Norwegian University of Science and Technology

Compiler Construction

Practical Exercise 6: Code generation

Guidelines and hints

Michael Engel

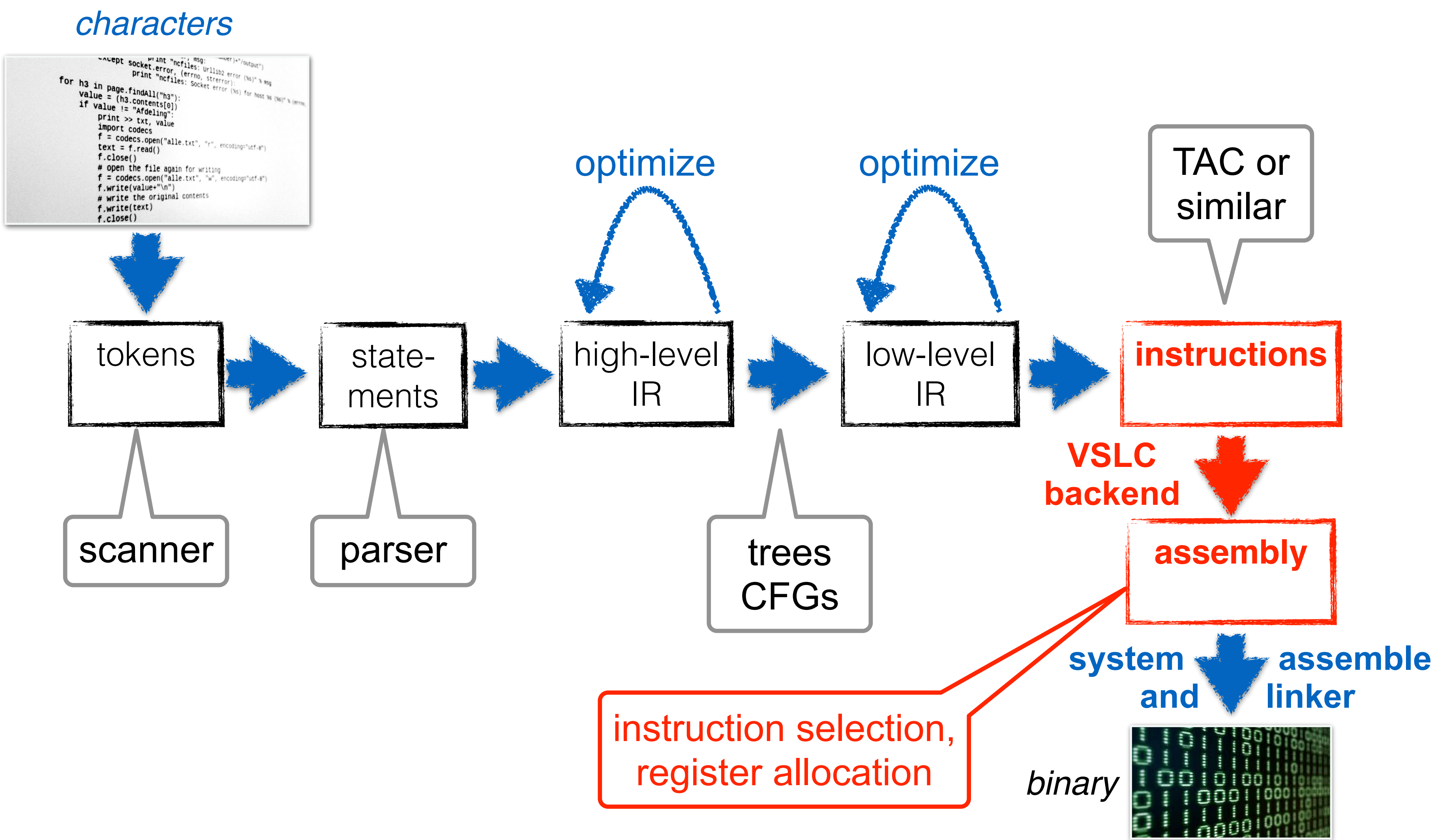
Topics related to code generation

- Part 1
  - Global String Table
  - Global variables
  - Functions
  - Function parameters
  - Arithmetic expressions
  - Arithmetic statements
  - Assignment statements
  - print statements
  - return statements

Topics related to code generation

- Part 2
  - Local Variables
  - Function calls
  - Conditionals (if and relations)
  - While loops
  - Continue (Null statement)

Compiler backend



Let's look at "Hello, world"

VSL source code "prog.vsl"

```
def hello()
begin
    print "Hello, world"
    return 0
end
```

x86-64 assembler "prog.s" **on Linux**

```
.data                          ; data section
strout:   .asciz "%s"
STR0:     .asciz "Hello, world !"

.text                          ; text section
.globl main ; main (not here) would be global
_hello:     ; function hello with prepended '_'
    pushq %rbp                 ; save frame ptr
    movq  %rsp, %rbp           ; rsp -> rbp
    movq  $STR0, %rsi          ; param 2 -> rsi
    movq  $strout, %rdi; param 1 -> rdi
    call  printf               ; call libc printf!
    movq  $'\n', %rdi
    call  putchar
    movq  $0, %rax             ; return value in rax
    movq  %rbp, %rsp           ; restore frame ptr
    ret                        ; return from hello
```

Assemble to .o file and link:

```
$ as -c -o prog.o prog.s
$ cc -o prog prog.o
```

The "cc" line does *not* call the C compiler, but only the linker – it automatically links libc

Caution!

- Many x86-64 assembler examples for Linux on the net use the "nasm" assembler!
  - This has an incompatible syntax (intel syntax)!
  - Especially the order of parameters is reversed…

- We use a version of the binutils assembler (in gcc or the clang project)
  - This uses Unix ("AT&T") assembler syntax

Is this a working program?

```
.data
strout:   .asciz "%s"
STR0:     .asciz "Hello, world !"

.text
.globl main
main:
    call _hello
    ret
_hello:
    pushq %rbp
    movq  %rsp, %rbp
    movq  $STR0, %rsi
    movq  $strout, %rdi
    call  printf
    movq  $'\n', %rdi
    call  putchar
    movq  $0, %rax
    movq  %rbp, %rsp
    ret
```

```
$ as -c -o prog.o prog.s
```

```
$ cc -o prog prog.o
/usr/bin/ld: prog.o:
relocation R_X86_64_32S against
`.data' can not be used when
making a PIE object; recompile
with -fPIE
collect2: error: ld returned 1
exit status
```

Many assembler examples you can find online use `movq` instructions to load the absolute address of a string (or other parameter).

This is no longer allowed on x86-64!

More fixes…

```
.data
strout:   .asciz "%s"
STR0:     .asciz "Hello, world !"

.text
.globl main
main:
    call _hello
    ret
_hello:
    pushq %rbp
    movq  %rsp, %rbp
    leaq  STR0(%rip), %rsi
    leaq  strout(%rip), %rdi
    call  printf
    movq  $'\n', %rdi
    call  putchar
    movq  $0, %rax
    movq  %rbp, %rsp
    ret
```

```
$ as -c -o prog.o prog.s
$ cc -o prog prog.o
```

```
$ ./prog
Segmentation fault (core dumped)
$
```

The correct way to load the address of a variable (here: string parameters for printf) is to use a *rip-relative* load instruction.
This encodes the offset of the string to the current rip of the instruction and turns it into an absolute address.

However, the program still crashes!

A working program!

```
.data
strout:   .asciz "%s"
STR0:     .asciz "Hello, world !"

.text
.globl main
main:
    call _hello
    ret
_hello:
    movq  %rsp, %rbp
    subq  $8, %rsp
    pushq %rbp
    leaq  STR0(%rip), %rsi
    leaq  strout(%rip), %rdi
    call  printf
    movq  $'\n', %rdi
    call  putchar
    movq  $0, %rax
    movq  %rbp, %rsp
    ret
```

```
$ as -c -o prog.o prog.s
$ cc -o prog prog.o
```

```
$ ./prog
Hello, world!
$
```

The problem here is the *stack alignment* requirement of x86-64: The stack pointer %rsp must be aligned to a multiple of 16 bytes when calling a libc function!

%rsp is aligned to 16 bytes when entering main, but then `call _hello` pushes 8 bytes to the stack.

We fix the alignment by subtracting 8 from %rsp

What else do we need?

- The main function
- Global variables?
  - they need mutable memory
- Arguments?
- Expressions
- Assignments
- Expressions in print statements

Main function

- Remember the calling convention from the x86-64 assembler lecture on instruction set
  - The first 6 args go into registers %rdi %rsi %rdx %rcx %r8 %r9
  - Further args in the stack
  - Stack will need 16 byte alignment
- All arguments (in VSL) are 64-bit integers

- However, `main(argc, argv)` is called in a different way:
  - called from the libc startup code crt0
  - 1st argument: number of command line args (int `argc`)
  - 2nd argument: pointer to a list of char-pointers (char **`argv`)

A generic 'main' for VSL programs

- At runtime this has to be done:
  - Find the count of arguments
  - If there are some translate them from text to numbers
  - Put them in the right places for an ordinary call
  - Call the 1st function defined in the VSL source program
  - Take the return value from that and return it to the calling shell
  - Return to shell

Generate main is provided

- A function to generate main will be supplied.
  - This will simply generate assembly to point to the symbol t that is the first defined function in the program.
- It expects the global names to be prefixed with the in the generated assembly
- It will fail if the shell provides an argument count that does not match that of the starting function in the source program
- A hard coded main to prevent the assembler from giving errors is also available. Replace that part such that you start off with the symbol t you supply

Generating the string table

- A generate_stringtable function is provided which prints the following:

```
.data
intout: .asciz "%ld"
strout: .asciz "%s"
errout: .asciz "Wrong number of arguments"
```

  - `errout` is only needed by main
  - `intout` and `strout` are handy for printing numbers and strings when translating ”print” statements
  - The contents of the data section is still missing from the source. Generate them them here with numbered labels like STR0:, STR1:,...
- Note the we do not generate a `.rodata` section here as would be expected – you can do this on Linux, but the macOS assembler complains…

Mutable Memory for Global Variables

- For global variables you need mutable memory. What can be done for this is as follows:
  - Emit a ”`.data`” section (mutable)
  - Put labels under it for the global vars, such as ”`x:`” for variable ”x”
  - Place a 64-bit zero value at that address, for the program to change at run time (the ’zero’ directive takes a byte count) i.e. `x: .zero 8`
  - In this way a reference to global variable ’x’ is translated as an access to ’x’

Arguments

- The first six arguments to a function reside in registers
- For convenient reference the call convention order is placed in a static string array ’`record[6]`’ which contains strings with the register names in order
- For function calls these registers will change values
- Copies of the arguments can be placed on stack as the first thing a function does
  - They then have an address of `%rbp + 8*argument index`

Stack alignment

- Accessing arguments takes place relative to the base pointer `%rbp` from the bottom up
- Every argument and local variable consumes 8 bytes
  - ***pushing an odd number creates a stack misalignment***
- Pad it with 8 bytes if required (prevents crashing of generated system calls (such as printf)

Expressions

- Treat the process as a stack machine when generating code
- Let `%rax` contain values and results
  - Numbers translate into a `movq` of the constant into `%rax`
  - Variables translate to copying their contents into `%rax`
- Operations are translated recursively
  - Recursively generate subexpression 1 (put the result in `%rax`)
  - Push result
  - Step 1 for subexpression 2
  - Combine the result with the top of stack element to obtain the result of the operation
  - Remove the temporary result of subexpression 1 from stack
  - Result is in `%rax` and the stack is restored
- Be aware of the multiply and divide instructions

Assignments

- Using the stack approach, assignments are implemented by simply generating the code for the RHS expression and moving the result in %rax to the location of the assignment destination

Printing

- Parameters to print are a list containing strings, numbers, identifiers and expressions
  - You can break this down to:
    - Generate code to print the 1st element
    - Same as above for the second and so on…
    - Generate code to print a new line character
  - The effect of the print statement is a concatenation of its parameters
- Iterate over the list of print items as follows:
  - Strings: setup and call `printf` with parameters `strout` and the string
  - Numbers: setup and call `printf` with `intout` and number
  - Identifiers: setup and call `printf` with `intout` and the contents of the identified address
  - Expressions: Generate the expression, setup and call `printf` with parameters `intout` and the contents of `%rax`

More fun…

- We still need to generate code for:
  - Local Variables
  - Function calls
  - Conditionals
  - Loops
  - Continue

Local variables

- Local variables are not accessed in the same way as the global variables
  - They go on the run-time stack.
  - Their sequence number can be used to find their offset from the base pointer
- Begin a function by creating space for local variables on the stack
  - Remember the 16 byte stack alignment!
- Local variables were counted in the process of generating the symbol table – use the sequence number as an index
- Otherwise local variables can go into expressions in the same manner as global variables

Function calls

- They appear in expressions

- Generating function calls requires you to follow the x86-64 calling conventions*:
  - Put first 6 arguments into their designated registers
  - additional arguments have to be put on the stack
  - Call the function
  - Restore the stack and return the result value in `%rax`

* you could define your own calling conventions, but this would just be extra effort and make interfacing to C code more difficult…

Conditionals (if and relations)

- Relation is generate in same manner as arithmetic expressions
  - Recursively generate code to evaluate the left expression, leaving the result in `%rax`
  - Put the result on the stack
  - Generate code to evaluate the right expression
  - Get previous result from the stack
  - Compare and jump as needed

Jumping

- The expression: ”`if (a=b) then A else B`” can be turned to

```
     evaluate a
     evaluate b
     compare
     jump-if-not-equal ELSE
     (code for "then" part A)
     jump ENDIF
ELSE:
     (code for "else" part B)
ENDIF:
     (rest of the program)
```

- This needs a numbering scheme – since the conditional statements can be nested, we need a stack to push and pop the counter values from the numbering scheme. This will track the nesting

Loops

- These are also treated like conditionals
- `while (condition) expression` becomes:

```
WHILELOOP:
        evaluate condition
        jump-if-false ENDWHILE
        code for expression
        jump WHILELOOP
ENDWHILE:
        (rest of the program)
```

- Treat the loops in the same way as IFs for nesting
  - i.e. implement a numbering scheme
  - Use a separate stack for loops

Continue

- Continue-Statements skip directly to the condition evaluation of the while loop
    - If a shared counting scheme for WHILEs and IFs is used, then the enclosing construct could be an IF
    - With separate stacks, the index of the enclosing while loop is on the stack top of the while stack

Linux vs. macOS

x86-64 assembler **on Linux**

```
    .data
strout:  .asciz "%s"
STR0:    .asciz "Hello, world!"
    .text
    .globl      main
main:
    call _hello
    ret
_hello:
    movq  %rsp, %rbp
    subq  $8, %rsp
    pushq %rbp
    leaq  STR0(%rip), %rsi
    leaq  strout(%rip), %rdi
    call  printf
    movq  $'\n', %rdi
    call  putchar
    movq  $0, %rax
    mov   %rbp, %rsp
    ret
```

x86-64 assembler **on macOS (11.2)**

```
    .data
strout:  .asciz "%s"
STR0:    .asciz "Hello, world!"
    .text
    .globl      _main
_main:
    call _hello
    ret
_hello:
    movq  %rsp, %rbp
    subq  $8, %rsp
    pushq %rbp
    leaq  STR0(%rip), %rsi
    leaq  strout(%rip), %rdi
    call  _printf
    movq  $'\n', %rdi
    call  _putchar
    movq  $0, %rax
    mov   %rbp, %rsp
    ret
```

macOS differences

- Tested against
  - macOS 11.2 "Big Sur"
  - macOS 10.14 "Mojave"
- We use `.data` as the section for strings instead of `.rodata` since the macOS assembler doesn't recognize `.rodata`
- Names of libc functions to be called and main have to be prepended by an underscore '_'
  - Use '__' (two underscores) for your own functions if this causes conflicts!
- Linking takes a bit more effort:

```
        .data
strout:   .asciz "%s"
STR0:     .asciz "Hello, world!"
        .text
        .globl      _main
_main:
        call _hello
        ret
_hello:
        movq  %rsp, %rbp
        subq  $8, %rsp
        pushq %rbp
        leaq  STR0(%rip), %rsi
        leaq  strout(%rip), %rdi
        call  _printf
        movq  $'\n', %rdi
        call  _putchar
        movq  $0, %rax
        mov   %rbp, %rsp
        ret
```

```
$ as -arch x86_64 -o prog.o prog.S
$ ld -arch x86_64 prog.o -o prog -lSystem -syslibroot \
      `xcrun -sdk macosx --show-sdk-path`
```

But… what's with my shiny new M1???

Program translated to Aarch64 **on macOS 11**

```
prog.aot:
(__TEXT,__text) section
_main:
00001000 adrp x24, -14 ; 0xffffffffff3000
00001004 add  x24, x24, #0xf2b
00001008 adr  x25, #0x10
0000100c stp  x24, x25, [x21, #-0x10]!
00001010 str  x24, [x4, #-0x8]!
00001014 bl   _hello
00001018 ldr  x22, [x4], #0x8
0000101c ldp  x23, x24, [x21], #0x10
00001020 sub  x25, x22, x23
00001024 cbnz x25, 0x102c
00001028 ret  x24
0000102c bl   0x352c
_hello:
00001030 mov  x5, x4
00001034 subs x4, x4, #0x8
00001038 str  x5, [x4, #-0x8]!
0000103c adrp x7, -9 ; 0xffffffffff8000
00001040 add  x7, x7, #0x20
00001044 adrp x24, -14 ; 0xffffffffff3000
00001058 bl   0x112c
…
```

- Apple has switched to ARM (Aarch64) CPUs… not all systems right now
- You can compile and link x86-64 programs using the Xcode command line tools
- They can also be executed
  - It "simply works"!
  - x86-64 code is statically translated to Aarch64 code by the Rosetta 2 translation system!
- You could also try writing an Aarch64 backend 🙂