CHAPTER 4

Data types and polymorphism

The main concepts in this chapter are:

- **polymorphism**: dynamic type checking and dynamic dispatch,
- **control flow**: computing different values depending on a conditional expression,
- **compile time versus run time**: the execution of your compiler that performs transformations of the input program versus the execution of the input program after compilation,
- **type systems**: identifying which types of values each expression will produce, and
- **heap allocation**: storing values in memory.

4.1. Syntax of $P_1$

The $P_0$ subset of Python only dealt with one kind of data type: plain integers. In this chapter, we add Booleans, lists, and dictionaries. We also add some operations that work on these new data types, thereby creating the $P_1$ subset of Python. The syntax for $P_1$ is shown in Figure 1. We give only the abstract syntax (i.e., assume that all ambiguity is resolved). Any ambiguity is resolved in the same manner as Python. In addition, all of the syntax from $P_0$ is carried over to $P_1$ unchanged.

A Python list is a sequence of elements. The standard Python interpreter uses an array (a contiguous block of memory) to implement a list. A Python dictionary is a mapping from keys to values. The standard Python interpreter uses a hashtable to implement dictionaries.

4.2. Semantics of $P_1$

One of the defining characteristics of Python is that it is a dynamically typed language. What this means is that a Python expression may result in many different types of values. For example, the following conditional expression might result in an integer or a list.

```python
>>> 2 if input() else [1, 2, 3]
```
key_datum ::= expression "::" expression
subscription ::= expression "[" expression "]"
expression ::= "True" | "False"
  | "not" expression
  | expression "and" expression
  | expression "or" expression
  | expression ">=" expression
  | expression ">" expression
  | expression "if" expression "else" expression
  | "[" expr_list "]"
  | "{" key_datum_list "}"
  | subscription
  | expression "is" expression
expr_list ::= ε
  | expression
  | expression "," expr_list
key_datum_list ::= ε
  | key_datum
  | key_datum "," key_datum_list
target ::= identifier
  | subscription
simple_statement ::= target "=" expression

**FIGURE 1.** Syntax for the $P_1$ subset of Python. (In addition to the syntax of $P_0$.)

In a statically typed language, such as C++ or Java, the above expression would not be allowed; the type checker disallows expressions such as the above to ensure that each expression can only result in one type of value.

Many of the operators in Python are defined to work on many different types, often performing different actions depending on the run-time type of the arguments. For example, addition of two lists performs concatenation.

```python
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

For the arithmetic operators, `True` is treated as if it were the integer 1 and `False` is treated as 0. Furthermore, numbers can be used in places where Booleans are expected. The number 0 is treated as `False` and everything else is treated as `True`. Here are a few examples:

```python
>>> False + True
```

Note that the result of a logic operation such as and and or does not necessarily return a Boolean value. Instead, $e_1$ and $e_2$ evaluates expression $e_1$ to a value $v_1$. If $v_1$ is equivalent to False, the result of the and is $v_1$. Otherwise $e_2$ is evaluated to $v_2$ and $v_2$ is the result of the and. The or operation works in a similar way except that it checks whether $v_1$ is equivalent to True.

A list may be created with an expression that contains a list of its elements surrounded by square brackets, e.g., $[3,1,4,1,5,9]$ creates a list of six integers. The nth element of a list can be accessed using the subscript notation $l[n]$ where $l$ is a list and $n$ is an integer (indexing is zero based). For example, $[3,1,4,1,5,9][2]$ evaluates to 4. The nth element of a list can be changed by using a subscript expression on the left-hand side of an assignment. For example, the following fixes the 4th digit of $\pi$.

```python
>>> x = [3,1,4,8,5,9]
>>> x[3] = 1
>>> print x
[3, 1, 4, 1, 5, 9]
```

A dictionary is created by a set of key-value bindings enclosed in braces. The key and value expression are separated by a colon. You can lookup the value for a key using the bracket, such as $d[7]$ below. To assign a new value to an existing key, or to add a new key-value binding, use the bracket on the left of an assignment.

```python
>>> d = {42: [3,1,4,1,5,9], 7: True}
>>> d[7]
True
>>> d[42]
[3, 1, 4, 1, 5, 9]
>>> d[7] = False
>>> d
{42: [3, 1, 4, 1, 5, 9], 7: False}
>>> d[0] = 1
>>> d[0]
1
```
With the introduction of lists and dictionaries, we have entities in the language where there is a distinction between identity (the is operator) and equality (the == operator). The following program, we create two lists with the same elements. Changing list x does not affect list y.

```python
>>> x = [1,2]
>>> y = [1,2]
>>> print x == y
True
>>> print x is y
False
>>> x[0] = 3
>>> print x
[3, 2]
>>> print y
[1, 2]
```

Variable assignment is shallow in that it just points the variable to a new entity and does not affect the entity previously referred to by the variable. Multiple variables can point to the same entity, which is called aliasing.

```python
>>> x = [1,2,3]
>>> y = x
>>> x = [4,5,6]
>>> print y
[1, 2, 3]
>>> y = x
>>> x[0] = 7
>>> print y
[7, 5, 6]
```

**Exercise 4.1.** Read the sections of the Python Reference Manual that apply to $P_1$: 3.1, 3.2, 5.2.2, 5.2.4, 5.2.6, 5.3.2, 5.9, and 5.10.

**Exercise 4.2.** Write at least ten programs in the $P_1$ subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs.

### 4.3. New Python AST classes

Figure 2 shows the additional Python classes used to represent the AST nodes of $P_1$. Python represents True and False as variables (using the Name AST class) with names ‘True’ and ‘False’. Python allows these names to be assigned to, but for $P_1$, you may assume that they cannot written to (i.e., like input). The Compare class is
for representing comparisons such as == and !=. The expr attribute of Compare is for the first argument and the ops member contains a list of pairs, where the first item of each pair is a string specifying the operation, such as '==', and the second item is the argument. For $P_1$ we are guaranteed that this list only contains a single pair. The And and Or classes each contain a list of arguments, held in the nodes attribute and for $P_1$ this list is guaranteed to have length 2. The Subscript node represents accesses to both lists and dictionaries and can appear within an expression or on the left-hand-side of an assignment. The flags attribute should be ignored for the time being.

```python
class Compare(Node):
    def __init__(self, expr, ops):
        self.expr = expr
        self.ops = ops

class Or(Node):
    def __init__(self, nodes):
        self.nodes = nodes

class And(Node):
    def __init__(self, nodes):
        self.nodes = nodes

class Not(Node):
    def __init__(self, expr):
        self.expr = expr

class List(Node):
    def __init__(self, nodes):
        self.nodes = nodes

class Dict(Node):
    def __init__(self, items):
        self.items = items

class Subscript(Node):
    def __init__(self, expr, flags, subs):
        self.expr = expr
        self.flags = flags
        self.subs = subs

class IfExp(Node):
    def __init__(self, test, then, else_):
        self.test = test
        self.then = then
        self.else_ = else_
```

**Figure 2.** The Python classes for $P_1$ AST nodes.

### 4.4. Compiling Polymorphism

As discussed earlier, a Python expression may result in different types of values and that the type may be determined during program execution (at run-time). In general, the ability of a language to allow multiple types of values to be returned from the same expression, or be stored at the same location in memory, is called polymorphism. The following is the dictionary definition for this word.

<table>
<thead>
<tr>
<th><strong>poly•mor•phism</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>noun</td>
</tr>
<tr>
<td>the occurrence of something in several different forms</td>
</tr>
</tbody>
</table>

The term “polymorphism” can be remembered from its Greek roots: “poly” means “many” and “morph” means “form”. Recall the following example of polymorphism in Python.
2 if input() else [1, 2, 3]

This expression sometimes results in the integer 2 and sometimes in the list [1, 2, 3].

```python
>>> 2 if input() else [1, 2, 3]
1
2
>>> 2 if input() else [1, 2, 3]
0
[1, 2, 3]
```

Consider how the following program would be flattened into a sequence of statements by our compiler.

```python
print 2 if input() else [1, 2, 3]
```

We introduce a temporary variable `tmp1` which could point to either an integer or a list depending on the input.

```python
tmp0 = input()
if tmp0:
    tmp1 = 2
else:
    tmp1 = [1, 2, 3]
print tmp1
```

Thinking further along in the compilation process, we end up assigning variables to registers, so we’ll need a way for a register to refer to either an integer or a list. Note that in the above, when we print `tmp1`, we’ll need some way of deciding whether `tmp1` refers to an integer or a list. Also, note that a list could require many more bytes than what could fit in a register.

One common way to deal with polymorphism is called **boxing**. This approach places all values on the heap and passes around pointers to values in registers. A pointer has the same size regardless of what it points to, and a pointer fits into a register, so this provides a simple solution to the polymorphism problem. When allocating a value on the heap, some space at the beginning is reserved for a tag (an integer) that says what type of value is stored there. For example, the tag 0 could mean that the following value is an integer, 1 means that the value is a Boolean, etc.

Boxing comes with a heavy price: it requires accessing memory which is extremely slow on modern CPUs relative to accessing values from registers. Suppose a program just needs to add a couple integers. Written directly in x86 assembly, the two integers would be stored in registers and the addition instruction would work directly
on those registers. In contrast, with boxing, the integers must be first loaded from memory, which could take 100 or more cycles. Furthermore, the space needed to store an integer has doubled: we store a pointer and the integer itself.

To speed up common cases such as integers and arithmetic, we can modify the boxing approach as follows. Instead of allocating integers on the heap, we can instead go ahead and store them directly in a register, but reserve a couple bits for a tag that says whether the register contains an integer or whether it contains a pointer to a larger value such as a list. This technique is somewhat questionable from a correctness perspective as it reduces the range of plain integers that we can handle, but it provides such a large performance improvement that it is hard to resist.

We will refer to the particular polymorphic representation suggested in these notes as pyobj. The file runtime.c includes several functions for working with pyobj, and those functions can provide inspiration for how you can write x86 assembly that works with pyobj. The two least-significant bits of a pyobj are used for the tag; the following C function extracts the tag from a pyobj.

```c
typedef long int pyobj;
#define MASK 3  /* 3 is 11 in binary */
int tag(pyobj val) { return val & MASK; }
```

The following two functions check whether the pyobj contains an integer or a Boolean.

```c
#define INT_TAG 0  /* 0 is 00 in binary */
#define BOOL_TAG 1  /* 1 is 01 in binary */
int is_int(pyobj val) { return (val & MASK) == INT_TAG; }
int is_bool(pyobj val) { return (val & MASK) == BOOL_TAG; }
```

If the value is too big to fit in a register, we set both tag bits to 1 (which corresponds to the decimal 3).

```c
#define BIG_TAG 3  /* 3 is 11 in binary */
int is_big(pyobj val) { return (val & MASK) == BIG_TAG; }
```

The tag pattern 10 is reserved for later use.

The following C functions in runtime.c provide a way to convert from integers and Boolean values into their pyobj representation. The idea is to move the value over by 2 bits (losing the top two bits) and then stamping the tag into those 2 bits.

```c
#define SHIFT 2
pyobj inject_int(int i) { return (i << SHIFT) | INT_TAG; }
pyobj inject_bool(int b) { return (b << SHIFT) | BOOL_TAG; }
```
The next set of C functions from runtime.c provide a way to extract an integer or Boolean from its pyobj representation. The idea is simply to shift the values back over by 2, overwriting the tag bits. Note that before applying one of these projection functions, you should first check the tag so that you know which projection function should be used:

```c
int project_int(pyobj val) { return val >> SHIFT; }
int project_bool(pyobj val) { return val >> SHIFT; }
```

The following C structures define the heap representation for big values. The hashtable structure is defined in the provided hashtable C library.

```c
eenum big_type_tag { LIST, DICT };

struct list_struct {
    pyobj* data;
    unsigned int len;
};
typedef struct list_struct list;

struct pyobj_struct {
    enum big_type_tag tag;
    union {
        struct hashtable* d;
        list l;
    } u;
};
typedef struct pyobj_struct big_pyobj;
```

When we grow the subset of Python to include more features, such as functions and objects, the alternatives within big_type_tag will grow as will the union inside of pyobj_struct.

The following C functions from runtime.c provide a way to convert from big_pyobj* to pyobj and back again.

```c
pyobj inject_big(big_pyobj* p) { return ((long)p) | BIG_TAG; }
big_pyobj* project_big(pyobj val)
    { return (big_pyobj*)(val & ~MASK); }
```

The inject_big function above reveals why we chose to use two bits for tagging. It turns out that on Linux systems, malloc always aligns newly allocated memory at addresses that are multiples of four. This means that the two least significant bits are always zero! Thus, we can use that space for the tag without worrying about destroying
the address. We can simply zero-out the tag bits to get back a valid address.

The runtime.c file also provides a number of C helper functions for performing arithmetic operations and list/dictionary operations on pyobj.

```c
int is_true(pyobj v);
void print_any(pyobj p);
pyobj input_int();
big_pyobj* create_list(pyobj length);
big_pyobj* create_dict();
pyobj set_subscript(pyobj c, pyobj key, pyobj val);
pyobj get_subscript(pyobj c, pyobj key);
big_pyobj* add(big_pyobj* x, big_pyobj* y);
int equal(big_pyobj* x, big_pyobj* y);
int not_equal(big_pyobj* x, big_pyobj* y);
```

You will need to generate code to do tag testing, to dispatch to different code depending on the tag, and to inject and project values from pyobj. We recommend accomplishing this by adding a new compiler pass after parsing and in front of flattening. For lack of a better name, we call this the ‘explicate’ pass because it makes explicit the types and operations.

### 4.5. The Explicate Pass

As we have seen in Section 4.4, compiling polymorphism requires a representation at run time that allows the code to dispatch between operations on different types. This dispatch is enabled by using tagged values.

At this point, it is helpful to take a step back and reflect on why polymorphism in $P_1$ causes a large shift in what our compilers must do as compared to compiling $P_0$ (which is completely monomorphic). Consider the following assignment statement:

\[(4.1) \quad y = x + y\]

As a $P_0$ program, when our compiler sees the $x + y$ expression at compile time, it knows immediately that $x$ and $y$ must correspond to integers at run time. Therefore, our compiler can select following x86 instruction to implement the above assignment statement.

```
addl x, y
```

This x86 instruction has the same run-time behavior as the above assignment statement in $P_0$ (i.e., they are semantically equivalent).
Now, consider the example assignment statement (4.1) again but now as a $P_1$ program. At compile time, our compiler has no way to know whether $x + y$ corresponds to an integer addition, a list concatenation, or an ill-typed operation. Instead, it must generate code that makes the decision on which operation to perform at run time. In a sense, our compiler can do less at compile now: it has less certain information at compile time and thus must generate code to make decisions at run time. Overall, we are trading off execution speed with flexibility with the introduction of polymorphic operations in our input language.

To provide intuition for this trade off, let us consider a real world analogy. Suppose you are planning a hike for some friends. There are two routes that you are considering (let’s say the two routes share the same initial path and fork at some point). Basically, you have two choices: you can decide on a route before the hike (at “plan time”) or you can wait to make the decision with your friends during the hike (at “hike time”). If you decide beforehand at plan time, then you can simplify your planning; for example, you can input GPS coordinates for the route on which you decided and leave your map for the other route at home. If you want to be more flexible and decide the route during the hike, then you have to bring maps for both routes in order to have sufficient information to make at hike time. The analogy to your compiler is that to be more flexible at run time ($\sim$ hike time), then your compilation ($\sim$ hike planning) requires you to carry tag information at run time ($\sim$ a map at hike time).

Returning to compiling $P_1$, Section 4.4 describes how we will represent the run-time tags. The purpose of the explicate pass is to generate the dispatching code (i.e., the decision making code). After the explicate pass is complete, the explicit AST that is produced will make explicit operations on integers and Booleans. In other words, all operations that remain will be apply to integers, Booleans, or big.pyobj*s. Let us focus on the polymorphic + operation in the the example assignment statement (4.1). The AST produced by the parser is as follows:

(4.2) \[
\text{Add}((\text{Name}('x'), \text{Name}('y'))) .
\]

We need to create an AST that captures deciding which “+” operation to use based on the run-time types of $x$ and $y$.

For $+$, we have three possibilities: integer addition, list concatenation, or type error. We want a case for integer addition, as we can implement that operation efficiently with an addl instruction. To decide whether we have list concatenation or error, we decide to leave
that dispatch to a call in runtime.c, as a list concatenation is expensive anyway (i.e., requires going to memory). The add function

\[ \text{big}_\text{pyobj}* \text{add}(\text{big}_\text{pyobj}* x, \text{big}_\text{pyobj}* y) \]

in runtime.c does exactly what is described here. To represent the two cases in an explicit AST, we will reuse the Add node for integer addition and a CallFunc node to add for the big_pyobj* addition. Take note that the Add node before the explicate pass represents the polymorphic + of \( P_1 \), but it represents integer addition in an explicit AST after the explicate pass. Another choice could have been to create an IntegerAdd node kind to make it clear that it applies only to integers.

Now that we have decided which node kinds will represent which + operations, we know that the explicit AST for expression (4.2) is informally as follows:

\[
\text{IfExp(}
\text{tag of Name('x') is 'int or bool'}
\text{and tag of Name('y') is 'int or bool'},
\]

convert back to 'pyobj'
\[
\text{Add(convert to 'int Name('x'), convert to 'int Name('y'))},
\]

\[
\text{IfExp(}
\text{tag of Name('x') is 'big'}
\text{and tag of Name('y') is 'big'},
\]

convert back to 'pyobj'
\[
\text{CallFunc(Name('add'))},
\]

\[\text{[convert to 'big Name('x'), convert to 'big Name('y')]},
\]

\[
\text{CallFunc(... abort because of run-time type error ...)}
\]

Looking at the above explicit AST, our generated code will at runtime look at the tag of the polymorphic values for \( x \) and \( y \) to decide whether it is an integer add (i.e., \text{Add}(...)) or a \text{big}_\text{pyobj}* add (i.e., \text{CallFunc}(... add, ...)).

What are these "convert" operations? Recall that at run time we need a polymorphic representation of values (i.e., some 32-bit value that can be an 'int', 'bool', 'list', or 'dict'), which we call \text{pyobj}. It is a \text{pyobj} that has a tag. However, the integer add at run time (which
corresponds to the `Add` AST node here at compile time) should take "pure" integer arguments (i.e., without tags). Similar, the `add` function call takes `big_p义务` arguments (not `pyobj`). We need to generate code that converts `pyobj` to other types at appropriate places. From Section 4.4, we have described how we get the integer, boolean, or `big_p义务` part from a `pyobj` by shifting or masking. Thus, for "convert to whatever" in the above, we need insert AST nodes that represent these type conversions. To represent these new type conversion operations, we recommend creating two new AST classes: `ProjectTo` that represents converting from `pyobj` to some other type and `InjectFrom` that represents converting to `pyobj` from some other type. Analogously, we create a `GetTag` AST class to represent tag lookup, which will be implemented with the appropriate masking.

Note that we might have been tempted to insert AST nodes that represent directly shifting or masking. While this choice could work, we choose to separate the conceptual operation (i.e., type conversion) from the actual implementation mechanism (i.e., shifting or masking). Our instruction selection phase will implement `ProjectTo` and `InjectFrom` with the appropriate shift or masking instructions.

As a compiler writer, there is one more concern in implementing the explicate pass. Suppose we are implementing the case for explicating `Add`, that is, we are implementing the transformation in general for

\[
\text{Add}((e_1, e_2))
\]

where \( e_1, e_2 \) are arbitrary subexpressions. Observe in the explicit AST example above, `Name('x')` corresponds to \( e_1 \) and `Name('y')` to \( e_2 \). Furthermore, observe that `Name('x')` and `Name('y')` each appear four times in the output explicit AST. If instead of `Names`, we have arbitrary expressions and duplicate them in the same manner, we run into correctness issues. In particular, if \( e_1 \) or \( e_2 \) are side-effecting expressions (i.e., include `input()`), then duplicating them would change the semantics of the program (e.g., we go from reading once to multiple times). Thus, we need to evaluate the subexpressions once before duplicating them, that is, we can bind the subexpressions to `Names` and then use the `Names` in their place.

We introduce a `Let` construct for this purpose:

\[
\text{Let}(\text{var}, \text{rhs}, \text{body})
\]

The `Let` construct is needed so that you can use the result of an expression multiple times without duplicating the expression itself, which would duplicate its effects. The semantics of the `Let` is that
class GetTag(Node):
    def __init__(self, arg):
        self.arg = arg

class InjectFrom(Node):
    def __init__(self, typ, arg):
        self.typ = typ
        self.arg = arg

class ProjectTo(Node):
    def __init__(self, typ, arg):
        self.typ = typ
        self.arg = arg

class Let(Node):
    def __init__(self, var, rhs, body):
        self.var = var
        self.rhs = rhs
        self.body = body

Figure 3. New internal AST classes for the output of the explicate pass.

The rhs should be evaluated and then assigned to the variable var. Then the body should be evaluated where the body can refer to the variable. For example, the expression

\[
\text{Add}(\text{Add}(\text{Add}(\text{Const}(1), \text{Const}(2)), \text{Const}(3)))
\]

should evaluate to the same value as

\[
\text{Let} (\text{Name}(x), \text{Add}(\text{Add}(\text{Const}(1), \text{Const}(2)), \text{Add}(\text{Name}(x), \text{Const}(3))))
\]

(i.e., they are equivalent semantically).

Overall, to represent the new operations in your abstract syntax trees, we recommend creating the new AST classes in Figure 3.

Exercise 4.3. Implement an explicate pass that takes a P_i AST with polymorphic operations and explicates it to produce an explicit AST where all such polymorphic operations have been transformed to dispatch code to monomorphic operations.

4.6. Type Checking the Explicit AST

A good way to catch errors in your compiler is to check whether the type of value produced by every expression makes sense. For
example, it would be an error to have a projection nested inside of another projection:

\[
\text{ProjectTo('int',}
\text{ProjectTo('int', InjectFrom('int', Const(1)))}
\]

The reason is that projection expects the subexpression to be a \text{pyobj}.

What we describe in this section is a type checking phase applied to the explicit AST produced as a sanity check for your explicate pass. The explicate pass can be tricky to get right, so we want to have way to detect errors in the explicate pass before going through the rest of the compiler. Note that the type checking that we describe here does not reject input programs at compile time as we may be used to from using statically-typed languages (e.g., Java). Rather, any type errors that result from using the checker that we describe here points to a bug in the explicate pass.

It is common practice to specify what types are expected by writing down an “if-then” rule for each kind of AST node. For example, the rule for \text{ProjectTo} is:

For any expression \(e\) and any type \(T\) selected from the set \{ \text{int}, \text{bool}, \text{big} \}, if \(e\) has type \text{pyobj}, then
\[
\text{ProjectTo}(T, e) \text{ has type } T.
\]

It is also common practice to write “if-then” rules using a horizontal line, with the “if” part written above the line and the “then” part written below the line.

\[
\begin{align*}
\text{If} & \hspace{1em} \text{e has type pyobj} & \hspace{1em} T \in \{ \text{int, bool, big} \} \\
\text{Then} & \hspace{1em} \text{ProjectTo}(T, e) \text{ has type } T
\end{align*}
\]

Because the phrase “has type” is repeated so often in these type checking rules, it is abbreviated to just a colon. So the above rule is abbreviated to the following.

\[
\begin{align*}
\text{e : pyobj} & \hspace{1em} T \in \{ \text{int, bool, big} \} \\
\text{ProjectTo}(T, e) & : T
\end{align*}
\]

The \text{Let(var, rhs, body)} construct poses an interesting challenge. The variable \text{var} is assigned the \text{rhs} and is then used inside \text{body}. When we get to an occurrence of \text{var} inside \text{body}, how do we know what type the variable will be? The answer is that we need a dictionary to map from variable names to types. A dictionary used for this purpose is usually called an \text{environment} (or in older books, a \text{symbol table}). The capital Greek letter \text{gamma}, written \(\Gamma\), is typically used for referring to environments. The notation \(\Gamma, x : T\) stands for
making a copy of the environment $\Gamma$ and then associating $T$ with the variable $x$ in the new environment. The type checking rules for Let and Name are therefore as follows.

\[
\frac{e_1 : T_1 \in \Gamma \quad e_2 : T_2 \in \Gamma, x : T_1}{\text{Let}(x, e_1, e_2) : T_2 \in \Gamma} \quad \frac{\Gamma[x] = T}{\text{Name}(x) : T \in \Gamma}
\]

Type checking has roots in logic, and logicians have a tradition of writing the environment on the left-hand side and separating it from the expression with a turn-stile ($\vdash$). The turn-stile does not have any intrinsic meaning per se. It is punctuation that separates the environment $\Gamma$ from the expression $e$. So the above typing rules are commonly written as follows.

\[
\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{Let}(x, e_1, e_2) : T_2} \quad \frac{\Gamma[x] = T}{\Gamma \vdash \text{Name}(x) : T}
\]

Overall, the statement $\Gamma \vdash e : T$ is an example of what is called a judgment. In particular, this judgment says, “In environment $\Gamma$, expression $e$ has type $T$.” Figure 4.4 shows the type checking rules for all of the AST classes in the explicit AST.

**Exercise 4.4.** Implement a type checking function that makes sure that the output of the explicate pass follows the rules in Figure 4.4. Also, extend the rules to include checks for statements.

### 4.7. Update Expression Flattening

The output AST from the explicate pass contains a number of new AST classes that were not handled by the flatten function from chapter 4. The new AST classes are IfExp, Compare, Subscript, GetTag, InjectFrom, ProjectTo, and Let. The Let expression simply introduces an extra assignment, and therefore no Let expressions are needed in the output. When flattening the IfExp expression, I recommend using an If statement to represent the control flow in the output. Alternatively, you could reduce immediately to labels and jumps, but that makes liveness analysis more difficult. In liveness analysis, one needs to know what statements can precede a given statement. However, in the presence of jump instructions, you would need to build an explicit control flow graph in order to know the preceding statements. Instead, I recommend postponing the reduction to labels and jumps to after register allocation, as discussed below in Section 4.10.

**Exercise 4.5.** Update your flatten function to handle the new AST classes. Alternatively, rewrite the flatten function into a visitor
\[
\begin{array}{cccc}
\text{n is an integer} & \text{b is a Boolean} & \Gamma[x] = T \\
\Gamma \vdash \text{Const}(n) : \text{int} & \Gamma \vdash \text{Const}(b) : \text{bool} & \\
\Gamma \vdash e_1 : \text{int} & \Gamma \vdash e_2 : \text{int} & \Gamma \vdash e : \text{int} & \Gamma \vdash \text{Name}(x) : T \\
\Gamma \vdash \text{Add}(e_1, e_2) : \text{int} & \Gamma \vdash \text{UnarySub}(e) : \text{int} & \\
\Gamma \vdash e_1 : \text{bool} & \Gamma \vdash e_2 : T & \Gamma \vdash e_3 : T & \\
\Gamma \vdash \text{IfExp}(e_1, e_2, e_3) : T & \\
\Gamma \vdash e_1 : T & \Gamma \vdash e_2 : T & \Gamma \vdash e : T & \Gamma \vdash \text{InjectFrom}(T, e) : \text{pyobj} & \\
T \in \{\text{int, bool, big}\} & T \in \{\text{int, bool, big}\} & & & \\
\Gamma \vdash \text{GetTag}(e) : \text{int} & \Gamma \vdash \text{Compare}(e_1, [(\text{is}, e_2)]) : \text{bool} & \\
\Gamma \vdash e_1 : T & \Gamma \vdash e_2 : T & \Gamma \vdash e : \text{pyobj} & \Gamma \vdash \text{Compare}(e_1, [(\text{op}, e_2)]) : \text{bool} & \\
& & T \in \{\text{int, bool}\} & \text{op} \in \{\text{==, !=}\} & \\
& & \Gamma \vdash \text{Subscript}(e, i) : \text{pyobj} & & \\
\end{array}
\]

Figure 4. Type checking rules for expressions in the explicit AST.

class and then create a new visitor class that inherits from it and that implements visit methods for the new AST nodes.

4.8. Update Instruction Selection

The instruction selection phase should be updated to handle the new AST classes `If`, `Compare`, `Subscript`, `GetTag`, `InjectFrom`, and `ProjectTo`. Consult Appendix 6.4 for suggestions regarding which x86 instructions to use for translating the new AST classes. Also, you will need to update the function call for printing because you should now use the `print_any` function.

Exercise 4.6. Update your instruction selection pass to handle the new AST classes.
4.9. Update Register Allocation

Looking back at Figure 5, there are several sub-passes within the register allocation pass, and each sub-pass needs to be updated to deal with the new AST classes.

In the liveness analysis, the most interesting of the new AST classes is the If statement. What liveness information should be propagated into the “then” and “else” branch and how should the results from the two branches be combined to give the result for the entire If? If we could somehow predict the result of the test expression, then we could select the liveness results from one branch or the other as the results for the If. However, it’s impossible to predict this in general (e.g., the test expression could be \texttt{input(0)}), so we need to make a conservative approximation: we assume that either branch could be taken, and therefore we consider a variable to be live if it is live in either branch.

The code for building the interference graph needs to be updated to handle If statements, as does the code for finding all of the local variables. In addition, you need to account for the fact that the register \texttt{a1} is really part of register \texttt{eax} and that register \texttt{cl} is really part of register \texttt{ecx}.

The graph coloring algorithm itself works on the interference graph and not the AST, so it does not need to be changed.

The spill code generation pass needs to be updated to handle If statements and the new x86 instructions that you used in the instruction selection pass.

Similarly, the code for assigning homes (registers and stack locations) to variables must be updated to handle If statements and the new x86 instructions.

4.10. Removing Structured Control Flow

Now that register allocation is finished, and we no longer need to perform liveness analysis, we can lower the If statements down to x86 assembly code by replacing them with a combination of labels and jumps. The following is a sketch of the transformation from If AST nodes to labels and jumps.

\begin{verbatim}
if x:
    then instructions
else:
    else instructions
\end{verbatim}
Exercise 4.7. Write a compiler pass that removes If AST nodes, replacing them with combinations of labels and jumps.

4.11. Updates to Print x86

You will need to update the compiler phase that translates the x86 intermediate representation into a string containing the x86 assembly code, handling all of the new instructions introduced in the instruction selection pass and the above pass that removes If statements.

Putting all of the above passes together, you should have a complete compiler for $P_1$.

Exercise 4.8. Extend your compiler to handle the $P_1$ subset of Python. You may use the parser from Python’s compiler module, or for extra credit you can extend your own parser. Figure 5 shows the suggested organization for your compiler.
Figure 5. Overview of the compiler organization.
CHAPTER 5

Functions

The main ideas in this chapter are:

- **first-class functions**: functions are values that can be passed as arguments to other functions, returned from functions, stored in lists and dictionaries, assigned to variables, etc.

- **lexical scoping of variables**: function bodies can refer to variables that are defined in the scope surrounding where the function is created.

5.1. Syntax of $P_2$

We introduce two constructs for creating functions, the `def` statement and the `lambda` expression. We also add an expression for calling a function with some arguments. To keep things manageable, we leave out function calls with keyword arguments. The concrete syntax of the $P_2$ subset of Python is shown in Figure 1. Figure 2 shows the additional Python classes for the $P_2$ AST.

```
expression ::= expression "(" expr_list ")" | "lambda" id_list ":" expression
id_list ::= ε | identifier | identifier ":" id_list
simple_statement ::= "return" expression
statement ::= simple_statement | compound_stmt
compound_stmt ::= "def" identifier "(" id_list ")" ":" suite
suite ::= "\n" INDENT statement+ DEDENT
module ::= statement+
```

FIGURE 1. Concrete syntax for the $P_2$ subset of Python.
(In addition to that of $P_1$.)

5.2. Semantics of $P_2$

Functions provide an important mechanism for reusing chunks of code. If there are several places in a program that compute the same thing, then the common code can be placed in a function and
then called from many locations. The example below defines and calls a function. The \texttt{def} statement creates a function and gives it a name.

\begin{verbatim}
>>> def sum(l, i, n):
...     return l[i] + sum(l, i + 1, n) if i != n \n...     else 0
...
>>> print sum([1,2,3], 0, 3)
6
>>> print sum([4,5,6], 0, 3)
15
\end{verbatim}

Functions are \textit{first class} which means they are treated just like other values: they may be passed as arguments to other functions, returned from functions, stored within lists, etc. For example, the \texttt{map} function defined below has a parameter \texttt{f} that is applied to every element of the list \texttt{l}.

\begin{verbatim}
>>> def map(f, l, i, n):
...     return [f(l[i])] + map(f, l, i + 1, n) if i != n else []
\end{verbatim}
Suppose we wish to square every element in an array. We can define a square function and then use map as follows.

```python
>>> def square(x):
...    return x * x
...
>>> print map(square, [1,2,3], 0, 3)
[1, 4, 9]
```

The `lambda` expression creates a function, but does not give it a name. Anonymous functions are handy in situations where you only use the function in one place. For example, the following code uses a `lambda` expression to tell the `map` function to add one to each element of the list.

```python
>>> print map(lambda x: x + 1, [1,2,3], 0, 3)
[2, 3, 4]
```

Functions may be nested within one another as a consequence of how the grammar is defined in Figure 1. Any statement may appear in the body of a `def` and any expression may appear in the body of a `lambda`, and functions may be created with statements or expressions. Figure 3 shows an example where one function is defined inside another function.

```python
>>> def f(x):
...    y = 4
...    return lambda z: x + y + z
...
>>> f1 = f(1)
>>> print f1(3)
8
```

**Figure 3.** An example of a function nested inside another function.

A function may refer to parameters and variables in the surrounding scopes. In the above example, the `lambda` refers to the `x` parameter and the `y` local variable of the enclosing function `f`.

One of the trickier aspects of functions in Python is their interaction with variables. A function definition introduces a new scope, so a variable assignment within a function also declares that variable within the function’s scope. So, for example, in the following code, the scope of the variable `a` is the body of function `f` and not the global scope.
>>> def f():
    ...    a = 2
    ...    return a
>>> f()
2
>>> a

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'a' is not defined

Python's rules about variables can be somewhat confusing when a variable is assigned in a function and has the same name as a variable that is assigned outside of the function. For example, in the following code the assignment \( a = 2 \) does not affect the variable \( a \) in the global scope but instead introduces a new variable within the function \( g \).

```python
>>> a = 3
>>> def g():
    ...    a = 2
    ...    return a
>>> g()
2
>>> a
3

An assignment to a variable anywhere within the function body introduces the variable into the scope of the entire body. So, for example, a reference to a variable before it is assigned will cause an error (even if there is a variable with the same name in an outer scope).

```python
>>> a = 3
>>> def h():
    ...    b = a + 2
    ...    a = 1
    ...    return b + a
>>> h()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in h
UnboundLocalError: local variable 'a' referenced before assignment
```

**Exercise 5.1.** Write five programs in the \( P_2 \) subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs.