CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the
Winter 2015/2016 term at the University of Toronto

Reading Assignment

Fischer, Cytron and LeBlanc

Sections 10.3, 14.7

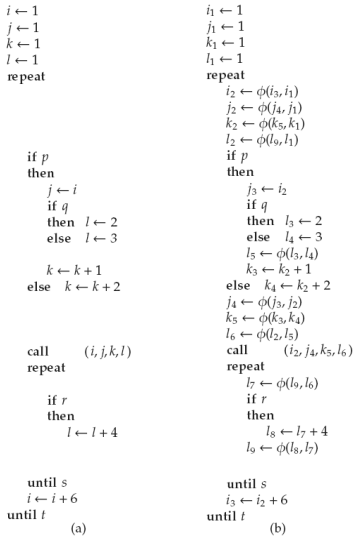
Sections 14.2 .. 14.2.3

Sections 14.3 .. 14.3.4

Static Single Assignment

- The Static Single Assignment (SSA) form is an IR used to facilitate certain kinds of optimization.
- Key features of SSA
  - Each assignment to a variable is a **definition** of the variable for the particular value being assigned.
  - *Rename* all variables in the program using some systematic scheme so that each variable is assigned a value exactly *once*.
  - If more than one definition for the same variable is available at some point in the program, add an *articulation point* (ϕ function) to identify the conflict.
  - Once the variables have been renamed, each uniique variable has fixed value from its point of definition to the end of the program
- The SSA form facilitates optimizations like constant propagation and constant folding (Slide 446) and variable folding (Slide 449)

SSA Example (Figure 10.5)



Data Flow Analysis

- Data Flow Analysis is a technique for discovering properties of the run-time behavior of programs during compilation.
- These properties are *universal*, i.e. they apply to all possible sequences of control and data flow.
- The information from data flow analysis is used to determine feasibility and safety of various optimizations.
- Control Flow Graphs ($\mathcal{G}_{cf}$) are used to represent the flow of control between statements in a single routine.
  Procedure Call Graphs ($\mathcal{G}_{pc}$) are used to represent the flow of control between functions and procedures.
- Data Flow Analysis and related optimizations are covered in much greater detail in the follow-on course ECE489H/ECE540H .

Analysis and Transformations

Data Flow Analysis	Transformation
Available expressions	Common subexpression elimination
Detection of loop invariants	Invariant code motion
Detection of induction variables	Strength reduction
Copy analysis	Copy propagation
Live variables	Dead code elimination
Reaching definitions	Identifying constants
. . .	. . .

Data Flow Analysis Overview

- Subdivide the program into Basic Blocks or finer grain (e.g. statements).
- Analyze the control structure of the program to determine the interrelationship of the blocks.
- Represent program control flow as a directed graph ($\mathcal{G}_{cf}$) .
  - Each node in $\mathcal{G}_{cf}$ is a basic block or an individual statement.
  - Each edge in $\mathcal{G}_{cf}$ represents a possible node to node transfer of control.
- Analyze each node to determine where some property of interest is defined, used and killed. e.g. where expressions are computed.
- Dataflow analysis can be applied

  Intraprocedurally within a single routine

  Interprocedurally within some set of routines

  Globally to an entire program
- Sets are represented using bit-vectors (one bit/element).
  Iterative solutions to data flow equations are typically $O(B^2 \times V)$ for a problem with B basic blocks and V variables or expressions.

Definitions for Data Flow Analysis

Basic Block A sequence of instructions that is always executed from start to finish. i.e. it contains no branches.

Definition Point Point where some interesting property is established. (i.e. a variable or an expression is given a value.)

Use Point Point where some interesting property is used.

Killed, Kill Point A point where in interesting property is rendered invalid.

- A variable is assigned a new value.
- *Any* of the variables used to compute the value of an expression is assigned a new value.

Forward Flow What can happen before a given point in a program.

Backward Flow What can happen after a given point in the program.

Any Path (May) analysis What property holds on some path leading to or from a given basic block. Examples: uninitialized variable, live variables.

All Path (Must) analysis What property holds on all paths leading to or from a basic block. Example: availability of an expression.

Predecessors(*b*) the set of all basic blocks that are *immediate* predecessors of block *b* in the control flow graph.

There is an edge in the control graph leading *from* each basic block in Predecessors(*b*) *to b*.

Successors(*b*) the set of all basic blocks that are *immediate* successors of block *b* in the control flow graph.

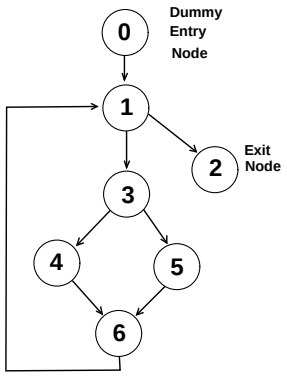
There is an edge in the control graph leading *from b to* each basic block in Successors(*b*).

Basic Block Example

Program and Basic Blocks

```
while ( 1 ) {
    scanf( " %f %f %f", &A, &B, &C ) ;                    1
    if( A == 0.0 )
        break ;                                           2
    DISC = B * B - 4.0 * A * C ;                          3
    if( DISC >= 0.0 ) {
        DROOT = sqrt( DISC ) ;
        R1 = ( - B + DROOT ) / ( 2.0 * A ) ;              4
        R2 = ( -B  - DROOT ) / ( 2.0 * A ) ;
    } else {
        DROOT = sqrt( - DISC ) ;
        R1 = - B / ( 2.0 * A ) ;                          5
        R2 = DROOT / ( 2.0 * A ) ;
    } ;
    printf( "%f %f %f ", DISC, R1, R2 ) ;                 6
} ;
```

Control Flow Graph



Predecessors

1	{ 0 , 6 }	4	{ 3 }
2	{ 1 }	5	{ 3 }
3	{ 1 }	6	{ 4 , 5 }

Successors

1	{ 2 , 3 }	4	{ 6 }
2	{ }	5	{ 6 }
3	{ 4 , 5 }	6	{ 1 }

Types of Data Flow Problems

Forward Flow problems

- Data flows from the first block to the last block.
- *out* sets are computed from *in* sets within basic blocks.
- *in* sets are computed from *out* sets across basic blocks.

Backward Flow problems

- Data flows from the last block to the first block.
- *in* sets are computed from *out* sets within basic blocks.
- *out* sets are computed from *in* sets across basic blocks.

Any Path problems

Values coming into a block through *any* path are valid.

Use $\bigcup$, start with minimum info[a]

All Path problems

Only values coming into a block through *every* possible path are valid.

Use $\bigcap$, start with maximum info[a]

[a] For first/last block other value may be necessary

Generic Data Flow Analysis

- Data flow analysis can be used to compute a wide variety of properties of basic blocks that are useful for optimization.
- Generic definitions

  $In(b)$ what properties hold on entry to basic block b.

  $Out(b)$ what properties hold on exit from basic block b.

  $Gen(b)$ the properties that are generated in basic block b.

  $Killed(b)$ the properties that are invalidated in basic block b.

- For each problem of interest, the precise rules for determining membership in the sets $In, Out, Gen, Killed$ have to be specified.
- Once the membership rules have been determined, one of the iterations over the control flow graph described on the following slide is used to determine In and Out for all basic blocks in the control flow graph.

Generic Forward Data Flow Equations

Any Path	$Out(b) = Gen(b) \bigcup (In(b) - Killed(b))$ $In(b) = \bigcup_{i \in Predecessors(b)} Out(i)$
All Paths	$Out(b) = Gen(b) \bigcup (In(b) - Killed(b))$ $In(b) = \bigcap_{i \in Predecessors(b)} Out(i)$

Generic Backward Data Flow Equations

Any Path	$In(b) = Gen(b) \bigcup (Out(b) - Killed(b))$ $Out(b) = \bigcup_{i \in Successors(b)} In(i)$
All Paths	$In(b) = Gen(b) \bigcup (Out(b) - Killed(b))$ $Out(b) = \bigcap_{i \in Successors(b)} In(i)$

Forward Data Flow Algorithm (All Paths)

for each block B **do**
    $out[B]$:= $gen[B] \cup (in[B] - kill[B])$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $in[B]$:= $\bigcap_{P \in Pred(B)} out[P]$
        oldout := $out[B]$
        $out[B]$:= $gen[B] \cup (in[B] - kill[B])$
        if $out[B] \neq$ *oldout* **then** *change* := true
    end
end

Available Expression Analysis
Forward All-Paths Analysis

- An expression is *available* at a point P in the program graph G if *every* path leading to P contains a definition of the expression which is not subsequently killed.
- Definitions

$In(B)$ the set of expressions available on entry to basic block B .

$Kill(B)$ the set of expressions that are killed in basic block B .

$Gen(B)$ the set of expressions that are defined in basic block B and are not subsequently killed within the block.

$Out(B)$ the set of expressions available at the exit of block B

$In(Init)$ is $\emptyset$

- Use Available Expression Analysis to implement global common sub-expression elimination.

An expression is *available* at the end of a basic block B if

it is defined in the block and not subsequently killed.

or it is available on entry to the block B and is not killed within the block.

Therefore

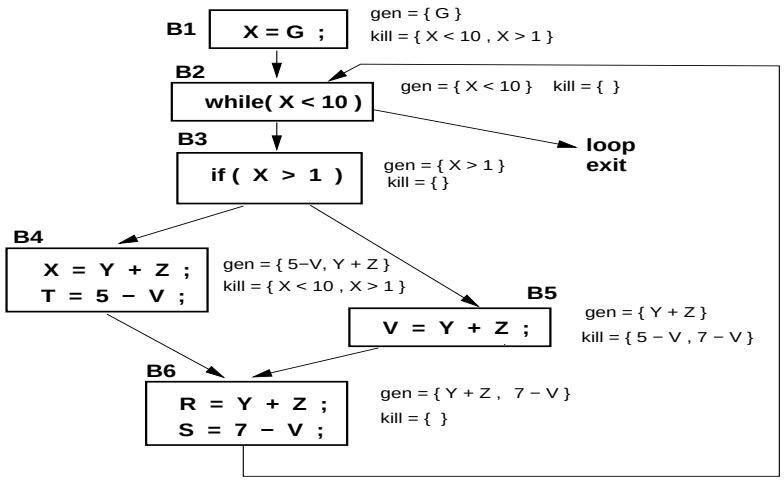
$$Out(B) = gen(B) \bigcup (In(B) - Kill(B))$$

An expression is available on entry to a basic block if it is available on *all* paths leading into the block.

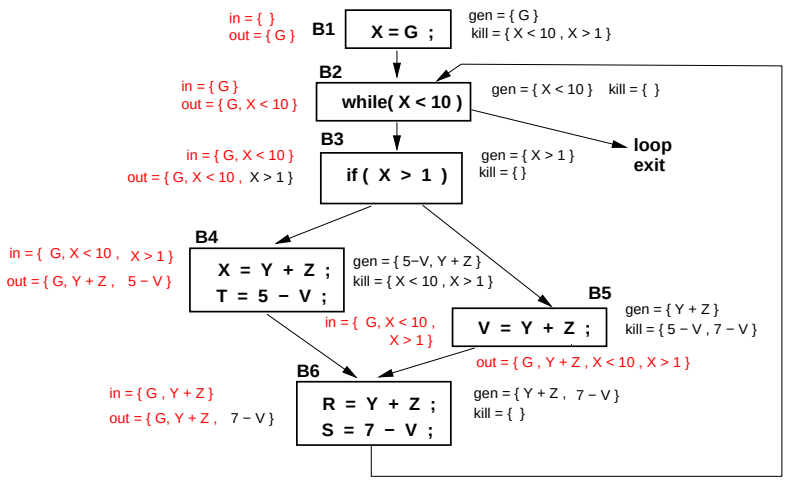
For all basic blocks B compute

$$In(B) = \bigcap_{p \, \varepsilon \, Predecessors(B)} Out(\, p \,)$$

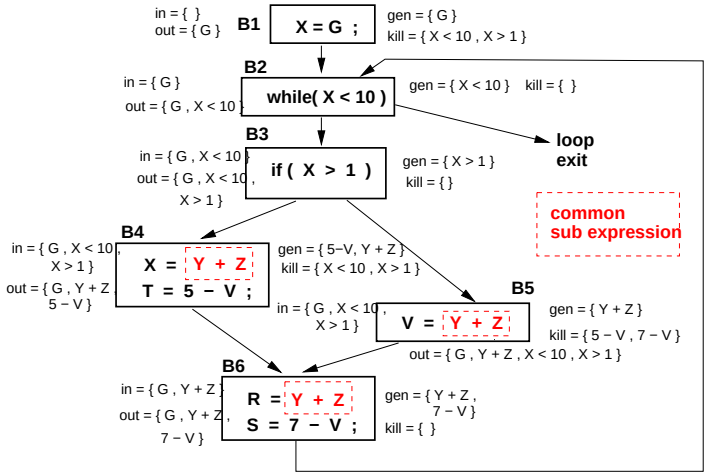
Available Expressions Example



Available Expressions Example



Common Sub−Expression Elimination Example



Forward Data Flow Algorithm (Any Path)

for each block B **do**
    $out[B] := gen[B]$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $in[B] := \bigcup_{P \in Pred(B)} out[P]$
        oldout := $out[B]$
        $out[B] := gen[B] \cup (in[B] - kill[B])$
        if $out[B] \neq$ *oldout* **then** *change* := true
    end
end

Reaching Definitions
Forward Any-Path Analysis

- A *definition* of a variable x is a statement that assigns a value to x
- A definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path
- Definitions

$In(B)$ definitions available upon entry to B

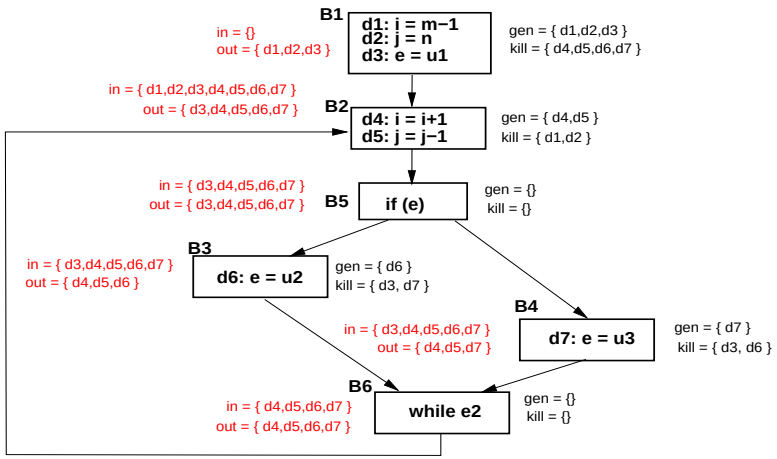
$Gen(B)$ definitions made in B

$Kill(B)$ definitions invalidated by B

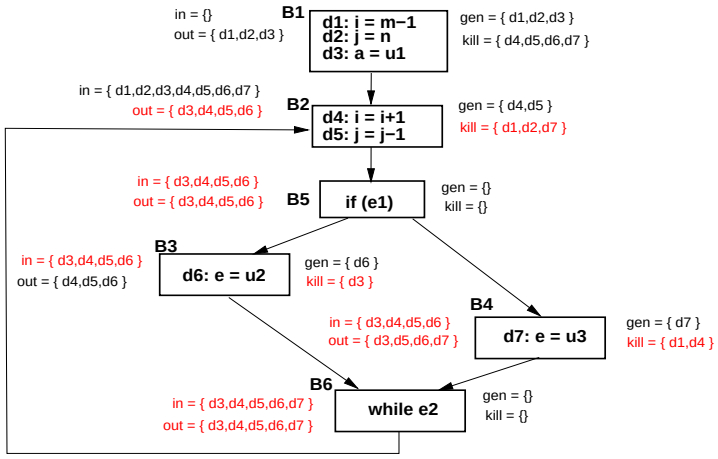
$Out(B)$ is $Gen(b) \bigcup (In(B) - Kill(B))$

$In(Init)$ is $\emptyset$

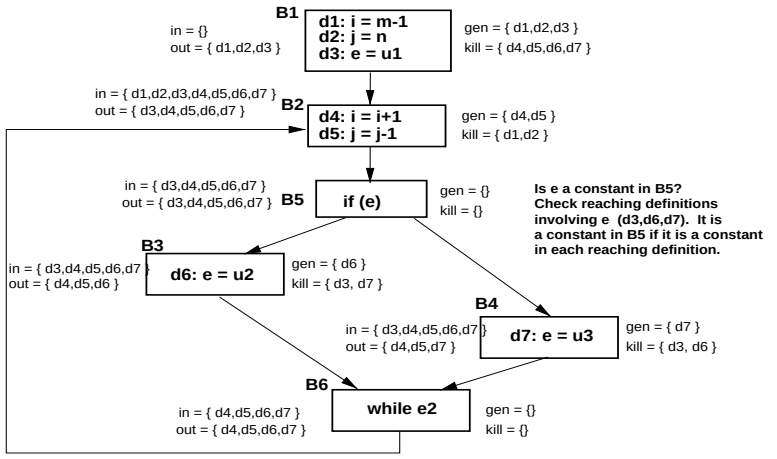
Reaching Definitions Example



Reaching Definitions Example (2nd Iteration)



Identifying Constants



Backward Data Flow Algorithm (Any Path)

for each block B **do**
    $in[B] := gen[B]$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $out[B] := \bigcup_{P \in Succ(B)} in[P]$
        oldin := $in[B]$
        $in[B] := gen[B] \cup (out[B] - kill[B])$
        if $in[B] \neq$ *oldin* **then** *change* := true
    end
end

Live Variable Analysis
Backward Any Path Data Flow Example

- For each definition/use of a variable V, Live Variable Analysis answers the question:

  Could the value of V computed/used here be used *further on* in the program?

- Used to enable certain kinds of optimizations
  - If V is being stored in a register over some stretch of code it is not necessary to store the register back into V if V is not live.
    Assignments to non-live variables can be deleted.
  - Duration of liveness can be used to pick variables that should be loaded into registers.
  - Example

```
r23 = V        /* V stored in a register */
r23 = r23 + 1  /* V = V + 1              */
/* no uses of V below here */
/* no need to store r23 back into V */
```


Live Variable Analysis
Backward Any-Path Analysis

- A path in the graph G is $V - clear$ if it contains no assignments to the variable V.
  A variable V is *live* at a point P in the graph G if there is a $V - clear$ path from P to a use of V, i.e. there is a path from P to somewhere that V is used which does *not* contain a redefinition of V
- Definitions

$in(B)$	Variables live on entrance to block B
$out(B)$	Variables live on exit from block B.
$def(B)$	Variables assigned values (redefined) in block B *before* the variable is used. (same as $Kill(B)$)
$use(B)$	Variables whose values are used before being assigned to. (same as $Gen(B)$)
$Out(Final)$	is $\emptyset$

A variable V is live at the entrance to a block B if

it is used in B *before* it is defined in B

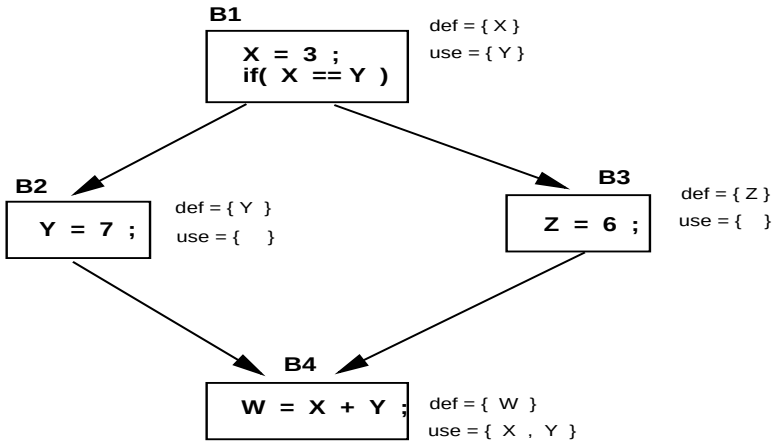
or it is live at the exit to block B and it is not defined within the block

$$in(B) = use(B) \bigcup (out(B) - def(B))$$

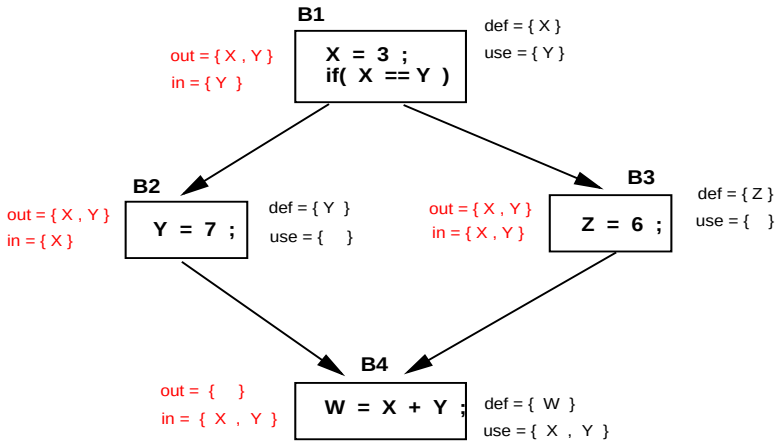
A variable V is live coming out of a block B if it is live going into any of B's successors.

$$out(B) = \bigcup_{S \in successors(B)} in(S)$$

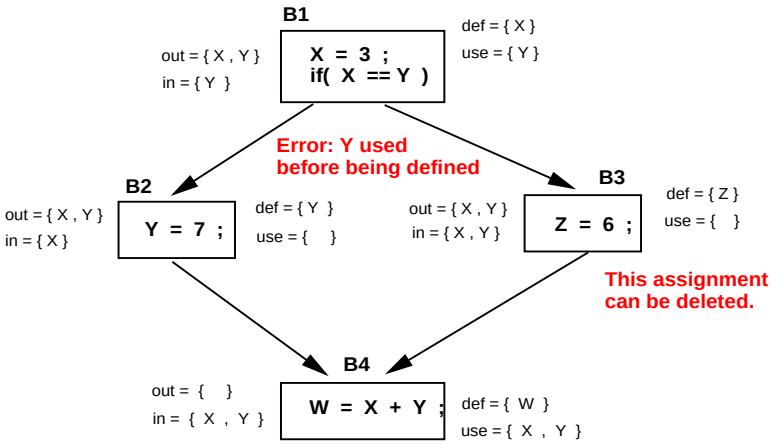
Live Variable Analysis Example



Live Variable Analysis Example



Live Variable Analysis Example



Backward Data Flow Algorithm (All Path)

for each block B **do**
    $in[B] := gen[B] \cup (U - kill[b])$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $out[B] := \bigcap_{P \in Succ(B)} in[P]$
        oldin := $in[B]$
        $in[B] := gen[B] \cup (out[B] - kill[B])$
        if $in[B] \neq$ *oldin* **then** *change* := true
    end
end

Very Busy Expressions
Backward All-Paths Analysis

- An expression *e* is *very busy* at point p if not matter what path is taken from p, the expression *e* will be evaluated before any of its operands are redefined.
- Definitions

$Out(B)$ expressions very busy after B

$Gen(B)$ expressions constructed by B

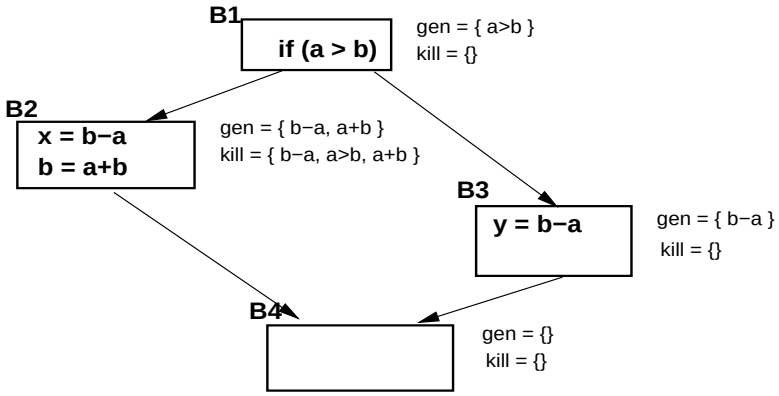
$Kill(B)$ expressions whose operands are redefined in B

$In(B)$ is $Gen(b) \bigcup (Out(B) - Kill(B))$

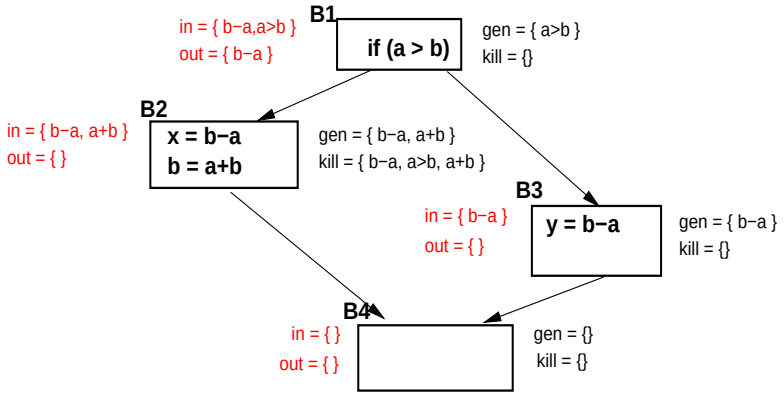
$Out(Final)$ is $\emptyset$

- Use Very Busy Expressions analysis to *hoist* expressions

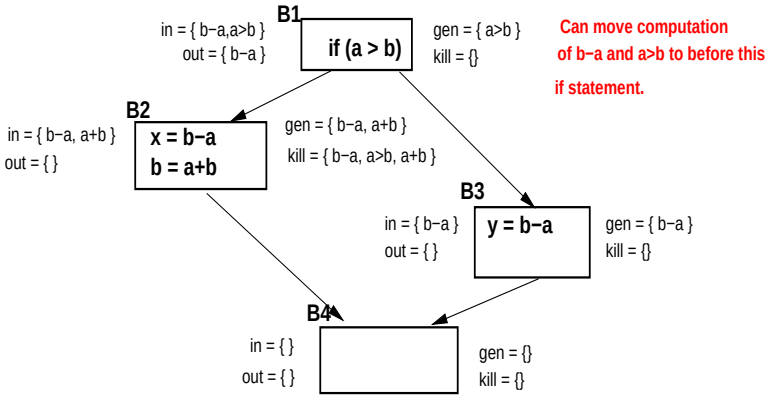
Very Busy Expression Example



Very Busy Expression Example



Code Hoisting

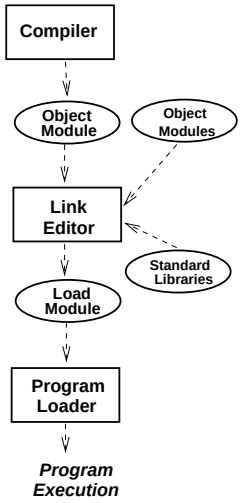


Linking and Loading[a]

An Object module produced by a compiler is linked together with other object modules and standard libraries to form a complete executable program.
Issues dealt with by the linking process

- Resolution of external symbol references
- Relocation of code addresses
- Support for separate compilation
- Support use of standard libraries
- Creation of loadable module

Loading reads a complete executable program (`a.out` or `a.exe`) from disk into memory and starts its execution..



[a]See: John R. Levine, *Linkers & Loaders*, Morgan Kaufmann, 2002

Object Module Information

A typical object module may contain

- Header information. e.g. code size, name of source file, creation date, etc.
- Object code. Binary machine instructions and data.
  Data may include initialized data (e.g. constants) and uninitialized space for static storage.
- Relocation information. Places in the object code that need to be modified when the linker places the object code.
- Symbols
  – **Exported names** defined in the object module.
  – **Imported names** used in the object module
- Debugging information. Information about the object module that is useful to a runtime debugger. e.g. source file coordinates, internal symbol names, data structure definitions.

Linking is a Two Pass Process

Input: a collection object modules and libraries

First Pass

- Scan input files to determine size of all code and data segments.
- Build a symbol table for all imported or exported symbols in all input files
- Resolve all connections between imported symbols and exported symbols
- Maps object code segments to image of output file
- Assign relative addresses in this output block to all symbols

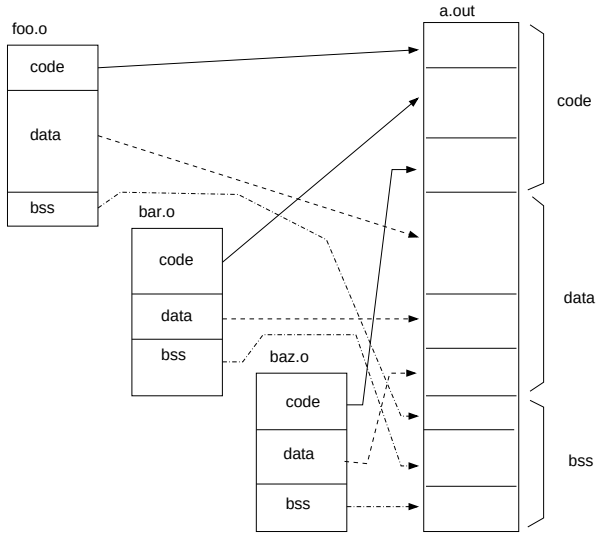
Resolving Symbols

- Every external symbol used in an object module should be resolved to a symbol exported from some other object module or library.
- Unresolved symbol are usually a link time error.
- If multiple instances of a external symbol are available there are several strategies:
  – emit a link time error message
  – pick one symbol with a warning message
  – silently pick one symbol

Second Pass

- Read and relocate object code and data to the output block.
- Replace symbolic addresses with block relative numeric addresses.
- Adjust memory addresses in the object code to reflect new location in the output block.
- Write output file
  – header information
  – relocated code and data segments
  – symbol table information
  – debugging information

Linking Example



Loading

- In most modern systems, the load module can simply be read into a block of memory and then executed. No load time relocation is required.
- To make this possible compilers must generate position independent code e.g. branches are relative to the program counter, data access is via a base register.
- Dynamic linking allows some binding of symbols to library modules to be deferred until runtime.

Dynamic Linking and Loading

- Defer much of the linking process until a program is loaded into memory and starts execution.
- Advantages
  - Dynamically linked shared libraries are easier to create.
  - Dynamic libraries are easier to update without required extensive relinking.
  - Dynamic linking allows runtime loading and unloading of modules.
- Disadvantages
  - Substantial runtime overhead for dynamic linking
  - Dynamic linking redone for every execution.
  - Dynamic link libraries are larger than static libraries due to symbol information.
  - Changes to/unavailability of dynamic libraries may make programs unexecutable.

Compiler Design Issues

- Like any large, long-lived program a compiler should be designed in a modular fashion that is easy to maintain over time.
- Need to design a software architecture for the compiler that allows it to implement the required language processing steps.
- A production compiler generally must implement the *entire* language. Student project and prototype compilers often omit the hard parts.
- Architecture of the compiler will be influenced by
  - The programming language being compiled.
  - Characteristics of the target machine(s).
  - Compiler design goals
  - Compiler's operating environment.
  - Compiler project management goals

Programming Language Influences on Compiler Structure

- Declaration before use?
- Typed or type less?
- Separate compilation? modules/objects?
- Lexical issues, designed to be lexable?
- Syntactic issues, designed to be parseable?
- Static semantic checks required? Implementable?
- Run-time checking required?
- Size of programs to be compiled?
- Compatibility with OS or other languages?
- Dynamic creation/modification of programs?

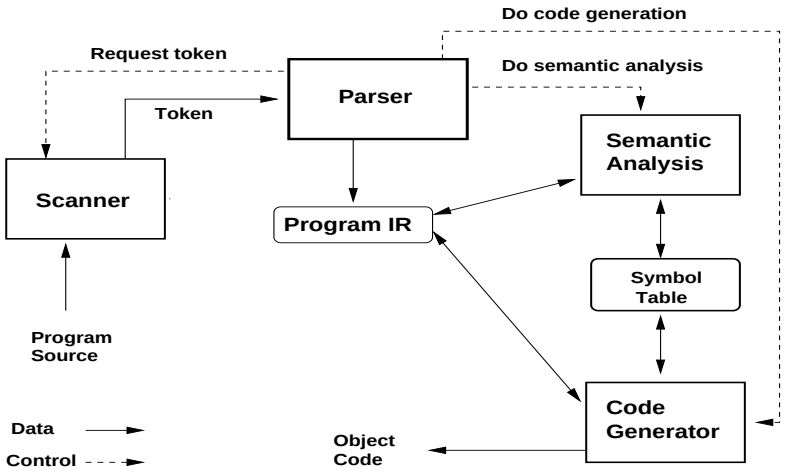
Target Machine Influences on Compiler Structure

- Limited or partitioned memory
- RISC vs. CISC instruction set.
- Irregular or incomplete instruction set.
- Inadequate addressing modes.
- Hardware support for run-time checking?
- Poor support for high level languages.
- Missing instruction modes?
- Inadequate support for memory management?

Single Pass Compilers

- Good for simpler languages (e.g. prototypes and course projects).
- Not feasible for languages that permit backward information flow (e.g. declaration *after* use).
- Single pass compilers are usually parser-driven.
  The parser coroutines with the lexical analyzer to obtain tokens.
  The parser calls semantic analysis and code generation routines as each construct in the language is parsed.
- By the time each declaration or statement is completely parsed, all semantic checks have been performed and all code has been generated.
- Pascal is an example of a language well suited to single pass compilation.
  Declaration before use, well designed declaration structure, simple control structures.

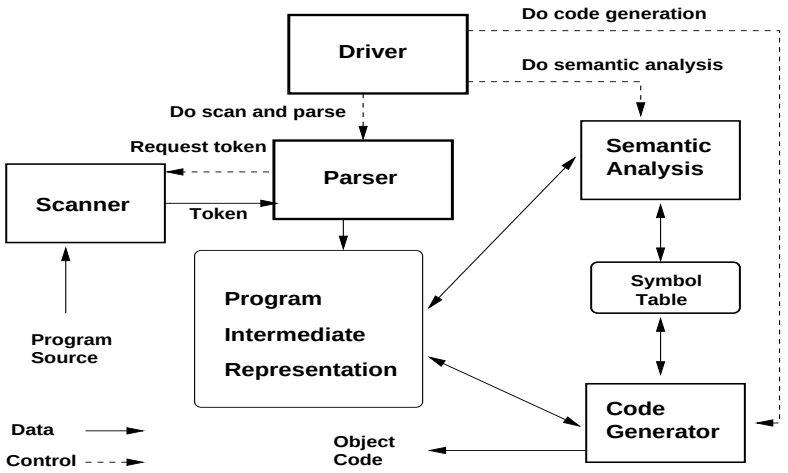
Single Pass Compiler Architecture



Multi Pass Compilers

- Multi pass compilers make several complete passes over the source program.
- Typically the first pass does lexical and syntax analysis and builds some intermediate representation of the program.
- Semantic analysis is a separate pass that processes this intermediate representation.
- Code generation is a separate pass that processes the intermediate representation.
- Optimization may make multiple passes over the program.
- Used for more complicated languages, e.g. Cobol, Ada, C, C++, Java
- Can couple lexical/syntax analysis for multiple languages to a common backend. Many compilers use gcc code generator as a back end itemActual situation can be more complicated
  - multiple intermnediate representations
  - multiple passes for semantic analysis/code gen

Multi Pass Compiler Architecture



Multi Pass Compiler Issues

- Compiler implementation effort increases with the number of passes.
- Compiler speed decreases with the number of passes.
- A multi pass compiler does much more input and output to communicate between passes.
- Efficient internal representations of programs become more important as size of compiler/language increases.
  Memory resident tables are faster than disk resident tables.
- Partitioning a compiler into an efficient multi-pass structure is often a difficult software engineering task.

Some Compiler Design Goals

- **Correctly implement the language.**
- Be highly diagnostic and error correcting.
- Produce time or space optimized code.
- Be able to process very large programs.
- Be very fast or very small.
- Be easily portable between environments.
- Have a user interface suitable for inexperienced users.
- Emit high quality code.

Example Compiler Goals

- Student Compiler
  - Interface for inexperienced users.
  - Be highly diagnostic at compile time and run-time.
  - Compile with blinding speed.
  - Do *no* optimization
- Production Compiler
  - Interface for experienced users.
  - Produce highly optimized object code.
- Quick and Dirty Compiler
  - Minimize compiler construction time
  - Minimize project resource usage and budget.
  - Do no optimization, omit hard parts of language.
  - Compile to interpretive code, assembly language or high-level language.

Compiler Design Choices

- Organization of compiler processing
  - Single pass or multiple pass?
- Choice of compiler algorithms
  - Lexical and syntactic analysis
  - Static semantic analysis, code generation
  - Optimization
- Compiler data representation
  - Symbol and/or type tables, dictionaries.
  - Memory resident compiler data?
  - Communication between passes?
  - Format of compiler output?

Compiler Output Choices

- Assembly language (or other source code)
  Let existing assembler do some of the hard work.
  Makes code generation easier. Used in early C compilers.
- Relocatable machine code Usually an object module.
  Allows separate compilation, linking with existing libraries.
- Absolute machine code
  Generated code is directly executable.
- Interpretive pseudo code
  Machine instructions for some virtual machine. Used for portability and ease of compilation.
- High level programming language
  Example Specialized language $\rightarrow$ C

Compiler Design Examples

- Student compiler
  - One pass for speed
  - In-memory tables
  - Compile to directly executable absolute code or to interpretive code.
  - Tune for compile speed and high quality diagnostics.
- Production Optimizing Compiler
  - Usually multi pass
  - Uses disk resident tables for large programs.
  - Data structures tuned for large programs.
  - Usually includes heavyweight optimization.

User Interfaces for Compilers

- The importance of good *Human Engineering* in designing the interface between the compiler and the user cannot be over emphasized.
- **The compiler should describe problems with a program in terms that the user can understand.**

Good: Syntax error on line 100 of file myProgram.c
The reserved word **if** cannot be followed by the identifier *myVar*

Bad: Illegal symbol pair **if** *identifier* . Parse stack dump:
123 ＜identifier＞
122 **if**
121 ＜statement＞
120 ＜block head＞
...

Really Bad: Syntax error in program. Compilation terminated.

Unacceptable: java.lang.NullPointerException: null
or Segmentation fault - core dump

What the Compiler Should Do

- Determine if the program is correct.
- Describe the statically detectable errors at compile time.
- Translate the source program into an equivalent object program.
- Emit code to detect and describe dynamically detectable errors during program execution.
- Language designer (should) specify the possible errors in a language.
  Note the difference between error, implementation defined and unspecified.
- The language implementor decides between static and dynamic detection of errors.
- All errors *should be* detectable.

Error Detection Tradeoffs

- Static detection of errors may greatly increase
  - Complexity of the compilers internal data structures and algorithms
  - The time required to compile all programs.
- Example: To detect aliasing of variables the compiler needs links between all calls of a routine and the definition of the routine.
  The time to do aliasing detection is quadratic in the size of the program.
- Dynamic detection of errors at runtime may greatly increase
  - The size of the program due to extra chacking code (e.g. array subscript checking).
  - The execution time of the program due to the time required to execute checking code.
- Examples of expensive run time checking:
  - Uninitialized variable checking .
  - Array subscript checking .
  - Dangling and Nil pointer errors .

Compiler Reaction to Errors in Programs

Bad: Compiler Crash or Infinite Loop.

Bad: Java execption stack trace.

Bad: Generate incorrect object program and no error messages.

Poor: Stop the compilation

Good: Recover from the error and continue compilation.
Hard to do well. Error cascades and false errors are problems.

Great: Repair error and continue the compilation
Perfect correction is undecidable.
Use heuristics and systematic strategies.

Errors should be detected as soon as possible.
Localize (in a routine) the production of error messages.
Optionally print error summary at end of compilation
13 Errors detected, Last error on line 219 of file urProg.c

How the Compiler Can Help the User

- Use symbol table to produce information for the programmer
  - Cross reference list for identifier definition and use.
  - Frequency of usage information for identifiers.
  - Diagnostics for possible errors
    Variable/constant declared but never used.
    Variable assigned to but never used.
  - List sizes of data structures. Warn of excessive fill in data structures.
  - List code size and usage information for routines.
- Identify potentially inefficient constructs for the user.
  Statement 415 in yourProg.c generated 600 bytes of code.
  Statement 311 in utility.c implemented by 10 calls to library routines.
- Optionally summarize compiler internal resource usage.
  Used 400 of 511 symbol table entries.
  Provides warning of possible overrun of compiler limits.

- Provide user control over compiler processing options
  Example: `gcc` (see `man gcc` for the gory details).
  Good:
  - allows user control over optimization
  - provides work around for compiler problems

  Bad:
  - can become *very* complicated[a]
    For example gcc has

13 general options	19 C language options
3 language independ options	40 C++ language options
80 warning options	65 debugging options
131 optimization options	hundreds of machine dependent options

  - more than most users will ever use
  - makes software maintenance harder, since compiler options change compiled program

[a] Care to guess what the gcc option `-fmudflap` means?

Compiler Should be Self Diagnostic

- Compiler should be *error imune*, i.e. never crash for *any* possible input.
- Compiler must be programmed to detect *all* violations of compiler internal limits. Do internal consistency checking of compiler data structures.
- Use exception handlers to trap programming errors and shut down gracefully.
  An internal error has occurred. Please file a bug report.
- Optionally print out compiler internal performance statistics
  10,301 identifier lookups performed. 73% were local scope.
  4,219 scanner tokens produced, 3745 nodes in abstract syntax tree.
  143,216 bytes of code generated, 40,219 instructions.
  10.37 seconds in semantic analysis, 21.7 seconds in code generation.
- Optionally provide information (perhaps embedded in the object program) about the options used in the compilation
  Compiled on 2010-03-22 at 14:00:01 using superCompiler version 4.3.21
  Options -noCheckArray -O17 -stuffStructs -fastStrings

Example: `perl -V`

```
Summary of my perl5 (revision 5 version 14 subversion 2) configuration:
  Platform:
    osname=linux, osvers=2.6.42-37-generic, archname=x86_64-linux-gnu-thread-multi
    uname='linux batsu 2.6.42-37-generic #58-ubuntu smp thu jan 24 15:28:10 utc 20
    config_args='-Dusethreads -Duselargefiles -Dccflags=-DDEBIAN -Dcccdlflags=-fPI
    hint=recommended, useposix=true, d_sigaction=define
    useithreads=define, usemultiplicity=define
    useperlio=define, d_sfio=undef, uselargefiles=define, usesocks=undef
    use64bitint=define, use64bitall=define, uselongdouble=undef
    usemymalloc=n, bincompat5005=undef
  Compiler:
    cc='cc', ccflags ='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -p
    optimize='-O2 -g',
    cppflags='-D_REENTRANT -D_GNU_SOURCE -DDEBIAN -fno-strict-aliasing -pipe -fsta
    ccversion='', gccversion='4.6.3', gccosandvers=''
    intsize=4, longsize=8, ptrsize=8, doublesize=8, byteorder=12345678
    d_longlong=define, longlongsize=8, d_longdbl=define, longdblsize=16
    ivtype='long', ivsize=8, nvtype='double', nvsize=8, Off_t='off_t', lseeksize=8
    alignbytes=8, prototype=define
  Linker and Libraries:
    ld='cc', ldflags =' -fstack-protector -L/usr/local/lib'
    libpth=/usr/local/lib /lib/x86_64-linux-gnu /lib/../lib /usr/lib/x86_64-linux-g
    libs=-lgdbm -lgdbm_compat -ldb -ldl -lm -lpthread -lc -lcrypt
    perllibs=-ldl -lm -lpthread -lc -lcrypt
    libc=, so=so, useshrplib=true, libperl=libperl.so.5.14.2
    gnulibc_version='2.15'
```

```
  Dynamic Linking:
    dlsrc=dl_dlopen.xs, dlext=so, d_dlsymun=undef, ccdlflags='-Wl,-E'
    cccdlflags='-fPIC', lddlflags='-shared -O2 -g -L/usr/local/lib -fstack-protect
Characteristics of this binary (from libperl):
  Compile-time options: MULTIPLICITY PERL_DONT_CREATE_GVSV
                        PERL_IMPLICIT_CONTEXT PERL_MALLOC_WRAP
                        PERL_PRESERVE_IVUV USE_64_BIT_ALL USE_64_BIT_INT
                        USE_ITHREADS USE_LARGE_FILES USE_PERLIO USE_PERL_ATOF
                        USE_REENTRANT_API
  Locally applied patches:
    <DELETED for brevity>
  Built under linux
  Compiled at Mar 18 2013 19:17:55
  @INC:
    /etc/perl
    /usr/local/lib/perl/5.14.2
    /usr/local/share/perl/5.14.2
    /usr/lib/perl5
    /usr/share/perl5
    /usr/lib/perl/5.14
    /usr/share/perl/5.14
    /usr/local/lib/site_perl
```


The Compiler Should Help the User

- Be correct.
  Generating incorrect code is one of the worst things a compiler can do to a user.
- Provide good diagnostics.
  Describe errors in user-appropriate messages.
  Try to localize the cause of the error as much as possible.
  Provide guidance to help the user correct their program.
  Know when to stop producing error messages if there may be an error cascade.
- Be fast.
  Software developers like fast compiles for large software systems.
- Generate efficient code.
- Guide the user towards good software engineering practices.
  Make parts of the language that encourage good software structure, e.g. functions, procedures, classes efficient so programmers will use them.