CMSC 430: Introduction to Compilers

Evildoer: Calling external functions

Evildoer: Calling external functions

- Evildoer adds a mechanism for interacting with the outside world. It will be able to read and write a byte of information at a time (i.e. an integer between 0 and 256) from the standard input port and output port, respectively.
- Evildoer adds the following operations:
  - write-byte : Byte -> Void: writes given byte to stdout, produces nothing.
  - read-byte : -> Byte or EOF: reads a byte from stdin, if there is one, EOF otherwise.
  - peek-byte : -> Byte or EOF: peeks a byte from stdin, if there is one, EOF otherwise.

Evildoer: Calling external functions

- Evildoer adds the following values:
  - eof : EOF bound to the end-of-file value, and
  - void : -> Void a function that produces the void value.
- adds a predicate that recognizes end-of-file value:
  - eof-object? : Any -> Boolean: determines if argument is the eof value.
- adds a sequence construct:
  - (begin e0 e1): evaluates e0, then e1.

Evildor: Syntax

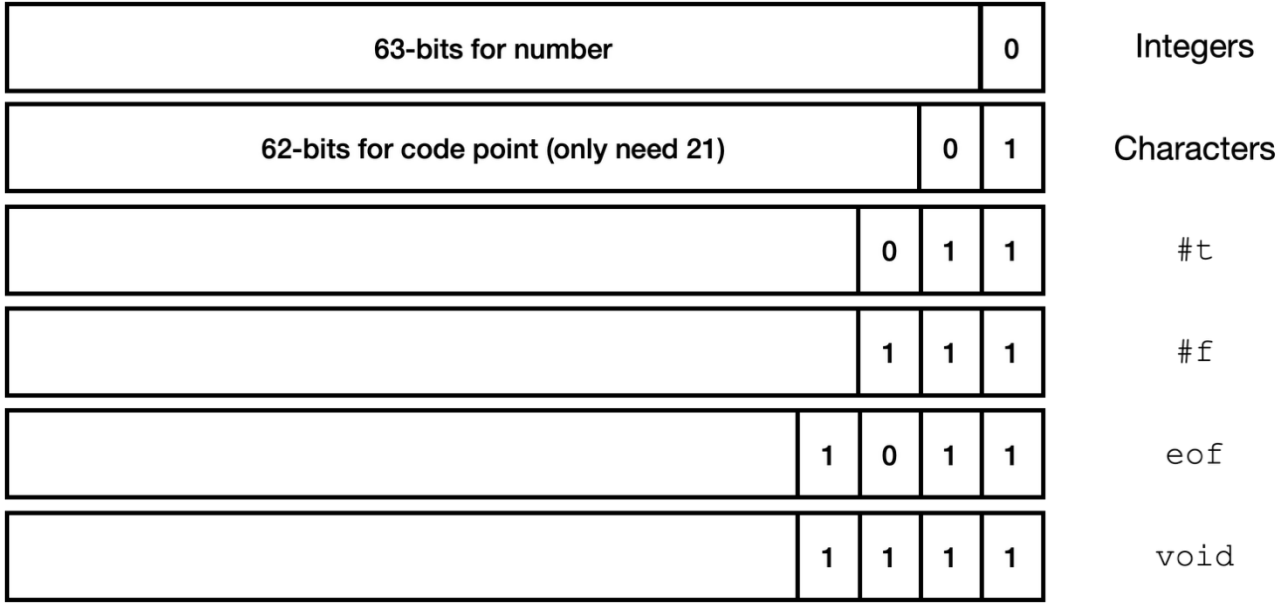
Concrete syntax

```
(begin e1 e2)
(read-byte)
(peek-byte)
(void)
(write-byte e)
(eof-object? e)
```

Abstract syntax

```
(Begin e1 e2)
(Prim0 'read-byte)
(Prim0 'peek-byte)
(Prim0 'void)
(Prim1 'write-byte e)
(Prim1 'eof-object? e)
```

Encoding Values in Evildoer



System V ABI Calling Convention

- The stack must be aligned to a 16-byte boundary when calling a function
- The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9
- Volatile (caller-saved): RAX, RCX, RDX, R8, R9, R10, R11
- nonvolatile (callee-saved): RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15

Example

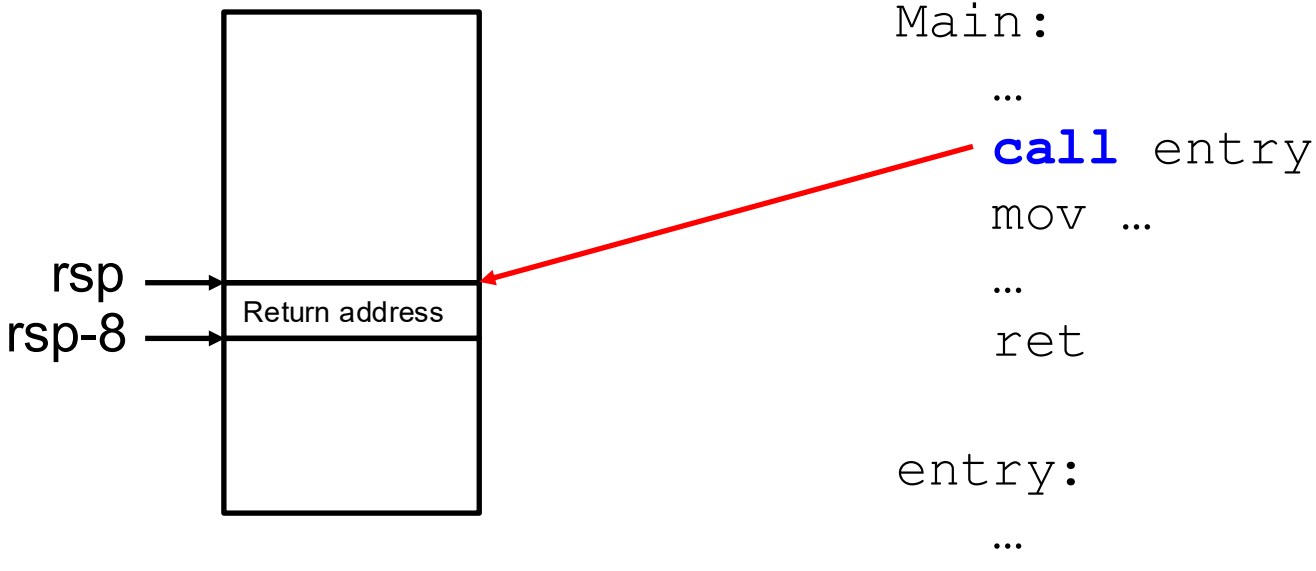
```
#include <stdio.h>
#include <inttypes.h>
int64_t entry();
int main(int argc, char** argv) {
  printf("%" PRId64 "\n", entry());
  return 0;
}
```

```
.intel_syntax noprefix
        .text
        .global
_entry:
         sub rsp, 8
         mov rdi, 21
         call _dbl
         add rsp, 8
         ret
```

```
#include <inttypes.h>
int64_t dbl(int64_t x) {
return x + x;
}
```

16-byte stack alignment

x86-64 System V ABI requires the stack is 16-byte aligned just before the call instruction is called.



P: 16-byte aligned

io.c

```
val_t read_byte(void)
{
  char c = getc(in);
  return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);
}

val_t peek_byte(void)
{
  char c = getc(in);
  ungetc(c, in);
  return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);
}

val_t write_byte(val_t c)
{
  putc((char) val_unwrap_int(c), out);
  return val_wrap_void();
}
```

How to test programs that do I/O?

- How do we test (read-byte)?

```
(check-expect (run '(read-byte)) ???)
```

- What to expect depends on the state of the input port
- Value is no longer a function of the expression alone

Interpreting I/O operations

```
(interp (parse '(read-byte))      ;; waits for input
(interp (parse '(write-byte 97)) ;; writes 'a'
```

We can use Racket facilities to redirect I/O to make functional tests:

```
(interp/io (parse '(read-byte)) "a") => '(97 . "")
(interp/io (parse '(write-byte 97)) "")
   => '(#<void> . "a")
```

Interpreting I/O operations

We can use Racket facilities to redirect I/O to make functional tests:

```
(interp/io (parse '(read-byte)) "a") => '(97 . "")

(interp/io (parse '(write-byte 97)) "")

   => '(#<void> . "a")
```

- State of input port given explicitly
- State of output port produced explicitly
- Behavior of program is a (mathematical) function once again

Compiling I/O operations

We can use run/io (compiles and sets up run-time appropriately):

```
(run/io (compile (parse '(read-byte)) "a"))
      => '(97 . "")

(run/io (compile (parse '(write-byte 97)) ""))
   => '(#<void> . "a")
```

Correctness

```
;; Expr String -> Void
(define (check-compiler e i)
  (let ((r (with-handlers ([exn:fail? identity])
    (interp/io e i))))
    (unless (exn? r)
    (check-equal? r (exec/io e i)))))
```

- If interpreter errors, behavior is unspecified
- Program is interpreted and executed with same input
- Checks that both value and output match