COM SCI 132 Week 4

Aidan Jan

May 6, 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_k2        // 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_k2             // jump over the else branch
else_k2:
    code_2                  // code for if if statement evaluates to false (0)
end_k2:
    ...
```

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_ks:
    code_e                  // result in t_k, conditional of while loop
    if0 t_k goto end_ks     // if conditional evaluates to false, don't loop.
    code_s                  // statements
    goto loop_ks            // after body of while loop executes, go back to the condi
end_ks:
    ...
```

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.

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

This example would give the block:

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

else_{k_if}:
    code

end_{k_if}:
```

Arrays

Allocation

```
e ::= new int[e] | e_1[e_2] | e.length
s ::= v[e_2] = e_3

void m([]int a) {
    ... a.length...
}
```

In the sparrow heap, arrays are stored as the following:

- index 0 is the length of the array. (arrays are fixed-length)
- indices 1- is the contents of the array.
- In minijava, each index is four bytes (because an int is 4 bytes)
  - offset = $4(\text{index}+1)$

$$\frac{e, k+1 \to \text{code}_e, k_e}{(\texttt{new int [e]}), k \to \text{code}_e \qquad \texttt{// result in } v_{k+1}}$$

- `(new int [e])` is a minijava expression
- k is the first free number
- The memory slot the element is stored in v_{k+1} corresponds to the $k+1$ in the 'numerator'.
- k_e is the offset value - that is, $v_{k+1}+1$
- Since this code is for declaring a `new` array, we must allocate the space in heap.

- Incrementing $t_{k_e} = v_{k+1} + 1$ is equivalent to doing $t_{k_e} = 4 \cdot t_{k_e}$
- $t_{k_e+1} = \texttt{alloc}(t_{k_e})$ // this statement allocates the entire array
- Ideally, after the allocate, we would traverse through and set all the values to 0. However, allocating the array means you can't read the values there before, so the step *can* be skipped.
- To set to 0, move the pointer, then set 0. Then move the pointer again, then set 0, etc... Continue until you reach the correct length.

Length Function

$$\frac{e, k+1 \rightarrow \texttt{code}, k_e}{\texttt{e.length}, k \rightarrow \texttt{code}_e \qquad \texttt{// result in } t_{k+1}}$$

- `e.length` is a minijava expression
- k is the first free number
  - $t_k = [t_{k+1} + 0]$
- The result of $\texttt{code}_e$ is stored in t_{k+1} because t_k contains the length of the array.

Array Access

$$\frac{e_1, k+1 \rightarrow \texttt{code}_1, k_1 \qquad e_2, k_1 \rightarrow \texttt{code}_2, k_2}{e_1[e_2], k \rightarrow \texttt{code}_1, \texttt{code}_2}$$

- $\texttt{code}_2$ is stored in t_{k+1}
- $\texttt{code}_3$ is stored in t_{k_1}
- $t_{k_2} = 4(t_{k_1} + 1)$
- $t_k = [t_{k+1} + t_{k_2}]$

Note that:

- t_k represents the location in heap where the number from the access is stored. That is, $e_1[e_2]$ is stored at that location.
- $t_k = [t_{k+1} + t_{k_2}]$ because t_{k+1} represents the first index of the content of the array, and t_{k_2} is the byte offset to the index from the top of the array.

$$\frac{e_2, k \rightarrow \texttt{code}_2, k_2 \qquad e_3, k_2 \rightarrow \texttt{code}_3, k_3}{(v[e_2] = e_3), k \rightarrow \texttt{code}_2, \texttt{code}_3}$$

- $v[e_2] = e_3$ is a minijava expression
- k is the first free number
- $\texttt{code}_2$ is stored in t_k
- $\texttt{code}_3$ is stored in t_{k_2}
- $t_{k_3} = 4(t_k + 1)$, and [v + t_{k_3}] = t_{k_2}

Classes and Objects

Objects are instantiated like groups of variables in a heap. The code for the methods and function are stored like how they are allocated in C.

- Methods in form `m(int a)` translates to c_m`(this a)` - a combination of a class and method name.

$$e ::= \texttt{new}C() \mid x \mid \texttt{this} \mid e_1.m(e_2)$$

- C is the class name
- x is a field name. (e.g., C.x)

Accessing methods in classes

$$\frac{e_1, k+1 \to \texttt{code}_1, k_1 \qquad e_2, k_1 \to \texttt{code}_2, k_2}{e_1 \,.\, m(e_2), k \to \texttt{code}_1, \texttt{code}_2}$$

- $e_1 \,.\, m(e_2)$ is a minijava expression (the dot is part of minijava syntax)
- $\texttt{code}_1$ is result in t_{k+1}
- $\texttt{code}_2$ is result in t_{k_1}
- $t + k = C - m(t_{k+1}, t_{k_1})$
  - C-m is the function name in sparrow.
  - Like normal functions in sparrow, $\texttt{code}_1$ is the parameters, $\texttt{code}_2$ is the body of the function

Instantiating Objects

$$\texttt{new}\ C(), k \to t_k = \texttt{alloc}(4 \cdot \#fields(c)), k+1$$

- `new` $C()$ is the minijava expression
- k is the first free number
- $\texttt{alloc}(4 \cdot \#fields(c))$ allocates a space in heap large enough to store all the fields of the object.

Back to Type Checking (with class inheritance)

```
class C {
    t m (s a) {
        ...
    }
}

class D extends C {
    ...
    t m (s a) {
        ...
    }
    ...
}
```

- Minijava does not allow *overloading*, the practice of having two methods of the same name but different parameters in the same class

- It does, however, allow *overwriting*, or having an inheriting class overwrite a class in its parent. In the above example, the two methods have type t, the same method name m, and single parameter of type s and name a.

In this case, having a type checker simplifies a lot of code, since now the compiler just has to figure out which method to call. If it is an object of Class C, it calls the method in class C. If it is an object of Class D, it calls the method in class D, because the one in D overwrites the one in C.

Method Tables

```
class C {
    ...
    void m() {
        ...
    }
    void q() {
        ...
    }
}
```

- Method table would store the locations of the start of C-m and C-q.
- A pointer to the method table location in heap is stored in the first index of each object.
- Calling a class method makes the compiler
    1. Load method table
    2. Load the method you want
    3. Call the method

Calling a Method

$$\frac{e_1, k+1 \to \texttt{code}_1, k_1 \qquad e_2, k_1 \to \texttt{code}_2, k_2}{e_1 \,.\, m(e_2), k \to \texttt{code}_1, \texttt{code}_2}$$

- $e_1 \,.\, m(e_2)$ is the minijava expression (method call)
- k is the first free number
- $\texttt{code}_1$ is stored in t_{k+1}
- $\texttt{code}_2$ is stored in t_{k_1}
- $t_{k_2} = [t_{k+1} + 0]$
- $t_{k_2} + 1 = [t_{k_2} + 4 \cdot (\text{index of m})]$
- $t_k =$ call $t_{k_2+1}(t_{k+1}, t_{k_2})$

A View of Method Tables

```
class C {
    void m() {...}
    void p() {...}
}

class D extends C {
    void m() {...} // override!
```

```
    void q() {...} // only in D object!

}

/*
C Object Method Table:
[C-m][C-p]
0    4

D Object Method Table:
[D-m][C-p][D-q]
0    4    8
*/
```