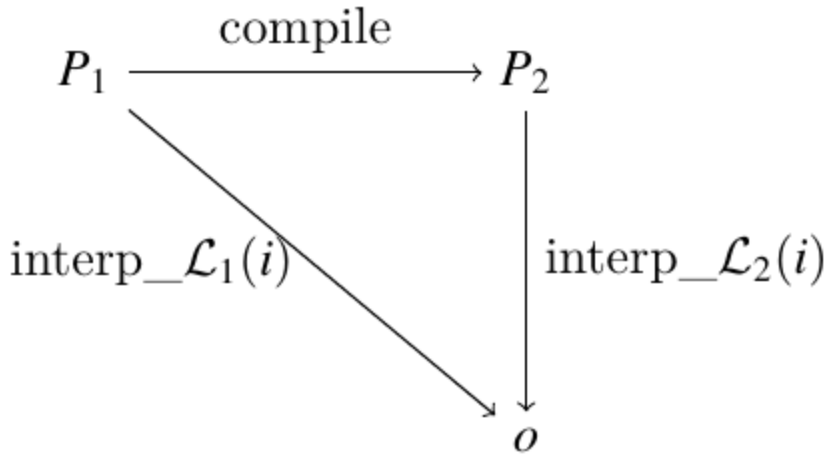
Abschluss-präsentation

Bachelorprojekt PicoC-Compiler

Definitionen

Definitionen

Compiler und Interpreter



Compiler und Parser

- **Compiler:** *High-level Programm* $\xrightarrow{übersetzen}$ *Maschinencode* (ließt *ganzen* Code ein)
- **Interpreter:** *Zeile für Zeile* einlesen und *direkt ausführen*

Definitionen

Concrete Syntax

- Programm als **Textrepräsentation**
- das was man Compiler als **Input** gibt
- durch **Grammatik** dargestellt:

```
<code_le> = <pred_2>
<pred_2> =  <pred_1> '||' <pred_1>
<pred_1> = <logic_operand> '&&' <logic_operand>
<logic_operand> = !<logic_operand> | (<code_le>) | <code_ae> | <code_ae> <cmp> <code_ae>
<cmp> = '<' | '>' | '<=' | '>=' | '==' | '!='
```

Definitionen

Abstract Syntax

- Darstellung **innerhalb** des Compilers
- **Abstract Syntax Tree**, der aus **Nodes** besteht und so aufgebaut ist, dass er die **Operationen**, die der Compiler ausführen muss **optimal unterstützt**
- durch **Grammatik** dargestellt:

```
<code_le> = LogicBinaryOperation(<logic_operand>, <logic_connective>, <logic_operand>)
<logic_operand> = Not(<logic_operand>) | <code_le> | ToBool(<code_ae>) | Atom(<code_ae>, <cmp>, <code_ae>)
<logic_connective> = LAnd() | LOr()
<cmp> = Lt() | Gt() | Le() | Ge() | Eq() | UEq()
```

Definitionen

Lexer und Tokens

- **Lexer:** erstellt *Tokens* aus einem Stream von Symbolen, indem er lexikalische Patterns erkennt

```
void main() {
  char var = 12 + 1;
}
```

⇓

```
[<TT.IDENTIFIER>, <TT.VOID>, <TT.MAIN>, <TT.L_PAREN>, <TT.R_PAREN>,
<TT.L_BRACE>, <TT.CHAR>, <TT.IDENTIFIER>, <TT.ASSIGNMENT>, <TT.NUMBER>,
<TT.PLUS_OP>, <TT.NUMBER>, <TT.SEMICOLON>, <TT.R_BRACE>]
```

Definitionen

Parser und Abstract Syntax Tree

- **Parser:** *Unwandlung* einer *Eingabe* in ein für die Weiterverarbeitung geignetes *Format*
  - *Tokens* $\xRightarrow{baut}$ *Abstract Syntax Tree*

```
[<TT.IDENTIFIER>, <TT.VOID>, <TT.MAIN>, <TT.L_PAREN>, <TT.R_PAREN>,
<TT.L_BRACE>, <TT.CHAR>, <TT.IDENTIFIER>, <TT.ASSIGNMENT>, <TT.NUMBER>,
<TT.PLUS_OP>, <TT.NUMBER>, <TT.SEMICOLON>, <TT.R_BRACE>]
```

⇓

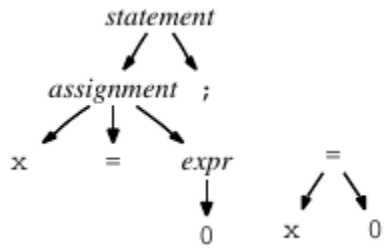
```
(stdin (void main ((char var) = (12 + 1))))
```

- **Terminalsymbole** innerhalb der Tokens dienen als Ankerpunkte zu **Unterscheidung** zweier Weggabelungen

Definitionen

Abstract Sytax Tree

- **Vorrausssetzungen:**
  - Nutzlose Nodes rauswerfen
  - **einfach** den Tree **entlanglzulaufen**, **Pattern** im Baum leicht **identifizierbar**
  - Beziehung von **Operatoren** und **Operanden** soll hervorgehoben werden, **unempfindlich** gegenüber **Änderungen** der Grammatik



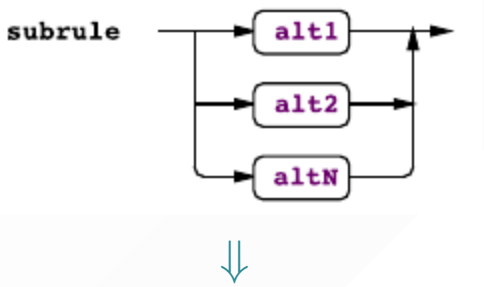
from Parse Tree to Abstract Syntax Tree

- durch Enkopplung von ursprünglicher Syntax, kommt man **Operator-Operand Model** des RETI-Assembler näher
- mehrere Sprachen in diese **Indermediate Representatio (IR)** übersetzbar

Definitionen

Grammatiken in Code übersetzen

```
(«alt1»|«alt2»|..|«altN»)
```



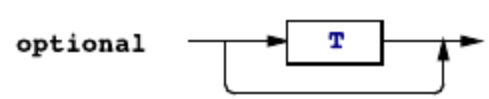
⇓

```
if ( «lookahead-predicts-alt1» ) { «match-alt1» }
else if ( «lookahead-predicts-alt2» ) { «match-alt2» }
...
else if ( «lookahead-predicts-altN» ) { «match-altN» }
else «throw-exception» // parse error (no viable alternative)
```

Definitionen

Grammatiken in Code übersetzen

(T)?



```
if ( «lookahead-is-T» ) { match(T); } // no error else clause
```

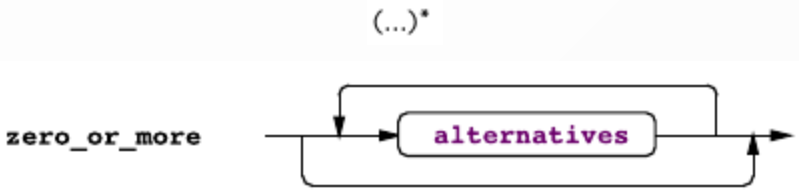
Definitionen

Grammatiken in Code übersetzen

```
do {
    «code-matching-alternatives»
} while ( «lookahead-predicts-an-alt-of-subrule» );
```

Definitionen

Grammatiken in Code übersetzen



```
while ( «lookahead-predicts-an-alt-of-subrule» ) {
    «code-matching-alternatives»
}
```

Definitionen

Passes

- **Problem:** von der **abstrakten Syntax** von **PicoC** zu **abstrakter Syntax** des **RETI-Assembler** übersetzen
  - dazu das Problem in mehrere **Passes** unterteilen
  - ein einem **Pass** nur **ein Ziel** erfüllen und nicht mehrere gleichzeitig
  - "passing over"

Definitionen

Passes

```
(stdin (void main ((char var) = (12 + 1))))
```

⇓ alle Passes

```
LOADI SP 256;
SUBI SP 1;
LOADI ACC 12;
STOREIN SP ACC 1;
# ...
LOADI IN1 -256;
OR ACC IN1;
STORE ACC 128;
JUMP 0;
```

Funktionsumfang

Funktionsumfang

Syntax von PicoC

- nur **ein Hauptprogramm**
- nur Datentypen **int**, **char**
- Kontrollstrukturen:
  - **if**, **if-else**
  - **while**- und **do-while**-Schleifen
- **Arithmetische Ausdrücke** mit
  - binären Operatoren `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`
  - unären Operatoren `-`, `~`
- **Logische Ausdrücke** mit
  - Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=`, `>=`
  - Logische Operatoren `!`, `&&`, `||`
- nur **einfache Zuweisungen** mit `=`

Funktionsumfang

Syntax von PicoC in der Vorlesung

Pico-C: Einschränkungen (1)

- Nur Hauptprogramm, sonst keine Funktionen / Prozeduren
- Keine Bibliotheksfunktionen
- Datentypen:
  - Keine Zeiger, kein dynamischer Speicher
  - Keine Felder
  - Keine Verbunde (**struct**)
  - Keine Datentypen **float**, **double**
  - Nur Datentypen **int**, **char**
- Kontrollstrukturen:
  - **if**, **if-else**
  - **while**- und **do-while**-Schleifen
  - Keine **for**-Schleifen

Funktionsumfang

Syntax von PicoC in der Vorlesung

Pico-C: Einschränkungen (2)

- Arithmetische Ausdrücke mit
  - binären Operatoren +, -, *, /, %, &, |, ^,
  - unären Operatoren -, ~
- Logische Ausdrücke mit
  - Vergleichsoperatoren ==, !=, <, >, <=, >=
  - Logische Operatoren !, &&, ||
- Nur einfache Zuweisungen mit =
  - Keine kombinierten Zuweisungen wie +=
  - Kein Inkrement ++, Dekrement --
- Keine Initialisierung im Deklarationsteil (außer bei Konstanten)

Funktionsumfang

Zusätze / Änderungen

- **Kommentare**
- "**else if**":
  `if <single-statement> else if { <statement(s)> } else { <statement(s)> }`
- **Präzidenzregeln**
- zu **großes Literal** für `char` Datentyp (**Implicit Conversion**)
- zu **großes Literal** für Parameter
- **Fehlermeldungen** und **Warnings**
- **Shell** oder **Datei** angeben
- **Config-** bzw. **Dot-Files** um Einstellungen und Historie zu speichern
- **Farbige Ausgabe** von **Fehlermeldungen**, **RETI-Code**, **Symboltabelle**, **Abstraker Syntax**, **Token** usw.

Funktionsumfang

Kommentare und else if

```
void main() {
  const int var = 12;  // Einzeiliger Kommentar
  /* Mehrzeiliger Kommentar,
  der sich über mehrere Zeilen
  erstreckt */
  char var2;
  if (var > 100) {
    var2 = 2;
  } else if (/* Störender Kommentar */ var > 10)
    var2 = 1;
  else
    var2 = 0;
}
```

Funktionsumfang

Präzidenzregeln

- **Konkrette Syntax:** `12 + 'c' - 1;`
  - **Astrakte Syntax:** `(12 + (99 - 1))`
- **Konkrette Syntax:** `12 * 'c' - 1;`
  - **Astrakte Syntax:** `((12 * 99) - 1)`
- **Konkrette Syntax:** `(12 < 1 + 2) * 2;`
  - **Astrakte Syntax:** `((12 < (1 + 2)) * 2)`
- **Konkrette Syntax:** `-(0 || !(12 < 3 || 3 >= 12));`
  - **Astrakte Syntax:** `(- (ToBool(0) || Not(((12 < 3) || (3 >= 12)))))`
- **Konkrette Syntax:** `12 < 1 + 2 && 12 || 0;`
  - **Astrakte Syntax:** `(((12 < (1 ^ 2)) && ToBool(12)) || ToBool(0))`

Funktionsumfang

Zu großes Literal für `char`

- Wertebereich von `char` ist zwischen -2^7 und $2^7 - 1$

```
char var = 127;    // 2^7-1 ✅
char var_2 = 128;  // 2^7   ❌
```

- **Implicit Conversion** von `int` zu `char`:

```
  00000000_00101011_10100110_01111111
& 00000000_00000000_00000000_11111111  // 255
  00000000_00000000_00000000_01111111
```

  - mit **Bitmaske** abhängig vom **"Vorzeichenbit"** an der **8ten Stelle** nach der **8ten Stelle** mit 0en oder 1en überschreiben
    - **Fall 1:** 8te Stelle, Wert auf rechter Seite **positiv**
      - keine **Signextension** nötig

Funktionsumfang

Zu großes Literal für `char`

- **Fall 2:** 8te Stelle, Wert auf rechter Seite **negativ**

```
  00000000_00000000_00000000_10000000  //  128
v 11111111_11111111_11111111_00000000  // -256
  11111111_11111111_11111111_10000000  // -128
```

- Vergleich **PicoC-Compiler** und **Clang**:

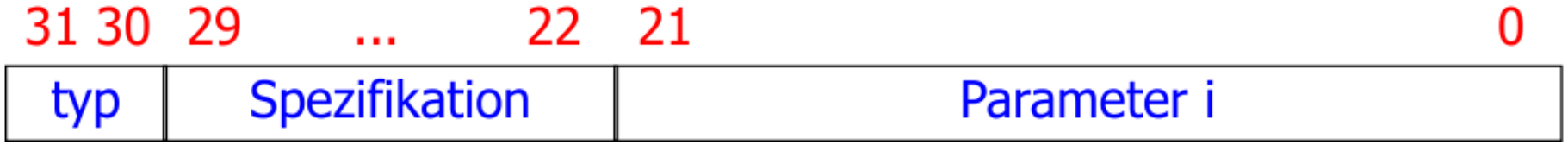
```
./tests/warning_atomic_assignment_char_char.picoc:2:13: ImplicitConversionWarning: Literal '128' will be implicitly conve
rted from 'int' to 'char' in the course of being assigned to 'var'. Changes value from '128' to '-128'
void main() {
  char var = 128;
            ~~~
}
./tests/warning_atomic_assignment_char_char.picoc:2:7: Note: Variable 'var' definied as type 'char' here:
void main() {
  char var = 128;
       ~~~
}
Note: Datatype 'char' has only range -128 to 127
```

```
> $ clang ./test_2.cpp
./test_2.cpp:4:14: warning: implicit conversion from 'int' to 'char' changes value from 128 to -128 [-Wconstant-conversion]
  char car = 128;
       ~~~   ^~~
1 warning generated.
```

Funktionsumfang

Zu großes Literal für Parameter

- semantischer Wert des **Literals** zwischen -2^{21} und $2^{21} - 1$




```
int var = 2097151;        // 2^21-1 ✅
int var_2 = 2147483647;   // 2^31-1 ❌
```

- Wert des Literals durch **Shiften** erreichen:

```
  00000000_00000000_01111111_11111111  // 2^15-1
* 00000000_00000001_00000000_00000000  // 2^16
  01111111_11111111_11111111_11111111  // 2^31-1
```

- aber sobald Wert des Literals $> 2^{31} - 1$ → `TooLargeLiteralError`

Funktionsumfang

Fehlermeldungen

- `MismatchedTokenError`
- `NoApplicableRuleError`
- `UnknownIdentifierError`
- `RedefinitionError`
- `ConstReassignmentError`
- `TooLargeLiteralError`
- `NoMainFunctionError`
- `MoreThanOneMainFunctionError`
- `InvalidCharacterError`
- `UnclosedCharacterError`
- `NotImplementedYetError`

Funktionsumfang

Fehlermeldungen

MismatchedTokenError

```
./tests/error_no_semicolon.picoc:3:2: MismatchedTokenError: Expected ';', found 'if'
void main() {
  int var = 32
              ^
              ;
  if (var < 3) {
  ~~
      var = 10;
  }
```

```
./tests/error_nested_main_functions.picoc:2:7: MismatchedTokenError: Expected 'identifier', found 'main'
void main() {
  void main() {;}
       ^~~~~
       identifier
}
```

Funktionsumfang

Fehlermeldungen

NoApplicableRuleError

```
./tests/error_if_no_condition.picoc:2:5: NoApplicableRuleError: Expected 'logic operand', found ')'
void main() {
  if() {}
     ^
     logic operand
}
```

```
./tests/error_no_operand_right.picoc:2:17: NoApplicableRuleError: Expected 'arithmetic operand', found ';'
void main() {
  int var = 42 % ;
                 ^
                 arithmetic operand
}
```

Funktionsumfang

Fehlermeldungen

UnknownIdentifierError

```
./tests/error_multi_assignment_not_definined.picoc:3:13: UnknownIdentifierError: Identifier 'var2' wasn't declared yet
void main() {
  char var1;
  int var3 = var2 = var1 = 1 + 1;
             ~~~~
}
```

```
./tests/error_not_initialised_variable.picoc:2:15: UnknownIdentifierError: Identifier 'var' wasn't declared yet
void main() {
  int x = 12 & var;
              ~~~
}
```

Funktionsumfang

Fehlermeldungen

RedefinitionError

```
./tests/error_redefinition_first_const.picoc:3:7: RedefinitionError: Redefinition of 'var'
void main() {
  const char var = 'z';
  char var = 'a';
       ~~~
}
./tests/error_redefinition_first_const.picoc:2:13: Note: Already defined here:
void main() {
  const char var = 'z';
             ~~~
  char var = 'a';
}
```

Funktionsumfang

Fehlermeldungen

ConstReassignmentError

```
./tests/error_const_reassignment.picoc:3:2: ConstReassignmentError: Can't reassign a new value to named constant 'var'
void main() {
  const int var = 12;
  var = 13;
  ~~~
}
./tests/error_const_reassignment.picoc:2:12: Note: Constant identifier was initialised here:
void main() {
  const int var = 12;
            ~~~
  var = 13;
}
```

Funktionsumfang

Fehlermeldungen

TooLargeLiteralError

```
./tests/error_too_large_literal_complex_assignment_int.picoc:3:14: TooLargeLiteralError: Literal '2147483648' is too large
void main() {
  int var = 2147483647;
  var = var + 2147483648;
              ~~~~~~~~~~
}
Note: The max size of a literal for a variable is in range '-2147483648' to '2147483647'
```

NoMainFunctionError

```
(error_empty)

NoMainFunctionError: There's no main function in file 'error_empty'
```

Funktionsumfang

Fehlermeldungen

MoreThanOneMainFunctionError

```
./tests/error_more_than_two_mains.picoc:1:5: MoreThanOneMainFunctionError: There're at least two main functions
void main(){;
     ~~~~
}void main() {;}
./tests/error_more_than_two_mains.picoc:2:6: Note: Second main function defined here:
void main(){;
}void main() {;}
      ~~~~
```

Funktionsumfang

Fehlermeldungen

InvalidCharacterError

```
./tests/error_invalid_character.picoc:2:15: InvalidCharacterError: '@' is not a permitted character
void main() {
  int var = 12 @ 3;
                ~
}
```

UnclosedCharacterError

```
./tests/error_unclosed_character_error.picoc:3:11: UnclosedCharacterError: Expected 'a', found 'a
void main() {
  int x = 'C' + 1;
  x = x + ('a - 'A');
           ^
           'a'
}
```

Funktionsumfang

Warnungen

- `ImplicitConversionWarning`

Funktionsumfang

Warnungen

ImplicitConversionWarning

```
./tests/warning_multiple_warnings.picoc:4:27: ImplicitConversionWarning: Literal '2147483647' will be implicitly converte
d from 'int' to 'char' in the course of being assigned to 'var3'. Changes value from '2147483647' to '-1'
  char var3;
  int var2;
  char var = var2 = var3 = 2147483647;
                           ~~~~~~~~~~
}
./tests/warning_multiple_warnings.picoc:2:7: Note: Variable 'var3' definied as type 'char' here:
void main() {
  char var3;
       ~~~~
  int var2;
  char var = var2 = var3 = 2147483647;
Note: Datatype 'char' has only range -128 to 127
```

Funktionsumfang

Warnungen

ImplicitConversionWarning

```
./tests/warning_multiple_warnings.picoc:4:13: ImplicitConversionWarning: Value of variable 'var2' will be implicitly conv
erted from 'int' to 'char' in the course of being assigned to 'var'
  char var3;
  int var2;
  char var = var2 = var3 = 2147483647;
             ~~~~
}
./tests/warning_multiple_warnings.picoc:4:7: Note: Variable 'var' definied as type 'char' here:
  char var3;
  int var2;
  char var = var2 = var3 = 2147483647;
       ~~~
}
Note: Datatype 'char' has only range -128 to 127
```

Funktionsumfang

Datei direkt kompilieren

```
./pico_c_compiler -c -t -a -s -p -v -b 100 -e 200 -d 20 -S 2 -C ./code.picoc
```

Shell

```
./pico_c_compiler
PicoC> compile -c -t -a -s -p -v -b 100 -e 200 -d 20 -S 2 "char bool_val = (12 < 1 + 2);";
PicoC> most_used "char bool_val = (12 < 1 + 2);";
```

- `compile <cli-options> "<code>";` (shortcut `cpl`, **multiline**)
- `most_used "<code>";` (shortcut `mu`, **multiline**)
- `color_toggle` (shortcut `ct`, **not** multiline), `-C` gets ignored
- `quit`: Shell verlassen

Funktionsumfang

Shell

- `←`, `→`: **Cursor** links und rechs bewegen
- `↑`, `↓`: in der **Historie** vor- und rückwärts gehen
- **Multiline Command**: mit `↩` weitere Zeile, mit `;` terminieren
- `history`: ohne Argumente Liste aller ausgeführten Commands
  - `-r <command-nr>`: command mit Nr. `<command-nr>` **ausführen**
  - `-e <command-nr>`: command mit Nr. `<command-nr>` **editieren** mit `$EDITOR`
  - `-c <command-nr>`: Historie **leeren**
  - `ctrl+r` command mit substring **suchen**
- **Config Dateien** `settings.conf` und `history.json` in `~/.config/pico_c_compiler/`
  - `color_on: True` um gleich mit angeschalteten colors zu starten

Funktionsumfang

Verwendung

- **Übersichtsseite:** https://github.com/matthejue/PicoC-Compiler
- **Help-page:** https://github.com/matthejue/PicoC-Compiler/blob/master/doc/help-page.txt
  - `pico_c_compiler -h`
  - in der **Shell**: `PicoC> help compile`

16-farbige Ausgabe

- (so gut wie) alle Terminals unterstützen **16 Farben ANSI-Escapesequenzen**
- **Windows Cmd-Terminal** wird speziell gehandelt

Bachelorabeit Themenvorschlag

Bachelorarbeit Themenvorschlag

Umfang

- ein **optimierter Compiler**, der **Graph Coloring** nutzt, um **Locations** möglichst optimal an **Register** zuzuweisen
  - es wird vermieden **Locations** zu **Stack Positionen** zuzuweisen
    - **Registerzugriffe** schneller als **Hauptspeicherzugriffe**
    - Zugriff auf Register braucht **weniger RETI-Code** (keine `push` und `pop` Operationen)
- **Web Interface**, indem man den Compiler bedienen kann

Bachelorarbeit Themenvorschlag

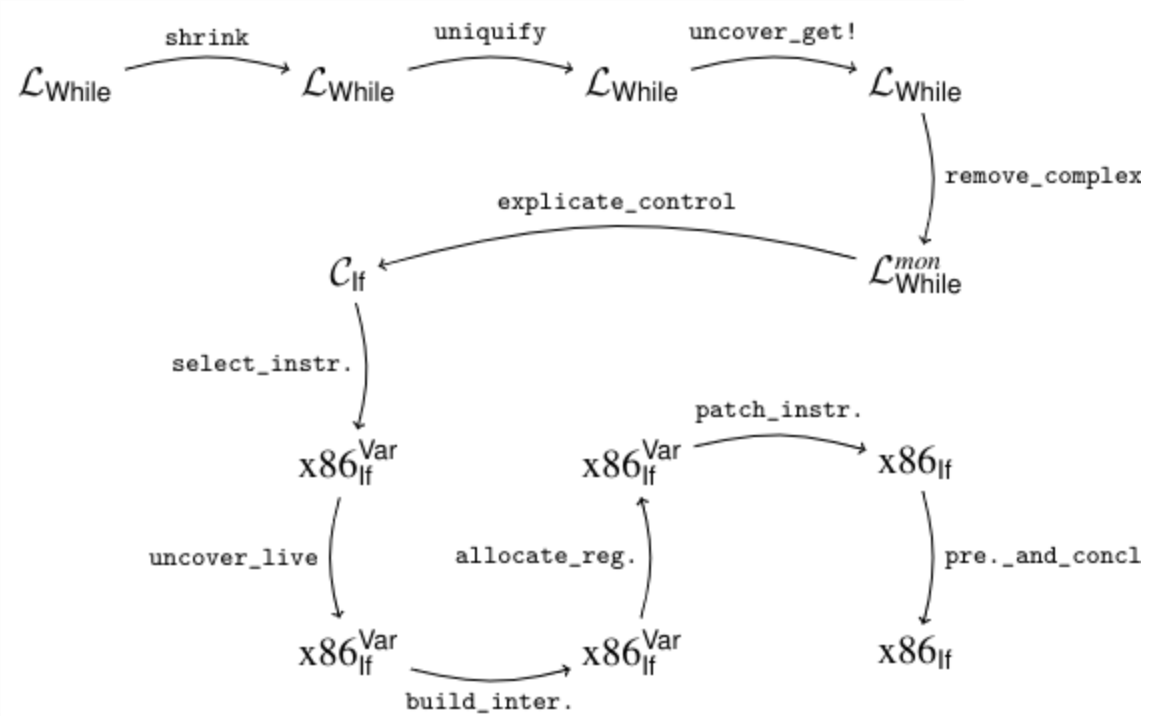
Beispiel optimierter Compiler

```
SUBI SP 1;
LOADI ACC 12;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
ADDI SP 1;
STORE ACC 100;
```

```
LOADI ACC 12;
STORE ACC 100;
```

Bachelorarbeit Themenvorschlag

Passes eines optimalen Compilers ähnlicher Funktionalität



Quellen

Quellen

Wissenquellen

- **[1]** Parr, Terence. Language implementation patterns: create your own domain-specific and general programming languages. Pragmatic Bookshelf, 2009.
- **[2]** IU-Fall-2021. “Course Webpage for Compilers (P423, P523, E313, and E513).” Accessed January 28, 2022. https://iucompilercourse.github.io/IU-Fall-2021/.

Quellen

Bildquellen

- **[3]** “Manjaro.” Accessed January 28, 2022. https://wallpapercave.com/w/wp9774690.

Quellen

Quellen des Projekts

- alle verwendete(n) Patterns, Software, Packages usw.: https://github.com/matthejue/PicoC-Compiler/blob/master/doc/references.md

Vielen Dank für eure Aufmerksamkeit!