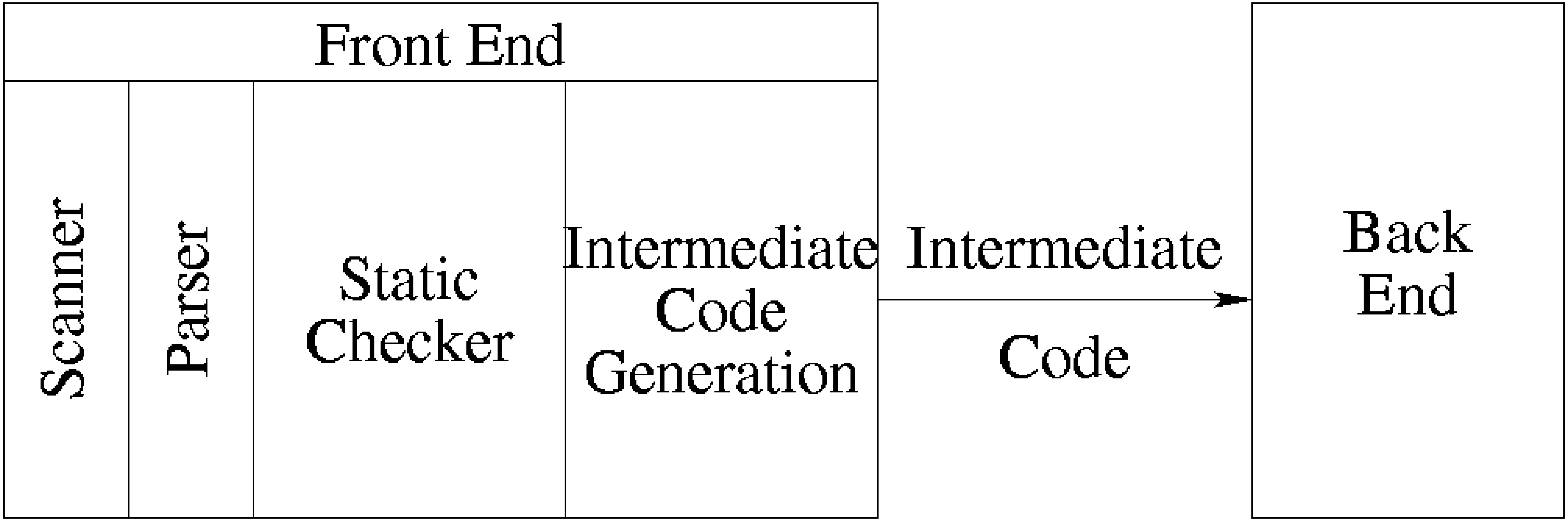
# Concepts Introduced in Chapter 6

- 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



# 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

*followed by Fig. 6.3, 6.4, 6.6*

# 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

*followed by Fig. 6.8*

# Types of Intermediate Representation

- 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

# 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*

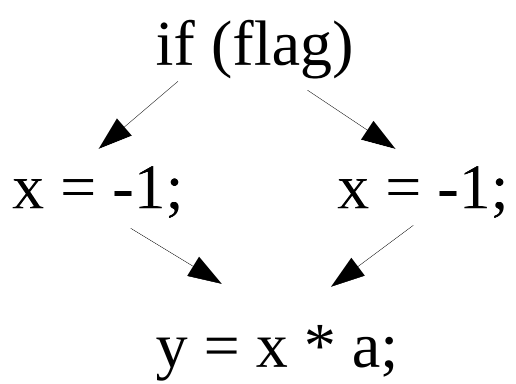
# Types of Three-Address Code (cont...)

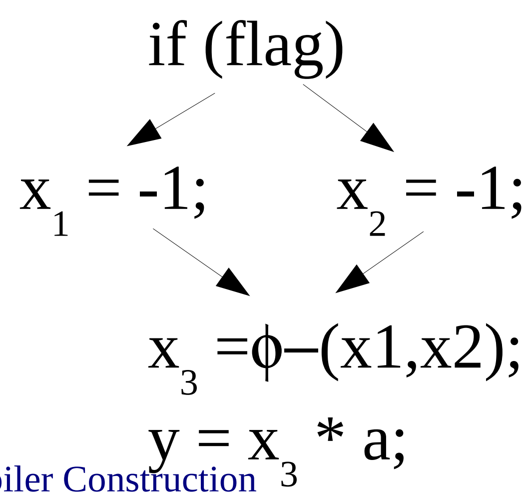
- Triples
  - similar to quadruples, but implicit results and temporary values
  - *result* of an operation is referred to by its position
  - triples avoid symbol table entries for temporaries, but complicate rearrangement of code.
  - indirect triples allow rearrangement of code since they reference 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
- ϕ–function to combine multiple variable definitions




*followed by Fig. 6.13*

# 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 |

# 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.

# 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:

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

# 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 *if*-Statement

- Consider statement:

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

$$
\begin{array}{ll}
 & \text{if } x < 100 \text{ goto } L_2 \\
 & \text{goto } L_3 \\
L_3: & \text{if } x > 200 \text{ goto } L_4 \\
 & \text{goto } L_1 \\
L_4: & \text{if } x \mathrel{!=} y \text{ goto } L_2 \\
 & \text{goto } L_1 \\
L_2: & x = 0 \\
L_1: &
\end{array}
$$

# 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 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);
    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.)

```
if_stmt      :   IF cexpr THEN m seq_of_stmts n elsif_list0
                     else_option END IF m ';'

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

# 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 ith 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

# 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

# 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

**if** *f* has type s → t **and** *x* has type *s*,
**then** expression *f(x)* has type *t*

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

**if** *f(x)* is an expression
**then** for some α and β, *f* has type α → β **and** *x* has type α

# Example of a Simple Type Checker

| Production | Semantic Rule |
|---|---|
| $P \rightarrow D; E$ | |
| $D \rightarrow D; D$ | |
| $D \rightarrow id : T$ | { addtype(id.entry, T.type); } |
| $T \rightarrow char$ | { T.type = char; } |
| $T \rightarrow integer$ | { 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); } |
| $E \rightarrow literal$ | { E.type = char; } |
| $E \rightarrow num$ | { E.type = integer; } |

# Example of a Simple Type Check (cont.)

| Production | Semantic Rule |
|---|---|
| $E \rightarrow id$ | { E.type = lookup(id.entry); } |
| $E \rightarrow E_1$ mod $E_2$ | { 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( ); } |

# 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

```
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