COM SCI 132 Week 3

Aidan Jan

April 22, 2024

Translation of Expressions

$$\varepsilon, k \to \text{code}, k'$$

- ε = minijava expression
- k = find unused number
- code = sparrow code
- k' = first unused number after we translated ε

To compile minijava into sparrow, we use an indexed list. We go expression by expression through the minijava program, and load all the variables into the list.

Consider the syntax:

$$5, k \to (t_k = 5), k + 1$$

- This means to put the *expression* (in this case 5), into index k of the list.
- The end result is $t_k = 5$, and k is incremented to point to the next index.

More generally,

$$v, k \to (t_k = v), k + 1$$

where v represents the value of a local variable.

Addition Example

$$\epsilon_1, k + 1 \to \text{code}_1, k_1. \qquad\qquad e_2, k_1 \to \text{code}_2, k$$

Suppose we have $e_1 + e_2$. How does that translate?

- In the indexed list, index k is taken by the result of $e_1 + e_2$ (1 'slot').
- From index $k + 1$ to k_1 is the space to store code$_1$. (many 'slots')
- From index k_1 to k_2 is the space to store code$_2$. (many 'slots')

We write:

$$\begin{aligned} e_1 + e_2, t_k &\to \text{code}_1 && // \text{ result is in } t_{k+1} \\ &\to \text{code}_2 && // \text{ result is in } t_{k_1} \end{aligned}$$

What is t_k in the first line? Well, $t_k = t_{k+1} + t_{k_1}$

Java Example

To implement the above example in Java,

```
class Result {
    String code;
    int t;

    public Result(String code, int t) {
        this.code = code;
        this.t = t;
    }

}

class Translator {
    Result visit(IntConst n, int k) {
        return new Result(
            "t" + k + "=" n, k + 1
        )
    }
    Result visit(Plus n, int k) {
        Result r1 = n.e1.accept(this, k + 1);
        Result r2 = n.e2.accept(this, r1.c);

        return new Result(
            r1.code + r2.code + "t" + k + "= t" + (k + 1) + "+ t" + r1.c
        );
    }
}
```

Translation of Statements

$$s, k \to \text{code}, k'$$

- s = minijava statement
- k = first free number in indexed list
- code = sparrow code
- k' = first free number after translation

$$\frac{e, k \to \text{code}, k_e}{(v = e), k \to \text{code}, v = t_k}$$

- v represents a local variable.
- This essentially stores the code in t_k.

$$\frac{s_1, k \to \text{code}_1, k_1 \qquad s_2, k_1 \to \text{code}_2, k_2}{(s_1 \; ; \; s_2), k \to \binom{\text{code}_1}{\text{code}_2}, k_2}$$

- $(s_1 \; ; \; s_2)$ = minijava statements
- k = first free number

In this example, in the indexed list, indices k to k_1 are taken up by s_1, and k_1 to k_2 are taken up by s_2.

If-Else Example

$$\frac{e, k \to \text{code}_e, k_e \qquad s_1, k_e \to \text{code}_1, k_1 \qquad s_2, k_1 \to \text{code}_2, k_2}{\text{if } (e) s_1 \text{ else } s_2, k \to \text{code}_e \quad // \text{ result in } t_k}$$

- (if $(e)s_1$ else s_2) = minijava statement
- k = first free number

For the if-else statement, we have two branches. As a result, we must allocate space to store two separate sections of code. The issue with our indexed list is that we execute it sequentially - we need some way to tell the computer to only execute one branch. The solution? GOTO statements!

Our block would look something like:

```
if0 $t_k$ goto else$_{k_2}$         // if0 is the same as JZ in assembly - jump if $t_k == 0$
    code$_1$                  // code for if if statement evaluates to true (1)
    goto end$_{k_2}$             // jump over the else branch
else$_{k_2}$:
    code$_2$                  // code for if if statement evaluates to false (0)
end$_{k_2}$:
    ...
```

In this example, the spaces in the indexed list are:

- Indices k to k_e are taken by the conditional of the if statement
- k_e to k_1 is the True branch of the if statement
- k_1 to k_2 is the False branch of the if statement
- k_2 is the label for else$_{k_2}$
- $k_2 + 1$ is the label for end$_{k_2}$

As a result, the k value after the if statement would be $k_2 + 2$. The next statement or expression would begin there.

While Example

$$\frac{e, k \to \text{code}_e, k_e \qquad s, k_e \to \text{code}_s, k_s}{(\text{while}(e)\ s), k \to \text{loop}_{ks}\ :\ k_e}$$

In this statement, we have two sections of code, one for the conditional, and one for the statements executed by the while loop. However, we must be able to loop in the code. This can also be done with GOTO statements!

Our block would look something like:

```
loop$_{k_s}$:
    code$_e$                  // result in $t_k$, conditional of while loop
    if0 $t_k$ goto end$_{k_s}$     // if conditional evaluates to false, don't loop.
    code$_s$                  // statements
    goto loop$_{k_s}$            // after body of while loop executes, go back to the condi
end$_{k_s}$:
    ...
```

In this example, the spaces in the indexed list are:

- k_e to k_s is taken by the conditional
- k_s to k is taken by the statements

- k is the label for loop_{k_s}
- $k+1$ is the label for end_{k_s}

While and If statements are the same construction in Sparrow - the major difference is that the while loop jumps backwards and the if statement jumps forwards.

If-Elif-Else Statements (Nested If)

We write elif as a nested if statement.

$$\frac{\dfrac{e_1, k \to \text{code}_{e_1}, k_{e_1} \qquad s_1, k_{e_1} \to \text{code}_{s_1}, k_{s_1} \qquad e_2, k_{s_1} \to \text{code}_{s_2}, k_{e_1} \qquad s_2, k_{e_2} \to \text{code}_{s_2}, k_{s_2} \qquad \ldots}{\text{if}(e_1) s_2 \text{ else } s_2, k_{e_1} \to \text{code}, k_{if}}}{\text{if}(e_1) s_1 \text{ else } (\text{if}(e_2) s_2 \text{ else } s_3).k \to}$$

This example would give the block:

```
code_{e_1}
if0
code_{s_1}
goto

else_{k_{if}}:
    code

end_{k_{if}}:
```