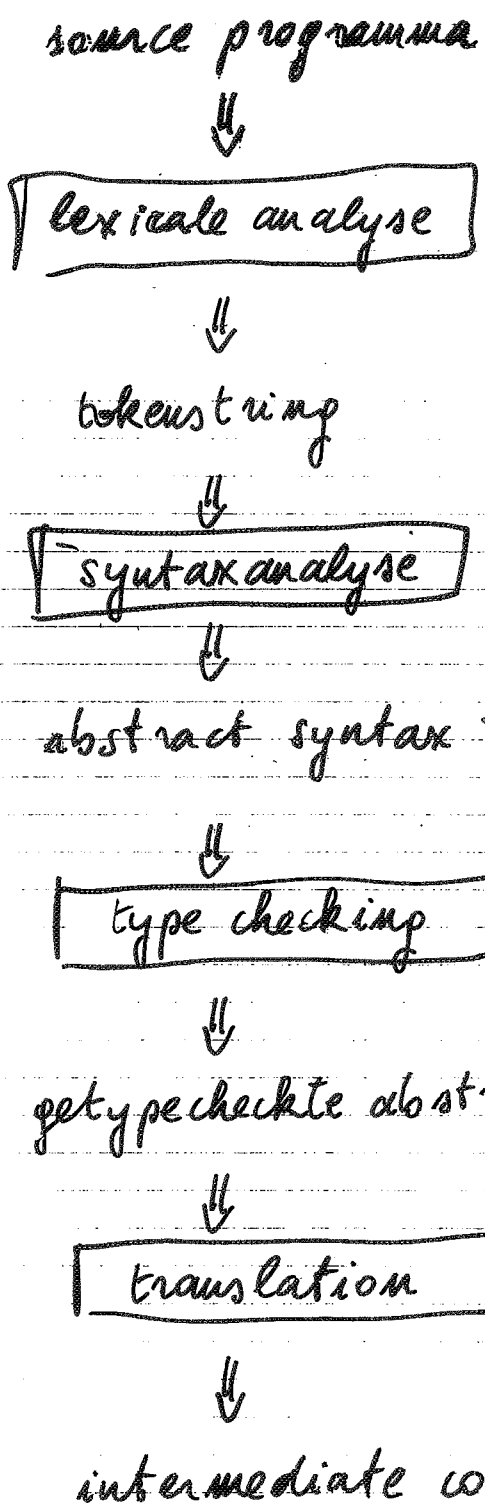
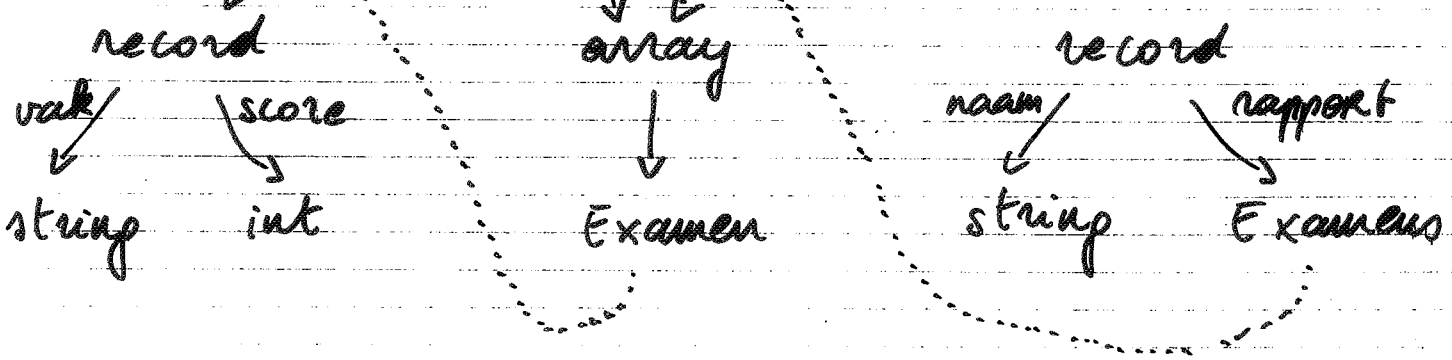
Front End


source programma
lexicale analyse
tokenstring
syntaxanalyse
abstract syntax tree
type checking
getypechekte abstract syntax tree
translation
intermediate code

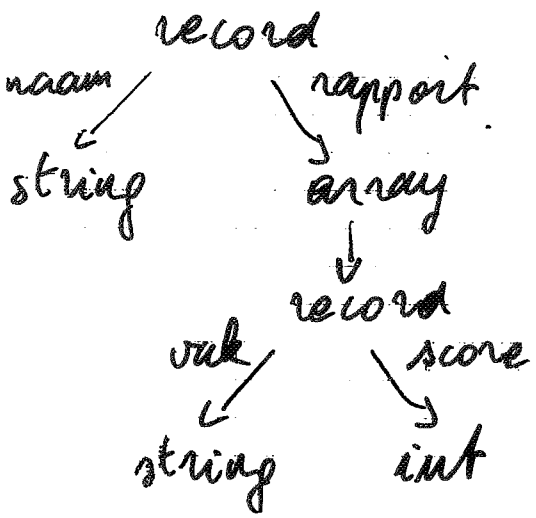
Type declaraties

```
type Examen = { vak : string,
                score : int }

Examens = array of Examen

Student = { naam : string,
            rapport : Examens }
```



Type-environment associëert typeexpressies aan typenamen

```
type Student2 = { naam : string,
                  rapport : array of
                      { vak : string,
                        score : int } }
```



Type-gelijkheid

Zijn Student en Student2 gelijk?

b.v. var x: Student
var y: Student2
x := y toegelaten?

- Structurele typegelijkheid: JA
- Naam equivalentie: NEE
  ↳ elke type expressie definiëert een ander type

! Naïef algoritme voor testen van structurele typegelijkheid werkt niet voor recursieve types

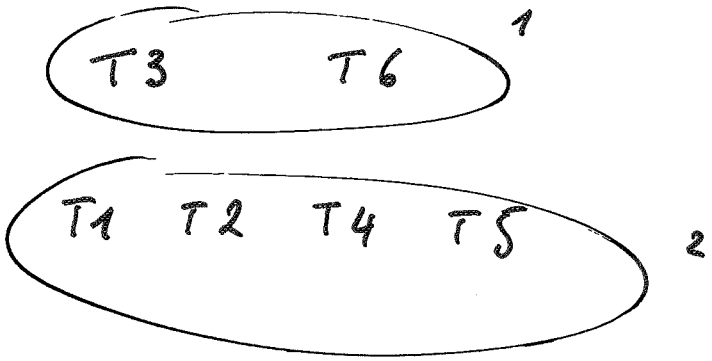
```
type list1 = { val: int, next: list1 }
type list2 = { val: int, next: list3 }
type list3 = { val: int, next: list2 }
```

- naïef algoritme: ∞ loop
- mooie oplossing: "Bisimilariteit" [Algol 68]

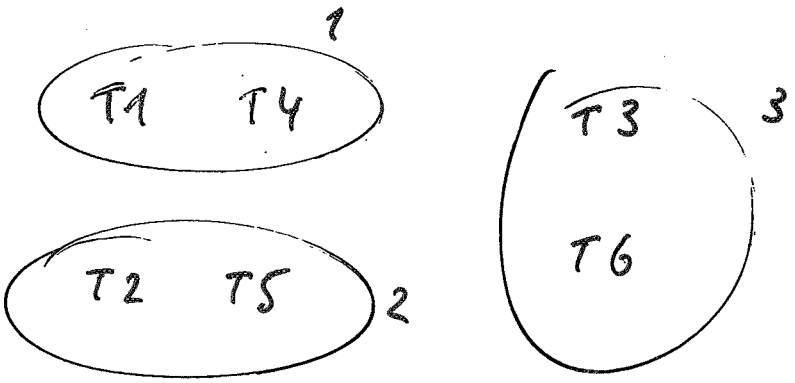
Type 1 = {a : Type 2}
Type 2 = {a : Type 3}
Type 3 = {b : Type 1}

Type 4 = {a : Type 5}
Type 5 = {a : Type 6}
Type 6 = {b : Type 1}

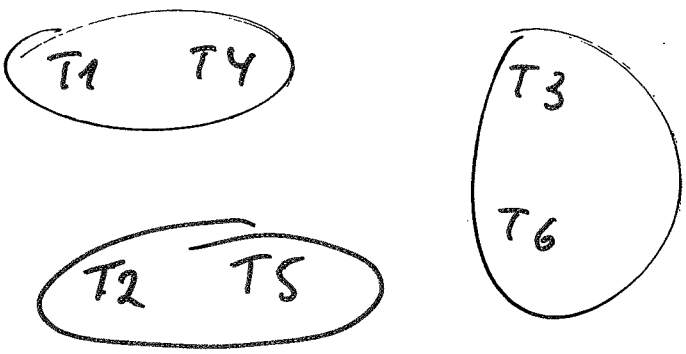
T1 = {a : 1}
T2 = {a : 1}
T3 = {b : 1}
T4 = {a : 1}
T5 = {a : 1}
T6 = {b : 1}



T1 = {a : 2}
T2 = {a : 1}
T3 = {b : 2}
T4 = {a : 2}
T5 = {a : 1}
T6 = {b : 2}



T1 = {a : 2}
T2 = {a : 3}
T3 = {b : 1}
T4 = {a : 2}
T5 = {a : 3}
T6 = {b : 1}



fixpunt!

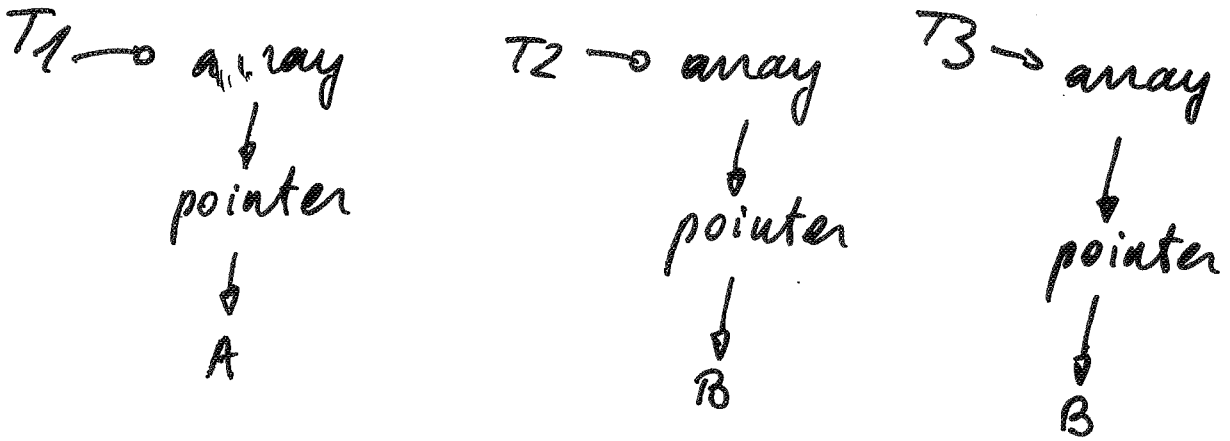
Java

Naamequivalentie (+ inheritance)

C (++)

mix
- structureel : pointers, arrays
- naam : record (struct)

```
struct A {int a; char b}
struct B {int a; char b}
typedef A *T1[]
typedef B *T2[]
typedef B *T3[]
```



$T_1 \neq T_2 = T_3$

Variabele declaraties

var x: Student

Variabele-environment associëert
type-expressie met elke variabele
(a.h.s. type-environment)

Functie declaraties

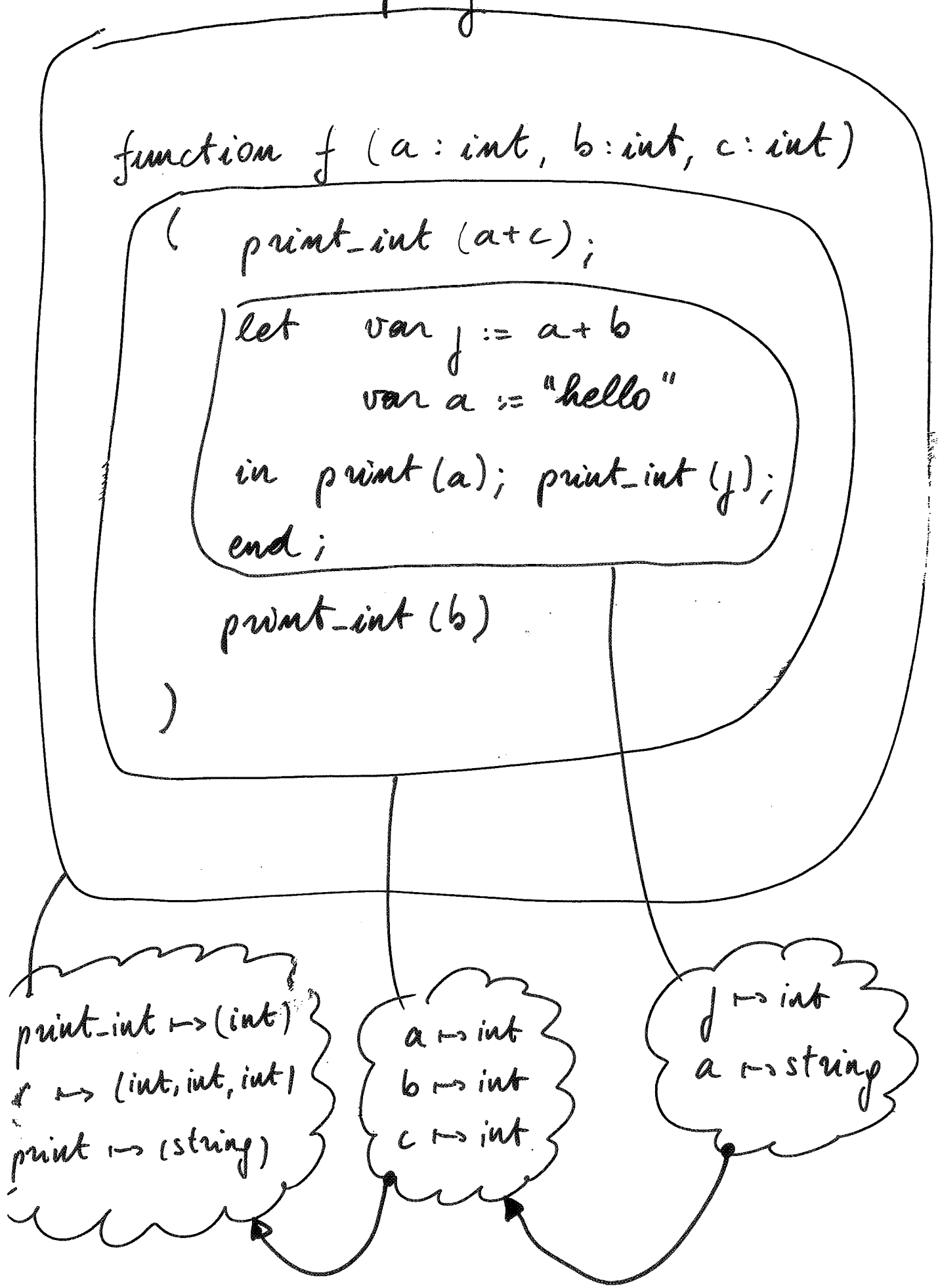
Functie-environment

per functienaam:

- type-expressies van formele parameters
- type-expressie van returntype

↳ niet voor procedures

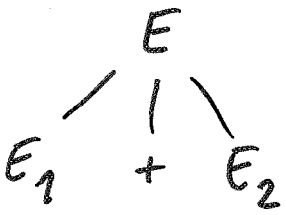
Scoping


function f (a : int, b : int, c : int)
(
print_int (a+c);
let var j := a + b
var a := "hello"
in print (a); print_int (j);
end;
print_int (b)
)
print_int ↦ (int)
r ↦ (int, int, int)
print ↦ (string)
a ↦ int
b ↦ int
c ↦ int
j ↦ int
a ↦ string

Typechecking van expressies

Controleer voor elke expressie de types van de deelexpressies (recursief) en, indien correct, ken type toe aan expressie.

- Variabele : lookup in current environment
  - scoping !
  - if not found : error

- 

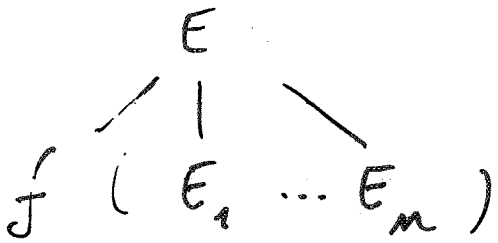
  als $E_1 \Rightarrow int$
  $E_2 \Rightarrow int$

  dan $E \Rightarrow int$

  else error

  - overloading ...

•



lookup f : $(T_1 \dots T_n ; T_0)$

als $E_1 \Rightarrow T_1$

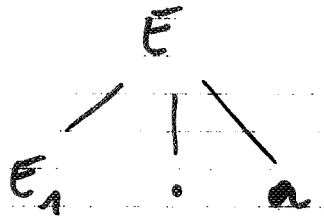
$\vdots$

$E_n \Rightarrow T_n$

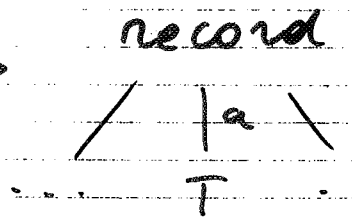
dan $E \Rightarrow T_0$

else error

•



als $E_1 \Rightarrow$



dan $E \Rightarrow T$

enz ...

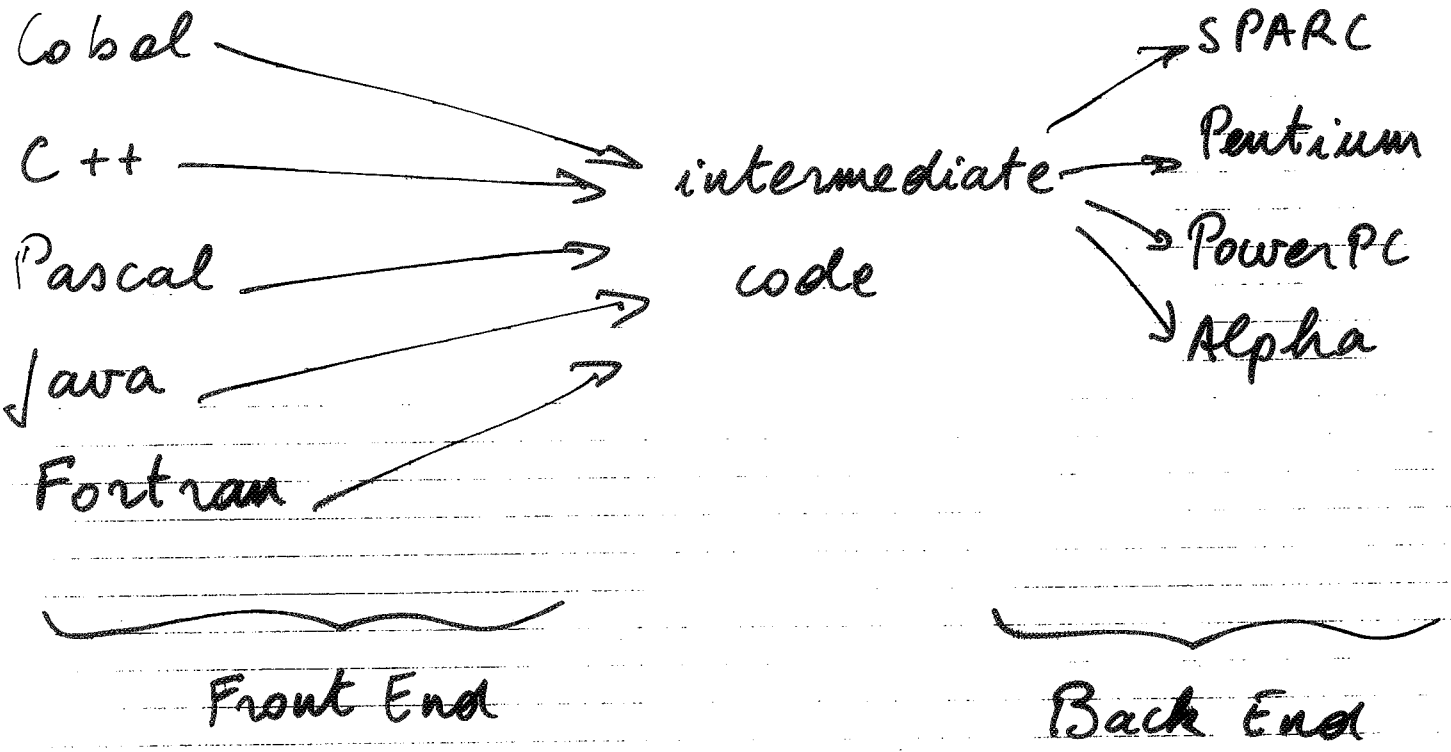
Typechecking van statements

- $E_1 := E_2$ test
  - $E_1 \Rightarrow T$
  - $E_2 \Rightarrow T$
  - E_1 l-value
- while (E_1) do B

  test $E_1 \Rightarrow$ boolean

enz ...

Vertaling naar intermediate code



Intermediate code:

- soort v. abstracte machinetaal
- vertalingen moeten gemakkelijk zijn

Expressies

- CONST(i)
  - ↳ constante int
- TEMP(t) registers
  - ↳ will. naam
- BINOP(e_1, op, e_2)
  - ↳ +, −, ×, /
- MEM(e)
- CALL(f; e_1, ..., e_m)

- enkel nog rekenkundige expressies!
  - L geen vergelijkings- of logische operatoren
  - L soms wel nog bitoperatoren zoals LSHIFT
- alles 1 machinewoord
  - L geen records, arrays
- expliciete memory-access

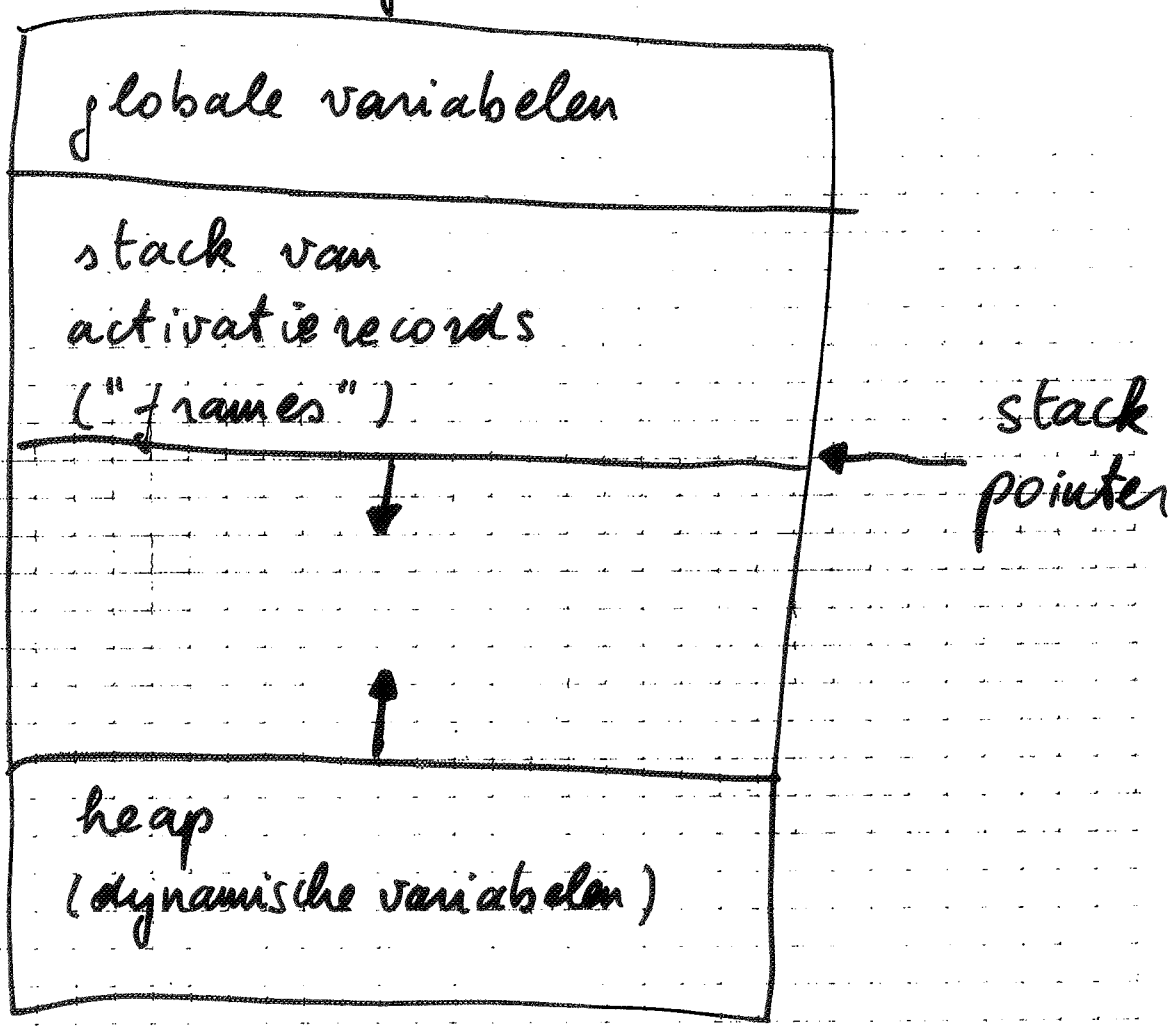
Statements

- MOVE (TEMP(t), e)
- MOVE (MEM(e_1), e_2)
- LABEL (ℓ)
- JUMP (ℓ)
- CJUMP (e_1, θ, e_2, t, f)
  ↳ $<, >, \leq, \geq, =, \neq$

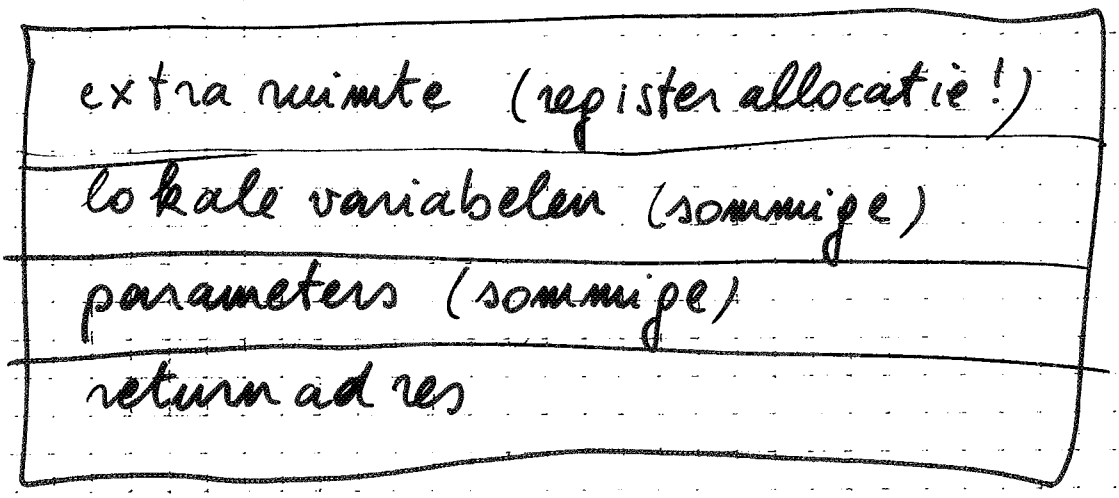
• geen gestructureerde statements
(if-then-else, while-do, repeat-until, for, ...)

Stack Frames

- Geheugen v/h programma:



- Structuur v/e frame:



- Welke parameters, lokale variabelen moeten in het frame ?

  Als we z'n adres nodig hebben : "escape"

  - & x
  - wordt doorgegeven "by reference"
  - startadres v. record of array

- Andere parameters en lokale variabelen houden we liever in registers :

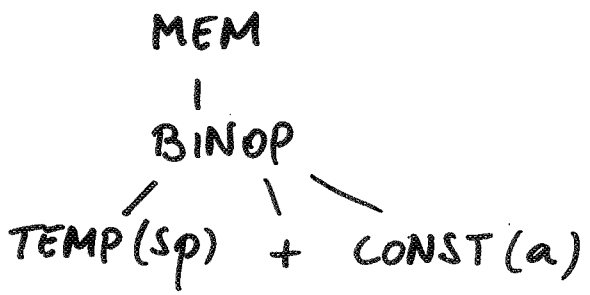
  - sneller dan geheugen
  - intermediate code : TEMPs, onbeperkt veel
  - echte machine : beperkt aantal

    => register assignment (zie later)

Vertaling van variabelen

- globale variabelen : vast adres MEM(a)
- escapende lokale variabele of parameter : vaste offset van stackpointer

  : MEM (BINOP (TEMP (sp), +, a))



- anderen :

  TEMP (naam)

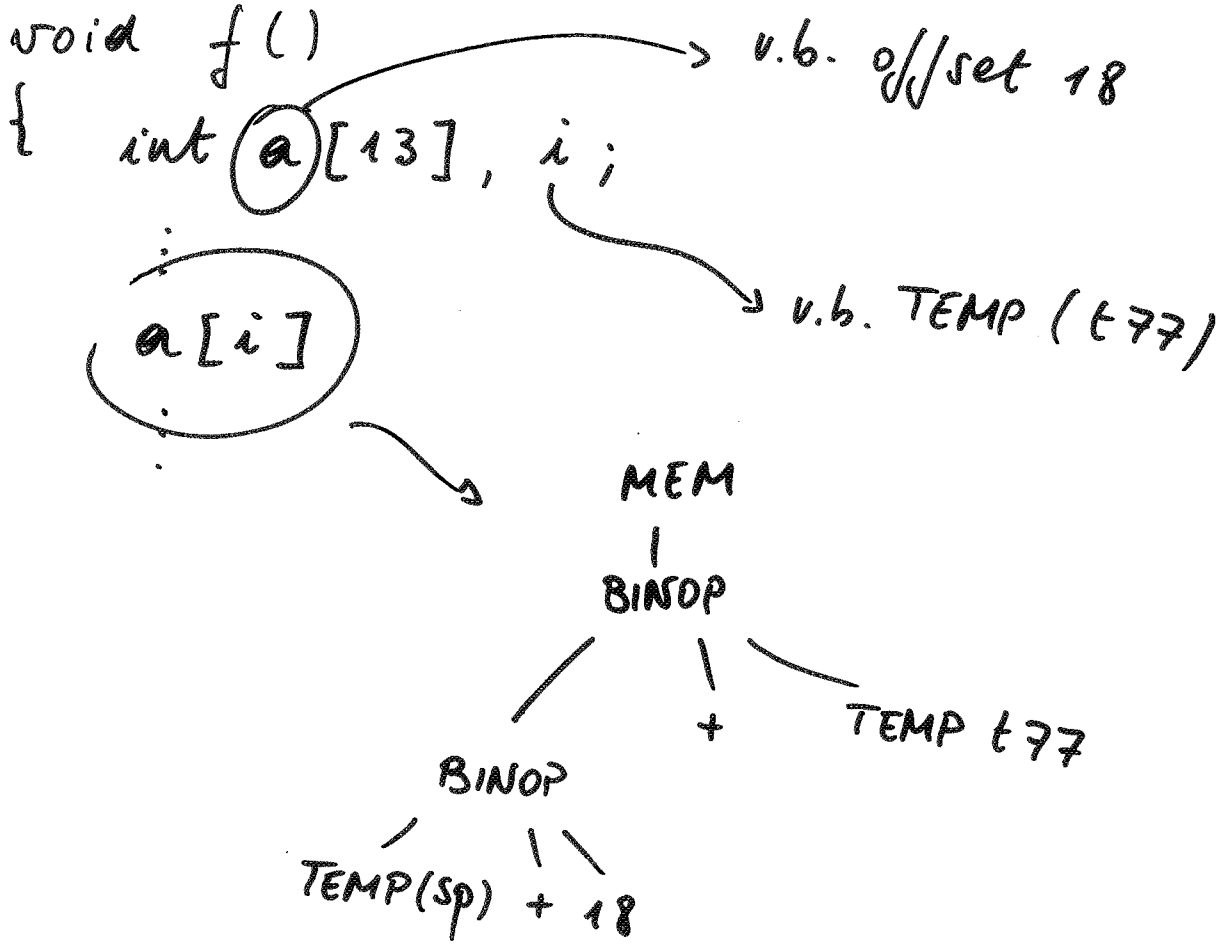
  L vb t 834

  L onthouden in symbol table

- Arrays : statisch / dynamisch

N.B. Records : analoog

Statisch : volledig in frame (of globaal geheugen)



vb C

→ vb Java, ML

Dynamisch: automatisch op heap alloceren

⇒ a staat enkel voor startadres

└ geen escape!

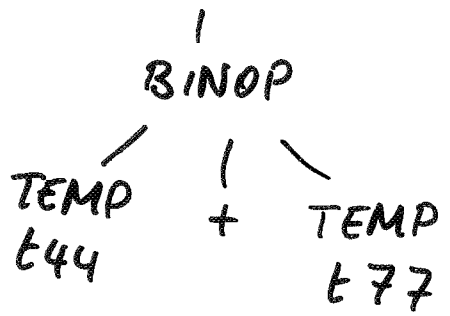
└ v.b. TEMP(t44)

⇒ creatie: call naar externe memory-allocator

wordt later ↓
mee gelinkt met objectcode

b.v. MOVE(TEMP(t44),
CALL(malloc, 13))

⇒ a[i] wordt nu



! Statisch of dynamisch, een goede compiler genereert ook automatisch bound checks:

a[i] ⇒ if i < 0 || i > 12
/* raise exception */

Vertaling van functiecalls

1. stackpointer verlagen : #bytes nodig voor frame van opgeroepen functie
   (is eigenlijk nog niet bekend ...)
   ↳ later "backpatchen"
2. argumenten moven in frame, of plaatsen in registers
3. returnadres (5) moven in frame
4. goto label v. functie body
5. returnwaarde recuperenen (register)
6. stackpointer herstellen

! Frame-layout en functie call-conventies zijn eigenlijk architectuurafhankelijk ...

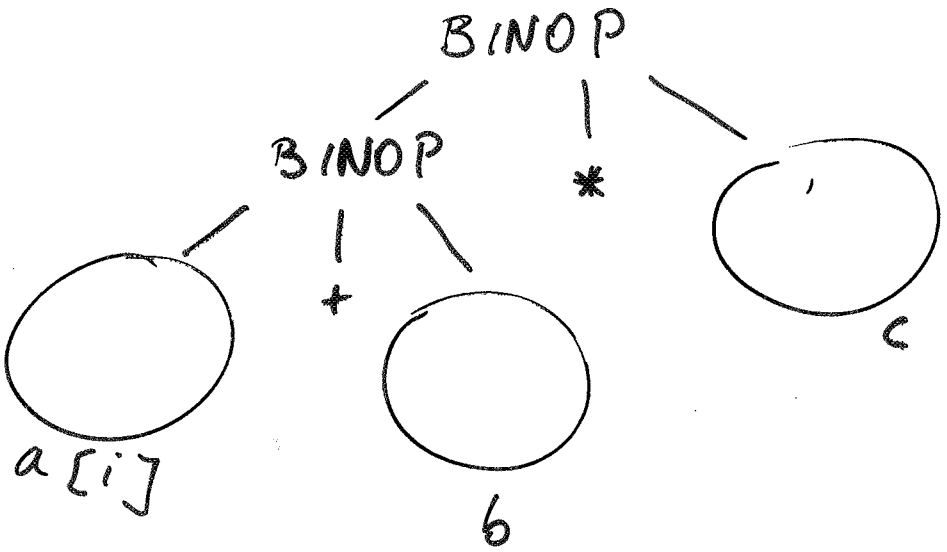
Vertaling van functies

1. body uitvoeren
2. terugspringen naar returnadres

Vertaling van rekenkundige expressies

Niets te doen: intermediate code heeft zelf rekenkundige expressies

b.v. `(a[i] + b) * c`



Booleaanse operatoren

b.v. `a || b`

- Naïef uitrekenen in een nieuwe temp:

```
if a != 0 then temp := 1
else if b != 0 then temp := 1
else temp := 0
```

- OK bij b.v. `flag := a || b`
- Echter dikwijls inefficiënt

vb if (a || b) then /* blah 1 */
else /* blah 2 */

bepaal temp;
CJUMP(temp, =, 1, t, f)
LABEL(t)
/* blah 1 */
LABEL(f)
/* blah 2 */

nieuw gemaakte labels

- Beter: short-circuit evaluatie

a || b :
- if a != 0 then goto □1
  else if b != 0 then goto □2
  else goto □3
- "trues": 1, 2
- "falses": 3

⇒ if (a || b) then /*blah1*/ else /* blah 2 */:

if a != 0 then goto t
else if b != 0 then goto t
else goto f
LABEL(t)
/* blah 1 */
LABEL(f)
/* blah 2 */

"backpatching":
vul t in "trues"
f in "falses"

Vertaling van controlestructuren

- if-then-else : CJUMP

b.v. if x < 5 then /* blah 1 */
(else /* blah 2 */

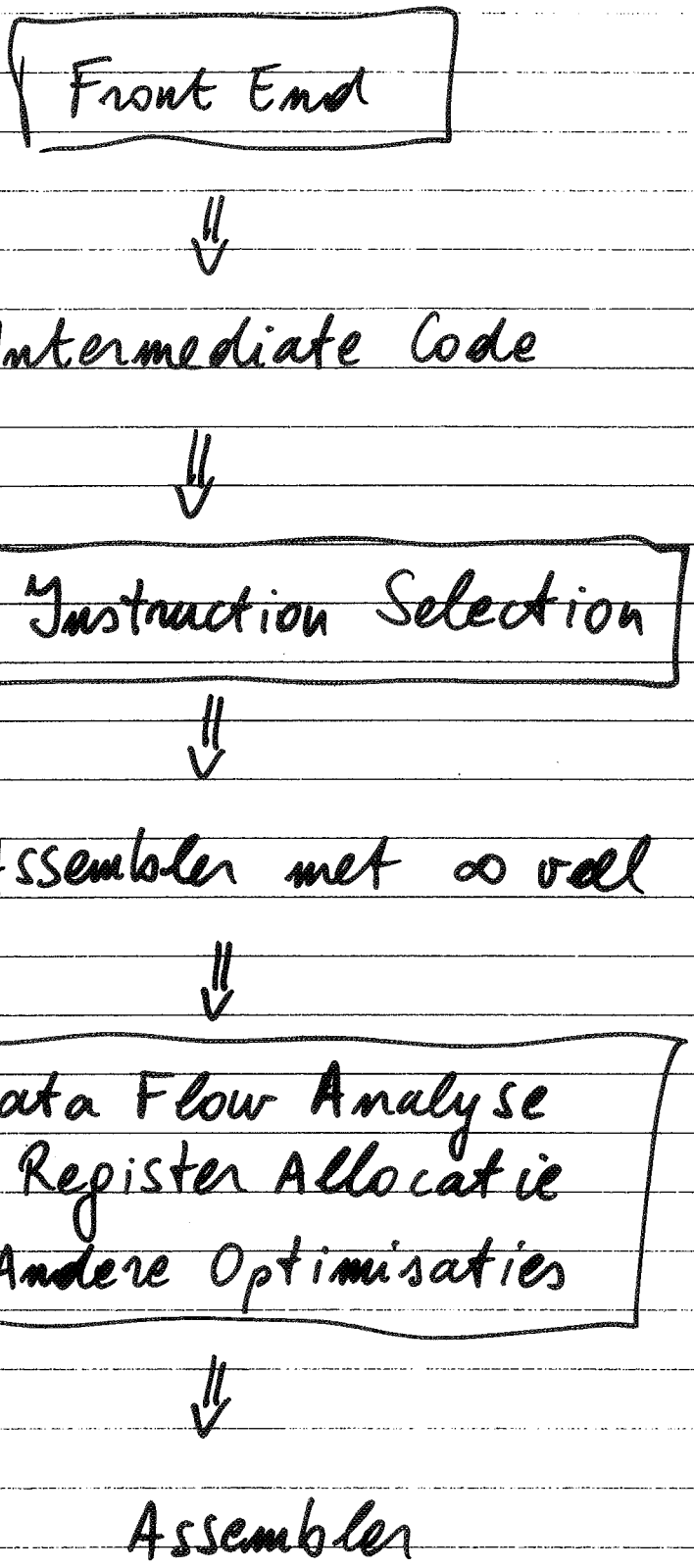
vb TEMP(t66)

CJUMP(TEMP(t66), <, 5, t, f)
LABEL(t)
/* blah 1 */
LABEL(f)
/* blah 2 */

2 nieuwe labels

- while, repeat, for, ... :
  vertaal mbv if-then-else en gotos

Back End

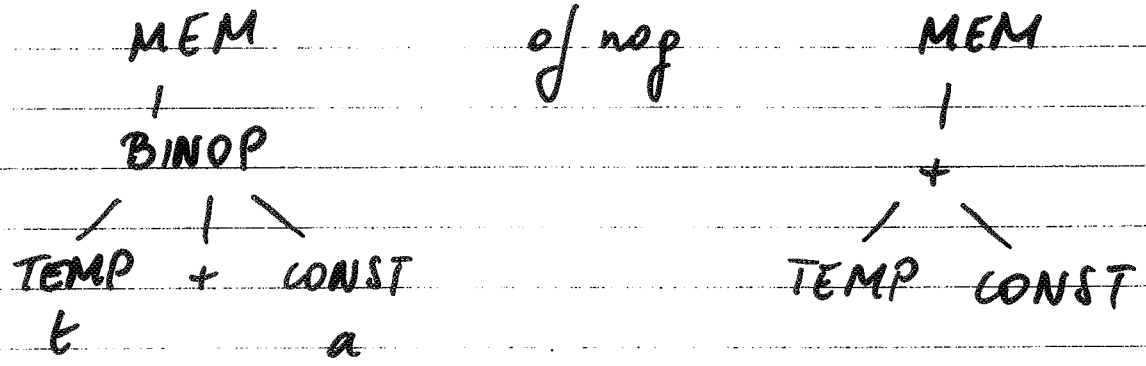

Front End
Intermediate Code
Instruction Selection
Assembler met ∞ veel registers
Data Flow Analyse
Register Allocatie
Andere Optimisaties
Assembler

Instruction Selection

Patronen

Typische machine-instructie doet meer dan slechts 1 operatie uit de intermediate code!

v.b.



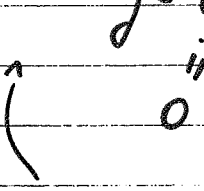
⇒ Stel elke machine-instructie voor als een patroon in intermediate code

Voorbeelden van Patronen van Machine-Instructies

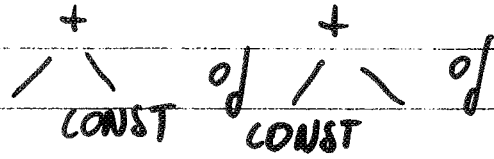
r_i TEMP

ADD $r_i \leftarrow r_j + r_k$

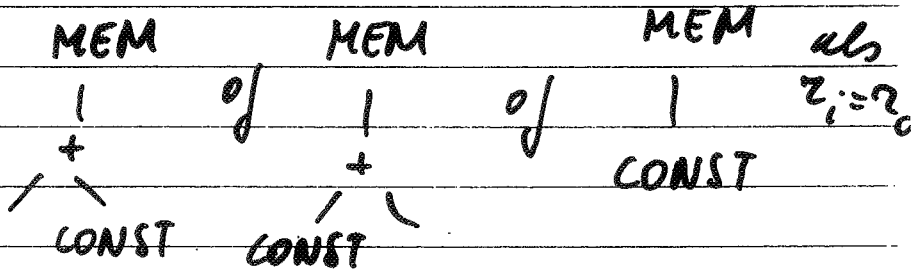
als $r_j = r_0$



ADDI $r_i \leftarrow r_j + c$

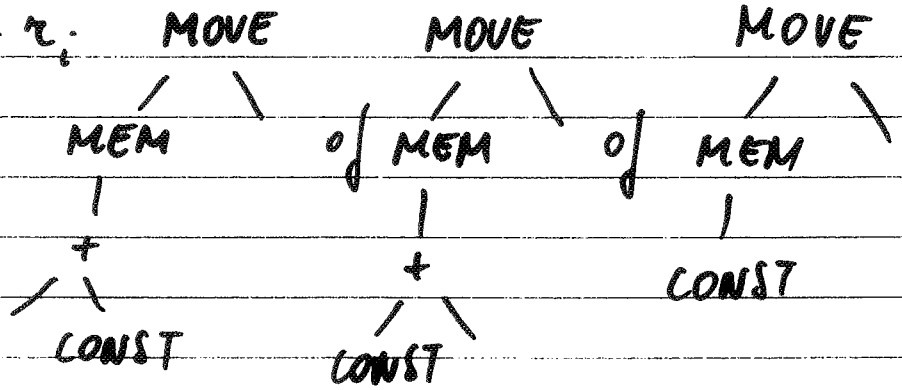


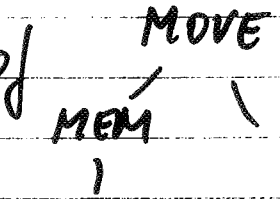
LOAD $r_i \leftarrow M[r_j + c]$



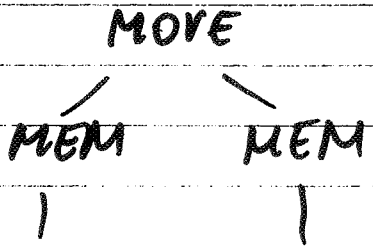
als $r_i = r_0$

STORE $M[r_j + c] \leftarrow r_i$



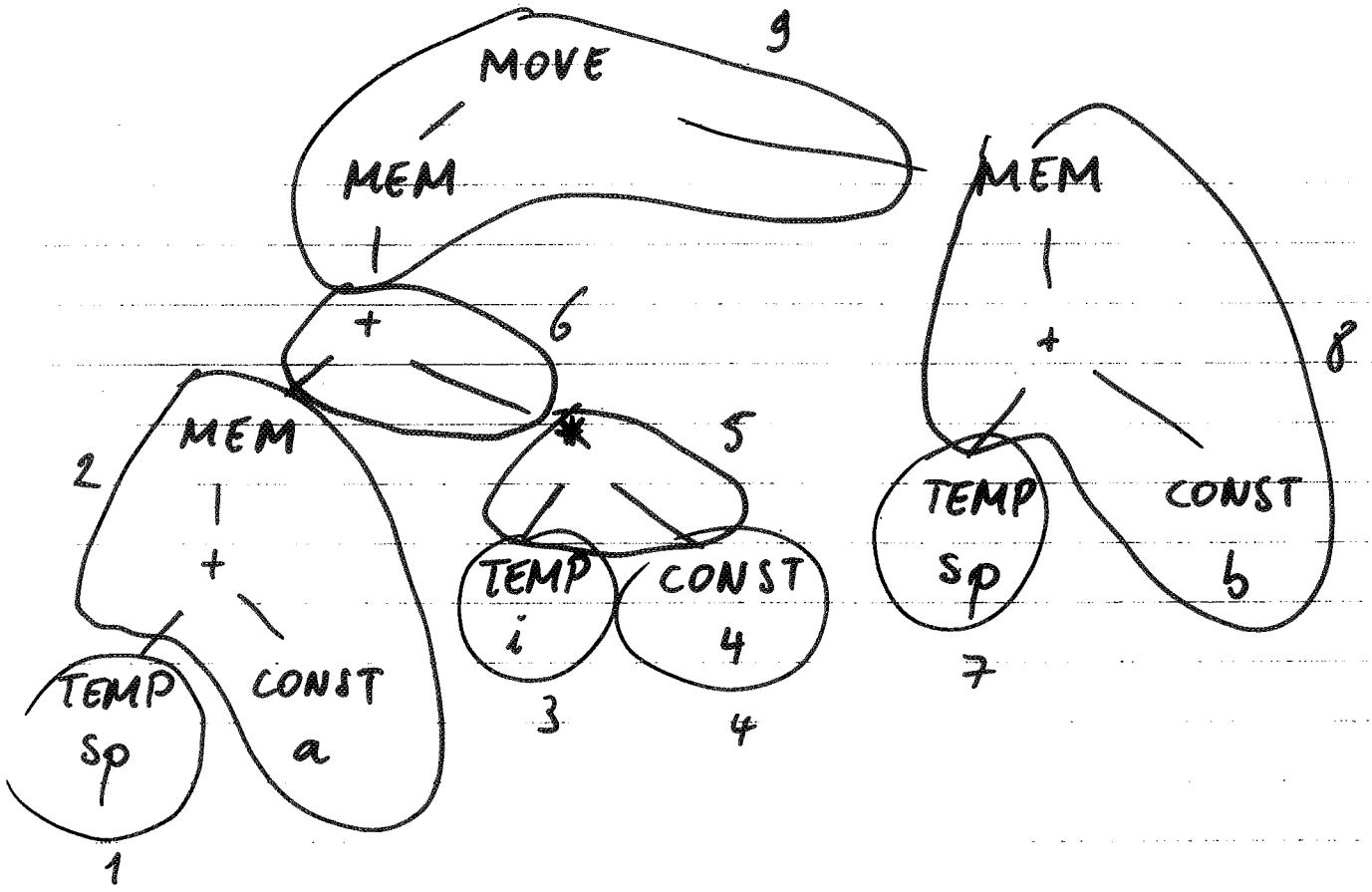


MOVEM $M[r_j] \leftarrow M[r_i]$



Instruction Selection bekijken als Betegeling ("Tiling")

a[i] := b




```
LOAD   r1 ← M[sp+a]
ADDI   r2 ← r0 + 4
MUL    r3 ← ti * r2
ADD    r4 ← r1 + r3
LOAD   r5 ← M[sp + b]
STORE  M[r4] ← r5
```

- gebruik telkens verse tijdelijke registers om tegels te verbinden

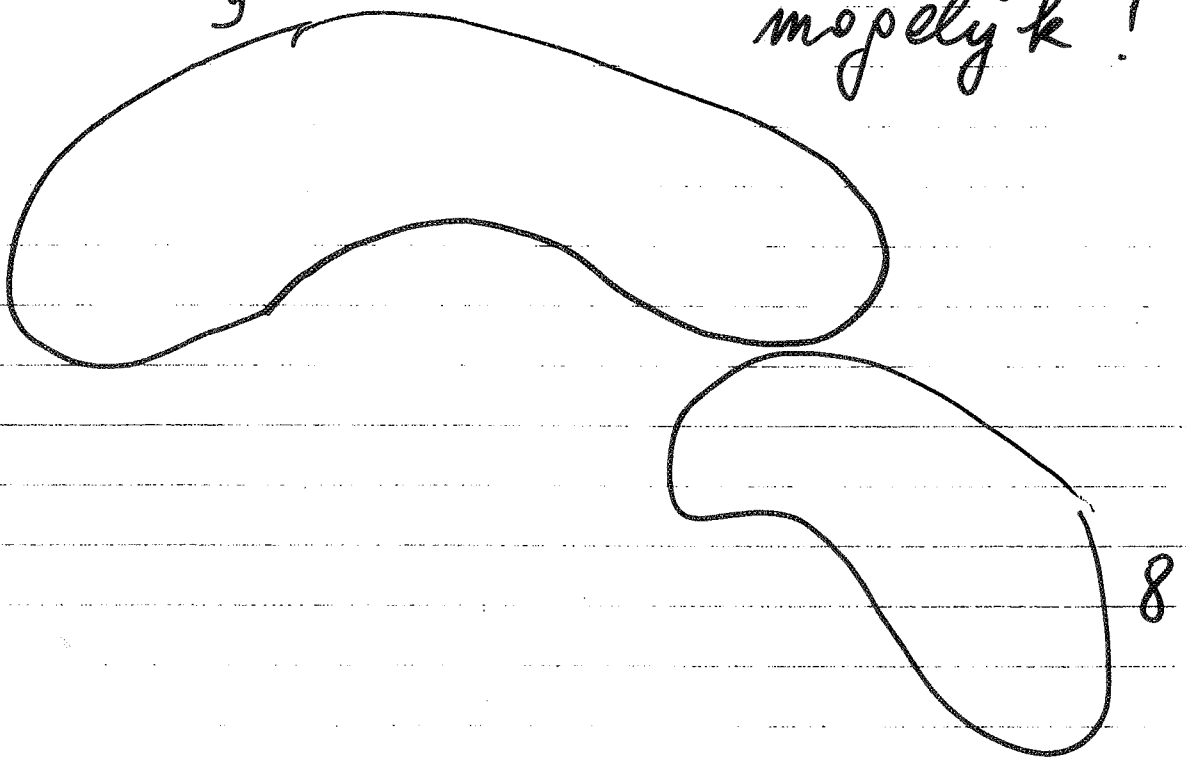
- na registerallocatie zien we dat we eigenlijk maar 2 registers nodig hebben!

$r_3 \mapsto r_2$ ←

$r_4 \mapsto r_1$

$r_5 \mapsto r_2$

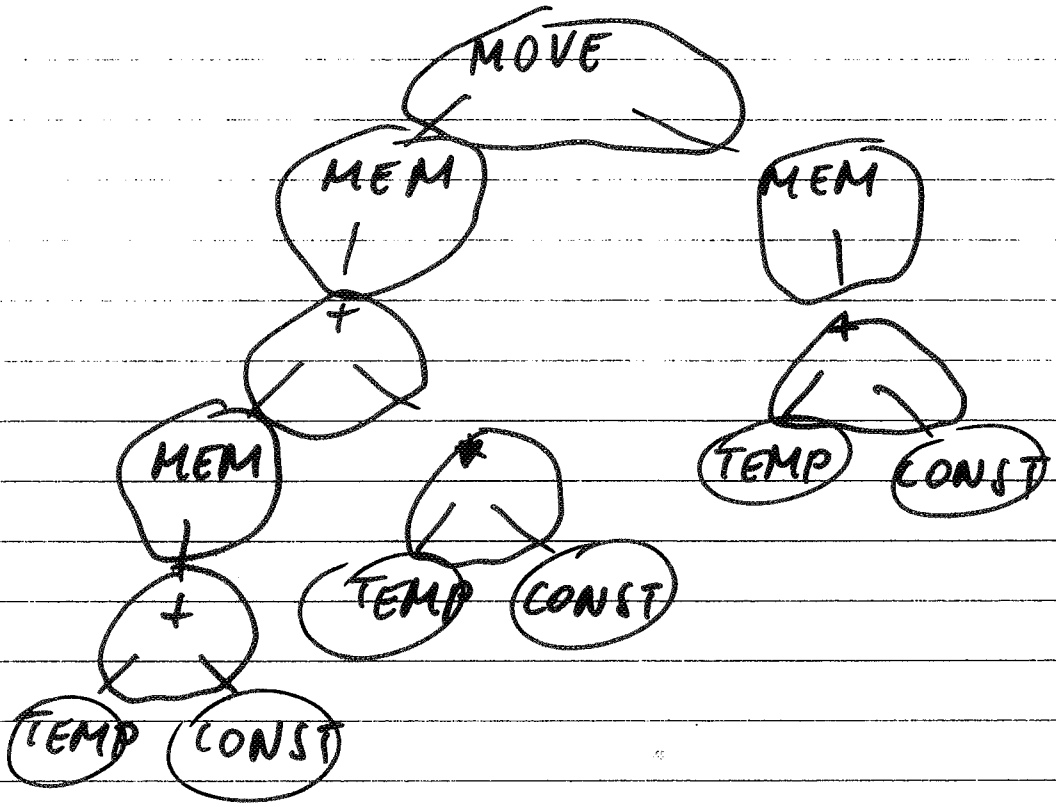
Meerdere betekelingen mogelijk!



ADDI $r_5 \leftarrow sp + b$

MOVEM $M[r_4] \leftarrow M[r_5]$

Altijd triviale betegeling mogelijk:



- Niet optimaal: buit de mogelijkheid om aanpreizende tegels samen te voegen niet uit

  - Indien geen samenvoegen meer mogelijk: "lokaal optimaal" (eerste twee tilings)
  - Meest efficiënte tiling: "globaal optimaal" (eerste efficiënter dan tweede als MOVEM trager is dan STORE)

"Maximal Munch" algoritme voor instruction selection

- Begin bij de root, en selecteer zo groot mogelijke instructie (tegel) die past. ↙ meeste knopen
- Herhaal voor de overblijvende deelbomen

⇒ eerste twee voorbeelden : tweede is maximal munch

- Garandeert lokale optimaliteit
- Globale optimaliteit kan ook bereikt worden op efficiënte manier, mits iets gesofisticeerder algoritme

⚠ Voor dit soort algoritme heb je een goed pattern matching hulpalgoritme nodig [bestaan]

CISC machines

- Hebben veel minder echte registers
  ⇒ probleem voor registerallocatie

- Hebben normaal slechts "2-address" instructies:

  niet $t_1 \leftarrow t_2 + t_3$

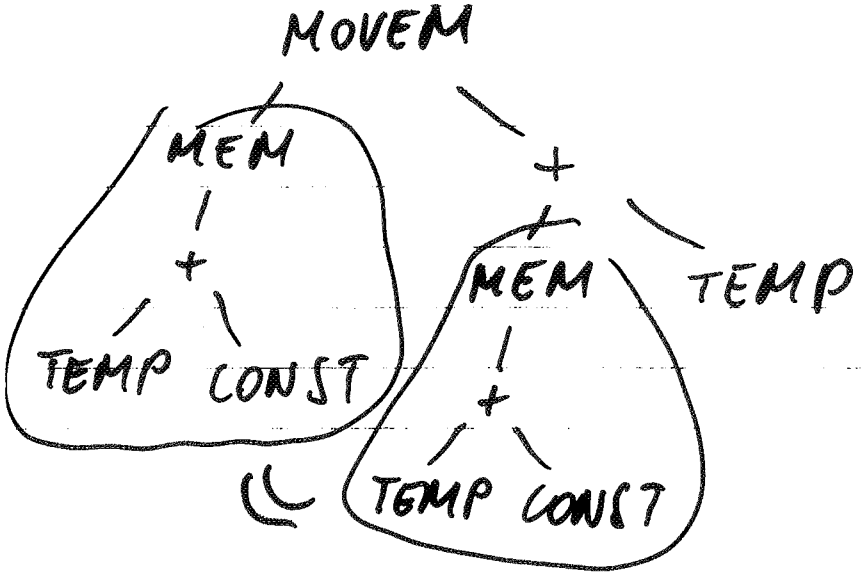
  wel $t_1 \leftarrow t_1 + t_2$

⇒ doe of ze het toch hebben, maar genereer

$t_1 \leftarrow t_2$

$t_1 \leftarrow t_1 + t_3$

- Hebben heel complexe tiles, b.v.
  $M[r_1 + a] \leftarrow M[r_1 + a] + r_2$



⇒ pattern matching ingewikkelder

⇒ compiler hoeft deze tiles niet te gebruiken

vb $MEM[r_1 + a] \leftarrow MEM[r_1 + a] + r_2$

versus

LOAD $r_3 \leftarrow MEM[r_1 + a]$

ADD $r_3 \leftarrow r_3 + r_2$

STORE $MEM[r_1 + a] \leftarrow r_3$

evenveel clockcycles !

- [RC]ISC is "passé"
  - nieuwste architecturen hebben nog allerlei extra features die het onderscheid erg moeilijk maken

(vb. pipelining (intra-processor parallellisme))

Typechecking voor OO programmeertalen

aanvullende nota's bij vak Compilers

academiejaar 2008–09

1 Inleiding

Alle hogere programmeertalen bieden een aantal primitieve datatypes, alsook een aantal primitieve operaties die werken op waarden van deze types. Zo is er in C het primitieve type `int` waarop operaties zoals `+` en `-` gedefinieerd zijn, en het primitieve type `int*` waarop eveneens een operatie `-` gedefinieerd is. Daarbij komt dan nog eens dat deze programmeertalen ons ook toelaten zelf nieuwe datatypes (zoals classes) te definiëren, en ook nieuwe operaties (fields, functies, procedures, methods, ...). Bijvoorbeeld, in Java kunnen we types `A` en `B` definiëren:

```
class A { B c; }
class B { }
```

waardoor we er meteen ook een operatie `x.c` bijkrijgen.

Van elke operatie, of ze nu primitief is of zelfgedefinieerd, is het belangrijk dat je ze enkel kan laten werken op waarden van de juiste types. Inderdaad, als we zouden toelaten dat elke operatie zomaar op eender welke waarde mag werken, opent dat de deur voor een hele hoop mogelijke programmeerfouten. Bijvoorbeeld, de C-operatie `-` op `int`s zou je in principe ook wel kunnen toepassen op `int*`s, maar geeft op de meeste hedendaagse computers een resultaat dat 2 of 4 keer groter is dan wat `-` op `int*`s teruggeeft. Of, in het Java-voorbeeldje van hierboven zou je `x.c` misschien ook wel kunnen toepassen op een `x` van type `B`, maar het resultaat zou onvoorspelbaar zijn.

Door verschillende types netjes van elkaar te onderscheiden; door het juiste type te berekenen voor elke uitdrukking in ons programma; en daardoor te controleren dat elke operatie slechts wordt uitgevoerd op uitdrukkingen van de juiste types waarvoor de operatie bedoeld is, kan de compiler vele programmeerfouten op tijd ontdekken. We noemen dit *static type checking:* "static" omdat het programma nog niet hoeft te lopen opdat we toch al bepaalde type-fouten kunnen opsporen. Sommige programmeertalen doen aan *run-time type checking:* ze houden tijdens het lopen van het programma van elke waarde het type bij, en controleren dat elke operatie slechts wordt uitgevoerd op waarden van de juiste types waarvoor de operatie bedoeld is. Indien dat niet zo is, wordt het programma afgebroken, of wordt een "exception" opgeworpen.

De programmeertalen C en C++ doen aan static type checking, maar niet aan run-time checking. Aangezien "type casts" toelaten het typesysteem te omzeilen,

```
class A { int a; }
class B extends A { int b; }
class C extends B { int c; }
class P
{
  public static void main(String[] args)
  {
    A x = new A();
    A y = new B();
    A z = new C();
    B u = (B) y; /* run-time check op cast:
                    y bevat inderdaad een B-object, OK dus */
    B v = (B) z; /* run-time check op cast:
                    z bevat een C-object, C is subklasse van B, OK dus */
    B w = (B) x; /* run-time check op cast:
                    x bevat een A-object, A is geen subklasse van B,
                    dus java.lang.ClassCastException wordt opgeworpen */
  }
}
```

Figuur 1: Dit Java-programma geraakt wel door de static type checks van de compiler, maar niet door de run-time checks omdat we de ClassCastException, opgeworpen door de laatste cast, niet opvangen. Het analoog programma in C++ zou de onveilige cast toch uitvoeren, waarna we via `w.b` in niet-gealloceerd geheugen kunnen rotzooien en crashen. (In C++ kan je weliswaar met `dynamic_cast` je casts zelf checken in je programma, en dat wordt ook sterk aangeraden.)

kunnen hierdoor crashes optreden. De programmeertaal Java doet aan static type checking, en zal daarenboven ook aan run-time type checking doen voor de problemen die ze niet statisch kan opsporen, met name, type casts. Programma's in run-time gecheckte programmeertalen kunnen nooit écht crashen: het run-time systeem zal ze opvangen. Een voorbeeldje hiervan vindt je in Figuur 1.

Ter vergelijking: de programmeertalen Prolog en Scheme doen helemaal niet aan static type checking, maar wel aan run-time type checking. Zo zijn er in deze talen operaties die enkel op atoms, of enkel op niet-lege lijsten, gedefinieerd zijn, en dit wordt dan ook run-time gecheckt. Ook Smalltalk, de allereerste object-georiënteerde programmeertaal en nog altijd veel gebruikt in sommige middens, doet aan run-time type checking maar niet aan statische. Over het algemeen zijn programmeertalen zonder static type checking erg soepel en lekker om in te programmeren, maar de prijs die je wel betaalt is dat de compiler je totaal niet kan helpen in het opspeuren van programmeerfoutjes die je onvermijdelijk toch zal maken. Deze zullen pas allemaal at run time naar boven komen.

2 Featherweight Java

Featherweight Java (FJ) is een pluimgewicht versie van de programmeertaal Java, en werd gedefiniëerd in een artikel geschreven door de programmeertaal-experten Igarashi, Pierce en Wadler.[1] Hun doel was voornamelijk educatief, namelijk, om het object-georiënteerd type-checking systeem van Java bloot te leggen.

FJ is een "fragment" van Java: elk geldig FJ programma is ook een geldig Java programma. Dit is leuk omdat je dus de werking van je eigen compiler kan controleren door te vergelijken met een echte Java compiler. Een fragment van Java zijnde, is FJ dus wat overblijft van Java nadat we een heleboel constructies eruit hebben gegooid. Voorbeelden van wat eruit is gegooid zijn toekenningsopdrachten; null pointers; arrays; overloading; access control; calls via `super`; hiding; abstract classes; interfaces; exceptions; en inner classes. Voorbeelden van wat er wel nog overblijft zijn classes en subclasses; fields; objectcreatie; methods; `this`; overriding; en recursie.

We kunnen FJ best inleiden aan de hand van een simpel voorbeeldje.

```
class A extends Object {
  A() { super(); }
}
class B extends Object {
  B() { super(); }
}
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

We kunnen onmiddellijk enkele karakteristieken van FJ opmerken:

- We geven altijd de superclass aan, ook als die `Object` is;
- We schrijven altijd de constructor uit, ook als die triviaal is. In FJ hebben alle constructors eenzelfde uniforme stijl. Een constructor moet precies evenveel formele parameters hebben als het aantal velden dat bekend is in de klasse (inclusief superklassen), en deze formele parameters hebben ook

[1] A. Igarashi, B. Pierce, P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

precies dezelfde namen als de velden. We roepen altijd eerst de `super` constructor voor de velden van de superklassen, gevolgd door toekenningen van de velden eigen aan de klasse zelf.

In het voorbeeld hierboven is de superklasse `Object`, die geen velden heeft, daarom wordt de `super` constructor aangeroepen zonder parameters.

FJ heeft geen toekenningsopdrachten, dus constructors zijn de enige plaats waar het `=` teken kan voorkomen.

- We schrijven altijd de receiver (van een field access of een method call) uit, ook als die `this` is.
- De body van elke method bestaat uit één enkel `return`-statement.

Wat de mogelijke uitdrukkingen betreft, er zijn vijf soorten in FJ:

1. Object constructor. Bijvoorbeeld:

   ```
   new Pair(new A(), new B())
   ```

2. Method aanroep. Bijvoorbeeld:

   ```
   new Pair(new A(), new B()).setfst(new B())
   ```

3. Field access. Bijvoorbeeld:

   ```
   new Pair(new A(), new B()).snd
   ```

4. Variabelen. Bijvoorbeeld, de variable `newfst` in de body van de method `setfst`. Aangezien FJ geen toekenningsopdrachten heeft, zijn formele parameters van methods de enige soort variabelen die bestaan.

5. Cast. Bijvoorbeeld:

   ```
   ((Pair) new Pair(new Pair(new A(), new B()), new B()).fst).snd
   ```

Dit rondt de features van FJ af.

3 Static type checking voor FJ

De enige types die we kennen in een FJ-programma zijn classes. (In volledige Java bestaan er ook nog primitieve types zoals `int`, en ook array types.) Static type checking voor FJ bestaat er dus in:

1. de klassedefinities te analyseren, zodat we weten welke types we in ons programma ter beschikking hebben, en ook weten welke de subtypes zijn van elke klasse;

2. voor elke uitdrukking in het programma het type te berekenen; en daarbij te

3. controleren dat een operatie wordt toegepast op uitdrukkingen van de juiste types;

4. tenslotte controleren of het type berekend voor de body van een method wel overeenkomt met het resultaattype gedeclareerd voor die method.

Het controleren van een uitdrukking volgt welbepaalde *type-regels:* voor elke soort uitdrukking (er zijn er zeven in FJ; zie de syntax) is er een welbepaalde type-regel. Deze type-regels zijn precies dezelfde als een standaard Java-compiler zou volgen bij het type-checken van een FJ-programma. Het enige dat er bij FJ anders is dan in standaard Java is de uniforme stijl voor constructors (zie syntax).

We gaan nu dieper in op elk van bovenvermelde stappen.

4 De analyse van de klassedefinities

Het eerste wat de typechecker moet doen is de klassedefinities analyseren, en er volgende informatie uithalen.

- De lijst van gedefinieerde klassen. Vergeet niet dat `Object` er zowiezo altijd bij is. Voor elke klasse mag er slechts één definitie zijn in het programma. Voor `Object` is er geen definitie, en we onderstellen dat deze klasse leeg is (geen velden en geen methods heeft).

- Voor eender welke twee klassen C en D moeten we snel kunnen bepalen of D een *subklasse* is van C, genoteerd als $D \leq C$ en gedefinieerd als volgt:

  1. voor elke klasse C is $C \leq C$;
  2. Als klasse D gedeclareerd is als `class` D `extends` C dan is $D \leq C$. We noemen C de *onmiddellijke superklasse van* D.
  3. Als E de onmiddellijke superklasse is van D, en $E \leq C$, dan ook $D \leq C$.

  Als D een subklasse is van C, noemen we C ook een *superklasse* van D.

  Het volstaat voor elke klasse haar onmiddellijke superklasse bij te houden. Als we dan willen weten of $D \leq C$, kijken we eerst of $D = C$; anders kijken we naar de onmiddellijke superklasse van D; de onmiddellijke superklasse van die klasse; enzovoort, tot we C tegenkomen. Indien we C niet tegenkomen is D geen subklasse van C.

- We moeten ook controleren dat er geen cycles zitten in de inheritance-relatie! Dit kan je doen door vanaf *elke* klasse C de ketting van superklassen af te lopen; je mag *nooit* in C zelf terug belanden, maar je moet altijd uiteindelijk in `Object` belanden.

```
class A { Object a; }
class B extends A { C b; }
class C extends B { A c; }
```

Figuur 2: Illustratie van inheritance van velden.

- Voor elke klasse C hebben we ook de lijst van haar *velden* nodig, gedefinieerd als volgt:

  1. De "eigen" velden van C zijn de velden die rechtstreeks gedeclareerd worden in de definitie van klasse C.
  2. Volgens de welbekende inheritance-regel worden nu alle eigen velden van alle superklassen van C beschouwd als de velden van C.

  Voor elk veld hou je niet alleen de naam bij, maar ook het type (klasse).

  Bijvoorbeeld, voor het programmafragmentje in Figuur 2 hebben we volgende velden voor de verschillende klassen: (dit tabelletje laat voor elke klasse *alle* velden zien, niet enkel de eigen velden)

klasse	A	B	C	Object

velden	`a Object`	`a Object`
		`b C`
			`a Object`
			`b C`
			`c A`
				geen

  Opgelet: In FJ laten we niet toe dat twee verschillende klassen C en D, met $D \leq C$, eigen velden hebben met dezelfde naam.

- We kunnen nu ook controleren dat de constructor van elke klasse voldoet aan de uniforme stijl zoals voorgeschreven door FJ: de lijst van formele parameters van de constructor van klasse C moet precies gelijk zijn aan de volledige lijst van velden van C, met de juiste types. De eigen velden worden toegekend met toekenningsopdrachten van de vorm `this.`f`=`f voor elke f die een naam is van een eigen veld: de overige formele parameters worden doorgeven aan de `super` constructor. Zie de syntax van FJ voor een concreet voorbeeldje van deze stijl.

- Tenslotte moeten we ook voor elke klasse C, en elke method naam m, snel kunnen opzoeken wat het *type* is van m in C. Het type van een method is niet gewoon een klassenaam, maar wel een expressie van de vorm:

  $$C_1, \ldots, C_n \to C_0$$

  waarbij C_1, ..., C_n de types zijn van de formele parameters van m, en C_0 het resultaattype is. We zoeken het type van m in C op als volgt: (in overeenstemming met de bekende inheritance-regel)

```
class A extends Object
{
  C foo(A x, B y)   { ... }
  A bar(C x)   { ... }
}
class B extends A
{
  A bar(C x)   { ... }
}
class C extends B
{
  B bar(A x, Object y)   { ... }
  C nix(B x)   { ... }
}
```

Figuur 3: Illustratie van method types.

1. Als een method met naam m in de definitie van klasse C zelf wordt gedefinieerd, als:

   C_0 `m(`C_1 x_1`,...,`C_n x_n`) { ... }`

   dan is het type van m in C uiteraard $C_1, \ldots, C_n \rightarrow C_0$.
2. Als geen method met naam m in de definitie van klasse C wordt gedefinieerd, zij dan E de onmiddellijke superklasse van C. We zoeken dan het type van m in E op.
3. Als we, de ketting van superklasses volgend, nergens een definitie van een methode met naam m tegenkomen, zeggen we dat het type van m in C ongedefinieerd is.

Bijvoorbeeld, voor het programmafragmentje uit Figuur 3:

het type van	`foo`	in	`A`	is	$\texttt{A}, \texttt{B} \rightarrow \texttt{C}$
	`bar`		`A`		$\texttt{C} \rightarrow \texttt{A}$
	`foo`		`B`		$\texttt{A}, \texttt{B} \rightarrow \texttt{C}$
	`bar`		`B`		$\texttt{C} \rightarrow \texttt{A}$
	`foo`		`C`		$\texttt{A}, \texttt{B} \rightarrow \texttt{C}$
	`bar`		`C`		$\texttt{A}, \texttt{Object} \rightarrow \texttt{B}$
	`nix`		`C`		$\texttt{B} \rightarrow \texttt{C}$

5 Type-contexten

Uitdrukkingen, met name in de body van een method, worden getypecheckt in een bepaalde *context*. Deze context kan je wiskundig bekijken als een functie, gedefinieerd op de formele parameters van de method in kwestie, die aan elke

parameter het type toekent zoals gedeclareerd in de hoofding van de method. Je kan contexten gemakkelijk implementeren in C++ als datastructuren van type `map<string,string>`. Zulke datastructuren worden ook wel eens *symbol tables* genoemd.

De type-context is nodig om een context te geven aan de variabelenamen die voorkomen in een uitdrukking. Immers, wat is volgens jou het type van de uitdrukking '`x`'? Zo gesteld is die vraag onzinnig, we weten immers helemaal niks af van de variabele `x`. De vraag wordt zinvoller wanneer we de uitdrukking bekijken in de context van een method body:

```
class A
{
  C foo(C x, D y)
  {
    return x;
  }
}
```

Het type van de uitdrukking '`x`' in bovenstaande method body is nu duidelijk `C`, wat we halen uit de lijst van formele parameters van de method. Een uitdrukking in de body van een method zal altijd getypeerd worden in een context bestaande uit de formele parameters van de method (met hun types), plus de speciale variabele `this`, die als type de klasse heeft waaruit de method body komt. Voor bovenstaande method body zal de context dus de volgende zijn:

$$\texttt{x} \mapsto \texttt{C} \quad \texttt{y} \mapsto \texttt{D} \quad \texttt{this} \mapsto \texttt{A}$$

We zullen contexten noteren met de letter Γ.

6 De typeregel voor methods

Een programma wordt getypecheckt door elke method in elke klasse te typecheken. Beschouw dus een klasse C, en in deze klasse de definitie van een method met naam m:

$$C_0\ m\texttt{(}C_1\ x_1\texttt{,...,}C_n\ x_n\texttt{) \{ return } u\texttt{; \}}$$

1. Eerst en vooral controleren we of de namen C_0, C_1, ..., C_n namen zijn van gekende klassen in ons programma; indien niet is deze methoddefinitie al fout.

2. Dan controleren we of deze definitie een correcte "overriding" is van een definitie van m in een superklasse. Daartoe kijken we naar E, de onmiddellijke superklasse van C. Op de wijze uitgelegd in paragraaf 4 bepalen we "het type van m in E". Er zijn dan twee mogelijkheden:

   - Het type van m in E bestaat en is $D_1, \ldots, D_k \to D_0$. Dan moet $k = n$ en moet $D_i = C_i$ voor $i = 0, 1, \ldots, n$. (We laten in FJ immers geen overloading toe, enkel pure overriding.)

- Het type van m in E is ongedefinieerd. Dan is er geen sprake van overriding en is alles in orde.

Bijvoorbeeld, in Figuur 3 zien we dat de overriding van `bar` in `C` fout is. De overriding van `bar` in `B` is wel goed want komt qua type perfect overeen met het type van `bar` in `A`.

3. Nu typechecken we de method body. We creëren volgende context:

$$x_1 \mapsto C_1,\ \ldots,\ x_n \mapsto C_n,\ \texttt{this} \mapsto C$$

In deze context bepalen we nu het type van de uitdrukking u. Hoe dit gebeurt zien we in de volgende paragraaf. Zij D het resulterende type voor u.

4. Nu moet gelden dat $D \leq C_0$. Indien niet, is de method body fout.

We merken nog op dat we ook printopdrachten hadden toegevoegd aan FJ, maar daaraan hoeft niets getypecheckt te worden.

7 De typeregels voor uitdrukkingen

We beschouwen nu de zeven mogelijke vormen van een uitdrukking u, en we typeren ze telkens in een willekeurige gegeven context Γ. Het kan ook zijn dat u *ongetypeerd* is in context Γ: dat is dan een type-fout.

- Stel dat u gewoon een variabelenaam x is.

  Het type van u in context Γ is dan simpelweg het type dat we vinden voor x in Γ. Indien x niet voorkomt in Γ, is u ongetypeerd in context Γ.

- Stel dat u de uitdrukking '`this`' is. We behandelen dit precies zoals het vorige geval, waarbij we `this` dus als een speciale variabelenaam beschouwen.

- Stel dat u van volgende vorm is:

$$u_0 \,.\, f$$

  waarbij u_0 zelf een uitdrukking is, en f een naam.

  1. Eerst en vooral typeren we recursief u_0 in context Γ; zij C_0 het type van u_0 in context Γ.
  2. Nu controleren we, op de wijze uitgelegd in paragraaf 4, of f een veld is van klasse C_0. Zo neen, is u ongetypeerd in context Γ. Zo ja, zij C het type van f.
  3. Het type van u in context Γ is dan C.

- Stel dat u van volgende vorm is:

$$u_0\texttt{.}m\texttt{(}u_1\texttt{,...,}u_n\texttt{)}$$

  waarbij u_0, u_1, ..., u_n zelf uitdrukkingen zijn, en m een naam.

  1. Eerst en vooral typeren we recursief de uitdrukkingen u_0, u_1, ..., u_n in context Γ; zij C_0, D_1, ..., D_n de resulterende types.
  2. Nu zoeken we het type op van method m in C_0, op de wijze uitgelegd in paragraaf 4. Zij $C_1, \ldots, C_k \to C$ dat type. Indien m ongedefinieerd blijkt te zijn in C_0, dan is u ongetypeerd.
  3. Nu controleren we dat $k = n$, en dat $D_i \leq C_i$ voor $i = 1, \ldots, n$. Indien één van deze controles faalt, is u ongetypeerd.
  4. Het type van u in context Γ is dan C.

- Stel dat u van volgende vorm is:

$$\texttt{new } C\texttt{(}u_1\texttt{,...,}u_n\texttt{)}$$

  waarbij u_1, ..., u_n zelf uitdrukkingen zijn, en C een naam.

  1. Eerst en vooral controleren we of C wel een bekende klassenaam is voor ons programma. Indien niet, is u uiteraard ongetypeerd. Indien wel, bekijken we de hoofding van de constructor voor C:

$$C\texttt{(}C_1\ x_1\texttt{,...,}C_k\ x_k\texttt{) \{ ... \}}$$

     We zien dat de lijst types van de formele parameters gelijk is aan $C_1, \ldots, C_k$.
  2. Er moet nu al zeker gelden dat $k = n$; anders is u ongetypeerd.
  3. Ook typeren we, recursief, de uitdrukkingen u_1, ..., u_n in context Γ; zij D_1, ... D_n de resulterende types.
  4. Nu controleren we of $D_i \leq C_i$ voor $i = 1, \ldots, n$; indien dit niet zo is dan is u ongetypeerd.
  5. Het type van u in context Γ is dan C.

- Stel dat u van volgende vorm is:

$$\texttt{(}C\texttt{)}\ u_0$$

  waarbij u_0 zelf een uitdrukking is, en C een naam.

  1. Eerst en vooral controleren we of C de naam is van een gekende klasse in ons programma; indien niet, is u uiteraard ongetypeerd.

2. Ook typeren we recursief de uitdrukking u_0 in context Γ; zij D het type van u_0 in context Γ.
3. Nu controleren we dat $C \leq D$ of $D \leq C$; indien geen van beiden het geval is, is er een type-fout en is u ongetypeerd.
4. Het type van u in context Γ is dan C.

- Tenslotte, als u van de vorm (u_0) is dan is het type van u gelijk aan het type van u_0.