

# Assignment 5: variables and control flow

ECEN 4553 & 5013, CSCI 4555 & 5525  
Prof. Jeremy G. Siek

October 7, 2007

(This assignment was too long. For next year, introduce the new language features and do iterative type analysis in the first week and do SSA conversion in the second week. -Jeremy)  
(Break and continue are extra hard to deal with so I'm removing them. -Jeremy)  
The main concepts for this week are

- Variable binding.
- Using recursive functions to analyze the program.
- Translating from one intermediate language to another.
- Extending type analysis to deal with variables and control flow. This uses static single-assignment form and iterative type analysis.

## 1 Implementing $P_2$

### 1.1 Syntax of $P_2$

```
identifier ::= # see the Python Reference Manual
expression ::= identifier
simple_statement ::= "pass"
                | (identifier "=")+ expression
                | expression
compound_statement ::= "if" expression ":" suite "else" ":" suite
                    | "while" expression ":" suite
stmt_list ::= simple_stmt (";" simple_stmt)* [";"]
statement ::= stmt_list NEWLINE | compound_stmt
suite ::= stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
```

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

Figure 1 shows the concrete syntax for the subset  $P_2$ . This subset adds variables, assignment, and control flow.

### 1.2 Semantics of $P_2$ and compiling to C

Compiling the variables and assignment is the more challenging aspect because in Python, there are no variable declarations, as there are in C. For example, the following program assigns the value 3 to the variable  $x$  and then prints out its contents.

```
>>> x = 3
>>> print x
3
```

The analogous program in C would need to first declare the variable `x` and then assign to it as follows.

```
int x;
x = 3;
printf("%d\n", x);
```

The compiler needs to analyze the Python program and figure out where to insert the variable declarations in the output C program.

**Exercise 1.1.** (Nothing to turn in here.) Read the sections of the Python Reference Manual that apply to  $P_2$ : 4.1 Naming and binding 6.1 expression statements, 6.3 assignment, 6.4 pass, 6.10 break, 6.11 continue, 7.1 if, and 7.2 while.

**Exercise 1.2.** (This should be included in what you turn in.) 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. Run the programs using the standard python interpreter. Add your test programs to the test directory.

Figure 2 shows the Python classes for the  $P_2$  AST nodes. The `If` class has two attributes, the `tests` attribute that is a sequence of pairs, where the first item is an expression (the condition) and the second item is a statement. The `else_` attribute is a statement. The `While` class has the attribute `test` (an expression), `body` (a statement), and `else_` (a statement). In the `Assign` class, the `nodes` attribute contains a list of left-hand sides (because assignment may be chained, as in `a = b = 2`) where each left-hand side is represented by a `AssName` node (There are other kinds of assignment nodes in Python, but we don't have to worry about them yet.) The `expr` attribute of `Assign` is the right-hand side expression. You can ignore the `flags` attribute of `AssName` for now, as it is used to distinguish between the use of `AssName` for assignment and variable deletion.

In the previous chapters, we output the C code directly from the Python AST. That approach works well when there is a close node-to-node correspondence between the Python AST nodes and the C code to be generated. However, for this chapter we need to deal with variables: variables must be declared before they are used in C whereas in Python there are no variable declarations. Thus, we need to add in variable declaration nodes before generating the C code. We define a new AST node `VarDecl`.

```
class VarDecl(Node):
    def __init__(self, name, type):
        self.name = name
        self.type = type
```

Figure 3 shows the architecture of the compiler for this chapter.

**Exercise 1.3.** (Don't turn this in yet.) Write a function that analyzes a  $P_2$  program and builds up a new program with `VarDecl` nodes inserted at the appropriate locations for every variable that is used in the program. You will find the Python `sets` module helpful.

**Exercise 1.4.** (Don't turn this in yet.) Extend your Python to C compiler to handle  $P_2$ . Test your compiler against the new test programs for  $P_2$  in addition to the tests for  $P_0$  and  $P_1$ . To keep things

```

class Discard(Node):
    def __init__(self, expr):
        self.expr = expr
class If(Node):
    def __init__(self, tests, else_):
        self.tests = tests
        self.else_ = else_
class While(Node):
    def __init__(self, test, body, else_):
        self.test = test
        self.body = body
        self.else_ = else_
class Pass(Node):
    def __init__(self):
        pass
class Assign(Node):
    def __init__(self, nodes, expr):
        self.nodes = nodes
        self.expr = expr
class AssName(Node):
    def __init__(self, name, flags):
        self.name = name
        self.flags = flags

```

Figure 2: The Python classes for  $P_2$  AST nodes.

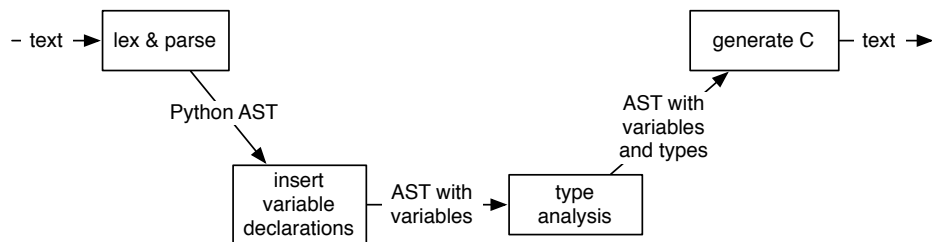


Figure 3: Diagram of passes in the part 1 compiler for  $P_2$ .

simple with respect to type analysis, just use the `pyobj` union type for all variables. In Section 2 of this assignment you will be doing more sophisticated type analysis to improve on this.

For extra credit, write the parser yourself using PLY, building on your parser for  $P_1$ . You will need to handle the peculiar indentation rules of Python. Look in the PLY distribution, in the directory `example/GardenSnake`, for code that will help you do this.

## 2 Type analysis and specialization for $P_2$

In Section 1 we simply used `pyobj` as the type for every variable. However, this approach produces poor code. For example, the variable `x` in the following code is assigned an integer, so we should generate C code with `x` declared as an integer.

```
x = 4
print x + 5
```

The following is the C code generated from Section 1 compared to what we would like to produce.

```
// Generated from the compiler for Section 1
pyobj x;
x = make_int(4);
print_any(add(x, make_int(5)));

// Goal for Section 2
int x;
x = 4;
printf("%d\n", x + 3);
```

Type analysis in the presence of variables is complicated because, in Python, a variable may be assigned values of different types. In the following code, variable `x` is first assigned an integer and later assigned a floating point number.

```
x = 3
y = x + 1
x = x + 0.14159
z = x / 2
```

One way to handle the above example is to split the variable `x` into multiple variables, each playing the role of `x` in different regions of the program. The following shows the above code transformed to split variable `x` into variables `x_0` and `x_1`.

```
x_0 = 3
y = x_0 + 1
x_1 = x_0 + 0.14159
z = x_1 / 2
```

Now the variable `x_0` can be given the type `int` and `x_1` can be given the type `float`. This resulting code is in what is called *Static Single Assignment (SSA)* form [1, 3, 5] because each variable is only assigned to at a single place in the code. (There is a chapter on SSA form in the recommended textbook [2].)

To convert a program to SSA form you traverse the program, and when you encounter an assignment, you replace it with an assignment to a new variable with its version number incremented by one. For each variable, you maintain a dictionary that maps each variable to its current version number. When you encounter a variable in an expression, you replace the variable with

its name with the version number appended. So, for example, when you see  $x = 3$  you transform it to  $x_0 = 3$  and add  $x_0$  to the dictionary. When you see  $x + 1$  you transform it to  $x_0 + 1$ .

For straight-line code in SSA form, type analysis is straightforward. You simply maintain a dictionary mapping variables to their type.

1. When you encounter a variable in an expression, look up the variable's type in the dictionary and set the type attribute. If the variable is not in the dictionary, then return the 'undefined' type.
2. When you encounter an assignment, perform type analysis on the right-hand side and then add to the dictionary an entry mapping the assigned variable to the type of the right-hand side.

Here's the C code that can be generated from the above. We are able to give each variable a specific type (not pyobj).

```
int x_0;
int y;
float x_1;
float z;
x_0 = 3;
y = x_0 + 1;
x_1 = x_0 + 0.14159;
z = x_1 / 2;
```

The language  $P_2$  also includes `if` statements and `while` loops, which introduces some complication in the SSA form and makes the type analysis more difficult. Consider the following program in which the variable `x` is assigned values of different types in two branches of the `if`.

```
if randint(0,1):
    x = 42
else:
    x = 3.14159
print x
```

SSA form deals with `if` statements by inserting a  $\phi$  node following the `if` for each variable that is assigned in the different branches. The  $\phi$  expression returns the value of the variable corresponding to the branch that was actually taken at runtime.

```
if randint(0,1):
    x_0 = 42
else:
    x_1 = 3.14159
x_2 =  $\phi(x_0, x_1)$ 
print x_2
```

The following C code is generated from the above. The  $\phi$  expression is implemented by adding an assignment to `x_2` at the end of each branch. Even with the SSA transformation, we still have to give `x_2` the type `pyobj` because we just don't know exactly what type it will hold at runtime.

```
int x_0;
float x_1;
pyobj x_2;

if (randint(0,1)) {
```

```

x_0 = 42;
x_2 = make_int(x_0);
} else {
x_1 = 3.14159;
x_2 = make_float(x_1);
}
print_any(x_2);

```

The translation of `while` loops to SSA form follows similar ideas to that of the `if` statement,  $\phi$  nodes are inserted where there is a join in the control flow. For a `while` loop, the join is at the head of the loop because control can enter the head of the loop from the preceding statement or from the end of the loop. Consider the following Python code.

```

x = 0
y = 1
z = x + 2
while x < 10:
    x = x + y
    y = z + 0.5
    x = x + 1

```

The SSA form for a `while` loop requires some fiddling with the control structure so that we have a place to put the  $\phi$  nodes. They must come before the loop test is evaluated but they still need to appear inside the loop. The loop is changed to an unconditional `while` and an `if` statement is inserted to check the condition and `break` out of the loop if the it is false.

```

x_0 = 0
y_0 = 1
while True:
    x_1 =  $\phi(x_0, x_2)$ 
    y_1 =  $\phi(y_0, y_2)$ 
    if x_1 < 10:
        x_2 = x_1 + y_1
        y_2 = 0.5
    else:
        break

```

Now let's consider how to perform type analysis on this loop. The major hurdle is that we cannot determine the type of a variable with a single-pass over the AST. Consider the assignment  $x_1 = \phi(x_0, x_2)$ . We have not yet analyzed the assignment for  $x_2$  so we can't really know what the type of  $x_1$  should be. In fact, there is a circular dependency because the later assignment  $x_2 = x_1 + y_1$  depends on  $x_1$ !

The way out of this conundrum is to make multiple passes over the AST, with each pass computing an ever-improving approximation of the correct result [4, 7]. What does it mean to have an approximation of a type? In general, we want to introduce types that represent varying degrees of information. At the bottom, we create a type that represents no information; let's call it `undefined`. We can use the type `int` for when we know that an integer value may flow into the variable, and similarly for `float` and `bool`. Suppose that we learn that both an integer and a floating point number may flow into a variable. A common way to handle this situation is to use type sets, such as `{int, float}`. However, for our purposes type sets are overkill, and we can just use the set of all types: `pyobj`. Figure 4 shows the types arranged according to their information content, with more informative types appearing higher in the figure. Such an arrangement of elements is called a *lattice* [6], though we will not go into the formal requirements of a lattice here. At the beginning

of the analysis, we create a dictionary that maps all variables to the type `undefined`. As the analysis iterates the variables will be updated with better approximations. Let  $F$  represent the change that each pass makes to the dictionary, so if  $d_0$  is the initial dictionary,  $d_1 = F(d_0)$  and in general after the  $i$ th iteration  $d_i = F(d_{i-1})$ .

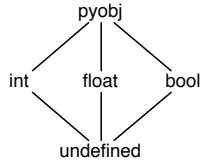


Figure 4: Types arranged according to information content (more informative is higher).

The next challenge is to figure out when to stop making passes over the AST, and how to ensure that we can stop after a finite number of steps. We can stop iterating the analysis when the results no longer change. That is, when  $F(d_i) = d_i$ . The dictionary  $d_i$  is called the *fixed point* of  $F$ . To ensure that we eventually reach a fixed point, each pass must either improve or leave untouched the approximation for each variable, and our lattice must not have any infinitely ascending chains, which is easy to see is not the case by looking at Figure 4. If a pass were to worsen the approximation, then the iterations could vacillate back and forth forever.

Let  $\sqsubseteq$  be the less or equally informative relation on types, where some type  $\tau_1 \sqsubseteq \tau_2$  if there is a line connecting  $\tau_1$  to  $\tau_2$  in Figure 4, with  $\tau_1$  lower than  $\tau_2$ , or if  $\tau_1$  and  $\tau_2$  are the same type. So, for example, `undefined`  $\sqsubseteq$  `int`, `int`  $\sqsubseteq$  `pyobj`, and `int`  $\sqsubseteq$  `int`. We can extend the informative relation to dictionaries by saying that some dictionary  $d_1$  is less or equally informative than dictionary  $d_2$ , written  $d_1 \sqsubseteq d_2$ , if for every variable  $x$  in  $d_1$ ,  $d_1[x] \sqsubseteq d_2[x]$ .

Now, to ensure that the approximation improves with every iteration, we will require that whenever the input to  $F$  increases (or stays the same), then the output also increases (or stays the same). That is, if for some  $d$  and  $d'$ ,  $d \sqsubseteq d'$ , then  $F(d) \sqsubseteq F(d')$ . This property is called *monotonicity*. Now, in the initial dictionary  $d_0$ , every variable assigned the `undefined` type, which is the bottom of the type lattice. So the first iteration will result in a dictionary  $d_1$  that is at least as informative as  $d_0$ , i.e.  $d_0 \sqsubseteq d_1$ . If  $d_0 = d_1$  then we have already found the fixed point of  $F$  and we are done. Otherwise, we apply  $F$  again. Because  $d_0 \sqsubseteq d_1$  then  $F(d_0) \sqsubseteq F(d_1)$  by the monotonicity of  $F$ , so we have  $d_1 \sqsubseteq d_2$ . We can continue this process until we can't go higher in the lattice (the lattice does not have infinitely ascending chains) and have therefore reached the fixed point.

The function  $F$  is determined by the program and how we propagate types through the various operations. For each operation, such as addition, we have corresponding rules for what the return type should be given the types of the arguments. Figure 6 shows the rules for addition.

To ensure that  $F$  is monotone, we need each of the typing rules also be monotone. Let's check that the typing rule for addition is monotonic. Suppose that the inputs to an addition go from being `(int, int)` to `(int, pyobj)`. Then the output goes from `int` to `pyobj`.

**Exercise 2.1.** (Don't turn this in.) Check two more combinations of inputs to addition to make sure the output does not decrease. Look at two more operations, such as less than comparison and if expressions.

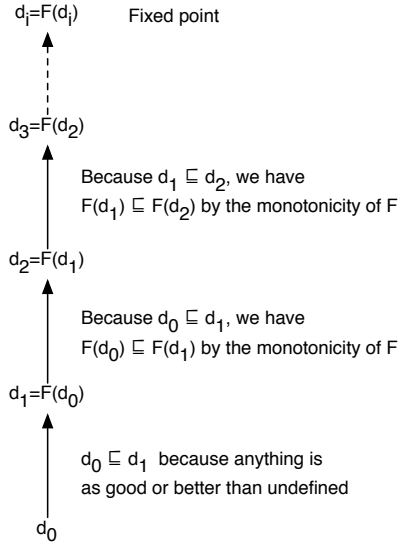


Figure 5: The monotonicity of  $F$  ensures that the process reaches a fixed point.

+	undefined	int	bool	float	pyobj
undefined	undefined	undefined	undefined	undefined	undefined
int	undefined	int	int	float	pyobj
bool	undefined	int	int	float	pyobj
float	undefined	float	float	float	pyobj
pyobj	undefined	pyobj	pyobj	pyobj	pyobj

Figure 6: Type rule for addition.

## 2.1 Implementing the SSA conversion and type analysis

Figure 7 shows a suggested organization for the compiler with the SSA conversion and iterative type analysis.

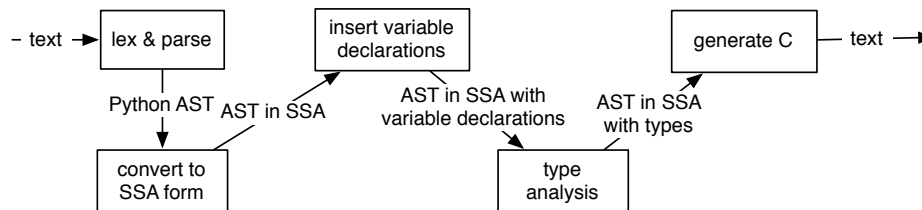


Figure 7: Compiler organization with SSA conversion and type analysis

For the SSA form, you'll need to decide how to add  $\phi$  nodes to your AST. One option is to add a new kind of expression node named Phi and then add assignment statements with Phi expressions on the right-hand sides. The problem with this approach is that it will be challenging to generate C for the Phi expressions because you'll have to consider context of the Phi to find the last join in the control flow graph and insert assignments on the other sides of the incoming edges. Alternatively, you can augment the If and While statement nodes to carry extra information about which variables need to be assigned a  $\phi$  and which variables go into which  $\phi$ 's.

**Exercise 2.2.** (Turn this in.) Change your  $P_2$  to C compiler to use SSA form and then do iterative type analysis for variables and code specialization based on the types. The conversion to SSA form should be done before inserting variable declarations, because SSA conversion introduces new variables. The type analysis, as before, should come after inserting the variable declarations, but should update the .type attributes of the variable declarations.

Make sure that you don't introduce bugs by adding this optimization. This assignment will be graded both on correctness (passing the test cases) and on the quality of the code that you produce (the fewer the pyobj's the better).

## References

- [1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.
- [2] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003.
- [3] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, 1994.
- [4] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 150–164, New York, NY, USA, 1990. ACM Press.

- [5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
- [6] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.
- [7] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM Press.