The second project picks up where the last left off. Beginning with the AST you produced in Project #1, you are to perform a number of static checks on its correctness, annotate it with information about the meanings of identifiers, and perform some rewrites. Your job is to hand in a program (and its testing harness), including adequate internal documentation (comments), and a thorough set of test cases, which we will run both against your program and everybody else’s.

1 Summary

Your program is to perform the following processing:

1. Add a list of indexed declarations, as described in §3.

2. Decorate each id and type_var node by adding a declaration index that links it to a declaration in the list. This is also described in §4.

3. Rewrite allocation expressions to use new AST nodes that were not produced by the parser.

4. Enforce the language dialect described in subsequent sections.

The remaining sections describe these in more detail.

Full Python is a very dynamic language; one may insert new fields and methods into classes or even into individual instances of classes at any time. One may redefine functions, methods, modules, and classes at will. For this project, we will greatly restrict the language to give it static typing, and your project will infer those types in most places.
2 Input and Output

You can start either from a parser that we provide, or you can augment your own parser. In either case, the output from your program will look essentially like that from the first project, but with some additional annotations. We’ll augment pyunparse to show your annotations.

In the skeleton, we have separated the first phase from the second as two programs, with a Python script to connect them. You can substitute your solution to Project 1 for the first phase unchanged, or you can use ours. The second phase reads in ASTs produced from running the first phase twice—one from parsing the source program, once from parsing the standard prelude, which defines the built-in functions and classes.

3 Output Format

The output ASTs differ from input ASTs in these respects:

- Identifier nodes and type variables will have an extra annotation at the end:
  
  $$(id \ N \ name \ D) \quad (type\_var \ N \ name \ D)$$
  
  where $D \geq 1$ is an integer declaration index.

- Compilations will now have the syntax
  
  $$\text{Compilation} : '(',') \ "module" \ N \ Stmt* ')' \ Decl*$$
  
  The Decls, described in Table 1, represent declarations. They are indexed by the declaration indices used in id and type_var nodes, and appear in order according to their index.

This choice of declaration indices in identifiers or type variables must reflect the scope rules of the language: two occurrences of identifier or type variable $I$ will have the same decoration if they both are supposed to refer to the same thing. The scope of a type variable is the class or function that uses it in its type parameters or its parameter list or return type. Furthermore, the scope of type variables defined in a class header, like that of method names and instance variables, does not include any def statements in the class. That is, the definition of $T$ at line 1, below is not available inside of $f$ and $g$.

```python
class Foo of [$T$]:  # Line 1
    def f(self, x::$T): # $T does not refer to the same type as Line 1
        pass
    def g(self):
        x::$T = 3 # Illegal; $T from Line 1 is not visible
```

The programmer may not otherwise introduce type variables. For example, this trivial program is illegal, because $a$ is not defined.

```python
foo::$a = 3
```
There is one declaration index (and corresponding declaration node) for each distinct declaration in the program: each class definition, local variable, parameter, type variable, method definition, and instance variable. There are also declarations for the built-in types and functions in the standard prelude. Only declarations reachable from the AST need to appear in the outputted list. As a result there may be gaps in the index numbers. Table II shows the formats of the declaration nodes.

The set of declarations is not the same as a symbol table (or environment). It is an undifferentiated set of all declarations without regard to scopes, declarative regions, etc. You’ll need some entirely separate data structure (which you’ll never output) to keep track of the mappings of identifiers to declarations at various points in the program. Some declarations don’t correspond to anything you can point to or name in the program. For example, under our rules, the module name __main__ is not defined within your program, and references to it is an error, even though this module certainly exists and contains lots of definitions you can reference.

4 Rewriting

For the sake of the code generator (and to some extent, to simplify parts of semantic analysis), the program must perform several rewrites.

4.1 Initial “post-phase 1” rewrites

There are some initial rewrites that we have done for you, since they are straightforward and we might just as well have had you do them in the first pass. The skeleton file proj2/src/post1.cc implements these:

1. The AST from the parsing the standard prelude is inserted at the front of the module containing the source program. This is to avoid having to deal with module constructs in this project.

2. All “block” ASTs are removed and their children inserted directly into the surrounding class or function definition.

3. All “type_list” ASTs are removed and their children inserted directly into the surrounding “type” or “function_type” node.

4. All function definitions in which the return type is defaulted are supplied with a new type variable as the return type. This gives an explicit return type without being explicit about what it is. Thus, the function header

\[
\text{def f():} \\
\ldots
\]

might become

\[
\text{def f():$#12:} \\
\ldots
\]
(we use $\#n$ to notate anonymous type variables produced by the compiler as opposed to being written by the programmer.)

5. Likewise, all formal function parameters that do not have explicit types are also given anonymous type variables, so that every parameter has an explicit formal type. For example:

   ```python
def f(x):
   ...
```

might become

   ```python
def f(x: $\#14$):$\#15$:
   ...
```

6. All “def” nodes that represent methods in a class are changed to “method” nodes (with the same format). This makes it a little easier to write code that handles the slight differences between them.

Again, we’ve written these transformations for you. The rest you supply.

4.2 Identifying types

During parsing, you can’t always tell which identifiers represent types. For example, the $A$ in $A(3)$ could denote either a type or a value. Rewrite any id node that is a type with a type AST node containing that id node (if it is not already part of one, that is.)

4.3 Allocators

Whenever you encounter a “call” node whose first operand is a type (which is Python’s way of writing the Java or C++ new operator):

   
   \[
   (\text{call} \ N \ T \ E_1 \ldots E_n),
   \]

convert it to the expression

   
   \[
   (\text{call1} \ N \ (\text{id} \ N \ _\text{init}_-) \ (\text{new} \ N \ T) \ E_1 \ldots E_n)
   \]

and decorate the id node with a declaration index as if this method had actually been written explicity. The new node, call1, is just like call, but returns the value of its first argument rather than the value returned by the _init_ function. If a class does not have an _init_ method, one can’t allocate anything with it using the allocator syntax. Hence, for user-defined classes, you’ll usually want to explicitly include an _init_ method (it is not defaulted).
4.4 Attributes of classes
Whenever you encounter a node of the form

\((\text{attributeref } N \ E_1 I)\),

where \(E_1\) denotes a known class that defines \(I\) (an id node) as a method, replace the entire attributeref with \(I\), after assigning the appropriate declaration index to \(I\). Thus, after the Python class declaration

```python
class A(object):
    def f (self): ...
```

The statement

```python
g = A.f
```

becomes, in effect,

```python
g = f
```

but with \(f\) decorated with the appropriate declaration of method \(f\). It is an error for \(E_1\) to denote a type that is not known to define \(I\). \(E_1\) can also be a parameterized type (as in \texttt{PriorityQueue of [Int].push}, but the type parameters are ignored in that case.

If \(E_1\) denotes a class and \(I\) denotes an instance variable rather than a method, the attribute reference is erroneous.

5 Overloading

We’re extending Python to allow overloaded functions (and as a result, operators). The rule is that there can be multiple definitions of functions (excluding methods) within a declarative region, thus overloading them. It is an error to attempt to overload any other kind of entity other than a function.

Overloaded names are disambiguated on the basis of type rules. For example, this is legal in our dialect:

```python
def f():
    print "f()"
def f(x):
    print "f(x)"
f(3)
```

To keep things simple, we’ll still say that any declaration of a function in one region hides all those in enclosing regions, so that the following is illegal, even though no definitions of \(f\) inside \(g\) can possibly satisfy the call:
def f():
    pass

def g():
    def f(x):
        pass
    f()  # ERROR, the outer f is not visible.

As you saw in Project 1, operator expressions are converted into nodes with the same format as function calls, and in fact are treated just like function calls in the semantics. The function names in these expressions (e.g., \texttt{\_\_add\_\_}) are are defined by ordinary \texttt{defs} (usually in the standard prelude) and can therefore be overloaded as well.

\section{Types}

For this project, the possible types are either builtin types, user classes, or function types.

\subsection{Type representation}

Type variables, class, and function types are represented as in project \#1, but with \texttt{type\_list} nodes replaced by their children, as described in §4.1:

\begin{align*}
\text{(type N (id N type-name) types)} \\
\text{(function\_type N return-type argument-types).} \\
\text{(type\_var N type-name)}
\end{align*}

(All id and type\_var nodes here and below should also have appropriate declaration indices attached.) If we have the Python statements:

```python
class A:
    def f(self, x::int)::bool: ...
    x::A = A()
```

then the expression \texttt{A.f} has the type

```python
(function\_type 0 (type 0 (id 0 bool)))
   (type 0 (id 0 A))
   (type 0 (id 0 int)))
```

(the line-number attributes here are irrelevant).

Each identifier and expression has the \textit{most general} static type that is consistent with the type rules of the language (§8). As discussed in lecture, the most general type is one that is compatible with all choices of types that obey the type rules and incompatible with all others. For example, the function
def id(x):
    return x

has type $(t)\rightarrow t$, since id can take any type of argument and returns a value of the same type. On the other hand, the functions

def sub(x,y):
    return x-y
def intid(z::int):
    return z

have types $(\text{int},\text{int})\rightarrow \text{int}$ and $(\text{int})\rightarrow \text{int}$, because ‘-’ in our subset operates only on integers and the type rule for :: notations requires that $z$ have the type int.

7 Various Restrictions

Our Python dialect is a rather violent restriction of Python designed, among other things, to make the language statically typed.

1. We restrict ourselves to the following types:
   - int.
   - bool.
   - str (string).
   - range (type of xrange’s result. This is not the standard Python type name.)
   - list($T$): that is, lists all of whose elements have the same type.
   - tuple0(), tuple1($T_1$), tuple2($T_1$, $T_2$), tuple3($T_1$, $T_2$, $T_3$): These are tuples with known, constant numbers of elements having the types $T_i$. Yes, we only do the ones up to 3, but that’s enough to make the point.
   - dict($K$, $V$): Maps from a type $K$ to a type $V$. The type $K$ is restricted to be int, bool, or str.
   - User-defined classes.
   - Function types.

2. All methods (defined by defs that occur immediately within a class definition) are instance methods (there are no static methods), and all therefore have at least one parameter. The first parameter of a method has the enclosing class as its type. (The first parameter of a Python method corresponds to this in a Java program.)

3. class and def statements declare constants, which may not be assigned to. If a variable is assigned to in some declarative region (thus becoming a local variable or instance variable), its name may not then be defined by def or class statements immediately within that same region.
4. Likewise, classes, methods, and functions may not be redefined immediately within the
same declarative region (function, class, or file).

5. The only attributes of a class (things referenced with `.`) defined by a `class` declaration
in the program are instance variables explicitly assigned to in the body of the class
(outside of any methods), or methods defined by `def` immediately within the class
body. Thus, the only attributes of class C:

```python
class C(object):
    a = 3
    def f(self): ...
```

are `a` and `f`.

6. The scope of parameters, local variable declarations (assignments to local variables)
and `def`s that are nested inside other function bodies or classes includes the entire
declarative region that contains them (before and after the declaration, in other words).
In the case of classes, this declarative region does not include the bodies of methods
within those classes (so that, for example,

```python
class A(object):
    x = 3
    def f(self):
        if self.x > 0:   # OK
            return x      # ERROR: x is unknown here.
```

This is as in regular Python.)

7. The scope of outer-level declarations (those that are not nested inside a `def` or `class`
declaration) begins with the declaration and continues to the end of the program (ex-
cept where hidden). Thus, at the outer level, you may not use identifiers before their
definition, so that the program

```python
def f():
    print y
y = 3
```

is erroneous (`y` is used before it is declared by assignment.) However,

```python
def g():
    def h():
        print y
    y = 3
```

is fine, because in this case, `y` is nested in `g`.

8. All instances of identifiers and type variables in the program other than the identifiers
denoting operators (like `+`) in `binop`, `unop`, `compare` and `compare_left` nodes must have
known declarations.
9. The type rules must successfully supply types for all (sub)expressions. Furthermore, for each complete statement at the outer level, there may be no free, unbound type variables; types of global variables must be completely determined by the statement that assigns them.

10. Classes may not be used as values. The only valid uses for a class name \( C \) are for allocators (\( C() \)), type designators ('\( ::C \)'), or for fetching method attributes (as in \( C.f \)). If \( I \) denotes an instance variable of \( C \), then \( C.I \) is erroneous (only \( x.I \), where \( x \) is an expression that evaluates to an instance of type \( C \)). Builtin classes may not be used for allocators.

11. User-defined class instances may not be allocated unless the class has an explicit \_init\_ method (there is no default method).

12. In the control statements \( \text{if } C: \ldots \) and \( \text{while } C: \ldots \), and the conditional expression \( (E_1 \text{ if } C \text{ else } E_2) \), the condition \( (C) \) must have type \text{bool} \ (\text{in ordinary Python, it can have any type}). The standard prelude has a function \text{truth} \ that converts any Python value into \text{True} \ or \text{False} \ according to the usual rules for true values in Python.

8 Type Rules

The language subset is chosen so that type inference can assign types to all expressions and statically check the validity of all constructs. A correct program obeys the rules in Figure 1 which are in the style of rules illustrated in Lecture 23. Be careful; these are definitely not the same as in ordinary Python, restricting or disallowing many expressions. Most of the things missing from the table are handled by the rules for calls.

Your program should resolve types on one outer-level construct, in sequence. An outer-level construct is a statement, class definition, or \text{def} \ that is not part of another statement, class definition, or \text{def} \ (hence, a module is a sequence of outer-level constructs). Initially, each defined quantity has some unknown type (that is, its type is represented by a type variable). Applying the language rules to an outer-level construct with unification gives bindings for those unknown types (or indicates an error).

In Lecture, we did not deal with what happens with overloading. Consider an overloading of \( f \):

\[
\text{def } f(a::\text{int}):\text{int}: \ldots \\
\text{def } f(a::\text{str}):\text{str}: \ldots
\]

and a declaration:

\[
\text{def } g(y):
\begin{align*}
  x &= 3 \\
  x &= f(y)
\end{align*}
\]

We should be able to figure out that \( y \) must be an \text{int}. However, we can’t apply the call rule in Figure 1 until we know which \( f \) we are dealing with. To deal with case, use brute force.
That is, identify all the identifiers that are overloaded, and perform type inference with every combination of these overloadsings in turn. If type inference works for exactly one of these, then all is well. Otherwise, report an error. Let’s call this process “basic type inference.”

There is a notable complication in this process. Consider an expression such as x.y. Until we know the type of x, we cannot determine which method(s) or instance variable to use for y, and therefore we don’t know which type to use for the attribute y or (equivalently) for the expression x.y. I suggest that you handle this with more brute force. When you resolve identifiers and encounter an expression E.x, scan all the classes in the environment and add a declaration for each class member named x (ignoring for the moment the type of E). Then when you do basic type inference for a particular combination of identifier resolutions, make sure that one of your checks for E.x is that the class that x comes from unifies with the type of E.

9 The standard prelude

The term standard prelude refers to the definitions of all built-in names in a language. In our case, these can be described by a set of ordinary declarations in our Python dialect, and handled with (mostly) the same rules (I say “mostly” because some built-in types have special significance in the language; type str, for example, is the type of string literals.) The rewrite that we supply (see §4.1) prepends the AST for the standard prelude that to the rest of your program.

10 Running the program

For this project, the command line looks like one of these (square brackets indicate optional arguments):

```
./apyc --phase=2 -o OUTFILE SOURCE.py
./apyc --phase=2 SOURCE.py
```

We’ve supplied the script apyc. It will call the phase 1 (the parser, previously written) and phase 2 programs. The command lines from project 1 will still do the same thing. That is, phase=1 should just parse your program and not do semantic analysis. The -o switch indicates the output file. By default (the second form), the output file is SOURCE.dast (".dast" for “decorated AST”).

The actual program you write will be called proj2/apyc2. The apyc script will invoke it like this:

```
proj2/apyc2 -o OUTFILE PRELUDE.py PRELUDE-AST.ast \\
SOURCE.py  SOURCE-AST.ast
```

The two .py arguments are used only to supply file names in error messages; apyc2 only reads the .ast files.

Usually, the apyc script puts the two .ast inputs into temporary files, which are removed after the second phase exits. The -k option to apyc tells it to keep them around instead in a directory named SOURCE.d for debugging purposes.
terminates on an exception or produces incorrect output that you’d like to diagnose, first run

```bash
./apyc -o foo.dast --phase=2 -k foo.py
```

You’ll get a directory named `foo.d` containing files `pre.ast` and `prog.ast`. Now you can directly run

```bash
./apyc2 -o foo.dast lib/prelude.py foo.d/pre.ast foo.py foo.d/prog.ast
```

with the debugger.

11 What to turn in

The directory you turn in (under the name `proj2-n` in your `tags` directory) should contain two subdirectories, `proj1` and `proj2`. The first, `proj1`, may contain either your own project 1 solution or ours. Your `proj2` directory must contain the `apyc` script (we supply a version you can use or modify as you like) and a file `Makefile` that is set up, again as in the skeleton we supply, so that

```bash
make
```

(the default target) compiles your program;

```bash
make check
```

runs all your tests against your program; and finally,

```bash
make APYC=PROG check
```

runs all your tests against the program `PROG` (by default, in other words, `PROG` is your program, `./apyc`). Finally,

```bash
make clean
```

should remove all files that are regeneratable or unnecessary. The Makefiles in the `proj2` skeleton do all this. You may have to modify them to make them continue to work as required in the face of certain changes you make.

12 Using Git to Get Started and to Submit

If you have set up your local copy of your `git` team repository as in our setup directions, one member of your team can start your project from this point with the command:

```bash
$ cd team-repo
$ git fetch shared
$ git checkout -b proj2 shared/proj2
```
(As usual, first be sure that you have first committed any changes in the current working directory.) This starts and checks out a branch called proj2 and puts a single subdirectory called proj2 in it.

Now you have two choices. To use your version of Project 1, you can execute the following command, assuming that you have put your Project 1 in a branch called proj1:

```bash
$ git merge proj1    # Assumes that you are in team-repo
$ git push -u origin proj2
```

This will add your proj1 directory to your proj2 branch and push the result to your shared repository. There should not already be a proj2 directory in your proj1 branch when you do this.

Alternatively, you can use our proj1 solution with the following commands:

```bash
$ git merge shared/proj1-solution
$ git push -u origin proj2
```

Should we make modifications to our skeleton, you can incorporate them into your files (if desired) with the following steps, assuming you are in your proj2 branch:

```bash
$ git commit -a  # If needed.
$ git fetch shared
$ git merge shared/proj2  # To incorporate get changes to proj2 skeleton
```

If you are using our proj1 solution, you can update that with the command

```bash
$ git merge shared/proj1-solution
```

Do not do this if you are using your own proj1 directory! See the Project 1 handout for more details about merging.

We will test your program by first using it to translate a suite of correct Python programs (checking that your program exits cleanly with an exit code of 0), and we will check the translations by unparsing them (using pyunparse with switches that check that you've gotten the declarations right), running the resulting Python programs, and checking their output. Next, we will run your program against a suite of erroneous, and check that you produce an appropriate error message (its contents are not important as long as the form is as specified) and that your program exits with a standard error code (as produced by exit(1) in C++).

Not only must your program work, but it must also be well documented internally. At the very least, we want to see useful and informative comments on each method you introduce and each class.

## 13 Assorted Advice

What, you haven’t started yet? First, review the Python language, and start writing and revising test cases. You get points for thorough testing and documentation, and it should not be difficult to get them, so start immediately by augmenting your project 1 tests. We won’t give credit for tests that are simply duplicates of ours.
Again, be sure to ask us for advice rather than spend your own time getting frustrated over an impasse. By now, you should have your partners’ phone numbers at least. Keep in regular contact.

Be sure you understand what we provide. The skeleton classes actually do quite a bit for you. Make sure you don’t reinvent the wheel.

Do not feel obliged to cram all the checks that are called for here into one method! Keep separate checks in separate methods. To the extent possible, introduce and test them one at a time. In fact, this project is structured in such a way that you can break it down into a set of small problems, each implemented by a few methods that traverse the ASTs.

Keep your program neat at all times. Keep the formatting of your code correct at all times, and when you remove code, remove it; don’t just comment it out. It’s much easier to debug a readable program. Afraid that if you chop out code, you’ll lose it and not be able to go back? That’s what Subversion is for. Archive each new version when you get it to compile (or whenever you take a break, for that matter). This will allow you to go back to earlier versions at will.

Write comments for classes and functions before you write bodies, if only to clarify your intent in your own mind. Keep comments up to date with changes. Remember that the idea is that one should be able to figure how to use a function from its comment, without needing to look at its body.

You still haven’t started?
**Table 1:** Declaration nodes. The list of the declaration nodes for a program in order by index follows the AST for the program. In each case, $N$ is the declaration index, unique to each declaration node instance. $T$ is a type (represented as usual by an AST).

<table>
<thead>
<tr>
<th>Node</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(vardecl $N I P T$)</td>
<td>Variable named $I$. $P$ is the declaration index of the enclosing function (or module, for a global variable). $T$ defines its static type (see §6 below).</td>
</tr>
<tr>
<td>(typevardecl $N I T$)</td>
<td>A type variable named $I$ and bound to type $T$. For an unbound typevardecl $T$ is a type_var node whose declaration index is $N$ (it refers to itself).</td>
</tr>
<tr>
<td>(paramdecl $N I P K T$)</td>
<td>Parameter named $I$ of type $T$ defined as the $K^{th}$ parameter (numbering from 0) of the function whose declaration index is $P$.</td>
</tr>
<tr>
<td>(instancetypecl $N I P T$)</td>
<td>Instance variable named $I$ of type $T$ defined in the class with declaration index $P$.</td>
</tr>
<tr>
<td>(funcdecl $N I P T$ (index_list $m_1\cdots m_n$))</td>
<td>A <strong>def</strong>ed function (including instance methods for this project, since we don't use inheritance) named $I$ of type $T$, defined in a function, class, or module with declaration index $P$. The $m_i$ are the declaration numbers of local variables, parameters, and local def's defined in the body of the function. The parameters come first, in the order they appear in the formals.</td>
</tr>
<tr>
<td>(classdecl $N I$ (index_list $p_1\cdots p_n$) (index_list $m_1\cdots m_n'$))</td>
<td>Class declaration for class named $I$. The $p_i$ are the declaration numbers of the type parameters of the class (all type variables). The $m_i$ are the declaration numbers of the members of the class. Each should be listed in order of appearance in the source text of the class.</td>
</tr>
<tr>
<td>(moduledecl $N __main__$ (index_list $m_1\cdots m_n$))</td>
<td>Module declaration for the module __main__ (the only one in our project). The <strong>index_list</strong> gives the indices of declarations in the module, in the order they appear in the source.</td>
</tr>
<tr>
<td>Name</td>
<td>Construct</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Lists</td>
<td>[]</td>
</tr>
<tr>
<td></td>
<td>$[ \ E_1, E_2, \ldots \ ]$</td>
</tr>
<tr>
<td>Tuples</td>
<td>$(E_1, E_2, \ldots, E_n)$</td>
</tr>
<tr>
<td>Numerals</td>
<td>$0, 1, \ldots$</td>
</tr>
<tr>
<td>Strings</td>
<td>&quot;\ldots&quot;, r&quot;\ldots&quot;, \ldots&quot;</td>
</tr>
<tr>
<td>Constants</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
<tr>
<td>Logical</td>
<td>$E_1$ and $E_2$</td>
</tr>
<tr>
<td></td>
<td>$E_1$ or $E_2$</td>
</tr>
<tr>
<td>Call</td>
<td>$E_0(E_1, \ldots, E_n)$</td>
</tr>
<tr>
<td>Call1</td>
<td>$__\text{init__}(E_1, \ldots, E_n)$</td>
</tr>
<tr>
<td>Allocate</td>
<td>$C()$</td>
</tr>
<tr>
<td>Identifier</td>
<td>$I$</td>
</tr>
<tr>
<td>For</td>
<td>for $T$ in $E$: \ldots</td>
</tr>
<tr>
<td>Control</td>
<td>while $C$: \ldots</td>
</tr>
<tr>
<td></td>
<td>if $C$: \ldots</td>
</tr>
<tr>
<td></td>
<td>$E_1$ if $C$ else $E_2$</td>
</tr>
<tr>
<td></td>
<td>return $E$</td>
</tr>
<tr>
<td>Print</td>
<td>print $E_1, \ldots, E_n[__]$</td>
</tr>
<tr>
<td>Typed ids</td>
<td>$x::T$</td>
</tr>
</tbody>
</table>

Figure 1: Type rules for the subset, part I. In general, type variables $a$, $b$, etc., refer to fresh type variables for each instance of the construct. Type rules for other most operators (including arithmetic operators and subscripts) fall out automatically from the rules for calls.