

Assignment 9: first steps towards x86

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

November 12, 2007

This week we begin the decent from C to x86 assembly language. Figure 1 shows the changes and new additions to the compiler.

1 Cast Insertion

The first difference between C and x86 that we need to deal with is that C allows implicit conversions between, for example, integers and floating point numbers. In x86 we must use specific conversion instructions. As a step in this direction, we introduce a new AST node class called `Cast` with which we can wrap an expression to specify a conversion from some source type to the destination type.

```
class Cast:
    def __init__(self, expr, source_type, dest_type):
        self.expr = expr
        self.source_type = source_type
        self.dest_type = dest_type
```

In the existing Type Analysis phase we have all the information we need to insert casts, so I recommend updating that pass to insert the `Cast` nodes where they are needed. In addition, your work in the next part of the assignment will be easier if you annotate operations such as `Add` with what kind of operation it is (i.e., integer addition, floating point addition, or dynamically-dispatched addition).

The cast insertion pass needs to come before removing complex opera* (Section 3) because inserting casts sometimes replaces simple expressions (variables or constants) with complex expressions (a cast).

2 Consolidate to a C-level AST

Before we begin the decent down to x86 assembly, we need to consolidate our position at the level of C. Our compiler currently ends with a translation from a Python-level AST to C text. Instead, we need to translate from a Python-level AST into a C-level AST, and then write a new text generator that converts the C-level AST to text.

In many ways, our Python-level AST is already quite similar to C. However, there are some important differences that we need to deal with.

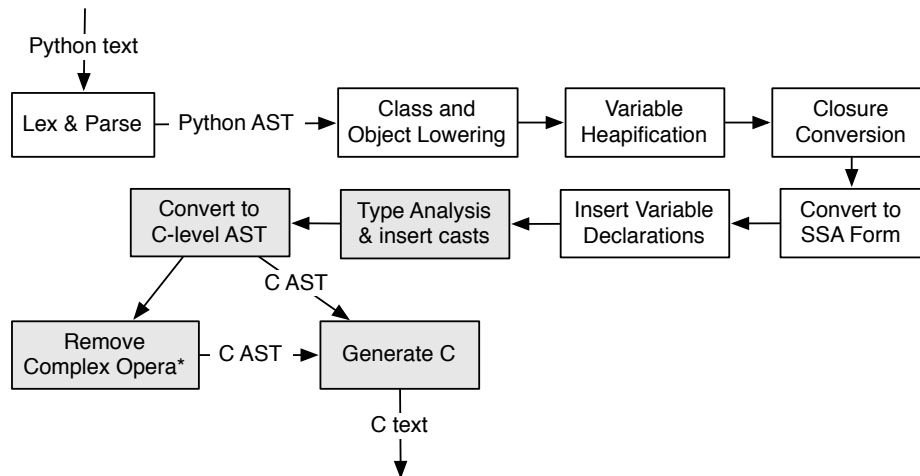


Figure 1: Structure of the compiler.

1. The `CallFunc` nodes in the Python-level AST represents a Python function call, which includes lots of dispatching, etc. Most of you compile `CallFunc` into a call to a C run-time helper named something like `call_pyfun`. Instead of translating to a string that represents a C-level call to `call_pyfun`, you need to translate to an AST representation for a C-level call to `call_pyfun`. It makes sense to reuse the `CallFunc` class for the C-level function call in the new AST. The following is a sketch of this translation.

```

CallFunc(e0, [e1, ..., en])
⇒
CallFunc(Name('call_pyfun'), [e0, [e1, ..., en]])

```

2. Expression nodes such as `Add`, `Mul`, etc. need to be converted into C-level AST nodes. Currently, the C text that you generate depends on the types of the arguments. You either generate an actual C operator expression, such as `+` or `*`, or you generate a call to a run-time function (if either argument is a `pyobj`). Similarly, when converting from the Python-level AST to the C-level AST, you'll either keep the operation in `Add` or `Mul` nodes, or you'll change it into a `CallFunc` node.
3. Many of you use the GNU extension “statement expressions” in your C output. You'll need to create a new AST class to represent them.

```

class StmtExpr:
    def __init__(self, stmts):
        self.stmts = stmts

```

4. When we finally generate x86 assembly, we will no longer be able to `#include` the runtime header file. So, for example, you won't be able to use macros to help with the generation of your output. Instead of using macros, you can write Python functions in your compiler that do the same thing as the macro, but that operate on and generate AST nodes.
5. A Python program is a `Module` containing a sequence of statements. A C program is a sequence of declarations, where one of those declarations is the `main` function. You should

take all of the function statements inside the `Module` and use them as the initial sequence of declarations in your C program. Then add a `main` function at the end that contains all of the other statements from the `Module`.

3 Remove Complex Opera*

Another major difference between C and x86 is that C has expressions with arbitrary nesting, whereas x86 just has instructions (which are analogous to C statements). In this pass we flatten expressions so that the operator and operand nodes (the child nodes) of an expression are always *simple*. The only expressions that are simple are variables and constants. All other expressions are complex.

Consider the following Python program.

```
x = 20
y = x + 1
def square(n):
    return n * n
z = 2 * y + square(x-10)
print z
```

Our goal is to translate it to the following program.

```
x = 20
y = x + 1
def square(n):
    tmp0 = n * n
    return tmp0
tmp1 = x - 10
tmp2 = 2 * y
tmp3 = square(tmp1)
tmp4 = tmp2 + tmp3
print tmp4
```

We can accomplish this translation by, as usual, writing a family of recursive functions, one for expressions and one for statements. The function for expressions should have two parameters, the AST node and a flag to say whether the expression is inside another expression or not. The function for expressions should return two things, the (possibly) new AST node for the expression and a list of statements (most of which will be assignments to new temporary variables). Let us consider a few cases.

Name Return the variable and an empty list.

Const Return the constant and an empty list.

Add Recursively process the `left` and `right` (with the flag set to true) to get new left and right nodes and two lists of statements.

- If we are inside a another expression (check the flag) then we create a new temporary variable and an assignment node that assigns the temporary to an `Add` node with the new left and right. We concatenate of the two lists of statements and append the new assignment node into a big list of statements. We return the temporary variable (in a `Name` node) and the big list of statements.

- If we are not inside another expression, return a new `Add` node created from the new left and right nodes and the concatenation of the two statement lists.

`StmtExpr` The statement expression contains a list of statements where the last statement is an expression. Recursively process the statements to get new versions of the statements and concatenate the returned lists of statements into one big list. Return the last expression and this big list of statements.

Take some care with the ordering of the list of statements so that side-effects occur in the same order as they would in the original program.

The recursive function that processes statements just returns a new version of the statement. Again, we consider a few cases.

`Discard` Recursively process the expression (with the flag set to false), getting back a new version of the expression and a list of statements. Return a `Stmt` nodes that contains the list of statements with a `Discard` node with the new expression appended at the end.

`If` Recursively process the condition expression (with the flag set to true) to get a new condition and a list of statements. Process the then and else branch statements to get new then and else statements. Return a `Stmt` node that contains the list of statements from the condition followed by an `If` statement with the new then and else branches.

`While` Recursively process the condition expression e (with the flag set to true) to get a new condition e' and a list of statements $s_0 \dots s_n$. Process the loop *body* to get a *newbody*. Then generate a `Stmt` node (containing a `While` node, etc.) according to the following plan. The list of statements need to be duplicated so that they are evaluated each time through the loop, before e' is evaluated.

```

while (e)
  body
 $\implies$ 
{
   $s_0 \dots s_n$ 
  while (e') {
    newbody
     $s_0 \dots s_n$ 
  }
}

```

4 Generate C Text from the C-level AST

This pass is similar to your existing C-generating pass. However, many of the cases will be simpler because the larger chunks of text are now represented as AST nodes, so you just need to output the list bit of text that goes with each AST node.

Exercise 4.1. Update your compiler with the cast insertion, conversion to C-level AST, and the removal of complex opera*. The C program that you output should not have any implicit casts and it should not have any complex opera*. Before implementing the pass that removes complex opera*, I recommend implementing the generation of C text from the C-level AST. This will allow you to catch any errors at that point before things get more complicated.