ELEC 876: Software Reengineering (Program Comprehension)

Dr. Ying Zou

Department of Electrical & Computer Engineering
Queen’s University

Compiler and Interpreter

- Compiler

  Source Code → Compile → Object Code → Execute → Results

  Compile time  Execute time

- Interpreter

  Source Code → Interpret → Results

  data
The structure (phases) of a compiler
Lexical Analysis

```plaintext
position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`.
2. The assignment symbol `:=`.
3. The identifier `initial`.
4. The plus sign.
5. The identifier `rate`.
6. The multiplication sign.
7. The number `60`.

The structure (phases) of a compiler
1. Any identifier is an expression
2. Any number is an expression
3. If expression₁ and expression₂ are expressions, then so are
   expression₁ + expression₂
   expression₁ * expression₂
   ( expression₁ )
The structure (phases) of a compiler

Semantics

Analysis

Synthesis

Source code

Symbol table

Lexical analyzer

Syntax analyzer

Semantic analyzer

Generate code

Improve code

Object code

Error handler

Semantic Analysis

(a)

position := initial + rate 60

(b)

position := initial * rate inttoreal 60

Fig. 1.5. Semantic analysis inserts a conversion from integer to real.
Program Representation Schemes – Basic

- Program representation schemes are chosen on the basis of objectives and tasks to be performed. Some popular program representation schemes are:
  - Abstract syntax trees
  - Control flow graphs
  - Data flow graphs
  - Structure charts
  - Module interconnection graphs
  - Informal information
  - Documentation
Abstract Syntax Trees

- Abstract Syntax Trees: offer a translation of an input string in terms of operands and operators, omitting superficial details such as which productions were used, syntactic properties of the input string etc.
- To produce an AST:
  - Lexical Analyzer: turns input strings into tokens
  - Grammar: turns sequences of tokens into abstract syntax trees (AST)
  - Domain model: defines the nodes and the arcs that are allowable in the abstract syntax tree
  - Linker: annotations the AST nodes and arcs with global information (i.e., data types, static variables, scoping information etc.)

Abstract Syntax Trees (con’t)

- Abstract Syntax Trees offer a good way to represent programs
  - do not emphasize on particular properties of the program
  - maintain all necessary information to generate further abstraction (i.e., Control Flow Graphs)
Example: define an example grammar for a subset of CALC, an imaginary language

grammar:
add-expression ::= add-arg1 “+” add-arg2
add-arg1 ::= integer-constant
add-arg2 ::= integer-constant
integer-constant ::= integer-value

We compile the grammar above, we can parse the text 1 + 2 to produce the abstract syntax tree:
Control Flow Graphs

- Control Flow Graphs: offer a way to eliminate variations in control statements by providing a normalized view of the possible flow of execution of a program.
  - Basic Blocks are nodes of a control flow graph
  - Arcs denote possible transfer of control from one basic block to another
- To provide a Control Flow Graph:
  - The AST of the program
  - A decomposition of a program into Basic Blocks combined with knowledge on the behavior of the control statements of the source language

Basic Block

- One of the first steps for analyzing a program is to subdivide the program into basic blocks
- A basic block is a sequence of consecutive instructions that are executed from start to finish without halt or the possibility of branching except at the end.
- A basic block can be entered at the first instruction and left at the last.
- The instruction that begins a basic block is called *leader instruction*
Basic Block (con’t)

• Once the program is subdivided into blocks, each block can be analyzed using local techniques
• Three address instructions are said to define and use (or reference) variables.
• For example, x=y+z
  – defines x and uses y and z

Three Address Instructions (example)

```
begin
  prod := 0;
i := 1;
do begin
  prod := prod + a[i] *b[i];
i := i + 1;
end
while i<= 20
end
```

Program to compute dot product

Three-address code computing dot product
Algorithm for finding Basic Blocks

Algorithm Partition into basic blocks

Input. A sequence of three-address statements

Output. A list of basic blocks with each three-address statement in exactly one block

Method.
1. We first determine the set of leaders, the first statements of basic blocks
   The rules we use are the following
   i) The first statement is a leader
   ii) Any statement that is the target of a conditional or unconditional goto is a leader
   iii) Any statement that immediately follows a goto or conditional goto statement is a leader

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Three Address Instructions (example)

Program to compute dot product

Three-address code computing dot product
Control Flow Graph

• A control flow graph is a directed graph that is constructed by adding flow-of-control information to the set of basic blocks making up a program.
  – The nodes of the flow graph are the basic blocks.
  – The node that contains a leader as the program’s first statement is called initial node

Control Flow Graph (con’t)

• There is a directed edge from block $B_1$ to block $B_2$ if $B_2$ can immediately follow $B_1$ in some execution sequence; that is if:
  – There is a conditional or unconditional jump from the last statement of $B_1$ to the first statement of $B_2$ or
  – $B_2$ immediately follows $B_1$ in the order of the program and $B_1$ does not end in an unconditional jump.
  – We say that $B_1$ is a predecessor of $B_2$, and $B_2$ a successor of $B_1$
Example Control Flow Graph

void quicksort(m, n) {
    int i, j, v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m - 1;
    j = n;
    v = a[n];
    while (1) {
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);
        if (i >= j) break;
        x = a[i];
        if t5 > v goto B3
        a[t6] := x
        goto B2
        if t1 = v goto B6
        a[t10] := x
        goto B2
        t1 := 4 * i
        j := n
        t2 := 4 * i
        t3 := a[t2]
        if t3 < v goto B2
        t4 := 4 * j
        t5 := a[t4]
        if t5 > v goto B3
        x := a[i];
        a[i] = a[j];
        a[j] = x;
    }
    x = a[i];
    a[i] = a[n];
    a[n] = x;
    /* fragment ends here */
    quicksort(m, j);
    quicksort(i + 1, n);
}

Elec 876: Software Reengineering - Program Comprehension
Loops

- In a flow graph we may have loops. In general a loop is defined as a collection of nodes in a flow graph such that:
  - All nodes in the collection are strongly connected; that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop.
  - The collection of nodes has a unique entry, that is a node in the loop such that the only way to reach a node of the loop from a node outside the loop is first to through the entry.

Loops (con’t)

```
prod := 0
i := 1

B1

\[ t_1 := 4 \times i \]
\[ t_2 := a[t_1] \]
\[ t_3 := 4 \times i \]
\[ t_4 := b[t_3] \]
\[ t_5 := t_2 \times t_4 \]
\[ t_6 := prod + t_5 \]
\[ prod := t_6 \]
\[ t_7 := i + 1 \]
\[ i := t_7 \]
\[ \text{If } i \leq 20 \text{ goto } B_2 \]
```

B2
Dominators

- We say that a node d of a flow graph dominates node n
  - Written $d \text{ dom } n$,
  - If every path from the initial node of the flow graph to n goes through d.
  - Under this definition every node dominates itself, and the entry of a loop dominates all nodes in the loop

- A useful way to represent dominator information is in a tree, called the dominator tree in which
  - The initial node is the root
  - Each node d dominates only its descendants in the tree.
Dominators (con’t)

• The existence of dominator trees follows from the property of dominators:
  – Each node has a unique immediate dominator \( m \) that is the last dominator of \( n \) on any path from the initial node to \( n \).
  – In terms of the \( \text{dom} \) relation, the immediate dominator \( m \) has the property that if \( d \neq n \) and \( d \text{ dom} n \) then \( d \text{ dom} m \)

Data Flow Graphs

• Data Flow Graphs: offer a way to eliminate unnecessary control flow constraints, focusing mostly on the exchange of information between program components (i.e., basic blocks, functions, procedures, modules)

• To produce a Data Flow Graph:
  – The AST of the program
  – A decomposition of a program into Basic Blocks
  – Combined with annotations on uses and definitions of variables
Structure Charts

• Structure Charts: offer a way to represent data and control information in a concise and compact form.
  – Nodes represent program or system entities (i.e., basic blocks, procedures, modules)
  – Arcs are annotated with control and data flow information

• To produce a Structure Chart:
  – The Control Flow Graph of a program
  – The Data Flow Graph of the program
  – Representations from which you can compute the interprocedural data flow

Structure Charts (con’t)
Module Interconnection Graphs

• Module Interconnection Graphs: offer a way to represent data coupling and data dependencies between program and system entities

• To produce a Module Interconnection Graph:
  – The Structure Chart of a program
  – Some information of parameter passing between procedures or functions

Module Interconnection Graphs (con’t)

<table>
<thead>
<tr>
<th>Number</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aircraft type</td>
<td>Status flag</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
<td>List of aircraft parts</td>
</tr>
<tr>
<td>3</td>
<td>Function code</td>
<td>--</td>
</tr>
<tr>
<td>4</td>
<td>--</td>
<td>List of aircraft parts</td>
</tr>
<tr>
<td>5</td>
<td>Part number</td>
<td>Part manufacturer</td>
</tr>
<tr>
<td>6</td>
<td>Part number</td>
<td>Part name</td>
</tr>
</tbody>
</table>

Module Interconnection Diagram

\[ p, t, u \text{ access the same database in update mode} \]
Call Graphs

- A graph in which
  - Nodes represent individual procedures
  - Edges represent call sites
  - Each edge is labeled with the actual parameters associated with that call site

1. program Sums
2. read(n);
3. i := 1;
4. while i ≤ n
5. sum := 0;
6. Acc(sum,i);
7. write(sum,i);
8. i := i + 1;
9. endwhile
10. end.

11. procedure Acc(x,y)
12. ref x, y
13. while j<y
14. Acc(x,j);
15. Inc(j);
16. endwhile
17. return

18. procedure Inc(x)
19. Add(x,1);
20. return

21. procedure Add(a,b)
22. a := a + b;
23. return

Call Graphs (con’t)

- A call graph is a multi-graph with more one edge connecting two nodes in the case of one procedure calling another at many points
- A call graph represents the procedural structure of a program and illustrates the calling relationship among procedures, it is useful during maintenance for program understanding
- A call graph is also useful for interprocedural data flow analysis
  - Interprocedural data flow analysis algorithms that use a call graph are flow insensitive since they do not consider the control flow of individual procedures
Program Summary Graphs

- Program Summary Graphs (PSG) are modifications of the call graphs by taking into account control and data flow in individual procedural call.
- A PSG contains information about actual reference parameters and global variables at call sites, and formal reference parameters and global variables at procedure entry and exit points.
- There are two kinds of edges in the PSG:
  - **Binding edges** that relate actual and formal reference parameters and
  - **Reaching edges** that summarize the flow of data between procedural control points such as entry, exit, call and return for formal and actual parameters.

```
1. program Sums
2.     read(n);
3.     i := 1;
4.     while i ≤ n
5.         sum := 0;
6.         Acc(sum,i);
7.     endwhile
8.     write(sum);
9.     end.
10.    procedure Acc(x,y)
11.       ref x, y
12.       j := 1;
13.       while j ≤ y
14.           add(x,j);
15.       endwhile
16.     return
17.    procedure Inc()
18.       ref z
19.       return
20.    procedure Add(a,b)
21.       ref a, value b
22.     a := a + b;
23.     return
```

Fig. 3. The program summary graph for Sums in Fig. 2. Circles represent call/return nodes, double circles represent entry/exit nodes, solid lines represent call/return binding edges and dashed lines represent reaching edges. Node numbers facilitate presentation and do not correspond to the line numbers of Fig. 2. The correlation between node pairs and lines is shown in the table to the right of Fig. 3.
Data Flow Analysis

• Data flow analysis provides the framework for analyzing the behavior of programs in terms of their data flow properties
• Data flow analysis problems can be solved by setting and solving sets of equations involving data flow properties of a program at the control flow graph (CFG) basic block level
• We will be looking for the following data flow analysis problems
  – Reaching Definitions
  – Use-Def Chains
  – Available Expressions
  – Live Variable Analysis
  – Copy Propagation
  – Inter-procedural analysis of changed variables

Data Flow Analysis Equations

• Data Flow information can be collected by setting and solving systems of equations of the form:

\[ \text{Out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S)) \]

With the interpretation
  – “The information at the end of a block is either generated within the block or enters at the beginning and is not killed as control flows through the block”
Data Flow Analysis Equations (con’t)

- Consider the data flow equation:
  \[ \text{out}(S) = \text{gen}(S) \cup (\text{in}(S) - \text{kill}(S)) \]

- **Gen(S):** A definition \(d\) is in Gen(s) if \(d\) reaches the end of \(S\) via a path that does not go outside of \(S\)

- **Kill(S):** Is the set of definitions that never reach the end of \(S\). In order for \(d\) to be in Kill(S), every path from the beginning to the end of \(S\) must have an unambiguous definition of the same variable defined by \(d\), and if \(d\) appears in \(S\), then following every occurrence of \(d\) along any path, must be another definition of the same variable

- **In(S):** The information flowing in the block

- **Out(S):** The information exiting the block

Example Data Flow Analysis Equations

- \(B_1\)
  - \(d_1: i:= m-1\)
  - \(d_2: j:= n\)
  - \(d_3: a:= u1\)
  - \(\text{gen}[B_1] = \{d_1, d_2, d_3\}\)
  - \(\text{kill}[B_1] = \{d_4, d_5, d_6, d_7\}\)

- \(B_2\)
  - \(d_4: i := i -1\)
  - \(d_5: j := j - 1\)
  - \(\text{gen}[B_2] = \{d_4, d_5\}\)
  - \(\text{kill}[B_2] = \{d_1, d_2, d_7\}\)

- \(B_3\)
  - \(d_6: a:= u2\)
  - \(\text{gen}[B_3] = \{d_6\}\)
  - \(\text{kill}[B_3] = \{d_3\}\)

- \(B_4\)
  - \(d_7: i:= u3\)
  - \(\text{gen}[B_4] = \{d_7\}\)
  - \(\text{kill}[B_4] = \{d_1, d_4\}\)
Solution of Data Flow Equations

- The most common way to solve Data Flow Equations is via iterative solutions.
- Iterative solutions to Data Flow equations require:
  - Build the Control Flow Graph
  - Compute In, Out for each Basic Block
- There are two types of equations:
  - Forward equations: Out sets are computed in terms of In sets
  - Backward equations: In sets are computed in terms of Out sets
Reaching Definitions

- The idea behind *reaching definitions* is that given a program point p find all definitions d, (i.e., read, or assignment statements of variables) that reach p.

If definition d that defines a Variable x is not killed then d reaches P

If there is another definition d₁ of variable x then d is killed.
int g(int m, int i);

int f(n)
{
    int n;
    int i = 0, j;
    if (n == 1) i = 2;
    while (n > 0) {
        j = i + 1;
        n = g(n, i);
    }
    return j;
}

Iterative Algorithm for Reaching Definitions

• For each basic Block b we can define In(B), Out(B), Gen(B), Kill(B) as follows:
  \[\text{In}(B) = U_p\text{Out}(P)\]
  – Where P is a predecessor of B

• If a flow graph has n nodes we get 2n equations

• The iterative algorithm for Reaching Definition is give in next slide
**Input.** A flow graph for which \( \text{kill}[B] \) and \( \text{gen}[B] \) have been computed for each block \( B \).

**Output.** \( \text{in}[B] \) and \( \text{out}[B] \) for each block \( B \)

**Method.** We use an iterative approach, starting with the “estimate” \( \text{in}[B] = \emptyset \) for all \( B \) and converging to the desired values of \( \text{in} \) and \( \text{out} \). As we must iterate until the \( \text{in} \)’s (and hence the \( \text{out} \)’s) converge, we use a boolean variable \( \text{change} \) to record on each pass through the blocks whether any \( \text{in} \) has changed.

/* initialize \( \text{out} \) on the assumption \( \text{in}[B] = \emptyset \) for all \( B \) */

for each block \( B \) do \( \text{out}[B] := \text{gen}[B] \)

\( \text{change} := \text{true} \);

while \( \text{change} \) do begin

\( \text{change} := \text{false} \)

for each block \( B \) do begin

\( \text{in}[B] := \bigcup_{\text{in \ is a predecessor of } B} \text{out}[P]; \)

\( \text{oldout} := \text{out}[B]; \)

\( \text{out}[B] := \text{gen}[B] \cup (\text{in}[B] – \text{kill}[B]) \)

If \( \text{out}[B] \neq \text{oldout} \) then \( \text{change} := \text{true} \)

end

Algorithm to compute \( \text{in} \) and \( \text{out} \)
Available Expressions

- An expression $E$ (i.e., $x+y$) is available at a program point $P$, if every path from the initial node to point $P$, evaluates $E$, and after the last such evaluation prior to reaching $P$, there are no subsequent assignments to the components of $E$ (i.e., $x$ or $y$ for $x+y$).
- For available expressions we say that a block $B$ kills an expression $E$ (i.e., $x+y$) if it assigns one or more of its components (i.e., $x$ or $y$) and does not subsequently redefines $E$.
- A block $B$ generate expression $E$ if it definitely evaluates $E$ and does not subsequently redefines one or more of the components of $E$ (i.e., $x$ or $y$).
- For the calculation of the available expressions we assume $U$ be the universal set of all expressions appearing in a program $A$.
Available Expression Algorithm

**Input.** A flow graph G with c_kill[B] and c_gen[B] computed for each block B. The initial block is B_1.

**Output.** The set in[B] for each block B.

**Method.**

\[
in[B_1] := \emptyset
\]

/* in and out never change for the initial node, B_1 */

\[
out[B_1] := c_gen[B_1];
\]

/* initial estimate is too large */

for B ≠ B_1 do

\[
out[B] := U – c_kill[B]
\]

change := true;

while change do begin

\[
change := false
\]

for B ≠ B_1 do begin

\[
in[B] := \bigcap \text{ out}[P]
\]

\[
oldout := \text{ out}[P]
\]

\[
\]

if out[B] ≠ oldout then change := true

end

end
Live Variable Analysis

- Live Variable analysis is a typical analysis that depends on information that is computed in the direction opposite to the flow of control in a program.
- In Live Variable analysis we wish to know for variable $x$ and point $P$, whether the value of $x$ at $P$, could be used along some path in the flow graph starting at $P$. In other words is $x$ used after $P$?

Live Variable Analysis (con’t)

- We formulate this Data Flow analysis problem with the following set of equations:

  $\text{In}(B) = \text{Use}(B) \cup (\text{Out}(B) - \text{Def}(B))$

  $\text{Out}(B) = U_S \text{In}(S)$

  Where:
  - $S$: is a successor basic block of $B$
  - $\text{In}(B)$: is the set of live variables at the beginning of block $B$
  - $\text{Out}(B)$: is the set of live variables when existing the block $B$
  - $\text{Def}(B)$: is the set of variables, definitely assigned values in $B$ prior to any use of that variable in $B$
  - $\text{Use}(B)$: is the set of variables whose values may be used in $B$ prior to any definition of the variable
Live Variable Analysis Algorithm

**Input.** A flow graph with def and use computed for each block.

**Output.** out[B], the set of variables live on exit from each block B of the flow graph

**Method.**

for each block B do in[B] := ∅
while changes to any of the in’s occur do
  for each block B do begin
    out[B] = ∪ in[S]
  end
Copy Propagation

- It is sometimes possible to eliminate copy statements $S$ (i.e. $x := y$) if we determine all places where this definition is used. We may then substitute $y$ for $x$, provided the following conditions are met by every such use $u$ of $x$
  1. Statement $S$ must be the only definition of $x$ reaching $u$ (that is, the $u$-chain for use $u$ consists only of $S$).
  2. On every path from $S$ to $u$, including paths that go through $u$ several times (but do not go through $S$ a second time), there are no assignments to $y$.
- Condition (1) can be checked using $u$-chaining information.
- For Condition (2) we define a new Data Flow analysis problem.

Copy Propagation (con’t)

- Let $\text{In}(B)$ be the set of copies (i.e. $S$: $x := y$) such that every path from the initial node to the beginning of $B$, contains the statement $S$, and subsequent to the last occurrence of $S$, there are no assignments to $y$.
- Let $\text{Out}(B)$ be defined correspondingly but with respect to the end of $B$.
- We say that the copy statement $S$: $x := y$, is **generated** in block $B$, if it occurs in $B$, and there is no subsequent assignment of $y$ within $B$.
- We say that the copy statement $S$: $x := y$, is killed in $B$, if $x$ or $y$ is assigned and $S$ is not in $B$.
- Notes:
  - That an assignment to $x$ kills $x := y$, is similar to the reaching definitions problem.
  - That $y$ to do so is special to this problem.
  - $\text{In}(B)$ can contain only one copy statement with $x$ on the left.
  - Different assignment $x := y$ kill each other.
Copy Propagation (con’t)

- Let $U$ be the universal set of all copy statements in a program. It is important to note that different copy statement $x:=y$ are different entries in $U$.
- Let's define:
  - $C_{\text{gen}}(B)$ to be all copies generated in block $B$.
  - $C_{\text{kill}}(B)$ to be all copies in $U$ but killed in block $B$.
- The Copy propagation equations are formulated as follows:
  - $\text{Out}(B) = C_{\text{gen}}(B) \cup (\text{in}(B) - C_{\text{kill}}(B))$
  - $\text{In}(B) = \bigcap_p \text{out}(P)$ for $B$ not initial, where $P$ is a predecessor block of $B$.
  - $\text{In}(B_1) = \emptyset$

Example: Consider the flow graph of the figure.
Here, $C_{\text{gen}}[B_1] = \{x:=y\}$ and $C_{\text{gen}}[B_3] = \{x:=z\}$. Also, $C_{\text{kill}}[B_2] = \{x:=y\}$ since $y$ is assigned in $B_2$. Finally, $C_{\text{kill}}[B_1] = \{x:=z\}$ since $x$ is assigned in $B_1$ and $C_{\text{kill}}[B_3] = \{x:=y\}$ for the same reason.

The other $C_{\text{gen}}$'s and $C_{\text{kill}}$'s are $\emptyset$. Also, $\text{in}[B_1] = \text{out}[B_1]$ by equation.
Algorithm in one pass determines that

$$\text{in}[B_2] = \text{in}[B_3] = \text{out}[B_4] = \{x:=y\}$$

Likewise, $\text{out}[B_2] = \emptyset$ and

$$\text{out}[B_3] = \text{in}[B_4] = \text{out}[B_4] = \{x:=z\}$$

Finally, $\text{in}[B_5] = \text{out}[B_2] \cap \text{out}[B_4] = \emptyset$.
Copy Propagation Example (con’t)

• From the previous example we observe:
  – Neither copy $x=y$ nor $x=z$ reaches the use of $x$ in $B_5$
  – It is not possible to substitute $y$ (respectively $z$) in all uses of $x$
    that definition $x = y$ (respectively $x = z$) reaches
  – We could have substituted $x$ for $z$ in $B_4$

Copy Propagation Algorithm

**Input.** A flow graph $G$ with ud-chains giving the definitions reaching block $B$, and which $c_{in}[B]$ representing the solution to, that is, the set of copies $x:= y$ that reach block $B$ along every path, with no assignment to $x$ or $y$ following the last occurrence of $x := y$ on the path. We also need du-chains giving the uses of each definition.

**Output.** A revised flow graph

**Method.** for each copy $s: x:= y$ do the following
1. determine those uses of $x$ that are reached by this definition of $x$, namely, $s: x:= y$.
2. determine whether for every use of $x$ found in (1) $s$ in in $c_{in}[B]$, where $B$ is the block of this particular use., and moreover, no definitions of $x$ or $y$ occur prior to this use of $x$ within $B$. Recall that if $s$ is in $c_{in}[B]$, then $s$ is the only definition of $x$ that reaches $B$.
3. if $s$ meets the conditions of (2), then remove $s$ and replace all uses of $x$ found in (1) by $y$. 