CS143 Compilers

Lecture 11

August 8, 2001

Code Generation

- Code generation is the actual synthesis of target code from the source program.
- This is one of the most difficult phases to implement, as the requirements are very stringent:
- Code generation employs many heuristics to achieve these goals, as the problem of generating optimal code is mathematically undecidable.
  - It must produce correct code
  - It must produce efficient code for the given target machine
  - It must run efficiently.

Input to the Code Generator

- The intermediate code previously generated from a translation scheme serves as input to a code generator, along with the symbol table, which is used to determine memory allocation of data.
- It is assumed that the source program has no lexical, syntactic or semantic errors.
- Type checking has already been performed, so any necessary conversions should already be specified.
- Values should be easily representable by the target language, such as values of basic types.

Output of the Code Generator

- The output of the code generator is a target program that performs the same operations as the source program.
- The target program can take different forms:
  - Absolute machine code, which can be executed immediately
  - Relocatable code, which must be linked and loaded, but enables separate compilation of subprograms
  - Assembly code, which relies on an assembler to produce machine code, making code generation simpler

Design Issues

- The following issues are usually to be considered in the design of any code generator, regardless of the actual target language:
  - Memory management
  - Instruction selection
  - Register allocation
  - Evaluation order

Memory Management

- Code generation includes mapping names in the intermediate code to addresses in memory, as well as the mapping of three-address statements to addresses of instructions.
- Allocation of memory for data uses information in the symbol table regarding the size of objects
- Assignment of instruction addresses takes into consideration the size of each three-address statement. To handle branches, a technique analogous to backpatching is used.

Instruction Selection

- For each operation performed by the intermediate code, equivalent instructions for the target program must be generated.
- If the target language does not support each data type uniformly, then special handling is required for any exceptions.
- To generate efficient code, one should consider the cost of each target instruction, especially if more than one translation is possible.
- Typically, naively generating code to implement each three-address statement yields poor code.

Register Allocation

- To maximize efficiency, it is important to make good use of registers, which are the most easily accessible storage locations to the CPU.
- First, a code generator must allocate registers, by selecting a set of variables that will reside in registers at any given point during execution.
- Second, specific registers must be assigned to these variables.
- Mathematically, optimal register assignment is an NP-complete problem. Restrictions imposed by the hardware or operating system complicate things further.

Choice of Evaluation Order

- Often, the order of operations within a program can be rearranged to some extent without changing the intended function of the underlying source program.
- In some cases, a rearrangement of instructions can result in fewer registers being used.
- Choosing the optimal ordering of instructions is another NP-complete problem.

Approaches to Code Generation

- Certainly, the most important criterion in designing a code generator is that it always produces correct code.
- As a result, an important secondary criterion is that it be simple to implement, test and maintain.
- Initially, we will present a straightforward code-generation method that tries to keep operands in registers for as long as possible. The code can then be improved using peephole optimization.
- Later, we will consider flow of control to achieve more optimal use of registers.

The Target Machine

- Design of an effective code generator requires thorough knowledge of the nuances of the target machine.
- While this criterion is of little help for a general discussion on code generation, a generic assembly-like language is sufficient to present the basic techniques.
- We will work with a language that uses n registers R_0, R_1, …, R_{n-1}, operations for manipulating data and performing operations such as MOV, ADD, and SUB, and a variety of addressing modes to facilitate data access.

Instruction Costs

- For simplicity, we will assume that each instruction has a cost of 1 unit, plus the cost of data access. This cost will be 0 for registers, and 1 for any other addressing mode.
- Aside from minimizing execution time, minimizing instruction length can be very helpful as well, since time to fetch an instruction from memory often exceeds the time required to execute it.

Run-Time Storage Management

- A code generator must output instructions that manage activation records for procedure calls.
- Such instructions must perform the following tasks:
  - allocating the space for the record
  - saving the state of the program
  - storing data in the record
  - invoking the procedure
  - obtaining retuned values
  - restoring the state of the program so that the caller can resume execution.
  - deallocating the activation record

Static Allocation

- If static allocation is used, run-time storage management is extremely simple, if inflexible.
- Invocation entails saving the return address in the procedure's static data area, and then branching to the beginning of the code for the procedure.
- Once the procedure finishes execution, it can simply obtain the return address from its own data and branch to that address, at which time execution can resume.
- In the case of the main procedure, control is instead transferred back to the operating system.

Stack Allocation

- By using relative address instead of static addresses, code for static allocation can easily be transformed into code for stack allocation.
- In this case, instead of storing a return address in a static location, it is stored in a location indicated by a special variable, a *stack pointer*, which consistenly points to the beginning of each activation record.
- Upon return, the return address is obtained via the stack pointer, which is then moved to point to the previous activation record.

Run-Time Addresses for Names

- During intermediate code generation, names for objects can be handled in two ways: associating names with symbol table locations, which is best for portability, or generating specific steps for accessing variables as part of the intermediate code, which can aid in optimization.
- Regardless of the approach used, names in the intermediate code must eventually be translated to addresses in the target code. Typically this includes using an offset from some as-yet-unknown base address, to be determined at runtime.

Basic Blocks and Flow Graphs

- To generate efficient target code, it is helpful to understand the structure of the intermediate code, recognizing each complex computation as a unit rather than a sequence of separate statements.
- To this end, one can construct a *flow graph* for the intermediate code. A flow graph consists of nodes representing computations, and edges representing the flow of control through the code.
- The nodes of a flow graph are called *basic blocks*. A basic block identifies a set of statements that can be treated as a unit for purposes of analyzing flow control.

Basic Blocks

- A basic block is defined to be a sequence of three-address statements in which the flow of control can only enter the sequence through the first statement and exit through the last statement, without possibility of branching into or out of the sequence.
- Identifying basic blocks amounts to finding *leaders*, the first statements of the basic blocks.
- The first statement is a leader, as is any statement that is the target of a branch, as well as any statement that immediately follows a branching statement.

Transformations on Basic Blocks

- Each basic block can be viewed as a “black box” that must produce certain output values, which are then used in subsequent statements. Such values are said to be *live* after exiting the block.
- Two basic blocks are said to be *equivalent* if they compute the same live values.
- Since only these values matter to the rest of the program, one can apply various transformations to the code within the block in order to produce the same outputs more efficiently.

Structure-Preserving Transformations

- One type of transformation that can be applied to a basic block is a *structure-preserving transformation*.
- Common structure-preserving transformations are:
  - Common subexpression elimination
  - Dead code elimination
  - Renaming of temporary variables
  - Interchange of independent statements

Algebraic Transformations

- Algebraic transformations are used to transform statements into mathematically equivalent statements that are more efficient.
- Some common algebraic transformations are:
  - Taking advantage of the identity properties of 1 or 0 to remove unnecessary operations
  - Converting exponentiation to more efficient multiplication, or even a shift in the case of a power of 2. These transformations are called *reductions in strength*.
  - Performing operations involving only constants. This is known as *constant folding*.

Flow Graphs

Once basic blocks have been identified, construction of a flow graph proceeds as follows:

- The nodes of the graph are the basic blocks
- There is a directed edge from a block B to another block C if there is a branch from the last statement of B to the first statement of C.
- There is also a directed edge from B to C if C immediately follows B in the code and the last statement of C is not an unconditional branch
- In either case, we say B is a *predecessor* of C, and C is a *successor* of B.

Representations of Basic Blocks

Some methods of representing basic blocks are:

- Representing each block using a record containing a pointer to the leader, the number of statements in the block, and lists of predecessors and successors
- Maintaining a linked list of statements within the block. This could be problematic if code is rearranged.
- Representing edges via pointers to blocks, rather than individual statements.

Loops

A *loop* in a flow graph is defined as a subset of basic blocks satisfying the following conditions:

- The subgraph consisting of these blocks and edges between them is *strongly connected*: there is a path from any block in the set to any other block in the set, using only edges between blocks in the set.
- There is a unique entry into the set of blocks from any block outside of the set.

A loop that contains no other loops is called an *inner loop*.

Next-Use Information

• To allocate registers efficiently, it is helpful to determine where a name in a three-address statement is used once its value has been assigned.

• Once a name's value is no longer needed, if it is stored in a register, the register can be assigned to another name.

• Furthermore, after the value is no longer needed, the temporary variable holding that value can be re-used.

Computing Next Uses

• Next-use information can be determined by scanning backwards through a basic block.

• Initially, all non-temporary variables are considered live upon exit from the block. If temporaries can be re-used across blocks, they must be considered live as well.

• When a statement x := y op z is encountered, x is marked as "not live" and "no next use" for preceding statements, and the next use of y and z is set to this statement. This information is recorded in the symbol table.

Storage for Temporary Names

• Next-use information can be very helpful in determining when temporary variables can be re-used.

• Specifically, the lifetime of temporary variables can be determined by computing next uses. When a temporary t_1 ceases to be live, it can be reused, thus replacing another temporary variable t_2 whose lifetime does not overlap with that of t_1.