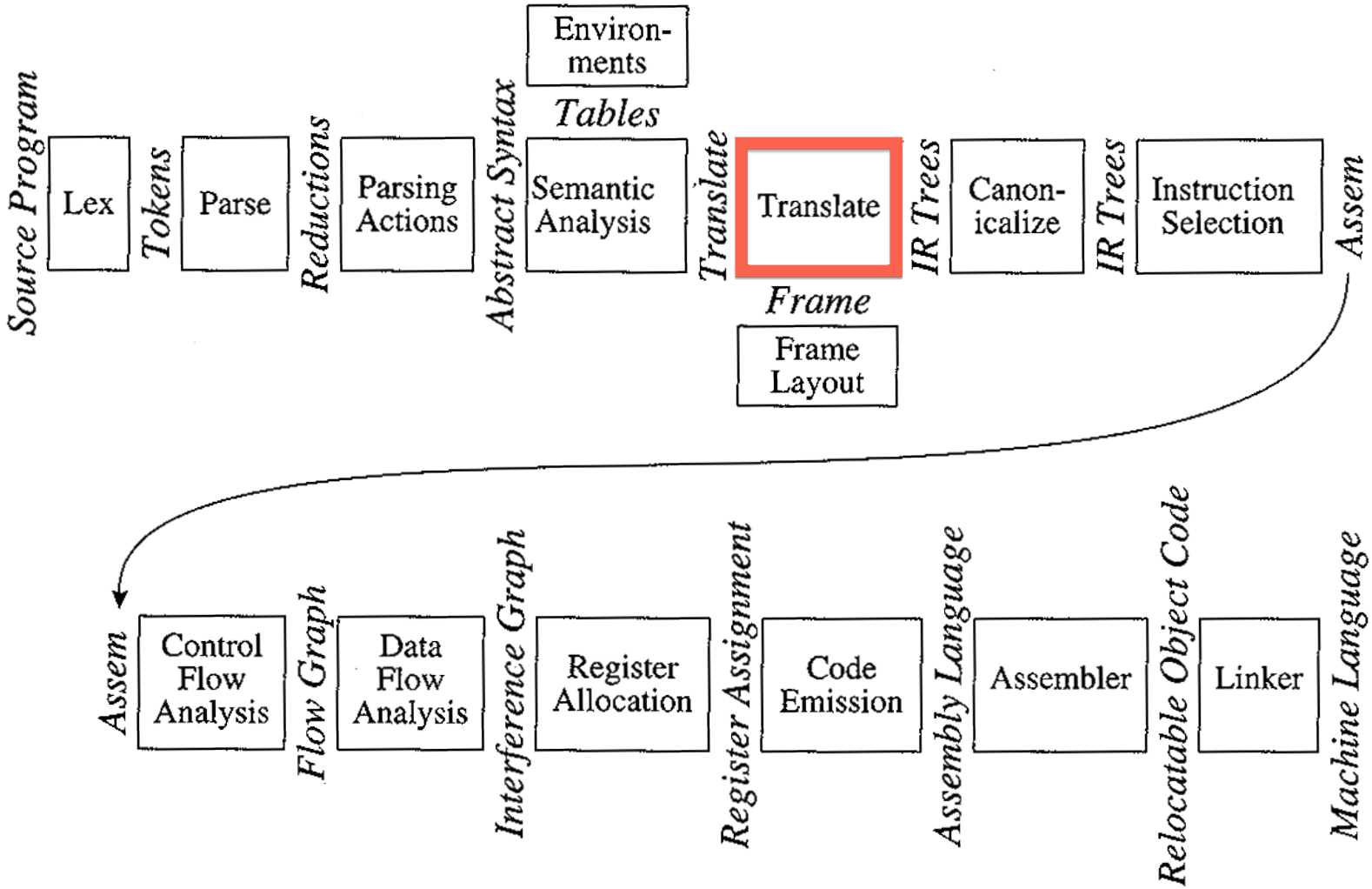
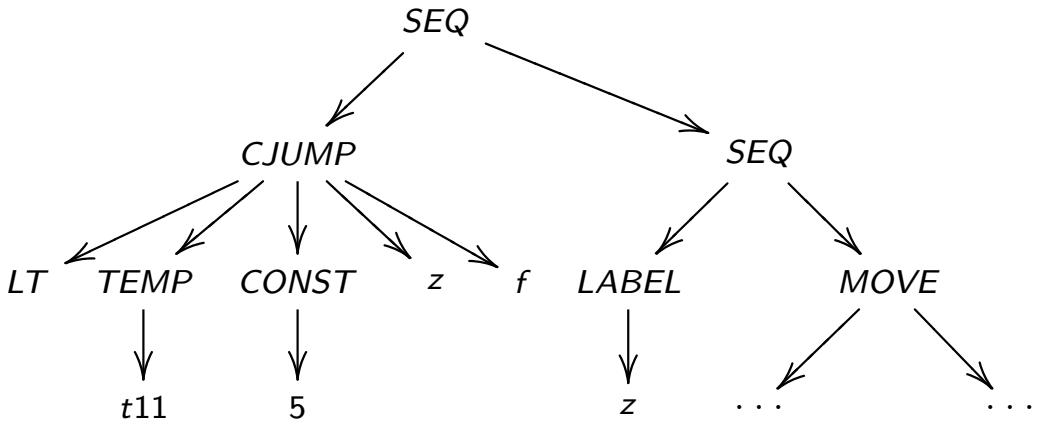
Übersetzung in Zwischencode

Übersetzung in Zwischencode



Überblick Zwischencode

- Die abstrakte Syntax soll in Zwischencode übersetzt werden.
- Die Zwischensprache befindet sich zunächst in etwa auf der gleichen Abstraktionsebene wie C, beschränkt sich aber auf eine minimale Anzahl von Konstrukten.
  (Manche Compiler nennen sie C--.)
- Die Übersetzung in die Zwischensprache entfernt höhere Programmiersprachenkonstrukte (z.B. Klassen).
- Für den Zwischencode wird keine konkrete Syntax angegeben; es wird gleich die abstrakte Syntax generiert.
  (Im Buch wird die Zwischensprache deshalb **Tree** genannt.)



Übersetzung höherer Sprachkonstrukte

Im Zwischencode gibt es keine Objekte oder komplizierte Datentypen.

Es gibt nur Integer-Werte und Zeiger auf Speicheradressen. Wir nehmen hier an, dass beide durch einen Typ `int32_t` von 32 Bit Integer-Werten repräsentiert sind.

Alle Datentypen, insbesondere Arrays und Objekte, müssen nun in solche einfachen Typen abgebildet werden und durch die Übersetzung entsprechend umgewandelt werden.

Übersetzung höherer Sprachkonstrukte

Modellierung von MiniJava-Typen:

- `int`: Integer-Werte
- `boolean`: Integer-Werte (0 = `false`, 1 = `true`)
- `int[]`: Ein Array `int[]` a wird durch einen Zeiger p in den Hauptspeicher repräsentiert.

Adresse	Inhalt
p	`a.length`
$p + w$	`a[0]`
$p + 2w$	`a[1]`
...	
$p + n \cdot w + w$	`a[n]`

wobei $n =$ `a.length` $- 1$ und $w = 4$.

Übersetzung höherer Sprachkonstrukte

- Klassen: Ein Objekt `obj` der Klasse `C` wird durch einen Zeiger p in den Hauptspeicher repräsentiert.

Adresse	Inhalt
p	Klassen-Id
$p + w$	`feld1`
$p + 2w$	`feld2`
$\ldots$	
$p + n \cdot w + w$	`feldn`

```
class C {
  int feld1;
  boolean feld2;
  ...
  SomeClass feldn;

  /* Methoden */
}
```

Ein Zeiger auf das Objekt `this` muss explizit gehalten und bei Funktionsaufrufen übergeben werden. Wir übergeben `this` immer als erstes Argument.

Beispiel: Von MiniJava . . .

```
class F {
  int lastnum;
  int lastresult;
  public int compute(int num) {
    int result;
    if (num < 1) {
      result = 1;
    } else {
      result = num * (this.compute(num - 1));
    }
    lastnum = num;
    lastresult = result;
    return result;
  }

  public int test() {
    System.out.println(this.compute(10));
    System.out.println(lastnum);
    System.out.println(lastresult);
    return 0;
  }
}
```

. . . nach C

```
#include <stdio.h>
#include <stdint.h>
#define MEM(x) *((int32_t*)(x))

int32_t F$compute(int32_t this, int32_t num) {
   int32_t result;
   if (num < 1) {
     result = 1;
   } else {
       result = num * (F$compute(this, num - 1));
   }
   MEM(this + 4) = num;
   MEM(this + 8) = result;
   return result;
}

int32_t F$test(int32_t this) {
   printf("%i ", F$compute(this, 10));
   printf("%i ", MEM(this + 4));
   printf("%i ", MEM(this + 8));
   return 0;
}
```

Zwischensprache

Wir definieren nun die abstrakte Syntax der Zwischensprache. Diese besteht nur aus Ausdrücken (TreeExp) und Anweisungen (TreeStm).

Diese Woche betrachten wir nur die Übersetzung der Körper von MiniJava-Funktionen in Zwischencode.

Nächste Woche befassen wir uns dann mit der Übersetzung von gesamten MiniJava-Programmen.

Ausdrücke (Expressions) in der Zwischensprache

- CONST(i): Die (Integer-) Konstante i.
- NAME(l): Die symbolische Konstante (z.B. Sprungadresse) l.
- TEMP(t): Die Variable t (*temporary*). Die Zwischensprache stellt beliebig viele Temporaries bereit. Temporaries können als lokale Variablen in C angesehen werden.
- BINOP(o, e_1, e_2): Die Anwendung des binären Operators o auf die Ausdrücke e_1 and e_2, die zuvor in dieser Reihenfolge ausgewertet werden.
  Operatoren:
  - PLUS, MINUS, MUL, DIV: vorzeichenbehaftete Rechenoperationen
  - AND, OR, XOR: *bitweise* logische Operationen
  - LSHIFT, RSHIFT, ARSHIFT: Schiebeoperationen

Ausdrücke (Expressions) in der Zwischensprache

- $\text{MEM}(e)$: Bezeichnet den Inhalt der Speicheradresse e.
- $\text{CALL}(f, e_1, \ldots, e_n)$: Aufruf der Funktion f mit Argumenten $e_1, \ldots, e_n$, die vorher in dieser Reihenfolge ausgewertet werden.
- $\text{ESEQ}(s, e)$: Ausführung von Statement s; danach Auswertung von e.

Anweisungen (Statements) der Zwischensprache

- $\text{MOVE}(\text{TEMP } t, e)$: Auswerten von e und abspeichern in temporary t.
- $\text{MOVE}(\text{MEM } e_1, e_2)$: Auswerten von e_1 zu Adresse a. Den Wert von e_2 in Speicheradresse a abspeichern.
- $\text{MOVE}(\text{ESEQ}(s_1, e_1), e_2)$: Entspricht $\text{SEQ}(s_1, \text{MOVE}(e_1, e_2))$.
- $\text{EXP}(e)$: Auswerten von e und verwerfen des Ergebnisses.
- $\text{JUMP}(e, \mathit{labs})$: Ausführung bei Adresse e fortsetzen, wobei die Liste *labs* alle möglichen Werte für diese Adresse angibt.
- $\text{CJUMP}(o, e_1, e_2, l_{\mathit{true}}, l_{\mathit{false}})$: Bedingter Sprung: Vergleiche die Werte von e_1 und e_2 mit Operator o. Ist der Vergleich wahr, so verzweige nach l_{true}, sonst nach l_{false}.
  Vergleichsrelationen:
  - EQ, NE, LT, GT, LE, GE: vorzeichenbehaftet
  - ULT, UGT, ULE, UGE: vorzeichenfrei
- $\text{SEQ}(s_1, s_2)$: Sequentielle Ausführung.
- $\text{LABEL}(l)$: Definition der Sprungadresse l.

Zwischensprache — Beispiel

```
int32_t LF$compute(int32_t t2, int32_t t3) {
  int32_t t0, t4, t5, t10, t11;
  /* MOVE(TEMP(t5), CONST(0)) */
  t5 = 0;
  /* CJUMP(LT, TEMP(t3), CONST(1), L$$3, L$$4) */
  if (t3 < 1) goto L$$3; else goto L$$4;
  /* LABEL(L$$3) */
L$$3: ;
  /* MOVE(TEMP(t5), CONST(1)) */
  t5 = 1;
  /* LABEL(L$$4) */
L$$4: ;
  /* CJUMP(EQ, TEMP(t5), CONST(1), L$$0, L$$1) */
  if (t5 == 1) goto L$$0; else goto L$$1;
  /* LABEL(L$$0) */
L$$0: ;
  /* MOVE(TEMP(t4), CONST(1)) */
  t4 = 1;
  /* JUMP(NAME(L$$2), [L$$2]) */
  goto L$$2;
  /* LABEL(L$$1) */
L$$1: ;
  /* MOVE(TEMP(t11), TEMP(t3)) */
  t11 = t3;
  /* MOVE(TEMP(t10), CALL(NAME(LF$compute), TEMP(t2), OP(MINUS, TEMP(t3), CONST(1)))) */
  t10 = LF$compute(t2, (t3 - 1));
  /* MOVE(TEMP(t4), OP(MUL, TEMP(t11), TEMP(t10))) */
  t4 = (t11 * t10);
  /* LABEL(L$$2) */
L$$2: ;
  /* MOVE(MEM(OP(PLUS, TEMP(t2), CONST(8))), TEMP(t3)) */
  MEM((t2 + 8)) = t3;
  /* MOVE(MEM(OP(PLUS, TEMP(t2), CONST(4))), TEMP(t4)) */
  MEM((t2 + 4)) = t4;
  /* MOVE(TEMP(t0), TEMP(t4)) */
  t0 = t4;
  return t0;
}
```

Laufzeitbibliothek und externe Aufrufe

Die Speicherverwaltung sowie sie Ein- und Ausgabe wird von einer Laufzeitbibliothek übernommen.

Wir verwenden zunächst:

- `L_halloc(n)`: reserviert *n* Bytes im Speicher, initialisiert diesen mit Nullen und liefert einen Zeiger darauf zurück.
- `L_println_int(i)`: gibt die Zahl *i* auf dem Bildschirm aus.
- `L_print_char(i)`: gibt das *i*-te Zeichen auf dem Bildschirm aus.
- `L_raise(i)`: gibt Fehlercode *i* aus und bricht das Programm mit Laufzeitfehler ab.

Diese Funktionen werden wie alle anderen Funktionen mit CALL aufgerufen, sind jedoch extern in einer Laufzeitbibliothek definiert.

Diese Bibliothek ist in `C` implementiert und wird am Ende mit dem übersetzten MiniJava-Code verlinkt.

Zwischensprache im Compiler

Zwischensprachen-Ausdrücke bzw. -Anweisungen werden durch die Klassen `TreeExp` bzw. `TreeStm` mit ihren Unterklassen repräsentiert.

Ein `ExpVisitor` bzw. `StmVisitor` übersetzt Objekte vom Typ `Exp` bzw. `Stm` in Objekte vom Typ `TreeExp` und `TreeStm`.

Ihre heutige Aufgabe

Entwicklung einer Übersetzung von Ausdrücken und Anweisungen MiniJava-Syntax in Zwischensprachen-Syntax.

- Überlegen Sie sich (evtl. zunächst auf Papier), wie die einzelnen Ausdrücke und Anweisungen der abstrakten MiniJava-Syntax in die Zwischensprache übersetzt werden können.
- Sie können auch schon mit der Implementierung der *Translate*-Visitorklassen beginnen; die Klassen für die Zwischensprachen-Syntax finden Sie auf der Vorlesungsseite.
- Für die Aufgabe ist teilweise auch noch in der nächsten Woche Zeit.

Zugriff auf Parameter und lokale Variablen

Der Zugriff auf Parameter (inklusive *this*) und lokale Variablen einer Methode ist von sogenannten Aktivierungssätzen (*Frames*) abhängig. Diese werden erst in der nächsten Woche besprochen. Im Moment können Sie davon ausgehen, dass es Methoden

- `TreeExp getParameter(String name)` und
- `TreeExp getLocalVariable(String name)`

gibt, die den Zwischensprache-Code liefert, der für den Zugriff auf die entsprechenden Parameter und Variablen notwendig ist.

Übersetzung — Ausdrücke

Viele MiniJava-Ausdrücke können direkt in TreeExp-Objekte übersetzt werden, z.B.:

$$\texttt{translate}(e_1 + e_2) = \textsc{binop}(\textsc{plus}, \texttt{translate}(e_1), \texttt{translate}(e_2))$$

Boolesche Vergleiche müssen durch Sprünge behandelt werden (analog zu bedingten Sprüngen in Assembler), z.B.:

$$\begin{aligned}
&\texttt{translate}(e_1 < e_2) = \\
&\quad \textsc{eseq}(\{\textsc{move}(t, \textsc{const}(0)); \\
&\qquad \textsc{cjump}(\textsc{lt}, \texttt{translate}(e_1), \texttt{translate}(e_2), l_{\text{true}}, l_{\text{false}}); \\
&\qquad \textsc{label}(l_{\text{true}}); \textsc{move}(t, \textsc{const}(1)); \\
&\qquad \textsc{label}(l_{\text{false}})\}, \\
&\quad \textsc{temp}(t))
\end{aligned}$$

Hierbei ist t eine *neue* Variable und l_{true}, l_{false} sind *neue* Labels.

Übersetzung — Ausdrücke

Beachte die Semantik von `&&` und `||`:
Wenn der linke Teilausdruck schon falsch beziehungsweise wahr ist, wird der rechte nicht mehr ausgewertet.

Bei den bitweisen Operatoren `&` und `|` müssen immer beide Teilausdrücke ausgewertet werden.

Beachte auch, dass Java bei Zugriffen auf Arrays einen Test auf Überschreitung der Arraygrenzen vorschreibt.

Übersetzung — Methodenaufrufe

Methodenaufrufe werden ziemlich direkt in Funktionsaufrufe in der Zwischensprache übersetzt. Zu bemerken ist folgendes:

- In MiniJava gibt es kein „overloading“ oder „dynamic dispatch“. Die Methodenauswahl richtet sich daher nach dem *statischen Typ* (durch semantische Analyse ermittelte Klasse) des aufgerufenen Objekts.
- Die Methode m in Klasse C wird durch eine Zwischensprachen-Funktion mit Namen $C\$m$ repräsentiert.
- Der Funktion $C\$m$ ist das aufgerufene Objekt als zusätzlicher Parameter zu übergeben.

Der Aufruf $e.m(e_1, \ldots, e_n)$ wird folgendermaßen übersetzt:
$\textsc{call}(C\$m, [\texttt{translate}(e), \texttt{translate}(e_1), \ldots, \texttt{translate}(e_n)])$
wobei $\texttt{translate}(e)$ den „this“-pointer der aufgerufenen Methode bestimmt.

Übersetzung — Anweisungen

Die Erzeugung von Objekten und Arrays wird durch den Aufruf der Bibliotheksfunktion `L_halloc(n)` realisiert, z.B.:

$$\texttt{translate}(\textit{new}\ \texttt{C}()) = \textsc{call}(\textsc{name}(\texttt{L\_halloc}), s)$$

Hierbei ist s die Größe in Bytes, die für ein Objekt der Klasse `C` im Speicher benötigt werden.

If und While Konstrukte werden in $\textsc{cjump}(\ldots)$ mit frischen Labels für die Zweige übersetzt. Vorerst erhalten beide Zweige eines If Konstrukts Labels, später wird der Code zu einem „fall-through“ im Else-Zweig umgewandelt.

Übersetzung — Allgemeine Hinweise

Nicht an Variablen und Labels sparen!

Für die kurzfristige Speicherung von Zwischenergebnissen immer eine *neue* Variable nehmen.

Temporäre Variablen nicht recyclen, neue nehmen.

So können subtile Fehler vermieden werden, in denen eine wiederverwendete Variable durch anderswo generierten Code verändert wird.

Wiederverwendung von Variablen macht später das Leben des Registerallokators schwerer.

Generierung neuer Variablen und Labels durch statische Funktionen `Temp.freshTemp()` und `Label.freshLabel()`.

Arrayzugriffe

Optimierung

Bei Arrayzugriffen mit konstantem Index wie $a[17]$ kann der Offset für die entsprechende Speicherstelle statisch berechnet werden.

Prüfung der Grenzen

Die MiniJava-Semantik verlangt, dass vor einem Arrayzugriff die Einhaltung der Arraygrenzen überprüft wird.

Für einen Zugriff $a[i]$ sollte daher eine Laufzeitprüfung $0 \leq i < a.\texttt{length}$ im Zwischencode generiert werden, die im Fehlerfall mit der Bibliotheksfunktion `L_raise` eine entsprechende Fehlermeldung auslöst.

Man beachte, dass bei Arrayzuweisungen nur die Adresse, nicht aber der ganze Code für die Laufzeitprüfung als Ziel in einem MOVE-Statement stehen kann.

Optimierte Verzweigungen

Übersetzung eines **if**-Statements aus MiniJava:

$$\begin{aligned}&\texttt{translate}(\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2) = \\ &\quad \text{SEQ}(\{\text{CJUMP}(\text{EQ}, \texttt{translate}(e), \text{CONST}(1), l_{\text{true}}, l_{\text{false}}); \\ &\qquad \text{LABEL}(l_{\text{true}}); \texttt{translate}(S_1); \text{JUMP}(l_{\text{exit}}); \\ &\qquad \text{LABEL}(l_{\text{false}}); \texttt{translate}(S_2); \text{LABEL}(l_{\text{exit}})\})\end{aligned}$$

Die Bedingung e wird mit bedingten Sprüngen ausgewertet und in den meisten Fällen in ein Temporary geschrieben. Erst dann wird das Ergebnis mit 1 verglichen und nach l_{true} oder l_{false} gesprungen.

Eine mögliche Optimierung ist, die Bedingung so zu kompilieren, dass sofort nach l_{true} oder l_{false} gesprungen wird, wenn klar ist dass sie wahr bzw. falsch ist.

Optimierte Verzweigungen

Verbesserte Übersetzung des **if**-Statements:

$$\begin{aligned}
&\texttt{translate}(\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2) = \\
&\quad \textsc{seq}(\{\texttt{translateCond}(e, l_{\text{true}}, l_{\text{false}}); \\
&\qquad \textsc{label}(l_{\text{true}}); \texttt{translate}(S_1); \textsc{jump}(l_{\text{exit}}); \\
&\qquad \textsc{label}(l_{\text{false}}); \texttt{translate}(S_2); \textsc{label}(l_{\text{exit}})\})
\end{aligned}$$

Dabei übersetzt `translateCond` Ausdrücke vom Typ `boolean` in Anweisungen der Zwischensprache:

$$\begin{aligned}
&\texttt{translateCond}(\texttt{true}, l_{\text{true}}, l_{\text{false}}) = \textsc{jump}(l_{\text{true}}) \\
&\texttt{translateCond}(\texttt{!}e, l_{\text{true}}, l_{\text{false}}) = \texttt{translateCond}(e, l_{\text{false}}, l_{\text{true}}) \\
&\texttt{translateCond}(e_1 \texttt{\&\&} e_2, l_{\text{true}}, l_{\text{false}}) = \\
&\qquad \textsc{seq}(\{\texttt{translateCond}(e_1, l, l_{\text{false}}); \\
&\qquad\quad \textsc{label}(l); \\
&\qquad\quad \texttt{translateCond}(e_2, l_{\text{true}}, l_{\text{false}});\ \})
\end{aligned}$$