Compiler Construction/Fall 2014/Project Milestone 2 part A

Eva Rose
evarose@cs.nyu.edu

Kristoffer Rose
krisrose@cs.nyu.edu

Assigned Thursday 10/16/2014, due Monday 11/3/2014 at 8am

Welcome to the second project milestone for the fall 2014 compiler construction class at the NYU Courant Institute! Your task is to program a *type checker* for the JST language using the HACS system.

However, to help structure your project (and isolate HACS issues and questions), we first ask in this *part A* that you create an SDD (Syntax-Directed Definition) that defines a *type attribute* for JST programs.

1 Assignment

To complete part A please submit one file to the **pr2a** assignment in the NYU classes system:

pr2a-*YourName*.pdf with an SDD that defines an attribute *type* for JST programs, which for every expression associates a type with it. Remember to define and explain all the attributes that you need to solve the task.

The syntax of the JST language is summarized in the appendix (and is the same as in *pr1* except with the two discussed fixes).

2 JST Typing

Fig. 1 is a small sample of a JST program, with line numbers, and with keywords and strings highlighted in color (we shall use this style throughout). When executed, this program will output the string `Hello, World!`.

The example shows how all declarations of JST must be explicitly typed. The class members are all declared: `greeting` is declared to be of `string` type and `greet` is a method that takes no parameters and returns a `string`.

In this section we give the typing rules for each of the syntactic forms in JST; we liberally use abbreviations like E for *Expression* and ℓ for *Literal*.

2.1 Definition (assignment compatible types)**.** We say that a type t_1 is *assignment compatible to* a type t_2 if a variable or member of type t_2 can be assigned a value of type t_1. In JST, every type is assignment compatible with `any`, and every type is assignment compatible with itself.

2.2 Definition. *Literals* have the following types:

1. An *Identifier* literal must have been declared (in the current scope), and has the type that it was declared to have.

2. A *Integer* literal has type `int`.

```
class Greeter {
  string greeting;
  string greet() {
    return "Hello, " + this.greeting + "!";
  }
}

function string main() {
  var Greeter g;
  g = {greeting : "World"};
  return g.greet();
}
```

Figure 1: JST greeter program.

3. A *String* literal has type `string`.

4. An object literal $\{m_1{:}\ell_1, \ldots, m_k{:}\ell_k\}$ (for $k \geq 0$), where each m_i must be a distinct *Identifier*, can only be assigned to a variable or member of some *Identifier* class type that has *precisely* the member names $m_1, \ldots m_k$ (in any order) and where the corresponding ℓ_1 has a type which is assignment compatible to the type declared for the member.

2.3 Definition (l-value)**.** An *Expression* on one of the following forms is an *l-value*: an *Identifier*, and member access $E.m$.

2.4 Definition. These are the type rules for *Expressions*. Each rule states the type constraints on the components and the type of the resulting expression.

1. The expression `this` is only permitted inside the body of member declarations inside a class declaration `class` C `{` $m_1 \ldots m_k$ `}`, where it then has the type C.

2. For $E.m$ with m a member name, E must be of a declared class type C that has the member m; the result type is the declared type of the member in C.

3. For a function call E`(`$E_1, \ldots, E_n$`)` (for $n \geq 0$), E must be of a "function type" written as

$$(p_1{:}t_1, \ldots, p_n{:}t_n) \Rightarrow t$$

   (with the same n) and each of the E_i must have type t_i; the resulting expression then has type t.

4. For `!`E, E must be of type `boolean` as is the result.

5. For `-`E, and `+`E, E must be of type `int` as will the result.

6. For E_1`*`E_2, E_1`/`E_2, E_1`%`E_2, and E_1`-`E_2, both E_1 and E_2 must have type `int` as does the result.

7. For E_1`+`E_2, both E_1 and E_2 must have type `int` or `string`; the result is type `int` if both have type `int` and `string` otherwise.

8. For E_1`<`E_2, E_1`<=`E_2, E_1`>`E_2, and E_1`>=`E_2, E_1 and E_2 must have the same type, which should be either `int` or `string`; the result is of type `boolean`.

9. For E_1`==`E_2 and E_1`!=`E_2, E_1 and E_2 must have the same type; the result is of type `boolean`.

10. For E_1`&&`E_2 and E_1`||`E_2, both E_1 and E_2 must have type `boolean` as will the result.

11. For E_1`=`E_2 then E_1 must be an *l-value* (Def. 2.3) such that the type of E_2 is *assignment compatible to* the type of E_1 (Def. 2.1); the result type is the type of E_2.

12. For E_1`+=`E_2 then E_1 must be an *l-value* (Def. 2.3) of either type `int` or `string`. If E_1 is of type `int` then E_2 must also be of type `int` as will the result type. If E_1 is of type `string` then E_2 must also be a `string` as will the result type.

13. An expression E_1`,`E_2 is only permitted inside function call argument lists (item 3 above) with the stated type rule.

2.5 Definition. Statements have the following type and scoping rules:

1. A block statement `{`$\ldots$`}` starts a new nested scope inside the braces.

2. A declaration `var` v`:`t`;`$\ldots$ introduces the new name v of type t scoped over the rest of the statements in the nearest enclosing `{}`s.

3. `while` and `if` tests must be of type `boolean`.

2.6 Definition. A class declaration `class` C `{` $m_1 \ldots m_k$ `}` declares C as a type in the top scope, with members as follows:

1. A field member declaration t m`;` declares the member name m of C to have type t.

2. A method member declaration t m`(`t_1 p_1`,`$\ldots$`,`t_n p_n`) {`$\ldots$`}` declares the member m of C to have the type `(`t_1 p_1`,`$\ldots$`,`t_n p_n`)=>`t.

2.7 Definition. Inside a method member or a function declaration, the parameters are in scope. Specifically,

1. For the class member declaration t m`(`t_1 p_1`,`$\ldots$`,`t_n p_n`) {`S`}` the declared parameters $p_1, \ldots, p_n$ are in scope in the statements S with types $t_1, \ldots, t_n$, respectively.

2. For the function declaration `function` t f`(`t_1 p_1`,`$\ldots$`,`t_n p_n`) {`S`}` the declared parameters $p_1, \ldots, p_n$ are in scope in the statements S with types $t_1, \ldots, t_n$, respectively.

2.8 Definition. Declarations of functions and classes occur at the top level in a JST program. All the defined classes and functions are available in the entire program, also before their declaration. (This permits mutually recursive classes and function calls.)

2.9 Definition. Values of type `string` permit additional operations as if `string` was defined with a declaration like the following (which is not syntactically correct):

```
class string {
  int length;
  int charCodeAt(int index) {...}
  string substr(int start, int length) {...}
}
```

A JST Grammar

The grammar for JST is given by the following HACS, which will be available for you in part B.

```
// [NYU Courant Institute] Compiler Construction/Fall 2014/Project Milestone 2 part B
//
// Base parser.
// See http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/pr2/pr2b.pdf

module edu.nyu.csci.cc.fall14.Pr2Base {

// PROGRAM

sort Program | ⟦ ⟨Declarations⟩ ⟧ ;

// DECLARATIONS

sort Declarations | ⟦ ⟨Declaration⟩ ⟨Declarations⟩ ⟧ | ⟦⟧ ;

sort Declaration
| ⟦ class ⟨ Identifier ⟩ { ⟨Members⟩ } ⟧
| ⟦ function ⟨Type⟩ ⟨ Identifier ⟩ ⟨ArgumentSignature⟩ { ⟨Statements⟩ } ⟧
;

sort Members | ⟦⟨Member⟩ ⟨Members⟩ ⟧| ⟦⟧;

sort Member
| ⟦ ⟨Type⟩ ⟨ Identifier ⟩ ; ⟧
| ⟦ ⟨Type⟩ ⟨ Identifier ⟩ ⟨ArgumentSignature⟩ { ⟨Statements⟩ } ⟧
;

sort ArgumentSignature
| ⟦ ( ) ⟧
| ⟦ ( ⟨Type⟩ ⟨ Identifier ⟩ ⟨ TypeIdentifierTail⟩ ) ⟧
;
sort TypeIdentifierTail | ⟦ , ⟨Type⟩ ⟨ Identifier ⟩ ⟨TypeIdentifierTail⟩ ⟧ | ⟦ ⟧ ; // For each comma

// STATEMENTS

sort Statements | ⟦ ⟨Statement⟩ ⟨Statements⟩ ⟧ | ⟦⟧ ;

sort Statement
| ⟦ { ⟨Statements⟩ } ⟧
| ⟦ var ⟨Type⟩ ⟨ Identifier ⟩ ; ⟧
| ⟦ ; ⟧
| ⟦ ⟨Expression⟩ ; ⟧
| ⟦ if ( ⟨Expression⟩ ) ⟨ IfTail ⟩ ⟧
| ⟦ while ( ⟨Expression⟩ ) ⟨Statement⟩ ⟧
| ⟦ return ⟨Expression⟩ ; ⟧
| ⟦ return ; ⟧
;

sort IfTail | ⟦ ⟨Statement⟩ else ⟨Statement⟩ ⟧ | ⟦ ⟨Statement⟩ ⟧ ; // Eagerly consume elses

// TYPES
```

```
sort Type
|  ⟦ boolean ⟧
|  ⟦ int ⟧
|  ⟦ string ⟧
|  ⟦ void ⟧
|  ⟦ ⟨ Identifier ⟩ ⟧
;

// EXPRESSIONS

sort Expression

|  sugar ⟦ ( ⟨Expression#e⟩ ) ⟧@10 →#e

|  ⟦ ⟨SimpleLiteral⟩ ⟧@10
|  ⟦ ⟨ Identifier ⟩ ⟧@10
|  ⟦ this ⟧@10

|  ⟦ ⟨Expression@9⟩ ( ⟨Expression⟩ ) ⟧@9
|  ⟦ ⟨Expression@9⟩ ( ) ⟧@9
|  ⟦ ⟨Expression@9⟩ . ⟨ Identifier ⟩ ⟧@9

|  ⟦ ! ⟨Expression@8⟩ ⟧@8
|  ⟦ − ⟨Expression@8⟩ ⟧@8
|  ⟦ + ⟨Expression@8⟩ ⟧@8

|  ⟦ ⟨Expression@7⟩ * ⟨Expression@8⟩ ⟧@7
|  ⟦ ⟨Expression@7⟩ / ⟨Expression@8⟩ ⟧@7
|  ⟦ ⟨Expression@7⟩ % ⟨Expression@8⟩ ⟧@7

|  ⟦ ⟨Expression@6⟩ + ⟨Expression@7⟩ ⟧@6
|  ⟦ ⟨Expression@6⟩ − ⟨Expression@7⟩ ⟧@6

|  ⟦ ⟨Expression@6⟩ < ⟨Expression@6⟩ ⟧@5
|  ⟦ ⟨Expression@6⟩ > ⟨Expression@6⟩ ⟧@5
|  ⟦ ⟨Expression@6⟩ <= ⟨Expression@6⟩ ⟧@5
|  ⟦ ⟨Expression@6⟩ >= ⟨Expression@6⟩ ⟧@5

|  ⟦ ⟨Expression@5⟩ == ⟨Expression@5⟩ ⟧@4
|  ⟦ ⟨Expression@5⟩ != ⟨Expression@5⟩ ⟧@4

|  ⟦ ⟨Expression@3⟩ && ⟨Expression@4⟩ ⟧@3

|  ⟦ ⟨Expression@2⟩ || ⟨Expression@3⟩ ⟧@2

|  ⟦ ⟨Expression@2⟩ = ⟨Expression@1⟩ ⟧@1
|  ⟦ ⟨Expression@2⟩ = ⟨ObjectLiteral⟩ ⟧@1 // ObjectLiteral treated separately
|  ⟦ ⟨Expression@2⟩ += ⟨Expression@1⟩ ⟧@1

|  ⟦ ⟨Expression@1⟩ , ⟨Expression⟩ ⟧
;

sort Literal        |  ⟦ ⟨SimpleLiteral⟩ ⟧  |  ⟦ ⟨ObjectLiteral⟩ ⟧ ;  // Not used from Expression.
sort SimpleLiteral  |  ⟦ ⟨String⟩ ⟧  |  ⟦ ⟨Integer⟩ ⟧ ;
sort ObjectLiteral  |  ⟦ { } ⟧  |  ⟦ { ⟨KeyValue⟩ ⟨KeyValueTail⟩ } ⟧;
```

```
sort KeyValueTail  |  ⟦ , ⟨KeyValue⟩ ⟨KeyValueTail⟩ ⟧ |  ⟦ ⟧ ;
sort KeyValue      |  ⟦ ⟨ Identifier ⟩ : ⟨ Literal ⟩ ⟧ ;

// LEXICAL CONVENTIONS

space [ \t\n\r] | '//' [^\n]* | '/*' ( [^*] | '*' [^/] )* '*/'   ; // Inner /* ignored

token Identifier   | ⟨LetterEtc⟩ (⟨LetterEtc⟩ | ⟨Digit⟩)* ;

token Integer      | ⟨Digit⟩+ ;

token String       | \' ( [^\'\\\n] | \\ ⟨Escape⟩ )* \'
                   | \" ( [^\"\\\n] | \\ ⟨Escape⟩ )* \"
                   ;

token fragment Letter      | [A−Za−z] ;
token fragment LetterEtc   | ⟨Letter⟩ | [$_] ;
token fragment Digit       | [0−9] ;

token fragment Escape | "\n" | \' | \" | \\ | [nt] | 'x' ⟨Hex⟩ ⟨Hex⟩ ;
token fragment Hex    | [0−9A−Fa−f] ;

// Dummy scheme to avoid 0.9.0 bug.
sort  Program | scheme Compile(Program);
Compile(#) →#;

}
```