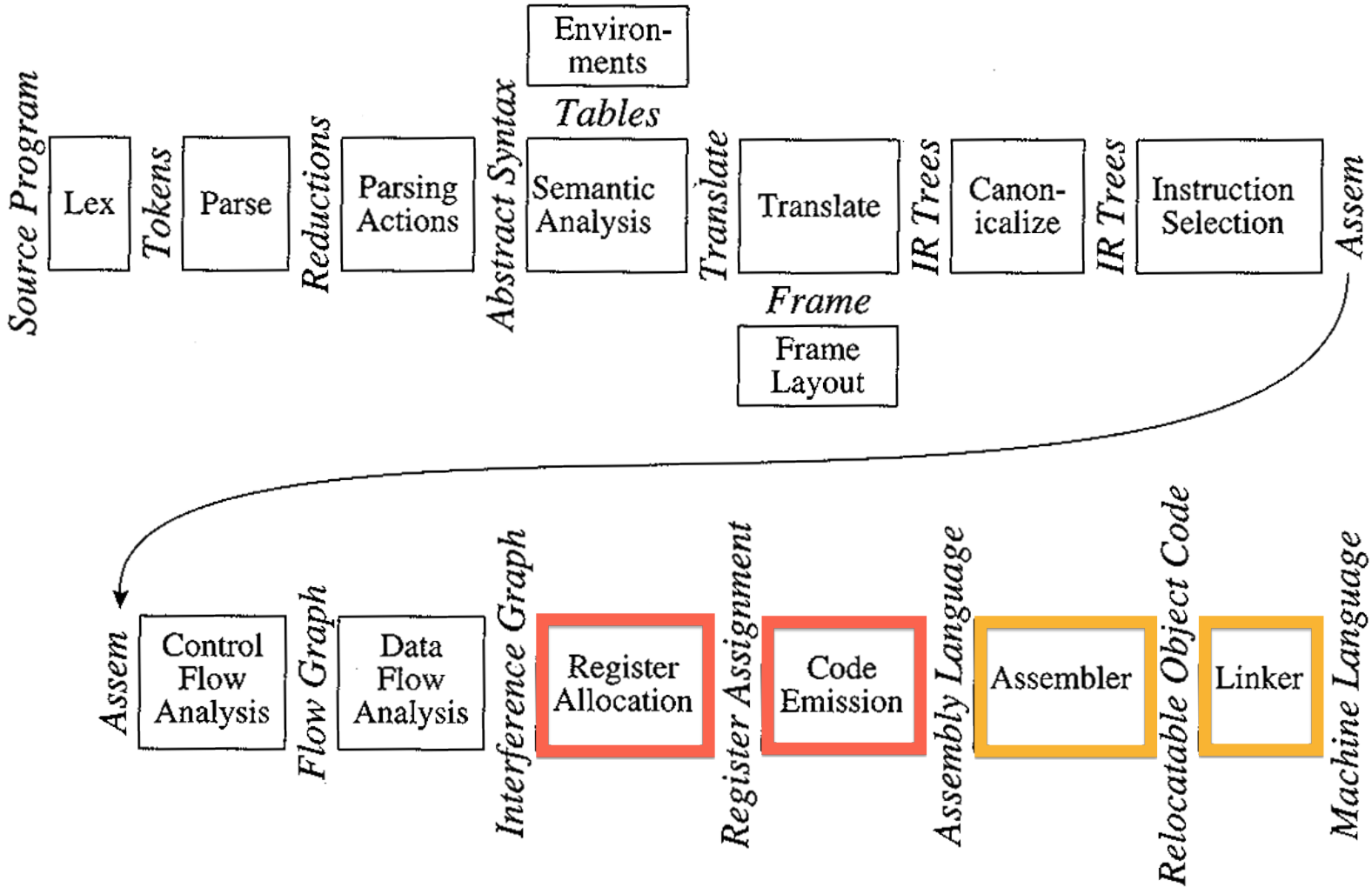
Registerverteilung

Registerverteilung



Registerverteilung

Ziel: Verteilung der Temporaries auf physikalische Register, bzw. Adressen im Frame (*spilling*).

Dabei:

- Minimierung der Adressen im Frame.
- Zusammenfassung von Temporaries, wenn möglich.
- Elimination von Verschiebeanweisungen.

Registerzuweisung als Graphenfärbung

Lassen wir einmal Spilling außer acht, nehmen wir also an, wir könnten alle Temporaries auf Register abbilden.

Eine legale Registerzuweisung weist jedem Knoten des Interferenzgraphen ein Register zu, so dass durch eine Kante verbundene Knoten verschiedene Register zugewiesen bekommen.

Es geht also darum, den Interferenzgraphen mit den Registern zu *färben*. Die Register sind also die Farben.

Graphenfärbung ist i.a. NP-vollständig (siehe Info IV). Für unsere Zwecke genügt aber eine einfache Heuristik von Kempe (1879):

Heuristik für Graphenfärbung

Um einen Graphen G mit K Farben zu färben gehe wie folgt vor:

- Ist G leer, sind wir fertig.
- Hat G einen Knoten v vom Grad $< K$, also weniger als K Nachbarn, so entferne v und färbe $G \setminus \{v\}$ rekursiv. Gelingt das, so kann auch ganz G gefärbt werden, da ja noch mindestens eine Farbe für v frei ist.
- Hat G nur noch Knoten vom Grad $\geq K$, so antworte mit „nicht färbbar“.

Färbung mit Spilling

Kann der Interferenzgraph nicht mit K Farben (K die Anzahl der Register) gefärbt werden, so müssen einige Temporaries in den Frame ausgelagert werden (*spilling*). Dazu verfeinert man obige Heuristik wie folgt:

- Ist G leer, so sind wir fertig.
- Hat G einen Knoten v vom Grad $< K$, so entferne v und färbe $G \setminus \{v\}$ rekursiv. Färbe dann v mit einer der verbleibenden Farben.
- Hat G nur Knoten vom Grad $\geq K$, so entferne Knoten v mit maximalem Grad und färbe den verbleibenden Graphen rekursiv. Durch das Entfernen des Spillkandidaten können nun wieder Knoten mit Grad $< K$ entstanden sein. Wurden die Nachbarn von v mit weniger als K Farben gefärbt, so kann man v doch noch färben. Andernfalls bleibt v ungefärbt und wird als „Spill-Knoten“ markiert.

Vorgefärbte Knoten

Temporaries, die den Maschinenregistern entsprechen, entsprechen vorab gefärbten Knoten im Interferenzgraphen. Sie werden nicht in den Frame ausgelagert, d.h. sind nie Spillkandidaten.

Dies wird erreicht, indem der Algorithmus nur ungefärbte Knoten aus dem Graphen entfernt, und die Rekursion endet, wenn nur noch vorgefärbte Knoten vorhanden sind.

Registerallokator-Algorithmus

Build Baue den Interferenzgraphen, mit vorgefärbten Knoten.

Simplify Entferne alle ungefärbten Knoten mit Grad $< K$ und lege sie auf einen Stack.

Spill Wenn es einen solchen Knoten nicht mehr gibt, entferne den ungefärbten Knoten mit maximalem Grad und lege ihn auf den Stack. Probiere wieder *Simplify*.

Select Wenn der Graph nur noch vorgefärbte Knoten enthält: Füge jeweils einen Knoten vom Stack dem Graph wieder hinzu, und färbe ihn mit einer Farbe, die keiner der schon vorhandenen Nachbarn hat. Ist dies nicht möglich, handelt es sich um einen Spill-Knoten.

Start over Falls Spill-Knoten vorhanden sind: schreibe das Programm um und fange wieder bei *Build* an.

Umschreiben des Programms

Verbleiben nach Ablauf des Algorithmus noch Spillkandidaten, so werden sie in den Frame ausgelagert; hierzu muss das Maschinenprogramm entsprechend umgeschrieben werden. Für jede Spillvariable t ist eine lokale Variable m im Frame anzulegen (mit Speicherort `IN_MEMORY`).

- Allen Instruktionen, die t benutzen, wird $t \leftarrow m$ vorangestellt.
- Allen Instruktionen, die t definieren, wird $m \leftarrow t$ nachgestellt.

Außerdem wird jeweils lokal in der Instruktion und der eingefügten Verschiebeoperation die Variable t in ein frisches Temporary t' umbenannt.

Beispiel:

```
add t4, 1
```

$\longrightarrow$

```
mov t132, [ebp+12]
add t132, 1
mov [ebp+12], t132
```

Wiederholung des Algorithmus'

Die gesamte Prozedur einschließlich Aktivitätsanalyse ist dann erneut durchzuführen, bis keine Spillkandidaten mehr auftreten.

Da der Registerallokator in den folgenden Runden nur noch ein Register für die (nur kurzzeitig verwendeten) Variablen t' finden muss, endet der Algorithmus in der Regel nach 1-3 Runden.

Nach der Registerverteilung kann der Code redundante Verschiebeinstruktionen der Form $t \leftarrow t$ enthalten. Diese können natürlich entfernt werden.

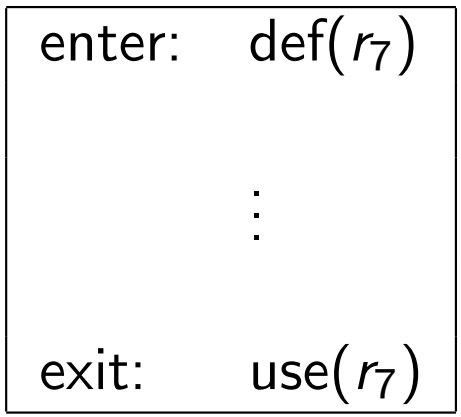
Caller-save-Register

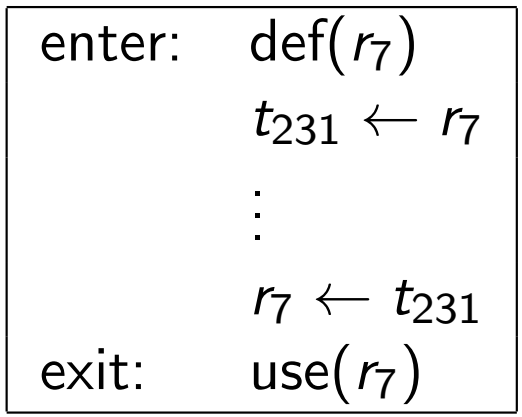
Caller-save-Register sind Register, die eventuell in einem Funktionsaufruf überschrieben werden. Daher müssen sie in der Def-Liste der `CALL`-Instruktion stehen. Der Registerallokator wird dadurch keine Variablen auf caller-save-Register abbilden, deren Inhalt nach einem Aufruf noch benötigt wird.

Callee-save-Register

Callee-save-Register müssen beim Verlassen einer Funktion die gleichen Werte enthalten wie beim Betreten der Funktion. Dies wird dadurch erreicht, dass alle callee-save-Register in der Use-Liste der `RET`-Instruktion stehen.

Dadurch sind diese Register aber während der gesamten Laufzeit der Funktion *live*, können also nicht vom Registerallokator verwendet werden. Es empfiehlt sich daher, deren Aktivitätsbereich durch sofortiges Verlagern in andere Temporaries beim Betreten einer Funktion zu minimieren, und am Ende wieder herzustellen.





Optimierung: Framegröße

Auch Spills sind oft nicht gleichzeitig aktiv, so dass sich mehrere Spillvariablen die gleiche Position im Frame teilen können.

Zur Optimierung des Spill-Speichers werden nur die Spill-Knoten betrachtet. Es wird ein vereinfachter Färbealgorithmus für den Interferenz-Teilgraphen benutzt, wobei jede Farbe einer Position im Frame entspricht.

Simplify Entferne jeweils den Knoten mit dem kleinsten Grad und lege ihn auf den Stack.

Select Füge den Knoten wieder hinzu und wähle die erste Farbe (niedrigste Frameposition), die nicht schon von einem vorhandenen Nachbarn benutzt wird.

Es gibt so viele Farben wie Spill-Knoten, denn der Spill-Knoten-Teilgraph soll ja färbbar sein (und nicht wieder neue Spills produzieren).

Optimierung: Verschmelzung (Coalescing)

Unter Umständen ist es sinnvoll, das Auftreten überflüssiger Zuweisungen dadurch zu begünstigen, dass man schon vor der Registerverteilung nicht-interferierende Knoten a und b verschmilzt, sofern zumindest eine Verschiebeinstruktion $a \leftarrow b$ vorhanden ist.

Dadurch erhöht sich aber i.a. der Knotengrad und zusätzliche Spills könnten resultieren, in diesem Fall muss das Verschmelzen unbedingt vermieden werden.

Dieser ungünstige Fall tritt sicher dann *nicht* auf, wenn eine der folgenden Bedingungen zutrifft:

Heuristiken für die Verschmelzung

Briggs: Knoten a und b können verschmolzen werden, wenn der resultierende Knoten ab weniger als K Nachbarn vom Grad $\geq K$ hat.

George: Knoten a und b können verschmolzen werden, wenn jeder Nachbar t von a entweder mit b interferiert, oder aber Grad $< K$ hat.

Man überlegt sich, dass das Verschmelzen in diesen Fällen keine Verschlechterung der Färbbarkeit nach sich zieht.

Es empfiehlt sich, während des Verlaufs der Registerverteilungsprozedur immer wieder zu prüfen, ob Verschmelzung nach diesen Kriterien möglich ist. Schließlich könnte ja durch das sukzessive Entfernen von Knoten eine der Bedingungen erfüllt werden.

Registerallokation mit Verschmelzung

Build	Erstelle den Interferenzgraphen und markiere Knoten, die Teil einer Verschiebeoperation sind, als „verschmelzbar“.
Simplify	Entferne nicht „verschmelzbare“ Knoten von Grad $< K$ und lege sie auf den Stack.
Coalesce	Ist kein solcher Knoten vorhanden, so verschmelze Knoten gemäß der Briggs- oder George-Heuristik. Weiter mit *Simplify*.
Freeze	Applizieren weder *Simplify* noch *Coalesce*, so deklariere einen „verschmelzbaren“ Knoten von Grad $< K$ als „nicht verschmelzbar“. Weiter mit *Simplify*.
Spill	Gibt es überhaupt keinen Knoten von Grad $< K$, so entferne einen beliebigen Knoten, lege ihn auf den Stack und probiere wieder *Simplify*.
Select	(wie vorher)
Start over	(wie vorher)

Verschmelzen von Spill-Kandidaten

Auch die Spill-Knoten können bei Vorhandensein einer entsprechenden Verschiebeinstruktion verschmolzen werden.

Hierbei werden sie aggressiv verschmolzen, bevor der oben beschriebene Färbealgorithmus zur Minimierung der benötigten Frameposition ausgeführt wird.

Optimierung: Direkter Framezugriff

Der Befehlssatz der Zielarchitektur kann Instruktionen enthalten, die direkt auf die Frame-Variablen zugreifen können. In diesem Fall sollten solche Spill-Kandidaten ausgewählt werden, die besonders oft in solchen hauptspeicheradressierende Instruktionen auftreten, da hierbei das Einfügen von Verschiebeoperationen entfällt.

Ausgabe des Assemblercodes

Nach der Registerverteilung entsprechen die einzelnen Fragmente dem ausführbaren Assemblercode. Die `Assem`-Objekte müssen also nur noch in Strings umgewandelt und in eine Datei ausgegeben werden.

Bei x86-Assemblercode unterscheidet man zwischen der *Intel*- und der *AT&T*-Syntax. Wir betrachten hier nur die Intel-Syntax, weil der x86-Simulator von der Homepage nur diese beherrscht (und sie etwas lesbarer ist).

Eine Assemblerdatei hat folgenden Inhalt:

```
.intel_syntax
.global Lmain
```

und dann für jedes Fragment folgenden Code:

Ein- und Austrittscode

```
L<Fragmentname>:
    push %ebp
    mov %ebp, esp
    sub %esp, <framesize>
    <... Assem-Liste in Intel-Syntax ...>
    mov %esp, ebp
    pop %ebp
    ret
```

Beachte:

- Der Eintrittscode kann erst nach der Registerverteilung generiert werden, weil erst dann die endgültige Framegröße bekannt ist.
- Der Austrittscode muss vor der Registerverteilung generiert werden, weil die callee-save-Register in der Use-Liste von `RET` stehen müssen.

Ausführen des Assemblercodes

Testen mit dem x86-Simulator:

```
./risc386 output.s
```

(Erinnerung: Der Simulator kann auch Assemblercode ausführen, der noch Temporaries enthält.)

Ausführen des Codes:
Wir verwenden den GCC, um abschließend die Laufzeitbibliothek mit dem Code zu linken:

```
gcc output.s runtime.c
./a.out
```

Das übersetzte Programm sollte dieselbe Ausgabe wie die Ausführung des ursprünglichen MiniJava-Programms mit dem Java-Interpreter haben.

Ihre heutige Aufgabe

- Implementieren Sie einen Registerallokator für MiniJava.
- Achten Sie auf eine korrekte Verwendung von caller-save- und callee-save-Register durch den Registerallokator.
- Optional: Implementieren Sie die beschriebenen Optimierungen.
- Geben Sie den Assemblercode aus, und führen Sie ihn aus.