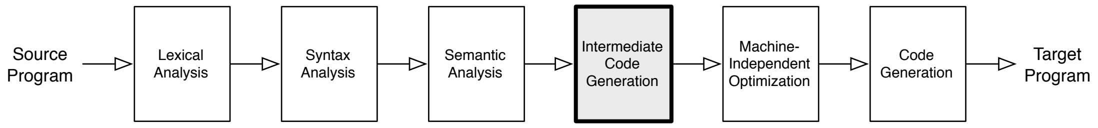
**CS 301** Spring 2016
**Meetings** March 14

# Intermediate Code, Object Representation, Type-Based Optimization

## Plan



This week brings our departure from the front end of the compiler as we start to consider translating to intermediate code and implementing objects.

- The first topic is *intermediate code generation*, the compiler phase that converts the high-level, hierarchical AST representation into a low-level, flat three-address-code representation. This "lowering" or "flattening" brings our program representation closer to a realistic machine model and provides a convenient platform for optimization and machine code generation, our topics for the next few weeks.
- The second topic is object representation: how to implement the run-time behavior of IC objects, including storage of fields and dispatch of method calls.
- Finally, we consider two optimization techniques that exploit realistic use of inheritance and polymorphism in object-oriented languages, statically (based on the class hierarchy) and dynamically (based on observed run-time behavior). These techniques are key in many modern language implementations.

There are several small readings on intermediate code and object representation, plus two classic research papers on type-based optimizations.

## Readings

- TAC specification, `https://cs.wellesley.edu/~cs301/project/tac.pdf`
- Dragon 6.2 – 6.2.1
- EC 5.3 (Skim as needed)
- EC 6.3.3 - 6.3.4
- Appel, *Modern Compiler Implementation in Java*, 14.1-14.2, 14.6-14.7. Available in bookcase outside my door.
- "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," Jeffrey Dean, David Grove, and Craig Chambers, ECOOP 1995.
- "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches," Urs Hölzle, Craig Chambers, David Ungar, ECOOP 1991.

## Exercises

1. (**Optional**) As you approach the translation of AST to three-address code (TAC), you may find it helpful to revisit the TINY compiler back end as a small-scale example.
`https://cs.wellesley.edu/~cs301/labs.html#tiny-backend`

2. This problem explores how to translate a program represented as an AST into three-address code (TAC), an intermediate representation of programs that we will use for optimization and translation to machine code. A specification of the TAC instruction set for this question is on the web site project page. As an example, consider the following while loop and its translation:

| | |
|---|---|
| <pre>n = 0;<br>while (n < 10) {<br>  n = n + 1;<br>}</pre> | <pre>n = 0<br>label test<br>t1 = n < 10<br>t2 = not t1<br>cjump t2 end<br>label body<br>n = n + 1<br>jump test<br>label end</pre> |

To leave the AST behind and move toward optimization and code generation, the compiler will translate programs to TAC. Here, we develop a definition of this step as a *syntax-directed translation*, meaning we give a function from syntactic forms of the source language to semantically sequences of TAC instructions. Your next project phase after spring break will start with a more concrete implementation of this translation, working from your AST.

Note that the operands of each TAC instruction are either program variable names (*e.g.*, `n`), temporary variable names introduced during translation (*e.g.*, `t2`, `t3`), or constants (*e.g.*, `0`, `10`, `1`). Branch instructions refer to label names generated during the translation.

Below, the function $T$ defines a translation such that $T[s]$ is an equivalent TAC representation for the high-level source statement $s$. $T[e]$ does the same for an expression $e$. When translating expressions, $e$, use $\texttt{t} := T[e]$ to denote the series of instructions to compute $e$, concluding by storing the result of $e$ into temporary variable $t$.

Translate expressions recursively. To translate an expression with subexpressions, first translate the subexpressions, then generate code to combine their results according to the top-level expression. For example, $\texttt{t} := T[e_1 + e_2]$ would be:

```
t1 := T[e1]
t2 := T[e2]
t = t1 + t2
```

The first two lines recursively translate $e_1$ and $e_2$ and store their results in new temporary variables `t1` and `t2`, which are then added together and stored in $t$. Here are a few other general cases:

| $e$ | t := $T[e]$ | (description) |
|---|---|---|
| v | t := v | (variable) |
| n | t := n | (integer) |
| $e_1$.f | t1 := $T[e_1]$<br>t = t1.f | (field access) |
| $e_1$[$e_2$] = $e_3$ | t1 := $T[e_1]$<br>t2 := $T[e_2]$<br>t3 := $T[e_3]$<br>t1[t2] := t3 | (array assignment) |

Generate new temporary names whenever necessary. For more complex expressions, apply rules recursively. $T[\ \texttt{a[i]} = \texttt{x} * \texttt{y} + \texttt{1}\ ]$ becomes:

```
t1 := T[a]            t1 = a              t1 = a              t1 = a
t2 := T[i]            t2 = i              t2 = i              t2 = i
t3 := T[x*y+1]        t4 := T[x*y]        t6 := T[x]          t6 = x
                  ≡                   ≡   t7 := T[y]      ≡   t7 = y
                                          t4 = t6 * t7        t4 = t6 * t7
                      t5 := T[1]          t5 = 1              t5 = 1
                      t3 = t4 + t5        t3 = t4 + t5        t3 = t4 + t5
```

Translation of statements follows the same pattern. $T$[ `while (` $e$ `)` $s$ ] becomes:

```
label test
  t1 := T[e]
  t2 = not t1
  cjump t2 end
  T[s]
  jump test
label end
```

(a) Define $T$ for the following syntactic forms:

- `t :=` $T[e_1$ `*` $e_2]$
- `t :=` $T[e_1$ `||` $e_2]$ (where `||` is short-circuited)
- $T$[ `if (` $e$ `)` $s_1$ `else` $s_2$ ]
- $T$[ $s_1$`;` $s_2$`;` `...;` $s_n$ ]
- `t :=` $T$[ `f(`$e_1$`,` `...,` $e_n$`)` ]

(b) These translation rules introduce more copy instructions than strictly necessary. For example, `t4 :=` $T$[ `x * y` ] becomes

```
t6 = x
t7 = y
t4 = t6 * t7
```

instead of the single statement

```
t4 = x * y
```

Describe how you would change your translation function to avoid generating these unnecessary copy statements.

(c) The original rules also use more unique temporary variables than required, even after changing them to avoid the unnecessary copy instructions. For example,

$T$[ `x = x*x+1; y = y*y-z*z; z = (x+y+w)*(y+z+w)`]

becomes the following:

```
t1 = x * x
t2 = t1 + 1
x = t2
t3 = y * y
t4 = z * z
y = t3 - t4
t5 = x + y
t6 = t5 + w
t7 = y + z
t8 = t7 + w
z = t6 * t8
```

Rewrite this to use as few temporaries as possible. Generalizing from this example, how would you change $T$ to avoid using more temporaries than necessary.

(d) Eliminating unnecessary variables here may seem like a good idea, but there are enough downsides that we will avoid it in our implementation. What are some reasons to stick with original, more verbose translation (for both or either of questions 2b and 2c)?

(e) **(Optional)** Translation of some constructs, such as nested `if` statements, `while` loops, short-circuit and/or statements may generate adjacent labels in the TAC. This is less than ideal, since the labels are clearly redundant and only one of them is needed. Illustrate an example where this occurs, and devise a scheme for generating TAC that does not generate consecutive labels.

3. Draw the run-time structures for the IC program given below, following the style of EC 6.3.4.

   - Show information about each class, including the method vector and superclass pointer, but omitting the class pointer and classmethods vector. (IC does not support classes as objects themselves or static methods.)
   - The representation of each object allocated by the sample program, including its class pointer, method vector pointer (point to class's method vector, not a copy as in EC), and fields.

   Draw only data and metadata. Omit representation of code itself (or abstract as a box).

```
class A {
  int x;
  void setX(int x) {
    this.x = x;
  }
  int f() {
    return this.x + 73;
  }
  int g() {
    return f() + 2;
  }
}
class B extends A {
  string s;
  void setS(string s) {
    this.s = s;
  }
  int f() {
    Library.println(s);
    return x + 12;
  }
  void main(string[] args) {
    A a = new A();
    B b = new B();
    a.setX(1);
    b.setX(2);
    b.setS("Yum. ");
    Library.printi(a.g());
    Library.printi(b.g());
  }
}
```

4. Give the series of lower-level steps taken to execute the `main` method using basic operations on the run-time structures from exercise 3. (You do not need to write these steps down, but be prepared to walk me through them.)

5. Exercise 14.3 from Appel, *Modern Compiler Implementation in Java*, (see above). This exercise (and the next) use ideas from "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," which Appel summarizes.

6. Exercise 14.4 from Appel, *Modern Compiler Implementation in Java*, (see above).

7. Read "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis." This paper introduces many of the ideas that power the preceding two exercises. Read and consider:

   - For what kinds of programs is CHA particularly effective or ineffective? What features or idioms does it affect?
   - What are the main limitations of CHA?
   - Section 2.2.3 discusses how CHA can be used for dynamically-typed languages like Smalltalk (or Ruby). Would these techniques be useful for Java or IC?

8. Read "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches."

   - What does this optimization do?
   - When are PICs most or least effective? What features or idioms do they target?
   - Compare PICs with CHA for dynamically-typed languages.
     - What is the same?
     - What is different?
   - SELF, like JavaScript, is a classless object-oriented language that relies on *prototypes* to define object behavior. This is not the same as a dynamically-typed class-based language. Prototypes essentially give every object its own method vector, which can be arranged at object allocation time by the programmer. Distinct objects may share the same prototype, giving them the same behavior, or include some of the same methods, but there are no classes or inheritance that enforce a strict hierarchy as in Java or IC. PIC was developed for such a language (SELF). Naturally, with no class hierarchy, CHA does not apply directly. Can any of the techniques from CHA be adapted for use with prototypes?