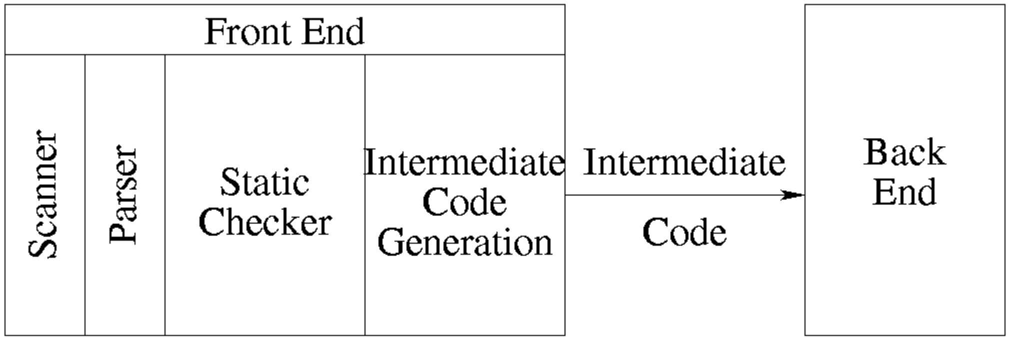
# Concepts Introduced in Chapter 6

- 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




# 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 op z`
    - 3 operands (2 src, 1 dst)
  - quadruples, triples, indirect triples

# Types of Intermediate Representations (cont.)

- two-address code
  - `x := op y`
  - where `x := x op y` is implied
- one-address code
  - `op x`
  - where `ac := ac 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       ⇒ ab+c*
a*(b+c)       ⇒ abc+*
(a+b)*(c+d)   ⇒ ab+cd+*
```

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

```
A=B*C+D ⇒ ABC*D+=
```

# Postfix (cont.)

| Activity | Stack |
|---|---|
| push A | A |
| push B | AB |
| push C | ABC |
| * | Ar* |
| push D | Ar*D |
| + | Ar+ |
| = | |

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

# Quadruples

Quadruples - a record structure with four fields

- operator
- source argument 1
- source argument 2
- Result

Example: `A=B*(C+D)`

| | Op | arg1 | arg2 | result |
|---|---|---|---|---|
| 1. `T1 ← B` | neg | `B` | — | `T1` |
| 2. `T2 ← C+D` | int add | `C` | `D` | `T2` |
| 3. `A ← T1*T2` | int mul | `T1` | `T2` | `A` |

# 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 op z` # binary operation
- `x := op y` # unary operation
- `x := y` # copy or move
- `goto L` # unconditional jump
- `if x relop y goto L` # conditional jump
- `param x` # pass argument
- `call p,n` # call procedure `p` with `n` args
- `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

| A=-B*(C+D) | A[i]=B | A=B[i] |
|---|---|---|
| `0. negi B` | `0. [ ]= A i` | `0. =[ ] B i` |
| `1. +i C D` | `1. =i (0) B` | `1. =i A (0)` |
| `2. *i (0) (1)` | | |
| `3. =i A (2)` | | |

# 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:

```
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 everytime the associated language construct is executed.

# Grammar for a Simple Language

```
P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑T
E → literal | num | id | E mod E | E[E] | E↑
```

# Example of a Simple Type Checker

| Production | Semantic Rule |
|---|---|
| `P → D; E` | |
| `D → D; D` | |
| `D → id : T` | `{ addtype(id.entry, T.type); }` |
| `T → char` | `{ T.type = char; }` |
| `T → integer` | `{ T.type = integer; }` |
| $T \rightarrow \uparrow T_1$ | $\{ T.type = pointer(T_1.type); \}$ |
| $T \rightarrow array[num] of\ T_1$ | $\{ T.type = array(num.val, T_1.type); \}$ |
| `E → literal` | `{ E.type = char; }` |
| `E → num` | `{ E.type = integer; }` |

# Example of a Simple Type Checker (cont.)

| Production | Semantic Rule |
|---|---|
| `E → id` | `{ E.type = lookup(id.entry); }` |
| $E \rightarrow E_1 \bmod E_2$ | $\{ E.type = E_1.type == integer$ && $E_2.type == integer\ ?\ integer : type\_error(); \}$ |
| $E \rightarrow E_1[E_2]$ | $\{ E.type = E_2.type == integer$ && $isarray(E_1.type, \&t)\ ?\ t : type\_error(); \}$ |
| $E \rightarrow E_1\uparrow$ | $\{ E.type = ispointer(E_1.type, \&t)\ ?\ t : type\_error(); \}$ |

# 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.)

| Variable | Type Expression |
|---|---|
| `next` | `link` |
| `last` | `link` |
| `p` | pointer (`cell`) |
| `q` | pointer (`cell`) |
| `r` | pointer (`cell`) |

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++

```
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

```
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 →  B || B | B && B | !B | (B) | E rel E |
     true | false
```

# Example of Short-Circuit Code

```
if (x < 100 || x > 200 && x != y) x = 0;
```

can translate into:

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

# Flow of Control Statements

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

S → **if** ( B ) $S_1$

S → **if** ( B ) $S_1$ **else** $S_1$

S → **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

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

# 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

- Examples:

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

# 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);
}
```

# Backpatching an Ada If Statement (cont.)

```
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, 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:

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

```
type :  CHAR                    { $$ = node(0, T_CHAR, 1, 0, 0); }
     |  DOUBLE                  { $$ = node(0, T_DOUBLE, 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            {$$ = makefield($2,$1);}
      |  field '[' CON ']' {$1→s_width = $1→s_width*$3;
                             $$ = $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

```
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 (cont.)

```
        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 low is the first index value.
- The location of the *i*th element in A.

  base + (i – low)*w

- Example:

```
INTEGER ARRAY A[5:52];
...
N = A[I];
```

  – low=5, base=addr(A[5]), width=4

```
address(A[I])=addr(A[5])+(I-5)*4
```

# Addressing One Dimensional Arrays Efficiently

- Can rewrite as:

  i*w + base – low*w

```
address(A[I])  = I*4 + addr(A[5]) - 5*4
               = I*4 + 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.

  address = base + ((i1 – low1)*n2 + i2 – low2)*w

- Example in Pascal:

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

- Can rewrite as

  address = ((i1*n2)+i2)*w + (base – ((low1*n2)+low2)*w)

```
addr(a[i][j]) = ((i*5)+j)*8+addr(a[3][4])-((3*5)+4)*8
              = ((i*5)+j)*8+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