CS 536 — Spring 1995
Announcement 9

Programming Assignment #4

CSX Type Checker and Code Generator

Due: Friday, May 12, 1995
Not accepted after Monday, May 15, 1995

You are to write two modules; one that implements a type checker for CSX programs, and one that generates MIPS R2000 assembler code for CSX programs. Write a main program that calls your CSX parser. If the parse is successful, call the type checker; if the program contains no type errors, call the code generator. The CSX source program to be compiled is named on the compiler's command line, error messages are written to standard output, and the assembler code generated is placed in file `name.asm` where `name` is the identifier that appears right after `main` on the program's first line. Skeletons for the type checker and code generator modules may be found in ~cs536-1/public/proj4/c and ~cs536-1/public/proj4/c++.

The Type Checker

The type checker should be implemented as a function that takes as input an abstract-syntax tree built by the CSX parser. The type checker should produce an error message for each scoping and type error in the program represented by the tree, and should return a boolean value indicating whether the input had any type or scoping errors.

The scope rules of CSX are similar to those of C. An identifier is either global or local to a procedure. Procedures do not nest. All identifiers must be declared before they are used, and within a procedure a local declaration overrides a global declaration. Parameters of a procedure are considered local declarations within that procedure.

You must print an error message if a use of an undeclared identifier is found or if an identifier is redeclared within the same scope (an identifier declared globally *may* be redefined locally with a procedure).

An identifier may denote a variable, a constant, a procedure parameter, or a subprogram (function or procedure). Variables, constants, functions and parameters may be either of type integer or type boolean.

The type rules of the CSX language require the following:

- Arithmetic operators may be applied only to integer values.
- Logical operators (&& and ||) may be applied only to boolean values.
- Relational operators (==, <, >, !=, <=, >=) may be applied only to a pair of integer values or to a pair of boolean values.
- The type of a control expression (in an if-then-else or while loop) must be boolean.
- Integers, booleans and string literals may be written.
- Only integers may be read.

- The types of an assignment statement's left- and right-hand sides must be identical.
- The types of an actual parameter and its corresponding formal parameter must be identical.
- Only variables (local or global) and parameters may be passed as `ref` parameters.
- Assignment to constant identifiers is illegal.
- Only identifiers denoting procedures (`proc`s without a result type) may be called in statements.
- Only identifiers denoting functions (`proc`s with a result type) may be called in expressions. The type of a function call is the result type of the function.
- Return statements may only appear in functions. The expression returned by a return statement must have the same type as the function within which it appears.
- Relational operators *can* be applied to boolean values; by definition, false is less than true.

To prevent one type error from causing multiple error messages, assume that the result of an arithmetic operation is always integer, and that the result of a logical or relational operation is always boolean. For example, the following expression should cause only one error message:

```
(true + 3) + 4
```

You can use the line numbers contained in AST nodes to improve the specificity of your error messages; try to make them as informative as possible.

The Code Generator

You will generate assembler code for the MIPS R2000 processor. You will use the SPIM S20 interpreter to execute the programs you generate. The MIPS R2000 architecture and the SPIM S20 interpreter are described in the document "SPIM S20: A MIPS R2000 Simulator," available at MACC and in CS 536's web (Mosaic) page.

Your CSX code generator should be implemented as a function that takes as parameters an abstract-syntax tree representation of a CSX program and a text file. The code generator traverses the abstract-syntax tree, writing into the file assembler code that implements the program the abstract-syntax tree represents. Your code generator is only expected to handle type-correct programs; don't worry about translating type-incorrect programs. If any errors are detected during code generation, the code generator should return false; the contents of the text file need not be valid. If no errors are detected by the code generator, true is returned and the contents of the file should be a valid MIPS assembly program that can be executed using spim or xspim.

Consider the following simple CSX program:

```
main test;
    int a;

    get(a);
    print("Answer = ",2*a+1,skip);
end
```

This program might be translated into the following MIPS assembler code:

```
        .data
skip:   .asciiz     "\n"
a.i:    .word       0
        .text
main:
        li          $v0,5
        syscall     # read an int into $v0
        sw          $v0,a.i
        .data
str1:   .asciiz     "Answer = "
        .text
        li          $v0,4
        la          $a0,str1
        syscall     # print a string literal
        lw          $t0,a.i
        mulo        $t0,$t0,2
        add         $t0,$t0,1
        move        $a0,$t0
        li          $v0,1
        syscall     # write an int from $a0
        li          $v0,4
        la          $a0,skip
        syscall     # force a skip to the next line
        li          $v0,10
        syscall     # exit
```

This program would be written into file `test.asm`, since the name of the CSX program is test. The following commands could be used to assemble and execute the program:

```
spim –file test.asm
```

or

```
xspim –file test.asm
```

spim is batch-oriented and is good for running translated programs. xspim presents an X-window interface to a interpreter/debugger. It is good for debugging the code you generate.

Translating AST Nodes

The following table outlines what your code generator is expected to do for each ASTNodeKind. Details of the translation process will be discussed in class. Details may also be found in the text-book.

ASTNodeKind	Code Generator Action
program	Generate beginning of program; translate decls; translate procDecls; translate stmts; generate end of program.
decls	Translate decl, then decls.
procDecls	Translate procDecl, then procDecls.
varDecl	Allocate a memory location for the ident; If exp is non-null, translate it and generate code to store exp at ident's memory location.
constDecl	Allocate a memory location for the ident; translate exp; generate code to store exp at ident's memory location.
procDecl	Generate a subprogram prologue; translate argDecls; translate decls; translate stmts; generate a subprogram epilogue.
argDecls	Translate argDecl, then argDecls.
refArgDecl	Allocate an argument register for the ident (a reference parameter).
valArgDecl	Allocate an argument register for the ident (a value parameter).
stmts	Translate stmt, then stmts.
asg	Translate exp; generate code to store exp at ident's memory location.
ifThenElse	Translate exp; generate code to conditionally branch around then part; translate stmts of then part; generate a jump past else part; translate stmts of else part.
whileLoop	Generate a label for head-of-loop; translate exp; generate a conditional branch around loop body; translate stmts; generate a jump to head-of-loop.
getStmt	Generate a syscall to read an integer; generate a store of `$v0` into ident's memory location; translate getStmt.
printStmt	Translate exp; generate code to load value of exp into `$a0`; generate a syscall to print an integer or string; translate printStmt.

ASTNodeKind	Code Generator Action
callStmt	Generate code to begin a procedure call; translate args; generate a `jal` to ident; generate code to complete procedure call.
returnStmt	Translate exp; generate code to load exp into `$v0`; generate a jump to function's epilogue.
args	Translate exp; generate code to load value or address of exp into an argument register; translate args.
plus, minus, times, divide, andOp, orOp, eq, notEq, less, lessEq, greaterEq, greater	Translate left exp; translate right exp; generate code to apply the appropriate operator.
fctCall	Generate code to begin a function call; translate args; generate a `jal` to ident; generate code to complete function call (return value is in `$v0`).
ident	Generate code to load value at ident's memory location into a register. Do an indirect load if ident is a `ref` parameter.
intLit	Generate code to load intLit's value into a register.
trueVal	Generate code to load 1 into a register.
falseVal	Use `$0` (which always contains 0).
skip	Load address of "\n" into `$a0`.
strLit	Load address of string's text into `$a0`.
nullNode	Do nothing.
intType	Do nothing.
boolType	Do nothing.

How to Proceed

First implement the type checker. This involves implementing a block-structured symbol table. Walk the AST recursively, entering identifiers when they are declared and looking them up when they are used. Label nodes that can have a type (any node that can appear in an expression) with their type. This will simplify checking type-correctness in expressions and statements. Mark args nodes as value or ref; this will simplify code generation.

When your type checker is complete, begin work on the code generator. Start with simple constructs like get, print and assignment statements and simple expressions. Implement harder constructs like ifs, loops and procedures after the simpler constructs are working. For each construct you implement, you have two things to do. First, you must decide *what* MIPS code you want to generate. Try out the code you chose by creating (by hand) simple assembler programs. Run them on SPIM to verify that the code you chose really works.

Once you know the code you chose is viable, modify your code generator to generate that code. Look at the output of your code generator (the `name.asm` file) to verify that what is generated *looks* correct. If the output looks correct, run it through SPIM to verify that it *is* correct.

Once you've implemented a few simple constructs, you'll see how it all works. You can then add additional features until all of CSX is supported.

What To Hand In

Hand in listings of the main program, the type checker module, and the code generation module. Also hand in test data and corresponding output, for both correct and incorrect programs. Be sure to try all the test programs included in ~cs536/public/project4/tests. These programs will be named test1.csx, test2.csx, ... You should also provide the grader with an executable version of your CSX compiler. You can do this by copying your CSX compiler (in executable form) into ~cs536-1/public/proj4/handin. Use your login as the name of the compiler you place in the handin directory. As explained earlier, this compiler program should take the name of a CSX program to be compiled on its command line (e.g., `fischer test.csx`). If the CSX program is invalid, appropriate error messages should be written to standard output. Otherwise a translation of the CSX program should be placed in `name.asm` where `name` is the program's name, obtained from the first line of the CSX program. `name.asm` should be executable using spim or xspim. The grader will test your executable by running a series of standard test programs. Be sure to have your compiler print its author's name each time it is run to ease grading.