Module 7
Compilation
Program Optimization

CS 332 Organization of Programming Languages
Embry-Riddle Aeronautical University
Daytona Beach, FL

M07 Outcomes

At the end of this module you should be able to …

1. State the purpose of the optimization phase of the compiler.
2. State the purpose of intermediate program representations.
3. Describe at least one optimization that is not covered in class.
4. Distinguish between static and dynamic binding
5. Provide examples of static and dynamic binding
6. Given a situation where dynamic binding must be used, explain why static binding will not work

Compilation Overview

- Authors differ on steps based on emphasis of the text
- Lexical Analysis – Determining the role tokens play
- Syntactical Analysis – Determining if a statement/program is legal in the language (statement construction)
- Semantic Analysis – Determining if a statement/program is compatible (statement behavior) with other statements and program usage
- Optimizations – Manipulating code sequences to reduce computational cycles and/or memory interactions
- Code generation – Generating the machine level instructions
- Assembly – Connecting various parts of the program

Static and Dynamic Binding

- Static binding: any binding that can occur prior to program execution.
- Dynamic binding: any binding that must occur during program execution.

```
x = 10;

z = <user input>;

if (bfoo()) {
    y = 54;
} else {
   y = 42;
}
```

Can we determine the binding for x prior to program execution?
Can we for z?
Can we for y?

Static and Dynamic Binding

- Static binding: any binding that can occur prior to program execution.
- Dynamic binding: any binding that must occur during program execution.

```
public class A {

  public void foo() {
    <method definition>
  }

}
```

```
public class B extends A {
  @Overwrite
  public void foo() {
    <new method definition>
  }

}
```

```
public class C {
A myObject;

  if (bfoo()) {
    myOBject = new A();
  } else {
    myObject = new B();
  }


  x = myObject.foo();

}
```

Which version of foo() is executed in the line “x = myObject.foo();”?

Can we determine statically?

DEF and USE

- DEF: The point where a <var>, <value> binding is DEFined.
- USE: The point(s) where a <var>, <value> binding is USEd.
- Liveness: the points in a program where a specific DEF is in effect.
- Optimizations can manipulate statement sequences as long as the DEF-USE pairings are not modified.
- For x: Lines 1, 4 are DEF, lines 2, 4, 5 are USE
- Concept used in intermediate representations, optimizations.

```
1. x = <user input>;

2. y = 2 * x;

3. z = x * x;

4. x = <user input>;

5. p = x + 10;
```

DEF and USE

- DEF and USE sequencing limits the changes compilers may make.
- Which of the changes are allowable (Programs B, C, D)?
- What happens in a parallel environment?

```
Program A:
1. x = <user input>;
2. y = 2 * x;
3. z = x * x;
4. x = <user input>;
5. p = x + 10;
```

```
Program B:
1. x = <user input>;
2. z = x * x;
3. y = 2 * x;
4. x = <user input>;
5. p = x + 10;
```

```
Program C:
1. x = <user input>;
2. y = 2 * x;
3. z = x * x;
4. p = x + 10;
5. x = <user input>;
```

```
Program D:
1. x = <user input>;
2. y = 2 * x;
3. x = <user input>;
4. z = x * x;
5. p = x + 10;
```

Intermediate Program Representations

- Definition: An intermediate program representation is any form that does not include the original source code nor the final machine executable code.
  - Intermediate forms are not seen by the programmer or the end user – only intermediate processes.

- Purpose: Intermediate representations are used to perform analysis (syntactical, semantic, optimizations) on the program during the compilation process.

- Must be independent of programming language (or paradigm).

- Typical forms: parse trees, flow graphs, program dependence graphs, three-address code.

Intermediate Representation: Three-Address Code

- Represents actions in terms with at most three operands.
- Used as part of transformation to machine code.
- Minor “keyhole optimizations” may be applied (example: optimizing register use)

```
# Calculate one solution to the [[Quadratic equation]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

Blatantly stolen from https://en.wikipedia.org/wiki/Three-address_code

Intermediate Representation: Program Dependence Graph

- Dependencies: relationships between program statements that require sequencing to be maintained for program input-output behavior to be maintained.

- (For those who have taken Operating Systems course): map directly to the four read-write collisions.

- Programs may be modified as long as the output, flow, and control dependencies (defined on following slides) are maintained.

- Widely used intermediate form.

Intermediate Representation: Program Dependence Graph

Definition 6 (Output Dependence) *B is output dependent on A iff the execution of A occurs before B in a strict execution semantics program sequence, and both A and B assign to the same variable.*

Definition 7 (Anti-Dependence) *Statement B is anti-dependent on statement A iff A precedes B in a sequential execution, and B assigns a value to a variable used as input in A.*

Definition 8 (Control Dependence) *B is control dependent on A iff*

1) A is a program control flow statement containing a predicate expression that will evaluate to Boolean True or False.

2) B executes upon either A's evaluation to True or False, but not both.

3) There are no intervening statements for which (1) and (2) apply to B.

From "A Sparse Program Dependence Graph for Object-Oriented Programming Languages“ PhD Dissertation, K. Garfield, 2006

Intermediate Representation: Program Dependence Graph

Definition 9 (Flow Dependence) *Statement B is flow dependent on A iff A is a DEF and B a USE statement for the same program variable, and there are no intervening DEF statements for that variable between A and B on some control flow path from A to B.*

Definition 10 (Def-Order Dependence) *B is Def-Order dependent on A iff*

1) Both A and B are DEF statements for the same program variable.

2) A precedes B in a strict execution sequence.

3) There is some statement C that is flow dependent upon both A and B.

From "A Sparse Program Dependence Graph for Object-Oriented Programming Languages“ PhD Dissertation, K. Garfield, 2006

Intermediate Representation: Program Dependence Graph

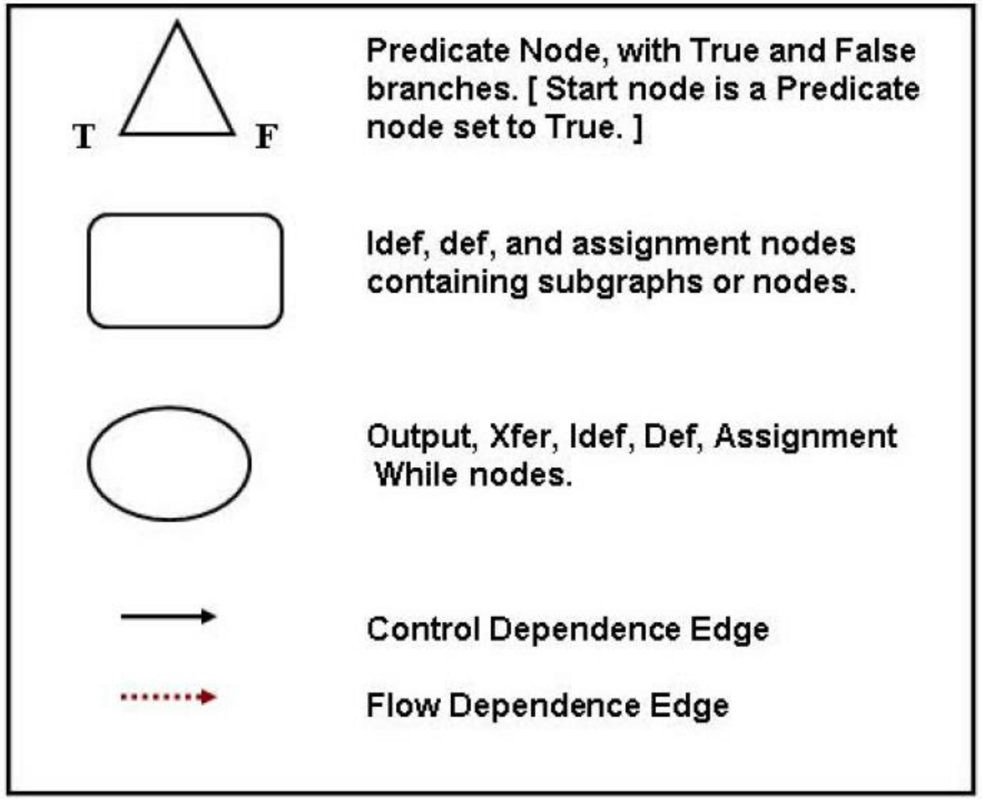
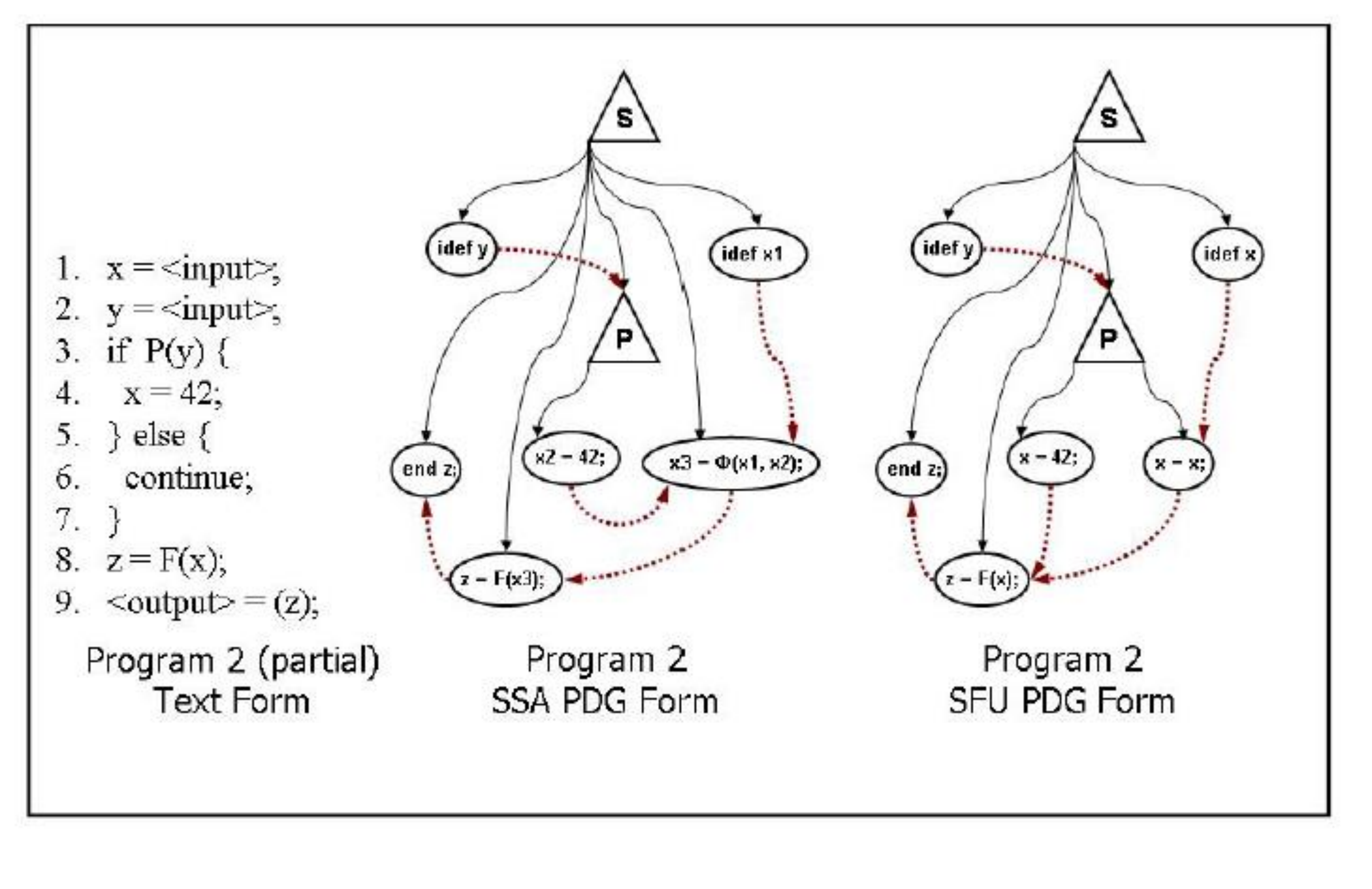


Figure 2.3: Key to Elements Used in Example Graphs

From "A Sparse Program Dependence Graph for Object-Oriented Programming Languages“ PhD Dissertation, K. Garfield, 2006

Intermediate Representation: Program Dependence Graph



SSA = Static Single Assignment.

SFU = Single Flow to USE.

From "A Sparse Program Dependence Graph for Object-Oriented Programming Languages“ PhD Dissertation, K. Garfield, 2006

Optimization Example: Constant Propagation

- If a computation can be performed during compilation, then do it!

```
x = 10;

y = 5;

z = x * y;
```

```
z = 50;
```

If x, y, z used in other places in the program all occurrences may be replaced with values, not memory references.

```
blah;
blah;
if (x > 0) {
  z = foo();
} else {
  z = 0;
}
blah;
blah;
```

```
blah;
blah;
z = foo();
blah;
blah;
```

If constant propagation, or axiomatic semantic analysis, proves “x>0” is always true, eliminated the if-then-else and all costs of performing the Boolean test and resetting program counter.

Optimization Example: Loop Unrolling

- Loops can see inefficiencies when checking whether to stop looping.
  - Computations cycles used to perform the Boolean test.
- Loop unrolling reduces the number of times the loop stopping condition is checked.

```
for (i=0; i<n; i++) {
   myArray[i] = foo();
}
```

```
for (i=0; i<n; i=i+5) {
  myArray[i] = foo();
  myArray[i+1] = foo();
  myArray[i+2] = foo();
  myArray[i+3] = foo();
  myArray[i+4] = foo();
}
```