CS 321: Programming Languages — Homework 9

Install the `plaitypus` language via DrRacket's *File | Install Package ...* menu item. Type `plaitypus` into the "Package Source" box and click "OK".

You can find documentation for plaitypus by going to the main documentation page for your installation (type F1 in DrRacket and then click the "Up" links until there aren't any more) and following the "Plaitypus Language" link you find there.

Start your program with: `#lang plaitypus`

1 Typechecker Extensions

In this assignment, you'll be extending the typechecker we wrote in lecture for the TFAE language to cover the extended TLFAE language. Thus, you will need representations for TLFAE programs and types. Your solution must provide `parse` and `parse-type` functions to construct them from the surface syntax described below.

You may use our definitions in `hw9-starter.rkt` for them, or come up with your own. As you prefer.

Note that the input to the parsers must be an s-expression. Quasiquote produces s-expressions, so the following works: ``(parse `{+ 5 6})``

Note that to get values of s-expression type, you will need to use quasiquote explicitly. Code that was equivalent before may not work in plaitypus. For example, the following would not work because these parser inputs do not have type s-expression:

```
(parse 7)
(parse 'true)
(parse-type 'number)
```

2 True, False, equals, and if

Add support for `true`, `false`, `{= ... ...}`, and `{if ... ... ...}` expressions to `TFAE`:

```
<TLFAE> = <num>
        | true
        | false
        | {+ <TLFAE> <TLFAE>}
        | {- <TLFAE> <TLFAE>}
        | {= <TLFAE> <TLFAE>}
        | <id>
        | {if <TLFAE> <TLFAE> <TLFAE>}
        | {fun {<id> : <type>} <TLFAE>}
        | {<TLFAE> <TLFAE>}
<type> = number
       | boolean
       | {<type> -> <type>}
```

- = accepts only numbers (and produces booleans).
- `if` requires an expression of type boolean for the test and also that the "then" and the "else" branches have the same type.

3 Typed Lists

Add support for lists to the language by extending it with the following expressions:

```
<TLFAE> = ...
        | {cons <TLFAE> <TLFAE>}
        | {first <TLFAE>}
        | {rest <TLFAE>}
        | {nil <type>}
        | {empty? <TLFAE>}
<type> = ...
       | {listof <type>}
```

All of which work as they do in PLAI, with the exception of `nil`, which is just the empty list, but must be annotated with a type. For example, the type of `{nil number}` is `{listof number}`.

Require that all elements of a list have the same type, which you can ensure when typechecking `cons`.

It might seem unusual that `nil` must come with a type. Since the empty list can have many types, the programmer has to explicitly pick the one that agrees with the rest of the list.

4 Typed Vectors and Generic Operations

Add support for vectors (i.e., arrays) to the language by extending it with the following expressions:

```
<TLFAE> = ...
        | {make-vector <TLFAE> <TLFAE>}
        | {set <TLFAE> <TLFAE> <TLFAE>}
<type> = ...
       | {vectorof <type>}
```

In this language, vectors are *homogeneous*: all their elements must be of the same type.

`make-vector` takes as its first argument a number specifying the length of the vector to create, and as its second argument the value to initialize all the vector elements to.

`set` takes a vector as its first argument, the index to modify as its second, and the new element as its third. It's result in the element that originally was at that index.

Of course, one may also want to access the nth element of a vector, so it makes sense to provide an operation that does that. But of course, one may also be interested in accessing the nth element of a list. Instead of providing two separate operations, we will add a single *generic* operation—`get`—that supports both vectors and lists. In types parlance, this is called *ad-hoc polymorphism*.

In addition to this generic *get* operations, we will add a generic *length* operation which returns the length of a given list or vector.

```
<TLFAE> = ...
        | {get <TLFAE> <TLFAE>}
        | {length <TLFAE>}
```

`get` takes a list *or* a vector as its first argument, an index as its second, and returns the element at that index in the list or vector.

5 Typing Rules

Here are the precise typing rules for the new constructs. The part in square brackets on the right of the rule is the name of the rule, for your convenience. It does not carry any semantic meaning.

$$\frac{}{\Gamma \vdash \text{true} : \text{boolean}}\ \text{[true]}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{boolean}}\ \text{[false]}$$

$$\frac{\Gamma \vdash e_1 : \text{number} \qquad \Gamma \vdash e_2 : \text{number}}{\Gamma \vdash \{= e_1\ e_2\} : \text{boolean}}\ \text{[=]}$$

$$\frac{\Gamma \vdash e_1 : \text{boolean} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \{\text{if}\ e_1\ e_2\ e_3\} : \tau}\ \text{[if]}$$

$$\frac{}{\Gamma \vdash \{\text{nil}\ \tau\} : (\text{listof}\ \tau)}\ \text{[nil]}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : (\text{listof}\ \tau)}{\Gamma \vdash \{\text{cons}\ e_1\ e_2\} : (\text{listof}\ \tau)}\ \text{[cons]}$$

$$\frac{\Gamma \vdash e : (\text{listof}\ \tau)}{\Gamma \vdash \{\text{first}\ e\} : \tau}\ \text{[first]}$$

$$\frac{\Gamma \vdash e : (\text{listof}\ \tau)}{\Gamma \vdash \{\text{rest}\ e\} : (\text{listof}\ \tau)}\ \text{[rest]}$$

$$\frac{\Gamma \vdash e : (\text{listof}\ \tau)}{\Gamma \vdash \{\text{empty?}\ e\} : \text{boolean}}\ \text{[empty?]}$$

$$\frac{\Gamma \vdash e_1 : \text{number} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \{\text{make-vector}\ e_1\ e_2\} : (\text{vectorof}\ \tau)}\ \text{[make-vector]}$$

$$\frac{\Gamma \vdash e_1 : (\text{vectorof}\ \tau) \qquad \Gamma \vdash e_2 : \text{number} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \{\text{set}\ e_1\ e_2\ e_3\} : \tau}\ \text{[set]}$$

$$\frac{\Gamma \vdash e_1 : (\text{listof}\ \tau) \qquad \Gamma \vdash e_2 : \text{number}}{\Gamma \vdash \{\text{get}\ e_1\ e_2\} : \tau}\ \text{[get-list]}$$

$$\frac{\Gamma \vdash e_1 : (\text{vectorof}\ \tau) \qquad \Gamma \vdash e_2 : \text{number}}{\Gamma \vdash \{\text{get}\ e_1\ e_2\} : \tau}\ \text{[get-vector]}$$

$$\frac{\Gamma \vdash e : (\text{listof}\ \tau)}{\Gamma \vdash \{\text{length}\ e\} : \text{number}}\ \text{[length-list]}$$

$$\frac{\Gamma \vdash e : (\text{vectorof}\ \tau)}{\Gamma \vdash \{\text{length}\ e\} : \text{number}}\ \text{[length-vector]}$$

6 Errors

If `typecheck-expr` is passed an expression that isn't well-typed, then it should throw an error. The possible errors are as follows.

- Free identifiers count as type errors and should fail with a message that includes `"free identifier"`.
- Applications whose function position don't typecheck with a function type should fail with a message that inlcudes `"expected function"`.
- The test position of `if` must be a boolean. If not, your typechecker should raise an error whose message includes `"expected boolean"`.
- Some subexpressions must have type `number` and should fail with a message that includes `"expected number"`.
- Some subexpressions must have list types. If not, your typechecker should raise an error whose message includes `"expected list"`.
- Other subexpressions must have vector types. If not, your typechecker should raise an error whose message includes `"expected vector"`.
- Other subexpressions still may be of either a list type or a vector type. If not, your typechecker should raise an error whose message includes `"expected list or vector"`.
- All other type errors (e.g., function called with an argument of the wrong type) are the result of two types not being identical when they should. In such cases, raise an error whose message includes `"type mismatch"`.

If there are multiple type errors in a program, raising any of them is fine.

7 Handin Instructions

- Provide a definition of `typecheck-expr : TLFAE -> Type`, as above. ***DON'T FORGET THAT ONE.*** It's been a common mistake historically.
- Provide a definition of `parse : s-expression -> TLFAE`, as above.
- Provide a definition of `parse-type : s-expression -> Type`, as above.
- Have the 8 rules from the Provost's website (see homework 1 for more details).
- Submit your code via Canvas.
- Your submission must include your test cases; submissions without test cases will get a grade of 0.