Kuratierte Inhalte aus docs/

Die folgenden Kapitel integrieren die bestehenden Inhalte aus dem Ordner `docs/` und ordnen sie in die Gesamtdokumentation ein.

- Architektur
- Sprachumfang (EBNF)
- VM-Instruction-Set
- Traceability-Matrix
- Qualitaet

Architektur (kuratiert)

Dieses Kapitel stellt die Architektur von TinyPl0 zusammengefasst dar und bindet die originalen Detailinformationen aus `docs/ARCHITECTURE.md` ein.

Zusammenfassung

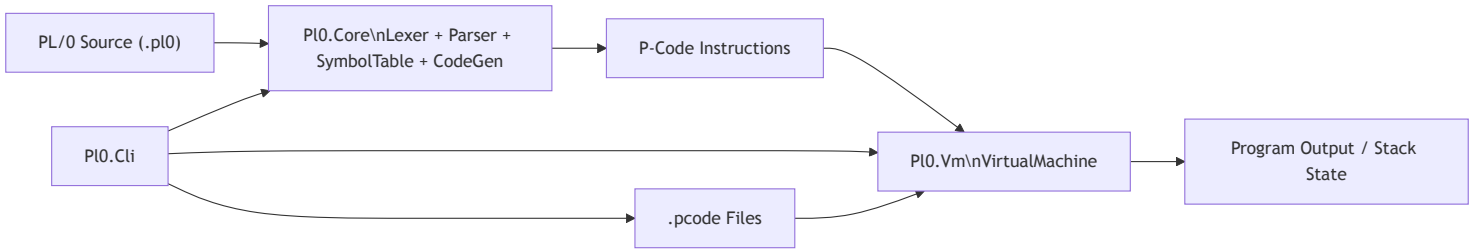
- Drei Projekte: CLI, Core, VM.
- Compiler-Pipeline: Lexer -> Parser -> Codegenerator.
- VM fuehrt den P-Code aus und kapselt I/O.

Originalinhalt

TinyPl0 Architektur

Überblick

TinyPl0 ist in drei Laufzeitmodule getrennt:



Module

- `Pl0.Core`: Sprachverarbeitung und Codegenerierung.
- `Pl0.Vm`: Stackbasierte P-Code-Ausführung inklusive I/O-Adapter.
- `Pl0.Cli`: Bedienoberfläche (`compile`, `run`, `run-pcode`) und Dateifluss.

Datenfluss

1. `.pl0` wird im Core lexikalisch analysiert.
2. Parser erzeugt aus Tokens P-Code-Instruktionen.
3. CLI kann P-Code als `.pcode` speichern oder direkt ausführen.
4. VM interpretiert Instruktionen deterministisch auf einem Integer-Stack.

Pascal -> C# Mapping (Kompakt)

Pascal-Referenz	C#-Implementierung
`getsym/getch`	`Pl0Lexer`

Pascal-Referenz	C#-Implementierung
`block/statement/condition/expression`	`Pl0Parser`
`enter/position/table`	`SymbolTable` + `SymbolEntry`
`gen`	`Pl0Parser.Emit`
`interpret`	`VirtualMachine.Run`
`base(l)`	`VirtualMachine.ResolveBase`
`PrintUsage`	`CliHelpPrinter`

Dialekte

- `Classic`: ohne `?`/`!`, nahe am Pascal-Vorbild.
- `Extended`: mit `? ident` und `! expression`.

PL/0 Syntax (kuratiert)

Dieses Kapitel enthaelt die konsolidierte EBNF fuer PL/0 und dient als Referenz.

Originalinhalt

TinyPl0 Sprachumfang und EBNF

Dialekte

- `classic`: orientiert am historischen PL/0 ohne `?` und `!`.
- `extended`: enthält zusätzlich Eingabe `? ident` und Ausgabe `! expression`.

Konsolidierte EBNF

```
program    = block "." ;

block      = [ "const" ident "=" number { "," ident "=" number } ";" ]
             [ "var" ident { "," ident } ";" ]
             { "procedure" ident ";" block ";" }
             statement ;

statement  = [ ident ":=" expression
             | "call" ident
             | "?" ident
             | "!" expression
             | "begin" statement { ";" statement } "end"
             | "if" condition "then" statement
             | "while" condition "do" statement ] ;

condition  = "odd" expression
           | expression relop expression ;

relop      = "=" | "#" | "<" | "<=" | ">" | ">=" | "[" | "]" ;

expression = [ "+" | "-" ] term { ( "+" | "-" ) term } ;

term       = factor { ( "*" | "/" ) factor } ;

factor     = ident | number | "(" expression ")" ;
```

Hinweise zur Kompatibilität

- `[` und `]` werden als historische Relationen fuer `<=` und `>=` akzeptiert.

- `?` und `!` sind im `classic`-Modus absichtlich nicht erlaubt.
- Der Sprachkern bleibt bewusst klein: Integer, keine Parameter, keine Rueckgabewerte.

Referenzen

- `/Users/thorstenhindermann/Codex/TinyPl0/PL0.md`
- `/Users/thorstenhindermann/Codex/TinyPl0/Pflichtenheft_PL0_CSharp_DotNet10.md`

VM-Instruction-Set (kuratiert)

Dieses Kapitel bindet das Instruktionsset der VM ein und ergaenzt es um Hinweise zur Nutzung im Unterricht.

Hinweise

- Die P-Code-Instruktionen sind zentral fuer das P-Code-Handbuch.
- Beispiele in diesem Handbuch referenzieren diese Liste.

Originalinhalt

TinyPl0 VM-Befehlssatz

Instruktionsformat

Jede Instruktion besteht aus:

- `op`: Opcode
- `l`: Lexikalische Level-Differenz
- `a`: Argument (Adresse oder Untercode)

Kodiert in C# als:

- `/Users/thorstenhindermann/Codex/TinyPl0/src/Pl0.Core/Instruction.cs`
- `/Users/thorstenhindermann/Codex/TinyPl0/src/Pl0.Core/Opcode.cs`

Opcodes

Opcode	Wert	Bedeutung
`lit`	0	Konstante `a` auf Stack laden
`opr`	1	ALU-/Kontrolloperation nach Untercode `a`
`lod`	2	Variable aus statischer Tiefe `l`, Offset `a` laden
`sto`	3	Obersten Stackwert in statische Tiefe `l`, Offset `a` speichern
`cal`	4	Prozeduraufruf mit statischem Link
`int`	5	Stack um `a` Zellen erweitern
`jmp`	6	Unbedingter Sprung zu `a`

Opcode	Wert	Bedeutung
`jpc`	7	Bedingter Sprung zu `a` bei `0` auf Stack

OPR-Untercodes

Untercode	Bedeutung
0	Return (Frame verlassen)
1	Vorzeichenwechsel (`-x`)
2	Addition
3	Subtraktion
4	Multiplikation
5	Division
6	`odd`-Test
8	Gleichheit (`=`)
9	Ungleichheit (`#`)
10	Kleiner (`<`)
11	Groesser-gleich (`>=`)
12	Groesser (`>`)
13	Kleiner-gleich (`<=`)
14	Integer-Eingabe (`?`)
15	Integer-Ausgabe (`!`)

Implementierung:

- `/Users/thorstenhindermann/Codex/TinyPl0/src/Pl0.Vm/VirtualMachine.cs`

Registermodell

- `P`: Program Counter

- `B`: Basiszeiger (aktueller Aktivierungsrahmen)
- `T`: Stack-Top

Statische Kette (`base(l)`) wird ueber `ResolveBase` aufgeloest.

Definierte Laufzeitdiagnosen

Code	Bedeutung
206	Division durch 0
98	EOF bei Integer-Eingabe
97	Ungueltiges Integer-Format bei Eingabe
99	Sonstiger VM-Laufzeitfehler (z. B. Stack-/Pointerfehler)

Traceability-Matrix (kuratiert)

Die Traceability-Matrix verbindet Anforderungen und Tests.

Originalinhalt

TinyPl0 Traceability-Matrix

Diese Matrix bildet die Coverage-Gate-Anforderung aus dem Pflichtenheft ab:

- jede Sprachregel aus Abschnitt 4.1.1
- jede VM-Regel aus Abschnitt 4.3

muss mindestens einem Pflichttestfall zugeordnet sein.

Quelle der Zuordnung

- Maschinenlesbare Matrix:
  - `/Users/thorstenhindermann/Codex/TinyPl0/tests/data/expected/traceability/matrix.json`
- Referenzkatalog der Pflichttestfaelle:
  - `/Users/thorstenhindermann/Codex/TinyPl0/tests/data/expected/catalog/cases.json`

Automatischer Gate-Test

- `/Users/thorstenhindermann/Codex/TinyPl0/tests/Pl0.Tests/TraceabilityMatrixTests.cs`

Der Test validiert:

1. Vollstaendigkeit aller geforderten Sprachregeln.
2. Vollstaendigkeit aller geforderten VM-Regeln.
3. Jede Regel verweist auf mindestens einen katalogisierten Pflichttestfall.

Qualitaet (kuratiert)

Dieses Kapitel fasst die Qualitaetsziele zusammen und bindet die Detailseite ein.

Originalinhalt

Qualität und Testabdeckung

Ziel

Abdeckung der Kernpfade aus Lexer, Parser/Codegenerator, VM und CLI.

Testebenen

- Unit: Parser-/Lexer-/CLI-Optionen und VM-Operationen.
- Golden: Referenzvergleich von Token- und P-Code-Streams.
- End-to-End: `source -> pcode -> vm`.

Kernpfad-Matrix

Bereich	Tests
CLI-Switches und Subcommands	`CliOptionsParserTests`
Lexer + Positionstracking	`LexerTests`, `LexerGoldenTests`
Parser + Codegen + Dialektregeln	`ParserGoldenTests`, `ParserDiagnosticsTests`
VM-Laufzeit + Fehlerfälle	`VirtualMachineTests`
P-Code Datei-Roundtrip und E2E	`PCodeSerializerTests`
Katalogpflichtfaelle 8.2 + Golden-Code	`CatalogCasesTests`
Traceability Coverage-Gate (4.1.1 + 4.3)	`TraceabilityMatrixTests`

Traceability-Matrix

- Dokumentation:
  - `/Users/thorstenhindermann/Codex/TinyPl0/docs/TRACEABILITY_MATRIX.md`
- Datenbasis:
  - `/Users/thorstenhindermann/Codex/TinyPl0/tests/data/expected/traceability/matrix.json`

Lokale Qualitätskommandos

```
dotnet restore
dotnet build TinyPl0.sln --configuration Release
dotnet test TinyPl0.sln --configuration Release --no-build
dotnet test TinyPl0.sln --configuration Release --collect:"XPlat Code Coverage"
```

CI

Der Workflow führt Build, Tests und Coverage-Collection aus.

Manuelles Golden-Update

- Script für Maintainer-Workflow:
  - `/Users/thorstenhindermann/Codex/TinyPl0/scripts/update-golden-code.sh`