Introduction to Compiler Construction

Compilation: Overview of *iota* to Marvin Compiler

Outline

1. The *iota* Compiler
2. *iota* Programs
3. The Marvin Machine
4. Marvin Programs

The iota Compiler

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The base *iota* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The base *iota* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/iota/src/iota` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `I*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating intermediate JVM code
- `N*.java` files for generating Marvin code
- Other supporting Java files

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The base *iota* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/iota/src/iota` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `I*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating intermediate JVM code
- `N*.java` files for generating Marvin code
- Other supporting Java files

The directory `$j/iota/bin` contains wrapper scripts

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The base *iota* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/iota/src/iota` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `I*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating intermediate JVM code
- `N*.java` files for generating Marvin code
- Other supporting Java files

The directory `$j/iota/bin` contains wrapper scripts

The directory `$j/iota/tests` contains test programs

The iota Compiler

iota is a compiler for a simple procedural language that supports methods, statements, and expressions

The base *iota* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/iota/src/iota` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `I*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating intermediate JVM code
- `N*.java` files for generating Marvin code
- Other supporting Java files

The directory `$j/iota/bin` contains wrapper scripts

The directory `$j/iota/tests` contains test programs

The file `$j/iota/build.xml` is the Ant build configuration file

The iota Compiler

The iota Compiler

To compile the compiler, run

```
>_ ~/workspace/iota

$ ant
```

The iota Compiler

To compile the compiler, run

```
>_ ~/workspace/iota
$ ant
```

Usage syntax for the compiler

```
>_ ~/workspace/iota
$ ./bin/iota
Usage: iota <options> <source file>
Where possible options include:
  -g  Allocate registers using graph coloring method; default = naive method
  -v  Display intermediate representations and liveness intervals
  -d  <dir> Specify where to place output (.marv) file; default = .
```

The iota Compiler

The iota Compiler

To compile an *iota* program `$j/iota/tests/Factorial.iota` for the Marvin architecture, run

```
>_ ~/workspace/iota
$ ./bin/iota tests/Factorial.iota
```

The iota Compiler

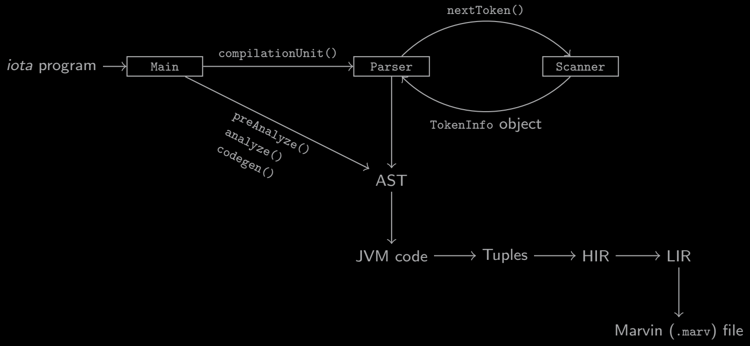
To compile an *iota* program `$j/iota/tests/Factorial.iota` for the Marvin architecture, run

```
>_ ~/workspace/iota
$ ./bin/iota tests/Factorial.iota
```

To run the generated Marvin program `Factorial.marv`, run

```
>_ ~/workspace/iota
$ python3 ./bin/marvin.py Factorial.marv
5
120
```

The *iota* compiler, like *j--*, is organized in an object-oriented fashion



`Combinations.iota`

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^{n}C_k = \frac{n!}{k!(n-k)!}$

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^nC_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ _
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^nC_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^nC_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ _
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^nC_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ python3 ./bin/marvin.py Combinations.marv
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^{n}C_{k} = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ python3 ./bin/marvin.py Combinations.marv
5
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^{n}C_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ python3 ./bin/marvin.py Combinations.marv
5
-
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^nC_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ python3 ./bin/marvin.py Combinations.marv
5
3
```

Combinations.iota

Standard input	n (int) and k (int)
Standard output	number of unordered choices of k items out of n unique items, ${}^{n}C_k = \frac{n!}{k!(n-k)!}$

~/workspace/iota

```
$ ./bin/iota tests/Combinations.iota
$ python3 ./bin/marvin.py Combinations.marv
5
3
10
$ _
```

Combinations.iota

```
// Returns n! computed iteratively.
int factorial(int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}

// Entry point.
void main() {
    int n = read();
    int k = read();
    int nFac = factorial(n);
    int kFac = factorial(k);
    int nMinusKFac = factorial(n - k);
    write(nFac / (kFac * nMinusKFac));
}
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ _
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
$ _
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
$ python3 ./bin/marvin.py Factorial.marv
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
$ python3 ./bin/marvin.py Factorial.marv
_
```

Factorial.iota

Standard input	n (int)
Standard output	$n!$ (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
$ python3 ./bin/marvin.py Factorial.marv
5
```

Factorial.iota

Standard input	*n* (int)
Standard output	*n*! (computed recursively)

~/workspace/iota

```
$ ./bin/iota tests/Factorial.iota
$ python3 ./bin/marvin.py Factorial.marv
5
120
$ _
```

iota Programs · Factorial

Factorial.iota

```
// Returns n! computed recursively.
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

// Entry point.
void main() {
    int n = read();
    write(factorial(n));
}
```

The Marvin Machine

The Marvin Machine

Marvin simulates a computer that has sixteen 16-bit registers and 65,536 32-bit words of main memory (RAM)

The Marvin Machine

Marvin simulates a computer that has sixteen 16-bit registers and 65,536 32-bit words of main memory (RAM)

In addition to the sixteen registers, Marvin has a 16-bit program counter `pc` and a 32-bit instruction register `ir`

The Marvin Machine

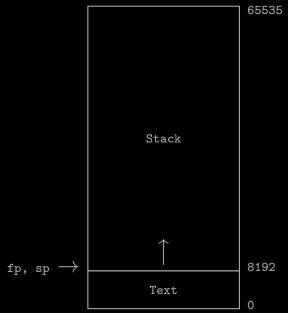
Marvin simulates a computer that has sixteen 16-bit registers and 65,536 32-bit words of main memory (RAM)

In addition to the sixteen registers, Marvin has a 16-bit program counter `pc` and a 32-bit instruction register `ir`

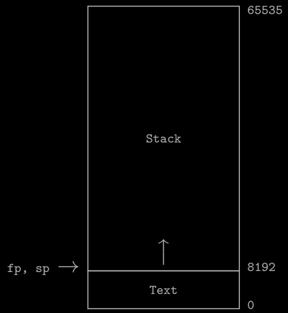
The sixteen registers

- `r0` – `r11` are general purpose registers
- `r12` is reserved to store the return address (`ra`) of the calling subroutine (aka function)
- `r13` is reserved to store the return value of a subroutine
- `r14`, called the frame pointer (`fp`), is reserved to store the base address of the most recent frame on the stack
- `r15`, called the stack pointer (`sp`), is reserved to store the address of the top of the stack

The machine's main memory is divided into a text segment and a stack segment

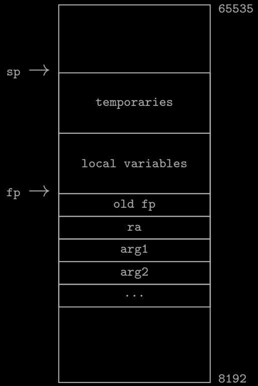


The machine's main memory is divided into a text segment and a stack segment



A Marvin program (ie, a `.marv` file) is assembled and loaded into the text segment starting at address 0

When a subroutine is called, a stack frame must be created for it on the stack



Marvin supports 32 instructions, each of which accepts between 0 and 3 arguments

Marvin supports 32 instructions, each of which accepts between 0 and 3 arguments

System instructions

`halt`	`00000000 00000000 00000000 00000000`	stops the machine
`read rX`	`00000001 00000000 00000000 0000XXXX`	sets `rX = N`, where `N` $\in [-2^{15}, 2^{15} - 1]$ read from standard input
`write rX`	`00000010 00000000 00000000 0000XXXX`	writes `rX` to standard output
`nop`	`00000011 00000000 00000000 00000000`	does nothing

Marvin supports 32 instructions, each of which accepts between 0 and 3 arguments

System instructions

`halt`	`00000000 00000000 00000000 00000000`	stops the machine
`read rX`	`00000001 00000000 00000000 0000XXXX`	sets `rX = N`, where `N` $\in [-2^{15}, 2^{15}-1]$ read from standard input
`write rX`	`00000010 00000000 00000000 0000XXXX`	writes `rX` to standard output
`nop`	`00000011 00000000 00000000 00000000`	does nothing

Arithmetic instructions

`neg rX rY`	`00001001 00000000 00000000 XXXXYYYY`	sets `rX = -rY`
`add rX rY rZ`	`00001010 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY + rZ`
`sub rX rY rZ`	`00001011 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY - rZ`
`mul rX rY rZ`	`00001100 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY * rZ`
`div rX rY rZ`	`00001101 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY // rZ`
`mod rX rY rZ`	`00001110 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY % rZ`

Jump instructions

`jumpn N`	`00001111 00000000 NNNNNNNN NNNNNNNN`	jumps to instruction `N`
`jumpr rX`	`00010000 00000000 00000000 0000XXXX`	jumps to `rX`
`jeqzn rX N`	`00010001 0000XXXX NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX == 0`
`jnezn rX N`	`00010010 0000XXXX NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX != 0`
`jgen rX rY N`	`00010011 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX >= rY`
`jlen rX rY N`	`00010110 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX <= rY`
`jeqn rX rY N`	`00010100 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX == rY`
`jnen rX rY N`	`00010101 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX != rY`
`jgtn rX rY N`	`00010111 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX > rY`
`jltn rX rY N`	`00011000 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX < rY`
`calln rX N`	`00011001 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = pc + 1` and jumps to instruction `N`

Instructions for setting register data

`set0 rX`	`00000100 00000000 00000000 0000XXXX`	sets `rX = 0`
`set1 rX`	`00000101 00000000 00000000 0000XXXX`	sets `rX = 1`
`setn rX N`	`00000110 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = N`, where `N` $\in [-2^{15}, 2^{15} - 1]$
`addn rX N`	`00000111 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = rX + N`, where `N` $\in [-2^{15}, 2^{15} - 1]$
`copy rX rY`	`00001000 00000000 00000000 XXXXYYYY`	sets `rX = rY`

Instructions for setting register data

`set0 rX`	`00000100 00000000 00000000 0000XXXX`	sets `rX = 0`
`set1 rX`	`00000101 00000000 00000000 0000XXXX`	sets `rX = 1`
`setn rX N`	`00000110 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = N`, where $N \in [-2^{15}, 2^{15} - 1]$
`addn rX N`	`00000111 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = rX + N`, where $N \in [-2^{15}, 2^{15} - 1]$
`copy rX rY`	`00001000 00000000 00000000 XXXXYYYY`	sets `rX = rY`

Instructions for interacting with memory

`pushr rX rY`	`00011010 00000000 00000000 XXXXYYYY`	sets `mem[rY++] = rX`
`popr rX rY`	`00011011 00000000 00000000 XXXXYYYY`	sets `rX = mem[--rY]`
`loadn rX rY N`	`00011100 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `rX = mem[rY + N]`, where $N \in [-2^{15}, 2^{15} - 1]$
`storen rX rY N`	`00011101 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `mem[rY + N] = rX`, where $N \in [-2^{15}, 2^{15} - 1]$
`loadr rX rY`	`00011110 00000000 00000000 XXXXYYYY`	sets `rX = mem[rY]`
`storer rX rY`	`00011111 00000000 00000000 XXXXYYYY`	sets `mem[rY] = rX`

Marvin Programs · Combinations

Combinations.marv

```
# Accepts n (int) and k (int) from standard input and writes C(n, k) = n!/(k!(n-k)!) to standard output.

0     read     r0            # read n
1     read     r1            # read k
2     copy     r2 r0         # r2 = n
3     calln    r12 13        # n! = factorial(n)
4     copy     r4 r13        # r4 = n!
5     copy     r2 r1         # r2 = k
6     calln    r12 13        # k! = factorial(k)
7     div      r4 r4 r13     # r4 = n!/k!
8     sub      r2 r0 r1      # r2 = n - k
9     calln    r12 13        # (n-k)! = factorial(n-k)
10    div      r4 r4 r13     # r4 = n!/(k!(n-k)!)
11    write    r4            # write n!/(k!(n-k)!)
12    halt                   # halt the machine

# int factorial(int n):
#   input : r2 = n
#   output: r13 = n!
#   temps: r3

13    setn     r13 1         # output = 1
14    copy     r3 r2         # i = n
15    jeqzn    r3 19         # if i = 0 jump to 19
16    mul      r13 r13 r3    # output = output * i
17    addn     r3 -1         # i = i - 1
18    jumpn    15            # jump to 15
19    jumpr    r12           # jump to caller
```

Marvin Programs · Factorial

</> Factorial.marv 1/2

```
# Accepts n (int) from standard input and writes n! (computed recursively) to standard output.

0     read      r0           # read n
1     pushr     r0 r15       # mem[sp++] = n
2     calln     r12 6        # n! = factorial(n)
3     addn      r15 -1       # sp = sp - 1
4     write     r13          # write n!
5     halt                   # halt the machine

# int factorial(int n):
#   input : r0 = n
#   output: r13 = n!
#   temps: r1, r2

# Save ra and fp, and set fp to sp.
6     pushr     r12 r15      # mem[sp++] = ra
7     pushr     r14 r15      # mem[sp++] = fp
8     copy      r14 r15      # fp = sp

# Save registers used.
9     pushr     r0 r15       # mem[sp++] = r0
10    pushr     r1 r15       # mem[sp++] = r1
11    pushr     r2 r15       # mem[sp++] = r2

12    loadn     r0 r14 -3    # n = mem[fp - 3]
13    jnezn     r0 16        # if n != 0 jump to 16 (recursive step),
                             # else fall through (base case)

# Base case.
14    setn      r13 1        # output = 1
15    jumpn     22           # jump to 22

# Recursive step.
16    copy      r2 r0        # r2 = n
17    addn      r2 -1        # n = n - 1
```

Marvin Programs · Factorial

Factorial.marv 2/2

```
18    pushr     r2 r15         # mem[sp++] = n - 1
19    calln     r12 6          # (n-1)! = factorialRec(n-1)
20    addn      r15 -1         # sp = sp - 1
21    mul       r13 r0 r13     # n! = n(n-1)!

# Restore registers used.
22    popr      r2 r15         # r2 = mem[--r15]
23    popr      r1 r15         # r1 = mem[--r15]
24    popr      r0 r15         # r0 = mem[--r15]

# Restore fp and ra, and jump to ra (caller).
25    popr      r14 r15        # fp = mem[--r15]
26    popr      r12 r15        # ra = mem[--r15]
27    jumpr     r12            # jump to caller
```