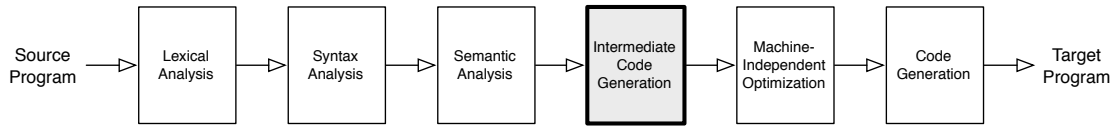
CS 301 Spring 2021 T4
Class Assignment
11 May

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.

2 Readings

Intermediate representation:

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

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/project/tiny/`

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 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.

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.)