UNIVERSITÄT
DES
SAARLANDES

Programmierung 2 - SoSe 2019

Projekt 6 - Compiler

Stephan-Alexander Ariesanu, Jens Petermann

03. Juli 2019

Universität des Saarlandes

Überblick

1. Technische Hinweise
2. Struktur des Compilers
3. Details
4. Häufige Fehler

Technische Hinweise

Git Projekt-Repository

Wir können das Projekt mit `git clone` unter folgender URL beziehen:

```
https://prog2scm.cdl.uni-saarland.de/git/project6/<NAME>
```

<NAME>: Euer CMS-Benutzername

Achtung!

Außerhalb der Uni nur mit VPN erreichbar!

Eine Anleitung steht auf der Website unter Software.

Import in Eclipse

Menüeintrag *File*

→ Unterpunkt *Import*

→ *General*

→ *Existing project into workspace*

→ geklonten Order auswählen

Struktur des Compilers

Projektübersicht

TinyC $\longrightarrow$ Compiler $\longrightarrow$ MIPS

TinyC

TinyC ist fast wie C, aber z.B.

- `float pi = 3.1415;`

wird **nicht** unterstützt.

TinyC

TinyC ist fast wie C, aber z.B.

- `float pi = 3.1415;`
- `struct Date { /* content */ };`

wird **nicht** unterstützt.

TinyC

TinyC ist fast wie C, aber z.B.

- `float pi = 3.1415;`
- `struct Date { /* content */ };`
- `int mask = 0xff << 8;`

wird **nicht** unterstützt.

TinyC

TinyC ist fast wie C, aber z.B.

- `float pi = 3.1415;`
- `struct Date { /* content */ };`
- `int mask = 0xff << 8;`
- `void foo(int a, int b, int c, int d, int e);`

wird **nicht** unterstützt.

TinyC

TinyC ist fast wie C, aber z.B.

- `float pi = 3.1415;`
- `struct Date { /* content */ };`
- `int mask = 0xff << 8;`
- `void foo(int a, int b, int c, int d, int e);`
- globale Variable

wird **nicht** unterstützt.

Grammatikbeispiel

```
// ExternalDeclaration = FunctionDeclaration
void printf(int); // Parameter without Identifier

// ExternalDeclaration = Function
int main(){ // Block
  int x; int y = 3; // Declarations

  // Statement = ExpressionStatement
  // = BinaryExpression
  x = 5 + y;

  printf(y); // Expression = FunctionCall
  return x; // Statement = ReturnStatement
}
```

Front End

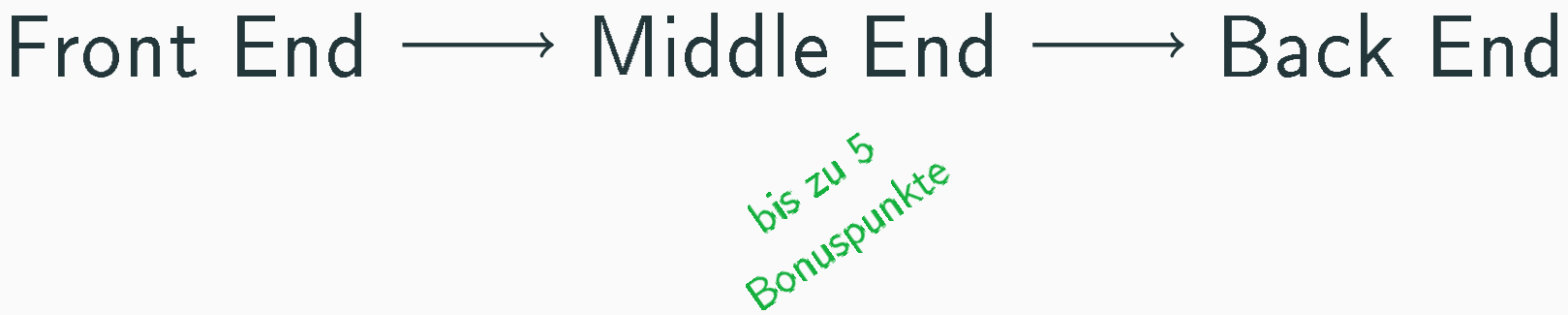
Compileraufbau

Front End $\longrightarrow$ Middle End

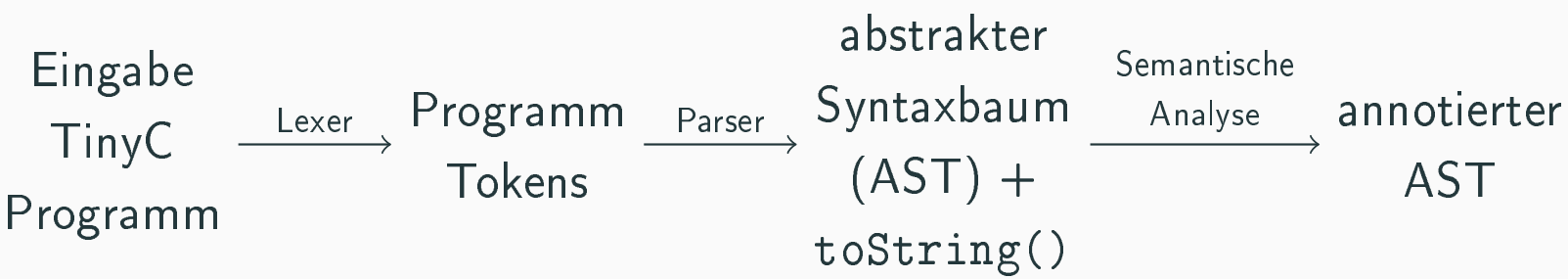
Compileraufbau

Front End $\longrightarrow$ Middle End $\longrightarrow$ Back End

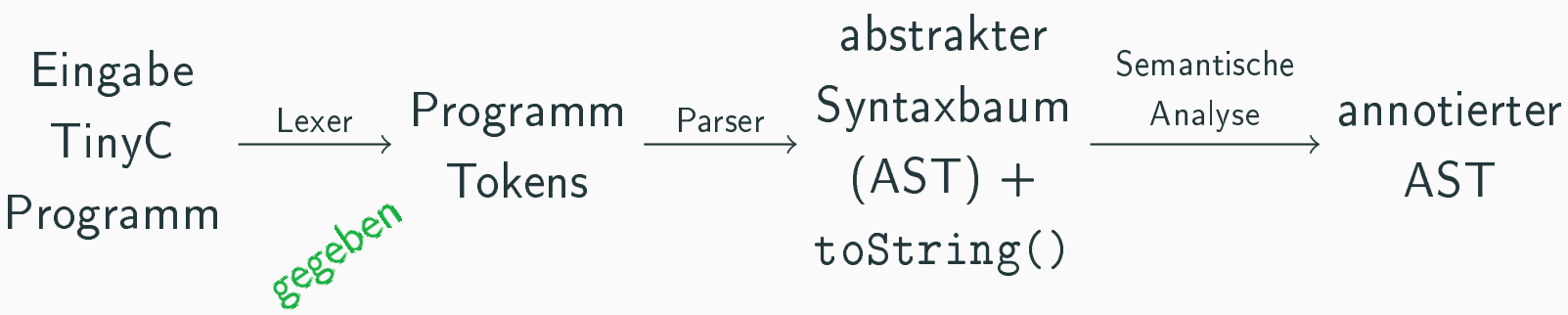
Compileraufbau



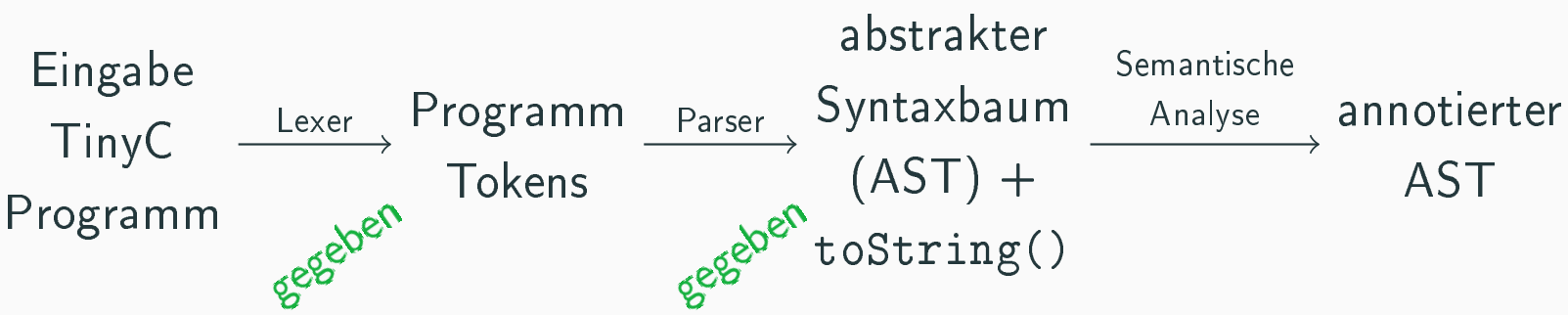
Front End


Eingabe
TinyC
Programm
Lexer
Programm
Tokens
Parser
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST

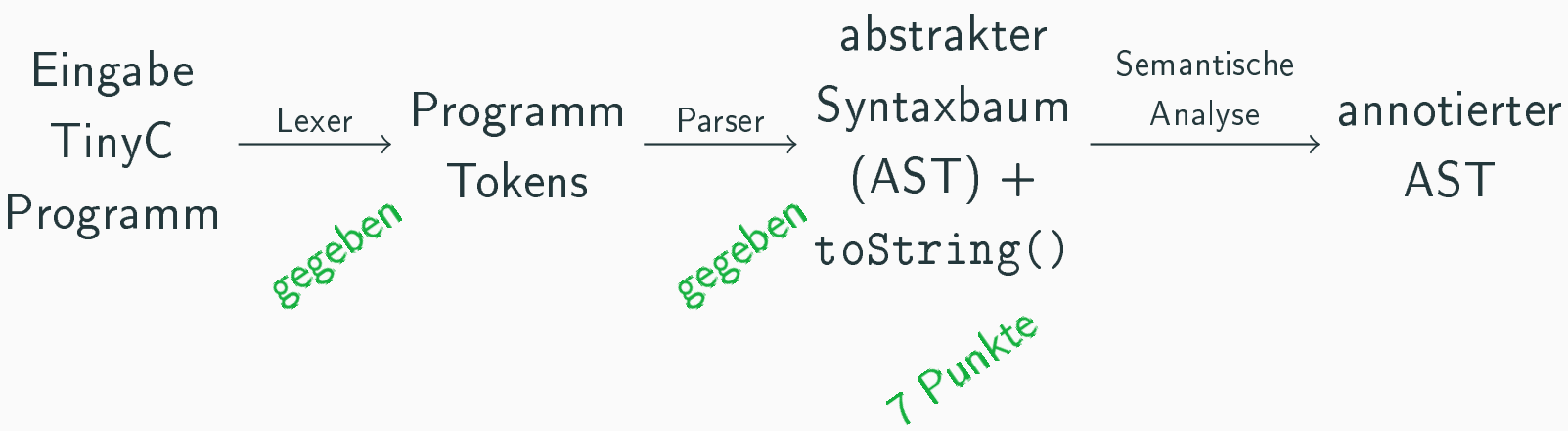
Front End


Eingabe
TinyC
Programm
Lexer
gegeben
Programm
Tokens
Parser
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST

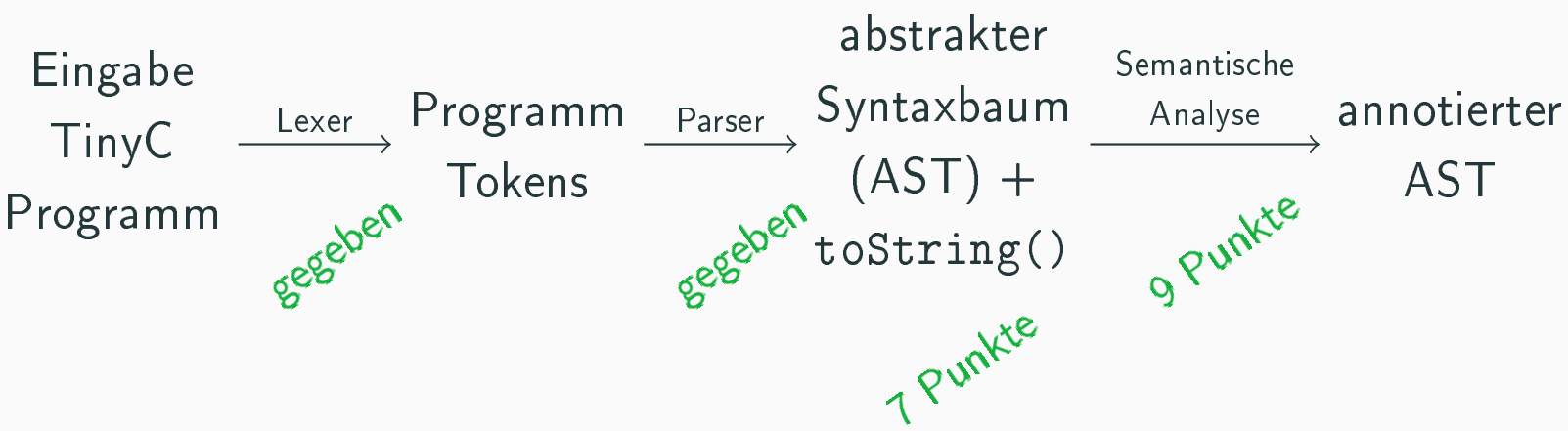
Front End


Eingabe
TinyC
Programm
Lexer
gegeben
Programm
Tokens
Parser
gegeben
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST

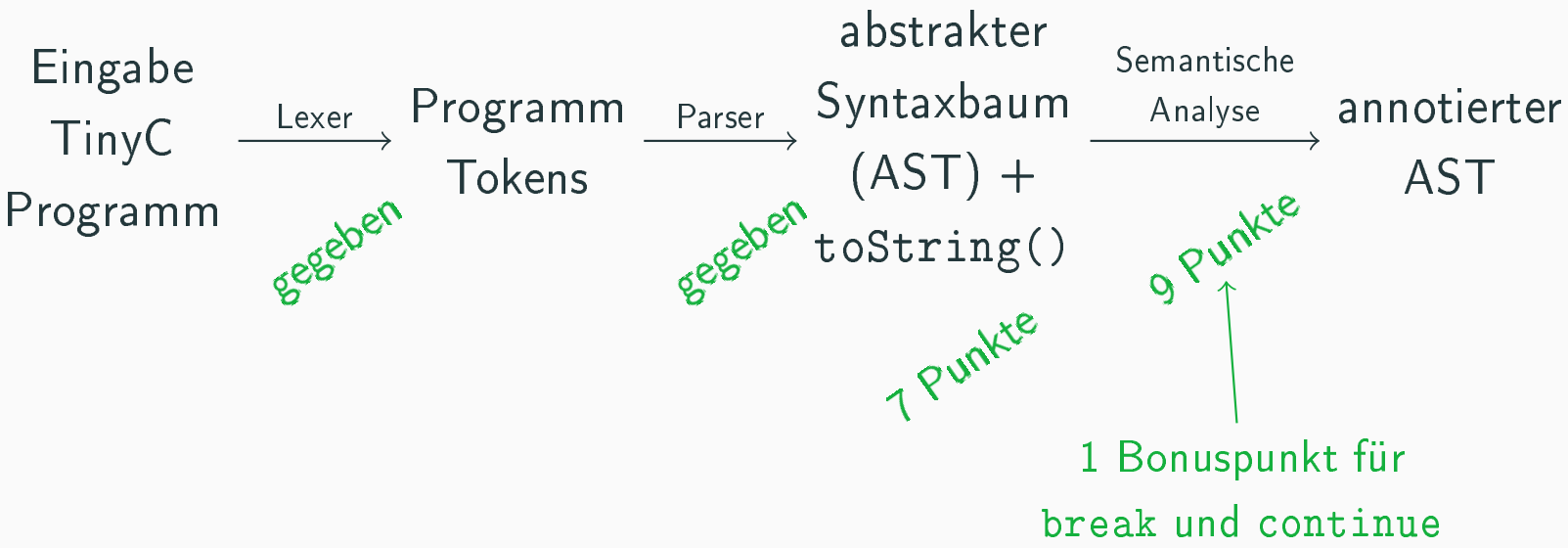
Front End


Eingabe
TinyC
Programm
Lexer
Programm
Tokens
Parser
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST
gegeben
gegeben
7 Punkte

Front End


Eingabe
TinyC
Programm
Lexer
Programm
Tokens
Parser
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST
gegeben
gegeben
7 Punkte
9 Punkte

Front End


Eingabe
TinyC
Programm
Lexer
Programm
Tokens
Parser
abstrakter
Syntaxbaum
(AST) +
toString()
Semantische
Analyse
annotierter
AST
gegeben
gegeben
7 Punkte
9 Punkte
1 Bonuspunkt für
break und continue

Back End

annotierter

AST

Back End

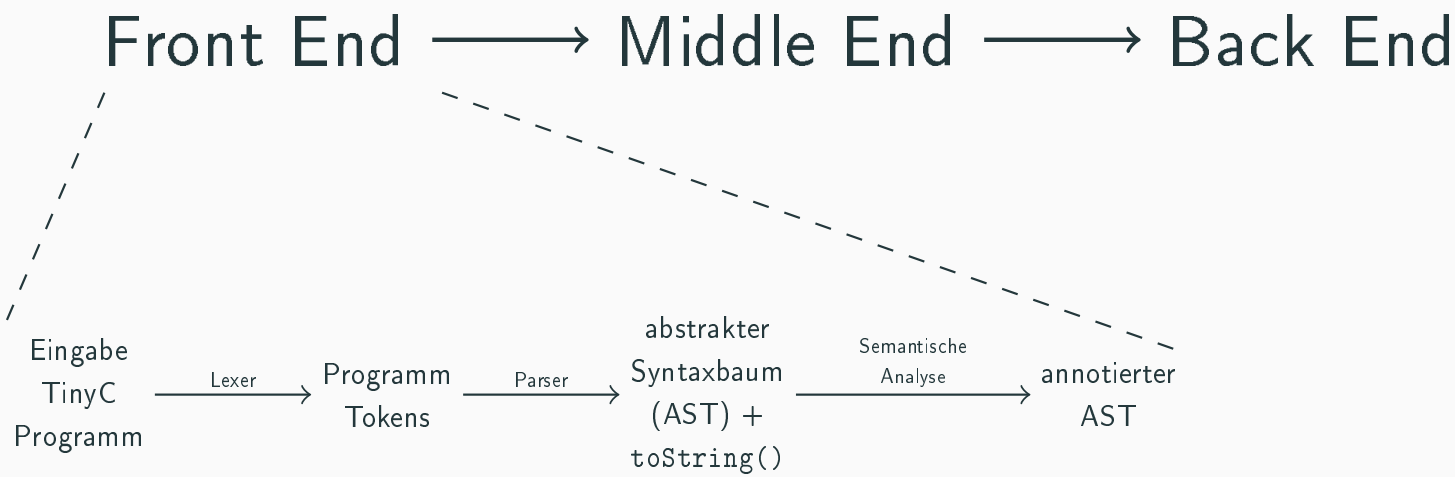
Back End


annotierter
AST
Codeerzeugung
Ausgabe
MIPS
Programm
9 Punkte

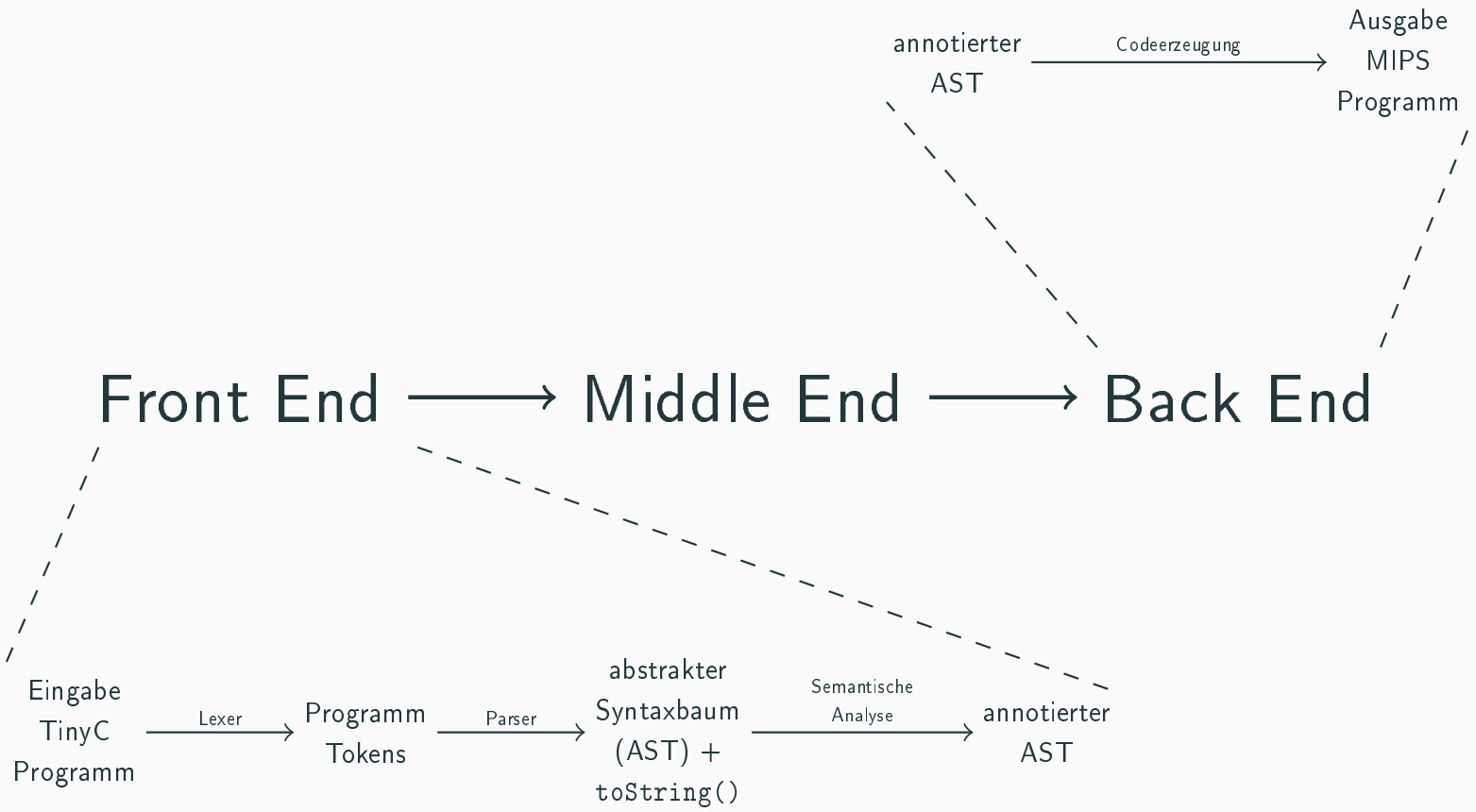
Zusammenfassung

Front End $\longrightarrow$ Middle End $\longrightarrow$ Back End

Zusammenfassung



Zusammenfassung



Projektpunkte

Regulär:

Projektpunkte

Regulär:

7 ASTFactory Implementierung + `toString()` Methoden

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Bonus:

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Bonus:

1	`break`, `continue` (Semantische Analyse)

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Bonus:

	1	`break`, `continue` (Semantische Analyse)
+	3	Operatoren `!`, `&&`, `||`, `e1 ? e2 : e3`, `++`, `--`

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Bonus:

	1	`break`, `continue` (Semantische Analyse)
+	3	Operatoren `!`, `&&`, `||`, `e1 ? e2 : e3`, `++`, `--`
+	5	`performOptimizations()`

Projektpunkte

Regulär:

	7	ASTFactory Implementierung + `toString()` Methoden
+	9	Semantische Analyse (Typen & Gültigkeitsbereiche)
+	9	syntaxgesteuerte Code-Erzeugung
$\sum$	25	

Bonus:

	1	`break`, `continue` (Semantische Analyse)
+	3	Operatoren `!`, `&&`, `||`, `e1 ? e2 : e3`, `++`, `--`
+	5	`performOptimizations()`
$\sum$	9	

4 Bonuspunkte - automatisch getestet

- (1 Punkt) Semantische Analyse: `break` und `continue`

4 Bonuspunkte - automatisch getestet

- (1 Punkt) Semantische Analyse: `break` und `continue`
- (1 Punkt) Logische Operatoren: `!`, `&&` und `||` mit Kurzauswertung

4 Bonuspunkte - automatisch getestet

- (1 Punkt) Semantische Analyse: `break` und `continue`
- (1 Punkt) Logische Operatoren: `!`, `&&` und `||` mit Kurzauswertung
- (1 Punkt) Ternärer Operator `e1 ? e2 : e3`

4 Bonuspunkte - automatisch getestet

- (1 Punkt) Semantische Analyse: `break` und `continue`
- (1 Punkt) Logische Operatoren: `!`, `&&` und `||` mit Kurzauswertung
- (1 Punkt) Ternärer Operator `e1 ? e2 : e3`
- (1 Punkt) Prä- und Postinkrement/-dekrement

5 Bonuspunkte - mündlich

- Konstantenfaltung

- oder vieles mehr . . .

Fragen?

Details

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

driver Treiber, der den Übersetzungsprozess steuert (parst Befehlszeile etc.)

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

driver	Treiber, der den Übersetzungsprozess steuert (parst Befehlszeile etc.)
parser	Lexer und Parser für `TinyC` (bereits implementiert)

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

driver	Treiber, der den Übersetzungsprozess steuert (parst Befehlszeile etc.)
parser	Lexer und Parser für `TinyC` (bereits implementiert)
mipsasmgen	Funktionalität, um MIPS-Assemblerbefehle zu erzeugen und auszugeben

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

driver	Treiber, der den Übersetzungsprozess steuert (parst Befehlszeile etc.)
parser	Lexer und Parser für `TinyC` (bereits implementiert)
mipsasmgen	Funktionalität, um MIPS-Assemblerbefehle zu erzeugen und auszugeben
implementation	Eure Implementierung muss in dieses Paket

Paketstruktur

Der Compiler befindet sich im Paket `tinycc` und ist dort in mehrere Unterpakete unterteilt:

driver	Treiber, der den Übersetzungsprozess steuert (parst Befehlszeile etc.)
parser	Lexer und Parser für `TinyC` (bereits implementiert)
mipsasmgen	Funktionalität, um MIPS-Assemblerbefehle zu erzeugen und auszugeben
implementation	Eure Implementierung muss in dieses Paket
diagnostic	Funktionalität für Fehlerbehandlung und Ausgaben (Fehlermeldungen und Warnungen) des Übersetzers

Diagnostic

Hat 2 Methoden:

- `printError`: Hiermit müssen Fehlermeldungen der Semantischen Analyse und ihre exakte Position ausgegeben werden.

Diagnostic

Hat 2 Methoden:

- `printError`: Hiermit müssen Fehlermeldungen der Semantischen Analyse und ihre exakte Position ausgegeben werden.

- `printNote`: Hiermit können zusätzliche Informationen zu Programmpositionen ausgeben werden (optional).

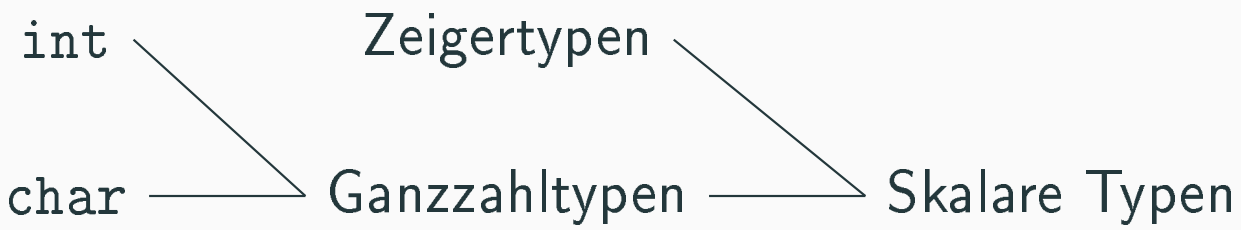
Alle neuen Klassen müssen in das Paket `tinycc.implementation` oder in Subpakete davon.

MipsAsmGen

- Bereits implementiert
- Hiermit wird die Codeerzeugung implementiert
- Erzeugt MIPS-Code (Labels, Anweisungen etc.)

Semantische Analyse - Typen

Semantische Analyse - Typen



Semantische Analyse - Typen

Semantische Analyse - Typen

Semantische Analyse - Typen

- unvollständige Typen
  (nur `void` und Funktionstypen)

Semantische Analyse - Typen

- unvollständige Typen
  (nur `void` und Funktionstypen)
- vollständige Typen
  (alles andere und insbesondere `void*`)

Klassenaufbau (AST)

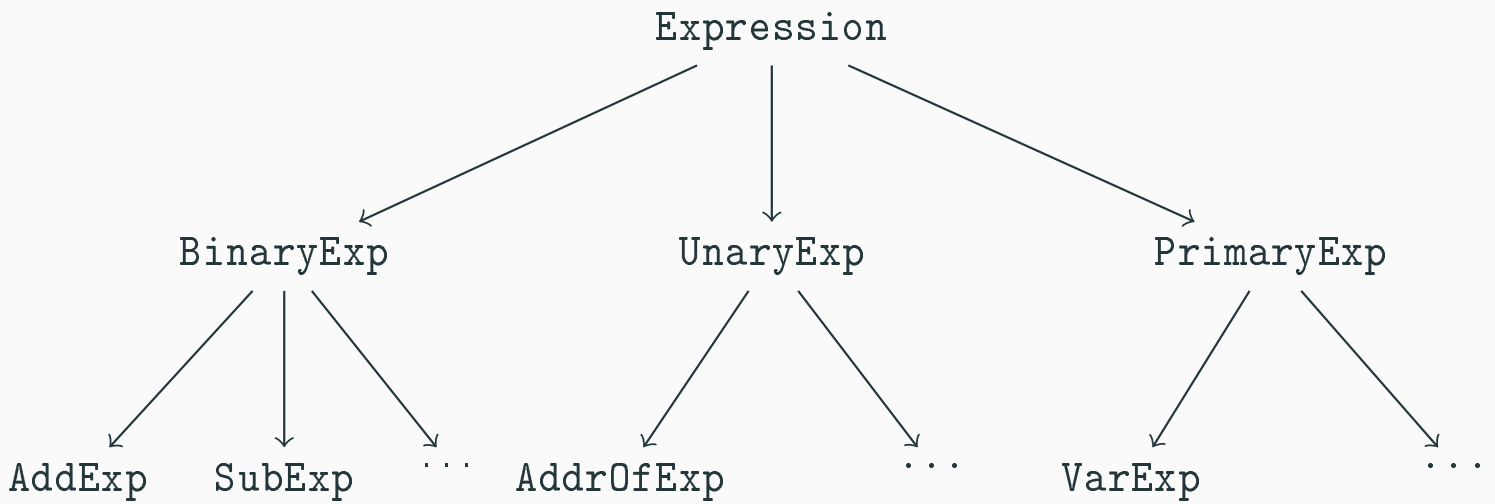
`Type` `Expression` `Statement`

Klassenaufbau (AST)

Type

Expression

Statement



ASTFactory

- Es gibt Methoden für alle Anweisungen, Ausdrücke und Typen
- `createExternalDeclaration` und `createFunctionDefinition` geben void zurück
- **Tipp:** Liste von `ExternalDeclarations` in `ASTFactory` Implementierung

Häufige Fehler

Projekt nicht bearbeiten

sizeof() ist ein Operator!

- sizeof ist ein Operator und keine Funktion!

sizeof() ist ein Operator!

- sizeof ist ein Operator und keine Funktion!
- sizeof wird beim Kompilieren des Programms und nicht bei dessen Ausführung ausgewertet

sizeof() ist ein Operator!

- sizeof ist ein Operator und keine Funktion!
- sizeof wird beim Kompilieren des Programms und nicht bei dessen Ausführung ausgewertet
- sizeof von einer Expression ist dasselbe wie sizeof vom Typen der Expression

sizeof() ist ein Operator!

- sizeof ist ein Operator und keine Funktion!
- sizeof wird beim Kompilieren des Programms und nicht bei dessen Ausführung ausgewertet
- sizeof von einer Expression ist dasselbe wie sizeof vom Typen der Expression
- `sizeof(1/0)` ist gültig und wertet zum selben Wert aus wie `sizeof(int)`

Den C Standard nicht neu erfinden!

- Zuweisungen sind binäre Ausdrücke - sie werden ausgewertet:
  `a = b = c = 3;` ist gültig

Den C Standard nicht neu erfinden!

- Zuweisungen sind binäre Ausdrücke - sie werden ausgewertet:
  `a = b = c = 3;` ist gültig

- Expressionstatements existieren: `if (4) 3; else 5;`
  ist gültig

Den C Standard nicht neu erfinden!

- Zuweisungen sind binäre Ausdrücke - sie werden ausgewertet:
  `a = b = c = 3;` ist gültig

- Expressionstatements existieren: `if (4) 3; else 5;`
  ist gültig

- Funktionsdeklarationen brauchen keine Parameternamen,
  Funktionsdefinitionen schon

Den C Standard nicht neu erfinden!

- Zuweisungen sind binäre Ausdrücke - sie werden ausgewertet: `a = b = c = 3;` ist gültig
- Expressionstatements existieren: `if (4) 3; else 5;` ist gültig
- Funktionsdeklarationen brauchen keine Parameternamen, Funktionsdefinitionen schon
- Der Typ void ist unvollständig und hat damit einige Sonderregeln

Weitere Tipps

- **Skriptkapitel 10** und Vorlesungsfolien sind sehr hilfreich

Weitere Tipps

- **Skriptkapitel 10** und Vorlesungsfolien sind sehr hilfreich
- Die Grammatik und die Typregeln strikt einhalten

Weitere Tipps

- **Skriptkapitel 10** und Vorlesungsfolien sind sehr hilfreich
- Die Grammatik und die Typregeln strikt einhalten
- Labels nur mit der Klasse `MipsAsmGen` erstellen

Weitere Tipps

- **Skriptkapitel 10** und Vorlesungsfolien sind sehr hilfreich
- Die Grammatik und die Typregeln strikt einhalten
- Labels nur mit der Klasse `MipsAsmGen` erstellen
- Registerverbrauch bei Ausdrucksauswertung nur optimal, wenn Teilausdruck mit höherem Registerverbrauch zuerst ausgewertet wird: Funktion *regs* aus dem Skript (S. 270) verwenden

Tests und Debugging

- Schreibt euch eigene Tests!
  Diese werden gegen die Referenz getestet

Tests und Debugging

- Schreibt euch eigene Tests!
  Diese werden gegen die Referenz getestet

- Nutzt zuerst Minimalbeispiele

Tests und Debugging

- Schreibt euch eigene Tests!
  Diese werden gegen die Referenz getestet

- Nutzt zuerst Minimalbeispiele

- `DEFAULT_TIMEOUT` in `prog2.tests.CompilerTests.java`
  kann lokal erhöht werden

Tests und Debugging

- Schreibt euch eigene Tests!
  Diese werden gegen die Referenz getestet

- Nutzt zuerst Minimalbeispiele

- `DEFAULT_TIMEOUT` in `prog2.tests.CompilerTests.java`
  kann lokal erhöht werden

- `PRINT_ASM_CODE = true` gibt euren erzeugten MIPS-Code im Terminal aus

Fragen?