Wellesley College ◊ CS301 Compiler Design ◊ October 27, 2000
Handout #09

PROBLEM SET 5

Due on Friday, November 3

OVERVIEW

The purpose of this problem set is twofold:

- to get experience with typing rules and type checkers; and
- to get some initial experience with the Bobcat language, an extension to Kitty that supports global first-order recursive functions.

This assignment has three problems worth a total of 100 points. It also has two extra credit problems worth a total of 100 points. If you have time, you are encouraged to attempt at least one of the two extra credit problems. (After all, what are you going to do with that extra hour you get on Sunday this week? `;-)`)

COLLABORATION DETAILS

As usual, you should work in three teams of two members each. You should work with the same partner with whom you worked on Problem Set 4.

SUBMISSION DETAILS

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The hardcopy packet should include the following:

- Your problem set header sheet.
- Your validity rules for Kitty from Problem 1.
- Your final version of the file `Check.sml` from problem 2.
- A transcript of the evaluation of `CheckTest.test()` from problem 2.
- A description of where you added calls to `validityCheck` within `Codegen.sml` in problem 2.
- Your final version of the file `TypeCheck.sml` from problem 3.
- A transcript of the evaluation of `TypeCheckTest.test()` from problem 3.

Your softcopy submission should consist of your local versions of the `kitty/check` and `bobcat/typecheck` program directories described below.

PROBLEM 1 [20]: VALIDITY RULES FOR KITTY

This is a pencil and paper problem in which you will write down validity rules that specify a conservative approximation of which Kitty expressions are statically valid.

A Kitty expression is **statically valid** if at compile-time it can be proven that when the expression is evaluated, none of the following dynamic errors can occur:

(1) unbound variable;

(2) an expression returns no value in a context where a value is expected; and
(3) an expression returns a value in a context where no value is expected.

Even in a simple language like Kitty, non-trivial properties like static validity can be computationally undecidable. However, it is possible to develop rules and algorithms that **conservatively approximate** such properties. In the case of static validity, a conservative approximation may give false negatives but may never give false positives. That is, it may say that an expression is invalid even if evaluating it would not give rise to one of the three errors listed above; but it may never say that an expression is valid if evaluating it would in fact give one of these errors. (Such an approximation is said to be **sound**.)

For example, consider the following Kitty expressions:

1. `3 + (if true then 4 else ())`

2. `(while true do (); x)`

The first expression will dynamically evaluate to 7 without encountering an error, but a conservative validity checker can say that it is invalid because the "valueness" of the two arms of the conditional do not match. The second expression will dynamically get stuck in an infinite loop without ever encountering the unbound variable `x`, but a conservative validity checker can still declare it to be invalid because it contains an unbound variable.

These examples suggest why exact validity checking can be undecidable. Imagine that we replace `true` in the above examples by an arbitrarily complex expression *E*. Then exact validity checking is equivalent to determining whether *E* evaluates to a non-zero number – the sort of property that "smells" undecidable. (Actually, because the range of Kitty's integers is limited, it turns out that this property *is* actually decidable in Kitty. However, if Kitty supported arbitrarily large integers, the property would not be decidable. See Challenge Problem 2.)

A **validity judgement** has the form $S \triangleright E : V$, where S is a set of identifiers, E is a Kitty expression and V one of the two symbols **some** or **none**. Intuitively, the judgement means that if E appears in the scope of the variables in S, then its "valueness" is V, where **some** indicates the expression returns a value and **none** indicates that it returns no value.

A **validity rule** has the form:

$$\frac{Judgement_1 \; ; \; \ldots \; ; \; Judgement_n}{Judgement_0}$$

Such a rule means that $Judgement_0$ can be derived given derivations for $Judgement_1$ through $Judgement_n$.

Write down a collection of validity rules for Kitty that conservatively approximate the static validity property. Your rules should look similar to the typing rules for Bobcat, which can be found in the Bobcat Reference Manual (Handout #10).

PROBLEM 2 [30]: A VALIDITY CHECKER FOR KITTY

In this problem, you are to implement a validity checker for Kitty based on the rules you developed in Problem 1. You should do this by fleshing out the definition of the following function:

`validityCheck : AST.exp -> unit`
: Returns unit if the given Kitty expression satisfies the conservative approximation of static validity from Problem 1. Otherwise, signals an `Invalid` exception with an explanatory error message.

A skeleton of this function definition can be found in the file `Check.sml`, which is available from the directory `~/cs301/download/kitty/check` in the cs301 download folder. You should make a local copy of this entire directory.

To implement validity checking, you will need an abstraction for a set of identifiers. Use the `SplaySetFn` functor from the SMLNJ library. The signature for the structure returned by this functor can be found in `/usr/share/smlnj/src/smlnj-lib/Util/ord-set-sig.sml`. Sample uses of this functor can be found in the code generators for SLiP in the directory `~cs301/download/slip`.

The following functions in the `Check` structure can be used to test your validity checker:

`checkString : string -> unit`
: Returns unit if the Kitty expression denoted by the given string satisfies the conservative approximation of static validity from Problem 1. Otherwise, signals an `Invalid` exception with an explanatory error message.

`checkFile : string -> unit`
: Returns unit if the Kitty expression that is the contents of the file with the given filename satisfies the conservative approximation of static validity from Problem 1. Otherwise, signals an `Invalid` exception with an explanatory error message.

The structure `CheckTest` contains a `test()` function that will apply the validity checker to all Kitty expressions in the list `testExps` and all files in the list `testFiles`. You should extend these lists with additional test cases.

The code generator from Problem Set 4 assumes that the Kitty source expression is statically valid. Insert calls to `validityCheck` in the code generator so that this assumption is justified.

PROBLEM 3 [50]: A TYPE CHECKER FOR BOBCAT

In this problem you will implement a type checker for Bobcat based on the Bobcat type checking rules in the Bobcat Reference Manual (Handout #10). You should do this by fleshing out the definition of the following function:

```
typeCheck : AST.prog -> unit
```
Returns unit if the given Bobcat program is type correct according to the Bobcat typing rules. Otherwise, signals a `TypeError` exception with an explanatory error message.

A skeleton of this function definition can be found in the file `TypeCheck.sml`, which is available from the directory `~/cs301/download/bobcat/typecheck` in the cs301 download folder. You should make a local copy of the entire bobcat directory structure rooted at `~/cs301/download/bobcat`.

Your type checker will walk over the abstract syntax tree of a Bobcat program. The SML definition of Bobcat abstract syntax can be found in `bobcat/AST.sml`. It is similar to Kitty abstract syntax except for the following differences:

- The `exp` datatype includes a `FunApp` constructor that takes an identifier (the name of the called function) and an expression list (the actual parameters to which the function is applied).

- The `exp` datatype no longer has `NullApp` and `UnApp` constructors. These are superseded by the `FunApp` constructor.

- The `exp` datatype includes an Error constructor that takes a single string argument.

- The `decl` datatype for variable declarations has been replaced with a `varDecl` datatype with two constructors: `VarDecl`, for declarations that do not specify the type of the declared identifier, and `VarDeclTyped`, for declarations that do specify the type of the declared identifier.

- A Bobcat program is introduced by the constructor `GlobalLet`, which takes a list of global declarations and an expression that is the body of the program. A global declaration is either a variable declaration (tagged with `VDecl`) or a list of mutually recursive function declarations (tagged with `FDecls`). A function declaration is an element of the `funDecl` datatype, created by the `FunDecl` constructor with four arguments: (1) the name of the function; (2) a list of argument name/type pairs; (3) the return type of the function (`Void` if the function returns no value); and (4) a Bobcat expression that is the body of the function. Global declarations should be processed in the order in which they appear within the list.

- The Bobcat AST expresses types as follows:

  - The datatype `baseType` specifies the types of Bobcat values. It has the constructors `Bool`, `Char`, and `Int`.

  - The datatype `expType` specifies the types of Bobcat expressions. It has two constructors: `Void` (the type of an expression with no value) and `Base(b)` (the type of an expression with base type `b`.)

  - The datatype funType specifies the types of Bobcat functions. The constructor invocation `FunType([`$argType_1$`,…,`$argType_n$`], `*returnType*`)` denotes the type of a function whose

parameters have base types $argType_1, \ldots, argType_n$ and whose result has expression type *returnType*. (The `funDecl` datatype is not used elsewhere within Bobcat abstract syntax, but it is handy for programs like type checkers that manipulate Bobcat programs.)

To implement type checking, you will need an abstraction for a type environment. It is recommended that you use the `StringTable` structure in `bobcat/utils/StringTable.sml`.

Since a Bobcat program may reference any standard library function, you need to correctly handle references to these. (A list of standard library functions in Bobcat can be found in the Bobcat Reference Manual.) A standard way to do this is to typecheck programs with respect to an initial type environment that includes entries for all standard library functions.

The following functions in the `TypeCheck` structure can be used to test your type checker:

`typeCheckString : string -> unit`
: Returns unit if the Bobcat program denoted by the given string is type correct according to the Bobcat typing rules. Otherwise, signals a `TypeError` exception with an explanatory error message.

`typeCheckFile : string -> unit`
: Returns unit if the Bobcat program that is the contents of the file with the given filename is type correct according to the Bobcat typing rules. Otherwise, signals a `TypeError` exception with an explanatory error message.

The structure `TypeCheckTest` contains a `test()` function that will apply the type checker to all Bobcat programs in the list `testPgms` and all files in the list `testFiles`. You should extend these lists with additional test cases.

CHALLENGE 1 [50]: A ONE-PASS VALIDITY CHECKER AND CODE GENERATOR

Adding calls to `validityCheck` within the Kitty code generator (as in Problem 2) is an easy way to guarantee that the code generator is called only on statically valid Kitty programs. But this simple approach involves making two passes over the Kitty AST: one to perform validity checking, and a second one to perform code generation. Show that it is possible to fuse these two passes into a single pass by implementing a function that performs validity checking and code generation in a single walk over the Kitty AST.

CHALLENGE 2 [50]: UNDECIDABILITY IN KITTY

Consider the following problem:

> Given a closed Kitty expression *E*, does evaluation of *E* terminate with a non-zero number?

(Recall that an expression is **closed** if it has no free variables.)

Part a. Prove that the above problem is decidable in Kitty. The key to decidability is the fact that Kitty integers are restricted to a limited range (32 bits).

Part b. Prove that if Kitty were extended to support integers with arbitrarily large magnitude, then the above property would be undecidable.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 5
Due Friday, November 3, 2000

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (*any teams collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the total time each team member spent on the parts of this problem set. (Note that spending less time than your partner does not necessarily imply that you contributed less.) Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	**Time For** _____ **(Member #1)**	**Time For** _____ **(Member #2)**	**Score**
General Reading			
Problem 1 [20]			
Problem 2 [30]			
Problem 3 [50]			
Challenge 1 [50]			
Challenge 2 [50]			
Total			