FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Compilers Labo

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 24 april 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

Prologue

1.1 Using Docker

Each lab has a `files` directory which you should change your working directory to. Run the following command to start an interactive Docker container.

```
$ docker run --rm -it -v "$(pwd)":/files tbesard/compilers:practx
```

1.2 Installing the nano editor

Editing files on a Windows computer brings incompatibility with linux tools. It is recommended to install a text editor on the container.

```
$ apt-get update
$ apt-get install nano
```

Hoofdstuk 2

Lexing

Hoofdstuk 3

Parsing

Hoofdstuk 4

Code Generation

4.1 Setup

Run a docker container and configure the project with CMake.

```
$ docker run --rm -it -v "$(pwd)":/files tbesard/compilers:pract3
$ cd /files
$ cmake .
```

Run `make` to compile the whole project after each change. Use `cheetah` to generate the assembly code.

```
$ ./cheetah test/dummy.c
.globl main
main:
  pushq $1
  popq  %rax
  ...
```

Een executable aanmaken kan met `make dummy`, of als je alle testen wilt compileren kan je `make test` gebruiken.

4.2 Debugging

Met `gdb` kan een executable geïnspecteerd worden.

```
$ gdb ./test/dummy
(gdb) run
```

4.3 Compiler infrastructure

The `codegen.hpp` header defines three important datastructures:

- `Program`: This represents the program that is being emitted, and is accesible as the argument to each `emit` function. It contains a list of `Blocks`.

- `Block`: A block is identified by a label and contains a list of `Instructions`.
- `Instruction`: An instruction has three fields:
  - `name`: the textual representation of the instruction name.
  - `arguments`: a list of arguments.
  - `comment`: an optional string that will be emitted as part of the generated code.

4.4 Emitting code

We will implement the compiler as a stack machine. This means that it should push and pop values onto the stack and only use registers when absolutely neccesary. An explanation of the most useful registers:

- `%rax`: Temporary register, mainly used as the return register.
- `%rbx`: Callee-saved register which can optionally be used as a base pointer.
- `%rbp`: Callee-saved register which can optionally be used as a frame pointer.
- `%rdi`: Used to pass the first argument to functions.
- `%rsi`: Used to pass the second argument to functions.
- `%rdx`: Used to pass the third argument to functions. Can also be used as the second return register.
- `%rcx`: Used to pass the fourth argument to functions.
- `%r8`: Used to pass the fifth argument to functions.
- `%r9`: Used to pass the sixth argument to functions.
- `%r12-r15`: Callee-saved registers.

A short summary of the special registers:

- **Stack Pointer**: The stack pointer points to the top of the stack. All locations beyond the stack pointer are considered to be garbage, and all locations before the stack pointer are considered to be allocated.
- **Frame Pointer**: The area on the stack devoted to local variables, parameters, return address and other temporaries for a function is called the function's `stack frame`. The frame pointer points at the beginning of a stack frame such that the stack pointer can be restored to the frame pointer. Equivalently, the frame pointer contains the value of the stack pointer just before a function is called.
- **Base Pointer**: The base pointer is derived from the stack pointer and is used to travel trough the stack.

Each AST object now has an `emit` function, which purpose is to generate assembly code for that AST object.

4.4.1 Function calls

In `emit.cpp`, complete the implementation of `CallExpr::emit(Program &prog) const` and implement the following features:

- **emit and store arguments**

  ```
  for (size_t i = 0; i < argc; i++){
    args[i]->emit(prog)
  }
  ```

  Remember that this is a method of the class `CallExpr`, so we can use the attributes `args` and `name` of this class. The attribute `args` is of type `ExprList`, which can contain pointers to various expression types such as `FloatLiteral` or `Assignment` (see `expr.hpp`). We will simply call the emit method for each `Expr` in this list.

- **generate a call**

  ```
  prog << Instruction{"call", {name->string}};
  ```

  Here we need to emit a `call` instruction. An instruction has a `name`, a list of `arguments` and optionally a `comment`. The name of the instruction is obviously `call`. In this case the list of arguments only contain one element: the name of the function. We opted to not include a comment here since a `call` instruction is fairly obvious.

- **return a value**

  ```
  if(std::get<0>(decl) == T_void){
    prog << Instruction{"pushq", {"0xABCDEF"}, "void return value"}
  } else if(std::get<0>(decl) == T_int){

  }
  ```

  Our language only has two possible return types: `void` and `int`.

4.4.2 Function Declarations

In `emit.cpp`, complete the implementation of `FuncDecl::emit(Program &prog) const` and implement the following features:

- **The function prologue**:
  - save callee-saved registers
  - set the base pointer
  - align the stack pointer by 16 bytes
- **The function epilogue**:
  - restore the stack pointer
  - restore callee-saved registers