COP5621 Project Phase 4
Code Generator

Purpose:

This project is intended to give you experience in writing a code generator as well as bring together the various issues of code generation discussed in the text and in class.

Project Summary:

Your task is to write a *code generator*, the final phase of your compiler, that produces assembly code for the SPARC machine (oldiablo), given an augmented abstract syntax tree and symbol tables produced by the previous phases of your compiler. This phase will consist of assigning memory address for each variable used in the PASC program and translating subtrees of the abstract syntax tree (intermediate language representation) into sequences of assembly instructions that perform the same task.

Predefined Functions in PASC:

You should write your own predefined procedures and functions (in C). The calls in your generated assembly program to these predefined functions should follow the calling convention of CC (or GCC).

Run-time environment:

Recursion in PASC prevents the use of a static storage allocation strategy. However, the PASC language has no features that prevent the deallocation of activation records in a last-in-first-out manner. That is, activation records containing data local to an execution of a procedure, actual parameters, saved machine status, and other information needed to manage the activation of the procedure can be stored on a run-time stack. The exact size of an activation record can be determined at compile time. Global variables and temporaries can be stored in the data segment (not in the stack).

Code generation:

This is the only phase of your compiler which is machine dependent. You will be generating SPARC assembly code. The important/interesting issues in generating code for PASC are discussed in the following paragraphs. For further details on these issues, you should consult Chapters 7 and 8 of the text book. You can make the following simplifying assumptions:

1. Code is generated on a statement by statement basis without taking into account the context.
2. Separating the code generation into two logical phases, intermediate code generation as we discussed in class and a naive mapping of the three address code into SPARC assembly code, will greatly reduce your program size.
3. You can allocate as many temporaries as needed.
4. You may take advantage of any special instructions of the machine when choosing target instructions; however, this is not required.
5. You need not perform any fancy register allocation or peephole optimization on your target code.

Computing memory addresses: Since address information has not been computed and entered into the symbol table by earlier phases, the first task of the code generator is to compute the offsets of each variable name (both global and local); that is, the address of each local data object and formal

parameter within the activation record for the block in which they are declared. This can be done by initializing a variable *offset* at the start of each declaration section, and as each declaration is processed, the current value of offset is entered as an attributed of that symbol, and offset is then incremented by the total width of that data object depending on its type. The offsets of fields within records should be computed and stored as an attributed of the field name, typically relative to the start of the record. Globals should be implemented using static storage.

Call–by–value parameters will have a width dependent on the type of the parameter, whereas call–by–reference parameters will have a width equal to 1 word containing an address. The total activation record size of each procedure should be computed at this time and entered in the symbol table as an attributed of the procedure name.

The machine architecture must be taken into account when computing these widths. Offsets of locals and parameters can be implemented as offsets of parameters from the frame pointer. Thus, the computation of offsets of arguments and local variables can be done independently. This information will be used upon every reference to the data object in addition to being used in the allocation of storage for activation records.

Handling structured data types: Storage for an array, record, string, or more complex structured types built from these types and the basic types is allocated as a consecutive block of memory. Access to individual elements of an array or string is handled by generating an address calculation using the base address of the object, the index of the desired element, and the size of the elements. You are free to choose the layout of elements of an array in your implementation (e.g., row major or column major). Access to individual components of a record is done by performing an address calculation that adds the based address of the record and the offset of the desired component within the record. Access to more complex objects can be done by combining the above actions. See Chapter 8 for more details on these calculations.

Constant folding: For efficiency at run-time, all references to constant identifiers should be replaced with the corresponding constant. Also, when generating code for an expression in which all operands are constants known at compile time, the code generator should compute the value of the expression at compile time and avoid generating the expression evaluation. That is, all expressions with constant operands should be folded. You need not perform any global analysis to determine whether the value of an identifier is a constant. This constant folding needs only be performed using local context.

Simple Control Flow: Code for simple control statements, namely conditionals and loops in PASC, can be generated according to the semantics of Pascal using the compare, test, and branch instructions of the assembly language. Unique target labels will also have to be generated. Chapter 8 of the text gives examples of code generation for various simple control statements.

Function calls, prologues and epilogues: Function calls result in the generation of a calling sequence. Upon a call, an activation record for the callee must be set up and control must be transferred to the callee after saving the appropriate information in the activation record. For each procedure, the generated code sequence will consist of a prologue, the code for the statements of the procedure, and the epilogue. Typically, the prologue indicates the registers that should be saved upon a call to this procedure and allocates space on the stack for local variables, whereas the epilogue consists of restoring the saved machine status and returning control to the point in the caller immediately after the point of call.

Parameter passing: In order to correctly handle the formal parameters within the body of the callee, the symbol table entry for each formal parameter must include an attribute that indicates the parameter passing mode (value or reference). Structure objects passed by either call-by-value or call-by-reference are handled the same as simple objects.

On a procedure call, call-by-value parameters are handled by allocating the local storage for the size of the object in the activation record of the callee and then evaluating the actual parameter and initializing the local store within the callee with the value of the actual parameter. All accesses to that formal parameter will change the value in the local space, with no effect on the caller. On a return, no values

are copied back to the caller.

Call-by-reference parameters are handled by allocating local space in the callee's activation record for the address of the actual parameter and then copying the address of the actual parameter into that local space. All access to that formal parameter during execution of the callee are indirect accesses through this address, having a direct effect on the caller. On return, no action is taken other than reclaiming the callee's activation record space.

Register usage: You do not have to do any fancy register allocation. You can just use registers when generating assembly sequence for each three address code.

Run time error checking:

Code should be generated to check and report any out of range error on array/string indexing and subrange type variable access. For example, if an array a has an upper bound of 5, then A(10) results in an array index out of bounds error. Scalar out of range error is reported when a subrange variable is assigned a value not within the subrange. Assignment of a string value larger than 256 characters to a string variable is an error.

Code should be generated to check and report any "division by zero" errors.

Testing the code generator:

There are two ways to test your code generator: (1) Assemble your generated code, run it and check the results. (2) Examine your generated code carefully. The first method will catch many errors, but you should also check your generated assembly code carefully by hand to ensure that it is generating correct code. Correct output from execution does not guarantee that your generated code is completely correct.

Grading:

To get full credit for this project, you only need to perform code generation for programs without procedures and functions in PASC (you will still need to be able to call IO routines). The detailed grading policy follows:

- Output character, integer and string (30).
- I/O with simple declaration (30).
- Assignment statement with simple types (10).
- Flow of control statement with simple types (10).
- Everything above with array type (5).
- Everything above with record type (5).
- Run time error checking (10).

Extra 10 points will be given to programs that perform code generation for routine and function calls.

Extra 10 points for programs that generate correct code for queens.pasc and hist1.pasc (as well as all other simpler testing programs).

Project Due date:

You will need to design a demo that shows the capability of your code generator on **April 22**. This is a hard deadline.