ALF
language crunch

ALF

Diagramme de flux de contrôle et WebAssembly

Bibliographie pour aujourd'hui

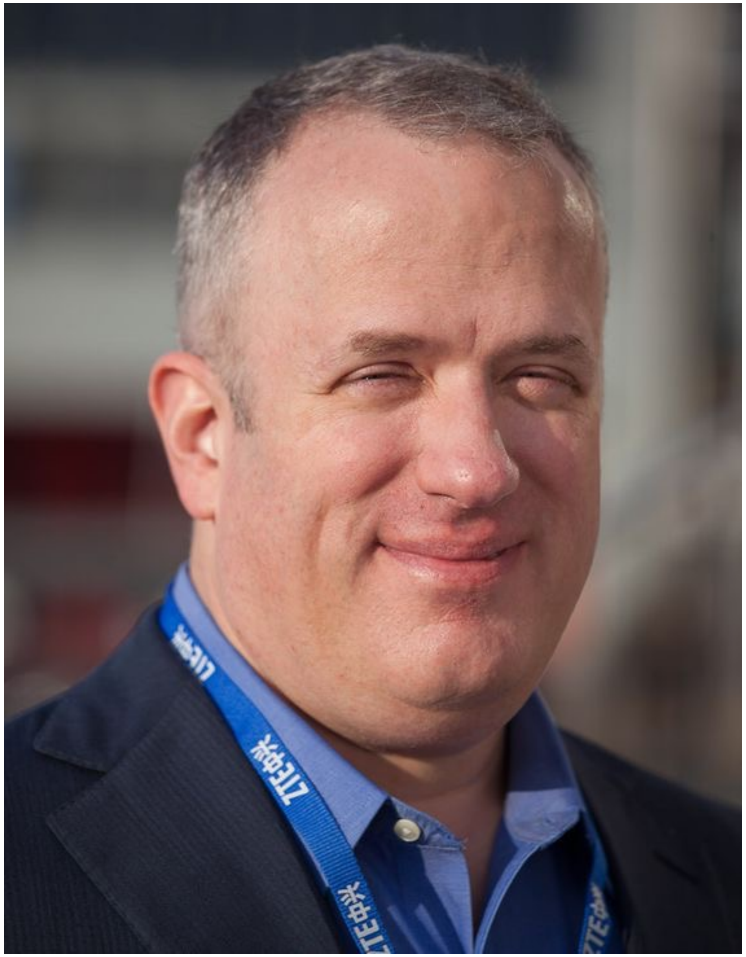
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*

- Chapitre 8
  - 8.1
  - 8.4

Contenu

- Diagramme de flux de contrôle
- Web Assembly

Brendan Eich



- Américain
- University of Illinois at Urbana-Champaign
- Co-fondateur de Netscape
- Auteur de JavaScript

Diagramme de flux de contrôle

- Organisation de Three Address Code a diagramme
- Start
- Stop
- Basic Block
  - le flux d'instructions n'est pas interrompu par un saut
  - il n'y a pas d'instruction d'étiquette (sauf la première instruction)
  - leader

Selection de leader

- premier instruction
- étiquette
- instruction après un saute (if, ifFalse, goto)

Exemple

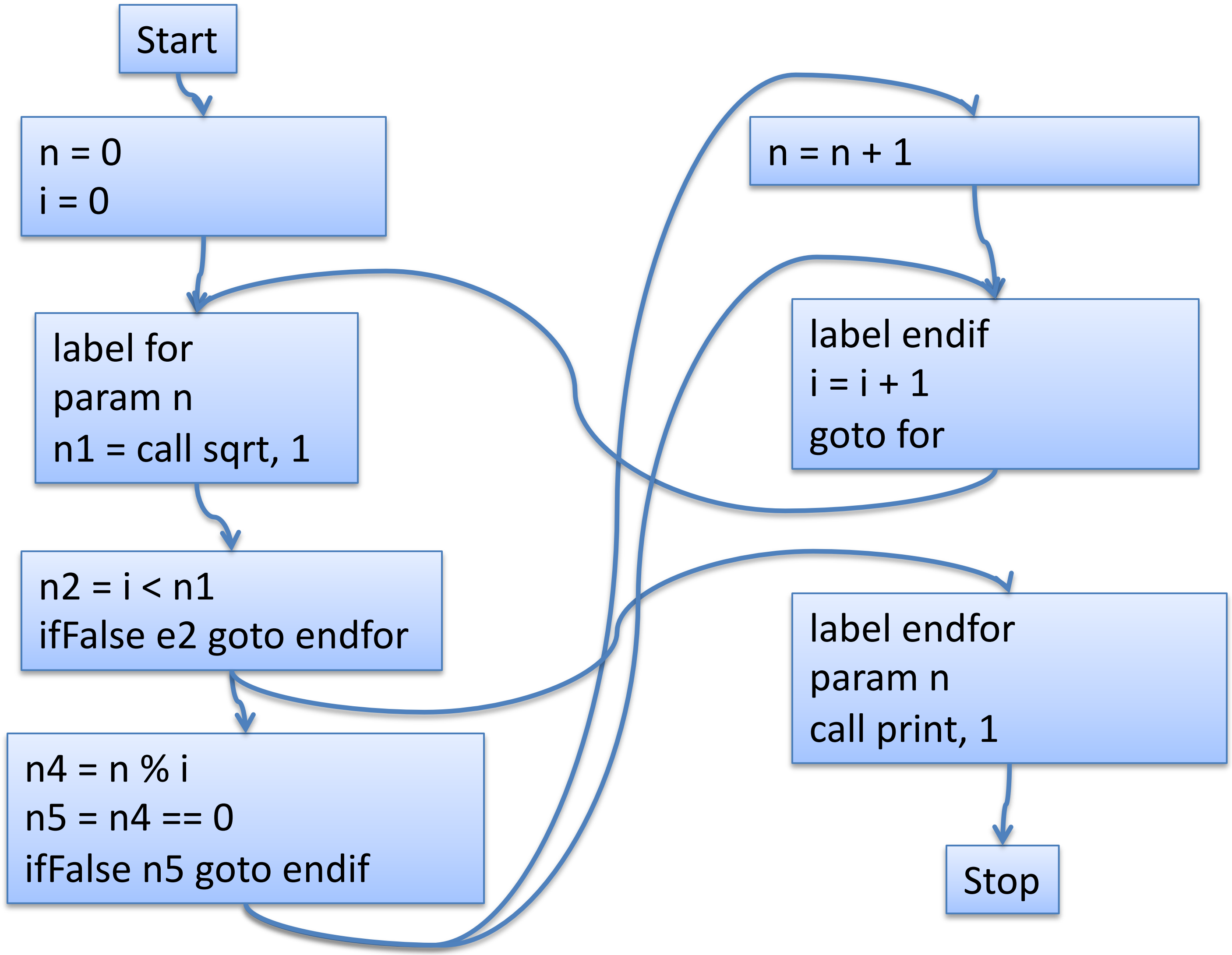
```
var n = 0;
for (var i = 0; i < sqrt(n); i++)
{
        if (n % i == 0) n = n + 1;
}
print (n);
```

Exemple

```
var n = 0;
for (var i = 0; i < sqrt(n); i++)
{
        if (n % i == 0) n = n + 1;
}
print (n);
```

```
n = 0
i = 0
label for
param n
n1 = call sqrt, 1
n2 = i < n1
ifFalse n2 goto endfor
n4 = n % i
n5 = n4 == 0
ifFalse n5 goto endif
n = n + 1
label endif
i = i + 1
goto for
label endfor
param n
call print, 1
```

Exemple



Start

n = 0
i = 0

label for
param n
n1 = call sqrt, 1

n2 = i < n1
ifFalse e2 goto endfor

n4 = n % i
n5 = n4 == 0
ifFalse n5 goto endif

n = n + 1

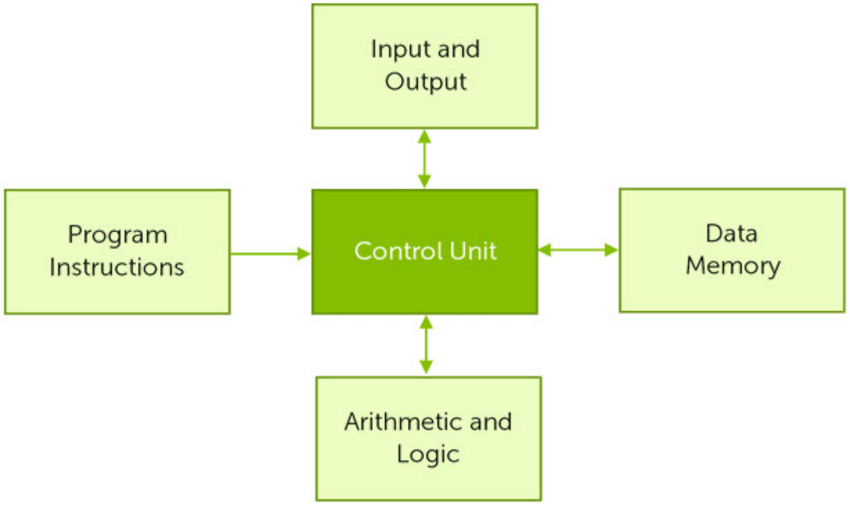
label endif
i = i + 1
goto for

label endfor
param n
call print, 1

Stop

WebAssembly

- Architecture Harvard
  - 32 bits
- Machine de pile infinie
- Mémoire program
- Mémoire données
- AST



S-expressions

```
(expression param1 param2 …)

(module …)

(module
        (import …)
   )
```

Instructions pour valeurs

Instruction	Equivalence
i32.const valeur	push valeur (32 bits int)
i64.const valeur	push valeur (64 bits int)
f32.const valeur	push valeur (32 bits float)
f64.const valeur	push valeur (64 bits float)

Instructions pour mémoire

Instruction	Equivalence
i32.load<dimension>_<sign>	push MEM [pop] (32 bits)
i64.load<dimension>_<sign>	push MEM [pop] (64 bits)
f32.load<dimension>_<sign>	push MEM [pop] (32 bits float)
f64.load<dimension>_<sign>	push MEM [pop] (64 bits float)
i32.store<dimension>_<sign>	MEM[pop] = pop (32 bits)
i64.store <dimension>_<sign>	MEM[pop] = pop (64 bits)
f32.store<dimension>_<sign>	MEM[pop] = pop (32 bits float)
f64.store<dimension>_<sign>	MEM[pop] = pop (62 bits float)

Instructions de saut

Instruction	Equivalence
br $etiquete	continue *ou* break etiquete
br_if $etiquete	if (pop) continue *ou* break etiquete
return valeur	push valeur return
loop $etiquete … end	continue
block $etiquete … end	break

Instructions arithmétique

Instruction	Equivalence
i32.add, i64.add, f32.add, f64.add	push pop + pop
i32.sub, i64.sub, f32.sub, f64.sub	push pop - pop
i32.mul, i64.mul, f32.mul, f64.mul	push pop * pop
i32.div_s, i64.div_s, f32.div_s, f64.div_s	push pop / pop (avec signe)
i32.div_u, i64.div_u, f32.div_u, f64.div_u	push pop / pop
i32.rem, i64.rem	push pop % pop
f32.sqrt, f64.sqrt	push sqrt (pop)
	push pop + pop

Instructions branche

Instruction	Equivalence

if (return type) then … else … end	if (){ … push valeur }
	else { … push valeur }

Instructions logique

Instruction	Equivalence
i32.and, i64.and …	push pop AND pop (bit par bit)
i32.or, i64.or …	push pop OR pop (bit par bit)
i32.xor, i64.xor …	push pop XOR pop (bit par bit)
i32.shl, i64.shl … (shift left)	push pop << pop (bit par bit)
i32.shr, i64.shr … (shift right)	push pop >> pop (bit par bit)
i32.gt, i64.gt …	push pop > pop (bit par bit)
i32.lt, i64.lt …	push pop < pop (bit par bit)

…

Assignement

r = x op y

r = op y

op: + - * / %

== <= >= < >

Assignement

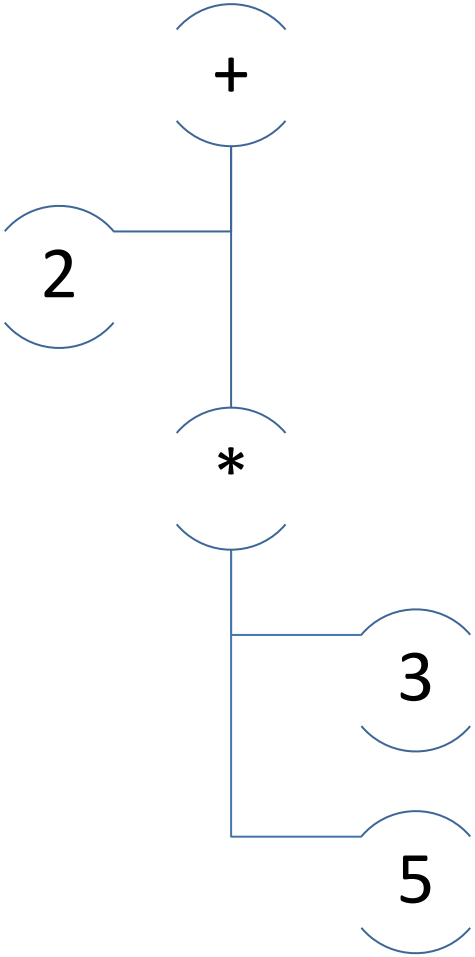
- r = x op y
  - prend x dans la pile
  - prend y dans la pile
  - op r x y
- r = x + y
  - i32.const &x
  - i32.load
  - i32.const &y
  - i32.load
  - i32.add

- r = N op y
  - pousser N dans la pile
  - prend y dans la pile
  - op r x y
- r = 60 + y
  - i64.const 60
  - i32.const &y
  - i64.load
  - i64.add

Exercices

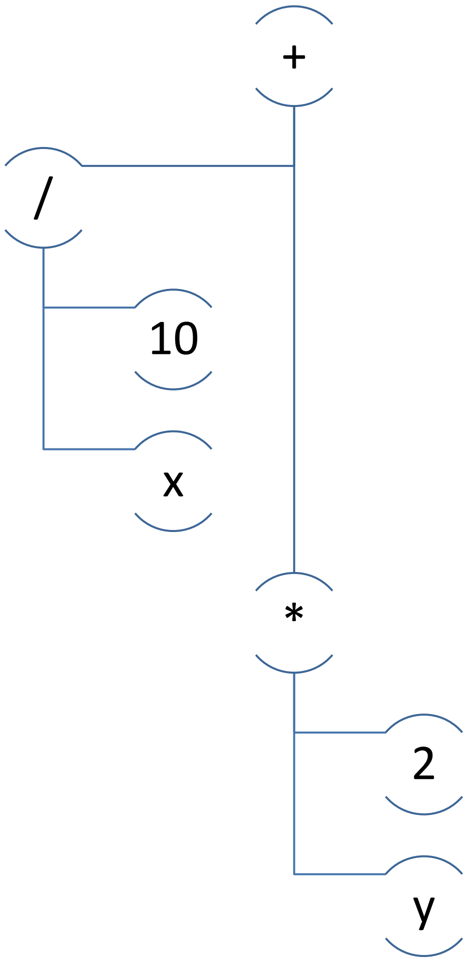
- 2+3*5
- (6-2)*4
- 10/x + 2*y
- 3- (-2) *6
- -10/2 – (2+4)/2*(7-(-1))

Exercices (2+3*5)



```
i32.const 2
i32.const 3
i32.const 5
i32.mul
i32.add
```

Exercices (10/x + 2*y)



```
i32.const 10
i32.const &x
i32.load
i32.div_u
i32.const 2
i32.const &y
i32.load
i32.mul
i32.add
```

Copie

$$x = y$$

Copie

- x = y
  - prend y dans la pile
  - stockez x de la pile

- x = y

```
i32.const &y
i64.load
i32.const &x
i64.store
```

-

Saut inconditionnel

- loop $name
- block $name
- br $name
- end $name

```
loop $next
br $next
end $next
x = 2 + 3 ; this is not reached

block $next
br $next
x = 2 + 3 ; this is jumped
end $next
```

Saut conditionnel

- if (result …)
- else
- end

```
i32.const &x
i32.load
if
;; x = 2 + 3 ; this is jumped if f is true
end
```

Exercises

```
if (x+y > 3)
{
        a = 11;
}
```

Exemple

```
if (x+y > 3)
{
        a = 11;
}
```

```
i32.const &x
i32.load
i32.const &y
i32.load
i32.add
i32.const 3
i32.gt_u
if
  i32.const 11
  i32.const &a
  i32.store
end
```

Exercises

```
if (x+y > 3)
{
        a = 11;
}
else
{
        a = 12;
}
```

Exemple

```
if (x+y > 3)
{
        a = 11;
}
else
{
        a = 12;
}
```

```
i32.const &x
i32.load
i32.const &y
i32.load
i32.add
i32.const 3
i32.gt_u
if
  i32.const 11
  i32.const &a
  i32.store
else
  i32.const 12
  i32.const &a
  i32.store
end
```

Exercises

```
if (x+y > 3 && y < x+90)
{
        a = 11;
}
else
{
        a = 12;
}
```

Exercises

```
while (x > 3)
{
        x = x + 1;
}
```

Exercises

```
while (x > 3)
{
        x = x + 1;
}
```

```
block $while_block
   loop $while_loop
      i32.const &x
      i32.load
      i32.const 3
      i32.le_u
      br_if $while_block
      i32.const &x
      i32.load
      i32.const 1
      i32.add
      i32.const &x
      i32.store
      br $while_loop
   end $while_loop
end $while_block
```

Exercises

```
do
{
        x = x + 1;
} while (x+y > 3 && y < x+90);
```

Exercises

```
for (x=1; x + y > 3; x = x + 1)
{
        y = y + 7;
}
```

Appel de fonction

- call $f

```
p = power (a, n);

i32.const &a
i64.load
i32.const &n
i64.load
call $power
i32.const &p
i64.store
```

Exercises

```
void print (int x, int y)
{
        printf (x);
}

print (2, 4);
```

Exercises

```
void print (int x, int y)
{
        printf (x);
}

print (2, 4);
```

```
(func $print
      (param $x i64)
      (param $y i64)
   get_local $x
   call $printf
)

i64.const 2
i64.const 4
call $print
```

Exercises

```
int expression (int x, int y, int z)
{
        return x*(y+z);
}

expression (1, 2, 5);
```

Exercises

```
int expression (int x, int y, int z)
{
        return x*(y+z);
}

expression (2+3, a+2*6, f(3));
```

Sujets

- Web Assembly
  - Mémoire
  - Instructions
- Three Address Code aWeb Assembly

WebAssembly

- WebAssembly Tutorial https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

- WebAssembly Instructions https://webassembly.org/docs/semantics/

Questions