CS143 Compilers

Lecture 9

August 1, 2001

Review: Type Expressions

- *Type expressions* lend a useful, universal notation for describing the types of language constructs and facilitating type checking.
- Type expressions can be constructed as follows:
  - All basic types supported by the language are type expressions.
  - Special basic types, indicating semantic errors or the absence of a value, are type expressions.
  - A *type constructor* applied to a type expression is a type expression. Common constructors include arrays, cartesian products, records, pointers, and functions.

Overloading of Functions and Operators

- An *overloaded* operator is an operator that has different meanings depending on the types of its operands. In most source languages, for instance, arithmetic operators are overloaded. However, target languages tend not to overload operators, as they contain machine-dependent instructions that require specific types.
- Whenever an overloaded operator is used, it must be *resolved* so that a unique operation is determined. This process is known as *operator identification*.

Set of Possible Types of Subexpressions

- In many cases, overloaded operators can be resolved simply by examining the types of the operands, but in some cases the type of an expression depends on its context. This yields a set of valid types that a given expression may have.
- By determining a unique type for an expression's context, we can narrow the set of possible types that the expression can have.
- This process requires heavy use of both synthesized and inherited attributes.

Polymorphic Functions

- A *polymorphic function* is a function that can accept arguments of varying types. Many operators in common source languages are polymorphic.
- While such functions can complicate type checking, they are highly desirable because they facilitate the implementation of abstract operations that act on data of varying types, such as determining the length of a list without caring about the types of the elements of the list.

Type Variables

- To facilitate type checking of polymorphic functions, *type variables* can be used. Such variables are essentially type expressions containing variables as well as fixed type expressions.
- Type variables aid in the process of *type inference*, which is the determination of the type of a language construct from the way in which it is used.
- Type variables can be used to ensure that the usage of a given construct is consistent throughout the source program.

Substitutions, Instances and Unification

• A natural question is, given two type variables, how do we know if they are compatible?

• This question is answered by the process of *unification*, which entails substituting the variables in each type expression with other type expressions in an attempt to attain structural equivalence.

• Unification proceeds by representing each type expression as a graph and attempting to "merge" the graphs by matching constructors, basic types, and variable substitutions.

Case Statements

• Translation of a case or switch statement entails the following:

– code to evaluate the selector expression

– code to execute the selected statements

– statements to branch to the appropriate (selected) statement based on the expression's value

• The last item is an *n*-way branch, where *n* is the number of cases, including the default case if there is one.

Implementing *n*-way Branches

• For a small number of cases, a sequence of conditional goto's is sufficient. Alternatively, a table of value-label pairs and a loop to iterate through the table is more efficient.

• For a larger number of cases, a hash table should be used instead.

• If all of the case values lie within a small range, then one can use a table can have rows associated with all values in the range, with the default label associated with all values without case labels. Then the appropriate label can simply be looked up.

Syntax-Directed Translation of Case Statements

• To determine the most efficient approach for a particular case statement, it is preferable to generate the code for all of the selected statements first, followed by the branching code.

• This organization allows the compiler to heuristically choose the optimal code for implementing the *n*-way branch.

• While this efficiency could also be gained by placing the tests at the beginning of the code for the case statement, this would force the compiler to delay generating code for the selected statements.

Backpatching

• When generating code involving branches, the targets of the branch are not always available on the first pass.

• To avoid making a second pass, we can generate branching statements with the arguments left unspecified, and keep track of such statements, filling in the targets when we know what they should be. This technique is known as *backpatching*.

• Backpatching entails the creation and combination of lists of statements, each associated with non-terminals in the translation scheme. As expressions or statements are reduced, statements in these lists are assigned a target label.

Procedure Calls

• The mechanics of a procedure call vary from language to language, due to differences in parameter passing, scoping, and other aspects.

• Regardless, procedure calls are translated in a fairly uniform manner. Basically, code to implement the procedure's *calling sequence* must be generated.

• Space must be allocated for the procedure's local data. This space is usually called an *activation record*.

Calling Sequences

- The arguments must be evaluated and made available to the procedure.
- A means of accessing nonlocal names must be provided.
- The state of the calling procedure must be saved so that it can be restored upon return.
- A jump is executed to the procedure's code.
- Upon return, a return value must be made accessible to the caller, the caller's state restored, and a jump made to the return address.

Run-Time Environments

- Before we translate our intermediate code to target code, we must be able to associate the objects in the source program with objects in the environment in which the corresponding target program will execute.
- In particular, provisions must be made to allocate space for data, provide access to this data, and process procedure calls as intended in the source program. Often, a run-time environment must be able to support different conventions from source languages for similar tasks, such as passing parameters to procedures.

Subdivisions of Run-Time Memory

- At run time, a target program is allocated a block of storage in which to run.
- This storage space may be divided to include the following:
  - target code
  - static data
  - a stack to hold blocks associated with procedure calls, called *activation records*
  - space for dynamically allocated blocks of memory, known as a *heap*.

Activation Records

- An *activation record* is a block of memory containing information relevant to a single invocation of a procedure.
- Activation records are typically maintained on a stack by the run-time system.
- When a procedure is called, an activation record is created and pushed onto the stack, and control is passed to the code for the procedure.
- Upon return, the activation record is popped and de-allocated, and control returns to the calling procedure.

Compile-Time Layout of Local Data

- Data is typically assigned at compile time, with each object in the source program assigned a block whose size is determined by the object's type.
- Each declaration of a local variable is assigned an *offset* value at compile time. The memory for this variable is allocated at *offset* bytes from a base address that is determined at run time.
- The actual location may be adjusted due to addressing constraints of the target machine, such as alignment considerations.

Storage-Allocation Strategies

- Different data areas are managed using different storage-allocation strategies.
- Static allocation is used to allocate static data such as global variables and code.
- Stack allocation is used to allocate activation records for procedures.
- Dynamic allocation is used to assign variable-length blocks of memory, the size and lifetime of which cannot be determined at compile time.

Static Allocation

- When using static allocation, addresses within the program's assigned memory are determined at compile time.
- This strategy allows values of local variables to be retained across procedure calls, since local names are consistently bound to the same location in memory.
- Some restrictions are:
  - Recursion is problematic, since different instances of local variables share the same memory location
  - Objects cannot be allocated dynamically.

Stack Allocation

- Using stack allocation, a block is allocated at run time, of size determined at compile time.
- Objects local to the block are allocated space within that block at offsets determined at compile time.
- Each block, upon allocation, is pushed onto a stack. A pointer is maintained to the previous block on the stack, to provide access to data within that block, and other blocks.
- The data is de-allocated when the block is popped off the stack.

Calling and Return Sequences

- When procedure calls are handled using stack allocation, with each block playing the role of an activation record, a compiler must generate segments of code to implement the invocation of, and return from, a procedure. These segments are *calling sequences* and *return sequences*.
- A calling sequence allocates an activation record and enters information into its fields, such as parameter values and return addresses. A return sequence restores the state of the machine so that the calling procedure can continue execution.

Placement of Data within an Activation Record

- As a rule of thumb, data whose size is fixed is best placed in the middle of an activation record.
- Actual parameters and a return value should be placed at the beginning (which is also the end of the caller's activation record), since the procedure may accept a variable number of arguments or return variable-length data, and the caller must know where to start placing this data, without necessarily knowing the size of the callee's activation record.
- Space for temporary variables is unknown until the last stages of compilation, since it may change during optimization.

Heap Allocation

- Heap allocation is necessary if values of names that are local to a procedure must persist beyond the deallocation of an activation record.
- Heap management can be difficult because much bookkeeping is required to keep track of blocks of arbitrary length being allocated and deallocated during execution, resulting in memory becoming fragmented.
- It is helpful to handle small blocks as a special case, maintaining a linked list of free blocks of a small size. The goal is to keep heap management time proportional to the amount of data being allocated, as that is often proportional to the amount of computational effort devoted to that data.

Parameter Passing

Different languages use different conventions for passing parameters, which must be considered during compilation:

- Call-by-value: the values of the parameters are copied to the activation record
- Call-by-reference: pointers to the actual parameter locations are copied to the activation record
- Copy-restore: a hybrid between call-by-value and call-by-reference
- Call-by-name: procedure bodies are textually substituted for the call, replacing formal parameter names with actual parameter names.