Compiler concepts: Symbol tables
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @input1
1  D=M
2  @input2
3  D=D-M
4  @output_first
5  D;JGT
6  @input2
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @input1
12 D=M
13 (output_d)
14 @output_val
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @input2
3  D=D-M
4  @output_first
5  D;JGT
6  @input2
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @16
12 D=M
13 (output_d)
14 @output_val
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @output_first
5  D;JGT
6  @17
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @16
12 D=M
13 (output_d)
14 @output_val
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @output_first
5  D;JGT
6  @17
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @16
12 D=M
13 (output_d)
14 @18
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @output_first
5  D;JGT
6  @17
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @16
12 D=M
13 (output_d)
14 @18
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
10 @16
11 D=M
12 (output_d)
13 @18
14 M=D
15 (infinite_loop)
16 @infinite_loop
17 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @output_d
9  0;JMP
10 @16
11 D=M
12 (output_d)
13 @18
14 M=D
15 (infinite_loop)
16 @infinite_loop
17 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @12
9  0;JMP
10 @16
11 D=M
12 (output_d)
12 @18
13 M=D
14 (infinite_loop)
15 @infinite_loop
16 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @12
9  0;JMP
10 @16
11 D=M
12 @18
13 M=D
14 (infinite_loop)
15 @infinite_loop
16 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @12
9  0;JMP
10 @16
11 D=M
12 @18
13 M=D
14 (infinite_loop)
14 @14
15 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @12
9  0;JMP
10 @16
11 D=M
12 @18
13 M=D
14 @14
15 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.

Tracking labels and variables

Recall in Hack assembly, `@` can be followed by either a number, a label or a variable. The assembler must:

```
0  @16
1  D=M
2  @17
3  D=D-M
4  @10
5  D;JGT
6  @17
7  D=M
8  @12
9  0;JMP
10 @16
11 D=M
12 @18
13 M=D
14 @14
15 0;JMP
```

- Allocate each variable a corresponding address in RAM, starting from 16.
- Replace variables by their addresses.
- Assign each label the address in ROM matching the machine code line of its declaration.
- Replace labels by their addresses.
- Only then replace `@` statements by A-instructions.

We do this using "symbol tables".

Identifiers and symbol tables

An **identifier** is a catch-all term for a token whose meaning is defined in the code itself rather than the language.

In Hack, our identifiers are labels and variables — in the statement `@output_first`, we know what `@` means, but the we can only translate `output_first` by looking for its definition.

In C, the names of functions are also identifiers.

Identifiers and symbol tables

An **identifier** is a catch-all term for a token whose meaning is defined in the code itself rather than the language.

In Hack, our identifiers are labels and variables — in the statement `@output_first`, we know what `@` means, but the we can only translate `output_first` by looking for its definition.

In C, the names of functions are also identifiers.

A **symbol table** is a data structure mapping the names of identifiers to their meanings.

In Hack, we will have one symbol table for labels (mapping each label name to its ROM address) and one for variables (mapping each variable name to its RAM address).

In C, a symbol table would also include e.g. the type of a variable and the arguments of a function. (The historical need to fill symbol tables efficiently is why function headers exist.)

Symbol tables in Hack: The goal

```
0  @input1
1  D=M
2  @input2
3  D=D-M
4  @output_first
5  D;JGT
6  @input2
7  D=M
8  @output_d
9  0;JMP
10 (output_first)
11 @input1
12 D=M
13 (output_d)
14 @output_val
15 M=D
16 (infinite_loop)
17 @infinite_loop
18 0;JMP
```

Label table:

Name	ROM address
output_first	10
output_d	12
infinite_loop	14

Variables table:

Name	RAM address
input1	16
input2	17
output_val	18

How do symbol tables work?

A symbol table must support the following operations:

- Add a new name and address to the table.
- Check if a name is in the table.
- If a name is in the table, retrieve the corresponding address.

How do symbol tables work?

A symbol table must support the following operations:

- Add a new name and address to the table.
- Check if a name is in the table.
- If a name is in the table, retrieve the corresponding address.

The right way to implement this is with a hash table, which you'll see in Programming in C in a few weeks. But you can do it less efficiently with a dynamically-sized collection, which you've already seen. (How?)

How do symbol tables work?

A symbol table must support the following operations:

- Add a new name and address to the table.
- Check if a name is in the table.
- If a name is in the table, retrieve the corresponding address.

The right way to implement this is with a hash table, which you'll see in Programming in C in a few weeks. But you can do it less efficiently with a dynamically-sized collection, which you've already seen. (How?)

This is a good C exercise but a bad architecture exercise, so we've done it for you in the assignment — see `symboltable.c` and `symboltable.h`.

How do we fill symbol tables?

In assembly, filling symbol tables is simple enough to integrate with lexing and parsing. (In e.g. C it would happen later, during semantic analysis.)

Both the label and variable tables start empty.

How do we fill symbol tables?

In assembly, filling symbol tables is simple enough to integrate with lexing and parsing. (In e.g. C it would happen later, during semantic analysis.)

Both the label and variable tables start empty.

We can tell whether an identifier is a variable or label by looking for a label declaration (`label`). So during lexing, we remove the label declarations and add them to the label table with the correct ROM addresses.

(Recall label declarations have no tokens!)

How do we fill symbol tables?

In assembly, filling symbol tables is simple enough to integrate with lexing and parsing. (In e.g. C it would happen later, during semantic analysis.)

Both the label and variable tables start empty.

We can tell whether an identifier is a variable or label by looking for a label declaration (`label`). So during lexing, we remove the label declarations and add them to the label table with the correct ROM addresses.

(Recall label declarations have no tokens!)

Then in parsing, for each identifier we find, we check the symbol tables:

- If it's in the label table, hooray — substitute in the ROM address.
- If it's in the variable table, hooray — substitute in the RAM address.
- If it's in neither table, it must be the first occurrence of some variable. So we add it to the variables table with the first unassigned RAM address.

Advanced symbol tables: Scopes

In high-level languages, the compiler needs to track scopes. We build one symbol table for each scope. After building the tables in semantic analysis, we could store them in a **stack** (see Programming in C) as we convert code to IR form.

```c
#include <stdio.h>

int main() {
    double foo = 7;
    char i = 'a';
    for (int i=0; i<=5; i++) {
        printf("%f, %d", foo, i);
        foo /= 2;
    }

    foo = 50;
    for (long i=0; i<= 10; i++) {
        double temp = foo + 500;
        printf("%f, %d", temp, i);
        foo *= 2;
    }

    printf("%c", i); // Prints 'a'
    printf("%d", i)  // Compile error
    return 0;
}
```

Table 1:

[Contains `main`, functions from `stdio.h`]

Table 2:

Name	Type	Address
`foo`	`double`	***
`i`	`char`	***

Table 3:

Name	Type	Address
`i`	`int`	***

Table 4:

Name	Type	Address
`i`	`long`	***
`temp`	`double`	***

The compiler could start with table 1, push table 2 on line 3, push table 3 on line 6, pop on line 9, push table 4 on line 12, pop on line 16, then pop on line 21.

Advanced symbol tables: Scopes

In high-level languages, the compiler needs to track scopes. We build one symbol table for each scope. After building the tables in semantic analysis, we could store them in a **stack** (see Programming in C) as we convert code to IR form.

```c
#include <stdio.h>

int main() {
    double foo = 7;
    char i = 'a';
    for (int i=0; i<=5; i++) {
        printf("%f, %d", foo, i);
        foo /= 2;
    }

    foo = 50;
    for (long i=0; i<= 10; i++) {
        double temp = foo + 500;
        printf("%f, %d", temp, i);
        foo *= 2;
    }

    printf("%c", i); // Prints 'a'
    printf("%d", i)  // Compile error
    return 0;
}
```

Table 1:

[Contains `main`, functions from `stdio.h`]

Table 2:

Name	Type	Address
`foo`	`double`	***
`i`	`char`	***

Table 3:

Name	Type	Address
`i`	`int`	***

Table 4:

Name	Type	Address
`i`	`long`	***
`temp`	`double`	***

To retrieve information about a variable, the compiler could then start with the top-most table on the stack and work its way down, returning the first result.