Compilerverifikation

Martin Strecker

19.6.2001

A priori Verifikation

- VerifiCard
- Was ist ein Compiler? Was gibt es zu verifizieren?
- Geschichtlicher Überblick

A posteriori Verifikation

- Hintergrund
- Compiler-Optimierungen

Testverfahren

- Manuelle Tests
- Ada Conformance Testsuite
- Automatisierte Tests

Verificard

Ziele

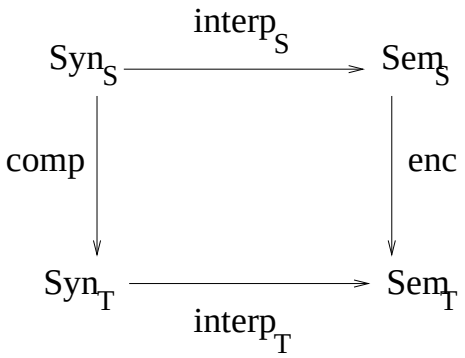
- Entwicklung sicherer Anwendungen in Java
- Formalisierung von Aspekten von Java
  - Operationelle und axiomatische Semantik von Java
  - Formalisierung der JVM (Instruktionen, Bytecode-Verifier)
  - Compiler
- Einschränkung auf JavaCard

Organisatorischer Rahmen

- EU-finanziertes Projekt (www.verificard.org)
- Laufzeit: Januar 2001 - März 2003
- Univ. Partner und Smartcard-Hersteller

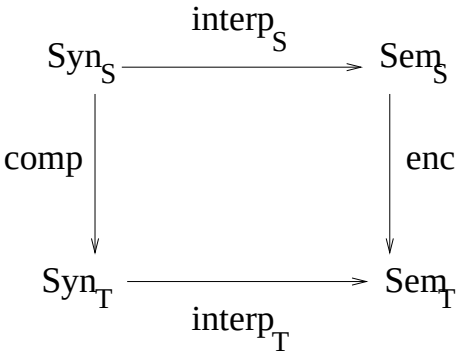
Compilerverifikation: Worum geht es? (1)

Compiler-Korrektheit - denotationell

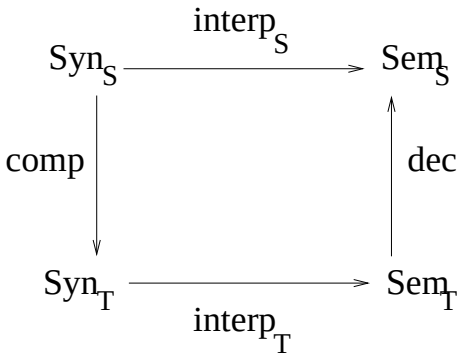


Compilerverifikation: Worum geht es? (1)

Compiler-Korrektheit - denotationell

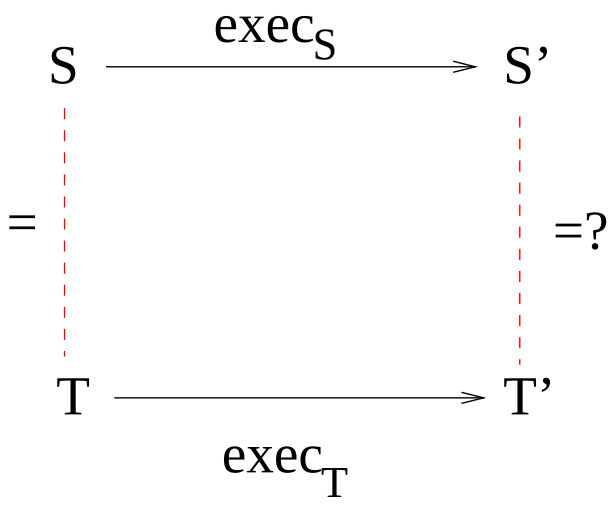


?



Compilerverifikation: Worum geht es? (2)

Compiler-Korrektheit - operationell

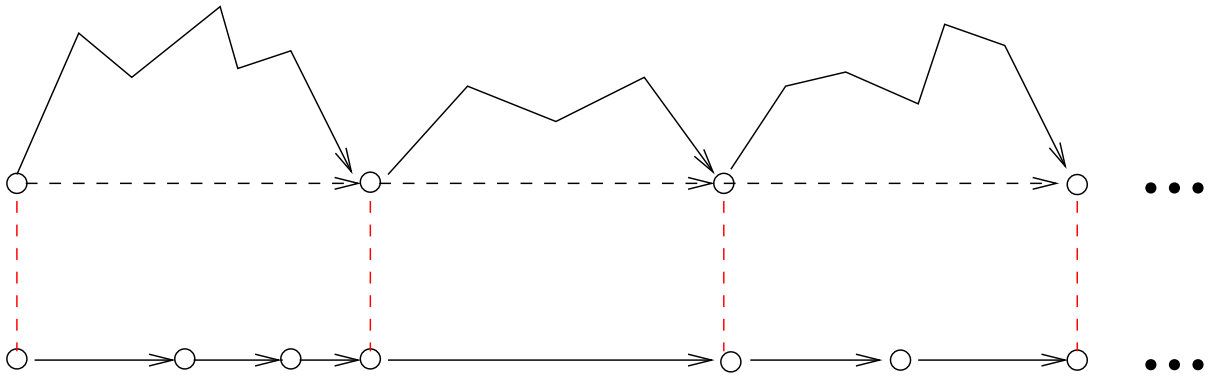


Big-step Semantik

Berechnen von Teilausdrücken in einem Schritt

Typische Regel:

$$\frac{eval(b,\sigma_0) = \mathsf{T} \quad (\sigma_0, c) \to \sigma_1 \quad (\sigma_1, \text{while } b \text{ do } c) \to \sigma_2}{(\sigma_0, \text{while } b \text{ do } c) \to \sigma_2} \qquad \frac{eval(b,\sigma_0) = \mathsf{F}}{(\sigma_0, \text{while } b \text{ do } c) \to \sigma_0}$$



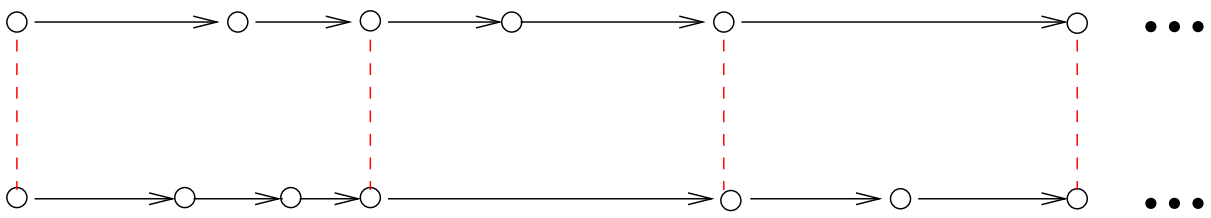
Schwierigkeit: Korrektheitsaussage bei Nichtterminierung?

Small-step Semantik

Auffalten von Konstrukten

Typische Regel:

$$\frac{eval(b,\sigma) = \mathsf{T}}{(\sigma, \text{while } b \text{ do } c) \longrightarrow (\sigma, (c; \text{while } b \text{ do } c))} \qquad \frac{eval(b,\sigma) = \mathsf{F}}{(\sigma, \text{while } b \text{ do } c) \longrightarrow (\sigma, \text{skip})}$$



Schwierigkeit: Korrespondenz zwischen Quell- und Zielzuständen

Vorgehen in Verificard (1)

Vorgaben:

- Quellsprache: Big-step Semantik
- Zielsprache (Instruktionen einer Stackmaschine): Small-step Semantik

```
exec_instr (Load idx) G hp stk vars Cl sig pc frs =
      (None, hp, ((vars ! idx) # stk, vars, Cl, sig, pc+1)#frs)
```

- Keine signifikante Datenverfeinerung
  - Zustände haben identische Struktur

⇒ Keine große Distanz zw. Quell- und Zielsprache
⇒ Compilation in einem Schritt

```
mkExpr jmb ( {cn}e..fn ) = mkExpr jmb e @ [Getfield fn cn]
```

Vorgehen in Verificard (2)

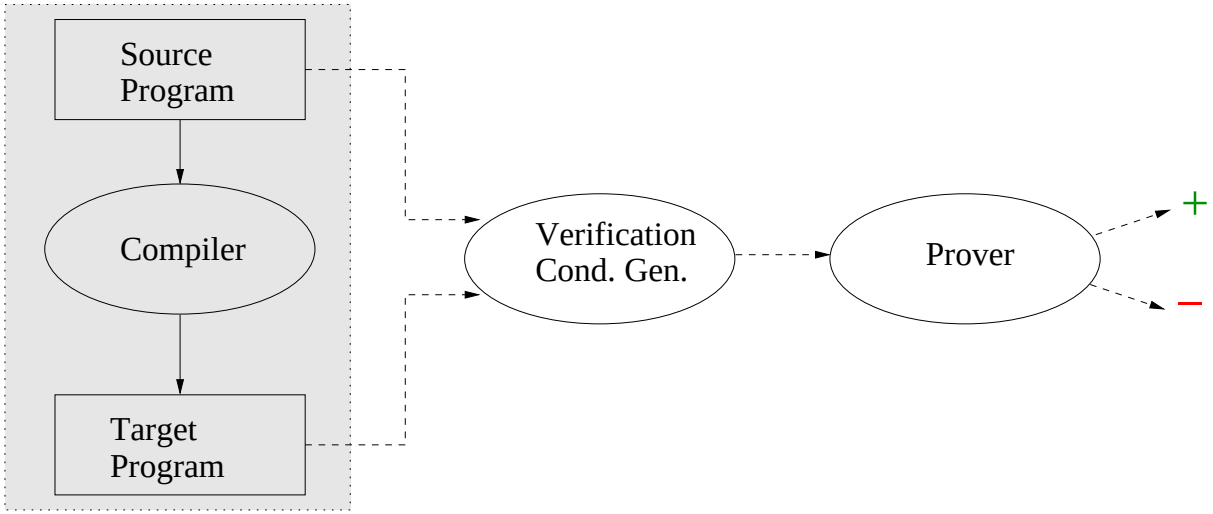
Mechanisierung in Isabelle:

- Umfangreiche Sprachformalisierung
- Homogene Struktur der Beweise
- Symbolische Auswertung durch Simplifier
- Extraktion eines lauffähigen Compilers
  - Verläßlichkeit von Extraktion und ML-Compiler?

Historisches ...

Autoren	*Jahr*	*Sprache*	*Formalismus*	*Mechanisierung*
McCarthy & Painter	1967	Ausdrücke		-
Milner & Weyhrauch	1972	Algol	denot. Sem.	LCF
Polak	1979	Pascal	denot. Sem.	Stanford Verifier
Moore et al.	1989	Gypsy / Piton	?	Boyer / Moore
Sampaio & Hoare	1993	Imperativ	algebraisch	OBJ3
Oliva et al.	1995	Scheme	denot. / oper. Sem.	-
Dold et al.	2000	Lisp	oper. Sem.	PVS
Börger et al.	2001	Java	ASMs	-

Singuläre Verifikation: Idee



- Keine Verifikation des gesamten Compilers

. . . *sondern nur*

- Verifikation individueller Durchläufe

Singuläre Verifikation: Hintergrund

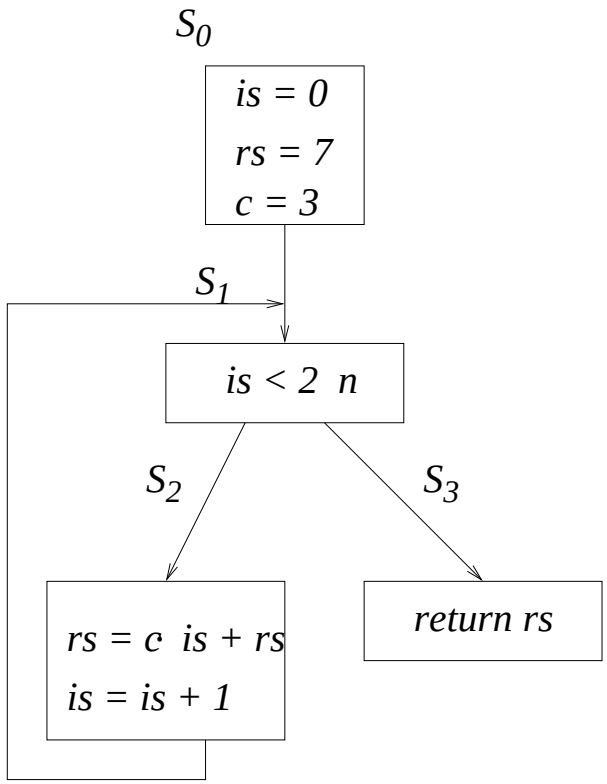
Idee

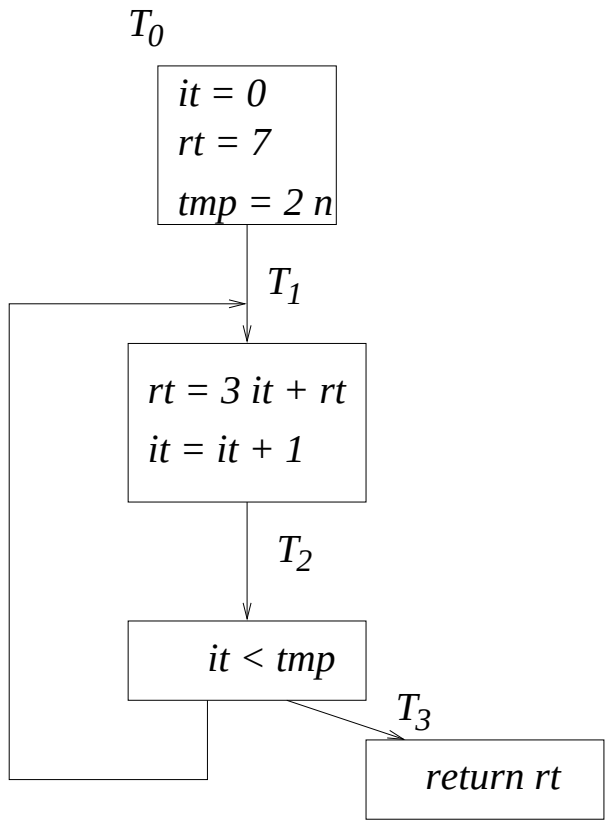
- "Result checking" [Blum & Wasserman]
  - Anstelle von Beweis $P(f)$
  - verifiziere $P'(x, f(x))$ für konkrete x
- Bsp.: Faktorisierungsalgorithmus $f(x) = (y_1, y_2)$
  - $P'(x, (y_1, y_2))$ ist $y_1 \times y_2 = x$
- Annahme: P' leichter zu beweisen als P

Verwendung bei

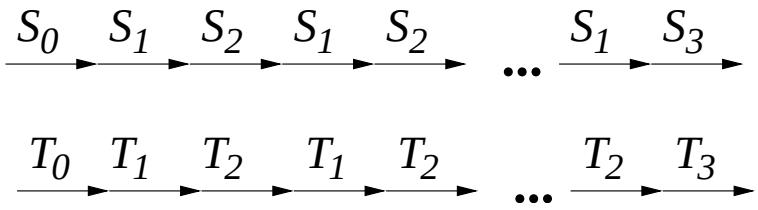
- Compiler-Optimierungen
- Erhalt gewisser Eigenschaften (Typsicherheit)
- Extrem einfach strukturierten Programmen

Singuläre Verifikation: Compiler-Optimierungen (1)





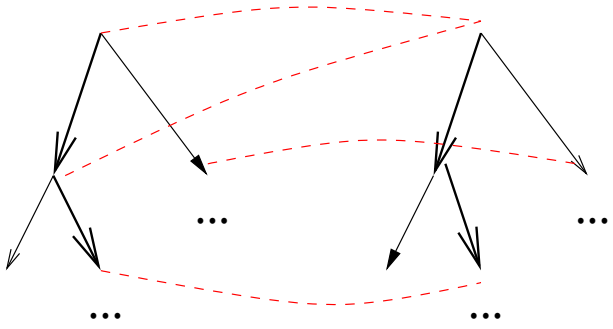
Für $n \geq 1$:



Singuläre Verifikation: Compiler-Optimierungen (2)

Zeige: gegen Ende der Ausführung ist $rs = rt$.

Def.: Unter der Invariante I_k *korrespondieren* S_i und T_j ($I_k \vdash S_i \simeq T_j$), wenn $I_k \vdash is = it \wedge rs = rt$



Vorgehen: Zeige: $\simeq$ ist Bisimulation, d.h.

- $I_0 \vdash S_0 \simeq T_0$
- Es gelte $I_k \vdash S_i \simeq T_j$.
  - Wenn $I_k \vdash S_i \rightsquigarrow S_i'$,
    dann existieren I_k', T_j', so daß
    $I_k' \vdash S_i' \simeq T_j'$
    und $I_k \vdash T_j \rightsquigarrow T_j'$ und I_k' ist erfüllt
  - ... und umgekehrt
- Endzustände korrespondieren

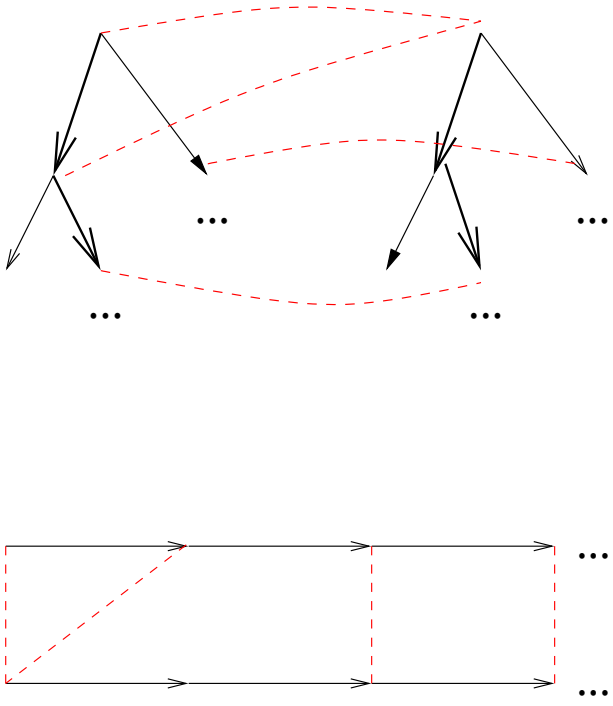
Singuläre Verifikation: Compiler-Optimierungen (2)

Zeige: gegen Ende der Ausführung ist $rs = rt$.

Def.: Unter der Invariante I_k *korrespondieren* S_i und T_j ($I_k \vdash S_i \simeq T_j$), wenn $I_k \vdash is = it \wedge rs = rt$

Vorgehen: Zeige: $\simeq$ ist Bisimulation, d.h.

- $I_0 \vdash S_0 \simeq T_0$
- Es gelte $I_k \vdash S_i \simeq T_j$.
  - Wenn $I_k \vdash S_i \rightsquigarrow S_i'$,
    dann existieren I_k', T_j', so daß
    $I_k' \vdash S_i' \simeq T_j'$
    und $I_k \vdash T_j \rightsquigarrow T_j'$ und I_k' ist erfüllt
  - . . . und umgekehrt
- Endzustände korrespondieren



Singuläre Verifikation: Compiler-Optimierungen (3)

Im **Beispiel** gilt:

- $n \geq 1, is = it, rs = rt \vdash S_0 \simeq T_0$
- $n \geq 1, c = 3, t = 2n, is = it = 0, rs = rt = 7 \vdash S_1 \simeq T_1$
- $n \geq 1, c = 3, t = 2n, is = it, rs = rt \vdash S_2 \simeq T_1$
- $\cdots \vdash S_1 \simeq T_2$
- $\cdots \vdash S_3 \simeq T_3$

Korrespondenz-Erhaltung:

- von $S_1 \simeq T_1$ mit $n \geq 1, is = 0$ erreichbar: $S_2 \simeq T_1$
- von $S_2 \simeq T_1$ mit $is = it, rs = rt$ erreichbar: $S_1 \simeq T_2$
- von $S_1 \simeq T_2$ mit $is = it, rs = rt, c = 3, t = 2n$ erreichbar:
  - $S_2 \simeq T_1$ (für $is < 2n$)
  - $S_3 \simeq T_3$ (für $is \geq 2n$)

Singuläre Verifikation – experimentell

Problem: Wie erhält man Beziehung $\simeq$?

- Annotation durch Compiler [Rinard]
  - Implementierung?
- Heuristische Inferenz [Necula]:
  - Untersuche Optimierungen des gcc
  - Vergleiche RTL-Dumps nach jeder Optimierung

Optimierungen des gcc

- Implementierung: in OCAML (10.000 Zeilen Code)
  Glaubwürdiger als gcc?
- Erfolgreich bei der Isolierung eines (vorher bekannten) gcc-Fehlers
- aber: bis zu 10 % "falscher Alarm"
  Vollautomatisch ?
- Übersetzung wird bis zu Faktor 4 verlangsamt

Singuläre Verifikation: Spezialfälle und Varianten

Zur Erinnerung: Zur Verifikation von Compiler f:

- Statt Nachweis von $P(f)$
- zeige $P'(x, f(x))$ für konkretes x

Proof carrying code: Wenn Beweis $P'(x, f(x))$ schwer zu führen ist...

- lasse Beweis prf vom Compiler generieren
- verifiziere $\vdash prf : P'(x, f(x))$

Bytecode-Verifikation:

- Wenn nur Eigenschaften von $f(x)$ interessieren, z.B.
  - Typsicherheit
  - keine Zugriffsverletzungen
- ... weise nur $P'(f(x))$ nach

Tests – manuell erstellt

- (Immer noch) Standard-Vorgehen bei Compiler-Tests
- Grundlage für Conformance-Tests von Compilern z.B. für
  - Fortran
  - Cobol
  - Pascal
  - Ada

Ada Conformance (1)

Historie von Ada:

- Entwickelt im Auftrag des US DoD
- Ziel: *einheitliche* Software-Platform
  $\leadsto$ Ada als "trademark"
- Standardisierung der *Sprache*: 1983 und 1995 (ISO 8652)
- Standardisierung der *Testprozedur* (ISO 18009)
- Festlegung einer *Testsuite* durch Ada Conformity Assessment Authority
  $\leadsto$ Ada als "certification mark"

Ada Conformance (2)

Ada Conformity Assessment Test Suite (ACATS)

- Black-Box Tests
- Ca. 3600 Testfälle, annotiert mit Testresultat
- Struktur:
  - Korrektes Compile / Link-Verhalten
  - Zurückweisung inkorrekter Programme
  - Korrekte Ausführung
  - Korrekte Verwendung von Bibliotheksfunktionen
- Inhalt:
  - Kaum Prüfung elementarer Eigenschaften
  - Stark spezialisiert:

    "Check that the actual parameter corresponding to a formal parameter of mode in out or out must denote a variable; in particular, that it may not be a dereference of an access-to-constant value. Check for the cases where the value is of a generic formal access-to-constant type, or of a non-formal access-to-constant type declared within a formal package."
  - *Keine* Performance / Stress-Tests

Conformance-Testsuites: Einschätzung

Keine Dialektbildung

Entdeckung von Inkonsistenzen während der Sprachdefinition

Conformance-Testsuites: Einschätzung

Keine Dialektbildung

Entdeckung von Inkonsistenzen während der Sprachdefinition

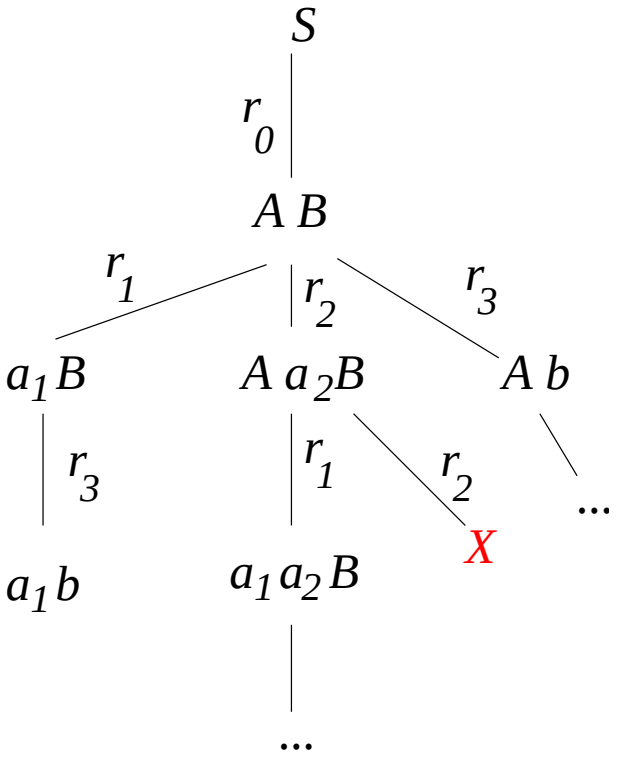
"Conformity to the standard in no way implies that a compiler is useful."

"As it turned out [..], more effort had to be spent on passing the tests than on improving performance or on removing bugs that were not detected by the tests"

[Goodenough 86]

Tests – automatisch generiert (1)

$r_0 : S \rightarrow A\ B$
$r_1 : A \rightarrow a_1$
$r_2 : A \rightarrow A\ a_2$
$r_3 : B \rightarrow b$

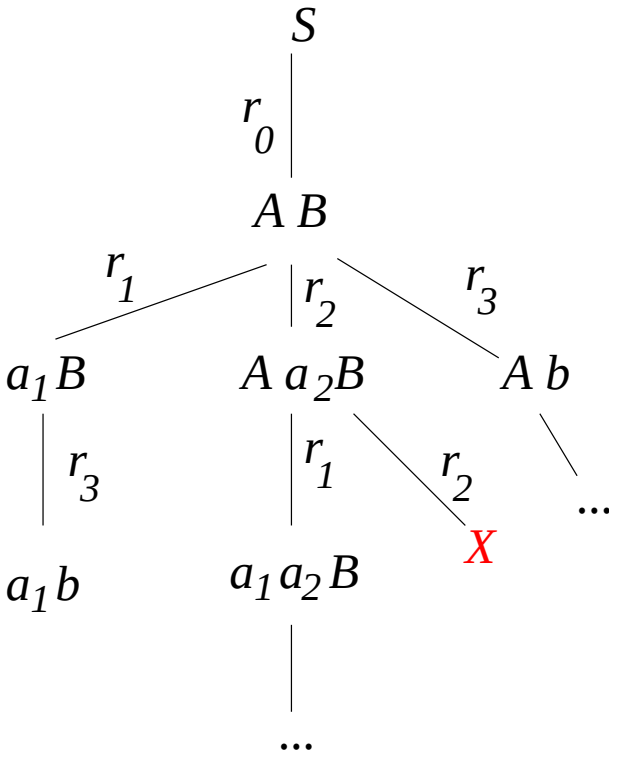


Meistens beschränkt auf Test des Parsers:

- Generierung von Programmtexten anhand der Grammatik
- Einschränkung der Testmenge durch Heuristiken
- Generierte Programmtexte syntaktisch korrekt (inkorrekt) per Konstruktion

Tests – automatisch generiert (1)

$r_0 : S \rightarrow A\ B$
$r_1 : A \rightarrow a_1$
$r_2 : A \rightarrow A\ a_2$
$r_3 : B \rightarrow b$



Meistens beschränkt auf Test des Parsers:

- Generierung von Programmtexten anhand der Grammatik
- Einschränkung der Testmenge durch Heuristiken
- Generierte Programmtexte syntaktisch korrekt (inkorrekt) per Konstruktion

Problem: Wie beschreibt man Semantik von Programmen?

Tests – automatisch generiert (2)

- Annotation von Produktionen mit Scheme-Code [Sirer & Bershad]
- Generierung von Programmen wie vorher

STMTS	$\Rightarrow$	STMT_0 STMTS_1		$S_1 \circ S_0$
STMTS	$\Rightarrow$	nil	λ t.	t
STMT	$\Rightarrow$	iconst i	λ t.	(cons i t)
STMT	$\Rightarrow$	iadd	λ t.	(cons (+ (car t) (cadr t)) (cddr t))
STMT	$\Rightarrow$	ifeq LN_1;	λ t.	(if (equal (car t) 0)
		LN_0: STMTS_0; goto LN_2		(S_2 (S_0 (cdr t)))
		LN_1: STMTS_1		(S_2 (S_1 (cdr t))))
		LN_2: STMTS_2		