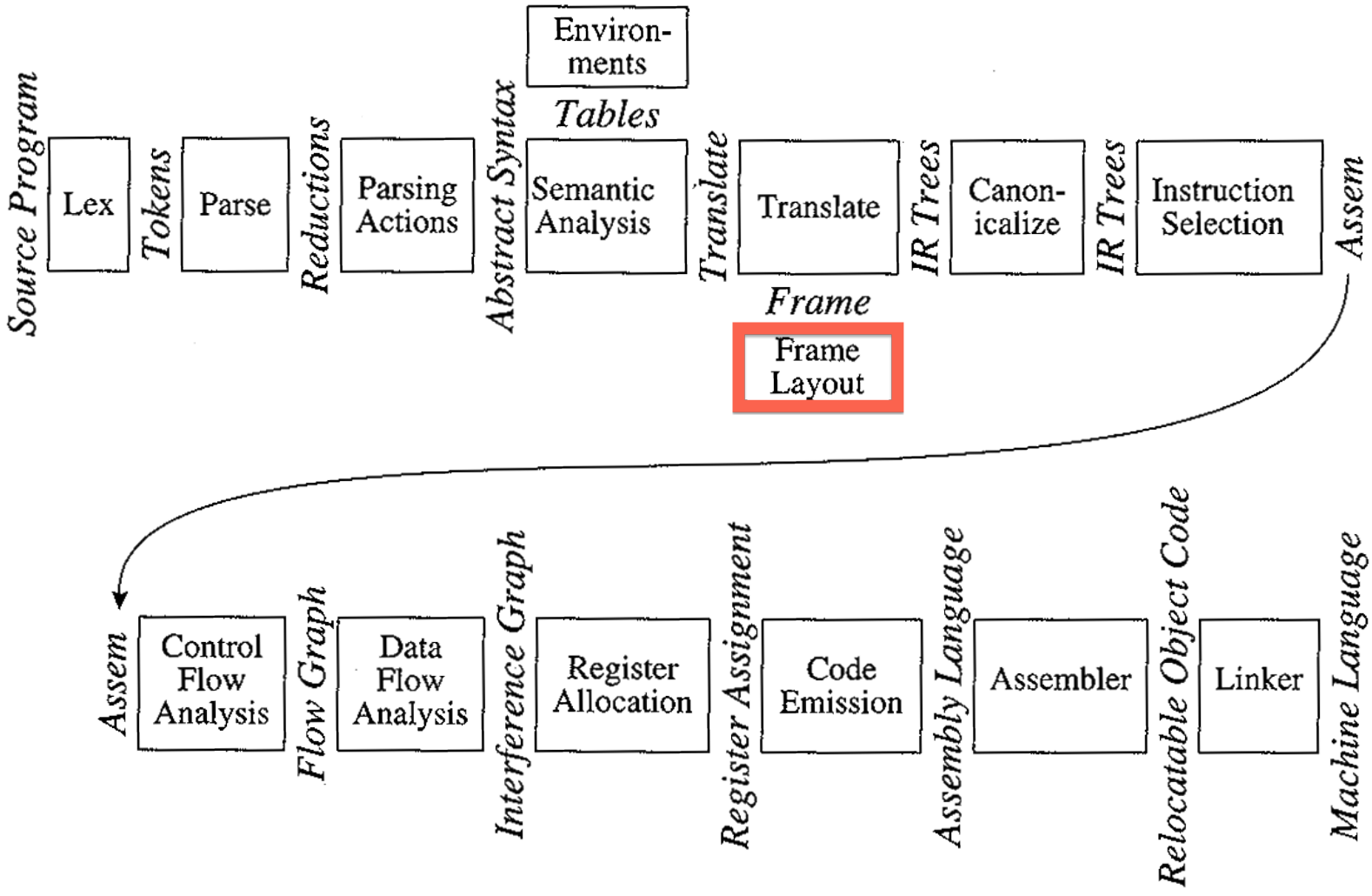
Aktivierungssätze (Frames)

Aktivierungssätze



Aktivierungssätze

- Aktivierungssätze (auch **Activation Records** oder **Frames** genannt) dienen zur Verwaltung von lokalen Informationen der aktuellen Funktionsumgebung.
- In MiniJava müssen Methodenparameter und lokale Variablen in Aktivierungssätzen abgelegt werden.
- Die Aufrufhistorie (call stack) wird in block-strukturierten Programmiersprachen mittels eines *Stacks* von Frames realisiert.
- Das genaue Layout eines Aktivierungssatzes ist hardwareabhängig.

Lokale Variablen

```
public int f (int x, int y) {
  int z;
  z=x+y;
  g(z);
  if (y<1) { return x; }
  else     { return f(z,y-1); }
}
```

Wenn `f` aufgerufen wird, so wird ein neuer Aktivierungssatz mit den Instanzen von `x`, `y` und `z` geschaffen. Die formalen Parameter werden vom Aufrufer (*caller*) initialisiert.

Inhalt eines Aktivierungssatzes

Ein **Frame** enthält die lokale Umgebung der aktuellen Funktion, d.h.:

- Argumente der Methode
- Lokale Variablen
- Administrative Informationen, z.B. Rücksprungadresse
- Temporäre Variablen und zwischengespeicherte Register

Der Prozessorhersteller gibt oft eine bestimmte Aufrufkonvention vor (Parameterlayout etc.), damit Funktionen sprachübergreifend verwendet werden können.

Stack

- Da lokale Variablen in MiniJava beim Verlassen einer Funktion gelöscht werden, können Aktivierungssätze auf einem Stack verwaltet werden.
- Der Stack ist ein Speicherbereich, dessen Ende durch den **Stack Pointer** (spezielle Variable) bezeichnet wird. Aus historischen Gründen wächst der Stack nach unten (hin zu niedrigeren Adressen). Genauer bezeichnet der Stack Pointer das letzte Wort am Ende des Stacks.
- Bei Aufruf einer Funktion wird der Stack um einen Aktivierungssatz erweitert (durch Erniedrigen des Stack Pointers). Beim Verlassen der Funktion wird der Stack Pointer entsprechend zurück (nach oben!) gesetzt, sodass der Frame entfernt ist.

Der Frame-Pointer

Zur Verwaltung des Stacks als eine Sequenz von Aktivierungssätzen werden zwei spezielle Register verwendet:

- **frame pointer** (FP): zeigt auf den Beginn des aktuellen Frames
- **stack pointer** (SP): zeigt auf das Ende des aktuellen Frames
- Ruft $f(x, y)$ die Funktion $g(z)$ auf, so wird der Inhalt von FP auf dem Stack abgelegt (als sog. *dynamic link*) und SP nach FP übertragen.
- Beim Verlassen von g wird FP wieder nach SP geschrieben und FP aus dem Stack restauriert.
- Ist die Framegröße jeder Funktion zur Compilezeit bekannt, so kann FP jeweils ausgerechnet werden als $SP + \texttt{framesize}$.

Register

Register sind in den Prozessor eingebaute extrem schnelle Speicherelemente. RISC Architekturen (Sparc, PowerPC, MIPS) haben oft 32 Register à ein Wort; der Pentium hat acht Register.

- Lokale Variablen, Parameter, etc. sollen soweit wie möglich in Registern gehalten werden.
- Oft können arithmetische Operationen nur in Registern durchgeführt werden.

Aufrufkonventionen

- Der Hersteller gibt vor, welche Register **caller-save**, d.h., von der aufgerufenen Funktion beschrieben werden dürfen; und welche **callee-save** sind, also von der aufgerufenen Funktion im ursprünglichen Zustand wiederherzustellen sind.
- Befindet sich eine lokale Variable o.ä. in einem caller-save Register, so muss sie der Aufrufer vor einem Funktionsaufruf im Frame abspeichern, es sei denn, der Wert wird nach Rückkehr der Funktion nicht mehr gebraucht.
- Der Pentium hat drei caller-save Register und vier callee-save Register.

Parameterübergabe

Je nach Architektur werden die ersten k Parameter in Registern übergeben; die verbleibenden über den Frame. Beim Pentium werden i.d.R. alle Parameter über den Frame übergeben, allerdings kann man bei GCC auch Werte $0 < k \leq 3$ einstellen.

Bei den Frameparametern hängt der Ort von der Aufrufkonvention ab:

- `__cdecl`: in die obersten Frameadressen des Aufrufers, so dass sie in der aufgerufenen Funktion dann jenseits des Frame-Pointers sichtbar sind
- `__stdcall`: direkt in den neuen Frame

Aufruf und Rücksprung

Wurde g von Adresse a aus aufgerufen, so muss nach Abarbeitung von g an Adresse $a + \texttt{wordSize}$ fortgefahren werden. Diese Adresse muss daher vor Aufruf irgendwo abgespeichert werden.

Eine Möglichkeit ist der Stack; das wird von den Pentium `call` und `ret` Befehlen explizit unterstützt.

Eine andere Möglichkeit besteht darin, die Rücksprungadresse in einem Register abzulegen, welches dann ggf. von der aufgerufenen Funktion im Frame zu sichern ist.

Escapes

Lokale Variablen und Parameter dürfen nicht in Registern abgelegt werden, wenn vom Programm auf ihre Adresse zugegriffen wird. Das ist der Fall bei Verwendung von Cs Adressoperator, Pascals Call-by-Reference, und bei der Verwendung von Zeigerarithmetik.

Solche Parameter („escapes“) können nur über den Frame übergeben werden.

In MiniJava gibt es keine Escapes.

Verschachtelte Funktionen

In manchen Sprachen (z.B. Pascal oder ML) können lokale Funktionen innerhalb eines Blocks deklariert werden. In lokalen Funktionen kann auf lokale Variablen des äußeren Blocks Bezug genommen werden. Der inneren Funktion muss daher mitgeteilt werden, wo diese Variablen zu finden sind.

- *static link*: im aktuellen Frame wird ein Verweis auf den Frame der äußeren Funktion abgelegt.
- *lambda lifting*: man übergibt alle lokalen Variablen der äußeren Funktion als eigene Parameter.

Höherstufige Funktionen

Können Funktionen nach außen gegeben werden, so bleiben unter Umständen lokale Variablen nach Verlassen einer Funktion am Leben:

```
fun incBy x =
  let fun g y = x+y
   in g
  end
let incBy3 = incBy 3
let z = incBy3 1
```

Wird `incBy` verlassen, so darf `x` noch nicht zerstört werden!
In MiniJava gibt es allerdings keine höherstufigen Funktionen.

Implementierung von Aktivierungssätzen

Im Compiler werden die genannten maschinenabhängigen Aspekte eines Frames werden in einem Interface abstrahiert.

Die konkrete Implementierung eines Frames beinhaltet:

- Zugriffsmethoden der Parameter
- Anzahl der bisher allozierten lokalen Variablen
- Label des Codeanfangs (`name`)
- ...

Interface für Frames

```
public interface Frame {

  enum Location { ANYWHERE, IN_MEMORY };

  Label getName();

  int getParameterCount();

  TreeExp getParameter(int number);

  TreeExp allocLocal(Location l);

  TreeStm makeProc(TreeStm body, TreeExp returnValue);

  int size();

  Frame clone();
}
```

Interface für maschinenspezifische Aufgaben

```
public interface MachineSpecifics {

  int getWordSize();

  Frame newFrame(String name, int paramCount);


  /* Folgende Methoden werden später erklärt: */

  Temp[] getAllRegisters();

  Temp[] getGeneralPurposeRegisters();

  List<Assem> spill(Frame frame, List<Assem> is, List<Temp> sp);

  Fragment<List<Assem>> codeGen(Fragment<List<TreeStm>> frag);

  String printAssembly(List<Fragment<List<Assem>>> frags);
}
```

Architekturspezifische Implementierungen

Die abstrakten Klassen werden architekturspezifisch implementiert:

- `I386Frame extends Frame ...`
  `I386Specifics extends MachineSpecifics ...`
- `SparcFrame extends Frame ...`
  `SparcSpecifics extends MachineSpecifics ...`

Vorläufig gibt es zum Testen eine Dummy-Architektur:

- `DummyMachineFrame` implementiert Frame einer Test-Architektur, in der alle Parameter und lokale Variablen in Temporaries übergeben werden
- `DummyMachineSpecifics` enthält u.a. Factory-Methode *newFrame*

Ihre heutige Aufgabe

Implementieren Sie die Übersetzung in Zwischencode.

- Zum Zugriff auf lokale Variablen und Parameter soll ein `Frame`-Objekt im `Translate`-Visitor verwendet werden.
- Der Übersetzer soll das MiniJava-Programm als eine Liste von Zwischencode-Fragmenten ausgeben.
- Auf der Vorlesungsseite finden Sie die Klassen für die Dummy-Maschine, einen Übersetzer in `C` und die `C`-Laufzeitbibliothek, womit Sie den Zwischencode ausführen können.

`Translate`-Visitor-Klassen und Frames

- Für jede Methode wird ein neues `Frame`-Objekt angelegt, dass der Visitor für die Übersetzung des Methodenrumpfes verwendet.
- `Frame`-Objekte enthalten nicht die Zuordnung von Bezeichnern auf den `TreeExp`-Code für den Zugriff. Diese Zuordnung kann entweder in die Symboltabelle geschrieben werden, oder in ein Objekt einer neuen Klasse, die `Frame`-Objekte kapselt.
- Mit der *makeProc*-Methode des Frames werden die benötigten Statements für die Rückgabe der `return`-Expression angefügt.
- `TreeStm` und `TreeExp` haben eine *toString*-Methode, mit der Sie den generierten Zwischencode zu Testzwecken ausgeben können.

Fragmente

Die Übersetzung soll eine Liste von Fragmenten liefern, die durch die abstrakte Basisklasse `Fragment` repräsentiert werden.

- Codefragmente stellen Codeabschnitte in der Assembler-Sprache dar. Jedes `TreeStm`-Objekt jeder übersetzten Methode wird zusammen mit dem zur Methode gehörigen Frame in ein `FragmentProc`-Objekt gepackt, das der Liste hinzugefügt wird.
- Datenfragmente stellen Datenabschnitte dar. Dazu zählen z.B. die Deklaration von Stringkonstanten. In MiniJava gibt es keine Datenfragmente.

Ausführen des Zwischencodes

Sie können die vorgegebene Klasse *IntermediateToCmm* verwenden, um für die Dummy-Maschine übersetzten Zwischencode in C-Syntax auszugeben.

Dieser Code kann mit einem C-Compiler wie *gcc* übersetzt werden. Für ein lauffähiges Programm benötigten Sie natürlich auch die Laufzeitbibliothek `runtime.c`.