CS 3091 Compiler Laboratory

SIL Compiler Design Project: **Problem Specification**.

1. 1.Implement an Interpreter for a subset (without subroutine abstraction) of the SIL Language (Simple Integer Language SIL – specification available at: http://www.athena.nitc.ac.in/~kmurali/Compiler/sil.html)
2. 2.Implement a two-pass compiler for SIL for SIM targer machine (Simple Integer Machine SIM – specification available at: http://www.athena.nitc.ac.in/~kmurali/Compiler/sim.html)
3. 3.Implement Extensions suggested for SIL as far as you can. This work will carry extra credit.

Learning Objectives:

The project aids the student to acquire a hands on pratical understanding of the following:

1. Practial realization of lexical and syntax analysis using the LEX/YACC tools and learn how the syntax directed translation scheme provided aids the construction of an intermediete abstract syntax tree from the source program.

2. Practical realization of symbol table, binding of variables, semantic analysis (including type and scope resolution), machine code generation, register allocation, parameter passing and run-time environment management for a very simple programming language.

Evaluation Policy:

1. The students have to form groups with maximum three members. There will be weekly evaluations and assessment of the progress will be recorded and graded.

2. Final grading will be based on cumulative scores of regular evaluations by relative grading.

3. Viva-voce/exam(s) conducted during the course of evaluation will carry maximum 25% marks.

4. S, A grades will be provided only after a special final evaluation among candidates who have completed all stages of the experiment.

Implementation Roadmap

The project must be carried out through various stages. This document does not contain information about extensions of SIL. Note that this is just a guideline for implementation and you shall and must deviate when you feel so, with of course, a reason.

Stage 1 **Simple Calculator**

Using the LEX/YACC environment implement a simple calculator. The input is an integer arithmetic expression. Integers may be recognized by a lexical analyzer created using LEX and passed to the YACC generated parser. Write the grammar in YACC. Use the syntax directed tranlation scheme in YACC to write semantic actions to output the post fix form of the expression. The grammar is E -> NUM | E+E | E*E | (E).

Stage 2: **Expression Tree Evaluation.**

Modify the attribute stack type (see usage of variable YYSTYPE in the YACC manual) to a structure that can store an integer value. Now modify the calculator program of Stage 1 to build an expression tree using the syntax directed translation scheme in YACC. This is the first phase (translation) where the source program is converted to an intermediate form - the expression tree. The second phase (execution) recursively traverses the expression tree and evaluates it.

Stage 3: **Straight Line program Interpreter**

Allow integer variable names a,b,c,...z in your expression evaluator. The token name “ID” may be used to denote a variable. Consider the grammar:

Slist -> Slist Stmt

Stmt -> ID = E; | read(ID); | write(E);

E -> E+E | E*E | (E) |NUM | ID

This allows programmes like the following (these are called straight line programs) :

a=3; b=5; read(c); write(a+b+c);

Write an interpreter for this small language using LEX and YACC.

Introduction of the assignment statement necessitates the concept of store. Each variable must be associated with a memory location which can hold its value. Since the variable names are fixed (not user defined) and there are exactly 26 variables, you can allocate an array of 26 integers and store the values of the variables in the array during execution.

Stage 4: **Adding Conditional and Iterative constructs.**

Expand the syntax of statements to include program control (conditional and iterative) statements.

Stmt -> ID=E; | read(ID) | write(E) | If (E) then Slist endif; | while (E) do Slist endwhile;

Allow relational operators: E -> E+E | E*E | (E) | (E<E) | (E>E) | E==E| NUM | ID

Use the synatax directed tranlation scheme to translate the source program to an intermediete form called the abstract syntax tree. Once the AST is constructed, the execution phase reduces to evaluating the AST. The node structure given below can be used for nodes in the AST (The type of the YACC variable YYSTACK must be set to be a pointer to this tree structure): http://www.athena.nitc.ac.in/~kmurali/Compiler/treenode.html

Stage 5: **Generating Machine Code**

Go back to stage 3 and modify the straight line program to build an abstract synatax tree (as in the previous stage) and then traverse the tree and generate target code in Simple Integer Machine (SIM) for the straight line program. SIM specification can be found here: http://www.athena.nitc.ac.in/~kmurali/Compiler/sim.html

The key issue in generating target code is register allocation. The simplest strategy would be to allocate the smallest numbered available register whenever a new register is needed. Your code generation can be designed in a way to free the highest numbered register after use. Hence a single variable for storing a count of how many registers are in use will be sufficient to keep track of the registers in use. The second issue is of binding variables to machine locations. In this simple case, 26 SIM machine locations can be pre-fixed to bind the variables.

Extend the code generator to generate code for conditional and iterative statements introduced in Stage 4. Here the key issue is generating labels for jump instructions necessary to implement the conditional and iterative constructs.

Stage 6: **Interpreter with User-defined variables and arrays**

In this stage Identifiers can be user defined variables. All variables in the program requires prior declaration. You will parse the declarations and construct a symbol table for storing variable information. The comiler decides the address where the variable is stored and saves this address in the symbol table (see the "binding" field in the following structure for symbol table: http://www.athena.nitc.ac.in/~kmurali/Compiler/symbol.html.)

The source program is translated into an AST in the first phase and the AST is interpreted in the next phase. The binding field of the symbol table must be modified (from what is given in the link) to contain a pointer to the location where the variable is stored. Since the size of an array is specified at the time of its declaration, the binding field for an array variable may be set to point to an array allocated to appropriate size.

Stage 7: **Compilation with User-defined variables and arrays**

The main conceptual difference here from Stage 6 is that we generate code for the target SIM machine instead of executing the AST immedietly. Thus run time environment is in the target SIM machine. Hence variables must be bound to storage locations in the SIM machine. For each variable, fixed memory addresses in the SIM machine may be allocated by the compiler. This strategy is called Static allocation. The addresses of the variables are stored in the symbol table. During the next phase (code generation), the compiler traverses the AST and generate target SIM code. References to variables are translated using the information in the symbol table.

Stage 8: **Adding Types**

Modify your syntax to make the programming language syntax compatible with the Simple Integer Language (SIL) specification except functions (we will take up functions in the next step). SIL specification is given here: http://www.athena.nitc.ac.in/~kmurali/Compiler/sil.html

You have to add logical operators to expression and boolean constants TRUE and FALSE.

Ensure that each sub tree is type checked before connecting them together while forming the Abstract Synatax Tree (AST) during syntax directed translation.   This will ensure that AST is created only if the program has no type errors at the end of the first pass.

If there is any error found, immediately report the error and stop compilation. Once a type checked AST is formed, the program must be free of type errors. Recursively traverse the tree to evaluate (interpreter) the tree or generate SIM target code (compilation).   You have to implement both the interpreter and compiler for this stage. You will be implementing only the compiler for the remaining phases.

Stage 9: **Function Call Implementation -Synatax and Semantics**

Now we are ready to build the full compiler for SIL. Please read the language specification carefully before proceeding with the implementation. You need to generate SIM target code for the whole language. You need not do the interpreter for this stage. Here is a structural outline for the grammar: http://www.athena.nitc.ac.in/~kmurali/Compiler/grammar.html

All stages of the compiler will need minor (and sometimes major) modifications to incorporate function calls. Functions require global declaration. Each function declaration shall result in creation of a global symbol table entry for the function that contain information such as its return type and names and types of each of its arguments.

Each function invocation must be type checked against the declaration. Note that SIL semantics requires checking function calls for name equivalence.

Stage 10: **Function call Implementation-Code generation.**

The SIL synatax and semantics is so designed to allow each function to be compiled independently if the global symbol table information is available.   Variables local to a function will be accessible through it local symbol table. The local symbol table information in one function is not required for the generation of code for another function.

The activities associated with a function call can be decomposed as the following:

1. Actions done by the caller before entry: This include pushing arguments into the run-time stack, allocating space in the stack for return value and finally making the call. The compiler must translate a function call into a sequence of machine instructions to perform these actions.

2.   Actions done by the callee upon entry: This code has to be generated in the beginning of the callee function. You must generate this code when the callee function is compiled. The callee will set the base pointer to set up its activation record after saving the base pointer of the caller in the stack.   It then processes its local declarations and allocate space in the stack for local variables.

3. Actions done by the callee before return: This code appears at the end of the calee function. It is generated during the compilation of the callee function. The callee places the return value into the appropriate location is stack, pops out space in the stack for local variables, restores the base pointer

of the caller (so that it points to the activation record of the caller) and then returns to the caller.

4. Actions done by the caller after return: This code is generated when the caller is compiled and the code must appear immedietely after the function call. The caller retrieves the return value from the stack and adjusts the stack pointer back to the position before the call was made.

Run time storage allocation will be necessary for this stage. Functions require run time store (called activation records) that hold local variables, call parameters and return value of the function. While processing the declaration part of a function, the compiler must assign storage to the local variables and parameters relative to the base of the activation record of the function. (The BP register of SIM is provided to store the base address of the current activation record in memory). Such relative bindings are required because a function may be invoked several times during run time and hence may have several activation records in the stack. The base of each activation record distinguishes the particular activation record. All Local variables and parameters of different invocations of the same function will have the same offset relative to the respective base. The SIL language enforces the stack discpline which permits activation records to be organized in a run time stack.

Integrity Policy:

Any case of cheating including copying of code will automatically qualify for an F grade in the course.

Wish you good luck !

Faculty :

1. Vineeth Paleri : vpaleri@nitc.ac.in

2. Saleena N : saleena@nitc.ac.in

3. K. Murali Krishnan : kmurali@nitc.ac.in