ECE251 Final Project, Part 2, Fall 2010*

Patrick Lam

Due: Monday, December 6

Brief Overview

In this part of the project, you will implement a **symbol table** (see L16 notes) and **type checker** (Lectures 22–23) following my skeleton code. I've modified my skeleton to dump the symbol tables if you give it the `-symbol` option, and the default behaviour of the compiler will now also type check the code.

Submission and Marking

I encourage you to submit this part of your project by **Sunday, November 28th**. I don't have a stick, but the carrot is that you will get 5% extra on this part of the project if you do so. (You can't use grace days to get the 5% extra.)

The best way to submit your project is by committing it to the SVN repository and also submit a `tar.gz` file to the external submission site. Once I've observed a handin to the external submission, I will try to pull your SVN and mark that; if I can't, then I will download and mark the submission to the submission site.

To mark your code, we will compare the output of your code with the output of our sample solutions. We will provide test cases and expected output for the good cases of the symbol table. This time, we will not provide tests for error cases, either for the symbol table or the type checker. We will evaluate your project based on both the good and the bad test cases.

To mark the error cases, we will expect that your compiler will emit a set of error codes as described in this document. We will try to automatically compare the first five characters of your error codes with ours; if that doesn't work, we will use our discretion to award marks.

Operation of the Symbol Table

The WIG symbol table maintains global namespaces for HTMLs, functions, schemas and sessions. It also maintains local scopes, with a top-level scope for each function and session, and nested scopes as necessary. Each scope contains names and types of variables in that scope. Your job is to **implement** methods to **insert** into and **query** the symbol tables. The global namespaces are easy: you just need to delegate to the appropriate `HashMap` operations. Scoped identifiers are a bit trickier: you need to look on the stack of identifiers for the proper identifier, if it exists.

At the same time, you also need to **implement** code which **populates** the symbol table and **checks** for definitions. I recommend that you implement two visitors (L14) over the Abstract Syntax Tree. The first visitor, `SymbolDefVisitor`, populates the symbol table and checks that symbols are not multiply-defined. The second visitor, `SymbolUseVisitor`, checks that all variables are defined before they are used.

This part of the project should take you under 2 hours. The skeleton `SymbolTable` contains 175 lines while my solution contains 234 lines, for a delta of 59 lines. The delta for the visitors is 69 lines.

*r2: added [S10], multiply defined function

Operation of the Type Checker

The type checker propagates computed type information through the AST and compares the computed types with declared types at various points. Your task is to fill in the `TypeCheckingVisitor` class with the appropriate method bodies for the `leave()` methods to implement a simple type checker.

I've provided a `Map`, `types`, in the `TypeCheckingVisitor` class. As your visitor visits expressions, it should add the types that it computes for expressions to the `types` map. For instance, if it sees a `BoolLiteralExp e`, it should `put` the `BOOL` constant into the `types` map for `e`. At an operation, it should check that the input types are appropriate, and then add the output type for that operation. Hint: don't use `==` on your types, use `.equals()`.

The `TypeCheckingVisitor` skeleton is 132 lines while my solution is 295 lines, for a delta of 163 lines.

Summary

In this part of the project, you will add code to my symbol table and type checker skeletons. The symbol table is just implementing a data structure on top of standard Java `Map`s from the Collections API. The type checker involves using the Visitor design pattern to verify that expressions have the correct types.

Type Checking Rules

Here are the rules for the non-tuple expressions. I've written text for tuples and statements below.

$$\frac{}{P;\Gamma \vdash \texttt{true} : \text{bool}} \quad \frac{}{P;\Gamma \vdash \texttt{false} : \text{bool}} \quad \frac{}{P;\Gamma \vdash n : \text{int}} \quad \frac{}{P;\Gamma \vdash \texttt{STRING\_LIT} : \text{string}}$$

$$\frac{P;\Gamma \vdash e : \text{int}}{P;\Gamma \vdash -e : \text{int}} \quad \frac{P;\Gamma \vdash e : \text{bool}}{P;\Gamma \vdash \sim e : \text{bool}}$$

$$\frac{P;\Gamma \vdash e_1 : t \quad P;\Gamma \vdash e_2 : t}{P;\Gamma \vdash e_1 \texttt{==} e_2 : \text{bool}} \quad \frac{P;\Gamma \vdash e_1 : t \quad P;\Gamma \vdash e_2 : t}{P;\Gamma \vdash e_1 \texttt{!=} e_2 : \text{bool}} \quad \frac{P;\Gamma \vdash e_1 : \text{int} \quad P;\Gamma \vdash e_2 : \text{int}}{P;\Gamma \vdash e_1 < e_2 : \text{bool} \quad P;\Gamma \vdash e_1 \leq e_2 : \text{bool} \quad P;\Gamma \vdash e_1 > e_2 : \text{bool}}$$

$$\frac{P;\Gamma \vdash e_1 : \text{int} \quad P;\Gamma \vdash e_2 : \text{int}}{P;\Gamma \vdash e_1 \geq e_2 : \text{bool} \quad P;\Gamma \vdash e_1 + e_2 : \text{int} \quad P;\Gamma \vdash e_1 - e_2 : \text{int} \quad P;\Gamma \vdash e_1 * e_2 : \text{int} \quad P;\Gamma \vdash e_1 / e_2 : \text{int} \quad P;\Gamma \vdash e_1 \% e_2 : \text{int}}$$

$$\frac{P;\Gamma \vdash e_1 : \text{string} \quad P;\Gamma \vdash e_2 : \text{string}}{P;\Gamma \vdash e_1 < e_2 : \text{bool} \quad P;\Gamma \vdash e_1 \leq e_2 : \text{bool} \quad P;\Gamma \vdash e_1 > e_2 : \text{bool} \quad P;\Gamma \vdash e_1 \geq e_2 : \text{bool} \quad P;\Gamma \vdash e_1 + e_2 : \text{string}}$$

$$\frac{P;\Gamma \vdash e_1, e_2 : \text{bool}}{P;\Gamma \vdash e_1 \texttt{\&\&} e_2 : \text{bool} \quad P;\Gamma \vdash e_1 \texttt{||} e_2 : \text{bool}} \quad \frac{P;\Gamma \vdash e_1 : t \quad P;\Gamma \vdash e_2 : t}{P;\Gamma \vdash e_1 \texttt{=} e_2 : t}$$

I won't write out the rules for function calls and returns, but a function call must have arguments with the same types as the function declaration, and gives the declared return type of the callee. Also, a return statement must return the same type as the declared return type of the function, and may only appear inside a function.

`if` and `while` statements must have boolean conditions. `PlugDocument` arguments must have non-void non-tuple types as parameters.

Tuple Expressions

Tuple expressions work like this. The WIG program defines a number of fixed schemas (i.e. struct definitions), which contain a set of fields. There are three tuple operators: `<<`, `\+` and `\-`. An expression `t1 << t2` creates a tuple expression with all of the fields from `t1` and `t2`. This type-checks only if there is a schema containing all of the fields in `t1` and `t2`. An expression `t1 \+ (ids)` creates a tuple where you keep all of the fields

in `ids` from `t1` and throw out the other ones. Again, there has to be a schema with exactly these fields. Finally, an expression `t1 \- (ids)` creates a tuple where you throw out the fields in `ids` from `t1` but keep the other ones.

Here is an example.

```
schema S { bool b; int i; string s; }
schema S1 { bool b; }
schema S2 { int i; string s; }

tuple S s; tuple S1 s1; tuple S2 s2;

s1 << s2; "creates a valid tuple of type S";
S \+ (b); "creates a valid tuple of type S1";
S \- (b); "creates a valid tuple of type S2";
```

Error Codes

Use the following error codes for symbol table errors and type checking errors. It'll make everyone's life much easier.

```
[S01]: undefined schema
[S02]: undefined HTML: H
[S03]: undefined variable name: V
[S04]: undefined function: F
[S05]: multiply-defined schema: S
[S06]: multiply-defined field name: F
[S07]: multiply-defined variable: V
[S08]: multiply-defined html: H
[S09]: multiply-defined session: S
[S10]: multiply-defined function: F

[T01]: not on non-boolean expression
[T02]: negation on non-int expression
[T03]: arg type for arg N in call to F doesn't match: expected T1, got T2
[T04]: unequal arg types for E /
         wrong number of args
[T05]: non-boolean types for E
[T06]: non-int types for E
[T07]: non-int, string types for E
[T08]: plug doesn't have simple non-void type: E
[T09]: non-equal types for assignment E
[T10]: mismatched return type for expression E
[T11]: returning void from non-void function
[T12]: if statement has non-boolean condition E
[T13]: while statement has non-boolean condition E
[T14]: return statement outside function
[T15]: undefined field F in schems S
[T16]: attempt to dereference non-tuple variable / attempt to use <<
           on non-tuple types
[T17]: no matching schema found for literal L
[T18]: tuple operation OP gives nonexistent schema
```

Line numbers

I forgot to add line numbers to the `ASTNode` class[1]. They would make error reporting much easier. You would have to modify your grammar to add the lexer token information to each `ASTNode` as you create it. Then you could report line and column information when you encounter an error, or to add debug information.

[1] OK, I didn't even have an `ASTNode` class in the original skeleton.