Compilation Labs (2020)

Laure Gonnord & Matthieu Moy & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021

1. LAB: RISCV
2. LAB : ANTLR Startup
3. LAB : Interpreter for MiniC
4. LAB: Direct Code Generation
5. LAB: Code Generation with smart IRs

Content

RiscV startup

MIF08

and a Python tutorial

1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Semantic actions in practice: ANTLR/Lab 2

(ariteval) Input:

```
20 + 22;
a = 4;
a + 2;
a * 5;
```

Output :

```
20+22 = 42
a now equals 4
a+2 = 6
a*5 = 20
```

Code Infrastructure (Python)

```
../TP02/ariteval$ ls
Arit2.g4       test_ariteval.py
testfiles/     arit2.py
Makefile       test_expect_pragma.py
```

- The grammar is written in `Arit2.g4`.
- `arit2.py` the main file (command line handling, . . .).
- `test_ariteval`: unit test script.
- `testfiles/` test files.

Code: Development

- Code and test the grammar (g4 file)
- Use semantic actions (in `Arit2.g4`) :

```
expr returns [int val]:
      e=expr '+' t=term
        {$val=$e.val+$t.val;}
| ... ;
```

- While developing, test **single files** with command line:

```
make ; python3 arit2.py testfiles/blablabla.txt
```

Code: Unit tests.

To test for a (quite) large number of testcases, we will use the `pytest` infrastructure. Test files have the form (expected results in comments):

```
1;
-12;
// EXPECTED
//1 = 1
//-12 = -12
```

and the rest is automatic, for one single file type:

```
python3 arit1.py testfiles/montest.txt
```

and for all tests:

```
make tests
```

- You should write (and deliver) your own test cases !

Grade, Plagiarism

- Part of this lab is graded (Individual work)
- No code sharing allowed for any graded work
- Students sanctionned regularly for plagiarism

1. LAB: RISCV
2. LAB : ANTLR Startup
3. LAB : Interpreter for MiniC
4. LAB: Direct Code Generation
5. LAB: Code Generation with smart IRs

Example 2 ANTLR/Python : MiniC Interpreter (Lab3)

Input: a .c file:

```
int main(){
  float s;
  s=3.14;
  print_float(s);
  return 0;
}
```

Output: on std output:

```
3.14
```

Code Infrastructure

```
>cap-labs20.git/MiniC$ ls
Errors.py  MiniC.g4              README-interpreter.md  test_interpreter.py
Makefile   MiniCInterpreter.py  test_expect_pragma.py  TP03
>cap-labs20.git/MiniC$ls TP03/
MiniCInterpretVisitor.py  MiniCTypingVisitor.py  __pycache__  tests
```

- The grammar of the MiniC language:`MiniC.g4`.
- The main file (command line, driver for the lexer/parser/visitor): `MiniCInterpreter.py`
- Two visitors: one for typing, the other one to evaluate.
- A Makefile, a README.
- Testfiles, and a test script `test_interpreter.py`.

MiniC typing, MiniC visit

- A `MiniCTypingVisitor` to type MiniC programs given, it rejects programs like:

  ```
  int x;
  x="blablabla";
  ```

  $\Rightarrow$ You only have to read the code and play with it to understand how it works.

- A `MiniCInterpretVisitor`, that executes the program. We provide you as an example the arithmetic expression evaluation (and the corresponding test `test00.c`).
  $\Rightarrow$ You have to complete the evaluation for assignments, tests, while.

Visitors

See course 3 or the pdf of the lab. Implement according to grammar rules names:

```
    | expr myop=(PLUS|MINUS) expr          #additiveExpr
```

(used to accept expressions like $43 - 1$ or $40 + 2$). While parsing, this rule will launch the function (in `MiniCInterpretVisitor`):

```
    def visitAdditiveExpr(self, ctx):
[...]
```

which eventually compute the addition/substraction of the two subexpressions.

How to store the interpreter state or the typing environnement?

```
x = 42;        // store the value during assigment (sigma)
print_int(x); // get back this value
```

The store should be global, thus a class variable, here we chose a dictionary: $name \rightarrow value$.

```
class MiniCInterpretVisitor(MiniCVisitor):
    def __init__(self):
        self._memory = dict()
# and somewhere:
    self._memory[name] = value
# and somewhere else:
   val = self._memory[name]
```

Test infrastructure (same as in Lab 2)

You write your testcases and expected results:

```
int main(){
  print_int(3^2+45*(-2/-1));
  print_int(23+19);
  print_int(false || 3 != 7)
  print_string("coucou");
  return 0;}
// EXPECTED
// 99
// 42
// 1
// coucou
```

```
int main(){
  int u; bool b;
  u=3;  b=true;
  if (b) { u=u+1; }
  else { u=u-1; }
  print_int(u);
  return 0;}
// EXPECTED
// 4
```

⇒ a helper script (using `pytest`) compares the actual and the expected outputs.

Test infrastructure 2/2

```
================================== test session starts =============================
platform linux -- Python 3.7.3, pytest-3.10.1, py-1.7.0, pluggy-0.8.0 -- /usr/bin/py
cachedir: .pytest_cache
rootdir: /home/laure/Documents/VCS/Teaching/compil-lyon/TP2019-20/TP03/MiniC-type-in
, inifile:
plugins: cov-2.7.1
collected 7 items

test_evaluator.py::TestEval::test_eval[./ex/test00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/double_decl00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type01.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type_bool_bool.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type03.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type02.c] PASSED

================================ 7 passed in 0.49 seconds ===========================
```

$\Rightarrow$ Using this test framework is mandatory.

1. LAB: RISCV
2. LAB : ANTLR Startup
3. LAB : Interpreter for MiniC
4. LAB: Direct Code Generation
5. LAB: Code Generation with smart IRs

Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;}
```

Output: a RISCV file:

```
[...]
        ;; (stat (assignment n = (expr (atom 6)) ;))
        LI t1, 6        ; t1 is a riscv register.
        MV t2, t1
[...]
```

Code Generation, first step

- 3-address code generation according to the code generation rules of the course:

```
// e1+e2 code generation rule
  temp_1 <- GenCodeExpr(e_1)
  temp_2 <- GenCodeExpr(e_2)
  dest_tmp <- new_tmp()
  code.add(InstructionADD(dest_tmp, temp_1, temp_2))
  return dr
```

- **TODO: implement them**:

```
tmpl = self.visit(ctx.expr(0))
tmpr = self.visit(ctx.expr(1))
dest_tmp = self._current_function.new_tmp()
if ctx.myop.type == MuParser.PLUS:
    self._current_function.addInstructionADD(dest_tmp, tmpl,
        tmpr)
```

Result after first step

The previous step uses instructions of an API like:

> self._current_function.addInstructionADD(dr, tmpl, tmpr)

whose side effect is to construct a RISCV prog as a list of 3 adresses instructions with temporaries (virtual registers, from the class `Temporary`.

This list can be dumped (with `printCode` in the API) into a .s file:

```
;; (stat (assignment n = (expr (atom 6)) ;))
li temp_1, 6
mv temp_2, temp_1
```

We cannot test: it is not executable!

Code Generation, second step

The allocation process:

- takes as input the preceding result
- modifies the list of instructions with temporaries into list of instructions with physical registers or accesses to memory.
- a trivial allocator is given.

TODO : all in memory allocation (see course)

Code Infrastructure (only files for THIS LAB))

```
MiniC$ ls
Makefile MiniC.g4 test_codegen.py MiniCC.py [...]

MiniC$ ls TP04/
APIRiscV.py        libprint.s                 Operands.py   tests
Instruction3A.py   MiniCCodeGen3AVisitor.py   printlib.h
```

- The MiniC grammar in `MiniC.g4`, a Makefile, as usual.
- Unit tests in `test_codegen.py`.
- API for generating RISCV code : `APIRiscV`, ...
- `Allocations.py`: allocators for RISCV code.
- **TODO : edit and fill `MiniCCodeGen3AVisitor.py` mainly**. You may have other changes to make in other files (Allocation).

RISCV API

In this API (APICodeRISCV, Instruction3A, Operands) :

- A class for a program RISCV `RiscVFunction`. The program contains a list of instructions, methods to add instructions, to increment temporary numbers, ...
- Classes for instructions: `Instruction`, `Instru3A`, `Label`
- A 3 address instruction contains arguments that can be `Immediate`, `Temporary`, `Register`, or a `Condition` in the special case of the condjump. ...
- the `CondJump` instruction (label,dr1,cond,dr2) has the meaning: `if (dr1 cond dr2) jump to label`
- Ignore code concerning graphs (print dot, add edges) and dataflow (in and out sets), this is for next lab.

Allocations/replace

In `TP05/Allocations.py`:

- `replace_*` functions replace temporary operands of a given instruction with the help of the current allocation (see the example for naive allocation).
- The allocation itself is done before in Allocator classes : `*Allocator` (ignore the smart allocation in Lab 4).

tests

While developing, write appropriate (mini) tests and use :

```
python3 MiniCC.py --reg-alloc=xxx /path/to/example.c
```

and have a look at the generated file.

Next step is to verify everything:

```
make tests
```

to launch all tests in `tests*/*` files (this setting is in `test_codegen.py`.

1. LAB: RISCV
2. LAB : ANTLR Startup
3. LAB : Interpreter for MiniC
4. LAB: Direct Code Generation
5. LAB: Code Generation with smart IRs

Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;
}
```

Output: a RISCV file:

```
[...]
        ;; (stat (assignment n = (expr (atom 6)) ;))
        li t6, 6
        mv t7, t6
[...]
```

Big picture

- Construct the CFG (already done)
- Compute liveness information:
  - **TODO** initialize GEN and KILL)
  - **TODO** ENSL Only fixpoint computation.
- Compute the interference graph (**TODO**: interfere function)
- Color it (**TODO** call an API method)
- Allocation: temps in registers, splilled temps in memory (**TODO**)
- Rewrite instructions wrt the allocation. (**TODO**)
- Pretty-print code (automatic)
- Test.

Liveness and interference graph

TODO for liveness, in `TP05/Allocations.py`

- Initialize dataflow sets (function `set_gen_kill`)
- ENSL Only implement the fixpoint computation. (function `run_dataflow_analysis`)
- Implement a `interfere` function, and complete `build_interference_graph`.

Coloring / Smart Allocator

TODO for coloring, in `TP05/Allocations.py`

Now for the coloring, in `smart_alloc`

```
(coloringreg, _, _)
      = self._igraph.color();
```

(calls the `Libgraph` coloring function).

then **TODO implement smart allocator**: (in smartalloc):

- if a temporary has a “register color” (`color < len(GP_REGS)`), allocate in a physical register.
- else in stack with an offset computed from the color.

Do not forget to implement `replace_smart`

Tests

Write **appropriate tests**; then run:

```
make tests
```

It launches tests for the dataflow, and for smart alloc and smart codegen.