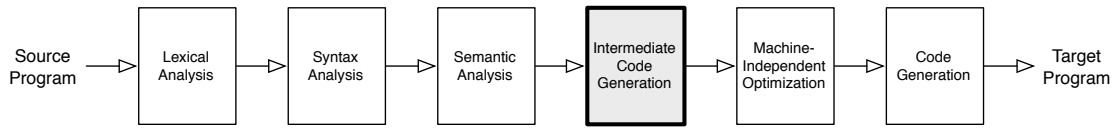
CS 301 Spring 2019
Tutorial Assignment
19 March

Intermediate Code, Method Dispatch

1 Plan



This week we move beyond the front end of the compiler to translate ASTs to intermediate code and implement dynamic method dispatch.

- 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 objects, including storage of fields and dispatch of method calls.
- Finally, we consider optimization techniques that reduce the cost of subtype polymorphism in languages with dynamic method dispatch.

2 Readings

Intermediate representation:

- TAC specification, attached.
- Dragon 6.2 – 6.2.1
- EC 5.3 (Skim as needed)

Runtime object representations and method dispatch:

- EC 6.3.3 - 6.3.4

Optimizing complex method dispatch (optional, but fascinating):

- *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches.*
  Urs Hölzle, Craig Chambers, David Ungar. ECOOP 1991.
  `https://www.cs.ucsb.edu/~urs/oocsb/papers/ecoop91.pdf`
- *Property Caches Revisited.*
  Manuel Serrano, Marc Feeley. CC 2019.
  `https://doi.org/10.1145/3302516.3307344`

3 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:

```
n = 0;
while (n < 10) {
  n = n + 1;
}
```

```
n = 0
label test
t1 = n < 10
t2 = not t1
cjump t2 end
label body
n = n + 1
jump test
label end
```

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 `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, `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` $:= T[e_1]$ `t = t1.f`	(field access)
e_1`[`e_2`] = `e_3	`t1` $:= T[e_1]$ `t2` $:= T[e_2]$ `t3` $:= T[e_3]$ `t1[t2] := t3`	(array assignment)

Generate new temporary names whenever necessary. For more complex expressions, apply rules recursively. $T[$ `a[i] = x * y + 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_1 `)` e_2] becomes:

```
label test
  t1 := T[e1]
  t2 = not t1
  cjump t2 end
  T[e2]
  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_1 `)` e_2 `else` e_3]
- T[{ e_1`;` e_2`; ...;` e_n }]
- `t :=` T[`e`$_0$`(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 ROOST program given below, following the general style of EC 6.3.4, but with some adaptations for ROOST.

   - Show information about the methods of each structure and signature type. Unlike Java or other languages with inheritance, ROOST does not need to support a relationship between subclass/superclass field layouts or method vectors, or some of the other details like static items or class methods. However, ROOST does need a way to lookup the relevant structure method for an arbitrary object that implements a particular signature. This lookup shares some similarities with lookups of methods in languages that support multiple inheritance or interfaces.
   - Show representation for each structure instance allocated by the sample program, including its field data layout and a reference to its structure's method lookup tables.

   Draw only data and metadata. Omit representation of code itself (or abstract as a box). To be clear, you will need to do some inventing to come up with a way to do some parts of this for ROOST.

```
sig A {
  method set(x: i64) -> unit
  method get() -> i64
}
sig B {
  method get() -> i64
  method setS(s: str) -> unit
}
struct C impl A B {
  field x: i64
  field s: str
  method setS(s: str) -> unit {
    self.s = s;
  }
  method get() -> i64 {
    Roost.println(s);
    self.x + 12
  }
  method set(x: i64) -> unit {
    self.x = i64;
  }
}
struct D impl A {
  field value: i64
  method get() -> i64 {
    self.value
  }
  method set(x: i64) -> unit {
    self.value = x;
  }
}
```

```
fn main(args: str[]) -> unit {
  let a: A = new D()
      b: B = new C()
      c: C = new C()
  in {
    a.set(1);
    b.setS("Roost");
    c.set(2);
    c.setS("Yum. ");
    Roost.printi64(a.get());
    Roost.printi64(b.get());
    Roost.printi64(c.get());
  }
}
```

4. (Optional, but fascinating) 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?
   - SELF 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 ROOST. PIC was developed for such a language (SELF).

5. (Optional, but fascinating) Skim *Property Caches Revisited*, an update techniques related to PIC as applied for server-side JavaScript. Like SELF, JavaScript is a classless prototype-based object-oriented language.

   - What has changed in the roughly 30 years since SELF was new?
   - How do *hidden classes* assist in optimizing prototype-based lookups?
   - What are polymorphic call sites and how do so-called *virtual tables* help optimize lookups at them?

4 Sample TAC Definition

This page summarizes a simple three-address code (TAC) intermediate language. There are many choices as to the exact instructions to include in such a language; we will use something a little different for the ROOST compiler.

- **Arithmetic and Logic Instructions.**

  The basic instruction forms are:

  ```
  a = b OP c          a = OP b
  ```

  where `OP` can be

an arithmetic operator:	`ADD`, `SUB`, `DIV`, `MUL`
a logic operator:	`AND`, `OR`, `XOR`
a comparison operator:	`EQ`, `NEQ`, `LE`, `LEQ`, `GE`, `GEQ`
a unary operator:	`MINUS`, `NEG`

- **Data Movement Instructions.**

Copy:	`a = b`	
Array load/store:	`a = b[i]`	`a[i] = b`
Field load/store:	`a = b.f`	`a.f = b`

- **Branch Instructions.**

Label:	`label L`	
Unconditional jump:	`jump L`	
Conditional jump:	`cjump a L`	(jump to `L` if `a` is true)

- **Function Call Instructions.**

Call with no result:	`call f(`a_1`, ..., `a_n`)`
Call with result:	`a = call f(`a_1`, ..., `a_n`)`

  (Note: there is no explicit TAC representation for parameter passing, stack frame setup, etc.)