node(elsif-s_place, 0,
                merge(n1, elsif-back.s_link), e-s_false);
  }
  else
    return node(m1, 0, n1, e-s_false);
}

Translation of Record Declarations (cont.)

fields: field ';
| fields field ';

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.)

```c
field : type ID  {$$ = makefield($2,$1);}
   | field '[' CON ']' {$1->s_width = $1-s_width*$3;
   $$ = $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->s_width = type->width;
        p->attributes = field_descriptor;
    }
    return (p);
}
```

Translating Switch Statements

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

Translating Large Switch Statements

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

Translating Large Switch Statements (cont.)

```c
    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
          offset := (expr - lowest_case_value) << 2;
          t := m[jump_table_base + offset];
          goto t;
    next:
```
Addressing One Dimensional Arrays

- Assume \( w \) is the width of each array element in array \( A[] \) and \( \text{low} \) is the first index value.
- The location of the \( i \)th element in \( A \).
  \[
  \text{base} + (i - \text{low}) \cdot w
  \]
- Example:
  
  INTEGER ARRAY \( A[5:52] \);
  
  \[
  N = A[I];
  \]
  
  \( \text{low} = 5, \text{base} = \text{addr}(A[5]), \text{width} = 4 \)
  
  \[
  \text{address}(A[I]) = \text{addr}(A[5]) + (I - 5) \cdot 4
  \]

Addressing One Dimensional Arrays Efficiently

- Can rewrite as:
  
  \[
  i \cdot w + \text{base} - \text{low} \cdot w
  \]
  
  \[
  \text{address}(A[I]) = I \cdot 4 + \text{addr}(A[5]) - 5 \cdot 4
  \]
  
  \[
  = I \cdot 4 + \text{addr}(A[5]) - 20
  \]

Addressing Two Dimensional Arrays

- Assume row-major order, \( w \) is the width of each element, and \( n2 \) is the number of values \( i2 \) can take.
  
  \[
  \text{address} = \text{base} + ((i1 - \text{low}1) \cdot n2 + i2 - \text{low}2) \cdot w
  \]
- Example in Pascal:
  
  \[\text{var } a : \text{array}[3..10, 4..8] \text{ of real; }\]
  \[
  \text{addr}(a[i][j]) = \text{addr}(a[3][4]) + ((i-3) \cdot 5 + j-4) \cdot 8
  \]
- Can rewrite as
  
  \[
  \text{address} = ((i1 \cdot n2 + i2) \cdot w + (\text{base} - ((\text{low}1 \cdot n2) + \text{low}2) \cdot w)
  \]
  
  \[
  \text{addr}(a[i][j]) = ((i \cdot 5) + j) \cdot 8 + \text{addr}(a[3][4]) - ((3 \cdot 5) + 4) \cdot 8
  \]
  
  \[
  = ((i \cdot 5) + j) \cdot 8 + \text{addr}(a[3][4]) - 152
  \]

Addressing C Arrays

- Lower bound of each dimension of a C array is zero.
- 1 dimensional
  
  \[
  \text{base} + i \cdot w
  \]
- 2 dimensional
  
  \[
  \text{base} + (i1 \cdot n2 + i2) \cdot w
  \]
- 3 dimensional
  
  \[
  \text{base} + ((i1 \cdot n2 + i2) \cdot n3 + i3) \cdot w
  \]