Compiler Optimisation
7 – Register Allocation

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2018
This lecture:

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code
Register allocation

- Physical machines have limited number of registers
- Scheduling and selection typically assume infinite registers
- Register allocation and assignment $\infty \rightarrow k$ registers

**Requirements**

- Produce correct code that uses $k$ (or fewer) registers
- Minimise added loads and stores
- Minimise space used to hold spilled values
- Operate efficiently
  - $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$
Register allocation
Definitions

**Allocation versus assignment**

- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values

**Interference**

Two values\(^a\) cannot be mapped to the same register wherever they are both *live*\(^b\).

Such values are said to **interfere**

\(^a\)A value is stored in a variable
\(^b\)A value is live from its definition to its last use

**Live range**

The live range of a value is the set of statements at which it is live.

May be conservatively overestimated (e.g. just begin → end)
Register allocation

Definitions

Spilling
Spilling saves a value from a register to memory
That register is then free – Another value often loaded
Requires $F$ registers to be reserved

Clean and dirty values
A previously spilled value is **clean** if not changed since last spill
Otherwise it is dirty
A clean value can be spilled without a new store instruction

Spilling in ILOC
$F$ is 0 (assuming $r_{arp}$ already reserved)

**Dirty value**
storeAI $r_x \to r_{arp},@x$
loadAI $r_{arp},@y \Rightarrow r_y$

**Clean value**
loadAI $r_{arp},@y \Rightarrow r_y$
Local register allocation

Register allocation only on basic block

**MAXLIVE**

Let $\text{MAXLIVE}$ be the maximum, over each instruction $i$ in the block, of the number of values (pseudo-registers) live at $i$.

- If $\text{MAXLIVE} \leq k$, allocation should be easy
- If $\text{MAXLIVE} \leq k$, no need to reserve $\mathcal{F}$ registers for spilling
- If $\text{MAXLIVE} > k$, some values must be spilled to memory
- If $\text{MAXLIVE} > k$, need to reserve $\mathcal{F}$ registers for spilling

Two main forms:

- Top down
- Bottom up
Example MAXLIVE computation

Some simple code with virtual registers

```
loadI 1028  \Rightarrow r_a  // r_a \leftarrow 1028
load r_a  \Rightarrow r_b  // r_b \leftarrow \text{MEM}(r_a)
mult r_a, r_b \Rightarrow r_c  // r_c \leftarrow 1028 \cdot y
load x  \Rightarrow r_d  // r_d \leftarrow x
sub r_d, r_b \Rightarrow r_e  // r_e \leftarrow x - y
load z  \Rightarrow r_f  // r_f \leftarrow z
mult r_e, r_f \Rightarrow r_g  // r_g \leftarrow z \cdot (x - y)
sub r_g, r_c \Rightarrow r_h  // r_h \leftarrow z \cdot (x - y) - 1028 \cdot y
store r_h \Rightarrow r_a  // \text{MEM}(r_a) \leftarrow z \cdot (x - y) - 1028 \cdot y
```
Local register allocation
MAXLIVE

Example MAXLIVE computation

Live registers

```
loadI 1028  \implies r_a  // r_a
load  r_a  \implies r_b  // r_a  r_b
mult  r_a, r_b  \implies r_c  // r_a  r_b  r_c
load  x  \implies r_d  // r_a  r_b  r_c  r_d
sub  r_d, r_b  \implies r_e  // r_a  r_c  r_e
load  z  \implies r_f  // r_a  r_c  r_e  r_f
mult  r_e, r_f  \implies r_g  // r_a  r_c  r_e  r_f
sub  r_g, r_c  \implies r_h  // r_a  r_c  r_g  r_h
store  r_h  \implies r_a  //
```
Local register allocation

MAXLIVE

Example MAXLIVE computation

MAXLIVE is 4

\[
\begin{align*}
\text{loadI} & \ 1028 \ \Rightarrow r_a \\
\text{load} & \ r_a \ \Rightarrow r_b \\
\text{mult} & \ r_a, r_b \ \Rightarrow r_c \\
\text{load} & \ x \ \Rightarrow r_d \\
\text{sub} & \ r_d, r_b \ \Rightarrow r_e \\
\text{load} & \ z \ \Rightarrow r_f \\
\text{mult} & \ r_e, r_f \ \Rightarrow r_g \\
\text{sub} & \ r_g, r_c \ \Rightarrow r_h \\
\text{store} & \ r_h \ \Rightarrow r_a
\end{align*}
\]
Local register allocation
Top down

Algorithm:
- If number of values $> k$
  - Rank values by occurrences
  - Allocate first $k - F$ values to registers
  - Spill other values
Local register allocation

### Top down

#### Example top down

#### Usage counts

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td>→ ra</td>
<td>ra</td>
</tr>
<tr>
<td>load ra</td>
<td>→ rb</td>
<td>ra rb</td>
</tr>
<tr>
<td>mult ra, rb</td>
<td>→ rc</td>
<td>ra rb rc</td>
</tr>
<tr>
<td>load x</td>
<td>→ rd</td>
<td>ra rb rc rd</td>
</tr>
<tr>
<td>sub rd, rb</td>
<td>→ re</td>
<td>ra rc re</td>
</tr>
<tr>
<td>load z</td>
<td>→ rf</td>
<td>ra rc re rf</td>
</tr>
<tr>
<td>mult re, rf</td>
<td>→ rg</td>
<td>ra rc re rf rg</td>
</tr>
<tr>
<td>sub rg, rc</td>
<td>→ rh</td>
<td>ra rc rh</td>
</tr>
<tr>
<td>store rh</td>
<td>→ ra</td>
<td>ra</td>
</tr>
</tbody>
</table>

#### Counts

<table>
<thead>
<tr>
<th>Register</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>4</td>
</tr>
<tr>
<td>rb</td>
<td>3</td>
</tr>
<tr>
<td>rc</td>
<td>2</td>
</tr>
<tr>
<td>rd</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>rf</td>
<td>2</td>
</tr>
<tr>
<td>rg</td>
<td>2</td>
</tr>
<tr>
<td>rh</td>
<td>2</td>
</tr>
</tbody>
</table>
Local register allocation
Top down

Example top down

Spill $r_c$. Now only 3 values live at once

```
loadI 1028  ⇒ $r_a$
load  $r_a$  ⇒ $r_b$
mult  $r_a$, $r_b$  ⇒ $r_c$
load  $x$  ⇒ $r_d$
sub  $r_d$, $r_b$  ⇒ $r_e$
load  $z$  ⇒ $r_f$
mult  $r_e$, $r_f$  ⇒ $r_g$
sub  $r_g$, $r_c$  ⇒ $r_h$
store  $r_h$  ⇒ $r_a$
```

Must have $r_d$

Counts

- $r_a=4$
- $r_b=3$
- $r_c=2$
- $r_d=2$
- $r_e=2$
- $r_f=2$
- $r_g=2$
- $r_h=2$

Spill $r_c$

Restore $r_c$

//
Local register allocation
Top down

Example top down
Spill code inserted

```
loadI 1028          ra
load   rb
mult rb, rb       =>  rc
store rc            rarp, spillc
load   rd          rd
sub   rb, rb       re
load   rf          rf
mult rb, rb        rg
load   rarp, spillc rc
sub   rf, rc       rh
store rh            ra
```
Local register allocation

Top down

Example top down

Register assignment straightforward

```
loadI 1028  \rightarrow r_1
load r_1 \rightarrow r_2
mult r_1, r_2 \Rightarrow r_3
store r_3 \rightarrow r_{arp, spill_c}
load x \rightarrow r_3
sub r_3, r_2 \rightarrow r_2
load z \rightarrow r_3
mult r_2, r_3 \rightarrow r_2
load r_{arp, spill_c} \rightarrow r_3
sub r_2, r_3 \rightarrow r_2
store r_2 \rightarrow r_1
```
Local register allocation
Bottom up

Algorithm:
- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:
- Spill the value whose next use is farthest in the future
- Prefer clean value to dirty value
Local register allocation
Top down

Example bottom down

Spill $r_a$. Now only 3 values live at once

```
loadI 1028  $r_a
load     $r_a  $r_b
mult    $r_a, $r_b  $r_c
load    $x      $r_d
sub     $r_d, $r_b  $r_e
load    $z      $r_f
mult    $r_e, $r_f  $r_g
sub     $r_g, $r_c  $r_h
store   $r_h    $r_a

// $r_a
// $r_a $r_b
// $r_a $r_b $r_c
// $r_a $r_b $r_c $r_d
// $r_a $r_c $r_e
// $r_a $r_c $r_e $r_f
// $r_a $r_c $r_e
// $r_a $r_c $r_h

$r_a$ used
latest
Spill $r_a$

$r_h$

Restore $r_a$
```
Local register allocation

Top down

Example bottom down

Spill code inserted

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td></td>
<td>load r_a</td>
<td>r_a</td>
</tr>
<tr>
<td>load r_a</td>
<td>r_b</td>
<td>mult r_a, r_b</td>
<td>r_c</td>
</tr>
<tr>
<td>store r_a</td>
<td></td>
<td>=⇒</td>
<td>r_arp, spill_a</td>
</tr>
<tr>
<td>load x</td>
<td>r_d</td>
<td>sub r_d, r_b</td>
<td>r_e</td>
</tr>
<tr>
<td>load z</td>
<td>r_f</td>
<td>mult r_e, r_f</td>
<td>r_g</td>
</tr>
<tr>
<td>sub r_f, r_c</td>
<td>r_h</td>
<td>load r_arp, spill_a</td>
<td>r_a</td>
</tr>
<tr>
<td>store r_h</td>
<td></td>
<td>=⇒</td>
<td>r_a</td>
</tr>
</tbody>
</table>
Global register allocation

Local allocation does not capture reuse of values across multiple blocks.
Most modern, global allocators use a graph-colouring paradigm:
- Build a “conflict graph” or “interference graph”
  - Data flow based liveness analysis for interference
- Find a k-colouring for the graph, or change the code to a nearby problem that it can k-colour
- NP-complete under nearly all assumptions\(^1\)

\(^1\)Local allocation is NP-complete with dirty vs clean
Global register allocation
Algorithm sketch

- From live ranges construct an interference graph
- Colour interference graph so that no two neighbouring nodes have same colour
- If graph needs more than k colours - transform code
  - Coalesce merge-able copies
  - Split live ranges
  - Spill
- Colouring is NP-complete so we will need heuristics
- Map colours onto physical registers
Global register allocation
Graph colouring

**Definition**
A graph $G$ is said to be **k-colourable** iff the nodes can be labeled with integers $1 \ldots k$ so that no edge in $G$ connects two nodes with the same label.

**Examples**

- **2-colourable**

- **3-colourable**
Global register allocation
Interference graph

The interference graph, $G_I = (N_I, E_I)$

- Nodes in $G_I$ represent values, or live ranges
- Edges in $G_I$ represent individual interferences
- $\forall x, y \in N_I, x \rightarrow y \in E_I$ iff $x$ and $y$ interfere

A $k$-colouring of $G_I$ can be mapped into an allocation to $k$ registers

\[^2\]Two values interfere wherever they are both live
Two live ranges interfere if their values interfere at any point
Global register allocation
Colouring the interference graph

- Degree\(^3\) of a node \((n^\circ)\) is a loose upper bound on colourability
- Any node, \(n\), such that \(n^\circ < k\) is always trivially \(k\)-colourable
  - Trivially colourable nodes cannot adversely affect the colourability of neighbours\(^4\)
  - Can remove them from graph
  - Reduces degree of neighbours - may be trivially colourable
- If left with any nodes such that \(n^\circ \geq k\) spill one
  - Reduces degree of neighbours - may be trivially colourable

\(^3\)Degree is number of neighbours
\(^4\)Proof as exercise
Global register allocation
Chaitin’s algorithm

1. While \( \exists \) vertices with \(< k \) neighbours in \( G_I \)
   - Pick any vertex \( n \) such that \( n^\circ < k \) and put it on the stack
   - Remove \( n \) and all edges incident to it from \( G_I \)

2. If \( G_I \) is non-empty \( (n^\circ >= k, \forall n \in G_I) \) then:
   - Pick vertex \( n \) (heuristic), spill live range of \( n \)
   - Remove vertex \( n \) and edges from \( G_I \), put \( n \) on “spill list”
   - Goto step 1

3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again

4. Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Colour with $k = 3$ colours

Vertex $a$ is assigned colour $r_3$.
Vertex $b$ is assigned colour $r_1$.
Vertex $c$, $d$, and $e$ are assigned colour $r_2$. 

Graph $G_I$
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

\[ a^\circ = 2 < k \quad \text{Choose } a \]

\[ G_I \]

\[ \begin{array}{c}
\text{Stack} \\
\end{array} \]

\[ \begin{array}{c}
\text{Colours} \\
r_1 \quad r_2 \quad r_3 \\
\end{array} \]
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Push a and remove from graph

\[ G_I \]

\[ \text{Stack} \]

\[ \text{Colours} \]

\[ r_1 \]

\[ r_2 \]

\[ r_3 \]
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

\[ b^\circ = 2 < k \text{ and } c^\circ = 2 < k \quad \text{Choose } b \]

\[ G_I \]

Stack

Colours

\[ r_1 \]
\[ r_2 \]
\[ r_3 \]
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Push b and remove from graph

- e
- d
- c
- G_I

- b
- a
- Stack

- r_1
- r_2
- r_3
- Colours
Example: colouring with Chaitin’s algorithm

\[ c^\circ = 2 < k, \quad d^\circ = 2 < k, \quad \text{and} \quad e^\circ = 2 < k \]

Choose \( c \)
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Push c and remove from graph

\[G_I\]

\[\text{Stack}\]

\[\text{Colours}\]

1. \(r_1\)
2. \(r_2\)
3. \(r_3\)
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

\[ d^\circ = 1 < k \text{ and } e^\circ = 1 < k \] Choose \( d \)

\[ G_1 \]

\[ \text{Stack} \]

\[ \text{Colours} \]

\[ r_1, r_2, r_3 \]
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Push $d$ and remove from graph

$G_I$

Stack

Colours

$r_1$

$r_2$

$r_3$
Global register allocation

Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

\[ e^{\circ} = 0 < k \]

Choose \( e \)

\[ G_I \quad \text{Stack} \quad \text{Colours} \]

\( r_1 \quad r_2 \quad r_3 \)
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Push e and remove from graph

G_i
Stack
Colours

r_1
r_2
r_3
Example: colouring with Chaitin’s algorithm

Pop e, neighbours use no colours, choose red
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Pop $d$, neighbours use red, choose green
Example: colouring with Chaitin’s algorithm

Pop c, neighbours use red and green choose blue
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Pop $b$, neighbours use red and green choose blue
Global register allocation
Chaitin’s algorithm

Example: colouring with Chaitin’s algorithm

Pop a, neighbours use blue choose red
Global register allocation
Optimistic colouring

- If Chaitin's algorithm reaches a state where every node has \( k \) or more neighbours, it chooses a node to spill.

Example of Chaitin overzealous spilling

\[ k = 2 \]

Graph is 2-colourable
Chaitin must immediately spill one of these nodes

- Briggs said, take that same node and push it on the stack
  - When you pop it off, a colour might be available for it!
- Chaitin-Briggs algorithm uses this to colour that graph
While $\exists$ vertices with $< k$ neighbours in $G_I$
  - Pick any vertex $n$ such that $n^\circ < k$ and put it on the stack
  - Remove $n$ and all edges incident to it from $G_I$

If $G_I$ is non-empty ($n^\circ \geq k, \forall n \in G_I$) then:
  - Pick vertex $n$ (heuristic) (Do not spill)
  - Remove vertex $n$ from $G_I$, put $n$ on stack (Not spill list)
  - Goto step 1

Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
  - If some vertex cannot be coloured, then pick an uncoloured vertex to spill, spill it, and restart at step 1

Step 3 is also different
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Colour with $k = 2$ colours

$G_I$

$\text{Stack}$

$\text{Colours}$

- $r_1$
- $r_2$
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

\[ a^\circ = 2 \geq k \quad \text{Don’t Spill! Choose } a \]
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Push a and remove from graph

\[ G_I \]

\[ \text{Stack} \]

\[ \text{Colours} \]

- \( b \)
- \( d \)
- \( c \)
- \( r_1 \)
- \( r_2 \)
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

$b^\circ = 1 < k$ and $c^\circ = 1 < k$ Choose $b$
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Push $b$ and remove from graph

$G_I$ $\rightarrow$ $\{c, d\}$

Stack: $\{a, b\}$

Colours: $\{r_1, r_2\}$
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

\[ c^\circ = 1 < k, \text{ and } d^\circ = 1 < k \]

Choose \( c \)
Example: colouring with Chaitin-Briggs algorithm

Push c and remove from graph

$G_I$  Stack  Colours

d  c  b  a  $r_1$

$G_I$  Stack  Colours

d  c  b  a  $r_2$
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

\[ d^\circ = 1 < k \quad \text{Choose } d \]

\[ G_I \quad \text{Stack} \quad \text{Colours} \]

- \( d \)
- \( c \quad b \quad a \)
- \( r_1 \quad r_2 \)
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Push $d$ and remove from graph

$G_I$ Stack Colours

$\text{d}$ $\text{c}$ $\text{b}$ $\text{a}$

$r_1$ $r_2$
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Pop $d$, neighbours use no colours, choose red
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Pop c, neighbours use red choose green

(G_I, c, d, a, b, r_1, r_2)
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Pop b, neighbours use red choose green
Global register allocation
Chaitin-Briggs algorithm

Example: colouring with Chaitin-Briggs algorithm

Pop a, neighbours use green choose red

G₁

Stack

Colours
Global register allocation
Spill candidates

- Minimise spill cost/ degree
- Spill cost is the loads and stores needed. Weighted by scope - i.e. avoid inner loops
- The higher the degree of a node to spill the greater the chance that it will help colouring
- Negative spill cost load and store to same memory location with no other uses
- Infinite cost - definition immediately followed by use. Spilling does not decrease live range
Global register allocation
Alternative spilling

- Splitting live ranges
- Coalesce
Global register allocation

Live range splitting

- A whole live range may have many interferences, but perhaps not all at the same time
- Split live range into two variables connected by copy
- Can reduce degree of interference graph
- Smart splitting allows spilling to occur in “cheap” regions
Global register allocation

Live ranges splitting

Splitting example

Non contiguous live ranges - cannot be 2 coloured

Live ranges

Interference Graph
Global register allocation
Live ranges splitting

Splitting example

Split live ranges - can be 2 coloured

Live ranges

Interference Graph
Global register allocation

Coalescing

If two ranges don't interfere and are connected by a copy coalesce into one – opposite of splitting
Reduces degree of nodes that interfered with both

\[ x := y \text{ and } x \rightarrow y \in G_I \text{ then can combine } LR_x \text{ and } LR_y \]
-Eliminates the copy operation
-Reduces degree of LRs that interfere with both \( x \) and \( y \)
-If a node interfered with both both before, coalescing helps
-As it reduces degree, often applied before colouring takes place
Coalescing can make the graph harder to color

- Typically, \( LR_{xy} > \max(LR_x, LR_y) \)

- If \( \max(LRx, LRy) < k \) and \( k < LR_{xy} \) then \( LR_{xy} \) might spill, while \( LR_x \) and \( LR_y \) would not spill
Global register allocation
Coalescing

*Observation led to conservative coalescing*

- Conceptually, coalesce $x$ and $y$ iff $x \rightarrow y \in G_T$ and $LR_{xy} \circ < k$
- We can do better
  - Coalesce $LR_x$ and $LR_y$ iff $LR_{xy}$ has $< k$ neighbours with degree $> k$
  - Only neighbours of “significant degree” can force $LR_{xy}$ to spill
- Always safe to perform that coalesce
  - Cannot introduce a node of non-trivial degree
  - Cannot introduce a new spill
Global register allocation

Other approaches

- Top-down uses high level priorities to decide on colouring
- Hierarchical approaches - use control flow structure to guide allocation
- Exhaustive allocation - go through combinatorial options - very expensive but occasional improvement
- Re-materialisation - if easy to recreate a value do so rather than spill
- Passive splitting using a containment graph to make spills effective
- Linear scan - fast but weak; useful for JITs
Global register allocation
Ongoing work

- Eisenbeis et al examining optimality of combined reg alloc and scheduling. Difficulty with general control-flow

- Partitioned register sets complicate matters. Allocation can require insertion of code which in turn affects allocation. Leupers investigated use of genetic algs for TM series partitioned reg sets.

- New work by Fabrice Rastello and others. Chordal graphs reduce complexity

- As latency increases see work in combined code generation, instruction scheduling and register allocation
Summary

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code
The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk

• 4-year programme: MSc by Research + PhD

• Research-focused: Work on your thesis topic from the start

• Collaboration between:
  ► University of Edinburgh’s School of Informatics
    ✴ Ranked top in the UK by 2014 REF
  ► Edinburgh Parallel Computing Centre
    ✴ UK’s largest supercomputing centre

• Research topics in software, hardware, theory and application of:
  ► Parallelism
  ► Concurrency
  ► Distribution

• Full funding available

• Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years