Recitation 4

February 24, 2014

Announcements

- Assignment 4 is out, it consists of two parts:
  - **Theory** Graded, individual, deadline 26.02.
  - **Code** Pass/fail, groups of two, deadline 05.03.
- The last theory problem is quite hard, and is optional. You can get full score without solving it. Additional points carry over to future assignments.
- The theory is in part based on material handed out during the lecture on Thursday.

Assignment 4

- Theory
- Symbol tables
- Type checking

Symbol tables

- A symbol table is a table containing symbol names (e.g. the variables, functions and classes), and properties of those symbols names, such as type and address.
- When we see a declaration, we enter the symbol and the properties into the table.
- Later, when we see a usage of the symbol, we look it up in the table to find the properties.

Symbol table, example

```
VOID FUNC main()
START
  INT a;
  ...
  PRINT a;
END
```

Name	Address	Type

Symbol table, example

```
VOID FUNC main()
START
  INT a;
  ...
  PRINT a;
END
```

Name	Address	Type
a	0x3ff5b1	INT

When we see a declaration, we enter the symbol into the table (here, we just make up some address).

Symbol table, example

```
VOID FUNC main()
START
   INT a;
   ...
   PRINT a;
END
```

Name	Address	Type
a	0x3ff5b1	INT

When the symbol is used, we can look it up in the table, to see where it is stored, its type etc. (To print `a`, we obviously need to know where it is stored, we also need to know the type to print it correctly.)

Hash tables

- A symbol table could be implemented as an array, when we insert something, we add it at the end, when we look up something, we search through the whole array.
- This is, however, slow. A *hash table* is a much better suited data structure.
- We can insert (key,value) pairs into the table. Later, we can get the value back, using the key.
- In our case, the keys would be symbol names, and the values structs storing information about the symbols.

Hash tables

- Unfortuneately, C does not have hash tables in its standard library.
- So, we'll use a third party library, `libghthash` (`http://www.bth.se/people/ska/simhome/libghthash.html`)
- I'll provide precompiled versions for 32 and 64 bit Linux, for other platforms, you can compile it from source.
- You'll not interact with this library directly, but use wrapper functions in symtab.c, these will be introduced as we go along.

Tree traversal

- We'll implement the symbol table insertion and retrieval with another tree traversal.
- We'll complete the bind names traversal, using the `bind_X` functions in `bindnames.c`, and the `bind_names` function pointer.
- Essentially, whenever we se a ”declaration” node, we insert ”something”, and when we see a ”usage” node, we retrieve ”something”, and store it in the ”usage” node.
- What ”declaration”, ”usage” and ”something” is depends on the type of symbol, There are three different types, which must be handled differently:
  - Variables
  - Functions
  - Classes

Variables

- Variables are declared with `declaration_statement` nodes, handled by the `bind_declaration()` function. They can be either in function bodies, or in parameter lists.
- When a variable is declared, we create a symbol table entry, and insert it.
- Variables are used in `variable` nodes, handled by the `bind_variable()` function.
- When we see a usage, we look up the variable in the symbol table, retrieve the entry, and assign the entry to the variable nodes `entry` field.

Variables

The symbol table entries for variables look like:

```
typedef struct {
    int stack_offset, depth;
    char *label;
    data_type_t type;
} symbol_t;
```

- `stack_offset` is the location/address of the variable.
- `depth` can be ignored.
- `label` is the name, `type` the data type.

Stack offset

- The stack offset is essentially the address of the variable (represented as an offset for the stack pointer).
- We'll use this scheme for picking the offsets (it will make sense when we get to the assembly programing):
  - Function parameters have decreasing, positive offsets, the last parameter has offset 8, each parameter has an offset 4 smaller than its predecessor.
  - Method parameters have decreasing, positive offsets, the last parameter has offset 12, each parameter has an offset 4 smaller than its predecessor.
  - Local variables have decreasing, negative offsets, the first local variable has offset -4, each local variable has an offset 4 smaller than its predecessor.
  - Class fields have positive, increasing offsets. The first field has offset 0, each field has an offset 4 larger than its predecessor.
- (Methods are functions inside classes)

Stack offsets

```
                      20          16           12           8
VOID FUNC foo(INT i, INT a, BOOL b, FLOAT c) START
  INT d;                              -4
  d := 5;
  INT e;                        -8
  e := bar(b,d,a);
END
                                 16          12          8
INT FUNC bar(BOOL a, INT b, INT c) START
  INT f;                                -4
  f := b + c;
  BOOL g;                          -8
  INT h;                         -12
  ...
END
```

Stack offsets

```
CLASS Klasse HAS
  INT f;            0
  BOOL g;           4
  INT h;            8

                    20              16             12
  WITH

  INT FUNC bar(INT a, INT b, INT c) START
    INT d;          -4
    FLOAT f;        -8
    ...
  END
END
```

Variables

- We cannot have two variables with the same name in the same function.
- But we can have two variables with the same name if they are in different functions.
- This is implemented by essentially having a separate symbol table for each function, when we enter a function, we create a new symbol table, when we leave the function, we destroy it.
- This should be done with the `scope_add()` and `scope_remove()` functions, whenever a new function is entered, `scope_add()` should be called, and when it is leaved, `scope_remove()` should be called.

Implementation details

- The `symbol_insert()` and `symbol_get()` functions should be used to insert and retrieve variables from the current symbol table.
- The `create_symbol()` function should be used to create a symbol table enty from a declaration node.
- The `stackOffset` argument to the `bind_X` functions should be used to keep track of the stack offset to use. You need to increase/decrease/reset it in various places to make it correct.

Functions

- The next type of symbols is functions, which should be handled differently from variables.
- While variables must be declared before they are used, this is not the case for functions (our language works differently from e.g. C)
- So we have to put all the functions into the symbol table before we go into any of the bodies.
- When we get to a `function_list` node, we have to look at all its children (the functions) and put them into the symbol table, before we continue the traversal.
- This way, whenever we get to a function call, and look up the function, it will be in the table.

Functions

```
VOID FUNC main() START
    INT a;
    a := sum(4,5); // This should work
END

INT FUNC sum(INT a, INT b) START
    RETURN a + b;
END
```

Calling a function before it is declared should work, `sum` must be inserted into the symbol table before we enter the body of `main`.

Functions

The function symbol table entry looks like:

```
typedef struct {
    char* label;
    data_type_t return_type;
    int nArguments;
    data_type_t* argument_types;
} function_symbol_t;
```

- `label` is the function name.
- `return_type` is the, well, return type of the function.
- `nArguments` and `argument_types` is the number of arguments, and an array of the argument data types.

Implementation details

- Functions should be inserted into the symbol table in the `bind_function_list()` function.
- They should be retrieved in the `bind_expression()` function (function calls are a kind of expression). The returned entry should be assigned to the nodes `function_entry` field.
- The functions `function_add()` and `function_get()` should be used to insert and retrieve function symbol table entries.
- (As oposed to variables, there is only one global function symbol table).
- The `create_function_symbol()` function should be used to create a function symbol table entry from a function node.

Implementation details

- As mentioned functions should be inserted into the symbol table in `bind_function_list()`.
- In addition, there is the function `bind_function()`, its purpose is to enable calls to `scope_add()`/`scope_remove()` and reseting the stack offset.

Classes

- The final type of symbols are classes.
- Classes works the same way as variables, a class can only use/reference classes already declared.
- For each class, we need to keep track of the fields and methods it contains, so each class has two private symbol tables, for the fields and methods.
- (In addition there's the symbol table for local variables in mehtods, which works like it does for functions).
- We should be able to call a method in a class from another method before it is declared, (like for functions).

Classes

```
typedef struct {
    int size;
    hash_t* symbols;
    hash_t* functions;
} class_symbol_t;
```

- `size` is the size (in bytes) of objects of this class, this is just 4 times the number of fields.
- `symbols` is a hash table with all the class fields.
- `functions` is a hash table with all the methods.

Class symbol table entries

Before you can insert anyting into the private symbol tables, you must initialize them, calling the ght library, e.g with code like this:

```
// Allocate memory for the entry
class_symbol_t* class_symbol =
    malloc(sizeof(class_symbol_t));
// Initialize the private tables
class_symbol->symbols = ght_create(8);
class_symbol->functions = ght_create(8);
```

The argumet to `ght_create()` is the number of hash buckets, 8 is a safe choice.

Class declarations

- The `bind_class()` function, which handles class nodes, should create and insert class symbol table entries.
- It needs to loop through the `declaration_list` and `function_list` nodes of the class, and create and insert entries in the class' private symbol table. (These nodes should not be handled by `bind_declaration_list()` and `bind_function_list()`.
- It should then recursively continue into the bodies of the methods.
- The functions `class_add()`, `class_insert_field()` and `class_insert_method()` can be used to insert a class entry into the global class symbol table, and insert variable and method entries into the classes private symbol tables.

Class usage

```
a = NEW MyClass; // New expression
a.a = 5; // Field access
a.bar(); // Method call
```

- Classes are used in method call, field access and new expressions, all handled in `bind_expression()`
- `class_get()`, `class_get_symbol()` and `class_get_method()` can be used to retrieve entries form the global class symbol table, and a given class' private symbol tables.

Class usage

- `class_entry` should only be set in the new expression nodes.
- For method calls, `function_entry` should be set, you'll need to retrieve the function/method symbol table entry from the class' private symbol table.
- For field access, `entry` should be set, the variable symbol table entry should be retrieved from the class' private symbol table.
- In all three cases, you'll need the class name. For new, it's in the type after NEW, for method calls and field accesses, its in the symbol table entry of the variable before the `.` (e.g. `a` in the example above).

THIS

- THIS is essentially a variable that is just used and never explicitly declared.
- Hence, when we see it, we cannot retrieve any entry from the symbol table, but have to create it, and assign it to the `entry` field of the THIS node.
- To know what kind of class THIS is, we need to keep track of what class we are currently inside. This should be done with the global variable `thisClass`.
- The offset should always be 8.
- The base type is obviously `CLASS_TYPE`.

Summary

Consider the code:

```
CLASS Klasse1 HAS
  INT a;
  INT b;

  WITH

  INT FUNC sum() START
    RETURN THIS.a + THIS.b;
  END
END

CLASS Klasse2 HAS
  INT d;
  INT c;

  WITH

  VOID FUNC foo() START
  ...
  END

  FLOAT FUNC bar(BOOL a, BOOL b) START
  ...
  END
END
```

```
VOID FUNC main() START
  Klasse1 k1;
  k1 := NEW Klasse1;
  INT b;
  INT c; // POSSITION 1
  c := sum(k1.a, b);
END

INT FUNC sum(INT a, INT b) START
  INT c;
  c := a + b;
  RETURN c; // POSITION 2
END
```

Summary

The state of the symbol tables at possition 1:

Class table:

Name	Size	
Klasse1	8	
Klasse2	8	

Fields:

Name	Offset	Type
a	0	INT
b	4	INT

Methods:

Name	# args	Return
sum	0	INT

Function table:

Name	# args	Return
main	0	VOID
sum	2	INT

Fields:

Name	Offset	Type
d	0	
c	4	

Methods:

Name	# args	Return
foo	0	VOID
bar	2	FLOAT

Variable table:

Name	Offset	Type
k1	-4	CLASS
b	-8	INT
c	-12	INT

Summary

The state of the symbol tables at possition 2:

Class table:

Name	Size	
Klasse1	8	
Klasse2	8	

Function table:

Name	# args	Return
main	0	VOID
sum	2	INT

Variable table:

Name	Offset	Type
a	12	INT
b	8	INT
c	-4	INT

Fields:

Name	Offset	Type
a	0	INT
b	4	INT

Methods:

Name	# args	Return
sum	0	INT

Fields:

Name	Offset	Type
d	0	
c	4	

Methods:

Name	# args	Return
foo	0	VOID
bar	2	FLOAT

Summary entries

The node_t struct has three different fields for storing entries retrieved from the symbol tables:

```
symbol_t *entry;
class_symbol_t* class_entry;
function_symbol_t* function_entry;
```

- `entry` should be used for variables,
- `function_entry` should be used for function and method calls.
- `class_entry` should be used for new expressions.

Example

```
CLASS Inner HAS
  INT a;

  WITH

  VOID FUNC print() START
    PRINT THIS.a;
  END
END

CLASS Outer HAS
  Inner i;

  WITH

  VOID FUNC print() START
    THIS.i.print();
  END
END

VOID FUNC main() START
  Outer o;
  o := NEW Outer;
  o.i := NEW Inner;

  o.i.a := 5; // Position 1
  o.i.print(); // Position 2
END
```

Example

At position 1 and 2, the symbol tables look like

Class table:

Name	Size	
Inner	4	
Outer	4	

Fields:

Name	Offset	Type
a	0	INT

Methods:

Name	# args	Return
print	0	VOID

Function table:

Name	# args	Return
main	0	VOID

Fields:

Name	Offset	Type
i	0	CLASS

Methods:

Name	# args	Return
print	0	VOID

Variable table:

Name	Offset	Type
o	-4	CLASS

Example

Now consider the line:

`o.i.a := 5`

Which has the AST:

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

Example

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

Field access expressions don't follow the usual post order traversal, we never enter the second child. The first thing we do is recursively handle the first child.

Example

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

The first child is also a field access expression, so we do the same thing.

Example

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

"o", stackOffset: -4, type:CLASS("Outer")

Now we're in a variable node. We look it up in the symbol table, and set the `entry` to what we get back.

Example

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

"o", stackOffset: -4, type:CLASS("Outer")

"i", stackOffset: 0, type:CLASS("Inner")

Back up in the field access expression, we look up the second child in the private symbol table of the first child. We assign the entry to both the second child, and the field access expression.

Example

```
ASSIGNMENT_STATEMENT()()
    EXPRESSION()(CLASS_FIELD)
        EXPRESSION()(CLASS_FIELD)
            VARIABLE()("o")()
            VARIABLE()("i")()
        VARIABLE()("a")()
    CONSTANT(INTEGER)(5)()
```

"o", stackOffset: -4, type:CLASS("Outer")

"i", stackOffset: 0, type:CLASS("Inner")

"a", stackOffset: 0, type:INT

And back up in the outer field access expression, we look up the second child in the private symbol table of the first child. We assign the entry to both the second child, and the field access expression. Note that since we assigned the symbol table entry to both the second child and the field access expression above, the code here is identical to that of the inner field access expression.

Type checking

- After the symbol table pass, we'll do yet another tree traversal, to implement type checking.
- This is done by the `typecheck_X` functions in `typecheck.c` and the `typecheck` function pointer.
- In this assignment we'll only do part of the type checking, for some kinds of expressions, the remainder will be done in the next assignment.

Type checking expressions

- Adding only makes sense if both operands are numbers (int or float) and have the same type. (Adding a int to a float would make sense, but would require type casting/conversion). The type out the expression is the same as the type of the operand.
- Logical and and or only makes sense if both operands are booleans. The type of the expression is also a boolean.
- Greater/less comparisons only makes sense when both operands are numbers. The type of the expression is a boolean.
- (These rules are somewhat arbitrary, and depends on the semantics of the language. The type checking stage just enforces whatever rules we decide on).

Typecheking

Complete set of rules for unary and binary expressions.

Operator	Operands	Result
+,-,*,/	INT,INT	INT
	FLOAT,FLOAT	FLOAT
<=,>=,<,>	INT,INT	BOOL
	FLOAT,FLOAT	BOOL
==,!=	INT,INT	BOOL
	FLOAT,FLOAT	BOOL
	BOOL,BOOL	BOOL
- (UNARY)	INT	INT
	FLOAT	FLOAT
!	BOOL	BOOL
&&	BOOL,BOOL	BOOL
||	BOOL,BOOL	BOOL

Implementation details

- The type checking should be done in `typecheck_expression()`
- In addition you'll have to implement `typecheck_default()` which should just continue the traversal.
- All the `typecheck_X` expressions return a type, the type of whatever they are type checking. This is needed for nested expressions. E.g. `5 + 3 + a`;

Error messages

- If you find a type error, you should call `type_error()`. It will print an error message and exit.

Testing

- Stage 6: The nodes encountered during tree traversal.
- Stage 7: The interaction with the symbol tables.
- Stage 8: The data segment of the assembly code.
- Stage 9: The tree including what is stored in the entry pointers (after the bind_names traversal).
- Stage 10: The nodes encountered during type checking. Some of the programs have type errors, and then the expected output is an error message. Some programs, e.g. *euclid.vsl* are correctly typed, but the correct output should still be a type error, because we have not completed all of the type checking.