CS143
Compilers

Lecture 8

July 30, 2001

Intermediate Code Generation

- Once a program is determined to be semantically valid, it is ready for translation into a target language.
- Translating into a generic intermediate language is more desirable, however, in order to facilitate translation into multiple target languages.
- Intermediate code can easily be optimized prior to translation into a target language, allowing the back end of the compiler to focus on platform-specific optimizations.

Intermediate Languages

- A syntax tree is one intermediate representation of a source program that we have examined.
- For intermediate code generation, we use a different notation, *three-address code*. This notation is well-suited for translation into a target language.
- We will proceed by building a translation scheme for the grammar, with semantic actions that will output the intermediate code.
- In many cases, the processes of lexical analysis, parsing, semantic analysis, and intermediate code generation can be completed in one pass through the source program.

Graphical Representations

- While a syntax tree is a natural graphical representation for the structure of a source program, a slightly modified graphical representation, a *dag* (directed acyclic graph) is more useful for intermediate code generation.
- The difference between a dag and a syntax tree is that in a dag, common subexpressions are identified, yielding a more compact representation and partially optimized code.
- A syntax-directed definition can easily be used to generate either representation.

Three-Address Code

- *Three-address code* is a sequence of statements of the form **x := y *op* z**, where **x**, **y**, and **z** are objects that hold values, and ***op*** is some operator. By translating source programs into such simple statements, target code can be more easily generated and optimized.
- This notation is a linearized form of a syntax tree or dag, in which each node in the graph is assigned a name that is used in a three-address statement.
- The name "three-address" comes from the fact that each statement contains three addresses, two to hold operands and one to hold the result.

Types of Three-Address Statements

- Three-address statements can be viewed as generic assembly code. Each name can be associated with a conceptual location in memory, and each operator with a particular instruction.
- Common three-address statements are:
  - Assignments of the form **x := y *op* z**, where ***op*** is a binary operator, or **x := *op* y** where ***op*** is a unary operator, or simply **x := y**, a *copy* statement.
  - An unconditional jump **goto L**, where **L** is the label of a three-address statement

Types of Three-Address Statements

- Conditional jumps of the form **`x op y goto L`**
- Array references of the form **`x[i]:=y`** or **`x:=y[i]`**
- Address and pointer assignments of the form
  `x := &y`, **`x := *y`** or **`*x := y`**
- Statements for implementing procedure calls:
  - **`param x`** for passing a parameter **`x`**
  - **`call p,n`** to call procedure **`p`** with **`n`** actual parameters
  - **`return y`** to return value **`y`** from the current procedure and pass control back to the called

Syntax-Directed Translation into Three-Address Code

- Three-address code can be generated by assigning names to the interior nodes of a syntax tree. These names then appear as operands to three-address statements.
- These statements rely on the attributes of the nodes of the syntax tree. These attributes may be, for example, an index into the symbol table for an identifier, a location in the constant pool of a literal's value, or even the label of another three-address statement, perhaps for purposes of implementing flow control.

Implementations of Three-Address Statements

- Three-address statements are typically implemented in one of three ways: *quadruples*, *triples* and *indirect triples*.
- Quadruples contain four fields, representing the two operands, the operator, and the resut.
- Triples are used to avoid entering temporary names into the symbol table, instead referring to such values by the position of the statement at which the value was computed.
- Indirect triples use pointers to triples as operands.

Comparison of Representations

- When translating three-address statements to target code, each name will be assigned some run-time memory location.
- If quadruples are used, the symbol table locations of each name are immediately accessible. Furthermore, moving statements around, as frequently occurs during optimization, does not cause difficulty, whereas with triples, moving a statement forces references to that statement's result to be renamed.
- Indirect triples pose no such problem. They require about as much space and are equally suitable for rearranging code as quadruples, but can save some space if temporary locations are re-used.

Declarations

- As declarations are examined during the analysis phase, storage can be laid out. For each variable name, a symbol table entry containing information about the variable's type and relative address can be created.
- Since source languages typically specify the number of bytes occupied by values of basic types, specific locations for each local name, relative to a base address, can be determined. If a specific machine is being targeted, issues such as alignment can also be considered.

Declarations in a Procedure

- Typically, variables declared within a single block, such as a procedure or block statement, are laid out in memory consecutively.
- When processing a block, we can use a variable, called *offset*, that is initially 0. When each variable is declared, space for the variable is assigned at the address *offset*, and the value of *offset* is then incremented by the amount of space occupied by the variable.

Keeping Track of Scope Information

- In a language that allows declarations within nested blocks, a stack of scopes is maintained. With each scope, we associate a separate symbol table that holds the names declared within that block.
- When a block begins, processing of declarations in the enclosing scope is temporarily suspended. The offset value for that scope is preserved on the stack until after the new block is processed.
- When a block is finished, the cumulative width of all entries in the corresponding symbol table is recorded, so that we know how much space to allocate for all local names during code generation.

Field Names in Records

- Records can be processed in a similar fashion as a block. Fields within the record can be stored consecutively in memory, and the total size of the record can be computed at compile time.
- When a variable is declared to be of a type that is a record, the width occupied by the variable can be obtained by looking up the type's symbol table entry.
- When a field of the record is accessed, the relative offset of the field within the record can be obtained from the field's symbol table entry.

Assignment Statements

- To generate intermediate code for an assignment statement, each reference to an identifier within the statement must be resolved to a variable within the source program, represented by some symbol table entry.
- For languages that support access to non-local names, such as Pascal, this resolution works by looking up a name in the symbol table for the current scope, then proceeding to the symbol table for the enclosing scope, and so on, until the name is found or the top-level scope has been searched.

Reusing Temporary Names

- In generating code for an expression, new temporary variables are created, used to hold results of subexpressions.
- Frequently, temporary variables may be reused, by taking into account the lifetime of each temporary.
- Temporary variables created for the purpose of storing the value associated with the non-terminal on the left side of a production only have a lifetime that extends to the point at which that non-terminal is used in another reduction.

Addressing Array Elements

- Typically, elements within an array are stored consecutively in memory.
- This storage allows references to individual array elements to be partially resolved at compile time.
- This resolution proceeds by generating code for multiplying the array index by the width and adding it to the base address of the array.
- For a multi-dimensional array reference, this technique can be used repeatedly for each dimension. The order in which dimensions are considered depends on whether arrays are stored in row-major or column-major form.

Type Conversions within Assignments

- During intermediate code generation, the compiler must often determine type conversions to be used during expression evaluation. A typical example is converting an integer to a floating-point number.
- Such a conversion can take the form of an intermediate expression, employing a unary operator whose sole task is to convert a value from one type to another. The result of this operation is used to continue evaluation of the original expression.

Boolean Expressions

- Boolean expressions play a central role in just about any source program. Not only are they used to obtain logical values, but also determine the flow of control through the source program as it is executed.
- Many languages allow more general types of operands to be used with boolean operators, requiring frequent type coercions.
- Often, complex boolean expressions can be evaluated very efficiently by taking into account values of subexpressions obtained at run-time.

Numerical Representation

- A typical method of representing boolean values is numeric. A few common conventions are:
  - Using 1 for true and 0 for false
  - Using any nonzero value for true and 0 for false
  - Using a nonnegative value for true and a negative value for false
- Using this representation, boolean expressions are evaluated in much the same way as arithmetic expressions. Results of relational expressions can be assigned using branching statements.

Short-Circuit Code

- Alternatively, boolean expressions can be translated without generating any code for the boolean operators and, or, and not.
- This method, called *short-circuit evaluation*, can efficiently evaluate a boolean expression by using flow control.
- The value of a subexpression is used to determine the next statement to be executed. If this value determines the value of the entire boolean expression, the remainder of the code can be skipped using a branching statement.

Flow-of-Control Statements

- Branching statements also play a key role in translating flow-control statements that depend on boolean expressions, such as if or while statements.
- Based on the value of a boolean expression, a conditional branching statement can be used to transfer control to the appropriate code that is to be executed. An unconditional branch can be used to implement iteration, or skipping code that is not to be executed because of the expression's value.
- Such code makes use of three-address statements that serve as labels as branch targets.

Control-Flow Translation

- Control-flow translation of boolean expressions entails assigning two labels to an expression, called *exits*. In addition to generating the code to evaluate the expression, statements are generated to branch to one of these exits, based on whether the value of the expression is determined to be **`true`** or **`false`**.
- By using short-circuit evaluation, such exits can be shared between subexpressions. For example, in an expression involving the or operator, the true exit for either operand is the same as the true exit for the entire expression.
- A negation is translated simply by interchanging the true and false exits.

Mixed-Mode Boolean Expressions

- Typically, expressions involve both boolean operators and arithmetic operators.
- To evaluate such expressions, control-flow translation can still be used for the boolean subexpressions.
- When a boolean value is used in an arithmetic expression, two three-address statements are generated, one for the case where the value is true, and the other for when it is false. These statements effectively perform the conversion of the boolean value to a numeric value. These statements are labeled as the true and false exists for the boolean subexpression.