COMPILER DESIGN

Thomas H. Payne
Department of Computer Science
University of California, Riverside

May 7, 2003

Contents

CS 152 SYLLABUS

CS152 is an introductory compiler course. It assumes familiarity with C++ programming (including the STL), data structures, automata theory, and principles of programming languages. It covers the design of the front-end of a compiler, including:

- Automatic generators for lexical analyzers: review of the Rabin-Scott algorithm, use of ordered EBNF to describe lexical categories, lookahead, left-context, case-study LEX, compression of state table.
- Automatic generators for LALR parsers: BNF to railroad diagrams to nondeterministic PDA. Construction of LR(0) table, LR(1) table, LALR table from LR(1) table, LALR table by identifying states during construction of LR(1) table, propagation of lookaheads.
- Syntax-directed translation.
- The run-time environment.

Text

The text is **Modern Compiler Implementation in Java (second edition)** by Andrew Appel. The syllabus is:

Chapter 1: Introduction

Chapter 2: Lexical Analysis

Chapter 3: Parsing

Chapter 4: Abstract Syntax

Chapter 5: Semantic Analysis

Chapter 6: Activation Records

Chapter 7: Translation to Intermediate Code

Chapter 8: Basic Blocks and Traces.

Grading

Course grading will be based on: final exam; project; quizzes (unannounced) in roughly equal weights. There will be a term project, writing (in C++, not Java) a compiler that generates three-address C from the MiniJava language, which is described in the appendix of the text.

One must do the project to pass the course. It requires extensive C++ programming, so if your programming skills are rusty, start practicing now.

The TA is in charge of the project portion of the course, and details will be covered during the lab sections. The use of several programming tools will be covered there:

- flex (a freeware version of lex), which is used to generate the scanner.
- bison (a freeware version of yacc), which is used to generate the parser.
- STL (the Standard Template Library), which is part of the C++ standard.
- CVS, which is a database for keeping past versions of documents (especially source code).

You will be required to keep a on-line project notebook that includes test logs. That notebook will be turned in at the end of the quarter.

You must do your own work in terms of programming and testing.

CS 201 SYLLABUS

CS201 is an advanced compiler course. It requires a familiarity with how the front end of a compiler (i.e., the lexical analyzer, parser, and semantic actions) produces a symbol table and intermediate code. In fact, it assumes that the student has had an undergraduate course in compiler design that involved writing a lexical analyzer, parser, semantic actions, and simple code generator.

Outline

We begin by studying the back end of the compiler, and will later show how to automate the production of the front end. The bulk of the time will be spend on back-end topics:

- Code generation: basic blocks, register allocation, global register allocation, code emission, automated code emission.

- Optimization: data flow analysis, validity considerations, code generation with optimized code.

Text

The former text was the dragon book, **Compilers: Principles, Techniques, and Tools** by Aho, Sethi, Ullman. The course covered:

Section 8.1: Intermediate languages

Section 8.2: Declarations

Section 8.3: Assignment statements

Section 9.4: Basic Blocks and Flow Graphs

Section 9.5: Next-use Information

Chapter 10: Code optimization

Chapter 9: Code generation

Section 3.8: Design of a lexical-analyzer generator

Section 4.9: Parser generators

There are two texts that I now recommend but don't require:

- **Modern Compiler Implementation in Java** by Andrew Appel (Cambridge University Press, 1998).
- **Advanced Compiler Design & Implementation** by Steven S. Muchnick (Morgan Kaufman). The following order corresponds roughly to the order for these notes:[1]
  - Chapters 1, 2, 3, and 4
  - Chapter 6
  - Sections 9.1 and 9.2
  - Chapters 16 and 17
  - Chapter 8
  - Chapter 11
  - Chapter 12
  - Chapter 13
  - Chapter 7
  - Chapter 14
  - Chapter 10

Grading

Course grading will be by formula: 30% final; 40% project 30% quizzes. There will be a term project, writing a portion of a code optimizer. It will be assigned about half way through the quarter, after the corresponding material has been covered. It is an individual project requiring extensive programming. This year the project will be done in Python. You must do the project to pass the course, so if your Python skills are rusty, start practicing. There is a good tutorial at www.python.org.

[1] as of 9/28/98

QUESTIONAIRE

```
Place photo ID here and then
photocopy this page so that
photo is clear.


Name: __________________________________ Login: _________________

Status: Graduate, Senior, Junior, other _________________________

Have you had a course in automata theory?           [] Yes  [] No

Have you had a course in programming languages?     [] Yes  [] No

Have you had an undergraduate course in compilers?  [] Yes  [] No

   Where?  ______________________________________________________

   What book was used?  _________________________________________

   Did you write a compiler?  [] Yes  [] No

   What kind of parser did it use?  [] Bottom-up  [] Top-down

Which of the following software tools are you familiar with?

  [] LEX [] YACC [] UNIX [] PASCAL [] C [] C++

Do you own a copy of the text for this course?  [] Yes  [] No
```

Chapter 1

BACKGROUND

1.1 Overview

The name "compiler design" is perhaps a misnomer. A better term would be "translation of programming languages." A programming language is, however, merely a language for getting a computer to do what one wants. Indeed, the interface to any program is a programming language — often a very weak and restrictive one. The techniques learned here will be of general utility. Two important specific applications of these techniques, not normally considered under the heading of "compiler construction," are the following.

Typesetting: When mathematicians write papers, they use a lot of complex symbols and combinations of symbols. In the past, setting mathematical text was very costly, because so much of it had to be set by hand. Now, thanks to the development of automated mathematical typesetting programs like TeX, it is relatively easy to set mathematical manuscripts into type. One uses an ordinary computer terminal to describe how the typeset copy should look, using a language designed for this purpose, e.g., TeX or EQN. These are languages, like programming languages, that must be scanned and parsed. But the output is not low-level code to be interpreted by some low-level machine. Rather, it is a sequence of commands to a typesetter that tell it where to put which symbol. The techniques needed to implement the front-end of such languages are very similar to those needed to implement a programming language.

VLSI design: To design an integrated circuit chip, one must define three or more masks, one for each layer of conductive paths on the chip. There is a very low-level language, CIF, for describing where the slots in the mask are to be cut. Each cut is a rectangle having a certain length, width, position, and rotation on the mask. The problem is that a modern VLSI chip may have millions of transistors and wires on it. To write all of these rectangles in CIF would take too much time and would probably never be done correctly. Hence, we need a higher-level language for describing the chip and a program for translating this higher-level description to CIF. Such programs are called "silicon compilers." It is now possible to give a VHDL description of an algorithm to a silicon compiler and have the compiler produce the CIF for a chip that will implement that algorithm. The difficulties occur in getting efficient implementations — ones that make efficient use of the area of the chip. The techniques used are often those that we will use in the study of optimization: dataflow analysis, basic blocks, register allocation, etc.

1.2 Bootstrapping

We use compilers to build compilers and sometimes the process gets a bit convoluted.

Definitions. The following definitions are need to be studied.

- A program expressed in language L is said to be *L-hosted*, and L is said to be that program's *implementation language*.
- A language is said to be *executable* if it has an interpreter, i.e., a hardware/software system that executes programs expressed in that language. Executable languages are often called "architectures."
- A program is *executable* if and only if its implementation language is executable.
- A compiler is a program for translating programs expressed in its *source language* into semantically equivalent programs expressed in its *target language*, which is assumed to be executable.
- A compiler whose source language is L is said to "compile from L" and to be an *L compiler*. An L compiler whose target language is M is said to "compile from L to M" and to be an *L-to-M compiler*.
- A compiler is said to be *self resident* if its implementation language is the same as its target language — otherwise, it is said to be a *cross compiler*.
- A compiler is said to be *self compiling* if its implementation language is the same as its source language.
- An *optimizing* compiler is one that generates target-language programs that run faster than average.

Fundamental facts. Here are the fundamental relationships among the above notions:

- For a compiler to actually translate programs, it has to be executable.

- Self-resident compilers are executable, since their implementation languages are their target languages, which are assumed to be executable.
- Submitting an L'-hosted L-to-M compiler as input to any executable L'-to-M' compiler yields an M'-hosted L-to-M compiler, which will be executable (since M' is a target language and target languages are presumed to be executable).

Bootstrapping a self-resident compiler. Given a self-compiling L-to-M compiler, C, and an executable L compiler, E, to create an M-hosted self-resident L compiler:

- Generate an executable L-to-M compiler, E', by submitting C to E.
- Generate an M-hosted self-resident L compiler by submitting C to E'.

Bootstrapping to new architectures. Given a self-compiling L-to-M compiler, C, and an executable L compiler, E, to port C to a architecture M', i.e., to obtain an M'-hosted L-to-M' compiler:

- Obtain a self-compiling L-to-M' compiler by rewriting the code-generation section of C.
- Generate an M'-hosted self-resident L compiler via the steps above.

Bootstrapping compiler performance. Given an optimizing self-compiling L compiler, C, and an executable L compiler, E, to create a self-resident optimizied optimizing L compiler:

- Generate non-optimized (i.e., probably slow) self-resident optimizing L compiler, E', by submitting C to E.
- Generate an optimized (i.e., faster than average) self-resident optimized optimizing L compiler by submitting C to E'.

Bootstrapping source-language features. Suppose that language L' is an extension of L. Given a self-compiling L compiler, C, and an exectuable L compiler, E, to create a self-compiling L'-compiler and an executable L'-compiler:

- Obtain a crude L' compiler, CC, by crudely extending C to an L-hosted L' compiler, CC. (Note: use only features of L in writing C.)
- Generate an executable L' compiler, EE, by submitting CC to E.
- Obtain a well-written self-compiling L' compiler, C', by carefully re-coding CC in L'.
- Generate an executable version, E', of C' by submitting C' to EE.

Given C' and E', we are ready to add the next feature/extension to the language.

1.3 The Front End

A compiler can be roughly broken into two portions: the *front end* and the *back end*. The duty of the front end is to produce intermediate code and a symbol table from the source form of the program. The back end takes the intermediate code and attribute table (symbol table with the symbol strings omitted) and produces object code for the target machine.

The front end consists of three segments: the scanner (lexical analyzer), the parser (syntax analyzer), and the translator (semantic actions). The front end is a syntax-directed translator, i.e., the parser is in charge and directs all activities. The parser calls the scanner when it wants another symbol from the input stream. It calls the appropriate semantic action as it uncovers the structure of a portion of a program.

1.3.1 The scanner

To the compiler the source-language form of a program is simply a file (i.e., stream) of characters. The first order of business is to break that stream into chunks called *tokens*. Tokens are typically identifiers, keywords, operators, literals, punctuation, etc., and doing so is the responsibility of a function called the scanner. On each call, the scanner begins where it left off and scans the program file, one character at a time, until it has figured out the next token. It then returns an index telling which kind of token found and sets two global variables, one holding the actual string of characters that constitutes that token and another which holds the "value" of that token (e.g., its numeric value if the token is a literal or the address of its symbol-table record if the token is an identifier).

1.3.2 The parser

The parser directs the activities of the front end of the compiler. Its duty is to construct a derivation of the source program from the grammar of the language. Attached to each grammar rule will be one or more translation routines (semantic actions), which the parser will call at the appropriate times as the derivation is reconstructed. The ultimate purpose of a semantic action is

- to create and manage entries in the symbol table,
- to create translation records for use by subsequent invocations of semantic actions, and
- to generate intermediate code.

The symbol table and intermediate code are then passed to the back end of the compiler for optimization and code generation. Because the parser decides when to invoke the various semantic actions, compilers are called *syntax-directed translators*.

1.3.3 Translation routines and the semantic graph

As the front end parses a program, it (implicitly) traverses the program's *parse tree* (a.k.a., *absolute syntax tree*) — a bottom-up parser, for instance, does a postfix traversal. As it visits each tree node, the parser calls the appropriate translation routine — hence the term *syntax-directed translation*.

These translation routines emit a representation of the *semantic graph* of the program, i.e., the parse tree with all semantically inessential nodes removed (a.k.a., *the abstract syntax tree*) and with nodes decorated with all necessary semantic attributes — for instance, an identifier leaf should contain a pointer to the symbol-table record for that identifier, and nodes corresponding, say, to relational operators should contain an indication of which relational operator is involved.

The representation of the semantic graph is constructed in two data structures: the interior nodes of the tree are represented by instructions of *intermediate code*, while the leaves are usually data objects or code labels, represented by *symbol-table* records, which contain the semantic attributes of the entities identified by the corresponding identifiers.

Each interior node has an opcode and pointers to the node's left and right descendants, respectively. For normal operations, such as addition, these descendants are the operands of that operation. The identifier fields of symbol-table records are irrelevant to the back end of the compiler but will be useful for the debugger.

Note that some edges in the semantic graph represent data connections, e.g., a node corresponding to a division operation has numerator and denominator descendants, each of which may be a variable or an internal node representing the result of a data operation. Other edges represent control structures, e.g., an if-else statement has a Boolean data descendant and two control descendants, one for "then" and the other for "else".

When edges to and from symbol-table entries are taken into consideration, the semantic graph of a program is not generally a tree. The fact that a particular variable may appear at more than one leaf makes the semantic graph a directed acyclic graph (DAG), whose leaves can have multiple predecessors, and during optimization the identification of more complex common subexpressions may introduce interior nodes with multiple predecessors. The data-edge subgraph remains acyclic, but the presence of goto statements allows cycles in the control subgraph.

1.4 Intermediate Code

1.4.1 Formats

Since there are many ways to represent graphs, there are many formats for intermediate code.

In general, the format for the intermediate code is *low-level* and *generic*. It is low-level in the sense that it is essentially machine code for a fictitious machine.

It is generic in the sense that the low-level machine selected has none of the idiosyncracies of real machines, e.g., it has an unlimited number of registers.

In selection of the format for nodes in intermediate code, we face the same possibilities faced in designing the instruction set for a real computer. When performing an operation, we would like to know six things:

1. what operation to perform (opcode),
2. what variable to perform it on (source),
3. possibly a second source variable (source2),
4. where to store the result (destination),
5. where to find the normal next instruction (next),
6. where to find the alternative next instruction (target) for branch instructions.

To save storage, instruction-set designers usually invoke several simplifications. The first is to get rid of the next-instruction field by assuming that the next instruction to be executed is the sequentially-next instruction, unless current instruction is a branch instruction, in which case the outcome of the current instruction determines which instruction gets executed next.

There is a second compression that is often performed: assuming that the result is to be stored where one of the operands came from. Such code is called two-address code, in contrast to the previous code format, which is called three-address code.

Additional simplifications are possible:

- *one-address code*, in which a special register called "the accumulator" is always taken as one of the operands and as the destination for the result;
- *stack code*, where the operands and result come from and go to the top of a stack.

Neither of these formats will be considered here, since they hide information that is helpful in implementing important optimizations.

1.4.2 The quadruple format

To turn the semantic tree (DAG) into low-level code for a generic machine, we perform a couple of transformations on the semantic graph.

First, we expand all control nodes (e.g., if-else nodes) into conditional and (possibly indirect) unconditional branch nodes, just as one does in writing an assembly language program and possibly indirect branches for returns, indirect calls, and switch statements. Then, to facilitate register allocation and other operations, we introduce on each internal edge an additional level of indirection through the symbol table:

- Each branch-instruction node refers to (i.e., points to) a symbol-table record for an instruction-pointer constant (i.e., label) that, in turn, points to the target instruction.

- Every value-producing nonleaf node is assigned a temporary variable (symbol-table entry) for holding values passed along edges to the node's parents. There are edges from the parents and the child to the symbol table record for this temporary, which is called a *source* operand of the parent nodes and the *destination* operand of the child node.

With these modifications, the intermediate program essentially becomes code for a three-address load-store architecture with an unlimited number of registers, one for each variable in the symbol table. This format is sometimes called *quadruples*, because each node consists of an opcode plus up to three operands: one or two sources and a destination (or three sources, or a destination and a place).

1.5 The Back End

The back end of the compiler translates the semantic graph (as represented by intermediate code and the symbol table) into assembly code for a specific machine. It consists of two parts.

The *optimizer* takes as input the symbol table and the intermediate code. It produces equivalent, but more efficient, intermediate code for the program. It does so by a sequence of transformations that are:

- *valid* — do not change the behavior of the program on any execution paths through the program along which behavior is defined,[1]

- *harmless* — do not lengthen the program's running time along any execution path from entrance to exit.

The optimizer information that is useful in the subsequent (code-generation) phase.

The *code generator* takes the output of the optimizer and produces an equivalent program in the assembly (or machine) language of the target machine.

The study of the compiler's back end is almost a general study in the translation of low-level languages.

[1]Lots of situations, e.g., the use of uninitialized variables, lead to undefined behavior, and once undefined behavior occurs, all subsequent behavior is undefined. To permit the generation of better code, for example, the C/C++ standard specifies that behavior is undefined whenever an object is modified more than once between two sequence points.

Chapter 2

SCANNING

2.1 Tokens

To a compiler, a source program is a file of characters[1] that include letters (A–Z, a–z), digits (0–9), punctuation symbols, blanks, tabs, and newline characters (which specify where each line ends). The compiler's first order of business is to group these characters into into short meaningful character strings called "lexemes." When the compiler attempts to find the next lexeme, it reads characters from the file one at a time building the longest lexeme it can understand. White space (blanks, ends of lines, tabs, ends of page, and comments[2]) are usually ignored, except that they are sometimes necessary to separate tokens. Consider the C/C++ compound statement

```
{ return x+=y; }
```

The sequence of lexemes found here is:

```
{
return
x
+=
y
;
}
```

Notice that `+` and `=` are each lexemes by themselves but the longest-match rule dictates that an occurrence of `+=` is a single lexeme.

In C/C++ the kinds of lexemes are:

[1] In most applications these are represented via the ASCII character encoding.

[2] In C++ the comment delimiters are `/*` and `*/` and `//`. An occurrence of `//` begins a comment that goes to the end of the line, while an occurrence of `/*` begins a comment that goes to the next occurrence of `*/` — no nesting.

words: unbroken strings of letters[3] and digits beginning with a letter. Most words are programmer-defined *identifiers*, names given to items in the program. Some words, called *keywords*, have predefined meanings assigned by the language.

- *keywords*: In C++ the keywords include:
  - type specifiers: `char double float int long short signed unsigned void`
  - type keys: `class enum struct union`
  - storage specifiers: `auto register static extern volatile`
  - other declaration specifiers: `const friend typedef inline virtual asm private protected public template virtual`
  - statement formers: `break case continue do else for goto if return switch while` **throw catch try**
  - operators: `sizeof` **delete new**
  - predefined identifiers: `default` **this**
- *identifiers*: words that the programmer uses to name the items of the program: objects, types, values, macros, statement labels, etc. In C++ identifiers consist of letters, digits, and the underscore symbol (_) but must begin with a letter or an underscore, not a digit — `foo7` and `_foo7` are identifiers, but `7foo` is not. Embedded spaces are not allowed, but `good_number` and `goodNumber` are okay. Identifiers are case sensitive — `foo7` and `Foo7` are distinct identifiers referring to different objects. Keywords are not allowed as identifiers.

literals: symbol sequences that name the values of certain data types according predetermined encoding schemes, like the decimal notation system for naming numbers and putting quotes around a string to create a name for it.

operator symbols: names for operations, which take values or objects (called *operands*) and produce another value or object (called the *result*) and possible *side-effect* behavior, like changing the content of an object or sending and receiving messages. These operations are encoded as sequences of machine instructions (code values) that act directly on the objects and the encoded representations of the values. In C++, for example, the operator symbols are:

```
!   %   ^   &   *   -   +   =   |   ~   <   >   ?   ,   .   /
->  ++  --  .*  ->*  <<  >>  <=  >=  ==  !=  &&
||  *=  /=  %=  +=   -=  <<=  >>=  &=  ^=  |=  ::
```

The side effects and result of each operator will be discussed in the section on operators and expressions.

[3] C and C++ allow underscore (_) as a letter, but the programmer should avoid using it as the first character of a word to avoid collision with special names introduced by the linker.

punctuation symbols: In C++ for example, the punctuation symbols are:
`( ) { } [ ] ; " ' : \`

2.2 The scanner

As mentioned above, the scanner reads the source code of the program being compiled from a file of characters and breaks that stream of characters up into symbols (character strings) called *lexemes*. Each time the parser calls the scanner, the scanner reads characters from the input and returns not the next lexeme itself but a *token* (actually an integer) that indicates the kind of lexeme found, e.g., an occurrence of an identifier, or the key word **if**, or a string literal, etc. Each token designates a set of strings, usually regular languages defined by a special regular expression called the token's *pattern*. Each member of that language is said to be an *instance* of that token. For most programming languages, each keyword and each punctuation symbol has its own token, whose pattern defines the language consisting only of that one string. All identifiers, however, have the same token, as do all integer literals, etc.

The scanner also returns a *lexical value* for each lexeme (usually in a global variable, e.g., `yylval` in scanners generated by the scanner generators LEX and Flex). Basically, a lexeme's lexical value tells which instance of that token was scanned. For example, if the lexeme is an identifier, the scanner returns the token for identifiers but sets the lexical-value variable to contain a pointer to the scanned identifier's symbol-table entry. If, however, the scanned lexeme was an integer literal, e.g., "345", the scanner converts that string of three characters to the corresponding integer value and returns that value as the lexical value of this token. The lexical value of a key word is not usually significant, since normally each key word has a unique token.

Note that the lexical values for different tokens may be of different types. Therefore, the global lexical-value variable must be of a union type.

The parser, which invokes the scanner, is only interested in which token (kind of lexeme) was found, but the error-reporting and symbol-table routines need to know the exact lexeme, and the semantic actions need to know the lexical value in order to generate correct code. For instance, as far as the parser is concerned, one integer constant is as good as the next, but the semantic actions need to know whether to generate code to multiply, say, by 5 or by 38 — it makes a difference.

2.3 Scanner construction

A scanner is commonly specified by a sequence of patterns, which are regular expressions, each with an associated program fragment called the pattern's *lexical action*. On each invocation of the scanner (i.e., on each *scan*), the scanner inspects characters from the input file, one at a time, starting with the character after the lexeme encountered on the previous scan and tries simultaneously to match each of those patterns. The scanner continues to inspect characters as long as there remains a potentially matchable pattern. The scanner may have already matched others, but it always goes for the longest possible match. At each character of a scan, a given pattern may:

- Match the string of characters inspected so far on this scan.
- Be eliminated as a possibility for a further matching during this scan, no matter what characters subsequently are encountered.
- Not match, but remain an active possibility for a longer match when more characters have been inspected.

Typically, the scanner ignores comments and treats all consecutive blanks, tabs, and newline characters (i.e., all *white-space* characters) as a single blank.

When all patterns have been eliminated for further matching, the scanner stops inspecting characters and performs the lexical action associated with the last matched pattern. Ties are broken by pattern priority (usually the order of presentation of the patterns in the specification of the scanner), so it is okay if the patterns define overlapping regular languages. A pattern's lexical action must (among other things) assign the proper value to the global lexical-value variable and return the proper token to the parser.

Normally, scanning resumes at the next character after the one where the last match occurred. Some patterns, however, do not consume all of the characters that they match. They have a *lookahead* indicator associated with them, and scanning resumes after the last character that matched the portion of the pattern up to that indicator. The standard example of the need for lookaheads is the FORTRAN line `DO10I=1,75`. (Note that blanks are optional in FORTRAN.) The portion of this line before the comma is a valid assignment statement for the variable `DO10I`. Hence, the scanner does not know whether it is scanning the key word `DO` or the variable `DO10I` until it reaches the comma. The next scan will resume at some character before the "7" in either case.

The way that scanner generators like LEX build a scanner is to form a nondeterministic finite automaton (NFA) from each of the patterns. LEX then adds a start state with epsilon transitions to the start states of each of those pattern-specific NFAs and labels each accepting state of this NFA with the index of the corresponding pattern.

Next LEX determinizes this NFA via the set-of-states construction from the proof of the Rabin-Scott theorem. Each set of states having an accepting state from the NFA is an accepting state of the resulting deterministic finite automaton (DFA), and LEX labels that accepting deterministic state with the same label as its (nondeterministic) accepting member of smallest index.

This DFA is the heart of the LEX-generated scanner. On each scan, it begins in its start state. It inspects characters, one at a time, until it reaches a state having no successor with respect to the current character. It then looks at the index of the most recent accepting state and invokes the corresponding lexical action. If no accepting state has been visited, an error occurs.

The next scan resumes just past the point in the input stream where the lookahead state for the accepted pattern was last encountered — a pattern's lookahead state is the same as its accepting state, unless otherwise specified, typically by a slash at the corresponding point in the pattern.

Each keyword and each punctuation symbol has its own token, which gets returned. The lexical values of such lexemes are irrelevant.

On encountering an identifier, the scanner (specifically, the lexical action corresponding to the pattern for identifiers) looks it up in the symbol-table, installing it if necessary. It returns a token corresponding to the kind of identifier: integer variable, function name, type name, undeclared, etc. The lexical value of an identifier is (a reference to) its symbol table record.

On encountering a literal, the scanner returns a token corresponding to the kind of literal and sets the lexical-value variable to (a reference to) the literal's value.

On encountering an operator, the scanner returns a token corresponding to the operator's category (`MULOP`, `ADDOP`, `RELOP`, etc.) and sets its lexical value to its index within that category.

The parser may set various modes for the scanner. For instance, PL/I allows `THEN` be both a keyword and a variable. Thus, there must be some communication back from the parser to the scanner, in terms of the content of the symbol table and in terms of various global state and mode variables to assist the scanner in determining which token and lexical value to return in such potentially ambiguous situations.

Mention compression of the transition table.

Chapter 3

LR PARSING

The syntactic structure of a programming language is usually specified via a context-free grammar that partially reflects the semantics of the language — the structure of a program's syntax tree reflects (in part) the program's semantics (i.e., meaning) and thus assists the compiler in producing an equivalent program in a lower-level language. But, most programming languages are not context free, e.g., the requirement that variables be declared before use cannot be specified via a context-free grammar. Such context-sensitive requirements are usually enforced via post-parsing semantic analyses. Thus, the language defined by the grammar in, say, the official C standard is context free but includes some strings that are not semantically valid C programs.

A *parser* for a given context-free grammar G is a function that, given a file containing a sequence of characters (i.e., a string), produces:

- a G-derivation (or syntax tree) of that string, if the string is in the laguage of G,

- some appropriate error message(s), otherwise.

A *parser generator* is a program that, given a context-free grammar, produce a parser for that grammar.

There are general parser generators that will produce a parser for each and every context-free grammars. Most are based on back-tracking or dynamic-programming algorithms, but even the best general parser generators produce parsers whose time complexity is slightly better than n^3 in the length of the input string. The parsers used in compilers usually have linear time complexity — so, of course, the known generators for linear-time parsers do not succeed in producing a parser for some context-free grammars. This chapter present one class of parser generators, namely the LR parser gernerators, which succeed on a relatively large fraction of interesting grammars.

3.1 LR parser generators

Given a context-free grammar, an LR parser generator produces a *parse table*, which is used by an LR parsing engine (i.e., an implementation of *the* LR parsing algorithm) to simulate a grammar-specific DPDA that yields the requisite derivations and error messages. The problem of producing an equivalent deterministic pushdown automaton from a context-free grammar is analogous to producing an equivalent deterministic finite-state (DFA) automaton from a set of regular expressions. The latter problem is solved by generating, from the regular expressions, an equivalent nondeterministic finite-state automaton (NFA), and from it generating a DFA via the classical set-of-states construction.

For any given context-free grammar, it is easy to produce an equivalent nondeterministic pushdown automaton (NPDA). But, in the case of pushdown automata, the set-of-states construction doesn't always succeed in producing an equivalent DPDA — it is well known that some context-free languages are not recognized by any DPDA. Specifically, the set-of-states construction fails when, for some input symbol, the NPDA states in a given deterministic state *conflict* (i.e., disagree) about what to do to the stack.

The LR parsing algorithm. A grammar's DPDA (if it has one) is encoded as a transition table, ie., a two-dimensional table of states indexed by states and grammar symbols, both terminal and nonterminal. LR transition tables, however, also have some special entries, called *reduce* entries, which refer to grammar rules rather than states. Each grammar rule has an associated *semantic-action* function, which gets invoked whenever a reduction via that rule occurs.

Given a string (i.e., program) in a grammar's language, an LR parser for that grammar traces, in reverse order, the steps of a right-most derivation of that string.[1] During that trace, it is easy for the semantic actions to build a representation of the derivation's syntax tree in a bottom-up fashion, i.e., leaves first and root last. Each step of the derivation invokes the corresponding rule's semantic action, which constructs and adds a new right-most root to the portion of the tree (forest) that has been constructed thus far, and returns a pointer to it. Each tree node corresponding to a nonterminal can be represented by a record containing semantic information about that node. Each tree node corresponding to a terminal (lexeme) can be represented by that terminal's lexical values (if any), stored in the parent's record.

Entries on the parse stack are pairs whose first components (i.e., syntactic parts) are states and whose second components (i.e., semantic parts) contain semantic information about the root nodes in the portion of the syntax tree (forest) that has been visited thus far — the bottom stack entry contains information about the left-most root, etc. The semantic information for a terminal symbol (lexeme) is its lexical value, while the semantic information for a nonterminal is (a pointer to) its record in our representation of the syntax tree.

[1] A *right-most* derivation is a derivation where each step is obtained via the application of a grammar rule to the right-most nonterminal of the previous step.

```
typedef int State;
State beginning_state = 0;

struct TableEntry {
  Kind  kind;    // which kind of entry
  Rule& rule;    // valid for reduce entries only.
  State state;   // valid for go and shift entries only.
}

union SemanticData {
  // fields the lexical value of each token and for
  // translation records generated by each rule.
}

SemanticData semanticAction( Rule );  // translation record.

TableEntry table[ maxState, maxToken ];

void parse() {
  static stack< pair<State,SemanticData> > parseStack;
  Token token = scan();
  parseStack.push( pair( beginning_state, 0 ) );
  for(;;) {
    TableEntry& current = table[ parseStack.top.first, token ];
    if ( current.isReduceEntry ) {
      if ( current.rule.leftSide == startSymbol ) return;
      for ( int i = 0; i < current.rule.rightSide.length(); ++i ) {
        stack.pop();
      }
      TableEntry s = table[parseStack.top.first, current.rule.leftSide];
      assert( s.kind == go );
      parseStack.push( pair( s.state,  current.rule.semanticAction() );
    } else {
      if ( current.state == errorState ) errorReport();
      stack.push( pair( current.state, lexicalValue ) );
      token = scan();
    }
  }
}
```

When a reduce entry is encountered during parsing, the next interior node has been reached in a virtual depth-first traversal of the program's implicit syntax tree. It is, therefore, time to construct, insert and return (the address of) that node in the actual syntax tree, which is being constructed bottom up. Doing so is the responsibility of the semantic action that is associated with the reduce entry's grammar rule. That node's children will have already had their semantic actions invoked, and their semantic information will be in the semantic parts of popped stack nodes, which are just above the new top of the stack, in left-to-right order. In YACC and Bison, the semantic parts of those stack entries are designated $1, $2, ..., respectively. A semantic action returns a value called $$, which gets paired with the next state and pushed onto the stack.

Under YACC, the semantic actions for building a syntax tree are of the form `{ $$ = ThisRuleCtor( $1, $2, ... ); }`, which constructs from the popped information a new instance of this rule's node class as right-most root in the syntax tree. Its address is assigned to `$$`, which gets paired with the next state and pushed onto the stack — to be passed, when the time comes, to the constructor of this new root's parent.

Example. Let G denote the following grammar:[2]

(0) $S \rightarrow A$
(1) $A \rightarrow BB$
(2) $B \rightarrow D$
(3) $D \rightarrow cD$
(4) $D \rightarrow e$

Note that A generates a pair of B's, each of which generates a D, which in turn generate a string of c's followed by an e. So the language of G consists of all strings of c's and e's having exactly two e's, one of which is the last character. We let $ denote the end of input and have added as rule #0 the rule $S \rightarrow A$ to guarantee that the start symbol (now S) has only one rule (even though we don't need to in this case). The following is a transition table for G:

States	A	B	D	c	e	$
1	g2	g3	g4	s5	s6	
2						r0
3		g7	g4	s5	s6	
4				r2	r2	r2
5			g8	s5	s6	
6				r4	r4	r4
7						r1
8				r3	r3	r3

Blank entries represent error states.
State #1 is the start state.
Reduction by rule #0 constitutes acceptance.

[2] We follow the metanotation conventions of the Dragon Book:

- A, B, ... denote nonterminal symbols.
- a, b, ... denote terminal symbols.
- X1, X2, ... denote symbols of either kind.
- u, v, ... denote strings of symbols of either kind.

The following is a right-most derivation of the string ccee.

S			
A	via	(0)	S → A
BB	via	(1)	A → BB
BD	via	(2)	B → D
Be	via	(4)	D → e
De	via	(2)	B → D
cDe	via	(3)	D → cD
ccDe	via	(3)	D → cD
ccee	via	(4)	D → e

The string ccee$ parses via the above transition table as follows:

<table>
<tr><th>Stack</th><th>Input</th><th colspan="2">Reduction</th><th>Tree</th></tr>
<tr><td>1*</td><td>ccee$</td><td></td><td></td><td>| | | |</td></tr>
<tr><td>1* 5c</td><td>cee$</td><td></td><td></td><td>c | | |</td></tr>
<tr><td>1* 5c 5c</td><td>ee$</td><td></td><td></td><td>| c | |</td></tr>
<tr><td>1* 5c 5c 6e</td><td>e$</td><td>(4)</td><td>D → e</td><td>| | e |</td></tr>
<tr><td>1* 5c 5c 8D</td><td>e$</td><td>(3)</td><td>D → cD</td><td>| | D |</td></tr>
<tr><td>1* 5c 8D</td><td>e$</td><td>(3)</td><td>D → cD</td><td>| D-+ |</td></tr>
<tr><td>1* 4D</td><td>e$</td><td>(2)</td><td>B → D</td><td>D-+ |</td></tr>
<tr><td>1* 3B</td><td>e$</td><td></td><td></td><td>B |</td></tr>
<tr><td>1* 3B 6e</td><td>$</td><td>(4)</td><td>D → e</td><td>| e---+</td></tr>
<tr><td>1* 3B 4D</td><td>$</td><td>(2)</td><td>B → D</td><td>| D</td></tr>
<tr><td>1* 3B 7B</td><td>$</td><td>(1)</td><td>A → BB</td><td>| B</td></tr>
<tr><td>1* 2A</td><td>$</td><td>(0)</td><td>S → A</td><td>A-+</td></tr>
<tr><td>1*</td><td>$</td><td></td><td></td><td>S</td></tr>
</table>

Note that this sequence of reductions is the right-most derivation of ccee in reverse order, which corresponds to a post-order traversal of its syntax tree. The right-hand column shows how to construct this derivation's syntax tree in the process of discovering the derivation in reverse order.

Note. Formally speaking, the transition relation for an NFA consists of a set of triples. Each triple $< p, w, q >$ contains a state p, a string w, and a state q that the automaton goes to when it is in state p and scans the string w. We can denote that triple by the instruction:

p : if read w then goto q ;

In our study of scanning, without loss of generality, we restricted our attention to the case where the length of w is less than two.

Generally, an instruction for a PDA has two more components, namely, what is popped from and what is pushed onto the stack:

p : if read w and pop u then push v and goto q ;

A PDA consists of such instructions, but in our study of parsing we restrict our attention to the case where the length of w is less than two and the length of v is one. It is easy to show that our restrictions lose nothing in generality, but we need to be aware that we've used different terminology. Specifically, our "state" is the stack token on the top of the stack, which, in effect, we pop and then push back on each transition.

3.2 The LR(0) table

The LR(0)-NPDA. We now show how to produce, for any given grammar, an NPDA (called its *LR(0)-NPDA*) that not only accepts but *parses* strings in that grammar's language, i.e., each of the LR(0)-NPDA's possible nondeterministic computations traces bottom-up in depth-first order a syntax tree of an initial segment of the input and all such trees will be represented, in particular, any trees for the entire input.

An *LR(0)-item* for a context-free grammar is a production rule of that grammar with a mark (usually denoted by a dot) somewhere on its right side. Items having their mark at the extreme left are called *initial items*. Those with their mark at the extreme right are called *completed items*. In noncompleted items, the symbol to the right of the mark is called the *featured* symbol. For example:

F → (**.** E)	is a typical item for F with E featured;
F → **.** (E)	is the corresponding initial item for F;
F → (E) **.**	is the corresponding completed item for F.

The grammar's LR(0)-items will be the states of the grammar's LR(0)-NPDA. The *starting state* is the initial item for the start symbol's single grammar rule.[3]

For any symbol X, either terminal or nonterminal, the X-*successor* of an item of the form $A \to \cdots \mathbf{.} X \cdots$ is $A \to \cdots X \mathbf{.} \cdots$. Also, in the case where X is nonterminal, every initial item for X (i.e., initial items having X as their left-side nonterminal) is an ε-*successor* of $A \to \cdots \mathbf{.} X \cdots$. An ε-transition to an initial item for X can be viewed as a nondeterministic procedure call to one of possibly several routines for accepting a string derivable from X. If this routine succeeds, it returns and we go to the X-successor of the calling item.

The classical way to handle procedures calls is to push the successor of the calling instruction onto a stack and, on return, to pop it from the stack and go to it. We could do the same thing here, pushing the successor state whenever the featured symbol is a nonterminal and popping that state upon return. Unfortunately, that stack-handling scheme will become a problem when, later on, we try to determinize the NPDA via the set-of-states construction, because each of the various nondeterministic threads of a computation can be doing something different to the stack at each transition. Instead, we push the new state at each transition and return from completed items by popping multiple states. Specifically, at a completed state for A, say $A \to X1 \cdots$

[3]We guarantee that there is only one such rule by adding the rule *newstart* → *start* and taking *newstart* as our new starting symbol.

Xn., we return from the current *call* to $A \rightarrow .X1 \cdots Xn$ by *reducing* by the rule $A \rightarrow X1 \cdots Xn$:

1. Pop n+1 states from the stack.[4]
2. Go to (i.e., push) the A-successor of the new top, since it was the item that *called* $A \rightarrow .X1 \cdots Xn$ in the first place, and we have successfully parsed a string derived from A via this rule.

Example. The following is the LR(0)-NPDA for our example grammar G:

```
[ S --> .A ] ----A---> [ S --> A. ] --> reduce
    |
   eps
    |
    V
[ A --> .BB ] ---B---> [ A --> B.B ] ---B---> [ A --> BB. ] --> reduce
    |                       |
    |                       |
   eps      ______eps______/
    |      /
    |     |
    V     V
[ B --> .D ] ----D---> [ B --> D. ] --> reduce
    |     |
    |    eps
    |     |
    |     V
    | [ D --> .cD ] --c--> [ D --> c.D ] --D--> [ D --> cD. ] --> reduce
    |     ^                     |
   eps    |                     |
    |      \______eps___________/
    |      /
    |     |
    V     V
[ D --> .e ] ----e---> [ D --> e. ] --> reduce
```

The LR(0)-DPDA transition table. To create a DPDA from LR(0)-NPDA via the set-of-states construction, we expand the stack alphabet so that it corresponds to sets of items rather than single items. Also, when we make an ε-transition, we will simply add the new item to the current top rather than pushing it. The X-successor of a set of states is the ε-closure of the set of X-successors of the items in that set. Using a work-list algorithm, we construct a transition table, called the grammar's LR(0)-table.

[4] When we determinize, the ε-transitions no longer generate transitions, so we pop only n states in a reduction in the deterministic machine.

```
typedef set<Item*> State;
set<State*> states;
map< pair<State*, Symbol*>, State* > table; // curry this.

State* lookUp( State* s ) {
  // Return 0, if s not in states.
  // Otherwise, return pointer to equivalent existing state.
}

void constructTable() {
  queue<State*> workList;
  states.add(new State);  // add empty state to workList.
  // also add initial state to states and to worklist.
  while ( workList.size() ) {
    State* s = workList.pop();
    for ( each X in symbols ) {
      State* newState = successor(X,s);
      State* oldState = lookUp( newState );
      table[s][&X] = oldState ? oldState : newState;
      if ( oldState ) {
        delete newState;
      } else {
        states += newState;
        workList.append( newState );
      }
    }
  }
```

When, during parsing, a deterministic state containing a completed item becomes the current (top) state, we pop the stack once for each symbol of that item's right-hand side. But, nondeterminism arises when such a state is not a singleton:

- If the deterministic state contains two completed items, there may be a difference in the number of symbols to pop from the stack or in which successor of the new top to go to. Such a situation is called a *reduce/reduce conflict*.

- If the deterministic state contains a completed item and a noncompleted one, there will be a question of whether to pop or to keep scanning the input. Such a situation is called a *shift/reduce conflict*.

If the table is conflict-free, it defines a DPDA for parsing the strings in the language of the given grammar, and the grammer is said to be an LR(0)-grammar. Unfortunately, conflicts occur too often with this construction. To diminish the number of conflicts, and hopefully eliminate all of them, we introduce the LR(1)-table.

3.3 The LR(1)-table

The LR(1)-NPDA. An *LR(1)-item* consists of an LR(0)-item together with a terminal symbol, called the item's *lookahead*. We usually represent these items in the form $\mathrm{A} \to \cdots\textbf{.}\mathrm{X}\cdots$ / b where b is the lookahead.[5] The LR(1)-NPDA for a grammar has these LR(1)-items as its states.

The *X-successor* for an LR(1)-item $\mathrm{A} \to \mathrm{u}\textbf{.}\mathrm{X\ v}$ / b is $\mathrm{A} \to \mathrm{u\ X}\textbf{.}\mathrm{v}$ / b. In the case where X is a nonterminal, $\mathrm{A} \to \mathrm{u}\textbf{.}\mathrm{X\ v}$ / b also has as *ε-successors*, all items of the form $\mathrm{X} \to \textbf{.}\cdots$ / c where c is the first symbol of some string of terminals derivable from the string vb, i.e., a member of FIRST(vb), where vb denotes the string v with the symbol b appended to it and FIRST is defined as in the next subsection.

We still view an ε-transition as a call to a subroutine for parsing a string derived from the featured symbol, but in LR(1) case that routine is trying to find such a string followed by a terminal symbol that could possibly follow that string in the current context. The only way that the stack handling by LR(1)-NPDAs differs from that by LR(0)-NPDAs is that, when it gets to a completed item, say $\mathrm{X} \to \cdots\textbf{.}$/ c, and the next input symbol is not a "c", it quits rather than reduce by $\mathrm{X} \to \cdots$ return, since, in such a circumstance, it would have had to quit after returning anyway. The key point is that we anticipate this error don't add to the possibility of a conflict in the set of states construction by continuing down this erroneous nondeterministic track; specifically, we don't unnecessarily pop the stack.

Computing FIRST. Let w be any string of symbols, FIRST(w) denotes the set of all terminal first symbols of strings derivable from w. Visibly, for any one-symbol string consisting of a nonterminal, say "X", any beginning terminal of a right side of an X-rules will be in FIRST of that one-symbol string. So will any member of FIRST of any non-terminal that begins the right side of an X-rule. The same holds for any right-side symbol whose predecessors are non-terminals that derive the empty string, i.e., that are *nullable*.

In determining the nullable symbols, we can omit all rules that have terminals on their right sice. One can use a least-fixed-point algorithm:

```
set<symbol> nullable;
for ( bool stable = false; ! stable; ) { // 'til stable
  stable = true;
  for ( each grammar rule R ) {
    if ( R.lhs not nullable && each symbol in R.rhs nullable ) {
      stable = false;
      nullable += R.lhs;
    }
  }
}
```

[5]The slash here has nothing to do with the slash used in Flex patterns to indicate the start of "following context."

Or, more efficiently, one can associate with each rule a counter initialized to the number of distinct symbols on the rule's right side. Whenever a rule with a counter value of zero is encountered, add its left-side nonterminal to the set of nullables and omit all of that nonterminal's rules. So long as there remain unprocessed nullables, select one and process it by decrementing the counters of all (remaining) rules in which that nonterminal occurs. Visibly, the number of steps is linear in the number of symbol occurrences in the grammar.

To determine FIRST of each symbol, consider a directed graph with an edge from each non-terminal X to symbols on the right-hand sides of X-rules whose right-hand-side predecessors are all nullable. Visibly, FIRST(X) is the set of all terminals reachable from X in this graph. (Appendix B, page 121, discusses algorithms for computing reachability.)

To compute the LR(1) NPDA for G, we first compute nullable, which is visibly empty. Then we compute `FIRSTs` of each nonterminal, and they all turn out to be `{ c, e }`. Then, we obtain the following NPDA:

```
[S-->.A / $] ---A--> [S-->A. / $]
      |
      V
[A-->.BB / $] ---B--> [A-->B.B / $] ---B--> [A-->BB. / $]
      |                      |
      |                      V
   +--+                [B-->.D / $] ---D--> [B-->D. / $]
   |                         |
   |         +---------------+
   |         |
   |         +---> [D-->.e / $] ---e--> [D-->e. / $]
   |         |
   |         V
   |   [D-->.cD / $] ---c--> [D-->c.D / $] ---D--> [D-->cD. / $]
   |
   |
   +-> [B-->.D / c] ---D--> [B-->D. / c]
   |         |
   |         +---> [D-->.e / c] ---e--> [D-->e. / c]
   |         |
   |         V
   |   [D-->.cD / c] ---c--> [D-->c.D / c] ---D--> [D-->cD. / c]
   |
   |
   +-> [B-->.D / e] ---D--> [B-->D. / e]
             |
             +---> [D-->.e / e] ---e--> [D-->e. / e]
             |
             V
       [D-->.cD / e] ---c--> [D-->c.D / e] ---D--> [D-->cD. / e]
```

In practice, we do not construct NPDAs for grammars — rather the NPDAs are a conceptual tool to show how the algorithms work. So, below we construct the LR(1) NPDA for G as an example to see what it looks like. (For each symbol X, all X-transitions are labeled with X. ε-transtions are unlabeled and have a vertical segment.)

Recall that an ε-transition is essentially a call to a subroutine to process an expansion of an item's featured nonterminal, followed by a terminal symbol that is in FIRST of the item's subsequent symbols including the lookahead symbol. Recall also that completed LR(1)-items represent reduce states and that when reducing by an X-rule, the stack is popped once for each symbol on the right side of the previous item plus once for the ε-transition. Then, we go to the X-successor of the new top.

The LR(1) DPDA transition table. When we carry out the set-of-states construction on the *LR(1)-items* of a given grammar, the resulting table is said to be that grammar's *LR(1)-table*. It has fewer conflicts than the LR(0)-table, and now those conflicts have the forms:

1. a *reduce/reduce conflict*, i.e., a state that contains two completed items with the same lookaheads,

   e.g., $\mathrm{A} \to \cdots\textbf{.}/\ \mathrm{b}$ and $\mathrm{B} \to \cdots\textbf{.}/\ \mathrm{b}$

2. a *shift/reduce conflict*, i.e., a state that contains a completed item whose lookahead symbol is the featured symbol of one of this state's noncompleted items,

   e.g., $\mathrm{A} \to \cdots\textbf{.}/\ \mathrm{b}$ and $\mathrm{B} \to \cdots\textbf{.}\mathrm{b}\cdots /\ \mathrm{c}$

In non-conflict states, it is clear what the DPDA should do to the stack, regardless of which symbol is next in the input. Note that, unlike the LR(0) situation, a nonsingleton LR(1) state that contains completed items can be conflict free.

Any grammar having a conflict-free LR(1) table is *unambiguous* (i.e., every member of its language has exactly one syntax tree), but not vice versa. For example the grammar, which is visibly unambiguous but has a reduce/reduce conflict in the a-successor of its start state:

(1) $\mathrm{S} \to \mathrm{Acd}$
(2) $\mathrm{S} \to \mathrm{Bce}$
(3) $\mathrm{A} \to \mathrm{a}$
(4) $\mathrm{B} \to \mathrm{a}$

Prove that LR(1) grammars are unambiguous.

3.4 The LALR table

Unfortunately, a grammar's LR(1) table is usually much larger than its LR(0) table. Because of the impractical size of the LR(1) table, we introduce the *LALR table*, which has the same size as the LR(0) table but only a slightly larger chance of conflicts than the LR(1) table. The *core* of a set of LR(1) items is the set of LR(0) items obtained by ignoring lookaheads. The LALR table is obtained from the LR(1) table by combining states having the same core, i.e., replacing them by their union. This works because, whenever two deterministic LR(1)-states have the same core, so do their X-successors for every symbol X.[6] Hence, the LALR table looks exactly like the LR(0) table for the given grammar, except that there are fewer situations where the stack could be popped for a return. It has only slightly more conflicts than the LR(1) table for the same grammar. In fact, although this collapsing of the LR(1) table to form the LALR table may introduce some reduce/reduce conflicts, any shift/reduce conflicts in the resulting DPDA were already in the LR(1) DPDA. A grammar's LALR table is identical to its LR(0) table, except that some conflicts are resolved.

Define tail(i) to be the rest of the body after the marker.

The easiest way to construct a grammar's LALR table is to first construct its LR(0)-table and then compute the set of lookahead terminals that is associated with each LR(0)-item in each LR(0)-state.

```
map< State, map< Item, Set<Symbol> > lookaheads;

void propagate( State s, Item j, Symbol a ) {
  assert( j is in s );
  if ( ! a in lookaheads[s][j] ) {
    lookaheads[s][j] += a;
    propagate( successor(j.featuredSymbol(),s), successor(j), a );
    if ( nullable( j.tail() ) {
      for ( each epsilonSuccessor k of j ) propagate(s,k,a);
    }
  }
}

void compute_lookaheads() {
  propagate( initial_state, initial_item, '$' );
  for ( each State s ) {
    for ( each Item i in s ) {
      for ( each epsilon-successor j of Item i ) {
        for ( each Symbol a in FIRST(tail(i)) ) propagate(s,j,a);
      }
    }
  }
}

void install_reduce_entries() {
  for ( each State s ) {
    for ( each completed Item i in s ) {
      for ( each Symbol a in lookaheads[s][i] ) {
        switch ( table[s][a].kind ) {
```

[6]The *X-successor* of a set of items is the ε-closure of the X-successors of those items.

```
            shift:   shift_reduce_conflict(s,a);
                     break;
            reduce:  reduce_reduce_conflict(s,a);
                     break;
            default: table[s][a].kind = reduce;
                     table[s][a].rule = i.rule;
                     break;
          }
        }
      }
    }
  }
```

To *install the reduce entries* in the table, go through the states and when you find a completed item $B \rightarrow w./\ a$ in a state s, install a reduce entry for reduction by $B \rightarrow w$ at `table[j][a]`. Of course, if `table[j][a]` already contains a shift entry, s has a shift/reduce conflict. And, if it already contains a reduce entry then s has a reduce/reduce conflict.

To assist the programmer, when an error occurs, the parser can list the acceptable terminals, i.e., all featured terminals plus the lookaheads for completed items. Even more, the parser could list the items of the state that is on the top of stack.

3.4.1 The LALR table for G

Recall that our example grammar has the production rules:

(0) $S \rightarrow A$
(1) $A \rightarrow BB$
(2) $B \rightarrow D$
(3) $D \rightarrow cD$
(4) $D \rightarrow e$

The transition table for G is as follows:

States	A	B	D	c	e	$
1	g2	g3	g4	s5	s6	
2						r0
3		g7	g4	s5	s6	
4				r2	r2	r2
5			g8	s5	s6	
6				r4	r4	r4
7						r1
8				r3	r3	r3

Blank entries represent error states.

The following are the exact states for G:

I[0]: empty

I[1]: S → .A / \$
A → .BB / \$
B → .D / c,e
D → .cD / c,e
D → .e / c,e

I[2]: S → A. / \$

I[3]: A → B.B / \$
B → .D / \$
D → .cD / \$
D → .e / \$

I[4]: B → D. / c,e,\$

I[5]: D → c.D / c,e,\$
D → .e / c,e,\$
D → .cD / c,e,\$

I[6]: D → e. / c,e,\$

I[7]: A → BB. / \$

I[8]: D → cD. / c,e,\$

Exercise. Give the LR(1) transition diagram and LALR table for the following grammar:

(1) S → E
(2) E → E + T
(3) E → T
(4) T → T * F
(5) T → F
(6) F → (E)
(7) F → id

3.5 Parsing via LEX, YACC and C++

The code for a compiler written in C++ using LEX (or its open-source equivalent, Flex) and YACC (or its open-source equivalent, Bison) is organized into several files. Here is a sample Unix-style makefile that uses Flex, Bison, and g++:

```
OBJS      =  yac.o  lex.o  main.o  tree.o
SFLS      =  yac.c  lex.c  main.cc tree.cc

CC        =  g++
CFLAGS    =  -g -I/usr/local/include/sgi-stl -Wall
CXXFLAGS  =  -g -I/usr/local/include/sgi-stl -Wall

compile: $(OBJS)
$(CC) $(CFLAGS) $(OBJS)  -o compile -lfl

lex.c: lex.src
flex lex.src
-cmp -s lex.yy.c lex.c || mv lex.yy.c lex.c

lex.o yac.o main.o tree.o : heading.H
lex.o main.o tree.o : tok.h
yac.o main.o lex.o tree.o : tree.H

yac.c: yac.src
bison -d -v yac.src
-cmp -s yac.tab.src   yac.c || mv yac.tab.src   yac.c
-cmp -s yac.tab.src.h tok.h || mv yac.tab.src.h tok.h
```

The conditional renaming of the output files from `flex` and `bison` avoids unnecessary recompilation. The makefile has been written so that:

- From `lex.src`, `flex` produces `lex.yy.c`, which gets conditionally renamed `lex.c`.
- From `yac.src`, `bison` produces `yac.tab.src`, which gets conditionally renamed `yac.c`.
- From `yac.src`, `bison` also produces `yac.tab.src.h`, which contains a definition of the union data type (`YYSTYPE`) for the semantic portion of parse-stack entries. It also `#define`s each token appearing in a `%token`, `%right`, `%left` or `%nonassoc` line in `yac.src` to a unique number above 256. (Individual characters correspond to numbers below 256 via their ASCII encodings.) To avoid unnecessary recompilation, we compare `yac.tab.src.h` with the existing `tok.h` (if any), and if they differ `yac.src.tab.h` gets renamed `tok.h` and, under that name, gets `#include`d in `lex.c` and `main.cc`.
- From `lex.c`, `yac.c`, `tree.cc` and `main.cc`, `g++` produces the compiler, which is called `compile`.

The semantic portion of parse-stack entries. During compilation, the semantic information about each occurrence of a given grammar symbol gets stored in an object of the symbol's *semantic type*. The first segment of `yac.src`, from "`%union{`" to "`%}`", contains the definition of the union type, `YYSTYPE`, whose members are of the respective semantic types of the symbols of the grammar. At each stage in the bottom-up construction of the parse tree, a number of subtrees will exist. Each stack entry contains the semantic information about the root of one of those trees, with information about the current right-most root on top, etc. For example,

Revise.

- The semantic value for an integer literal gets held in a union member, `int_val`, of type `int`.
- The semantic value for a real (floating-point) literal gets held in a union member, `flt_val`, of type float.
- The semantic value for a token for a singleton lexical category gets held in a union member of type `int` and named `junk` (because it is of no interest).
- The semantic value for a token for a non-singleton lexical category gets held in a union member of type `int` and named `which` (because it is an index telling which token of that category has just been scanned).
- The semantic value for a nonterminal named, say, `Widget` gets held in a union member named `widget` (for lack of any better name) of type `Widget*`, where `Widget` is the base class of the node classes for `Widget`-rules.

The correspondence between `YYSTYPE` members and grammar symbols is established via a series of lines in the declaration section of `yac.src`, which immediately follows the union declaration. A nonterminal, say `Widget` in a line of the form:

```
%type  <member-name>  symbol-name
```

for example

```
%type <Widget> widget
```

Each token appears in a line of the form:

```
%token <member-name> token_1 token_2 ... token_n
```

where member-name might be selected per the recipe above. The token declarations are listed in order of increasing *precedence* (i.e., binding strength), discussed below. Replacing the YACC keyword `%token` with `%left`, `%right`, or `%nonassoc` attaches *associativity* (discussed below) to the listed symbols. All tokens on a given line receive the same precedence and associativity.

Disambiguation. YACC forms a parse table for a DPDA even in the presence of conflicts. By default, YACC reports conflicts and resolves them — reduce/reduce conflicts are resolved in favor of the earlier rule in the input sequence and shift/reduce conflicts are resolved in favor of the shift. However, a conflict where both rules and the input (lookahead) token have precedence (as defined below) is not reported and is resolved in favor of the rule of higher precedence — shift/reduce ties are broken on the basis of associativity of the completed item: "left" implies reduce, "right" implies shift, "nonassoc" implies error. Revise

The precedence and associativity of a grammar rule are those of the last token in the rule's body, unless the rule is followed by `%prec` and an overriding precedence/associativity token. The use of ambiguous grammars and these associativity- and precedence-based resolution rules permits much more compact and comprehensible language representations than disambiguating by adding new terminals and additional production rules to the grammar.

Building the parse tree. During a parse, this compiler produces the detailed syntax tree of an program — subsequent top-down traversals of that tree perform semantic analysis and (intermediate) code generation, respectively, on that program. By convention, a nonterminal symbol named `Widget` will have its semantic values stored in the `widget` component of the semantic portion of its stack entries. These portions will have type pointer-to `Widget` — this time `Widget` is the name of the base class for the node-classes of `Widget`-rules. These per-nonterminal base classes are defined in the upper portion of `tree.H`.

With each grammar rule for expanding Widget (i.e., each `Widget`-rule), we associate a semantic action of the form

```
$$ = new Widget_whatever($1,$2, ..., $n);
```

where:

- `Widget_whatever` is the node-class (unique to that rule). It is derived from `Widget` and has a single constructor, `Widget_whatever(...)`, which has a parameter for each symbol on the right-hand side of that `Widget`-rule.

- `$$` denotes the semantic value for the `Widget` that is about to be added to the tree. `$$` will be the value of the `YYSTYPE` object in the semantic portion of the new stack top.

- `$1, $2, ..., $n` are the `YYSTYPE` values in the semantic portions of the top `n` entries on the stack (with `$n` on top) and contain or point to the semantic information for instances of the grammar symbols in the body of the Widget-rule upon whose reductions this semantic action gets invoked.

  1. If the corresponding grammar symbol is a token, then its stack entry holds the semantic (lexical) value assigned to `yylval` during that token's lexical action, and pushed onto the stack by the parser during that token's shift operation.

2. If the corresponding grammar symbol is a nonterminal, then its stack entry is the value assigned to `$$` during the reduction operation that pushed that entry onto the stack, a pointer to the corresponding root in the forest that has been constructed so far.

Of course, the types of the constructor's parameters must correspond to the semantic types (union members) for the respective symbols. The constructor parameters, `d1`, `d2`, ..., hold the values of the corresponding arguments, `$1`, `$2`, ..., respectively. The constructor must initializes local data members, (often called `c1`, `c2`, ..., respectively) with the values of the corresponding `d`'s. These per-rule derived classes are defined in the lower portion of `tree.H`. Each node in the syntax tree is an instance of one of these derived classes.

Each of these classes has a translation function, `void gen()` — in case of the per-symbol base classes, `virtual void gen()`. The translation for a syntax-tree node (i.e., for each of the per-rule derived classes) must emit code and invoke the translation functions of some of its children. Note that this translation function gets an opportunity to emit code in the before and after each child. The constructor for the start symbol of the grammar must initiate the top-down translation process by invoking its own translator function. Note that parent nodes obtain information from the values of their children's data members (a.k.a., attributes), a technique called *attribute synthesis*. Parents can also set the values of their children's attributes, a technique that is called *attribute inheritance*.

The strategy of simply building the syntax tree bottom-up parse and postponing translation until a subsequent top-down pass gets complicated a bit by the fact that the grammars for many programming languages require that identifiers be properly categorized by the scanner, which requires information that the scanner can only obtain from the identifier's symbol-table record — information that had to be put into the symbol table by the parser during the bottom-up construction of the syntax tree by the constructors of certain semantic classes.

Moreover, symbol-table management must take into account the scoping rules of the language. So, during the bottom-up, tree-building phase, the scope-entry and scope-exit operations of the symbol table must be invoked by appropriate constructors, e.g., the constructor for `ProgramHook()` in the attached code.

Chapter 4

SYMBOL TABLES

A *declaration* is any syntactic construct, i.e., subtree of the syntax tree, whose root note establishes the program-specific meaning of an identifier. The *scope*[1] of a declaration is the portion of the program over which occurrences of that identifier resolve to that meaning, i.e., to that declaration. A *context* is a portion of the program in which declaration's scopes are confined, i.e., out of which some or all don't reach, e.g., a compound statement, the body of a function, the body of a class declaration, a translation unit, etc.

So, we need to look at scopes of identifiers and how one must handle such matters inside a compiler. First of all we must give a *context* to each procedure, function and record. Then we must keep a stack of contexts that directs the compile-time resolution (lookup) of identifiers. On each occurrence of an identifier, we search the contexts from the top of the stack to the bottom for the first context in which the identifier is declared. On leaving a context we pop the top context from the stack. Whenever we enter the declaration of a function/procedure, a declaration of a record, or a WITH-statement, we must push its context onto the context stack. We leave a context (and pop it from the stack) whenever compilation reaches the end of a declaration of a function/procedure, a declaration of a record, or a WITH-statement.

In the symbol table we qualify each identifier with its context. This way when we leave the scope of an identifier, it will automatically become inaccessible. Also, when we need to look up a qualified identifier of the form **widget.field**, we first lookup **widget**. Then, in a context obtained from **widget**'s local-context attribute, we lookup the identifier field. This system is simple, flexible and easy to implement. It takes care of all of the scope problem in languages (like PASCAL) having nested static scopes. It takes care of the problems of qualified identifiers and of the WITH-statement. There is no problem easily handling blocks (i.e., compound statements having internal declarations whose scope is the block) as in Algol and C.

Whenever the compiler enters a new context, it must set aside an *allocation*

[1] This refers to static scopes.

counter for that context. This counter will be used to assign offsets to the various data items that must be allocated within the corresponding run-time segment of memory (i.e., *object*, in the terminology of the C and C++ standards). The way that the allocation counter is managed depends to some extent on the kind of context that we are dealing with. Say that a record is being declared. Then, as each field of the record is declared, its offset attribute is specified to be the current value of the allocation counter, and the allocation counter is incremented by the size of the type of that field. Also the size of this record type is incremented accordingly. If instead a union were being declared, then we would not increment the counter on each declaration of a field, but we would set the size of this type (of union) to the max of its current size and the size of the item being allocated.

Functions and procedures are handled in essentially the same way as records. However, space must be set aside for parameters, the access link, the return link, any registers that need to be saved etc. As blocks are encountered within the compilation of a procedure, they are treated as the fields in a union, i.e., two nonoverlapping blocks will share the same space for their locals.

4.1 Details

There are several language details that impinge on the design of the symbol-table mechanisms:

- Redeclaration: what happens when an identifier is re-declared in a context? Is that an error? Does the more recent declaration take precedence?

- Anticipatory scoping: do scopes ever include portions of the program that precede the declaration? For instance, are forward declarations required for mutually recursive declarations/definitions?

- Categorical scoping: can entities of different sorts (e.g., variables, functions, and types) have the same name and overlapping scopes? How are occurrences of such names within the overlap disambiguated?

To illustrate such complexities, here are the scoping rules for the Tiger laguage, which was the project laguage in the first edition of Modern Compiler Implementation in {C, Java, ML}.[2]

- "There are two different name spaces: one for types and one for functions and variables. A type *a* can be "in scope" at the same time as a variable *a* or a function *a*, but variables and functions of the same name cannot both be in scope simultaneously (one will hide the other). [...] A variable or function declaration may be hidden by the redeclaration of the same name (as a variable or function) in a smaller scope; [...] Functions hide variables of the same name, and vice versa. Similarly, a type declaration may be

[2] Tiger was superceded by MiniJava in the second edition.

hidden by the redeclaration of the same name (as a type) in a smaller scope. However, no two functions in a sequence of mutually recursive functions may have the same name; and no two types in a sequence of mutually recursive types may have the same name."

- "Mutually recursive functions [...] are declared by a sequence of consecutive functions declarations (with no intervening type or variable declarations): [...] Mutually recursive types are declared by a consecutive sequence of type declarations without intervening value or function declarations. Each [type] recursion cycle must pass through a record or array type."

- "In the expression **let** $\cdots$ *tydecs* $\cdots$ **in** *exps* **end** the scope of a type identifier starts at the beginning of the consecutive sequence of type declarations defining it and lasts until the **end**. This includes the headers and bodies of any functions within the scope." Similarly for *fundecs* and functions.

Visibly, Tiger requires complex policies for symbol-table management. Below is a symbol table designed for that purpose:

- To accomodate nested contexts, we use integers to represent contexts and we maintain a context stack.

- To accomodate Tiger's categorical declarations, we use three distinct tables and pair two of them, `varTable` with `funTable`, in an *ad hoc* manner.

- To accomodate redefinitions, we introduce the notion of a *run*, which corresponds to Tiger's *mutually recursive consecutive sequence*. We prohibit in-run redefinitions.

- To accomodate Tiger's anticipatory scoping, which is required for mutual recursion without forward declarations, we make three passes through each run, during the analysis phase. Thus, each declaration is processed in three distinct steps:

  - installation, wherein the declared identifier is installed into the symbol table — these are the only places where data is entered into the symbol table,
  - resolution, wherein other identifiers occurring within the declaration are looked up and local pointer attributes are bound to the addresses of the entities that those identifiers designate,
  - semantic analysis, wherein the declaration's subtree is traversed and type-checked, and its local attributes are bound.

  These steps are operations on the class `Declaration`, not on the symbol table.

```
class TigerSymbolTable {
public:

  typedef int Context;         // represent contexts via ints
  list<Context> contexts;
  void enterContext() {
    static Context c = 0;
    contexts.push_back( c++ );
  }

  void exitContext() {
    contexts.pop_back( c++ );
  }

  SymbolTable() { newContext(); }

  typedef int Run;                // represent runs via ints
  Run thisRun;
  void newrun() {
    static Run c = 0;
    thisRun = c++;
  }

  template< class T >
  class Table {
  public:
    typedef map< pair<string,Context>, pair<T*,Run> > data;
    data table;
    bool install( string& name, T* x, Context context, Run run ){
      data::iterator p = table.find( make_pair(name,context) );
      if ( p != table.end() && p->second.second == run ) {
        return false;                             // failure
      }
      table[ make_pair(name,context) ] = make_pair(x,run);
      return true;                                // success
    }
    T* lookup( string name, list<Context>& contexts ) {
      for (   // from top to bottom, i.e., from back to front
        list<Context>::reverse_iterator c = contexts.rbegin();
        c != contexts.rend();
        c++
      ) {
        data::iterator  p = table.find( make_pair(name,*c) );
        if ( p != table.end() ) return p->second.first;
      }
      return 0;                                   // failure
```

```
    }
  };

  Table< VarDec > varTable;
  Table< FunDec > funTable;
  Table< TyDec  > tyTable;

  bool install( string& name, VarDec* x ) {
    varTable.install( name, x, contexts.back(), thisRun );
    return funTable.install(name,0,contexts.back(),thisRun);
  }
  bool install( string& name, FunDec* x ) {
    funTable.install( name, x, contexts.back(), thisRun );
    return varTable.install(name,0,contexts.back(),thisRun);
  }
  bool install( string& name, TyDec* x ) {
    return tyTable.install(name,x,contexts.back(),thisRun);
  }

  VarDec* varLookup( string name ) {
    return varTable.lookup( name, contexts );
  }
  FunDec* funLookup( string name ) {
    return funTable.lookup( name, contexts );
  }
  TyDec*  tyLookup(  string name ) {
    return tyTable.lookup( name, contexts );
  }

};
```

Chapter 5

TRANSLATION

5.1 Programming-language concepts

Programs normally deal with three kinds of *entities*:

- objects
- values
- operations
- types

Types. A type includes a *range of values* together with various operations on those values. Any implementation of a language that involves a given type must provide an encoding scheme for representing that type's values as bit strings plus instruction sequences that implement the type's operations in terms of those bit strings. Many languages also provide an algorithmic scheme for naming values of certain types. The names assigned by such schemes are called *literal* names (or literal constants).

A run-time occurrence of such a representation for a value is called *a reference to* the corresponding value. Of course, a given bit pattern might represent distinct values with respect to the encodings for distinct types. It is the type of the expression that accesses a bit string determines what value that string refers to at at a given access.

Objects. An *object* is a region (segment) of memory whose *content* (i.e., current bit pattern) is intended to represent (refer to) values of a given type. At a given point in time, a given object of type *integer* might, say, hold a reference to the integer three.[1]

[1] The literature on Algol68 takes the view that the type of an integer variable is reference-to-integer and that an object of the type that C++ calls `int&` is a really reference to a reference

The type of an object determines the length of the object's region and what value its content represents, i.e., what types of expressions are allowed to access it. During execution of a program, the content of an object may change many times, but the type of the object remains stable (in most languages).

An object whose type is, say, Widget is simply a place to store (a reference to) a Widget value and is usually called a "Widget-valued object" or a "Widget object" or simply a "Widget." Be careful, however, Widget values are also called Widgets,[2] and we can confuse the values of a given type with variables of that type. When we say "integer," we typically mean an integer-valued object rather than an integer value. So, how can you tell the difference? If it has an address, it's an object.

At run time an object has:

- state, namely its value,
- identity, namely its address,[3]
- class, namely its type
- behavior, as determined by the operations that its type supports.

Therefore, objects in this sense correspond reasonably well to the concept of object used by the gurus of object-oriented programming.

Depending on its type, an object can contain either code or data values. We *access* a data object by reading or modifying its content (i.e., writing to it). A code object contains machine-language instructions for performing a certain function. In principle, we can read from and write to code objects, but primarily we access a code object by invoking it, i.e., by having the computer perform the object's machine language instructions.[4]

In some languages (e.g., C/C++) code objects are called *functions*. In other languages they are called *procedures*. Still others, such as Pascal, use the term "procedure," only for functions with no return values, i.e., void functions. Function types are categorized according to the data types of the function's parameters and return value.

Constants and variables. A named object is commonly called a *variable*, but some authors use that term for any modifiable (mutable or read/write) object. In either case, an unmodifiable (read-only or immutable) object is called a *constant*.

---

to an integer. C++ uses the term "reference" only for references to an objects, which are a particular kinds of references. C++ takes the view that the objects of a particular type, say `T`, are the values of another type, which C++ designates `T&`.

[2] In some contexts, Widget-valued objects are called "instances of Widget," and creating a Widget in described as "instantiating the type Widget." Also, some authors call such an object a "references to a Widget."

[3] Every object has an *address* (location in memory), which is itself a value of a particular type: the address of a Widget object is a value of type Widget-address.

[4] Synonyms for "invoke" are: activate, call, perform and execute.

Many authors use the term "constant" to mean a particular value of a particular type, or the name of such a value. A language usually associates with certain types a scheme for naming values of that type; such names are called *literal constants* or simply *literals*. If a language makes provision for programs to give programmer-chosen names to values of a given type, those names are said to be a *symbolic* (or *manifest*) constants of that type. For example, `3.14159` is a *literal constant*, while a program might define `pi` to be a *symbolic constant* denoting the same value. There are almost no manifest constants for the built-in types in C/C++; rather, there are read-only variables — the exceptions are tags for enumerated types.

In early days of computers, modifiable code objects were thought to be an advantage, but experience has proved them to be a debugging nightmare. So code objects are constant in most modern languages and architectures. An in-line function, however, is a symbolic name for a code value. When a compiler encounters a call to an in-line function, a parameterized copy of that code value becomes part of the content of the code object being compiled. (Actually, C/C++ allows conforming compilers to decide whether or not to expand in-line functions in line. Also, each in-line function must have a program-wide address.) It should be noted that lambda abstraction is a mechanism for generating literal names for functions.

Independent compilation. To facilitate concurrent work by members of a programming team and to minimize recompilation time, programs are broken up into independently compilable units, called *modules*.

In C/C++ a module `whatever` consists of two files containing (mostly) *declarations* written in C/C++, usually by a programmer using an *editor* program:

- A *header* file, `whatever.h`, containing declarations of the items defined by the module for use by other modules. To prevent multiple-definition errors these declarations must be partial in the case of objects. The source files of every modules that accesses any items exported by `whatever` must *include* `whatever.h` so that `whatever`'s declarations can be used in the compilation of those source files.

- A *source* file, `whatever.cc`, containing full declarations (i.e., definitions) of the module's objects, both exported and non-exported. The source file should include `whatever.h`, both for coding efficiency and as a consistency check.

When the source file `whatever.cc` is submitted to a compiler, the compiler puts equivalent machine-language instructions into an *object* file, `whatever.o`, that tells how to layout and initialize the initial objects of that module. An object file has two tables associate with it:

- a table of content, which tells what symbols are defined in this module and gives their offsets in the code or data region defined by this module,

- an external-reference table, which tells which symbols are used but unresolved in this file.

Declarations. A source file must introduce each item via a *declaration*, which specifies the item's name and some of its attributes, before making any other reference to that item — the compiler quits if it sees an undeclared name. (In C/C++ labels are implicitly declared and, hence, are exempt from this rule.)

A declaration specifies (i.e., *binds*) the *attributes* and name of the item it introduces. The values of these attributes are stored by in a description record, called a *descriptor*, which in turn is stored in the *symbol table*, where compiler can look up subsequent occurrences of that name.

Just as construction scaffolding is discarded before a building is occupied, the symbol table is discarded after compilation, before the program is ever run, thus obliterating all record of program entities, other than the layout and initial values of initial objects. When a program runs, it creates and destroys many other data objects as a result of the code generated with the help of that scaffolding.

Scopes. A declaration occurring inside another declaration (say within the definition of a function or a record) is said to be *local* to that surrounding declaration. Those that are not local, are said to be *global*. A preprocessed C/C++ source file consists of a sequence of global declarations.

The *scope* of a declaration is that portion of the program over which references to the identifier it introduces refer to the entity it declares, i.e., places where a search of the symbol table for this identifier will find the descriptor created by that declaration. In C++ there are four possibilities:

1. The scope of a global declaration is from the point of declaration to the end of the file in which it occurs.
2. The scope of a declaration statement within a compound statement is from the point of declaration to the end of the compound statement.
3. The scope of a statement label is the entire function in which it occurs.
4. The scope of a declaration of a class member is the declaration fo that class plus the body of any function member, friend, or class derived from the given class.

In a nested scope, C++ allows introduction of a new item with a name already declared in an outer scope.

Partial vs. full declaration. Some declarations give additional meaning to an already declared names, binding additional attributes. A declaration that binds all compile-time attributes of an item is called a *definition* — non-defining declarations are called *partial*. In a given scope an item can have multiple partial declarations, but they must be compatible in their attribute bindings — no rebinding at compile time, since translation may have already been done using the original binding. Ultimately, each partial declaration must be resolved by a definition of that item.

Definitions of objects prearrange their run-time memory segments, hence *an object can be defined only once within its scope* — a subsequent definition would create another object with the same name, violating the attribute-compatibility rule, since distinct objects have different values for their address attributes. Hence, the compiler-enforced portion of the single-definition rule for objects is a corollary of the attribute-compatibility rule.

One reason for partial declarations is that a program-wide object must be declared in each module accessing it, but allocated (defined) only in its exporting module. The other reason is to take care of the case where items are each defined in terms of the other (e.g., mutually recursive functions or data types). Each must be at least partially declared before it can occur in the definition of the other. These partial declarations are called "forward declarations."

Lifetime. The lifetime of an object is the run-time interval from the time that memory is allocated for the object to the time that memory is reclaimed (i.e., deallocated).[5] Some objects such as Unix files are *persistent* and have lifetimes that transcend any process (program run) that accesses them. Other objects are *static* and have lifetimes that are exactly that of the process in which they exist. Others are *automatic* and have lifetimes that correspond to the invocation of the function/procedure to which they are local. Yet others are *dynamic* and are programmatically created and destroyed (e.g., via invocations of `new` and `delete` in C++). Note that in C/C++ a function-local static variable has function scope (i.e., it is not accessible outside that function) but static lifetime (i.e., its value is remembered from one invocation of the function to the next).

Linkage. It is the business of the linker to take the object files for the modules of a program and produce a program (executable) file by resolving all unresolved references. This involves *backpatching* appropriate entries in the code. There are two possible errors that can take place: an external reference may appear in no table of content in the set, or it may appear in more than one.

A symbol with *external linkage* (program-wide scope) creates an entry in the unresolved reference table. It also creates an entry in the table of content of the file in which it is defined. Those with *internal linkage* have no entries in these tables and, therefore, are local to the module in which they are declared.

5.2 Describing Entities

5.2.1 Type specifications

In most languages types can be defined heirarchically to an arbitrary depth. Therefore, no simple record can describe a given type, but, rather, a complex

[5] C/C++ uses the term *storage class* instead of "lifetime."

data structure is needed. The most straightforward scheme is to simply use syntax tree for the type specification as the description for that type. (Some of the tedium of indirection in this structure can be circumvented by using references to descendants, rather than pointers.)

In this structure, the leaves are built-in types such as characters, reals, and integers, possibly of various precisions. In addition, user-defined enumerated and subrange types can appear as leaves.

The interior nodes correspond to more complex data types:

- An arrays has two subsidiary types, its index type and its element type. Typically the index type must be from (a subrange of) the integers or an enumerated type. The element type can be any type whose objects are of uniform size.

- A function has a parameter list and a union, structure, class-object, or record has a list of fields (members), each of which has a type of its own.

- A pointer or reference has a referent type, i.e., the type of the thing that it refers to.

Note that multiple instances of the same type might occur in the syntax tree. For instance, the rule `Type->INT` might occur multiple times in the derivation of a program, and each occurrence will produce a node in the program's syntax tree. In fact, each integer-variable declaration will have such a child node.

Obviously, integer variables all need to point/refer to a single common object that represents the integer type. So we must introduce a new notion of type that is distinct from the type nodes in the syntax tree. The same goes for the Boolean type, the Real (i.e., float) type, etc. Let's call the class of such types, `SemanticType`. We can declare global instances of `SemanticType` called `theIntType`, `theBooleanType`, etc. But struct types, classes, and record types are also types. Therefore, the declarations of such entities should be derived from `SemanticType`. Then, two variables will have the same type if and only if their `type` attributes point/refer to the same instance of `SemanticType`.

5.2.2 Constants

The symbol table record for a symbolic constant needs to specify its type and its value as well as its name and context. At run time such a constant is treated pretty much the same as a variable: it is simply a symbolic register whose value is set at the beginning of the program and never changed. Of course, that works only for constants of a simple type whose value can be held in a register.

It should be noted that the value of a constant can get bound very late in the compilation process.

I suggest that one treat literal constants of as manifest constants for purposes of generating intermediate code. The difference is that for simple types they should be looked up by value rather than by name. Thus, there should be a value table for each simple type. In the best case, a manifest constant is registered in both the symbol table and the value table for its type.

Most languages allow literals of string types. It is probably not worthwhile keeping these in a table, since it is unlikely that the same string will be used in many places, and if it is, the programmer can maintain his/her own table.

5.2.3 Variables

The descriptor for a variable must specify the variable's name, scope, type, lifetime and linkage. Also some allocation information, such as whether it is to be kept in memory and, if so, where — *where* is typically specified by a *base* (which is represented as a pointer to the descriptor for a variable) and an *offset* (which is represented as a pointer to a constant, whose value may not yet be known).

5.3 Translating Expressions

Every expression is of a particular type. The referrent of the reference generated by an expression is called the value of that expression. The implementation of the expression's type determines that value.

In the translation of expressions, we generate a new temporary for the result of each operation. Recycling temporaries and minimizing the required number of temporaries is left to the optimizer. As each operation is translated, we push onto the semantic stack a pointer to the descriptor of the temporary that holds the operation's result. This descriptor will have the type of the result as one of its fields.

We view variables, constants, and temporaries of simple types as residing in registers of the generic machine that is the target architecture for intermediate code. The exception is any variable whose address gets taken. Such a variable may be changed via a pointer expression and, hence, must be viewed as residing in memory. After each indirect assignment any register-based copy of its value must be viewed as stale, and prior to each indirect read any dirty register copy must be written back. In fact, this same problem occurs whenever there is a potential for indeterminate aliasing. A reasonable protocol is to set aside a register for holding the value of such a variable and to refresh the value before each use and to write it back immediately after any change. A relatively small number of such variables will be named in any given program.

5.3.1 Type issues

The *type* of an expression is the type of the value it computes. Each operator requires that its operands be of certain types. When an operator allows only a particular type for each particular operand, it is said to be *monomorphic*; otherwise, it is said to be *polymorphic*. One way to achieve polymorphism is to have a separate declaration for the operator for each sequence of operand types. This approach is called *overloading*. The type of the result is given by a table that is keyed on the type of the operands.

Some operators, however, do the generally same operation on a vast range of types, e.g., dereferencing applies to any pointer type. That approach is sometimes called *genericity*. In this case the type of the result is given by a type expression involving the types of the operands.

It is possible that the arguments to a given function are of incompatible types, e.g., one might try to add an integer to a real. Addition can be viewed as a polymorphic operator, but it does not automatically apply to mixed types. We have the alternative, however, of converting the value of the integer subexpression to the corresponding real value, which involves an entirely different bit pattern. Operators for doing so are available in most programming environments. When such a conversion is applied automatically by the compiler, that process is called *type coercion* or *type conversion*.

Given the possibilities of conversion and polymorphism, the type of a given expression is no longer definite but rather it consists of a set of possibilities, which includes all the result type of each polymorphic form of the expression's main operator that can be supported by the possible types of the subexpressions. This set must be expanded to include all possible coercions of its members as well. For each of the possible types for a given expression, there will usually be a particular form of the main operator and type for each of its operands that yields that type for the expression. Occasionally, however, there are multiple ways that a given formula can have a particular type. For instance, `3+4` could be of type `real`[6] by converting each operand to type `real` and then adding the results. Alternatively, one could simply add the integer forms of the operands and coerce the result. Obviously, the latter approach requires one fewer coercions. The result is the same either way, but in the presence of user-defined overloading such commutativity of coercions may not hold.

In practice, these type sets are synthesized bottom up. Then a selection is made top down. That selection can be facilitated by attaching to each type in the set, the last coercion or overloading step by which it was produced.

Lvalues and references

We cannot create compile-time descriptors for each possible array element or record field. Such nameless objects must be viewed as residing in memory and must be accessed via object-denoting expressions, called *lvalues*. The result of (evaluating) an lvalue is a reference to the denoted object. Examples of lvalues are:

- simple variables, e.g., if X is a T-valued variable, the result of evaluating the lvalue "X" is a reference to that variable's T-valued object (region of memory).[7]

- subscripted array names, e.g., `a[i+3]`.

[6] In C/C++ `real` is called `float`.

[7] In certain contexts, through lvalue-to-rvalue conversion, "X" represents value of type T that is referred to by the content of that object.

- record names qualified with the appropriate selector name for a data field (member)
- a dereferenced pointer (i.e., address) expression.

The type of an object determines the types of the lvalues that are are allowed to access that object. On any given access to an object, the type of the lvalue determines the interpretation of content of (i.e., the bit pattern stored in) the object's region of memory.

The referrent of the reference stored in an object is called the *value* (or the *rvalue*) of that object. Certain operations/functions require an lvalue for certain operands/arguments. For example, the assignment operator `=` requires that its left operand be an lvalue, which is the origin of the term "lvalue" as shorthand for left-hand-side value in contrast to an "rvalue" which is the sort of expression that can appear on the right-hand-side of an assignment. Note that if an lvalue occurs where an rvalue is expected, the lvalue is immediately converted to the current value of the lvalue-designated object via an implicit transformation called *lvalue-to-rvalue conversion*.

By definition, *a pointer to* a given object is a reference to *the address of* that object, i.e., an instance of the representation for the address of that object. Now *a reference to* a given object is an instance of the representation for that object, i.e., an instance of the bit string that identifies that object, i.e., an instance of the representation for that object's identity/address. So, both are instances of the same bit pattern, but the pointer refers to the object's address while the object reference refers to the object itself — a somewhat tedious distinction — but, in general, it is the type of the expression that accesses an instance of a bit pattern that determines what that reference refers to.

A reference to an object can be viewed as a pointer that is automatically dereferenced wherever it occurs after its declaration — view it as a pointer that always has an invisible "`*`" in front of it.[8] The object-at operator `*` converts an instance of the address of an object of type `T` into a reference to that object.[9]

[8]The creators of C++ threw in a few complexities such as the fact that `sizeof(T&)` is defined to be `sizeof(T)`.

[9]More formally the address-of operator `&` induces a referrent-preserving one-to-one mapping from `T&` to `T` minus 0. Inversely, the object-at operator (a.k.a. the indirection or dereferencing operator) converts an instance of the representation for an object's address to a reference to that object.

Chapter 6

Run-time environment

In writing a compiler, it is important to distinguish between what happens at compile time and what happens at run time. The only thing that ties the two together is the object code, which is generated at compile time and executed at run time. In designing a compiler, one must ask ask for a given language construct:

- What should happen at run time?
- To cause that to happen at run time, what code must be emitted at compile time?
- To cause that code to be emitted, what data needs to be kept in the symbol table and in the translation descriptors that are pushed onto the semantic portion of the parse stack?

For every concept, data structure, and activity, ask yourself whether it exists/happens at run time or at compile time.

In a Unix process there are three main areas of run-time data storage:

- the static area, which holds global variables and other objects, such as string literals, whose life-time lasts until the end of the program,
- the dynamic area, which is managed by malloc and free and which holds objects whose lifetime is explicitly managed by the program,
- the automatic area, which contains the *control* stack onto which the activation records corresponding to invocations of functions are pushed.

There is a stack pointer SP, which points to the top of the control stack. Usually there is also a frame pointer FP, which points to the base of the top activation record. An activation record for a function contains space for this activation's copy of the function's local variables, including the return link (i.e., the location of the next instruction after the call).

Note that these areas correspond to the lifetime of an object, which is not necessarily correlated with the object's scope. For instance, in C the scope of a static local variable is confined to the block in which the variable is declared. It's lifetime is from the first activation of that block to the end of the program, however.

Discuss the dynamic link and the code for procedure call and return.

6.1 Computing alignments, sizes and offsets

Alignments. On many architectures there is a method of placing objects of basic data types in memory that is required for their (efficient) access. For instance, it might be best to align a four-byte quantity such as an integer on a four-byte boundary, i.e., at an address that is a multiple of four. In such a case, the alignment for integers is said to be four. Typically, the alignment for characters is one, for integers is four, for reals is four, for doubles is eight, etc. To adjust a given size or offset x to a particular alignment y, simply increment x by $y - x\%y$ if $x\%y$ is non-zero.

The alignment for complex datatypes is as follows:

- Arrays: the alignment of its entry type.
- Records (structs or class objects): the maximum of the alignment of its field (data member) types.
- Unions: same as records

It is tempting to say that the alignment requirement for a record is that of its first member. In fact, that works for any single-instance record type. But, all instance of a given record type have the same member offsets and the same alignment requirement. Since alignments are powers of two, it is necessary that multi-instance record types observe the most restrictive alignment for any of its fields.

Size and offset. The size of basic data items is specified by the target architecture. The size of a union is the maximum of the sizes of its members. The size of an array is the number of entries times the size of each entry padded to a multiple of the array's alignment factor.

The *size* of a record is the offset of its last field plus the size of its last field. The *offset* of a field is the size of the record consisting of its predecessors padded to a multiple of the field's alignment factor. (Note the mutual recursion in the definitions of size and offset for records.) The first field of a record has an offset of zero and the size of a record with no fields is zero.[1]

At compile time, with each context, we keep a counter that tells the length of the record so far. This holds for records in general, including the activation record of a procedure. We increment that counter each time a locally allocated object is declared. To handle the case of variables declared within nested blocks

[1] But, for technical reasons, the C/C++ standards say that it is one.

(i.e., compound statements), we save the value of the of the allocation counter when we compile the beginning of a block and restore it when we reach the end of the block. We handle dynamically sized arrays by allocating a pointer to the array as a local variable in the block and allocating the array on another stack. At run time, we keep a local pointer to the top of that second stack and, on block entry, initialize it from the current value of pointer for the surrounding block. Nothing need be done on block exit, which facilitates the case where the exit occurs as a result of a nonlocal goto.

6.2 Lexical nesting

In Algol and most of its derivatives, one can nest procedures, i.e., declare one procedure inside of another. In such cases, the outer procedure is referred to as the lexical parent of the inner one. The code in the body of a given procedure has access to the items declared in itself and its lexical ancestors. For instance, it can call any procedure that is a *direct* descendant of itself or of any of its lexical ancestors. Therefore, it can be called by its parent and by any descendant of its parent.

We rank declarations by lexical level. Global variables and procedures are of lexical level zero. Local variables and procedures are of one greater level than that of the procedure in which they are declared.

During the execution of an activation of a procedure, there needs to be a (run-time) mechanism for accessing (the correct instances of) variables that are declared in the procedure's lexical ancestors. This matter is complicated by the fact that in the presence of recursion there may be multiple activation records for a lexical ancestor on the stack at any given time and the mechanism needs to provide access to the correct copies of the ancestor's locals.

Note that a procedure can call procedures that are at most one level deeper than itself; specifically, it can call its children. Therefore, when a procedure is called, an activation of its parent are, by convention, the most recently activated procedure of that level. The corresponding copies of the parent's locals will be the correct ones for the procedure to access during this activation. There are three mechanism that are generally used for accessing variables declared within lexical ancestors: static chains, global displays and local displays.

Static chain. In the first method, with each activation of a procedure there can be a special local pointer called the "static link," which points to the most recent activation of the procedure's parent. (Because a procedure can be called by any descendant of its parent, there is no reason to believe that the static would be the same as the dynamic link, which simply points to the activation record of the caller.) These static links form a chain to access a level n variable from a level m procedure we must go $m - n$ links down this static chain. The value of the static link for a given activation of a procedure is passed to the activation by the caller, which is either the most recent activation of the parent or one of the parent's direct descendants. In any case, the most recent activation of the parent is available to the caller.

Global display. Another method is to maintain a global array of pointers into the control stack This array is called a *display*. The n^{th} element of the array points to the most recent activation of a n procedure. To access a level n variable, one adds its offset to the value of the n^{th} element of the display. On entry to a level n procedure, one saves the n^{th} element of the display in a special local and then sets that element to point to the current activation record (the current value of FP). On return, the former value is return to this entry of the display.

Local display. A local display is simply a local copy (i.e. stored in the activation record) of an initial segment of a global display. On entry to a level n procedure, one copies the first $n-1$ levels of the caller's display and then adds an n^{th} entry that points to the new activation record. The display is used in the same way as a global display. During a return from a procedure invocation, the local display automatically disappears automatically, along with all other locals.

Discuss passing a procedure as a parameter.

6.3 Volatility

An object is addressable if its address is taken or recorded. Any member of an addressable object is addressable. An activation record is addressable and hence, so is every local variable. Any malloc'd object is addressable. Small non-addressable objects are allocated to registers. The rest are "memory allocated." To operate on a memory-allocated object, it must be copied to a register and before almost any subsequent load of that object its value must be restored. Also, after almost any store it must be reloaded before the next operation. Basically, this is a cache-consistency issue. Registers can be viewed as a working cache for memory-based objects, and memory can be viewed as a backing store for registers, which isn't needed when there are sufficiently many registers.

As far as fields of records are concerned, we are looking at available-expression issues intermixed with aliasing issues. any assignment to a possible alias for `*p` kills both `r = *0` and `*p = r`, as does any assignment to `p` or `r`.

Revise this section.

6.4 Parameter passing

Callers must tell their callees which registers are live at which calls. A callee's replacement policy should favor registers that are live at none of its invocations. The callees can then tell their callers the callee always/sometimes replaces. The caller should backup the registers that the callee usually replaces. The callee should backup those that its seldom replaces. For those that are in between, it doesn't make much difference which one backs them up.

Chapter 7

FLOW ANALYSIS

Distinguish between defining instructions and branch instructions where appropriate.

Each instruction has a list of control successors. Most have a single control successor, namely the next instruction in the intermediate code list, but unconditional branches have as control successors only their target instruction, while conditional branches will have both their target and their code-list successor as control successors. Halt instructions have no successor.

Indirect-jump and return instructions can have any number of successors, but the front end generates code in which they have constrained sources, so we accurately can determine at compile time their possible successors. (Multiple passes through the code can collect this information, and determine the possible values of such variables.) Not only do return instructions have constrained successors, but their behavior is even more constrained, since return instructions always take control back where it came from. For now, however, we will analyze each function independently.

By drawing a directed edge from each instruction to each of its control-successors, we obtain a directed graph, whose nodes are the instructions of the intermediate code. An *entry* node is one that has in-degree zero, and an *exit* node is one that has out-degree zero. (We will assume that all such entry instructions are no-ops and all exit instructions are function returns.) It is common, but not necessary, to assume that there is only one entry. We assume that each instruction lies on at least one path from an entry to an exit.

In fact, we expand each node into two "points," called the node's `entry` and its `exit`, respectively. Incoming edges go to the node's entry and outgoing edges go from the node's exit. (Technically, this makes the control graph a bi-partite graph with instructions being special entry-to-exit edges, which establish a one-to-one correspondence between entry and exit points. We will not, however, view instructions as edges.)

7.1 Flow graphs

An edge whose tail has out-degree one and whose head has in-degree one is said to be *sequential.* A *block* is a path segment whose edges are sequential. In a block all nodes but the first have a single predecessor, and all but the last have a single successor.

Add pre-entry and post-exit nodes/blocks.

A minimal block consists of a single node. Maximal blocks are called *basic blocks* — every node in the graph is an element of exactly one basic block.[1] A node is the *leader* (i.e., first element) of a basic block if and only if it is an entry node (i.e., has in-degree zero) or it has a non-sequential incoming edge. Similarly, a *trailer* of a basic block will either be an exit node (i.e., have out-degree zero) or will have a non-sequential outgoing edge. A basic block includes its leader, the leader's successor, that successor's successor, and so on, up to and including the first trailer encountered.

A block is usually represented by a descriptor that includes:

- a pointer to its first node
- the number of nodes in the basic block
- a list of its successors,
- a list of its predecessors.

If we partition the graph into blocks and collapse each block into a single "special" edge from the entry point of its leader to the exit point of its trailer, we obtain a structure called a flow graph. (We identify each block with the corresponding special edge.)

Note that the overall flow graphs must have special entry nodes and exit nodes having no predecessors and successors, respectively. We are interested only in flow graphs where every node is on a path from an entry to an exit.

Possibly define critical edge here and do the appropriate transformation.

Notes. Unless otherwise specified, when we refer to *the flow graph* of a program, we mean the flow graph whose blocks are basic (i.e., maximal). Also, when we refer to *the control graph* of a program, we mean the flow graph whose blocks are singletons (i.e., minimal). We generally identify each instruction with its corresponding singleton block.

Definitions and uses. An instruction form is said to *use* its source variables and to *define* its destination variable(s). Each instruction has a reference to an instruction form and is said to be a *use* of that form's sources and a *definition*

[1] The first node of a basic block need not have multiple predecessors; its predecessor might have multiple successors. Similarly, the last node of a basic block need not have multiple successors; its successor might have multiple predecessors.

of its destination(s).[2] The form itself is said to be a formal definition/use of its destination/sources.

Instructions for which a source is also a destination present tedious boundary cases and exceptions to our general treatment. But, we can get rid of such instructions, say `X = X+Y`, by assigning the value of the expression to a temporary that is distinct from all other operands, e.g., `{ T = X+Y; X = T; }`.

Variables whose values are subject to external manipulation are called *input ports*. Variables whose accesses are externally observable are called *output ports*. IO ports and instructions that involve them are said to be *volatile*.

Load and store instructions introduce *possible definitions* and *possible uses* of the memory locations they might refer to. A store instruction `*x = y` has `x` and `y` as sources and anything that `x` can point to as of this point in the program as a "possible" destination. Likewise, the indexed store instruction `p[d] = y` has `p`, `d`, and `y` as sources and anything that `p+d` can point to as a "possible" destination. Also, a load instruction `x = *y` has `x` as its destination and `y` as a source and has anything that `y` can point to as of this point in the program as a "possible" source. An indexed load instruction `x = p[d]` has `x` as its destination, `p` and `d` as sources, and anything that `p+d` can point to as a possible source. For now, we take all pointers to be constant. Later on we will deal with pointer variables.

We often view a basic block as a jumbo-instruction that defines all variables that its members define and that uses all variables that have an in-block use with no prior in-block definition (of that variable).

7.2 Flow-analysis model

One node set S is said to *trump* another node set T on an incoming path to a point if and only if there is a member of S union T on that path and the last one is a member of S.[3]

We say that S *dominates* a point against T if and only if S trumps T on every incoming path to that point. We say that S *reaches* a point against T if and only if S trumps T on some incoming path to that point.

Duality. We must work with two kinds of duality:

- The *logical dual* of a concept is obtained by swapping the quantifiers "all" and "some" in its definition. The logical dual of "dominate" will be called

[2] ADDR instructions use not their source but rather the base and offset of their source. Also a definition of the base of a memory-mapped variable is a definition of that variable — this works hereditarily in that, for example, a definition of the base of the base of a variable also defines that variable.

[3] S trumps T on a given path evenif the last member of S union T on that path is a member of both S and T. On the other hand, if there are no members of S union T on the path, then S doesn't trump T.

"reach" — a game is said to *reach* a given point if and only if the game reaches that point via *some* incoming paths.[4]

Note quite. By this definition, every game dominates the pre-entry blocks. Dominates is a "one and all" notion.

- The *directional dual* of a concept is that concept applied to the dual graph, i.e., the graph obtained by reversing all edges. We will follow the common mathematical practice of designating the directional dual of any concept by prefixing it with "co-".[5] For example, a game is said to *co-reach* a point if and only if the game reaches that point in the dual graph, i.e., in the original graph there is an outgoing path from that point along which there is an attacker before there is a defender. Similarly, a game is said to *co-dominate* a point if and only if, in the dual graph, the game dominates that point.[6]

7.2.1 Flow-graph concepts

We now define a number of important notions. Note that we identify instructions with their corresponding singleton blocks in the control graph.

- A given block is said to *dominate* a point in a flow graph if and only if every path to that point includes that block, i.e., the singleton consisting of the given block dominates that point against the empty set). Block B1 is said to dominate block B2 if and only if B1 dominates B2's exit.[7]

- A given definition of a variable is said to *reach* any point to which there is a path segment from that definition along which that variable is not again defined, i.e., any point that the singleton set consisting of the given definition reaches against other (occurrences of) definitions of that variable. A given point is said to reach a given use of a variable if and only if there is a path to that use along which the variable is not defined, i.e., that use co-reaches the given point.

Let "busy" become "very busy" (i.e., "fully busy") or "co-available". Also, "fresh" becomes "co-dead".

- A non-volatile defining form is said to be *available* at points where all incoming paths include an occurrence of that form with no subsequent definitions of any of its operands,[8] i.e., at the dominatees of occurrences of that form against the definitions of the form's operands. Dually, a non-volatile defining form is said to be *busy* wherever it is co-available, i.e, at the co-dominatees of its occurrences against the definitions of its operands,

[4] In practice, the word "partially" before any concept defined in terms of all paths refers to the logical dual of that concept, so "partially dominates" means "reaches." Similarly, the word "fully" or "very" before any concept defined in terms of some paths refers to the logically dual concept, so "fully reaches" means "dominates."

[5] Some authors instead use the prefix "post-".

[6] The notion of *co-dominatee* is defined the same way as "dominatee" but relative to the dual graph.

[7] Every block dominates its own exit but not its entrance. So, to make the dominance relationship among blocks reflexive, we specify "exit" rather than "entrance."

[8] After an occurrence of `X = Y+Z` the condition `X == Y+Z` holds until the next definition of `X`, `Y`, or `Z`.

which are the points where all outgoing paths have an occurrence of that form with no prior definitions of any of the form's operands.[9]

- A variable is said to be *fresh* at points where along each incoming path, after each use of that variable there is a subsequent definition of it, i.e., at the dominatees of its definitions plus the entry instructions against its uses. Dually, a variable is said to be *dead* wherever it is co-fresh, i.e., at the co-dominatees of its definitions plus the exit instructions against its uses, i.e., wherever it has no reachable uses, i.e., at points where along every outgoing path, prior to each use of that variable there is a definition of it.[10]
- A defining form is said to be *dead* (respectively, *fresh*) wherever its destination is dead (respectively, fresh).
- A defining instruction is said to be *available* (respectively, *fresh*) if and only if its form is available (respectively, fresh) at its entrance.
- A defining instruction is said to be *dead* (respectively, *busy*) if and only if its form is dead (respectively, busy) at its exit.
- A variable, a defining instruction, or a defining form is said to be *live* wherever it is not dead.

All dead or available instructions are validly and harmlessly removable from the program.[11] *Also, an instruction is validly insertable wherever its form is dead or available.*[12]

Refer to dead and available instructions as "useless" and "redundant," respectively.

7.3 The algorithm

For each form F, there is there is a pair of node sets such that the points dominated by the first of those sets against the second are the points where the form is available. Let's call that pair F's *game* and call the nodes in the first of those sets the *attackers* and those in the second set the *defenders*. We need an algorithm that determines, for each point in the flow graph, the set of all forms whose game's attackers dominate that point against that game's defenders.

More generally, any ordered pair of node sets will be called a *game*. The first set will be called the *attackers* and the second of those sets will be called the *defenders*. The game is said to *dominate* a point if and only if the game's attackers dominate that point against the defenders. We need an algorithm

[9] Some authors use the term *anticipated* rather than *busy*.

[10] Variables of type Input or Output are never dead.

[11] When applied to instructions, the terms "dead" and "useless" mean the same; so do "available" and "redundant."

[12] Some instructions have side effects that set flags, cause exception traps, and/or perform I/O. One must treat such flags and I/O as additional destinations for such instructions, and traps as the result of conditional branch instructions that test (possibly hidden) flags.

that, given an indexed family of games, determines for each point in the flow graph the set of all indices whose games dominate that point.

It turns out that we can (and will) restrict our attention to points that are entrances of basic blocks. For each basic block B in the graph, let `kill[B]` denote the set of all indices of games having a defender in B, and let `gen[B]` denote the set of all indices of games having an attacker in B that reaches the exit of B (i.e., an attacker for which there is no subsequent defender within B).[13] Let `attacks` and `defends` be boolean functions of two arguments such that for every index i and node q,

- `attacks(i,q)` if and only if q is an attacker in i's game and
- `defends(i,q)` if and only if q is a defender in i's game.

Then, at any time during execution of the following algorithm, `close[B]` denotes the set of indices whose games (so far) have been determined to reach the entrance of basic block B.[14]

To save both effort and space, instead of analyzing the dual graph to determine, say live variables, we modify (actually, parameterize) our analysis algorithm and apply it to the original graph. In the dual analysis, we initialize the `kill` and `gen` sets via a front-to-back traversal of each basic block rather than back-to-front. And, in the main loop, we pass information to the predecessors of each basic block instead of the successors. At termination of the algorithm, close[B] contains indices of the games that co-reach the exit of B.

```
template<class Index>
map< Block, set<Index> >
analysis(
  bool attacks(const Index&, const Node&),
  bool defends(const Index&, const Node&),
  set<Index>& I,                              // simply indices
  list<Block>& basicBlocks,    // all basic blocks of flow graph
  enum{all,some} quantifier,
  enum{fwd,bkwd} direction
) {
  map< Block, set<Index> > close, kill, gen;
  for ( each B in basicBlocks ) {
    close[B]  =  quantifier == some                    ? empty :
                 direction  == fwd  && ! B.in_degree() ? empty :
```

[13] Each game on a flow graph induces a corresponding game on the flow graph of basic blocks. An index i is in kill[B] (respectively, gen[B]) if and only if B is a defender (respectively, attacker) in the game on basic blocks that is induced by i's game.

[14] As a practical matter, we often represent subsets of a given collection by bit vectors (one-dimensional boolean arrays) indexed by the members of that collection. We need a standard enumeration of the collection. Lacking any other mechanism, we can simply generate an arbitrary list that contains the members of the collection without repetitions. We then represent subsets of that collections by bit vectors of the same length as that list. The i-th member of the list is in the set represented by a given vector if and only if the vector's i-th bit is `true`.

```
                direction  == bkwd && ! B.in_degree() ? empty :
                I ;
    gen[B] = kill[B] = empty;
    for ( each Node q in ( direction == fwd ? B.reverse() : B ) ) {
      for ( each i in I ) {
        if ( attacks(i,q) && i !in kill[B] ) gen[B].add(i);
        if ( defends(i,q) ) kill[B].add(i);
      }
    }
  }
  queue<Block> workList;
  for ( each Block B ) worklist.push( B );
  while ( ! workList.empty() ) {
    Block B = workList.front();  workList.pop();
    for ( each BB in ( direction == fwd ? succ[B] : pred[B] ) ){
      if ( quantifier == some ) {
        close[BB]  +=  gen[B] + ( close[B] - kill[B] );
      } else {
        close[BB]  &=  gen[B] + ( close[B] - kill[B] );
      }
      if ( BB changed && BB !on workList ) workList.push(BB);
    }
  }
  return close;
}
```

Note that we can speed up the algorithm by taking advantage of the fact that when `quantifier` is `some`, adding gen[B] to close[BB] is significant only the first time we work on B. Likewise, when `quantifier` is `all`, after the first pass we will have removed from close[BB] all indices that are in kill[B] but not gen[B], so we can ignore kill[B] from then on.

Note that the version in section 8.4 of Muchnick's book only does a set comparison for each node, not for each edge. In effect, however, it computes `temp` for each edge.

Claim: When `direction` is `fwd`, for every basic block B, the final value of close[B], as computed above, is exactly the set R_B of all indices whose games reach the entrance of B. (Correctness for the case where `direction` is `bkwd`, will follow by directional duality. The cases where `quantifier` is `all` follow by de Morgan's laws, i.e., logical duality.)

Proof: The algorithm's correctness is based on two facts:

1. The games that reach the entrance of a given basic block are those that reach the exit of at least one of its predecessors.

2. The game for index i reaches the exit of a basic block B if and only if i is in gen[B] + (R_B − kill[B]).

The first of these is obvious, and it is easy to show the second via induction on block length, more specifically, induction on the number of iterations through the inner initialization loop, which builds gen[B] and kill[B].

To show that after each iteration of the main working loop, $R_B \supseteq$ close[B] for every basic block B, we assume, by way of induction, that the hypothesis has held after each iteration so far. Before the current iteration,

$R_{BB} \supseteq$ close[BB], by initialization, if this is the first iteration, and by hypothesis, otherwise.

and

$R_{BB} \supseteq$ gen[B], by facts #1 and #2 above,

and

$R_{BB} \supseteq R_B$ – kill[B], by facts #1 and #2 above,
$\supseteq$ close[B] – kill[B], by hypothesis.

So

$R_{BB} \supseteq$ close[BB] + gen[B] + (close[B] – kill[B]),
which is the new value of close[BB] after the current iteration.

Since the close sets of all other basic blocks retain their former value, the hypothesis continues to hold for all basic blocks.

Conversely, toward a contradiction, consider the shortest path via which some block is reached by the game for an index i not in that block's close set. Then i's game reaches the exit of that block's predecessor P on that path. So, by fact #2 above, either:

Huh?

- i is in gen[P],
- or i is not in kill[P] and i's game reaches the entrance of P (i.e., is in R_P), in which case i is already in close[P] since the path segment to P is one step shorter than the shortest path via which some block is reached by the game of an index not in that block's close set.

In either case, i is in gen[P] + (close[P] – kill[P]). So, by the following lemma, P is on the work list, which is, therefore, not empty. So, when the algorithm terminates, no such path exists, i.e., $R_B \subseteq$ close[B]. Q.E.D.

Lemma: Before and after each iteration of the loop, the work list includes every basic block P such that gen[P] + (close[P] – kill[P]) is not a subset of the close set of one of P's direct successors BB.

Proof: Before the first iteration, all basic blocks are on the work list so the hypothesis automatically holds. Assume that the hypothesis has continued to hold after each main-loop iteration so far. Then either:

- `gen[P] + ( close[P] - kill[P] )` is a subset of `close[BB]`, in which case nothing changes, or

- `gen[P] + ( close[P] - kill[P] )` is not a subset of `close[BB]`, in which case the only change is that `close[BB]` increases and `BB` gets put on the work list.

In either case, the hypothesis continues to hold after the current iteration is completed. Q.E.D.

Note. Since every basic block starts out on the work list and returns to the work list when and only when it receives additional indices, the number of iterations of the main loop clearly is bounded by the number of basic blocks times one plus the number of indices. +1?

Claim: Assuming that the worklist is FIFO, then the number of iterations of the main loop is bounded by the number of basic blocks times the length of the longest acyclic path in the flow graph of basic blocks.

Proof: Let i be any index. Pick any maximal acyclic path segment consisting of basic blocks B such that i is in each R_B but not yet in close[B]. By maximality, i is in gen[P] + (close[P] − kill[P]) for some predecessor P of the first such B, which we'll call B1. By the above lemma, P is on the work list and, since the work list is a FIFO queue, P will be worked on within at most n iterations, where n designates the number of basic blocks in the flow graph. After that, i will be in close[B1] and B1 will be on the work list.

Clearly, the algorithm must terminate when there are no more such paths, and every n iterations the length of the each such path decreases by one. So, the number of iterations is bounded by n times the length of the longest such path. Q.E.D.

7.3.1 Logical duality

De Morgan's law allows us to determine which games dominate the entrance of which basic blocks by simply complementing the set operations in the reachability analysis. Then, we initialize `close[B]` to `I` rather than to the empty set, unless B is an entrance block — we assume that entrance points have a single incoming path on which there can be no attackers. And, when we work on a block `B`, we *remove* from each successor's `close` set all indices not in `gen[B] + ( close[B] - kill[B]) )`.

Note. For node sets S and T and for any incoming path to a given point, exactly one of the following holds:

1. The path contains no members of `S` union `T`.
2. The last member of `S` union `T` on that path is in `S` but not in `T`.
3. The last member of `S` union `T` on that path is in `T` but not in `S`.
4. The last member of `S` union `T` on that path is in both `S` and `T`.

So, `S` reaches that point against `T` via that path in exactly cases (2) and (4), and `T` reaches that point against `S` in exactly cases (3) and (4).

Note. A node that is both an attacker and a defender behaves as an attacker, and entry nodes can be regarded as defenders. So, if E denotes the set of all entry nodes, the game S against (T+E)–S reaches the same points via the same paths as the game S against T.

Claim: S against T dominates a given point if and only if (T+E)–S against S doesn't reach that point, where E denotes the set of all entry nodes.

Proof: S against T dominates the given point if and only if it reaches that point via each incoming path. Similarly S against (T+E)–S fails to reach the given point if and only if, via each incoming path, that game fails to reach the point. But, for any path to the given point, the following are logically equivalent:

1. S against T reaches that point via that path.
2. S against (T+E)–S reaches that point via that path.
3. The last member of S union (T+E)–S on that path is a member of S.
4. The last member of S union (T+E)–S on that path is not a member of (T+E)–S.
5. (T+E)–S against S fails to reach that point via that path.

The equivalences of (1) and (2), of (2) and (3), and of (4) and (5) follow from the definition of "reach." The equivalence of (3) and (4) follows from the first note of this subsection and the fact that S and (T+E)–S are disjoint and include all entry nodes. Q.E.D.

Note. If `B` is not an entry block, `analysis(a,d,I,blocks,all,fwd)[B]` is the complement of `analysis(dd,a,I,blocks,some,fwd)[B]`, where `dd` is defined as follows:

```
bool dd( Index i, Node q ) {
  if ( a(i,q) ) return false;
  if ( q is an entry node ) return true;
  return d(i,q);
}
```

To prove this, note that `dd` has been constructed so that `dd(i,q)` if and only if `q` is in (T+E)-S, where E represents the set of entry nodes and S and T are the sets of attackers and defenders in the game associated with index `i`.

7.3.2 Index sets

We identify games with their indices, which can be objects of any type. In the case of available-forms analysis, the index set will be the set of all forms occurring in the program. For live-variable analysis, it will be the set of all variables occurring in the program. We can sometimes reduce the running time of the algorithm by reducing the size of the index set. The following example illustrates this point.

A given use of X within a basic block B is reachable from the entry of B if and only if the intervening path has no definitions of X. If there is any such use, then the pair <X,B> is said to be a *(global) use* of X involving B. Similarly, only the final definition of X within B can reach the exit of B. If there is any definition of X within B, then the pair <X,B> is said to be a *(global) definition* of X involving B. Note that a definition that reaches the entrance of B reaches B's exit if and only if B contains no definitions of X. Similarly, a use of X that is reachable from the exit of B is reachable from B's entrance if and only if B contains no definition of X. We can use bit vectors to represent sets of global definitions, i.e., pairs <X,B> such that B is a block that contains a definition of X.

Mention Muchnick's initialization heuristic.

Chapter 8

CODE GENERATION

Mention ud and du chains.

There are three main problems in the generation of code for a program:

- register allocation,[1]
- code ordering within each block, which can affect both the amount of traffic between registers and memory and the opportunities for parallel execution of operations,
- instruction selection, which is architecture dependent.

8.1 Optimal ordering of a block

DAGs. In producing intermediate code from an abstract syntax tree for an expression, we produce blocks from DAGs whose leaves are decorated with[2] variables and whose non-leaf nodes were decorated with operations that produce values. The DAGs are such that, if two nodes have side effects that define the same variable or one of them defines a variable and the other uses it, then one of those nodes must be a (possibly indirect) descendant of the other, so that those activities occur in a well-defined order. To produce an equivalent block from a DAG:

1. Attach to each node a reference to its destination variable if any (e.g., if it is an assignment operator).
2. Attach to each of the rest of the nodes a reference to a new arbitrary temporary destination variable of the appropriate type.
3. Turn each non-leaf node into an instruction:

[1] Most operations go faster if (or possibly require that) their operands are in registers, rather than needing to be fetched from memory. So it is important that instructions find their operands in registers as often as possible.

[2] In the context of nodes in trees and DAGs, "decorated with whatevers" means "having a whatever as a data member."

- whose opcode is the node's operation,
- whose destination is the node's destination variable, and
- whose sources are the destination variables of the node's respective children.

4. Form a block by arranging those instructions in any arbitrary topological (children-precede-their-parents) order.

Each topological ordering of the DAG yields a distinct basic block that corresponds to this DAG.

Conversely, to obtain the corresponding DAG from a given block, we draw an edge from each instruction to the nearest preceding in-block definition (if any) of each of its operands — otherwise, to the symbol-table record for the corresponding variable.[3] We then turn each instruction into a non-leaf node decorated with the instruction's destination variable and operation. It turns out that a reordering of a block is valid if and only if the new block has the same DAG as the original, i.e., each of these blocks corresponds to a different choice of topological ordering of their common DAG. So, for example, *we validly can transpose two adjacent defining instructions provided that neither defines an operand of the other.*

An edge from an instruction/node to prior definitions of its destination sometimes imposes an unnecessary restriction on possible orderings — consider, for instance, {`int x; x=1; a=x*x; x=2; b=x*x;`}. One can, however, validly relabel operands within a block so that each variable has at most one definition in the block and the final definition of each defined variable remains unchanged:

```
for ( each instruction in B from first to last ) {
  for ( each source operand X of that instruction ) {
    if ( occurrences of X are being replaced by another variable ) {
      replace this occurrence of X by that other variable;
    }
  }
  if ( this instruction is not the last def'n of its dest'n ) {
    start replacing occurrences of that variable by a new one;
  }
}
```

Such a block is said to be in *static single-assignment (SSA)* form.

Local optimization. Given a basic block, to produce an equivalent block having no superfluous copy instructions and no recomputation of common subexpressions, make a front-to-back pass through the block keeping track of the most recent definition of each variable.

[3]For now, we treat all array elements and all variables whose addresses get taken as (possible) sources of any load instructions and as (possible) destinations for store instructions.

- Propagate copy instructions by replacing source variables whose most recent definition is a copy instruction with the source of that copy instruction, which (by induction) will have been most recently defined by a non-copy instruction.

- Any copy instruction that is not the final definition of a variable that is live at the block's exit will become dead and should therefore be marked for elimination.

- If the current instruction is a defining instruction other than a copy or load or store, say Z = X − Y, normalize it, i.e., replace it by a pair of instructions, $\mathrm{T}_{X-Y} = \mathrm{X} - \mathrm{Y}$ and $\mathrm{Z} = \mathrm{T}_{X-Y}$, where T_{X-Y} is a special variable that occurs as the destination only of instructions of the form $\mathrm{T}_{X-Y} = \mathrm{X} - \mathrm{Y}$.

- If the current instruction is available, eliminate it.[4]

Give example.

Register pressure. The number of variables that are live at a given point in a block is called the *register width of the block at that point*.[5] Notice that a given variable, say X, is live at a given point in a block whenever there is an edge in the block's DAG from a node after that point to an X-labeled node before that point.[6] The register width turns out to be the number of distinct labels on such edges, i.e., the number of distinct head nodes of those edges.

The *register width of a block* is defined to be its register width at its register-widest point. To execute the block using fewer registers clearly requires the introduction of *spill code*, which stores certain intermediate results to memory and reloads them when needed.

The *register width of a DAG* is defined to be the minimum register width of any of its equivalent blocks, i.e., the number of registers required for spill-free evaluation of this DAG.

The problem of determining the register width of a general DAG is NP-complete. For a tree-structured DAG,[7] however, evaluating the wider subtree first and saving its value in a register while evaluating the narrower subtree, requires a number of registers that is the width of the wider subtree if they have different widths, and otherwise one plus the common width of the DAG's two subtrees. That width is "best possible" for a tree-structured DAG, since after evaluating the first value of one subtree, that subtree's intermediate results will occupy at least one register throughout the rest of the DAG's evaluation and will be disjoint from the intermediate results of the other subtree.

[4] Store instructions block the availability of all previous load instructions.

[5] Here we are considering the block in isolation. It may be that other variables, not defined or used in the block are live at its exit, but we don't count them here.

[6] The number of such edges is called the DAG's *cut width* at that point. The register width of a DAG is the cardinality of the upper boundary of the corresponding cut.

[7] Most everything we say about trees, generalizes easily to forests.

The width of a left or right subtree that consists of only a leaf depends on whether the architecture allows `source` and/or `source2` to come from memory. Suppose, for example, we are given a two-address architecture where the destination operand must be a register and the other operand can be in either a register or a memory location, i.e., an architecture whose instructions are of the form `source op= source2`, where source must be in a register. Then, for each node n, its left child n_L must be computed into a register, while its right child n_R can be either in a register or in memory. For such an architecture, the following recursive formula computes register width $rw(n)$ for each node n:

$$rw(n) = \begin{cases} 0 & \text{if } n \text{ is a right leaf (fetched from memory)} \\ 1 & \text{if } n \text{ is a left leaf (loaded into a register)} \\ max\{rw(n_L), rw(n_R)\} & \text{if } rw(n_L) \neq rw(n_R) \text{ (widest child first)} \\ 1 + rw(n_L) & \text{otherwise (first child occupies a register).} \end{cases}$$

8.2 Optimal code for trees

It is not difficult to tell whether a DAG is a tree — simply count each node's parents. One can use dynamic programming[8] to efficiently generate optimum code for a tree-structure DAG where the cost of the code for a tree is the sum of the costs for the code to compute each of its direct subtrees plus the cost of the additional code to compute the root's value from its descendants.[9] We are particularly interested in *cost* functions, like execution time or code size, where cost is a non-increasing function of the number of registers allowed and also depends on whether the result is left in memory or in a register.

Assume that we are given a reference to a node n in a tree-structured DAG and a pool of available locations (registers and memory) one of which is designated as the place that the computed value of n should be stored. Our algorithm will generated optimal code that

- computes the value of n
- leaves the value of n in that designated place
- uses only locations in the pool.

To accomplish the above, we must determine an optimal target-code segment S for computing the value of n from the values of some of its (possibly indirect) descendants — these are the *source nodes* for S. We must also determine an optimal ordering $n(1), n(2), \cdots, n(k)$ of those source nodes. For each $n(i)$, we arbitrarily select from the pool a locations (register or memory, whichever S specifies as the kind of location in which it expects to find $n(i)$) to hold the value of $n(i)$. We will configure S accordingly.[10]

[8]Dynamic programming refers to the tactic of caching recursively computed values to avoid recomputation on subsequent recursion involving the same arguments.

[9]Note that in the presence of overlapped execution, running time is no longer such a cost function.

[10]Think of S as a macro, and the specific locations it uses are parameters.

Then, for each source node $n(i)$, we (recursively) generate an optimal code sequence S_i that deposits the value of that source into its designated location — we remove that location from the pool. Finally, we emit the configured code for S, restore the removed locations to the pool, and return.

Let $C(n, reg, p)$ denote the cost of an optimal code sequence that leave the value of n in a register and require no more than p registers. Let $C(n, mem, p)$ donote the same for code sequences that leave the result in a memory location. We assume that cost is independent of which register or memory location the result occupies. Impossible combinations have infinite cost.

The search for an optimal S is confined to a given table of target-code segments, each typically one or two instructions. Along with each segment, we store its cost and its intermediate-language equivalent, expressed as a *tree template*.[11] To determine whether a given target-code segment is a candidate for computing the value of a given node, we check whether the segment's tree template matches an upper segment of the subtree rooted at that node. To facilitate such matching, we attach to each node in the tree, bottom up, the family of all subtemplates that have the same root operation as that node and whose subtrees match that node's respective children.[12]

A minimum-cost target-code sequence for computing n via S will have the form $S_1S_2 \cdots S_kS$, where $n(1), n(2), \cdots, n(k)$ is a list of the (possibly indirect) descendants of n from which S computes the value of n, and each S_j is optimal target-code for computing $n(j)$ to the location in which S expects to find it.[13] The configured form of S specifies the exact register or memory location in which each of those k descendants of n must reside. Note that the ordering of $n(1), n(2), \cdots, n(k)$ corresponds to the optimal order of computation for these nodes and not to their left-to-right occurrence in the tree-structured DAG.

The total cost of the target-code sequence $S_1S_2 \cdots S_kS$ is the cost of S plus the sum of those costs for each individual S_j. Of course, each S_j gets to use all of the p allotted registers not occupied by results of its predecessors, i.e., by $n(1), n(2), \cdots, n(j-1)$. We may assume that those sources that reside in memory are computed first using all p of the allotted registers — their order makes no difference. Say that there are q such values. Then $n(1), n(2), \cdots, n(q)$ reside in memory, and for $j = 1, \cdots, q$, the cost of S_j is $C(n(j), mem, p)$. Then, we compute $n(q+1)$ into a register via S_{q+1} at a cost of $C(n(q+1), reg, p)$, since all p registers are still available. Next, we compute $n(q+2)$ via S_{q+2} at a cost of $C(n(q+2), reg, p-1)$, since one register is occupied by the previous result, namely $n(q+1)$. Next, we compute $n(q+3)$ via S_{q+3} at a cost of $C(n(q+3), reg, p-2)$, since two registers are occupied by previous results. So, for $j = q+1, \cdots, k$, the cost of S_j is $C(n(j), reg, p-(j-(q+1)))$.

The cost of computing n via S using only p registers is obtained by finding

[11] To avoid the possibility of a failure to match, it suffices that the set of templates include all trees of height one, which happens automatically if the table includes all single-instruction target-code segments.

[12] This match check can be speeded up by compressing these sets of subtrees in a manner similar to the set-of-states construction used to construct a DFA from an NFA.

[13] $S_1S_2 \cdots S_kS$ denotes the concatenation of these $k+1$ code sequences.

the ordering of the sources of S that gives the minimum such cost. The value of $C(n, reg, p)$ is obtained by finding the S that gives the minimum cost over all code segments whose template fits an initial segment of the tree that is rooted at n and who leave their result in a register. Similarly for $C(n, reg, p)$. *At each call to C, we keep track not only of the minimum cost but also which target-code segment and which ordering of its sources gave us that minimum, so that we can generate optimal target-code during a subsequent top-down code-generation pass.*

Give the exact code.

Minimizing spill code for a tree

We now apply the above dynamic-programming scheme to minimize the amount of spill code, i.e., we take the cost of a segment of code to be its number of store instructions. We assume that each node has at most two descendants, n_R and n_L, and that the cost to compute n into memory using p registers is one plus the cost to compute it into a register, i.e., $C(n, mem, p) = 1 + C(n, reg, p)$.

In generating optimal code to compute n into a register using p registers, one must choose which child to compute first and whether to compute it to memory or to a register:

- If one of the children can be computed for free (i.e., without spill code) using $p-1$ registers, first compute its sibling to a register using p registers, then compute the free child using the remaining $p-1$ registers, and finally compute n to the register holding either child. The lowest attainable cost $C(n, reg, p)$ in this case is $C(n_R, reg, p) + C(n_L, reg, p)$, which is the $max\{C(n_R, reg, p), C(n_L, reg, p)\}$ since one of these costs is zero.

- Otherwise, $C(n, reg, p)$ will be $1 + C(n_R, reg, p) + C(n_L, reg, p)$, since we must do one of the following:

  - spill whichever child is computed first, e.g., compute n_R to memory using all p registers, then compute n_L to a register using all p registers, and then compute n at a total cost of $C(n_R, mem, p) + C(n_L, reg, p)$, which is $1 + C(n_R, reg, p) + C(n_L, reg, p)$.

  - squeeze the computation of whichever child is computed second into $p-1$ registers, which costs more than using p registers, since you can always cut cost by using another register and doing one of the spills to that register instead of memory.

  Obviously, the first option is optimal.

Therefore, to generate optimal code in this context, we don't need to compute C — we need only determine which child, if either, can be computed for free. To do so, it suffices to know the register width for each subtree, which we can compute and store with each subtree's root, bottom-up.

Example. Just to make things interesting, suppose we are given a two-address architecture where the destination operand must be a register and the other operand can be in either a register or a memory location. That is, instructions are of the form: *source op= source2*, and n_L must be computed into a register, while n_R can be either in a register or in memory. We have already given the formula for register widths for such an architecture, which as a dynamic program becomes:

```
int rw( const Node& n ) {
  static map< Node*, int > cache;
  if ( cache[&n] exists ) {
    return cache[&n];                          // dynamic programming
  } else {
    return cache[&n] =
      n is a rt child and leaf  ?  0 :        // fetch from memory
      n is a leaf               ?  1 :       // load to a register
      rw(n.left) != rw(n.right) ?  max(rw(n.left), rw(n.right) ) :
      /*otherwise*/                1 + rw(n.left) ;
  }
}
```

The Gencode algorithm takes as its argument a node n in a tree-structured DAG for a block. Its available-location pool consists of a global stack of, say p, free registers (*reg-stack*) plus a stack of temporary locations in main memory (*temp-stack*).[14] It generates optimal two-address code that uses only those free registers and leaves the value of n in the top register of reg-stack and leaves the available location pool exactly as it was received: Why?

```
void
gencode( Node& n ) {
  int p = reg_stack.size();
  if ( n.is_leaf() ) {             // Then n is a left descendant.
    cout << reg_stack.top() << "=" << n.name );
  } else if ( n.right.is_leaf() ) {   // Take care of right leafs.
    gencode(n.left);
    cout << reg_stack.top() << n.op << "=" << n.right.name;
  } else if ( rw(n.right) < p ) {
    gencode(n.left);              // Computes n.left into top reg.
    string R = reg_stack.pop();            // Remove top register.
    gencode(n.right);               // Compute n.right without it.
    cout << R << n.op << "=" << reg_stack.top(); // Now it gets n.
    reg_stack.push(R);                // Now push it back onto top.
  } else if ( rw(n.left) < p ) {
    reg_stack.swap();            // Pre-swap the top two registers.
    gencode(n.right);          // Computes n.right into old second.
```

[14] Keep in mind that these stacks exist at compile time and hold the names of registers and the addresses of memory locations that will be used at run time.

```
    string R = reg_stack.pop();               // Remove old second.
    gencode(n.left);                // Computes n.left into old top.
    cout << reg_stack.top() << n.op << "=" << R;    // And then n.
    reg_stack.push(R);             // Slip old second into position,
    reg_stack.swap();                // by undoing the initial swap.
  } else {                // rw(n.left) >= p and rw(n.right) >= p.
    gencode(n.right);    // Something must spill, since the first
    string T = temp_stack.pop(); // child computed occupies a reg.
    cout << T << "=" << reg_stack.top();     // So, spill n.right.
    gencode(n.left);                 // Emit code to compute n.left.
    cout << reg_stack.top() << n.op << "=" << T;
                                             // Pull n.right from mem.
    temp_stack.push(T);                       // Return T to the pool.
  }
}
```

Exercise. Show that for nontrees that compute pure expressions, the Gencode algorithm generates code that is correct but possibly suboptimal in that it sometimes generates code that needlessly recomputes common subexpressions.

8.3 Code generation for non-tree DAGs

Although the problem of determining the register width of a general DAG is NP-complete, there is a simple greedy heuristic for generating a reasonably narrow[15] re-ordering of an SSA block. First create the block's DAG. Then generate the the reordered block instruction by instruciton, from last to first. At each step, prepend to the block generate so far, an instruction whose parents have already been selected. Among those choose an instruction that minimize the number of live variables[16] the least. Among those, choose an instruction with the maximum number of distinct descendants, i.e., the greatest number of distinct sources. Among those, choose randomly.

Unfortunately, this heuristic may produce suboptimal orderings for tree-structured blocks.

Discuss the maximal-munch strategy, which is based on the same template matching process as the dynamic-programming algorithm.

Alternatively, one can decompose the DAG into disjoint trees — each multiparent node is the root of a tree consisting of its single-parent children, single-parent grandchildren, etc. Form a DAG whose nodes are these trees and whose edges connect each tree to those trees whose roots it contains. Then order the evaluation of these trees, per the above heuristic. Use dynamic programming to generate optimal code for each tree. Finally, allocate registers via the following algorithm, adapted to the target instruction set.

So, why should anyone think that this second heuristic is any better than the first?

[15] I.e., one whose register width is small.

[16] A variable is live at the current entry in the new block if and only if one of that variable's uses has been enumerated but the corresponding definition has not.

8.3.1 Almost optimal (LNU) register allocation for blocks

Given a block of code having a fixed order, the allocation of registers to variable and generation of spill code can be viewed as a problem in the management of a two-level storage system (i.e., cache management), where the registers are the primary (faster) level and main memory is secondary. The problem is analogous to that of virtual memory — in both cases there is a small, fast, expensive memory where most operations take place. There are questions of what item to replace when the primary level is full and another item needs to be cached, questions of whether the replaced item needs to be written back and where to write it, and questions of consistency among various copies of an item.

There is, however, an interesting difference between cache management and register allocation. Cache-management systems make their *replacement* decisions (which item to kick out when another must be cached) on the basis of *dynamic* (i.e., run-time) behavior, e.g., the best item to replace may be the one that is the longest to next use, but there is no general way statically to tell which one that will be. A cache-management system might pick the item that is longest since last use on the assumption, called *temporal locality*, that the future will be like the past — but normal compilers don't have access to such run-time information. A compiler can, however, inspect the *static* (i.e., compile-time) program. Specifically, it has the ability to look ahead to see where data is going to be used. On the basis of data-flow analysis, the compiler can make replacement decisions, e.g., it can sometimes tell that, at a given point in the program, a certain item has no reachable use or that the item has not been modified since it was cached — in either case the item can safely be overwritten without storing its current value.

Dynamic replacement algorithms are by their nature on-line algorithms, i.e., they make their decisions on the basis of incomplete informaiton, the information that has occurred so far. By contrast, static replacement decisions (i.e., those that occur before the program runs) are off-line, i.e., they have all the information they are ever going to have before the individual decisions are made. The procedure we present replaces the variables with registers and inserts spill code per the "longest-to-next-use" replacement policy, also known as LNU and/or Belady's algorithm. LNU is an optimal off-line policy for cache replacement when replacement cost is uniform over all cached items. Unfortunately, in case at hand, the cost depends on whether the item is dirty or not.

The database. We will generate code for a three-address load/store architecture on a block-by-block basis. We assume that, at the beginning of each block, the current value for each variable is stored in that variable's home location in main memory. Therefore, the generated code must ensure that such is the case at the end of the block, as well.

At a given point in a block, the database for code generation consists of the following items:

1. A set, `dirty_variables`, that holds (references to) all variables whose

home location might contain an obsolete run-time value at the given point in the code.

2. A two-dimensional boolean array indexed by variables and by registers. We let `placement[X]` denote the row associated with the variable X — it is a set of registers. Similarly, `holdings[L]` designates the column associated with the register L — it is a set of variables.

3. With each variable's descriptor (i.e., symbol-table record), a `nextUse` attribute that how soon that variable will be used *after the given point in the code*:

   - the distance from that variable's soonest in-block use to the end of the block, if there is such a use,
   - minus one, if the variable is currently dead,
   - zero, otherwise.

   The larger the value of a variable's `nextUse` attribute, the sooner that variable will be used.

Here, we can assume that copy instructions have been propagated.

4. With each instruction, for each of its operands, a similar indication of how soon will be that variable's next use after the instruction's exit. After generating code for an intermediate instruction, we copy the `nextUse` of each of its operands, which are stored as part of the instruction, to the `nextUse` attribute of the corresponding variable's symbol-table record. If and when a variable becomes dead, we remove it from `dirty_variables`.

```
void adjustNextUse( Operand operand ) {
  operand.variable.nextUse = operand.nextUse;
  if ( operand.nextUse < 0 )  dirty_variables -= operand.var;
}
```

To initialize this next-use information for a given block, we do a live-variable analysis.[17] and then invoke the following function.

```
void
setNextUses( Block& B ) {
  for ( each variable occurring in B ) {
    its nextUse = it's dead at the end of B ? -1 : 0;
  }
  int distanceToEnd = 0;
```

[17] We will later show how to do a precise live-variable analysis. For now, it is safe to make errors on the side of assuming liveness and forego the improvement possible with more precise information. For instance, we can assume that all user-defined variables are live, and all temporaries are dead at the end of each block — the front-end is often structured to emit code in such a way that no temporary is reused after the end of its block. Of course, we could crudely assume that all variables are live.

```
    for ( each in-block instruction, last to first ) {
      ++distanceToEnd;
      for ( each operand this instruction defines ) {
        copy its nextUse from its variable's nextUse;
        set its variable's nextUse to -1;
      }
      for ( each operand that this instruction uses ) {
        copy its nextUse from its variable's nextUse;
        set its variable's nextUse to distanceToEnd;
      }
      // We must update the defined variables first, if
      // any variable might be both used and defined by
      // the same instruction.
    }
  }
```

Allocating registers and generating spill code. We generate code for a basic block instruction-by-instruction, from front to back. At the end, we make sure that, if the code were executed, there would be a valid copy of each variable at its home location, i.e., we `writeBack` the values of all `dirty_variables` to their home locations, since that is where successor blocks will expect them.[18] Of course, dead variables do not have to be written back, and that is why we give them the most distant possible next-use distance, thereby, making them the most suitable candidates for replacement.

```
void
generate( Block& B ) {
   dirty_variables = empty;
   for ( each Register  L ) holdings[L] = empty;
   setNextUses(B);
   for ( each Instruction i in B from first to last ) generate(i);
   for ( each Variable X in dirty_variables ) writeBack(X);
}

void
writeBack( Variable& X ) {
   assert( X is in dirty_variables );
   dirty_variables -= X;
   Register& L = any( placement[X] );
   cout << X << "=" << L << ";\n";
}

Priority
soonestUse( Register& L ) {
```

[18] Later, we will discuss global register allocation, in which, variables are held in registers across block boundaries.

```
   Priority soonest = -1;
   for ( each Variable X in holdings[L] ) {
      soonest = max( soonest, X.nextUse );
   }
   return soonest;
}

Register&
mostReplaceableRegister() {
   Register* leastSoon = any( registers );
   for ( each Register L in registers ) {
      if ( soonestUse(*leastSoon) > soonestUse(L) ) leastSoon = & L;
   }
   for ( each Variable X in holdings[*leastSoon] ) {
      if ( X in dirty_variables )  writeBack(X);
   }
   holdings[L] = empty;
   return *leastSoon;
}

Register&
fetch( Variable& X ) {
   if ( placement[X] is nonempty ) return any( placement[X] );
   // otherwise
   Register& L = mostReplaceableRegister();
   cout << L << "=" << X << ";\n";
   holdings[L] = singleton(X);
   placement[X] = singleton(L);
   return L;
}

void
generate( Instruction& i ) {
   Register* L = 0;   // points to new location of destination.
   if ( i has the format:  destination = source  ) {
      L = & fetch( source );
      adjustNextUse( source );
   } else if ( i has format: destination = source op source2 ) {
      Register& L1 = fetch(source);
      Register& L2 = fetch(source2);
      adjustNextUse( source );
      adjustNextUse( source2 );
      L = & mostReplaceableRegister();
      cout << *L <<  " = " << L1 << op << L2 << ";\n";
   } else if ( i has format: destination = op source ) {
      Register& L1 = fetch(source);
```

```
      adjustNextUse( source );
      L = & mostReplaceableRegister();
      cout << *L <<  " = " << op << L1 << ";\n";
   } else if ( i has format: *source = source2 ) {
      Register& L1 = fetch(source);
      Register& L2 = fetch(source2);
      adjustNextUse( source );
      adjustNextUse( source2 );
      cout << "*" << L1 << " = " << L2 << ";\n";
   } else {
      // Other instruction format are straightforward.
   }
   if ( L ) {
      placement[destination] = singleton(L);
      dirty_variables += destination;
      adjustNextUse( destination );
   }
}
```

We treat all variables as memory based, and for a memory-based variable, the output operator `<<` is defined to emit the string `*(`*type*`*)(`*base*`+`*offset*`)` corresponding to that variable. Similarly, for a register, `<<` is defined to print out the register's name.

Chapter 9

OPTIMIZATION

Add tail-call optimization.

9.1 Basic transformations

Once we know which indices dominate/reach the entrance to each basic block, a simple front-to-back traversal of each block will show which indices dominate/reach each point in the block, and we can do whatever processing we choose based on that information. If the analysis is based on the reverse graph, we need to walk through each block from its last instruction to its first.

For the basic optimizations, we first do an analysis and then remove optional instructions during an instruction-by-instruction pass through each basic block. Recall that the complexity of the analysis is the number of basic blocks times the length of the longest acyclic path in the flow graph, which can be as long as the number of basic blocks. And, that is the number of set operations, each of which can involve as many operations as there are indices. The point is that analysis is very costly compared to a simple pass through the intermediate code.

9.1.1 Available-code elimination

Recall that a form is available at the points dominated by its occurrences against definitions of its operands. So, to determine which forms are available at the entrance of each block, we execute `map<Block,set<Form>> available = analysis( attacks, defends, forms, blocks, all, fwd )`, where forms is the set of all `Forms` occurring in the program, `blocks` is the set of all `Blocks` occurring in the program, and `attacks()` and `defends()` are defined as follows:

```
bool
attacks( Form f, Instruction q ) {
  return( q.form == f );  // q is an occurrence of f.
}

bool
defends( Form f, Instruction q ) {
```

```
    // return  q defines an operand of f ? true : false;
    if ( q.form.destination == f.destination ) return true;
    if ( q.form.destination == f.source  )     return true;
    if ( q.form.destination == f.source2 )     return true;
    /* otherwise */                            return false;
  }
```

To speed things up a bit, we compute the following map:

```
  map<Variable,set<Form>> K;
  for ( each variable v ) {
    K[v] = set of all forms having v as an operand;
  }
```

Then we do a forward traversal of each block, removing the available instructions.

```
  for ( each block B in the flow graph ) {
    Set<Form> current = available[B];
    for ( each instruction i in B from first to last ) {
      if ( i.in_degree() == 0 ) current = empty;
      if ( i.form is in current, is non-volatile, and has a dest. ) {
        remove i from B;
      } else {
        current -= K[i.form.destination];
        current += i.form;
      }
    }
  }
```

Note that the above code does not work correctly for instructions like `A=A+B`, but works correctly on the equivalent sequence, `T=A+B;A=T`.

Eliminating an available occurrence of a form will not decrease the availability of any other forms and may, in fact, availabilize some occurrences of some other forms. For example, in the block

```
   x=y; z=x; x=y; z=x; w=z;
```

eliminating the available occurrence of `x=y` make the second `z=x` optional. So additional passes of available-code elimination can be of benefit.

Note. To repeat some sequence of instructions until nothing further changes, i.e., *stability* occurs, use a loop of the form:

```
  bool stable;
  do { stable = true;     // We quit if  stable  stays true.
    //  The repeated sequence of instructions:
    ...
    if ( whatever ) {
```

```
      ...                  // Change some stuff.
      stable = false;
    }
    ...
  } while ( ! stable );   // Quit if  stable  stayed true.
```

The resulting code is likely to be correct and more readable than most alternative forms.

9.1.2 Dead-code elimination

To eliminate dead code, we first do a dead-variable analysis and then a removal pass. Dead-code elimination sometimes removes the only reachable use for the destination of other instructions, thus making them dead, etc. So, additional passes of dead-code elimination may yield additional improvement.[1] After performing dead-code elimination until stability, we are left with only those defining instructions that are directly or indirectly used in calculating a value that is output or used as the condition for a conditional jump.[2]

The points at which a variable is dead are those that its definitions plus the exit instructions co-dominate against its uses. To determine the dead variables at the exit of each basic block, we execute `map<Block,set<Variable>> dead = analysis(attacks, defends, variables, basicBlocks, all, bkwd)`, where `attacks` and `defends` are defined as follows:

```
bool
attacks( Variable v, Instruction q ) {
  if ( q is an exit instruction ) return true;
  if ( q.form.destination == v )  return true;
  return false;
}

bool
defends( Variable v, Instruction q ) {
  return( q.form.source == v || q.form.source2 == v );
}
```

We then make a back to front pass through each block removing dead instructions.

```
for ( each basic block B in the flow graph ) {
  Set<Variable> current = dead[B];
  for ( each instruction i in B from last to first ) {
    if ( i.form.destination is in current && i is non-volatile) {
      remove i from B;
```

[1] Of course, eliminating a dead instruction occurrence cannot bring another to life.

[2] Actually we might have a situation where variables are simply used to define each other. There are removed through "aggressive" dead code elimination, which we study later.

```
      } else {
        current += i.form.destination;
        current -= i.form.source;
        current -= i.form.source2;
      }
    }
  }
```

Note that the above works for instructions of the form `A=A+B`.

9.1.3 Normalization

We commonly set aside a special temporary, say T_{X-Y}, for holding the value of a particular expression, say `X-Y`, and replace occurrences of instructions of the form `Z = X-Y` by a pair of instructions, namely, T_{X-Y} `= X-Y` and `Z =` T_{X-Y}. This transformation, which we will call *normalization*, visibly is valid and can help by making T_{X-Y} `= X-Y` available wherever `X-Y` has an unkilled occurrence on each incoming path. The transformation is viewed as *essentially harmless*, since copy propagation will usually remove the additional copy instruction.

Immediately after normalization, in a correct program, any normalized form, say T_{X-Y} `= X-Y`, must be available wherever it is live (i.e., it is dead, hence optional, wherever it is unavailable)[3] — otherwise an incorrect value of `X-Y` could reach a use of T_{X-Y}. For example, such an instruction is available at the entrance of every block at which it has an in-block reachable use, and dead at the exit of every block in which one of its sources has an in-block definition with no subsequent occurrence of that form. Under valid and harmless code rearrangement, however, that situation will not necessarily continue to hold, e.g., `T = X-Y; Z=T; X=A;` remains correct when the last two instructions are reversed.

9.1.4 Copy propagation

This information should be stored as equivalence classes of variables.

Wherever `X == Y` holds, we can validly and harmlessly replace any use of `X` with `Y` (and vice versa), thereby, for instance, converting occurrences of `X = Y` and `Y = X` into *trivial* copy instructions of the form `X = X`, which visibly are optional. Such transformations are called *copy propagation*.

The most common heuristic is to substitute uses of `Y` for uses of `X` wherever `X = Y` is available, in hopes of killing (some of) those incoming occurrences of `X = Y` and to replace uses of ordinary variables with normalization variables.

Note, however, that there are points at which `X = Y` is available on some incoming paths and `Y = X` is available on others. At such points `X == Y` holds, even though neither `X = Y` nor `Y = X` is available. (Such points are those dominated by occurrences of `X = Y` and `Y = X` against definitions of `X` or `Y`.) A possible heuristic for dealing with such situations is to give each variable a priority, say via alphabetic ordering, and propagate in the direction of

[3] This does not mean that each occurrence is optional — a form is dead ahead of each of its occurrences and available after each.

higher priority, i.e., replace variables of lower priority with those of higher priority whenever possible. This policy can availabilize more instructions and kill or trivialize some copy instructions. Of course, each possible ordering gives a different optimization, so after a pass of copy propagation, available-code elimination, dead-code elimination, and trivial-copy elimination, it can be helpful to reorder (e.g., reverse) the priorities of the variables and make another such pass.

Note that we can expand our sets of available instruction forms by substituting equals for equals. Expand.

9.2 Code Motion

Since a control graph's entry instructions have in-degree zero, the flow graph's entry blocks also have in-degree zero, and instructions in entry blocks are start-up code that gets executed only once during an invocation of the graph — in the case of a program graph we view such instructions as "essentially free." Just ahead of each entry block, we add a pre-entry block that we imagine to precompute a possibly garbage initial value for each variable. An edge from a pre-entry block to the corresponding entry block is called an *entry edge*.

If the control graph is that of a function, pre-entry blocks represent activity of the function's caller, i.e., the initialization of parameters and globals. If the control graph is that of a program, pre-entry blocks represent the precomputation of initial values by the compiler, linker, or loader — values that get loaded into memory by the loader or incorporated into target code as "immediate" operands.

Dually, we add a post-exit block after every exit block and call the edge from an exit block to its post-exit block an *exit edge*.

9.2.1 Code hoisting (partial redundancy elimination)

Two instruction forms are said to be *independent*[4] if and only if neither defines an operand of the other — i.e., the forms that are dependent on a given form are exactly those that define one of its operand or use its destination. For technical reasons, we treat a form as independent of itself.

If two independent non-volatile defining forms have adjacent occurrences, we can validly and harmlessly reverse the order of those two instructions — validly insert a copy of the second instruction just ahead of the first instruction, where its destination is dead, then validly remove the instruction that got copied, since it is now redundant (i.e., available).

A path segment is said to be *clean* for a given non-volatile defining form if and only if all other forms occurring on that path segment are independent of the given form. A given instruction is said to be *upwardly mobile* if and only if the path from its block's entrance to that instruction is clean for its form —

[4]In the literature, the following are equivalent: two instructions are *independent*; neither *blocks* the other; one is *definition transparent* for the other.

i.e., at the block's entrance, the given instruction's form is busy and dead, and the given instruction is its form's first occurrence in that block.

Throughout this section "instruction" means "assignment instruction."

Local. Given an upwardly mobile instruction, one can validly insert a copy of it into each of its block's incoming edges, since its form is dead there. This transformation is harmless, since these insertions availabilize the original instruction, whose subsequent elimination restores the original length of all paths through the block. Subsequent available-instruction elimination will also remove the inserted occurrences from any incoming edges where the instruction's form was already available. So this transformation is helpful on any path through such an edge.

Can we move I/O instructions? To some extent, all I/O variables are dependent on each other. Perhaps we can partition them into dependent blocks.

Global. An *upper-boundary edge* for a given non-volatile defining form is a flow edge where that form is both busy and dead at the edge's head but not so somewhere in the block at the edge's tail, i.e., the tail block is not clean for the given form or that form is not busy or not dead at the tail block's exit. Note that entry edges fulfill this condition for instructions that are busy and dead there, since pre-entry blocks are assumed to define each variable with a possibly garbage initial value and, therefore, not clean for any form.

Transformation. To *hoist* a given non-volatile defining form, insert an occurrence of it into each of its upper-boundary edges, and then remove all of its upwardly mobile occurrences.[5] To insert an occurrence of a form into an edge, if the block at the edge's head has in-degree one, we can simply prepend the inserted instruction at the head of that block. If the block at the edge's tail has out-degree one, then we can simply append the inserted instruction to the end of that block (just before any final goto-instruction). Otherwise, we must break that edge by inserting a new block containing only the inserted instruction. (To facilitate this transformation, as soon as the flow graph is formed, we can insert dummy blocks into each edge whose tail block has out-degree greater than one and whose head block has in-degree greater than one. The program is then said to be in *edge[-split* form.)

Fundamental fact. If a non-volatile defining form is busy and dead at a given point, then on each out-bound path that form will occur and will be busy and dead up to its first occurrence. So the intervening path segment will be clean for that form. If the given point is a block entrance (e.g., the head of an upper-boundary edge), that first occurrence will be upwardly mobile, and the intervening segment will include no upper-boundary edges, since the form is busy and dead at each point on that path. (So the number of occurrences that hoisting removes from an execution path is at least as large as the number it inserts, i.e., hoisting is harmless.)

[5] Care must be taken in moving instructions with side effects other than defining a destination. Often, we can treat such a side effect as defining an additional destination variable, e.g., a divide-by-zero flag, which might be a source for a subsequent conditional branch.

Similarly, on each incoming path to that point, that form will be busy and dead back to its nearest dependent instruction or branchpoint where that form is not busy and dead. If the given point is a block entrance, then the first edge on that intervening path segment is the form's most recent upper-boundary edge along the given incoming path.[6] Of course, the intervening segment of that path will be clean for the given form. Also, assuming that available code and dead code have been eliminated, that form will have no intervening occurrences along that path.

Corollary. If each of two distinct forms are busy and dead at a given point, then they are independent, since on any outgoing path the one that occurs first will occur on a path that is clean for the other. QED

Validity. The hoisting transformation's insertion steps are valid, since a form is dead at its upper-boundary edges. After those insertions, the form will be fresh and available at each block entrance at which it was (and still is) busy and dead, since any incoming path to such a point will be clean for that form after the form's final upper-boundary edge along that path. In particular, after those insertions, the form will be available at all of its upwardly mobile occurrences, which can, therefore, validly be removed. QED

Improvement. After the hoisting of a given defining form and the subsequent elimination of its available occurrences, the sequence of instructions along any path through the transformed graph can be obtained from the corresponding path through the original graph by moving each upwardly mobile occurrence of the given form to its nearest preceding upper-boundary edges, and removing it if it is available there. Improvement takes place along any path through an edge where an inserted occurrence gets removed by available-instruction elimination. So a form is *hoist-worthy* (i.e., hoisting it yields improvement) if and only if the form is available at some of its upper-boundary edges.

Commutativity. After hoisting one non-volatile defining form, we can immediately hoist another, without further analysis. The harmlessness of the second hoisting follows from the fact that it would have been harmless had the first not occurred. Its validity follows from the fact that the original hoisting cannot decrease the set of points where the second form is busy and dead. Otherwise, the original hoisting would have inserted or removed an occurrence of the first form at some point where the second form was busy and dead. But, the first form was busy and dead (just ahead of) wherever an occurrence of it got inserted or removed by the original hoisting. So, the forms are independent, since both were busy and dead at that particular point. QED

[6] Such an edge will exist since, by assumption, the form is not busy somewhere in the pre-entry block, so that first block at whose head the form is busy and dead will be an upper-boundary edge.

Notes. It may be that re-analyzing between two hoistings would cause the second form to be hoisted further (i.e., to be busy and dead at more points), thus exposing additional opportunities for optimization. Also, note that hoisting a form cannot decrease the set of points where that form is fresh or available.

Note that, when an occurrence of a form is inserted into an upper-boundary edge, that form becomes available at the edge's head, but forms that use the inserted edge's destination are no longer available there. Moreover, when an upwardly mobile occurrences is removed, the form's destination becomes live at some of the upper-boundary edges where it was previously dead if any of the form's upwardly mobile occurrence was live. Hoisting a variable has no effect on the liveness of other variables at the form's upper-boundary edges.

9.2.2 Code sinking (partial uselessness elimination)

The directional-dual transformation to hoisting (i.e., co-hoisting) is called *sinking*. It exchanges *available* for *busy* and *fresh* for *dead*. It has properties that are dual to those of hoisting, and its validity and harmlessness can be proved by the directional duals of the arguments for the validity and harmlessness of hoisting. There is one twist, however. Even though deadness and availability are not directional duals of each other, they play corresponding roles, providing validity and improvement, for sinking and hoisting, respectively, e.g., after sinking a form improvement takes place along any path through an edge where an inserted occurrence of that form gets removed by subsequent dead-code removal.

Local. If a given occurrence of a defining form is available and fresh at its block's exit, one can harmlessly and validly insert an occurrence of that form into each of the block's outgoing edges, thereby, killing the form's final in-block occurrence. Subsequent dead-code elimination will remove all inserted occurrences from outgoing edges where the form was dead, so this local transformation is helpful on any path out of the given occurrence along which the form is dead.

Global. Sinking inserts occurrences of a given non-volatile defining form into every edge where the form is available and fresh at the edge's tail but not so somewhere in the block at the edge's head, i.e., at each *lower-boundary edge*. This transformation makes the form busy and dead at each block exits where it was fresh and available. It thus kills each *downwardly mobile occurrence* of the form. Obviously, a form is *sink-worthy* (i.e., sinking it yields improvement) if and only if the form is dead at some of its lower-boundary edges.

9.2.3 Oscillation

CodeMotion is the program transformation that results from repeating the following steps until stability:

1. Hoisting phase:

   (a) Dead-code analysis and removal, until stability.

   (b) Busy-code analysis.
   (c) Hoist each hoist-worthy form.

2. Sinking phase:
   (a) Available-code analysis and removal, until stability.
   (b) Fresh-code analysis.
   (c) Sink each sink-worthy form.

Here, we can omit the removal of upwardly (downwardly) mobile occurrences when hoisting (sinking), since the subsequent removal of available (dead) code will remove them anyway.

Minimizing live-ranges via sinking. To diminish register pressure, it is helpful to diminish the range of the program over which variables are live. Since sinking moves definitions closer to the uses they reach, live ranges can be dimished via a pass that sinks all forms, including those that are not sink-worthy.

This really needs some proof. Moving an instruction downward decreases the live range of its one destination but may increase the live ranges of its, possibly multiple, sources.

9.2.4 Unhoisting and unsinking

Whenever there is an occurrence of a form on each outgoing edge from a block (i.e., each successor block has in-degree one and a clean path to an occurrence of that form), and the block's final (branch) instruction is independent of that form, then an occurrence of that form can validly be appended to the end of the block (just ahead of the final branch) where that form is dead. The occurrences in the immediate successors will subsequently be eliminated by available-code elimination. The dual holds for incoming edges. Note that these transformations are the opposites of local hoisting and local sinking.

9.2.5 Hoisting initializations (constant folding)

An edge from a pre-entry block to its entry block is called an *entry edge*. As previously mentioned, pre-entry blocks (are imagined to) contain definitions that initialize all variables (possibly with garbage), so each entry edge is an upper-boundary edge for forms that are busy and dead at that entry. An occurrence of a form can be inserted into an entry edge either by appending that occurrence to the pre-entry block or by prepending it to the entry block.

In the case of a program graph, instructions in pre-entry blocks can be precomputed at compile time or may need to be deferred until assembly, linking, loading or program initialization — the value of some constants, e.g., the size or offset of an item, may not be known during compilation. In any case, for program graphs, appending an instruction to a pre-entry block is harmless, and hoisting any form that is busy and dead at an entry edge is helpful, i.e., improves the code, since an occurrence inserted into an entry edge has no cost.

The transformation that consists of inserting a copy of a non-volatile defining form into (the pre-entry block of) each entry edges where it is dead[7] and not already available will be called "pseudo-hoisting." Visibly pseudo-hoisting is valid, harmless on program graphs, and can be helpful by availabilizing (making redundant) some of that form's occurrences, specifically, those dominated by the entry instructions at which the instruction's destination is dead against the definitions of the instruction's operands.

The removal of such an available occurrence will usually make the form's destination live at some of those entry edges where it was dead — no further definitions of that variable can validly be inserted such an edge. Therefore, the order in which forms are pseudo-hoisted makes a difference. One possible approach is to follow a greedy heuristic:

```
do {
  dead_variable_analysis();
  remove_dead_code();
  for ( each variable ) {
    Among the program's non-volatile defining forms having that
    variable as their destination, pseudo-hoist the one, if any,
    whose pseudo-hoisting is estimated to yield the largest
    improvements, e.g., removes the most occurrences..  Then
    update analysis information at the affected entry edges.
  }
} while ( ! stable );
```

Of course, in a normalized program there are fewer choices to make, since there is only one form that defines each expression variable.

Definition. A non-volatile variable is said to be a *constant of rank zero* if and only if it has no definitions other than those in the pre-entry blocks. A non-volatile variable is a *constant of a rank n* where $n > 0$ if and only if

1. That variable has only one defining form.
2. Occurrences of that form dominate all uses of that variable, i.e., that form is dead at each entry edge.
3. That form's sources are constants of rank lower than n.

The important point is that the values of the rank-zero constants of a program graph are pre-computed.

Constant folding. After the sole defining form for a rank-one constant has been pseudo-hoisted, that form will be available at each entry edge (by property #2). Thus, since that constant has no definitions of any other form (by property #1) and its sources have not defintions outside the pre-entry blocks (property #3), the constant's sole defining form will be available everywhere in the graph. Removing the form's now available occurrences will reduce the constant's rank

[7]If an uninitialized variable is live at an entry edge, its has a use that can be reached with an uninitialized value — not a good thing.

to zero. Pseudo-hoisting all rank-one constants reduces by one the minimum rank of all constants of rank greater than one. By recursion, all constants can be *folded*, i.e., reduced to rank zero.

Along with constant folding, one should simplify expressions of the form `X*1`, `X*0`, `X+0`, and so on. Also, if the condition of a conditional branch is a known constant, that branch should be replaced with an unconditional branch, an operation that restructures the flow graph.

The following paragraph is a work in progress. Don't try to figure it out. Perhaps this should follow the SSA form.

A variable is *predictable* if and only if each of its uses is dominated by occurrences of a single form whose destination is that variable and whose sources are constants — possible a different form at each use of that variable. A variable is said to be *constrained* if and only if it is predictable or it is the destination only of copy instructions whose right side are constrained. Such variables can hold only a small set of possible values, which can be determined at compile time via induction on rank, just like constants.

9.3 Loop Optimization

A block B is said to *dominate* a block B′ if and only if all paths from an entry block to B′ reach B, i.e., B and B′ are the same block or B's singleton dominates the entrance of B′ against the empty set. It is easy to show that the dominance relationship:

- is *reflexive*, i.e., each block dominates itself.
- is *transitive*, i.e., B_1 dominates B_3 whenever B_1 dominates B_2 and B_2 dominates B_3.
- is *antisymmetric*, i.e., B_1 is equal to B_2 whenever B_1 dominates B_2 and B_2 dominates B_1.
- has *linear predecessor sets*, i.e., given two dominators of a block, one will dominate the other.[8] It follows that the dominance graph is a tree.

9.3.1 Finding Loops

A *loop* is a set of blocks that:

1. has a *header*, i.e., a member that dominates the rest of the set,
2. is *strongly connected*, i.e., between any two blocks in the set, there is a path that stays entirely within the set.

[8] From one of those dominators there's a path to the dominated block that avoids the other. Were there a path from an entry to the one that avoided the other, we could concatenate those paths, thereby, constructing a path to the dominated block that avoids that other dominator — a contradiction.

It is easy to see that a header block must dominate at least one of its immediate predecessors, e.g., the next-to-last block on any in-loop path from any block in the loop to the header. Conversely, we will show that any block that dominates one of its immediate predecessors is a header. To find all headers, check each block to see which are dominated by one of their successors and mark those successors as headers.

Suppose that B dominates one of its immediate predecessors, say P. The edge from P to B is called a *back edge*. The *natural loop* associated with this back edge consists of all blocks on paths from B to P that have only one occurrence of B:[9]

- To see that the natural loop is strongly connected, note that each of its members has a path to and from B.
- To see that its members are dominated by B, use a simple proof by contradiction.

To construct the natural loop, put B into the set under construction and put P onto a worklist, and, as long as there are blocks on the worklist, move one of them to the set under construction and put its immediate predecessors that aren't already in the set onto the worklist. Induction on length shows that each block of each path segment to P that avoids B is eventually in the constructed set — so the constructed set includes the natural loop. Conversely, step induction shows that any given member of the constructed set is on such a path segment. But, since B dominates P, the given member is dominated by B and, therefore, has an incoming path from B. By concatenating that path's final segment from its last occurrence of B with the member's B-avoiding path segment to P, we see that the given member of the constructed set is in the natural loop.

The following notes show the relationship between natural loops and loops in general:

1. If two loops intersect, then the header of one is a member of the other, since one header must dominate the other, which, in turn, must occur on any path from the dominating header to any block in the intersection. (By strong connectedness of loops, to each member of the intersection there is such a path lying entirely within the dominator's loop.)
2. Any given loop headed by a member of a natural loop other than the natural loop's header is a subset of that natural loop (by induction on the length of the shortest path from a member of the given loop to that loop's header).
3. If two natural loops with distinct headers intersect, then one is included in the other.
4. The union of a nonempty family of loops that have the same header is also a loop.

[9]Such paths may contain cycles, but they can't cycle through B.

5. Unless a family of natural loops is a singleton, its union will not be a natural loop. In fact, the union will have a back edge for each distinct natural loop in the family.

6. Any given loop having a given header is a subset of a union of the natural loops associated with the back edges to that header that are in the given loop.

7. Many natural loops have proper subloops with the same header, but, of course, such subloops aren't natural.

9.3.2 Code Motion and Loops

Code motion can move code out of loops, even loops that have not been discovered.

Can we do a recursive defiition of invariants, as we have with constants?

Hoisting and loops. A non-volatile defining form is said to be *invariant* with respect to loops that have no definitions of its operands.[10] If a form is busy and dead at the entrance of the header of a loop in which that form is invariant, hoisting that form makes it available at the header's entrance. If, in addition, occurrences of that form are the only in-loop definitions of its destination, as would be the case with normalized forms, then that form will be available everywhere in the loop, and all in-loop occurrences of that form will be eliminated via the elimination of available code.

Note the analogy to "constant" and to constant folding. Also, perhaps the term "constant" should be "invariant everywhere."

Loop rotation. Consider a leading-test loop where the header consists of only that leading test. An invariant form that is busy on all other paths through the loop might not be busy at that exit. In such a case, the form would not be busy at the loop entrance, and hoisting would not make it available there. But, if there is a path consisting only of blocks with a single in-loop successor, from the header to a block at whose entrance the form is busy and dead, then the loop can be *rotated* until that block becomes the header. To rotate a loop whose header has a single in-loop successor, clone the header and switch all out-of-loop edges that pointed to the original header to point to the clone, which is not in the loop. The clone has the same in-loop successor as the original, and that successor is the loop's new header.[11] (When two immediate successors of the header are in the loop, we can create two copies of the loop and rotate each so that each successor heads one of the loops.)

Sinking and loops. Sinking too can improve loops, but the duality isn't perfect, since a loop can have multiple exits but only one entry. If a form is fresh and available at all loop exits where it is live, then (even if the form isn't

[10] Each in-loop occurrence of an invariant form would be reached by some such definition if there were any.

[11] If you find this construction unclear, think about how one rewrites a leading-test loop as a trailing-test loop inside an if-statement.

invariant) sinking will kill all of its in-loop occurrences that are loop-dead, i.e., that don't reach any in-loop uses of the form's destination.

9.3.3 Loop Unrolling

Loop unrolling involves replicating the body of the loop n times and then repeating the loop only $1/n$ as often. Thus the control instructions are executed only $1/n$ as often. When this transformation is combined with available-code elimination, it can have a beneficial effect, since a value used in one iteration of a loop may have been computed in the prior iteration. Loop unrolling can, however, lead to diminished performance, whenever the expanded loop body overflows the instruction cache, thereby decreasing the cache's hit ratio.

Note that values computed in one iteration of a loop might be used in the next. Unrolling combined with hoisting and available-code removal can automatically take advantages of such situations.

Generalize both of these.

9.3.4 Loop Jamming

Loop jamming consists of combining two loops that are executed the same number of times. The combined body is governed by a single set of control instructions. Note that loop jamming is subject to the same performance anomaly mentioned for loop unrolling.

Exercise. Give reasonable sufficient conditions for loop jamming to be a valid transformation. How can these conditions be checked?

9.3.5 Elimination of Induction Variables (Strength Reduction)

The following optimization will, for example, turn a search that steps through an array by successively incrementing a subscript into an equivalent search that steps a pointer through that array. Such optimization is especially important in FORTRAN, which has no pointer variables.

Affine functions

With respect to a given loop, an *affine function* of a variable I is a function of the form:

$$\mathrm{A} * \mathrm{I} + \mathrm{B}$$

where A and B are invariant with respect to that loop.[12] Visibly, an invariant plus or times an affine function of I is also an affine function of I. So, the composition of affine functions is an affine function.

[12] An expression is said to be invariant with respect to a loop if and only if none of its sources has a definition in that loop.

Suppose that J is a non-invariant affine function of I, say $A_J * I + B_J$, where A_J is nonzero. Then I is equal to $(1/A) * J + (-B/A)$. Also, it is easy to show that every affine function of I is an affine function of each non-invariant affine function of I. In particular, each non-invariant affine function has an (affine) inverse.

Note that whenever I is incremented by a loop invariant C, the affine function $A * I + B$ is incremented by $A * C$, which is also loop invariant.

Define the notion of "affine relative," which will include both correlates and derived induction variables.

Induction variables

For a given loop, a *basic induction variable* I is one whose (possibly multiple) definitions in the loop increment or decrement I by a loop invariant, i.e., the in-loop definitions of I are all of the forms:

$$I = I + C$$
or
$$I = I - C$$
or
$$I = C + I$$

where C is a loop invariant, but possibly a different one at each of these definitions.[13] (Note that loop invariants fulfill this definition vacuously, i.e., they are basic induction variables.)

Two basic induction variables I and J are said to be *correlated* if they are incremented or decremented at essentially the same points in the loop (i.e., points separated only by instructions that increment or decrement other basic induction variables) and they are incremented or decremented by amounts that are always in the same loop-invariant ratio, say A. Then, except between those corresponding updates, J is an affine function of I, namely $A * I + B$, where B is the value of $J - A * I$ at the loop entry. (Note that loop invariants fulfill this definition vacuously, i.e., they are correlated with all basic induction variables.)

Let I be a basic induction variable. A variable X is said to be an *I-based*[14] *induction variable (of rank n)* if and only if either $n = 0$ and X is I or $n > 0$ and X has a single in-loop definition, and that one has the form:

$$X = J \; op \; D$$

where

1. *op* is $+$, $-$, $*$, or $/$.
2. D is loop invariant.
3. J is an I-based induction variable of rank $n - 1$.

[13] In a normalized program these definitions would be of the form $I = T_{I+C}$, etc.

[14] Some authors use the terminology "in the family of I" instead of "I-based."

4. If J is non-basic (i.e., $n > 1$), its lone definition dominates this definition of X against all definitions of I (i.e., that definition of J is the only definition of J reaching this definition of X, and there are no intervening definitions of I.)

A simple induction on rank shows that X gets assigned an affine function of I, say $A_X * I + B_X$, at that lone definition. Code to compute the invariants A_X and B_X can be built recursively from A_J, B_J, and D. For example, if *op* is $*$, then A_X is $D*A_J$ and B_X is $D*B_J$.

Elimination of non-basic induction variables

From affine computations. Given any non-basic I-based induction variable, we can eliminate its use in defining other I-based induction variables. For instance, in the above definition of X, pick a new temporary T_X to hold the value of the affine function $A_X * I + B_X$, replace the definition of X by $X = T_X$, so that subsequent copy propagation will replace all in-loop uses of X by corresponding uses of T_X. Then either:

- Insert, ahead of $X = T_X$, the instruction pair:

  $$U = A_X * I;$$
  $$T_X = U + B_X;$$

  where U is another new temporary. This insertion is essentially harmless, whenever B_X is 0 or A_X is 0 or 1, and is always valid since the inserted instructions are equivalent to $T_X = A_X * I + B_X$. This transformation helps by possibly killing some non-basic induction variables.

- Or, turn T_X into a basic induction variable that is correlated with I by inserting the above instruction pair into all external edges into the loop and maintaining the value of T_X throughout the loop by incrementing or decrementing it by appropriate amount whenever I gets incremented or decremented, i.e., after each in-loop instruction of the form $I = I \pm C$ or $I = C + I$ add

  $$U = \pm C * A_X;$$
  $$T_X = T_X + U;$$

  where U is a new temporary. Note that $U = \pm C * A_X$ is loop invariant that will be hoisted out of the loop if its occurrences dominate all exits, and will be eliminated by constant folding and copy propagation if its sources are constant.

From tests. Suppose that the only use that keeps a non-invariant basic induction variable X from being eliminated is an occurrence of $Z = X$ *relop* Y, where Y is another (possibly invariant) basic induction variable correlated with X. Let I be any non-invariant basic induction variable correlated with X — e.g., I could be Y if Y is non-invariant. Then X and Y are affine functions of I, and X *relop* Y equals A_X*I+B_X *relop* A_Y*I+B_Y, which equals I *relop* $(B_Y-B_X)/(A_X-A_Y)$ whenever A_X-A_Y is positive.[15] In such a case, we can validly replace Z = X *relop* Y by Z = I *relop* C, where C is a loop invariant initialized to the value $(B_Y-B_X)/(A_X-A_Y)$, i.e., to (Y – X)/(A_X-A_Y) + I.

Give the algebra and an example.

From incrementing/decrementing instructions. Aggressive dead-code elimination remove any basic induction variable whose only remaining uses are to increment and/or decrement itself — see section 9.4, page 97.

Exercise. Consider each of the six possible *relop*s individually and show how to deal with rounding issues in cases where $(B_Y-B_X)/(A_X-A_Y)$ is not an integer.

Exercise. Show that the concept of *I-based induction variable* can be extended to include definitions of the form X = D *op* J where *op* is +, –, or ∗, but not /.

Exercise. Show that the definition of *I-based induction variable* can be extended so that sums of I-based induction variables are also allowed as I-based induction variables.

9.4 SSA Form

Note that the following are equivalent:

- B dominates all predecessors of BB.
- B strictly dominates BB.[16]
- Some dominance-child of B dominates BB.

A given block belongs to the *dominance frontier* of a block B if and only if B dominates some but not all of that block's predecessors, i.e.:

```
inline set<Block*> df( Block* B ) {
  // Return the dominance frontier of B.
  static map< Block*, set<Block*> > cache;
  if ( ! cache[B] exists ) {
    // Compute and memorize df(B) -- dynamic programming
    for ( each successor X of B ) {        // loop #1
```

[15]If A_X-A_Y is negative, we simply replace *relop* with its converse, but it is important that this determination be possible at compile time.

[16]In general, an object is said to be *strictly* related to objects other than itself to which it is related, e.g., less-than is a strict, while less-than-or-equal is not.

```
      if ( B doesn't dominate X ) cache[B] += X;
    }
    for ( each dominance-child C of B ) {  // loop #2
      for ( each block X in df(C) ) {
        if ( B doesn't dominate X ) cache[B] += X;
      }
    }
  }
  return cache[B];
}
```

It is easy to prove that df(B) is a subset of the dominance frontier of B. To show conversely that the dominance frontier of B is a subset of df(B), suppose that X is a member of the dominance frontier of B. If X is a successor of B, loop #1 adds X to df(B). If X is not a successor of B, then B strictly dominates some predecessor of X, i.e., some dominance-child C of B dominates that predecessor of X — but C can't dominate X (otherwise, B would dominate X), so C can't strictly dominate X, so X has a predecessor that's not dominated by C, so X is in df(C), so loop #2 adds X to df(B). QED

Note that in both loops, B dominates X if and only if B = idom(X), where idom(X) denotes the dominance-parent of X, a.k.a., the *immediate dominator* of X. That's obvious in loop #1. In loop #2, suppose that B dominates X. Then B strictly dominates X and, therefore, dominates idom(X). C dominates one predecessor of X but not some other, so C does not dominate idom(X), so idom(X) strictly dominates C, so idom(X) dominates idom(C), i.e., B. Thus, to each block, we need only attach its idom and a list of its dominance children — we need not store the entire dominance relation.

Note. A block is said to be *control dependent* on the members of its co-dominance frontier. Once execution crosses crosses a block's co-dominance frontier, that block will surely be executed. Until that crossing, however, there is a possibility (depending on the outcome of tests) that the block will not be executed.

Establishing SSA form. We put a flow graph into *static single-assignment (SSA) form* by splitting each variable. To *split* a variable X, we prepend an occurrence of the pseudo-definition X = ϕ to the dominance frontiers of all blocks containing a definition of X. When that process has been repeated to stability for all definitions of X, including newly added pesudo-definitions, *each point where X is live is dominated by one definition of X against all other definitions of X.*[17] That definition will be that variable's nearest ancestral

[17] To see that this is so, note that before reaching a point that a given definition of X doesn't dominate, any out-bound path must first cross that defintion's dominance frontier, thereby reaching a redefinition of X.

definition up the dominance tree.[18] Next, we append a unique suffix to the destination of each definition of X. Finally, we relabel each use of X with its suffixed version whose definition dominates that use.[19]

Under this relabeling, an occurrence of $X = \phi$ prepended to a given block becomes $X\_S = \phi$ for some suffix S. For purposes of presentation, such a pseudo-definition is usually represented $X\_S = \phi(X\_S_1, X\_S_2, \cdots, X\_S_n)$, where X_S_i is the suffixed version of X whose definition dominates the block's i-th incoming edge. This pseudo-definition has the same semantic meaning as inserting $X\_S{=}X\_S_i$ into the i-th incoming edge, for $i = 1 \cdots n$. Note that the definition of an operand of a pseudo-definition does not dominate the pseudo-definition's block but dominates its respective predecessor block(s). That operand is dead, however, at the exit of that predecessor block, since *after a variable has been split, no more than one of its suffixed versions is live at any point*, namely the one that dominates that point against all other definitions of suffixed versions of that variable.

Optimization of programs in SSA form. Repeat the following steps until stability (or the point of diminishing returns):

- Eliminate the single definition of any variable having no uses.
- Replace, with `X=Y1` any instruction of the form `X =` ϕ`(Y1,Y2,`$\cdots$`,Yk)` where all the `Y`'s are the same variable.
- Eliminate any occurrence of `X=Y`, and replace all uses of `X` with `Y`.
- Pseudo-hoist to the entry edges the single defining form for any rank-one constant.
- Replace, with the corresponding unconditional jump, any conditional jumps whose condition variables is constant, and eliminate the corresponding control edge.[20]
- Eliminate any block, other than pre-entry blocks, whose predecessor count is zero, and eliminate its outgoing control edges.

Is a formal definition of X available at a given point if and only if a suffixed version of that definition dominates that point? I don't think so.

Aggressive dead-code elimination. It is possible that two variables are live simply by virtue of the fact that each is a source in a definition of the other, but neither participates in any real computation. A variable is *fundamental* if and only if it is an output variable, the condition for a conditional branch, the source for an indirect jump. Any fundamental variable is *significant*, as is any variable that is a source of an instruction that defines a significant variable.

[18] After we split all variables, the locations of dominating definitions can be passed down the dominance tree via a traversal wherein we attach to each block a map that associates with each variable the location of its nearest ancestral definition from the exit of that block.

[19] As a matter of implementation, we represent a suffixed version of X via a pointer to its unique defining instruction.

[20] Also, decrement the predecessor count of the corresponding successor block and adjust argument lists of its ϕ-instructions.

Exercise: Give an algorithm for determining the significant variables in an SSA flow graph. (Use a depth-first search in graph of sources.) Show that it is valid to eliminate all definitions of non-significant variables.

Function form There is a straigtforward method to turn graphs in edge-split SSA form into equivalent collections of function definitions. The basic idea is that each block that has multiple prededessors defines a function whose formal parameters are the destinations of the block's pseudo-defintions. Append to each of the block's multiple predecessors a call to that function; the call's i-th argument should be the value that this predecessor contributes as its source to the block's i-th pseudo-definitions. That function's definition is hoisted to the entrance of the block's immediate dominator (i.e., dominance parent), so that the block's immediate predecessors can issue their calls to the block's function.

9.5 Global Register Allocation

Graph coloring. The graph-coloring problem, that of coloring the nodes of a graph so that adjacent nodes have distinct colors, serves as a paradigm for many resource-allocation problems. The colors correspond to interchangeable but non-sharable units of a resource to be allocated from a pool of say k units of the resource. The nodes correspond to resource users. Coloring a node a given color corresponds to allocating the resource unit corresponding to that color to the user corresponding to that node. We construct an undirected edge between any two users (i.e., nodes) that might require simultaneous use of a resource unit. In such cases, those nodes are said to *interfere*, so the resulting graph is called the *interference graph*. The interference graph is k-colorable if and only if a pool with k units of the resource is enough so that interfering users can each get exclusive access to distinct resource units.

Global register allocation can be viewed as the graph-coloring problem where registers are the resources and variables are the users. Two variables interfere if they are simultaneously live at some point in the control graph,[21] i.e., if a definition of one (other than a copy from the other) occurs where the other is live. Allocating the same name (i.e., register) to distinct variables corresponds to coloring them the same color — specifically, X can validly be replaced by Y throughout a program provided that X and Y don't interfere. This transformation trivializes occurrences of the forms $\mathrm{Y} = \mathrm{X}$ and $\mathrm{X} = \mathrm{Y}$, turning them into optional instructions of the form $\mathrm{Y} = \mathrm{Y}$.

An *X-web* is a set of occurrences of X that reach no use of X outside that set and are reached by no definition of X outside that set. Visibly, we can validly replace all occurrences of X in a given X-web by any new variable not already occurring in the program. Splitting a variable into its disjoint webs introduces no additional copy instructions. The two occurrences of X in a trivial instruction of the form $\mathrm{X} = \mathrm{X}$ may be in different X-webs, and, assuming

[21] For efficiency, some implementations round live ranges up to be unions of basic blocks.

that X is live and defined at that instruction, removing that instruction would coalesce those webs. Conversely, we can split an X-web into two disjoint subsets and validly turn them into distinct webs by relabeling the the members of the respective sets with distinct new variables, say X_1 and X_2, and then inserting an occurrence of X_2=X_1 just after each definition of X_1 and inserting an occurrence of X_1=X_2 just after each definition of X_2. It is helpful to then propagate and sink both of these copy forms.

For every point in the program, the interference graph contains a clique[22] consisting of all variables that are live at that point. If more than k variables are live at some point, then the interference graph contains a local $(k+1)$-clique and is, therefore, not k-colorable. To break up such cliques, we must (perhaps repeatedly) *spill* one of the clique's variables to a memory location, i.e., make it a memory-mapped variable – see below.

Whenever no more than k variables are simultaneously live, it is possible to avoid the need for spill code by shuffling variables among the registers. But, even if k registers are sufficiently many to hold all variables that are live at any point, the interference graph still may have a $(k+1)$-clique. And, even if the interference graph doesn't have a $(k+1)$-clique, it still may not be k-colorable.[23] Mention rematerialization.

Heuristics. Since the graph-coloring problem is known to be NP-complete, we must use a heuristic. First, we set aside any node that is *unconstrained*, i.e., has interference-degree less than the number of registers, say k. Removing such a node can diminish the interference-degree of some remaining nodes, possibly unconstraining some of them. We repeat this step as long as there remain unconstrained nodes in the graph. This *deconstruction heuristic* [CHA] leads to a residual graph (often empty) in which every node is constrained.[24] After k-coloring it, we can easily k-color the removed nodes in the opposite order from that in which they were removed — each, when restored to the graph, will have an available color, since at that point it will have fewer neighbors than there are colors, just as it had when it was removed.

If the residual graph is nonempty, we can use the *saturation heuristic*, i.e., color first those nodes for which the fewest colors remain available [BRE], breaking ties in favor of nodes with the most uncolored neighbors [KOR]. The standard saturation heuristic backtracks whenever it reaches a variable having no available colors, but, when the graph is not k-colorable, that process involves a number of steps that is exponential in the number of variables. So, if possible, we split the uncolorable variable into multiple subscripted versions, each of which might interfere with fewer variables and thus be colorable.[25] But splitting

[22] A *clique* is a set of nodes each adjacent to (i.e., having an edge to) each of the rest.

[23] For example, the Chrysler emblem (a pentagon plus a center point connected to each corner) has no 4-cliques but is still not 3-colorable.

[24] The dragon book (page 546) wrongly claims that such a graph cannot be k-colored, but for example the vertices of a rectangle are two-constrained and two-colorable — in fact, the vertices of a cube are two-colorable even though they are three-constrained.

[25] This SSA-inspired tactic is a refinement of the node-splitting tactic introduced by Chow [?]. Of course, if the graph is already in SSA form, such splitting has already occurred.

might constrain neighbors that interfere with multiple suffixed version. So, before resuming, we must undo the deconstruction process back past the removal of all such neighbors.

If, however, the variable with no available colors is a suffixed version X_S of an already split variable X, give X_S any color not taken by the variables that are live at its single definition. If no such color exists, more than k variables are live at this point, so spill X_S to memory, i.e., remove it from the interference graph and generate code to store it to memory immediately after its definition and to load its value just before each of its uses [CHA]. Otherwise, trim X_S to fit the available range, i.e., insert an occurrence of X=X_S just ahead of all definitions of variables of the same color as X_S at which X_S is live — to insert an occurrence of X=X_S just ahead of a pseudo-definition, append an occurrence of X=X_S to each predecessor of the pseudo-definition's block. Resume and complete the splitting of X by inserting pseudo-definitions of X at the dominance frontiers of the inserted occurrences of X=X_S, etc. The destination of each inserted occurrence of X=X_S will obtain its own suffix, and X_S will be dead immediately following each of those occurrences, since at most one suffixed version of a split variable is live at any point. Finally, remove from the interference graph the corresponding edges to X_S, and add the new subscripted versions of X along with their edges.

Mention the UC Davis paper on the use of integer linear programming for optimal register allocation and minimizing spill code.

9.6 Effects of Aliasing

An instruction of the form `p[d] = x` introduces a *possible definition* of any variable that `p+d` might point to at that point in the program. Similarly an instruction of the form `x = p[d]` introduces a *possible use* of those variables. Note that references, including reference parameters, generate such indirection and the resulting pointer-induced aliasing.

In the optimizations associated with some-path (reaching) analyses, we optimize only when the final sets are sufficiently small, e.g., when there are no reachable uses of a particular variable at a given point in the intermediate code. In such cases, we must include possible definitions and uses in the sets computed in our analyses. Specifically, we must include them in the gen[B]. But the possible definitions cannot be counted on to "kill" a reachable use or reaching definition, so possible definitions have no effect on the kill[B].

In the all-paths (dominance) analyses, we optimize only when something happens on all paths into (or out of) a particular point, e.g., available-instruction elimination. In such cases we must consider possible definitions when computing the kill[B]'s, but do not consider possible uses or definitions when computing the gen[B]'s.[26]

The goal is to shrink the set of possibilities for each pointer, i.e., to find things that the pointers can't p;oint to.

[26] In considering the availability of (possibly indexed) load and store instructions, we need to consider *possible* modifications of operands via store operations. The standard approach is to take the attitude that memory is one of the sources of load instructions and the destination for store instructions.

Making allowances for aliasing is also a problem in the specification of programming languages. Consider for example the C expression `x = (*p)++` where `x` is memory-mapped. Efficient code would fetch the value of `*p`, store it to `x` meanwhile incrementing that value, and then store it back to `*p`. But does that give the correct result in the case where `p` points to `x`? The value of `x` would be incremented in this case, but most programmers would expect the value to be unchanged. To relieve the compiler writer of some of these concerns, the 1989 C Standard states:

> Between the previous and next sequence point [e.g., end of expression] an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. [6.3]

So, according the Standard, evaluating `x = (*p)++` results in undefined behavior, when `p` points to `x`. That allows the compiler to generate code that is optimal for the most common case and to blame the programmer if anything goes wrong.

9.6.1 Safe approximation

Here are some safe assumptions in order of decreasing caution: i.e., increasing precision:

1. A pointer `P` can point to any variable of the appropriate type.
2. `P` can point to any variable `X` whose address gets assigned to a pointer variable anywhere in the program (i.e., an instruction of the form `Q = &X` occurs) and is of the appropriate type for `P` to point to, a policy that the front-end should enforce. (GNU C allocates a variable to main memory if its address is computed.)
3. `P` can point to any variable whose address gets assigned, directly or indirectly, to `P` anywhere in the program.
4. `P` can point to any variable whose address has been assigned directly or indirectly to `P` as of this point in the program — to find these, put the program into SSA form.

Mention volatility and the way gcc that handles variables whose address is taken as semivolatile.

9.6.2 More background

C-minor treats "`*P`" as an abbreviation for "`P[0]`". The allowed pointer definitions are those of the forms `P = Q` and `P = Q + D` and `P = &X`, where `P` and `Q` are pointers, `D` is an integer, and `X` is any variable. The allowed uses of pointers are in pointer definitions (see above), in pointer comparision (provided that the pointers being compared point into the same array), and in lvalues (see below).

A variable is said to be "memory-mapped" if and only if it has a non-null base and offset. For such a variable X:

Should we allocate a special variable to track the memory-mapped variable's address?

- The experession "$\text{base}_X + \text{offset}_X$" is called the *address* of X and is represented "&X".[27]

- The experession "$\text{base}_X[\text{offset}_X]$" is called the *lvalue* of X.

- The instruction forms "$X = \text{base}_X[\text{offset}_X]$" and "$\text{base}_X[\text{offset}_X] = X$" are *load-X* and *store-X*, respectively.

A memory-mapped variable whose base and offset are constant is said to be *static*. A memory-mapped variable whose base is defined by an allocator is said to be *dynamic*. A memory-mapped variable whose base is SP is said to be *automatic*. In a caller-save environment, variables local to (even indirectly) recursive functions must be automatic so that they can be caller-spilled.

We assume that only memory-mapped variables have addresses and that the only loads and stores are those associated with memory-mapped variables. Note that occurrences of load-X kill the availability of any form having X as an operand. Similarly, occurrences of store-X kill the deadness of X of any form having X as its destination.

Two memory-mapped variables, X and Y, are (possible) *aliases* if and only if their addresses are possibly equal. In such a case, a use or definition of one is a (possible) use or (possible) definition of the other.

We can always determine the type of an lvalue since they only occur in load and store instructions, whose variable operand will have a type. This is helpful in turning an lvalue into a memory-mapped variable.
Are the various subscripted versions of a variable in SSA form aliases?

Question. Can a memory-mapped variable acquire an alias if and only if its address gets assigned to a variable.

A memory-mapped variable X is said to be *store-reconcilable* wherever on each incoming path there is a load or store of X with no subsequent possible definitions of X, base_X, or offset_X. X is *load-reconcilable*, where on each incoming path there is a load or store of X with no subsequent definition of X.

A memory-mapped variable is said to be *reconciled* wherever it is both load reconcilable and store reconcilable[28] — at such points, its register and its memory location hold the same value. A memory-mapped variable is *reconcilable* wherever it is load reconcilable or store reconcilable. For program validity, it is necessary that each memory-mapped variable be reconciled at each of its (possible) uses. To achieve this it suffices to store the variable immedately after each of its definitions and to load the variable immediately before each of its uses.[29] Loads and stores are optional, however, wherever the corresponding variable is reconciled. The determination of whether a variable is reconciled is much easier, of course, when the variable is known to have no aliases.

Random considerations. The following are some possibly relevant considerations:

[27]We view an occurrence of &X as an implicit occurrence of X.

[28]That is, at the dominatees of the variable's loads and stores against definitions and possible definitions of the variable, its base, and its offset.

[29]So, any variable that is live and unreconcilable presents a serious problem.

- We can treat each definition of a variable as a grammar rule, e.g., `X = Y + Z` would be treated as the rule `X` $\rightarrow$ `(Y + Z)`. Each symbol then generates the language consisting of all expressions whose value it might hold. When the program is in SSA form, the corresponding grammar tells what values a given variable can have *at each given point in the program*. Also, SSA expressions tell exactly which definition of each variable they involve.

- The possibile values of a variable that generates a non-recursive language are easy to determine. Those with recursive languages can sometimes be restricted, e.g., this or that constant plus the multiples of three and five.

- Our analysis algorithm for determining possible aliases introduces a special indefinite lvalue `Q[?]`, but only when `Q` is an array. We will follow the convention that the indeterminate value `?` plus or minus any integer yields `?`.

- We can tell which aggregates a given pointer might point into by looking for occurrences of their address-of expressions in FIRST of the language generated by that pointer. `P[D]` and `Q[E]` are possible aliases only if they have the same type and `P` and `Q` might point into the same array, a fact that is now easy to check.

- If two variables `X` and `Y` generate different languages, then they are not necessarily equal.

- If `D` is not (necessarily) equal to `E` then `P[D]` is not (necessarily) equal to `P[E]`.

- If `P` is not (necessarily) equal to `Q` then `P[D]` is not (necessarily) equal to `Q[D]`.

- When we take indirect jumps into account, building the control graph is a bit of a chicken/egg problem. We need to know where label-valued pointers might point before we can build the graph, but we must know the graph before we can precisely determine where those pointers might point. We can, however, do a certain amount of flow-independent possibility analysis.

- The front-end converts by-reference formal parameters into by-pointer C-monor parameters. So by-reference parameters of the same type are possible aliases when separately compiling the code for a procedure. When compiling in a context where all calls to the procedure are known ahead of time, we can make a more precise determination, as shown in a later subsection.

Antialias considerations. For each object, its subobject and subsubobjects, etc., form a tree. At run time the objects form a forest, whose roots are the staticly declared objects, the various instances of local variables in the in the stack frames, and the objects allocated by calls to the heap allocator.

Some objects are directly allocated. Others are allocated indirectly as part of the direct allocation of a larger surrounding object. Distinct directly allocated items are disjoint, whether the allocation is static or dyanmic or automatic. If two objects overlap, then one is nested within the other (except for members of a union).[30] We can take advantage the fact that disjoint objects cannot be aliases of each other.

Information such as the type of each pointer and the identity/type of its (allocated) ancestral root need to be presented by the front-end via the symbol table. The absence of such information inhibits optimization.

- The within-relation is a forest-structured partial ordering, with objects of builtin types as atoms.
- * Two objects are *disjoint* if no objects is within both of them, which, given that within is a forest structured relation, implies that one is within the other.
- Definition: two objects *intersect* iff they are not disjoint.
- Definition: one type *majorizes* another iff the first type hereditarily has a component of the second type.
- Majorize is a forest-structured partial ordering on types, and typeof is a strictly monotonicly decreasing function from objects (under within) to types (under majorizes).
- Sizeof is strictly monotonic from types (under majorizes) to ints (under less-than).
- Two objects are the same object iff they intersect and have the same size.
- If *P and *Q are disjoint, then for all integers D and E, $P+D \neq Q+E$, i.e., no P-based variable can possibly be aliased to a Q-based variable. It follows, of course that $P \neq Q$.
- Objects with separate static allocations have separate dynamic allocations. For instance, a given call to operator-new() in the source code can be executed many times. If all we know about two pointers is that they got their value indirectly as a result of one of those invocations, we don't know that they point to distinct objects. If, however, one pointer got its value indirectly from that call and the other got its value indirectly from the execution of another call to operator-new() somewhere else in the source code, then we know that they point to distinct objects.
- If two objects were allocated by distinct constructs in the program's syntax tree, then they were allocated by separate actions, in which case they are disjoint, in which case they are distinct, i.e., not aliases.

[30] Arrays of indeterminate length may also present problems.

Possibilities. To determine possible aliases, put the program into SSA form. With each instruction that defines a pointer `p`, keep a list of its *possibilities*. Each possibility includes:

- A reference to the data structure, usually a tree, that describes the type of a root object of which `*p` is a possible descendant part.[31]
- A reference to the intermediate-code instruction where that root object got allocated.
- The offset of `*p` within that root object (from which we can compute the exact object to which `p` points).[32]

Two possibilities overlap only if they have the same root ancestor type, the same point of allocation[33] of their root ancestor, the offset of one is between the offset and offset+size of the other.[34] If two objects have distinct root ancestors that are allocated distinct executions of the same instruction, it is possible that they fulfill the above criteria without overlapping, i.e., these criteria are necessary but not sufficient for overlap to occur.

Two pointers are *possible aliases* if and only if a possibility for one overlaps a possibility for the other.

The set of possibilities associated with a pointer depends on the form of its definition:

- If p is constant, then we attach the single possibility consisting of the type of the object allocated where p points, this instruction as the point of allocation, and an offset of zero.
- If that defintion is of the form `p = allocate(...)`, then
- If the instruction is of the form `p = q` where `q` is non-constant, the possibility list for `p` will be the same as that for `q`.
- If the instruction is of the form `p = q + d` where `p` and `q` are different variables and `d` is an integer constant, then the possibilities for `p` are exactly those for `q` but with their respective offsets actually incremented by `d`.
- If the instruction is of the form `p = q + d` where `p` and `q` are different variables and d is an integer non-constant, then the possibilities for `p` are exactly those for `q` but with their respective offsets symbolically incremented by `d`. (Whatever that means.)

[31] All objects in the hierarchy have a known size, except that the root may be an array of indeterminate size.

[32] From the offset of `*p`, we can determine the offsets of each of `*p`'s ancestors.

[33] Distinct points of allocation would imply distinct allocations, which would imply disjoint allocations.

[34] Size can be determined from `p`'s referrent type.

- If the instruction is of the form `p = p + d` where `d` is an integer constant, the resulting list for `p` will be the same as the former list for `p` except that the offsets will have `d` added to them. (FIX: not SSA. But, otherwise, what happens with loops?)
- At pseudo-definitions `p =` ϕ, we take the union of the possibilities of the operands.

9.6.3 A more precise determination of aliases

With each point in the program, we can associate two relations among lvalues,[35] the sure-alias relation SA and the possible-alias relation PA. In the case of sure-aliases, close[B] is the intersection of all `out`-sets of immediate predecessors of B. For possible-aliases, it is their union.

Each of these relations is reflexive and symmetric. The sure-alias relation is transitive, but the possible-alias relation is not — for instance, `*P` could be a possible alias for two distinct fields of a record that are not possible aliases for each other. However, a sure alias of a possible alias is a possible alias.

We begin with the simpler problem of non-flow construction of the SA and PA relations. The result of this analysis is less precise, but it is helpful and instructive.

- `P[0]` is a sure alias `Q[D]` if and only if `P` is a constant whose sole value is `Q+D`. Similarly, `P[0]` is a possible alias for `Q[D]` if and only if `Q+D` is a possible value of `P`, in the sense that `Q+D` is derivable from `P` in the program's assignment grammar.
- If `D1`, `D2`, `D3`, and `D4` have sure/possible values such that `D1 - D2 == D3 - D4`, then `P[D1]` is a sure/possible alias for `Q[D2]` if and only if `P[D3]` is a sure/possible alias for `Q[D4]`.
- If `D` is unconstrained (i.e., the assignment grammar for `D` is recursive), then we take the conservative view that `P[D]` is a possible alias for every possible alias of every P-based lvalue.

Flow-based analysis. To determine the effect of a block on the alias relations, we look at the effect of each instruction from first to last, but first we need to tell how to add a given lvalue to an alias set. The procedure is recursive, so that aliasing is inherited by fields of aliased records and entries of aliased arrays.

```
void clear_aliases( lalue P[D] ) {
  set row[ P[D] ] and col[ P[D] ] to { P[D] } in both arrays;
  if ( P[D] is a pointer ) {
```

[35] Mathematically speaking, a relation is a set of ordered pairs. But sets can be represented as boolean-valued functions or arrays, so relations can be implemented as two-dimensional bit arrays. We can represent SA and PA via a single two-dimensional array whose entries are from `enum{no=0,maybe=1,yes=2}`.

```
    for ( each constant D' where p[D][D'] is in symbol table ) {
      clear P[D][D'] ;
    }   // This includes p[D][0], i.e., *P[D].
  }
}
```

Huh? You only alias lvalues.

```
void point_to ( P, Q[D] ) {  // in either array (SA or PA):
  clear each P-based lvalue;
  for ( each D' such that P[D'] and Q[D+D'] appears in the symbol table ) {
    add P[D'] to the Q[D+D'] row and column in that array;
    add the Q[D+D'] row and column to the P[D'] row and column;
  }
}
```

We can now go through the block and consider the effects of each instruction on the alias relations.

Why can't we simply use availability and its logical dual?

```
if ( the instruction has the form  "P=Q+C"  where C is constant ) {
   alias P[0] to Q[C] in both arrays;
} else if ( it has the form  "P=Q+V"  where V is not constant ) {
   // Q must point to an array (or V is a member-pointer).
   for ( each Q[D] in the symbol table (including Q[?]) ) {
      alias P[0] to Q[D] in PA ;
   }
} else if ( it has the form  "P=P+C"  where C is a constant ) {
   // P must point to an array or could be the stack pointer, etc.
   // (This kills loads and stores involving P.)
   for ( each Q[D] in former row[ P ] ) {
      alias P to Q[D+C] in both arrays, if there is such a variable
      alias P to Q[?] in PA, otherwise ;
   }
   // NOT QUITE!
} else if ( it has the form  "P=P+V"  where V is a variable ) {
   // P must point into an array.
   // (We can do better when V is constrained.)
   for ( each Q[D] in former row[ P ] ) {
      alias P to Q[?] in PA;
   }
   // NOT QUITE!
} else if ( it has the form "P=Q[D]" (also P=P[D] and P=*Q ) {
   use one of the cautious approximations from last section;
} else if ( it has the form "P=malloc()" ) {
   we can assume that P points into the heap;
} else {
   it has no effect;
}
```

P[D1] eq Q[D2] implies P[D1+C] eq Q[D2+C].

Exercise. We now know the effect that each kind of instruction has on PA and SA. Show how to precalculate the effect that an entire basic block has on PA and SA. Show how to store this information so that it can be efficiently applied to these arrays during iterative analysis.

9.6.4 Effects of procedure calls

When optimizing code for a function, it is important to note that a formal parameter passed by reference could refer to almost any nonlocal variable of the same type. In fact, two such parameters may refer to the same nonlocal.

To constrain (i.e., minimize) the possibilities, we inspect all calls to a given function, say P, and see which nonlocal variables are actually passed to it as arguments. Of course, P may pass one of its formal parameters as an argument of a function call that P makes. So, an occurrence of a function call can introduce new possible definitions of nonlocal variables in three ways:

1. Those that it affects through possible definitions,
2. Those passed to it by reference, where the corresponding formal has a possible definition in the function. (Possible uses can be passed by value.)
3. Those affected via (1) and (2) in embedded calls within the function.

A cautious estimate of the alias set for any by-reference formal of a function is given by the reflexive, transitive closure of

{ <X,Y> : On some function call, X is the actual parameter that corresponds to the formal parameter Y }.

Note that parameter Y of function P is a different variable from parameter Y of function Q.

To establish a more precise estimate of possible definitions that occur due to calls to functions, let `PossDef[P]` denote the set of all nonlocals and formals having possible definitions in P. To calculate `PossDef[P]`:

```
for ( each function P ) {
  PossDef[P] = empty;
  for ( each variable X ) {
    if ( X is or formal parameter or a nonlocal defined in P ) {
      PossDef[P] += { X };
    }
  }
}
until ( stability ) {
  for ( each function P ) {
    for ( each function-call "Q(X1,...Xn)" in P ) {
      for ( int i = 1; i <= n; ++i ) {
        PossDef[P] += PossDef[Q];
```

```
          if ( i-th formal of Q is by-reference and in PossDef[Q] ){
            PossDef[P] += { Xi };
          }
        }
      }
    }
  }
```

A similar analysis must be done for *possible uses* introduced by function calls.

Note that a function call introduces new occurrences of all expressions involving nonlocals that are available at the end of the function. At each call, one can replace formals by actuals and possibly get occurrences of still more expressions. Doing so within a function can enlarge the set of available expressions at its end, which introduces another opportunity for code improvement, requiring yet another dataflow analysis.

Bibliography

[AH] Aho, Sethi, Ullman, **Compilers: Principles, Techniques, and Tools**, Addison-Weseley.

[BRE] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM* **22** (1979) 251 - 256.

[CH] Chow, F.C., A Portable Machine-independent Global Optimizer - Design and Measurements, CSL T.R. 83-254, Computer Systems Laboratory, Stanford University, $8.25.

[CHA] Chaitin, G. J. [1982]. “Register allocation and spilling via graph coloring,” ACM SIGPLAN Notices 17:6, 201-207.

[XX] *Pointer induced aliasing — a taxonomy* in the Jan 1991 **Principles of Programming Languages Conference**.

[KRS] Jens Knoop, Oliver Ruthing, and Bernhard Steffen, *Optimal Code Motion: Theory and Practice* in **ACM Transactions on Programming Languages and Systems**.

[KRS2] Jens Knoop, Oliver Ruthing, and Bernhard Steffen, *The Power of Assignment Motion* in **SIGPLAN Notices**, June 1995.

[KOR] S. M. Korman. The graph coloring problem. In: N. Christophides, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pp 211-235 (Wiley, New York, 1979).

Appendix A

Intermediate-Code Details

Give format for symbol-table records.

A program in intermediate format consists of a symbol table and a list of instructions, where each *instruction* is an *occurrence* of an *instruction form*, represented via a reference (or pointer) to that form. (We organize the instruction forms occurring in the program into a set through which we can search and iterate.)

A form's opcode field contains the operation's index within an enumerated type. Its `destination`, `source`, `source2`, `source3`, and `place` fields point to symbol-table records (descriptors) for variables. Depending on the opcode, certain of those pointer fields may be null. For example, arithmetic instructions don't have a `place`, and branches don't have a `destination`. Also, unary arithmetic operations don't have a second or third source variable.

Our machine model assumes that all operations take place on data that is in registers. We can take the address of any register and store it in a pointer register. Pointer registers can also point to individual elements within aggregates (arrays). Load and store instructions copy such elements into and out of registers of the corresponding scalar types.

The following interpreter code specifies the instructions set and the semantics of each op-code. During interpretation, the *value* attribute of a register variable always holds the variable's current value, which is of type `Value`.

For simplicity, we've assumed four-byte int's, floats, and pointers.

```
template< class T0, T1, T2, T3 >
class Form {
  Variable<T0>* dest;
  Variable<T1>* src;
  Variable<T2>* src2;
  Variable<T3>* src3;
  virtual void exec() = 0;
};

template< class T0, T1, T2, T0 op(T1,T2) >
```

```
class BinOp : Form<T0,T1,T2,void> {
  exec() { dest.val() = op( src.val(), src2.val() ); }
};

template< class T0, T1, T0 op(T1) >
class UnOp : Form<T0,T1,void,void> {
  exec() { dest.val() = op( src.val() ); }
};

BinOp< T0,T1,T2, operator-+ > add<T0,T1,T2>;
BinOp< T0,T1,T2, operator-- > sub<T0,T1,T2>;
BinOp< T0,T1,T2, operator-* > mul<T0,T1,T2>;
BinOp< T0,T1,T2, operator-/ > div<T0,T1,T2>;
BinOp< T0,T1,T2, operator-% > mod<T0,T1,T2>;
BinOp< T0,T1,T2, operator-& > and<T0,T1,T2>;
BinOp< T0,T1,T2, operator-| > or<T0,T1,T2>;
UnOp < T0,T1 operator-! > not<T0,T1>;
UnOp < T0,T1 operator-- > neg<T0,T1>;
UnOp < T0,T1 operator-& > addr<T0,T1>;
RelOp< bool,T1,T2, operator-== > eq<T1,T2>;
RelOp< bool,T1,T2, operator-!= > neq<T1,T2>;
RelOp< bool,T1,T2, operator->  > gt<T1,T2>;
RelOp< bool,T1,T2, operator-<  > lt<T1,T2>;
RelOp< bool,T1,T2, operator->= > ge<T1,T2>;
RelOp< bool,T1,T2, operator-<= > le<T1,T2>;
UnOp < T0,T0*, operator-* > load<T0>;
BinOp < T0,T0*,int, operator-[] > indexed_load<T0>;
UnOp < T0*,T0, operator-* > store<T0>;
BinOp < T0*,int,T0, operator-[] > indexed_store<T0>;
```

Notes:

1. The instructions are subject to certain semantic restrictions, which the front end observes as a matter of course.

2. Normal variables are called *symbolic registers* and may be thought of as the names of registers in a machine having an unlimited number of registers, each with a *type* and a modifiable run-time *value*. Initial values for symbolic registers are pre-set before execution begins.

3. An *aggregate* is a pointer register initialized to point to an array, say via the `new` operator. The only operation that is allowed on an aggregate is to initialize a pointer with its address. Of course, an aggregate may be a singleton, i.e., have only one component.

4. In real architectures, there is no ADDR instruction — registers don't have addresses that can be stored and used to access them indirectly, and the

address of a memory-mapped variable is obtained only by adding its offset to its base. To adapt our code to such architectures, any variable that is the operand of both an address-of instruction and some other instruction must be split into a symbolic register and a memory-mapped variable that are synchronized, i.e., any time one is (or might have been) modified, its value must be copied to and from the other via indexed load/store instructions having that variable's base and offset as source and source2, respectively. We will discuss the details later.

5. The semantics of indexed-load is visibly equivalent to:

   ```
   temp = src+src2;
   dest = *(int*)temp;
   ```

   which can be expressed as an add instruction followed by a non-indexed simple load instruction. Similarly, an indexed-store instruction can be replaced by an add and a simple store.

   Eliminating indexed loads and stores facilitates certain optimizations. For other optimizations, it is helpful to keep track of memory-mapped variables via symbol-table entries to see whether their current values are in real registers at various points during code generation.

6. Ultimately, we need more types than just pointer (char*), int, and float. For example, there are char, long, short, and double. Also, there are unsigned versions of char, long, short and int.

7. The front end creates symbol-table records for each literal constants that occurs and initializes its value attribute appropriately.

8. The `source` fields of goto and ifto instructions point to the descriptor (symbol-table record) of an instruction-pointer variable, whose value attribute contains the address of the target instruction. A goto instruction is also called an unconditional *jump* or *branch* instruction. It is said to be *direct* if its source is constant; otherwise, the branch instruction is said to be indirect.

9. An ifto instruction is said to be a conditional *jump* or *branch* instruction. Its source must be constant.

10. There are three special forms of goto called "call", "case" and "retn". As their names suggest, call and retn are used to call and return from functions (including calls via function pointers), while case is used to implement switch constructs. Most other occurrences of goto should have constant sources.

    Distinguishing these special uses of goto gives the back end some hints as to what to expect in the flow of control. For example, at an occurrence of a call instruction, the back end can expect that at run time control will return to this point as soon as more subsequent retn instructions have been executed than subsequent call instructions.

A.0.5 A readable format for intermediate code

Blend the rest of this stuff. Down to ...

For convenience, the assembly language for our three-address load/store architecture (machine) uses C syntax and semantics but does not include the entire C language. Let C-minor designate that subset of C that corresponds to the above operations. We will use it to specify and inspect the intermediate programs.

C-minor is a subset of C++ that is intended to represent the assembly language for a three-address load/store architecture with an arbitrarily large set of registers. It supports the data types int, void*, string, string*, Word and Word*. Word is defined to be `union{int T_int, void* T_voidstar, Word* T_Word}`. One can create an array of m Words via the expression new `Word[m]`, which returns a `Word*` pointer to that array.

The fundamental instructions in C-minor are of the form `x = y#z;` where `#` is any of the standard ALU operations available in C, stuff like `+`, `*`, `/`, `%`, etc., but not `&&` or `||` which involve flow-of-control games. Of course we allow `x = y` and `x = !y` and `x = -y` as C-minor instructions. We also allow load instructions `x = *y` and store instructions `*x = y` as mechanisms to copy information in and out of arrays. For flow-of-control capabilities, we allow:

- unconditional jumps, e.g., `goto L`,
- conditional jumps, e.g., `if ( x ) goto L`,
- label pseudo-instructions, e.g., `L:`,
- label-load instructions, e.g., `p = && L`, where `p` is a `void*` pointer,
- indirect jumps, e.g, `goto* p`, where `p` is a `void*` pointer.

Indirect jumps via label-loaded `void*` pointers only works for GCC but can be faked less conveniently in standard C. Finally, calls to standard library functions are allowed.

A C-minor program is a list of C-minor declarations, with optional initializers, followed by C-minor instructions, including declarations without initializers. By prepending and appending appropriate stuff, it will be a legitimate C++ program.

... here. This stuff was pulled from the CS182 assignment.

```
program:
      declaration-list instruction-list

declaration:
      qualifier-list type variable ;
      qualifier-list type variable = simple-expression ;
      qualifier-list type variable = literal ;

instruction:
      label :  instruction
      variable = simple-expression ;
      * ( type ) variable = variable ;
```

```
* ( type * ) ( variable + variable ) = variable ;
if ( variable ) goto label ;
goto label ;
call label ;
goto* variable ;
case variable ;
return ;
exit(0) ;
cout << string-literal ;
;
```

simple-expression:

```
variable binop variable
unop variable
* ( type * ) ( variable + variable )
variable
```

type:

```
float
double
int
short
long
char
Input
Output
```

qualifier:

```
volatile
unsigned
static
extern
```

qualifier-list:

```
qualifier qualifier-list
ϵ
```

unop:

```
!
-
*(type*)
~
(char*)&
(char*)&&
```

binop:

```
+
-
*
/
%
&
|
^
==
!=
<
<=
>
>=
>>
<<
```

Allow for comments.

1. The sources for the simple expressions in initializers must be previously declared and initialized.

2. All lexemes are separated by white space.

3. Labels and variables are simply C identifiers.

4. Identifiers, lists, expressions and string literals are as in C.

5. An instruction of the form *identifier* `=` *identifier* assembles into a simple copy instruction when both identifiers are of the same type. Otherwise, it assembles into a conversion instruction that converts the values of the right-hand type into a corresponding value of the left-hand type, e.g., an I2F instruction. Some conversion can be assembled into a simple COPY instruction, e.g., the conversion from `unsigned short` to `unsigned int`.

6. The result of all other mixed-type operations is undefined, and it is expected that the front end will not generate them.

7. All pointers are assumed to be of type `char*`, hence the need for built-in casts on load/store statements.

8. We adopt `goto*` and unary `&&` from gcc. A `void*` variable `p` can be set to point to a label `L` via the assignment expression `p = && L`,[1] after which, `goto* p` behaves just like `goto L`.

9. In C/C++, the qualifier `volatile` means that:

[1] The statement `p = && L` assembles into a simple COPY instruction. In this context, think of `&&` as a label-to-pointer conversion operator.

- This variable might be an output register, so its value at sequence points is part of the behavior of this program. In particular, this variable is never dead.
- This variable might be an input register, i.e., its value might change spontaneously between sequence points[2] — so, before using its value, fetch a copy that is more recent than the most recent sequence point.

An instruction or form is said to be *volatile* if and only if one of its operands is volatile.

10. For diagnostic purposes, we introduce two special types, `Input` and `Output`, which will designate input registers and output registers, respectively. Such variables should occur only as operands of COPY instructions. Such instructions are never available, and such variables are never dead. When a COPY instruction is executed, if the destination is an `Output` variable, its newly assigned value also gets written to the standard output. Similarly, if the source is an `Input` variable, its new value will be read from the standard input. In all other respects these types behave like integers.

    Statements of the form `cout << string-literal` are translated into a COUT instructions whose source is a pointer to a symbol-table record whose value field points to a copy of the string literal.

11. To call a function, we use the following idiom:

```
void* link;
...
link = && L1;
goto funct;
L1:;
...
...
funct:;
// body of funct
...
...
return;
...
```

    where `return` assembles into a goto-instruction with `link` as its source.

A.0.6 Exercises

- Write simple prolog and epilog files that can be prepended and appended to a C-minor file to turn it into an equivalent C++ program file that can be compiled under the g++ compiler.

[2]Sequence points occur before and after each maximal expression and before each function call.

- Write a simple assembler that takes a program in C-minor and produces the corresponding symbol table and intermediate instructions.
- Implement an interpreter for intermediate code.
- Write a decompiler that produces a C-minor source listing of a program from its symbol table and intermediate code. The current value of each variable should be shown in the initializer portion of its declaration in this listing.
- Combine the above into an interactive system for exercising and observing the results of various optimization routines. The user should have several options:

  1. Load a particular C-minor program, which gets assembled into a symbol table and intermediate instructions.
  2. Analyze and/or transform that program via any of the several transformation that will be specified later.
  3. Display (or dump to a specified file) in C-minor format the results of those analyses and/or transformations. Code listing should show the basic blocks[3] and, as comments, the current information associated with each block, e.g., a list of definitions that reach its entrance. Subsequent assignments will require several analyses and optimizations on the code. It is useful to be able to show the original form of the program alongside the transformed version, so that the two versions can easily be compared.
  4. Interpret the program either in its original form or its transformed form.

[3]The concept of a *basic block* will be explained later.

Appendix B

Algorithms for Transitive Closure

A directed graph G = (V,E) can be represented by its edge set E or by its *adjacency matrix*, a two dimensional boolean array whose rows and columns are indexed by vertices wherein a given entry is true if and only if the graph contains an edge from the entry's row's index to its column's index. Mathematically, an edge set is the same thing as a binary relation — two metaphors for one concept.

Often, directed graphs are sparse, i.e., include relatively few of their $|V|^2$ possible edges, in which cases their adjacency matrices are sparse, i.e., most of the entries are false. It is common to represent sparse matrices via row lists and/or column lists, which in the case of a directed graph each vertex's row represents its set of successors and its column represents its set of predecessors.

A directed graph is *transitive* if and only if there is an edge from vertex u to vertex v whenever there is an edge from some vertex w to v and from u to w, i.e., v is a direct successor of u whenever v is an indirect successor of u. Visibly, we can take a graph and add edges to all indirect successors, thereby, obtaining a transitive graph, called the *transitive closure* or the *reachability relation* of the original graph .

There are three common algorithms for determining the transitive closure of a graph.

- *Least-fixed-point, complexity* $|V|^4$. Make repeated passes through the edges. For each given edge add an edge from the given edge's tail to each successor of the given edge's head. Stability occurs by the time that the number of passes reaches the length of the longest acyclic path through the graph, which is bounded by the number of vertices in the graph, $|V|$. Each pass visits each edge, of which there are up to $|V|^2$. And at each edge we may need to added up to $|V|$ edges. So we get an upper bound of $|V|^4$. In practice, the number of passes is usually very small.

- *Dynamic programming (Floyd-Warshall), complexity* $|V|^3$. Suppose that the vertices are ordered and you know which vertices are connected via paths through the first $k-1$ vertices. Obviously, finding which vertices are connected via paths through the first k vertices requires only $|V|^2$ steps. So, by induction, the transitive closure can be computed in $|V|^3$ steps. (This can easily be generalized to find the shortest path between vertices. We can allowed weighted edges, even with negative weights but, obviously, no negative cycles.)

```
for ( each Vertex v ) {
  for ( each Vertex x and each Vertex y ) {
    if ( adjacent[x][v] && adjacent[v][y] ) {
      adjacent[x][y] = true;
    }
  }
  assert(
    adjacent[x][y] iff there is a path from
    x to y via predecessors of v plus v itself.
  );
}
```

- *Depth-first traversal (Tarjan?), complexity* $|V| \times |E|$. Do a depth-first search through the graph, pushing each vertex onto a stack the first time it is visited, and propagating each vertex's reachability set back to its predecessors. Whenever a subtree that borders no stack entry lower than the subtree's root vertex v has been traversed, simply pop v and all stack entries above it — they are the members of v's strong-connectivity component. Their reachability sets should be the same as v's, which is now complete. Obviously, this algorithm performs a set operation per edge, i.e., $|V| \times |E|$ vertex operations. For sparse matrices, this can be much faster than $|V|^3$.

Appendix C

Optimization Project

Any project to implement various optimizations requires framework in which to test those routines. This framework must be capable of creating symbol tables and lists of intermediate instructions. It must also be able to display the optimized intermediate code for comparison with the original. Obviously we need to pick a notation for input and display of intermediate code, and we've chosen `C-minor`. (I recommend use of the RIF toolkit and some GUI builder to build and interface that allows side-by-side comparison of optimized code with the original.):

- The user should have the ability to select a C-minor program to load as intermediate code, and there should be virtual buttons to invoke various optimizations. The user should also be able to dump the optimized program (in C-minor form) to a specified file.
- Next, add the ability to break up the program into basic blocks and display that partitioning of the program in an easy-to-understand format.
- Next, add the ability to determined which instructions are available and which variables are fresh at the entrance of each basic block. Dually, one should be able to determined which instructions are busy and which variables are dead at the exit of each basic block.
- Next, add the capability to selectively invoke dead-code elimination and available code elimination.
- Next, add partial redundancy elimination and partial uselessness elimination.
- Next, add constant folding and constant propagation.

The entire system should be well tested and easy to use.

Appendix D

Considerations

These are half-baked notes from the author to himself.

D.1 General notes

1. Add SSA.
2. Determine and fold constants
3. Propagate copies
4. Rank the rankable variables, e.g., rank(z = x + y) is one place the maximum of rank(x) and rank(y).
5. Determine "possibilities," e.g., possibilities(z) = possibilities(x) + possibilities(y), i.e, set addition.
6. We can still restrict some not-well-founded variables, say to certain arrays or records. These may be a well-founded chair of what they point relative to. The key here is to keep track of both value and basis, i.e., what a given pointer points relative to. Again, well-foundedness is the key.
7. Add a section on analyses of variables:
   - constant
   - predictable
   - constrained

   Perhaps this should be done in section 8.2.5.
8. Add diagrams showing three kinds of intermediate code.
9. We can handle the problem of local variables by using the "spill-code" approach.

(a) Fields of the activation record hold spilled outer-level variables.

(b) In each routine there will be assigned registers. We don't need to assign them locals unless they are spilled, e.g., on a call.

10. Every field of every record type and every record variable should have a symbol table entry (symbolic register). (If such a type descriptor has a pointer field (e.g., `next`), we can do a possibility analysis on all assignments to this field over all records of this type, e.g., it might only get its values from `malloc`, implying it can only point into the heap.)

11. For the front end to handle all referencing code, it would be necessary for offsets to have their own symbol-table record: X=B[offset-X]. Here offset-X is a symbolic constant of type displacement. It remains undefined until memory allocation is performed.

12. Constants do not have memory locations, but they may ultimately be given some during code generation.

13. Values usually originate in registers but may get stored to memory locations, but note that there are initialized strings in memory — not every variable originates in a register.

14. Minimize the number of copy instructions. Successive passes can possibly reduce them. Also try simulated annealing.

15. In selecting variables to keep in memory at a given block, give preference to those not used in the block. Treat these like another register.

16. We must add adapter blocks to rearrange the variables along each edge in the flow graph. We are trying to minimize the amount of adapter code. Mostly this code will permute the contents of the registers, but there are also memory locations to deal with. If there is a free register, it is easy to implement cyclic permutations, and each permutations is just a bunch of disjoint cycles.

17. In the local code generator, note that we should cluster values known to have equal values at a given point. No value should ever be in more than one register.

D.2 Addresses

Technically, a pointer is any variable X appearing in a instruction of the form *X = Y or of the form Y = *X, where X may be either constant or variable.

Note that load and store instructions are really copy instructions. As such, they can be subjected to the same analyses that are used for ordinary copy propagations. But there are a couple of differences. In the statements *X = Y both X and Y are sources and *X is the destination — there may be many *possibilities.* Similarly, in Y = *X the destination is Y with X as source and *X as a possible source. Any assignment to X or Y or one of the possibilities for *X will kill either of the above copy statements.

At register allocation time, we have to get down to the fact that we are dealing with only a finite number of registers, and then we will often have to store variables into main memory so that the corresponding register can be reused for another variable that is currently live and likely to be used sooner. For these purposes, we set aside a special location in main memory for each variable. Such locations are usually specified by expressions of the form P+D where P is a pointer variable and D is a displacement constant (the offset of the variable) that gets defined during memory allocation. (Note that P[D] is *(P+D) and is called "the item at P+D".)

An *address expression* is an expression of the form P $\{\pm$ D$\}$, where P is a pointer and D is a displacement. Each of these may be either constant or variable.

D.2.1 Keeping Track of Addresses

One can keep track of the fully determined pointers (those having only one possibility) at any time by arranging them in equivalence classes called *strips* of pointers differing by a constant. Each is given a relative offset in the strip.

We don't care where the origin is — we need to know the distance between any two pointers — absolute values of offsets are irrelevant.

Whenever a pointer is assigned an address with a variable displacement, a new strip is created with the new pointer as its only member and the variable as an interstrip *adjustment term* between the new strip (the child strip) and the strip of the base of the address (the parent strip). At the same time, the pointer is deleted from its old strip.

Note that constant folding may turn these variable adjustments into constants thereby collapsing two strips into one.

An assignment to an adjustment variable kills it. Up to that time, adding the same displacement to any pointer in the parent strip gives an offset in the child strip and subtracting it from any pointer into the child produces an offset in the parent. A strip is never deallocated until its adjustment is killed. This is important because it allows such constants to be treated the same as constants that never get assigned a new value.

The strips are compile time representations of virtual segments. Each is identified by a globally unique (never reused) identification number. When a statement of the form X = P $\pm$ D occurs, where D is constant, a descriptor is created for the offset corresponding to the value of P plus/minus the value of D within the strip of P. A reference to this descriptor is entered into the symbol table record for X.

Initially, each pointer is at location zero in its own strip. There are a few absolute strips that remain fixed throughout the program. These correspond to the fundamental segments of the program's address space (e.g., code, stack, and data) having base pointers that are never assigned a value by the running program. These are the fundamental strips and others are relative to them, hereditarily. But this happens automatically, we do not need to know about them.

D.2.2 Pointer Possibilities

Things get more complicated when we add possibilities. Then pointers have several possible strips, but not all possibilities are compatible. (Also, don't forget that there can be pointers to pointers.)

I think that the only thing that we need to do is make sure that every strip gets an identity that never gets reused. The possibilities for any pointer are a bunch of offsets in these various strips. Every strip/offset pair that gets used gets a descriptor. Possibility sets are sets of pointers to these. They are deallocated when the strip gets deallocated.

Each assignment to a pointer reduces its possibilities to a singleton — either a new strip with itself as the only resident or an offset in the strip corresponding to the base of the assigned expression.

(* We need to worry about the case of loops. In such cases there can be circular dependencies between pointer values. These can be determined by looking back through chains of reaching definitions. These definitions can be turned into a context free grammar that produces all of the possibilities for the value of a particular pointer at a particular point. *)

D.3 Notes on the intermediate language

1. Note that one of the places a pointer can point is into the heap, e.g., when it is set up via `new` or `malloc`. At this point it is at a variable offset from the base of the heap.

2. Labels are pointers to objects of type instruction. They are undefined during optimization, but they have pointers to instructions that allow them to be backpatched correctly during code generation. We assume for the time being that all labels are constants.

3. We introduce a new type called "displacement" (i.e., offset). The destination of a statement of the form X= stuff will be a

   - displacement if and only if stuff is
     (a) y, where y is a displacement;
     (b) $*$y, where y is a pointer to a displacement;
     (c) i$*$y, where y is a displacement, and i is an integer;
     (d) y+z or z-y, where y and z are displacements;
     (e) y-z, where y and z are pointers.
   - pointer if and only if stuff is
     (a) y, where y is a pointer;
     (b) $*$y, where y is a pointer to a pointer;

   (c) &y, where y is a variable;
   (d) y+z or y-z, where y is a pointer, and z is a displacement.

4. The copy statement ∗x=y has x and y as sources and anything that x can point to at this point in the program as "possible" destination. The statement x=∗y has x as its destination and y as a source. Also, it has anything y can point to at this point in the program as possible source. We try to propagate these out of existence. Note that an assignment to x or y or to any possible alias of ∗x will kill both of these instructions.

5. The statement x=&y means that ∗x denotes y until x is redefined. Ultimately, all occurrences of &y will be replaced by expressions involving pointers and displacements. (Note that, while a use of ∗x will be a use of y, it will not be a use of the basis for y, since x points there absolutely.)

6. We omit indexed expressions via the following conventions:

   - x=y[z] becomes t=y+z; x=∗t and
   - x[y]=z becomes t=x+y; ∗t=x.

   Here t is a temporary variable of appropriate type.

7. GOTOXs are branches. At the time of the branch we do an assignment to a label-valued variable, the return link. The RETURN is a branch with a variable destination.

8. In reality, the possibility analysis for label-valued pointer variables is not that difficult. The return link for a procedure can only point to the calls for that procedure. The label-valued variable for a switch statement can only point to its various cases.

9. A pointer can point only to memory-based variables, and &Y makes sense only if Y is memory based i.e. is an address expression. Expressions of the form `&Y` must often have their value computed at run time.

10. Every expression of the form P+D, where P is a pointer and D is a displacement constant, deserves a symbol-table record. There can't be that many displacement constants. This will at most double the size of the symbol table. But what about statements of the form P=P+D? Visibly, P+D does not represent a variable here? Whenever P is modified, this constitutes as definition of all variables based on it. They get new values at such points. So if we are careful to invalidate this variable P after we record that P points to it, thus immediately removing it from the possibility set for P, everything will be O.K.

    Note that any time we modify the basis for a variable we should delete it from the possibility sets for all pointers.

11. Consider P points to new record; Q = P; P = whatever. Now what are the possibilities for Q?

12. In every expression of the form $*x$, the x must be a pointer — either constant or variable.

13. Pointed-to variables (i.e., address expressions) serve as an intermediary between offsets in segments and the pointer. These are useful for keeping track of what the pointer can point to. But maybe they (pointed-to variables) should be automatically inserted, since they aren't always there.

14. Also, if p is a pointer to a record, we set aside symbol table records for each field of *p. But, in the case of linked lists, one of those fields, usually one named "next", will be another pointer to a record of the same type. We can fill the symbol table via recursion if we don't cut this process off somewhere.

15. Every expression of the form **base + const** gets a symbol table record. It is a pointer. We add every constant-offset expression to the symbol table as well (i.e., give each a symbolic register). That would take care of certain array elements, as well as fields of records.

    - In our intermediate code, we have replaced expressions of the form $\cdots X[Y] \cdots$ with constructions of the form

      $$\begin{gathered} t = X + Y \\ \cdots * t \cdots \end{gathered}$$

      where t is a special temporary globally dedicated to holding the value of X + Y.
    - Convention: &X is an abbreviation for $\text{base}_X + \text{offset}_X$.
    - Perhaps X should be represented by X.base[X.offset].

16. If p is pointing into a structure (array) and p's base changes ... it now points somewhere else! (Huh?)

17. There are two obvious paradigms for our intermediate machine:

    - Load/store with three-address code
    - Memory-to-memory with three-address code

    An attractive blend would be load/store with a home-address map and automatic write-through. The home address of each register would be updated whenever the register is and *vice-versa*[1] (thereby, propagating values to aliases, for instance). These addresses consist of a base register plus a (possibly not yet defined) constant (register) offset.

    This model allows us to postpone the introduction of spill code to propagate values to aliases. This is important, because it could be difficult to make the spill code go away when the reasons for it are removed via optimization. For instance:

[1] It isn't so clear how this *vice-versa* part could be implemented in a simulator for this machine.

```
   P = &X
q: X = Y
       .
       .
   T = &X
  *T = X
       .
       .
   Z = *P
```

could have an unwanted residue if q got trivialized.

D.4 Special notes

- Certain pointers and pointer fields of certain record types are constrained by the program to always point into the heap.
- Perhaps there should be a type like "pointer within some particular segment or array."
- Note that **P is simply a P-based pointer P[0][0].
- If P gets aligned with Q then *P is a subsegment of *Q, even if the alignment is a variable, so simply recording relative offsets is insufficient — we need to know who came first. That way we get a tree.
- You need to search this tree for aliases, but with each node we keep a size, and after reaching up the tree (toward the root), we see ever increasing (nested upward) webs, but searching back down among cousins, most will be disjoint from us, and this disjointedness can be easily determined. At disjointedness we can stop a downward search.
- There is a nesting structure to data-structure allocation. But there is a problem as well — we can't distinguish between pointer to an array and a pointer to its first member (in say a parameter) in C.
- An assignment to P[V] is a possible definition of all P-based variables (including those lacking symbol-table records, i.e., the implicit ones).
- Similarly for uses of P[V].
- We need to hereditarily align various pointers:

```
      P := Q+C
 =>   P[0] aligns with Q[C]
 =>   P aligns with Q at offset C
 =>   P aligns at offset C with every Q-aligned pointer
      and every P-based pointer aligns at offset 0 with
```

```
its Q-based counterpart, and this continues
hereditarily to whatever depth is indicated
by the symbol table
```

- In aliasing the item at P+D to the item at Q+DD, corresponding P-based and Q-based items must be aliased: the item at P+(D+C) must be aliased to the item at Q+(DD+C) for every constant C for which such items exist. Also, when these items are pointers, we must (recursively) alias P[D+C][0] to Q[DD+C][0]. Also, we must alias corresponding P-based items to P-based and R-based items if any Q-based item has been aliased to an R-based item.

Appendix E

Old approaches to aliasing

This appendix contains some tedious obsolete material that needs to be rethought for how it fits with the current approaches.

E.0.1 Effects of pointers

At any given point in the program, let S denote

{ (P,A) : P possibly points to the addresses designated by constant A }.

Let S_P denote { A : (P,A) is in S }. Note that

- If P is a constant then S_P is {P} at all times.
- If P is a variable then S_P is {null} initially.
- In general, execution of a instruction P = Q causes S_P to take on the same value as S_Q.

At the entry to a block, S_P is the union of its values at the exits of this block's immediate predecessors.

The effect on S_P of executing a block B is to set S_P to $S_{xfer[P]}$, where $xfer$[P] is computed by

```
for each pointer P do
   xfer[P] = P
od;
for each instruction  q  in  B  from first to last do
   if  q  is of the form  "P = Q" then
      xfer[P] = xfer[Q]
   fi
od
```

The analysis of possibly values for pointers proceeds just as the computation of reaching definitions. We let `CLOSE[B]` denote the value of S at the entry to block `B` which is the union of the corresponding values for each of its immediate prdecessors. We let `PASS[B]` denote its value at the exit of B:

$$\text{PASS[B]} = \{\ (\text{P,A}) : (xfer[\text{P}],\text{A}) \text{ is in CLOSE[B] }\}$$

To determine the set of possibilities for P, we consider how a specific statement would change the possibility set

$$\text{S} = \{\ (\text{P,X}) : \text{P is a pointer, X is a variable, and possibly P points to X }\}.$$

Then we have:

1. P = &X + C (where '+C' is in case X is an array):
   S = S – { (P,B) : any B } + {(P,X)}
2. P = Q + C (where '+C' is in case *Q is an array):
   S = S – { (P,B) : any B } +
   { (P,B) : (Q,B) in S and, in the case '+C', B is an array }
3. other assignments to P:
   S = S – { (P,B) : any B }
4. other statements:
   S = S

The dataflow analysis is the same as for reaching definitions. CLOSE[B] = {} initially. PASS[B] can be calculated from CLOSE[B] by (1), (2), and (3). By knowing the final CLOSE[B], we know the possibilities at any statement.

E.0.2 Determining possible aliases

It is safe to assume that everything is a possible alias for every thing else, but that assumption has a devastating effect on our ability to optimize. We compute a less cautious upper-bound on the set of possible aliases — a symmetric, reflexive, but nontransitive relation, denoted $\doteq$. That the relation should be nontransitive, is apparent when one considers two blocks having a common successor where the statement P = &X in one and P = &Y in the other. At the beginning of the common successor, *P is a possible alias for both X and Y, but they are not possible aliases of each other (in the usual case).

We may think of $\doteq$ as a two-dimensional bit array. Because $\doteq$ is

- reflexive, i.e., there are ones on the diagonal,
- symmetric, i.e., the row and column corresponding to a given symbol in the symbol-table are equal. For the pointer P we let S_P denote their common value.

The effect of a definition of P on S_P varies according to what is assigned to P:

- Case malloc(): {P}
- Case &X: {P, &X}
- Case Q: $\{P\} \cup A[Q]$
- Case Q+C, where C is a displacement constant: $\{P\} \cup \{$ exp + C : exp is in $S_Q\}$
- Case Q+V, where V is a displacement variable: $\{P\} \cup \{$ exp + ? : exp is in $S_Q\}$
- Case *Q: everything reasonable.

Rules: any displacement "+?" is "?", and we add constants as they occur. It is left as an exercise to construct a mechanism to take care of undefined constants.

The definition of reasonable prohibits possibilities R where *R is of a type different from *P. Also, it prohibits possibilities of the form &X where &X has never been assigned to a pointer.

If P and Q are possibly equal, then *P and *Q are possible aliases, and so are P→ D and Q → D for every displacement D.

If P and &X are possibly equal, then *P and X are possible aliases, and so are P → D and X[D] for every displacement D.

Morgue

These are obsolete presentations.

.1 Dominance

To find which blocks dominate which, we do a downward, forward analysis:

```
for ( each Block B ) {
   B.dom = ( B is an entry block ? singleton(B) : ALL );
}
for ( bool change = true; occurred(change); ) {
   for ( each Block B ) {
      for ( each immediate predecessor P of B ) {
         if ( B.dom - singleton(B) not included in P.dom ) {
            change = true;
            B.dom *= P.dom;
         }
      }
      B.dom += singleton( B );
   }
}
```

Use a standard closure computation.

CLAIM: For every block B, B.dom as calculated by this algorithm is the set of all dominators of B.

Align.

PROOF: Notice that a block dominates B if and only if it is B or it dominates all predecessors of B that are not dominated by B.

Let the *depth* of a block B be the length of the longest acyclic path from any entry block to B. Clearly, the depth of B is one greater than that of its deepest predecessor that it doesn't dominate.

Suppose that after depth-of-B passes, the `dom` set of each block of lesser depth than B is exactly its set of dominators — all predecessors of B not dominated by B will have correct `dom` sets after that pass. On the next pass,

$$\begin{aligned}
\text{B.dom} &\subseteq \{\text{B}\} + \bigcap \{ \text{ P.dom } \mid \text{ P is an immediate predecessor of B } \} \\
&\subseteq \{\text{B}\} + \bigcap \{ \text{ P.dom } \mid \text{ P is an immediate predecessor of B not dominated by B } \} \\
&= \{ \text{ B}' \mid \text{B}' \text{ dominates B } \}.
\end{aligned}$$

Conversely, it is easy to show by induction that, after every pass of the algorithm, each block's `dom` set includes all that block's dominators. So, the final value of B.dom is exactly the set of dominators of B. Q.E.D.

.2 Reachable uses and reaching definitions

For global (i.e., interblock) analysis, we are interested in which definitions reach the entry of each block. From this information, it is easy to determine via local (i.e., intrablock) analysis which global definitions reach any point in the program, specifically, by checking which reach its entry and which of those get killed in the block prior to that point.

The following algorithm is based on two facts:

1. The definitions reaching the entry of a given block are those that reach the exit of at least one of its predecessors.

2. The definitions reaching the exit of a given block are those that occur in the block and are preserved through the remainder of that block plus those that reach the block's entry and are preserved throughout the block.

At any time during execution of the following algorithm, `B.close` denotes the set of all global definitions that (so far) have been determined to reach the entry of B.

As a practical matter, we often represent subsets of a given collection by bit vectors (one-dimensional boolean arrays) indexed by the members of that collection. We need standard enumeration of the collection. Lacking any other mechanism, we can simply generate a list that contains the members of the collection without repetitions. We then represent subsets of that collection by bit vectors of the same length as that list. The i-th member of the list is in the set represented by a given vector if and only if the vector's i-th bit is `true`.

In the case at hand, the bit vectors represent sets of global definitions, i.e., there is a bit in these vectors for each <X,B> such that B contains a definition of X, which is not larger than the number of blocks time the number of variables in the program.

```
ALL denotes the set of all global defs.

for ( each block  B ) {    // Initialize B.
   B.gen  is a subset of ALL: namely, all global defs involving B,
         i.e., B.gen[ <X,B'> ]   = true  iff B' is B;
   B.kill is a subset of ALL: namely all global defs killed in B,
         i.e., B.kill[ <X,B'> ]  = true  iff X is defined in B;
   B.close is a subset of ALL: namely, the empty set,
         i.e., B.close[ <X,B'> ] = false for all X and B';
}

while ( changes occur ) {  // Make successive passes through the blocks,
   for ( each block B ) {  // propagating definitions to successors.
      for ( each immediate predecessor P of B ) {
         B.close += P.gen + ( P.close - P.kill );
      }
   }
}
```

Use STL style.

Whether to save or recompute `P.gen + (P.close - P.kill)` on each iteration depends on whether time or space is more critical. The following implementation avoids the need to do either.

```
inline bool occurred( bool& b ) {
   bool temp = b;
   b = false;
   return temp;
}

initialize as above;

for ( bool change = true; occurred(change); ) {
   for ( each block P ) {
      Set temp = P.gen + ( P.close - P.kill );
      for ( each immediate successor B of P ) {
         if ( B.close doesn't include  temp ) {
            change = true;
            B.close += temp;
         }
      }
   }
}
```

Claim: For every block B, the final value for B.close, as computed by the above loop is exactly the set R_B of all global definitions that reach the entry of B.

PROOF: It suffices to show, separately, that for each block B, $\mathrm{R}_B \supseteq$ B.close and that $\mathrm{R}_B \subseteq$ B.close.

To show that after each iteration of the loop, $\mathrm{R}_B \supseteq$ B.close for every block B, we suppose (toward a contradiction) that the current iteration is the first on which this hypothesis fails, i.e., that for the first time some B.close gets assigned a value that is not a subset of R_B. Then, at the start of this iteration, it hasn't yet failed, and

$\mathrm{R}_B \supseteq$ B.close, by initialization, for the first iteration
and, by hypothesis, for subsequent ones,

and

$\mathrm{R}_B \supseteq$ P.gen, by facts #1 and #2,

and

$\mathrm{R}_B \supseteq \mathrm{R}_P$ – P.kill, by facts #1 and #2,
$\supseteq$ P.close – P.kill, by hypothesis.

So

$\mathrm{R}_B \supseteq$ B.close + P.gen + (P.close – P.kill),
which is the new value of B.close after the current iteration.

A contradiction!

To show conversely that, at *stability* (i.e., when no more changes occur), $\mathrm{R}_B \subseteq$ B.close for every block B, suppose (toward a contradiction) that some global definition <X,B′> reaches a block B by a given X-preserving path of length n but is not in B.close after n passes (iterations through the outer loop). Also suppose that this path is the shortest path for which this happens for *any* <X,B′> and block B, and let P be the immediate predecessor of B on this path.

Case $n = 1$: Then B′ is P and <X,B′> $\in$ P.gen, which is a subset of B.close. A contradiction!

Case $n > 1$: Then <X,B′> $\notin$ P.gen but reaches B via a path through P, so it's not in P.kill. By hypothesis, <X,B′> $\in$ P.close after $n-1$ passes, since the corresponding path to P has length $n-1$. So, after n passes, <X,B′> $\in$ P.close – P.kill, which is a subseteq of B.close. Again, a contradiction!

In either case, we reach a contradiction, proving thereby that no such path can exist, i.e., after n passes all global definitions that reach B via paths of length n are in B.close.

Therefore, $\mathrm{R}_B \subseteq$ B.close at stability, which occurs by the time the number of passes reaches the length of the longest acyclic path in the graph. Of course, we can quit as soon as stability occurs, since subsequent passes have no effect. Q.E.D.

Note. One can speed up the algorithm by modifying the definition of `temp` so that P.gen is added to B.close only on the first iteration of the outer loop. It could instead be added just after initialization. Also, to avoid the need to store P.kill, one can subtract P.kill from P.close by simply resetting those bit positions corresponding to global definitions whose variables are defined by P.

.3 Availability analysis

.3.1 Availability

Analysis. The following algorithm is based on two facts:

1. The forms available at the entry of a given block are those available at the exit of one and all of its predecessors.
2. The forms available at the exit of a block are those having an occurrence in the block that reaches the block exit, plus those that are available at the block's entry and whose operands are not defined in the block.

To find all forms available at the beginning of each block:

```
// B.close denotes the set of all instruction forms that have
// (so far) been determined to be available at the entry to B.
// ALL denotes the set of all forms in the program.
for ( each block  B ) {
   B.close = ( B is an entry block ? empty : ALL );
   B.gen   = ALL forms having an occurrence in B without any
             subsequent definition of any of its operands in B;
   B.kill  = ALL forms having an operand defined in B;
}

for ( bool change = true; occurred(change); ) {
   for ( each block  P ) {
      Set temp = P.gen + (P.close - P.kill);
      for ( each immediate successor  B  of  P  ) {
         if ( temp  doesn't include  B.close  ) {
            change = true;
            B.close *= temp;
         }
      }
   }
}
```

On the first pass, we eliminate from B.close all forms killed and not regenerated in one of its immediate predecessors. On the second pass, we eliminate from B.close all that were removed from the "close" set of a predecessor and not regenerated in it, i.e., those killed in a predecessor of a predecessor and not regenerated in it or in the corresponding immediate predecessor. On the third pass, we eliminate everything killed in a third-level predecessor and not generated in it or along each path from it to B.

We keep going until stability is achieved, which will occur by the k-th iteration, where k is the number of blocks in the program, for if there is a path to some point along which a given form is unavailable then we can remove all cycles before and after the point where it is last killed on that path — each block can appear once before and once after that killing form.

.4 Reachable uses

Reachable uses. We also wish to compute for each variable X and each block B, the set of all uses of X reachable from the exit of B. The analysis is identical to that for reaching definitions except that it goes in the opposite direction, i.e., we replace the word "precedes" with "follows." Also, in this case, the bit vectors are indexed by global uses. We define B.kill to be the set of all such uses <X,B′> such that X is defined in B. Also, we define B.gen to be the set of all global uses involving B. After the process has stabilized, B.close will be the set of all uses reachable from the exit of B.

Exercise. What will the members of `temp` be in this case?