UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Prof. R. Fateman

Fall, 2005

CS 164 Assignment 5: Typechecking for MiniJava

Due: Thursday, Nov 3, 2005, 11:59PM

Warning: this assignment is going to take you more time than anything previously in this course. Start immediately.

Overall objectives

This assignment requires you to take the MJ abstract syntax tree from the previous exercise and process it further. You can use the AST which you produce, or use ours. We prefer you use ours, but if you have some good reasons to prefer yours (please state them!) you can use your own. The goal is to detect as many error or warning situations as possible among those errors in the program text that can be determined by static analysis. That is, we can look at the program, but we can't, generally speaking, try to run it. Although we we will call this type-checking, and certainly we will be checking for proper definition and agreement of types where they are used, we will be concerned with static semantic analysis.

The result of the processing should be a verification that the program is free of static-detectable errors, and/or a collection of error messages or warnings about the program.

Can we be more specific about how much code is needed?

Writing this typechecker will be more time-consuming than any of the previous projects, and will result in a program that is longer. The code should take less than about 600 additional non-comment lines of Lisp beyond what you've done so far, and what we are giving you. By contrast, the code for assignment 3 or 4 was about 100 lines, mostly of "grammar" and augments.

This project will require close attention to detail, and you will again have to try test cases of correct and incorrect program. We will try to install some "buggy" MJ programs in the `software/tests/` directory, but part of this exercise is for you to invent short but devious programs where you are particularly proud of finding errors. We encourage you to post on

the newsgroup test cases that you think are going to break the other students' type-checkers. We hope you don't break ours! Remember, these test cases must pass the earlier phases and produce an AST. You can also look at the text (section 5.1-5.2) which seems to illustrate that what you are doing would be more work in Java.

The sample MJ implementation (simple-interp) can serve as a model of what you need to do in terms of detecting and reporting errors: you can use it as a model in two ways: the manufacturing of an environment, and a framework for typechecking. We will give you much of the source code for the simple-interp, leaving out some critical but (we hope) well-documented routines.

We also have a typechecker for you to use, included in simple-interp.fasl. There is no guarantee that our typechecker is complete or correct; in some cases we may have allowed a program to pass through the system because our implementation of MJ is more liberal than the one envisioned by Appel. On the other hand, there may still be outright bugs in our system, in which case we would appreciate your showing us examples (and why you think they are processed incorrectly). As usual we will try to fix bugs reported to us promptly.

What typechecks are necessary?

It may appear from a reading of the text that Appel is somewhat vague about the possible forms of type errors in MJ programs, but in fact there are many indications of errors. These hints are presented in chapter 6, but also throughout the appendix defining MJ.

You should not check again for any errors that would have been intercepted by lexical analysis or parsing; you may assume that the abstract syntax tree was produced correctly.

Here are some considerations.

Declarations

Typechecking a `class` requires processing declarations, which in turn must install in a data structure any new information on types, variables, and functions. This structure should be an extension of the data structure that initially contains the built-in MJ function(s), and parent classes.

The details of the symbol table as partially provided to you in the interpreter provide a guide to how the symbol table for the typechecker would appear. You will find it necessary to add to the structure, however.

It should be clear that inheritance, mutually-recursive methods, shadowing of names, require a full understanding of the language, and some careful planning for programming.

Expressions

Every kind of expression has its own type rules. MJ is particularly impoverished in this sense, but you should nevertheless make a little table of what arguments are allowed for operators.

Please write down an explanation of why you think that EQUALITY is not part of the MJ language.

Statements

Statements are like expressions which do not return values. The fact that Lisp has essentially only expressions should make you wonder why they are so prevalent in other languages. Anyway, typechecking statements is not especially hard, and is similar to type-checking expressions.

What next?

We will, shortly, post the simple-interp code we would like you to modify for the typechecker.