1 Introduction

Our CS 164 project this year involves a typed dialect of a Python 2.5 subset. Here, we document the differences between real Python 2.5 and this dialect.

2 Restrictions

Our dialect imposes a rather stringent set of restrictions on full Python 2.5.

2.1 Lexical structure

L1. No Unicode strings (e.g., `u"Foo"`).

L2. No long integer literals. All integer literals must be in the range $[0, 2^{30}]$, and a literal equal to 2^{30} actually means -2^{30} (Yeah, it's an ugly compromise. Sue me).

L3. No imaginary numbers.

L4. No floating-point literals.

L5. The following words may not appear, either as keywords or identifiers. The statements or expressions that use them in full Python are not in our dialect: as assert del except exec from finally future global import lambda raise try with yield

2.2 Expressions

E1. No list comprehensions (such as `[x for x in xrange(0,10)]`), or generator expressions (such as `(x for x in xrange(0,10))`), just expression lists.

E2. No string conversions (backquotes).

E3. No extended slices (such as `s[1:n:2]`).

E4. No keyword, `*`, or `**` arguments. No default parameters.

E5. We don't use the obsolescent `<>` operator.

E6. No **lambda** expressions.

E7. No `|`, `&`, `^`, `<<`, `>>`, or `~` operators.

2.3 Statements.

S1. No **import** statements.

S2. Only class declarations with a default parent class, as in

```
class Foo:
    ...
```

S3. No decorators ('`@A`').

S4. Class declarations may not be nested inside any other construct.

S5. Function declarations may not appear within the statements of an **if**, **while**, or **for**.

S6. Function declarations may contain only simple identifiers as parameters, so that

```
def foo(x, (y, z)): ...
```

is illegal.

S7. No augmented assignment (e.g., `+=`).

S8. Target lists (in assignments and **for** statements) are not nested. One may not write

```
(a, (b, c)) = L        a, (b,c) = L
```

Instead, to get the same effect, one must use

```
(a, T) = L             a, T = L
(b, c) = T             b,c = T
```

Regardless of how it is written, the type of a target list is some kind of tuple. Thus, the assignment

```
[a, b] = [1, 2]
```

is erroneous; you can only assign a list to a list variable.

S9. No **global** statement.

S10. No `>>` clauses on **print** statements (all output goes to the standard output.)

S11. The identifiers `None`, `True`, and `False` are keywords, and may not be used as variables (in Id nodes). Thus, they may not be assigned to.

2.4 Built-in Types

B1. The built-in simple types are restricted to int bool str range. Here, `range` is the type of the result of `xrange`.

B2. The list type is `list of [`T`]`–that is, lists of items all of which have type T.

B3. The tuple types are

> `tuple0`: (whose only member is the empty tuple),
>
> `tuple1 of [`T_1`]`: single-element tuples whose sole element has type T_1,
>
> `tuple2 of [`T_1, T_2`]`: two-element tuples whose elements have types T_1 and T_2, respectively.
>
> `tuple3 of [`T_1, T_2, T_3`]`: three-element tuples whose elements have types T_1, T_2, and T_3, respectively.

That is, we only have tuples going to up to length 3.

B4. The dictionary type is `dict of [`T_1, T_2`]`, mapping items of type T_1 to type T_2.

B5. The type of an n-argument function is $(T_1, \ldots, T_n) \to T_0$, where T_0 is the return type and the T_i are the argument types.

2.5 Declarations

D1. All methods (defined by **def** statements that occur immediately within a class definition) are instance methods (there are no static methods), and all therefore have at least one parameter.

D2. **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.

D3. Likewise, classes, methods, and functions may not be redefined immediately within the same declarative region (function, class, or file).

D4. 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:

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

are `a` and `f`.

D5. The scope of parameters, local variable declarations (assignments to local variables) and **def** statements 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 or headers of methods within those classes, so that, for example,

```
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.

D6. The scope of an outer-level declaration (one that is not nested inside a **def** or **class** statement) begins with the declaration and continues to the end of the program (except where hidden). Thus, at the outer level, you may not use identifiers before their definition, so that the program

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

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

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

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

D7. All instances of identifiers and type variables in the program must have known declarations.

3 Types and Overloading

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 each *outer-level construct*, in sequence. An outer-level construct is a statement, class definition, or **def** that is not part of another statement, class definition, or **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, or

a function type all of whose constituents are type variables). Applying the language rules to an outer-level construct with unification gives bindings for those unknown types (or indicates an error).

Each outer-level definition of a variable must resolve to a type that contains no free type variables. So, for example, this is illegal as an outer-level definition:

```
x = []
```

since the type rules for `[]` say that its type is `list of [$t]`, where `$t` is some new type variable and the assignment gives it no binding. On the other hand,

```
x = [3]
y::list of [int] = []
```

are both fine, since after type resolution on either one, any type variables will be bound. A function defined with **def** may have free variables in its type, of course, since otherwise we'd have no polymorphic types.

The types determined for instance variable immediately inside a class definition must contain no free type variables except those in the class header. So:

```
class A of $T:
   x::list of $T = []  # T is in the class header
   y = 3               # y has type int.
   z = []              # z's type is eventually resolved...
   z[0:0] = [3]        # ... at this point.
```

If E is an expression whose type is `A of [`Q`]`, then to find the type of $E.x$, we substitute Q for `$T`, giving `list of` Q. It might not be obvious how to do this in the general case of an expression $E.x$, but it is not difficult. First, generate fresh type variables for the type parameters of E's type. Substitute those into the type that was determined for instance variable `x` when the class was resolved, and finally unify those fresh type variables with the type parameters of E.

Overloading The picture is complicated by overloading. Consider an overloading of `f`:

```
def f(a::int)::int: ...
def f(a::str)::str: ...
```

and a declaration:

```
def g(y):
    x = 3
    x = f(y)
```

We should be able to figure out that `y` must be an `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 overloadings in turn. If type inference works for exactly one of these, then all is well. Otherwise, there is a compilation error.

When dealing with an expression such as `x.y`, where until you know the type of `x`, you cannot determine which method(s) or instance variable to use for `y`, and therefore don't know which type to use for the attribute `y` or (equivalently) for the expression `x.y`, I again suggest you use brute force. Given an expression $E.x$, scan all the classes in the environment and consider each declaration of a class member named x as a possible candidate. 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.

Name	Construct	Type	Conditions
Lists	`[]`	list($a)	
	`[` $E_1, E_2, \ldots$ `]`	list($a)	E_i: $a, for all i
Tuples	$(E_1, E_2, \ldots, E_n)$	tuple$n$ $(T_1, \ldots, T_n)$	$E_i : T_i$, for all $1 \leq i \leq n$, where $0 \leq n \leq 3$.
Numerals	`0, 1, ...`	int	
Strings	`"...", r"...", ...`	str	
Constants	`None`	$a	
	`True`	bool	
	`False`	bool	
Logical	E_1 `and` E_2	$a	E_1: $a, E_2: $a
	E_1 `or` E_2	$a	E_1: $a, E_2: $a
Call	$E_0(E_1, \ldots, E_n)$	$a	E_0:($a1,...,$an)->$a, E_1:$a1,...,E_n:$an.
Call1	`__init__`$(E_1, \ldots, E_n)$	$a1	__init__: ($a1,...,$an)->$b , E_1:$a1,...,E_n:$an. This is used only by the special `call1` node described in the project 2 specification.
Allocate	C`()`	C	where C is a class (this is the `new` node described in the project 2 specification.)
Assignment	L `=` R	$a	L: $a, R: $a.
For	`for` T `in` E`:` `...`	Exactly the same rules as for T = `__choose__`(E) (See Note 2).	
Control	`while` C`:` `...`	—	C: bool
	`if` C`:` `...`	—	C: bool
	E_1 `if` C `else` E_2	$b	C: bool, E_1: $b, E_2: $b
	`return` E	—	E: T, where T is the enclosing function's return type.
Print	`print` $E_1, \ldots, E_n$`[,]`	—	E_i: $ai, $1 \leq i \leq n$.

Figure 1: Type rules for the subset. In general, type variables $a, $b, etc., refer to fresh type variables for each use of the rule.

Notes.

1. As you can see, there are no rules given for most operators (among them `+`, `-`, `*`, and other arithmetic operators, as well as subscripting and slicing). That's because all of these are equivalent to calls on specific functions defined in the standard prelude (such as `__add__(x, y)` for `x+y`), so that their typing rules fall out of the rules for functions.

2. The function `__choose__` is a convenient fiction. It isn't actually present at runtime, but has an (overloaded) definition in the standard prelude that provides a convenient way to define the typing rules of the **for** statement.

Name	Construct	Type	Conditions
Identifier	I	T	T is the type declared for I (See Note 3).
	$C.I$	T	As above, but here C is a class and I a method in it.
Typed ids	$x::T$	T	x: T
Instance variable references	$E.I$	See Note 4.	

Figure 2: Type rules for the subset (part 2). Identifiers.

Notes.

3. In effect, we can think of the type of each identifier as being a unique type variable. The type rules in these tables will typically bind this variable to specific types. For example, the rule for typed ids ($x::T$ above) explicitly equates the declared type of x with T.

4. For an instance variable, I, defined in a class C with type parameters $P_1, \ldots, P_n$, the type is that inferred for I when the definition of C is resolved, but with all free type variables replaced with fresh ones. E must have type C `of` `[`$T_1, \ldots, T_n$`]`, and the appropriate T_i must unify with the corresponding fresh variables in I.