

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

Kolloquiumspräsentation

Präsentator:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

28. September 2022

Universität Freiburg, Lehrstuhl für Betriebssysteme

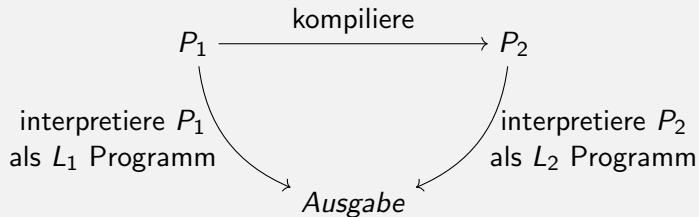
Einführung

Motivation

Compiler



- übersetzt ein Programm von einer Sprache L_1 in eine andere Sprache L_2 .
- beide Programme gleiche Semantik.



Aufgabenstellung

- L_C Programm kompilieren: `> gcc program.c -o machine_code .`

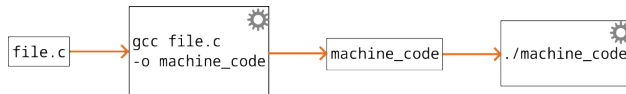


Abbildung 1: Schritte zum Ausführen eines Programmes mit dem GCC.

- L_{PicoC} Programm kompilieren: `> picoc_compiler program.picoc .`

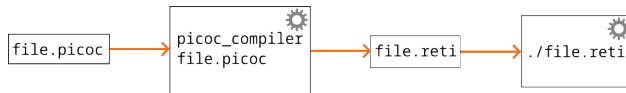


Abbildung 2: Schritte zum Ausführen eines Programmes mit dem PicoC-Compiler.

PicoC

Funktionalitäten

Die Sprache L_{PicoC} ist eine **Untermenge** der Sprache L_C , welche:

- ▶ **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/* comment */`.
- ▶ die **Basisdatentypen** `int`, `char` und `void`.
- ▶ die **Zusammengesetzten** Datentypen **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; int attr2;}`) und **Zeiger** (z.B. `int *pntr`), inklusive:
 - ▶ **Initialisierung** (z.B. `struct st st_var = {.attr1=42, .attr2={.attr={&var, &var}}}`).
 - ▶ dazugehörige **Operationen** `[i]`, `.attr`, `*` und `&`.

PicoC

Funktionalitäten

- ▶ die **Zusammengesetzten** Datentypen **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; int attr2;}`) und **Zeiger** (z.B. `int *pntr`), inklusive:
 - ▶ **Kombinationen** der eben genannten **Operationen** (z.B. `(*complex_var[0][1])[1].attr`) und **Datentypen** (z.B. `struct st (*complex_var[1][2])[2]`).
 - ▶ **Zeigerarithmetik** (z.B. `*(var + 2)`).
- ▶ `if(cond){ }-` und `else{ }-`**Anweisungen**, inklusive:
 - ▶ Kombination von `if` und `else`, nämlich `else if(cond){ }`.
- ▶ `while(cond){ }-` und `do while(cond){ };`**Anweisungen**.

PicoC

Funktionalitäten

- ▶ **Arihmetische und Bitweise Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` und **unären Operatoren** `-`, `~` umgesetzt sind.
- ▶ **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.
- ▶ **Zuweisungen**, welche mithilfe des **Zuweisungsoperators** `=` umgesetzt sind, inklusive:
 - ▶ Zuweisung an **Feldelement**, **Verbundsattribut** oder **Zeigerelement** (z.B. `(*var.attr)[2] = fun() + 42`).

PicoC

Funktionalitäten

- ▶ **Funktionsdefinitionen** (z.B. `int fun(int arg1[3], struct st arg2){}`), inklusive:
 - ▶ **Funktionsdeklarationen** (z.B. `int fun(int arg1[3], struct st arg2);`).
 - ▶ **Funktionsaufrufe** (z.B. `fun(ar, st_var)`)
 - ▶ **Sichtbarkeitsbereiche** innerhalb der Codeblöcke `{}` der Funktionen.
 - ▶ **Argumentübergabe** erfolgt ausschließlich über die **Call-by-Value** Strategie.
 - ▶ bei **Feldern** wird ein **Zeiger** in den Stackframe der **aufrufenden Funktion** geschrieben.
 - ▶ bei **Verbunden** wird der **komplette Verbund** in den Stackframe der **aufrufenden Funktion** **kopiert**.

PicoC

Sonstiges

- ▶ Implementierung ist aufgebaut auf **RETI-Codeschnipseln** aus der Vorlesung Scholl, „**Betriebssysteme**“, Kapitel 3 Übersetzung höherer Programmiersprachen in Maschinensprache.
 - ▶ bei **Inkonsistenzen** und **Umstimmigkeiten** angepasst.
- ▶ im Appendix ab Folie **61** weitere Informationen.

RETI-Architektur

- **32-Bit Architektur**, die in den Vorlesungen Scholl, „Betriebssysteme“ und Scholl, „Technische Informatik“ zu Lernzwecken eingesetzt wird. Basis für L_{RETI} .

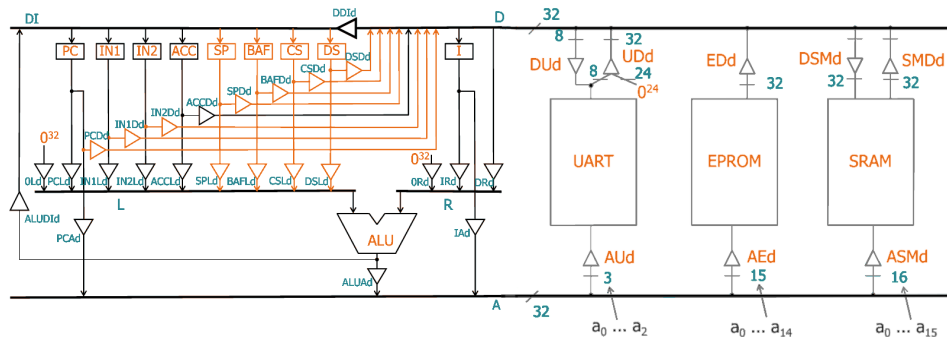


Abbildung 3: Datenpfade der RETI-Architektur, nicht selbst erstellt, leicht abgeändert.

RETI-Architektur

Speicherorganisation

- Register haben bestimmte Aufgaben bei der Umsetzung von Prozessen.

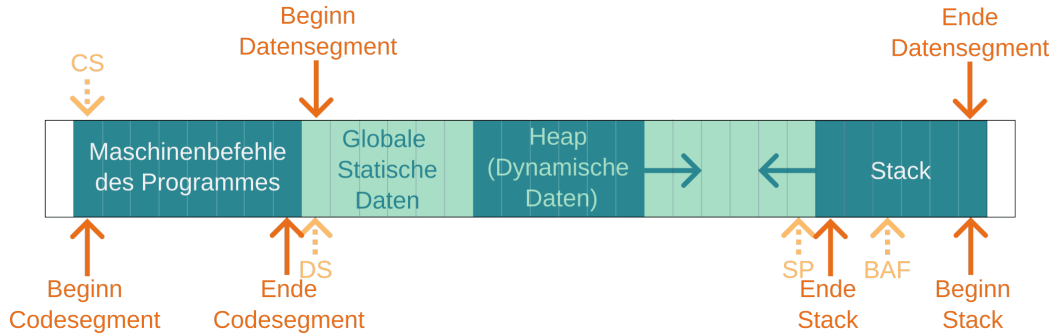


Abbildung 4: Speicherorganisation.

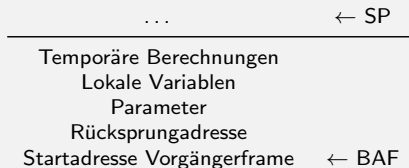
Speicherorganisation

Lokale Variablen und Parameter von Funktionen

- ▶ alle Funktionen, **außer** der `main`-Funktion besitzen einen **Stackframe** für **Lokale Variablen** und **Parameter**.
- ▶ **Globale Statische Daten** sind **Globale Variablen**, sowie **Lokale Variablen** und **Parameter** der `main`-Funktion.

Stackframe

- ▶ Datenstruktur, um **Zustand** einer Funktion zur **Laufzeit** zu „konservieren“.



Implementierung

Lexikalische Analyse

Aufgabe

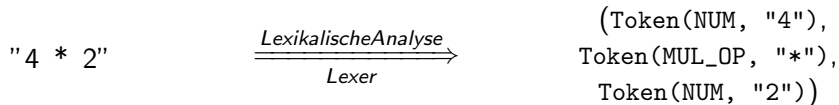


Abbildung 5: Aus Eingabewort Tokens generieren.

- ▶ im Appendix ab Folie 68 genauer erklärt.

Syntaktische Analyse

Aufgabe



Abbildung 6: Aus Tokens einen Abstrakten Syntaxbaum generieren.

- ▶ **Parser** generiert im Zusammenspiel mit **Lexer** den **Ableitungsbaum**.
- ▶ **Visitor** vereinfacht den **Ableitungsbaum**.
- ▶ **Transformer** generiert den **Abstrakten Syntaxbaum**.
- ▶ im Appendix ab Folie 71 genauer erklärt.

Syntaktische Analyse

Zwischenschritte

"4 * 2"

Parser
 $\xrightarrow{\hspace{1cm}}$
Lexer

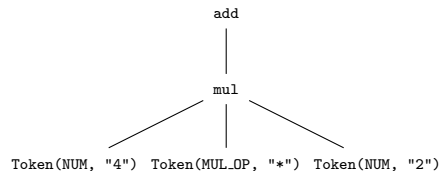
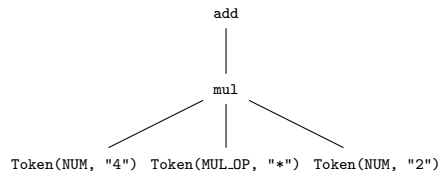


Abbildung 7: Aus dem Eingebewort einen Ableitungsbaum generieren.



Visitor
 $\xrightarrow{\hspace{1cm}}$
Transformer

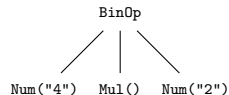


Abbildung 8: Aus Ableitungsbaum Abstrakten Syntaxbaum generieren.

Syntaktische Analyse

Lark Parsing Toolkit

- ▶ erleichtert **Lexikalische Analyse** und **Syntaktische Analyse**.
- ▶ **Basic Lexer**, **Earley Parser**, **Visitor** und **Transformer** implementiert.

NUM	::=	"4" "2"	<i>L_Lex</i>
ADD_OP	::=	"+"	
MUL_OP	::=	"*"	
mul	::=	<i>mul MUL_OP NUM</i> <i>NUM</i>	<i>L_Parse</i>
add	::=	<i>add ADD_OP mul</i> <i>mul</i>	

Grammatik 1: Grammatik für Lexer oben und Grammatik für Parser unten.

Code Generierung

Aufgabe

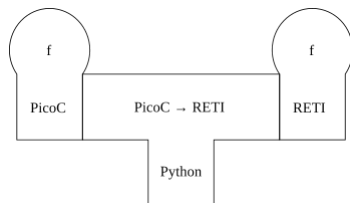


Abbildung 9: T-Diagramm für die Aufgabe der Code Generierung.

- ▶ Abstrakter Syntaxbaum für Sprache L_{PicoC} soll zu Abstraktem Syntaxbaum der Sprache L_{RETI} umgeformt werden.
- ▶ mit Passes kleinschrittig immer mehr der Syntax der Maschinensprache annähern.

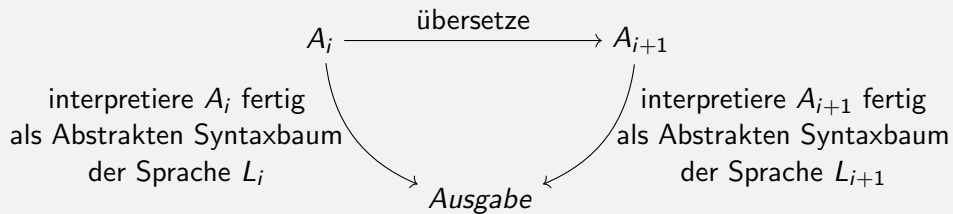
Code Generierung

Definitionen

Pass



- Übersetzungsschritt eines Abstrakten Syntaxbaumes von L_i zu L_{i+1}
- beide Abstrakten Syntaxbäume gleiche Semantik
- übernimmt Teilaufgabe, keine Überschneidung



Code Generierung

Überblick über Passes des PicoC-Compilers

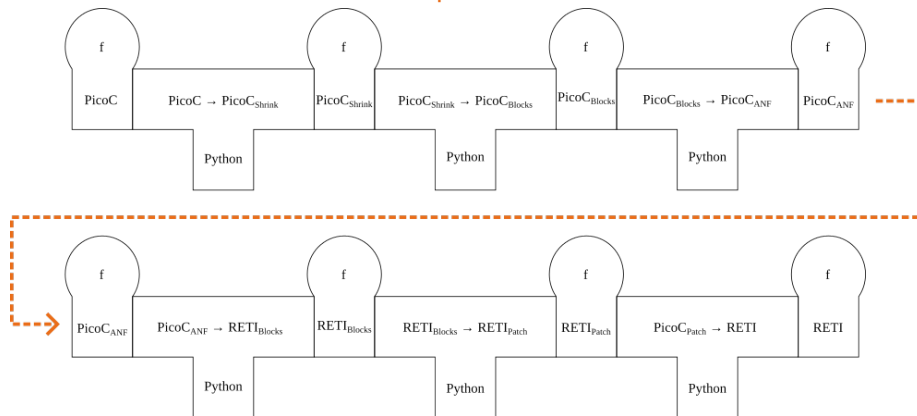
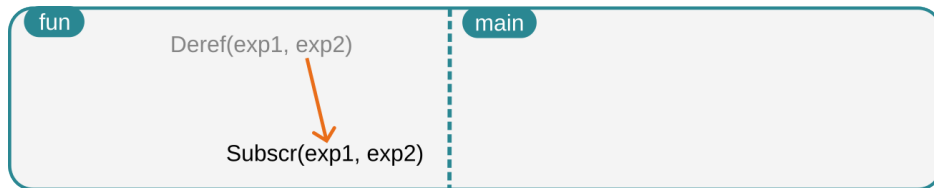


Abbildung 10: Architektur mit allen Passes ausgeschrieben.

Code Generierung

PicoC-Shrink Pass

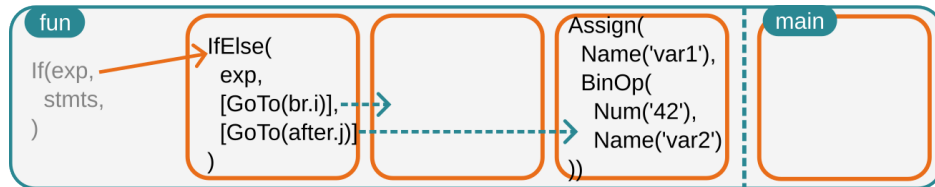
- ▶ gleiche Semantik des Dereferenzierungsoperators `*(pntr_or_ar + i)` und des Operators für Indexzugriff auf ein Feld `pntr_or_ar[i]`, sind austauschbar.
- ▶ Ersetzen von `Deref(exp, i)` durch `Subscr(exp, i)`.
- ▶ Dereferenzierung `*(pntr_or_ar + i)` wird von den Routinen für einen Indexzugriff auf ein Feld `pntr_or_ar[i]` übernommen \Rightarrow kein redundanter Code.



Code Generierung

PicoC-Blocks Pass

- ▶ If(exp, stmts), IfElse(exp, stmts1, stmts2), While(exp, stmts) und DoWhile(exp, stmts) durch Block(name, stmts_instrs-, GoTo(lable)- und IfElse(exp, stmts1, stmts2) ersetzt.
- ▶ im Appendix ab Folie 84 genauer erklärt.



Code Generierung

PicoC-ANF Pass

- ▶ formt **Abstrakten Syntaxbaum** um, sodass er die **Syntax** der Sprache L_{PicoC_ANF} erfüllt, deren Grammatik in **A-Normalform** ist.
- ▶ **Funktionen** werden aufgelöst.
- ▶ im Appendix ab Folie 86 genauer erklärt.

```
Exp(Global(Num('addr1'))),  
IfElse(Stack(Num('1'),  
  [GoTo(br.i)],  
  [GoTo(after.j)])  
)
```

```
Exp(Num('42')),  
Exp(Global(Num('addr1'))),  
BinOp(Stack(Num('2')), Mul(), Stack(Num('1'))),  
Assign(Global(Num('addr2')), Stack(Num('1'))),
```

PicoC-ANF Pass

A-Normalform

- ▶ **Zweck:** Maschinenbefehlen annähern, die meist nur eine Aktion ausführen. Eine Anweisung wird aufgespalten, wenn sie mehreren Aktionen entspricht. Nebeneffekte, die den Kompiliervorgang beeinflussen werden isoliert.
- ▶ im Appendix ab Folie 90 genauer erklärt.

ziehe **Komplexe Ausdrücke** aus
Anweisungen und Ausdrücken **vor**

Code

```
void main() {  
    int x = 1 - 5 * 4;  
}
```

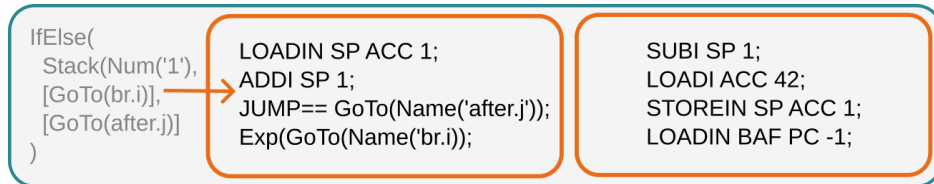
Code in A-Normalform

```
void main() {  
    int x; // allocate at address  
    ↪ 0 relative to DS Register  
    1;  
    5;  
    4;  
    stack(2) * stack(1);  
    stack(2) - stack(1);  
    global(0) = stack(1);  
}
```


Code Generierung

RETI-Blocks Pass

- **PicoC-Knoten**, die **Anweisungen** darstellen, werden durch **semantisch** entsprechende **RETI-Knoten**, die **Befehle** darstellen ersetzt.



Code Generierung

RETI-Patch Pass

- ▶ **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes** durch:
 - ▶ **Einfügen** eines `GoTo(Name('main'))` in den `global.<number>`-Block, wenn `main`-Funktion **nicht** die **erste Funktion** ist.
 - ▶ **Entfernen** von `GoTo()`s, deren Sprung nur **eine** Adresse weiterspringt.
 - ▶ weitere Aufgaben im Appendix ab Folie 96 aufgezählt.

```
Exp( global
  GoTo(Name(
    'main.k')
  )
)
```


```
LOADIN SP ACC 1;
ADDI SP 1;
JUMP== GoTo(Name('after.j'));
Exp(GoTo(Name('br.i'));
```

```
LOADIN SP ACC 1;
ADDI SP 1;
JUMP== GoTo(Name('after.j'));
# // not included
```

Code Generierung

RETI Pass

- ▶ verbliebene **PicoC-Knoten** werden durch entsprechende **RETI-Knoten** ersetzt:
 - ▶ keine **Blöcke** mehr, Knoten genauso **zusammengefügt**, wie sie in diesen **angeordnet** waren.
 - ▶ `GoTo(Name(str))` werden durch einen **Immediate** mit passender **Distanz** / **Adresse** oder einen **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))` ersetzt.

<pre>Exp(GoTo(Name('main.k'))) JUMP <distance-to-main.k></pre>	<pre>LOADIN SP ACC 1; ADDI SP 1; JUMP== GoTo(Name('after.j')); # // not included</pre>		<pre>LOADIN SP ACC 1; ADDI SP 1; JUMP== <distance-to-after.j>; # // not included</pre>
--	--	--	--

Code Generierung

Codebeispiel

- ▶ im Appendix ab Folie 118 sind Tokens, Ableitungsbaum, Vereinfachter Ableitungsbaum, Abstrakter Syntaxbaum und die modifizierten Abstrakten Synntaxbäume der verschiedenen Passes an einem Codebeispiel erklärt.
- ▶ `> make test TESTNAME=example_presentation VERBOSE=-v`

Code Generierung

Zugriff auf Zusammengesetzte Datentypen

```

1 // in:1
2 // expected:42
3 // datasegment:36
4
5 struct stt {int attr1; int attr2[2];};
6
7 struct stt ar_of_sts[3][2];
8
9 int fun(struct stt (*param)[3][2]){
10     ((*param+2))[1].attr2[input()] = 42;
11     return 1;
12 }
13
14 void main() {
15     struct stt (*pntr_on_ar_of_sts)[3][2] = &ar_of_sts;
16     int res = fun(pntr_on_ar_of_sts);
17     if (res) {
18         print(((*pntr_on_ar_of_sts+2))[1].attr2[1]);
19     }
20 }

```

Datentyp der Variable

```

// ...
struct stt
↪ (*pntr_on_ar_of_sts)[3][2] =
↪ &ar_of_sts;
// ...

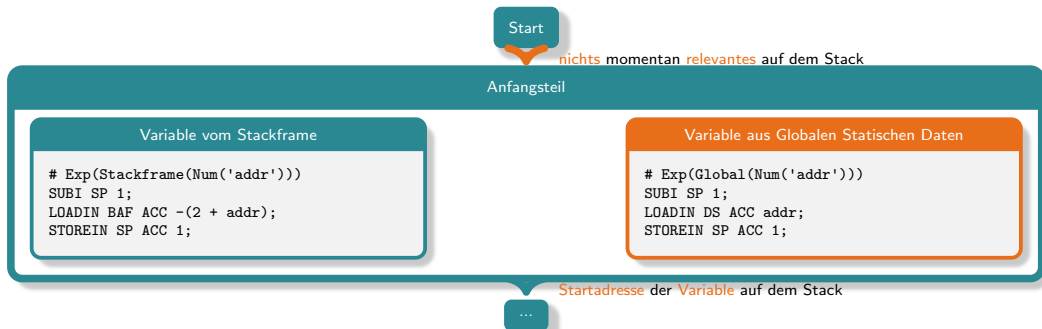
```

- ▶ ein „Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs stt“.
- ▶ *Clockwise/Spiral Rule*

Code Generierung

Codebeispiel

► `(*(*pntr_on_ar_of_sts+2))[1].attr2[1]`



► `(*(*pntr_on_ar_of_sts+2))[1].attr2[1]`  `struct stt (*pntr_on_ar_of_sts)[3][2]...`

 Startadresse der Variable auf dem Stack

Mittelteil

Feldindexzugriff auf Feld

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN1 2;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{dim}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Feldindexzugriff auf Zeiger

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN2 2;
LOADIN IN2 IN1 0;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{dim}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Verbundsattribut-zugriff auf Verbund

```
# Ref(Attr(Stack(Num('1')),
↪ Name('attr')))
LOADIN SP IN1 1;
ADDI IN1  $\sum_{k=1}^{\text{idx}-1} \text{size}(\text{datatype}_{1,k})$ ;
STOREIN SP IN1 1;
```

*1

 *1: Startadresse eines Zeigerelementes, Feldelementes oder Verbundsattributes auf dem Stack

► `(*(*pntr_on_ar_of_sts+2))[1].attr2[1]`  `struct stt (*pntr_on_ar_of_sts)[3][2]...`

Startadresse der Variable auf dem Stack

Mittelteil

Feldindexzugriff auf Feld

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↳ Stack(Num('1'))))
LOADIN SP IN1 2;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Feldindexzugriff auf Zeiger

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↳ Stack(Num('1'))))
LOADIN SP IN2 2;
LOADIN IN2 IN1 0;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Verbundsattribut-zugriff auf Verbund

```
# Ref(Attr(Stack(Num('1')),
↳ Name('attr')))
LOADIN SP IN1 1;
ADDI IN1  $\sum_{k=1}^{\text{idx}-1} \text{size}(\text{datatype}_{1,k})$ ;
STOREIN SP IN1 1;
```

*1

*1: Startadresse eines Zeigerelementes, Feldelementes
oder Verbundsattributes auf dem Stack

► `(*(pntr_on_ar_of_sts+2))[1].attr2[1]`  `struct stt (pntr_on_ar_of_sts)[3][2]...`

Startadresse der Variable auf dem Stack

Mittelteil

Feldindexzugriff auf Feld

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN1 2;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Feldindexzugriff auf Zeiger

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN2 2;
LOADIN IN2 IN1 0;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Verbundsattribut-zugriff auf Verbund

```
# Ref(Attr(Stack(Num('1')),
↪ Name('attr')))
LOADIN SP IN1 1;
ADDI IN1  $\sum_{k=1}^{\text{idx}-1} \text{size}(\text{datatype}_{1,k})$ ;
STOREIN SP IN1 1;
```

*1

*1: Startadresse eines Zeigerelementes, Feldelementes
oder Verbundsattributes auf dem Stack

► `(*(pntr_on_ar_of_sts+2))[1].attr2[1]` `struct stt (pntr_on_ar_of_sts)[3][2]...`

Startadresse der Variable auf dem Stack

Mittelteil

Feldindexzugriff auf Feld

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN1 2;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Feldindexzugriff auf Zeiger

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN2 2;
LOADIN IN2 IN1 0;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Verbundsattribut-zugriff auf Verbund

```
# Ref(Attr(Stack(Num('1')),
↪ Name('attr')))
LOADIN SP IN1 1;
ADDI IN1  $\sum_{k=1}^{\text{idx}-1} \text{size}(\text{datatype}_{1,k})$ ;
STOREIN SP IN1 1;
```

*1

*1: Startadresse eines Zeigerelementes, Feldelementes
oder Verbundsattributes auf dem Stack

► `(*(*pntr_on_ar_of_sts+2))[1].attr2[1],` `struct stt {int attr1; int attr2[2];};`



Startadresse der Variable auf dem Stack

Mittelteil

Feldindexzugriff auf Feld

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN1 2;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Feldindexzugriff auf Zeiger

```
# z.B. Exp(Num('idx'))
SUBI SP 1;
LOADI ACC idx;
STOREIN SP ACC 1;
# Ref(Subscr(Stack(Num('2')),
↪ Stack(Num('1'))))
LOADIN SP IN2 2;
LOADIN IN2 IN1 0;
LOADIN SP IN2 1;
MULTI IN2  $(\prod_{j=i+1}^n \text{din}_j) \cdot \text{size}(\text{datatype})$ ;
ADD IN1 IN2;
ADDI SP 1;
STOREIN SP IN1 1;
```

Verbundsattribut-zugriff auf Verbund

```
# Ref(Attr(Stack(Num('1')),
↪ Name('attr')))
LOADIN SP IN1 1;
ADDI IN1  $\sum_{k=1}^{\text{idx}-1} \text{size}(\text{datatype}_{1,k})$ ;
STOREIN SP IN1 1;
```



*1

*1: Startadresse eines Zeigerelementes, Feldelementes
oder Verbundsattributes auf dem Stack



► `(*(pnter_on_ar_of_sts+2))[1].attr2[1],` `struct stt {int attr1; int attr2[2];};`



Startadresse eines Zeigerelementes, Feldelementes
oder Verbundattributes auf dem Stack

Schlussstil

Letzter Datentyp ist Verbund,
Zeiger oder Basisdatentyp

```
# Exp(Stack(Num('1')))
LOADIN SP IN1 1;
LOADIN IN1 ACC 0;
STOREIN SP ACC 1;
```

Letzter Datentyp ist Feld

```
# not included Exp(Stack(Num('1')))
```

Inhalt der Speicherzelle an der berechneten
Adresse oder die berechnete Adresse selbst

Ende

Fehlermeldungen

Kategorien

- ▶ UnexpectedCharacter
- ▶ UnexpectedToken
- ▶ UnexpectedEOF
- ▶ DivisionByZero
- ▶ UnknownIdentifier
- ▶ UnknownAttribute
- ▶ ReDeclarationOrDefinition
- ▶ TooLargeLiteral
- ▶ NoMainFunction
- ▶ ConstAssign
- ▶ DatatypeMismatch
- ▶ PrototypeMismatch
- ▶ ArgumentMismatch
- ▶ WrongNumberArguments
- ▶ WrongReturnType
- ▶ im Appendix ab Folie 101 mit Erklärung.

Qualitätssicherung

Tests

Typischer Test

```
// in:21 2 6 7
// expected:42 42
// datasegment:4
```

```
void main() {
    print(input() * input());
    print(input() * input());
}
```

convert_to_c.py →

```
#include<stdio.h>
```

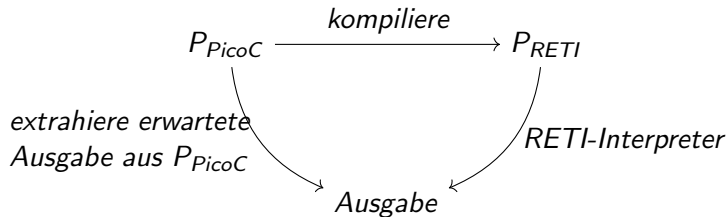
```
void main() {
    printf(" %d", 21 * 2);
    printf(" %d", 6 * 7);
}
```

- ▶ `// in:<space-sep-values>` sind **Eingaben** für die `input()`-Anweisungen.
- ▶ `// expected:<space-sep-values>` sind die **Erwarteten Ausgaben** der `print(exp)`-Anweisungen.
- ▶ `// datasegment:<size>` ist die **optionale Datensegmentgröße**.
- ▶ `convert_to_c.py`: **jedes** `print(exp)` wird durch `printf("%d", exp)` und **jedes** `input()` wird der Reihenfolge nach durch die **Eingaben** ersetzt.

Tests

Ablauf

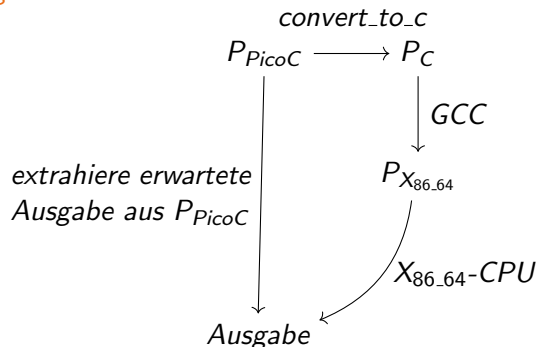
- ▶ prüfen, ob Erwartete Ausgaben und Ausgaben der `print(exp)`-Anweisungen identisch sind.
- ▶ Schreiber der Tests = Implementierer des PicoC-Compiler \Rightarrow Tests bestätigen nur das PicoC-Compiler genauso implementiert, wie diese Person die Semantik von L_{PicoC} interpretiert hat.



Tests

Ablauf

- ▶ der **GCC** setzt die **Semantik** von L_{PicoC} sehr wahrscheinlich korrekt um.
- ▶ prüfen, ob **Erwartete Ausgaben** und **Ausgaben der** `printf("%d", exp)-Anweisungen` identisch sind.



Tests

Durchlauf aller Tests

- `> make test <more-options>` (siehe Appendix auf Folie 116)

```
> make test
=====
= ./tests/basic_array_init.picoc =
=====
...
=====
=          Verification          =
=====
./tests/basic_array_init.c
...
=====
=          Results              =
=====
Verified: 104 / 104
Not verified:
Running through: 180 / 180
Not running through:
Passed: 180 / 180
Not passed:
```

Tests

Sonstiges

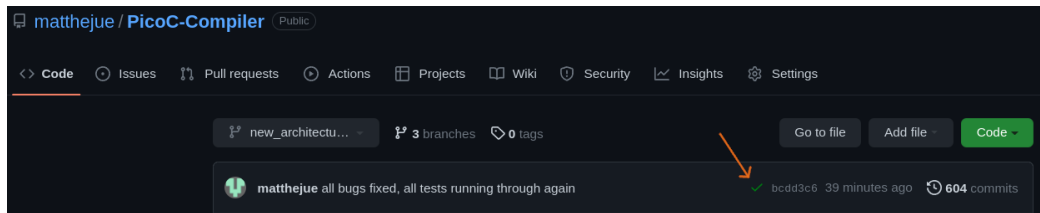


Abbildung 11: Autotesting mit GitHub Actions.

- ▶ <https://github.com/matthejue/PicoC-Compiler>.
- ▶ im Appendix auf Folie 113 sind die verschiedenen **Bedienmöglichkeiten** zum Ausführen der Tests erklärt.
- ▶ im Appendix auf Folie 117 sind die verschiedenen **Testkategorien** mit Erklärung zu finden.

Vorführung

Bedienung

Wichtigste Funktionalitäten

- ▶ **Kompilieren:** `> picoc_compiler <cli-options> program.picoc .`
 - ▶ alle `<cli-options>` im Appendix ab Folie 104 aufgelistet.
- ▶ **Kompilieren + Interpretieren:**
`> picoc_compiler <cli-options> -R program.picoc .`
- ▶ **Shell-Mode:** `> picoc_compiler` ohne Argumente.
 - ▶ alle Befehle des Shell-Modes im Appendix ab Folie 108 aufgelistet.
- ▶ **Show-Mode:**
 - ▶ **Bedienung** des Show-Modes im Appendix ab Folie 111 erklärt.

Bedienung

Shell-Mode

```
> picoc_compiler
PicoC Shell. Enter `help` (shortcut `?`) to see the manual.
PicoC> cpl "6 * 7;";
----- RETI -----

SUBI SP 1;
LOADI ACC 6;
STOREIN SP ACC 1;
SUBI SP 1;
LOADI ACC 7;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
LOADIN BAF PC -1;

Compilation successfull

PicoC> quit
```

Code 1: Shell-Mode und die Befehle `compile` und `quit`.

Bedienung

Shell-Mode

```
PicoC> mu "int var = 42;";
----- Code -----
// stdin.picoc:
void main() {int var = 42;}
----- Tokens -----
...
----- Abstract Syntax Tree -----
...
----- PicoC Shrink -----
...
----- RETI -----
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
LOADIN BAF PC -1;
----- RETI Run -----
...

Compilation successfull
```

Code 2: Shell-Mode und der Befehl most used.

Codebeispiel 1

Finbonacci

```
1 // in:10
2 // expected:55
3 // datasegment:64
4 // from the Operating Systems Lecture by Prof. Dr. Christoph Scholl
5
6 int ar[11] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
7
8 int fib_efficient(int n, int* res){
9     if (n == 0)
10         return 0;
11     else if (n == 1){
12         res[0] = 0;
13         res[1] = 1;
14         return 1;
15     }
16     res[n] = fib_efficient(n - 1, res) + res[n - 2];
17     return res[n];
18 }
19
20 void main() {
21     print(fib_efficient(input(), ar));
22 }
```

Codebeispiel 2

Bubble Sort

```
1 // in:0 5
2 // expected:-2 314
3 // based on a function from the Operating Systems Lecture by Prof. Dr. Christoph Scholl and
  ↳ https://de.wikipedia.org/wiki/Bubblesort
4
5 struct stt {int len; int *ar;};
6
7 int ar[6] = {314, 42, 4, 42, -2, 5};
8
9 struct stt st_ar = {.len=6, .ar=ar};
10
11 int swap(int *x, int *y) {
12     // in the lecture this function is called pairsort
13     int h;
14     int swapped = 0;
15     if (*x > *y) {
16         h = *x; *x = *y; *y = h; swapped = 1;
17     }
18     return swapped;
19 }
```

Codebeispiel 2

Bubble Sort, Teil 2

```
1 void main() {
2     int swapped;
3     int i;
4     int n = st_ar.len-1;
5     do {
6         i = 0;
7         while (i < n) {
8             swapped = swap(&st_ar.ar[i], &st_ar.ar[i+1]);
9             i = i + 1;
10        }
11        n = n - 1;
12    } while(swapped);
13    print(st_ar.ar[input()]);
14    print(st_ar.ar[input()]);
15 }
```

Codebeispiel 3

Min Sort

```
1 // in:
2 // expected:-2 4 5 42 42 314
3 // from the Algorithms and Datastructures Lecture by Prof. Dr. Bast
4
5 struct stt {int len; int *ar;};
6
7 void min_sort(int *ar, int len) {
8     int i = 0;
9     int j;
10    int minimum;
11    int minimum_index;
12    int tmp;
13    while (i < len) {
14        minimum = ar[i];
15        minimum_index = i;
16        j = i + 1;
17        while (j < len) {
18            if (ar[j] < minimum) {
19                minimum = ar[j];
20                minimum_index = j;
21            }
22            j = j + 1;
23        }
```

Codebeispiel 3

Min Sort, Teil 2

```
1     tmp = ar[i];
2     ar[i] = ar[minimum_index];
3     ar[minimum_index] = tmp;
4     i = i + 1;
5 }
6 }
7
8 void main() {
9     int len = 6;
10    int ar[6] = {314, 42, 4, 42, -2, 5};
11    min_sort(ar, len);
12    print(ar[0]);
13    print(ar[1]);
14    print(ar[2]);
15    print(ar[3]);
16    print(ar[4]);
17    print(ar[5]);
18 }
```

Codebeispiel 4

Fakultät

```
1 // in:3 4
2 // expected:6 24
3 // from the Operating Systems Lecture by Prof. Dr. Christoph Scholl
4
5 int fakul(int n) {
6     int res_f; int h;
7     if (n == 1) {
8         res_f = 1;
9     } else {
10         h = fakul(n-1);
11         res_f = n * h;
12     }
13     return res_f;
14 }
15
16 void main() {
17     int res;
18     print(fakul(input()));
19     res = fakul(input());
20     print(res);
21 }
```

Codebeispiel 5

Binary Search

```
1 // in:41 42
2 // expected:-1 5
3 // datasegment:64
4 // from the Introduction to Programming Lecture by Peter Thiemann
5
6 struct ar_with_lent {int len; int *ar;};
7
8 int ar[10] = {1, 3, 4, 7, 19, 42, 128, 314, 512, 1024};
9
10 struct ar_with_lent ar_with_len = {.len=10, .ar=ar};
11
12 int bsearch_rec(int *ar, int key, int lo, int hi) {
13     int m;
14     if (lo == hi)
15         return -1; // key not in empty segment
16     m = (lo + hi) / 2; // position of root
17     if (ar[m] == key)
18         return m;
19     else if (ar[m] > key)
20         return bsearch_rec(ar, key, lo, m);
21     else // ar[m] < key
22         return bsearch_rec(ar, key, m+1, hi);
23 }
```

Codebeispiel 5

Binary Search, Teil 2

```
1 void main() {  
2     print(bsearch_rec(ar_with_len.ar, input(), 0, ar_with_len.len - 1));  
3     print(bsearch_rec(ar_with_len.ar, input(), 0, ar_with_len.len - 1));  
4 }
```


Codebeispiel 6

Primzahlen bis Zahl n

```
1 // in:30
2 // expected:2 3 5 7 11 13 17 19 23 29
3 // from the Introduction to Programming Lecture by Peter Thiemann
4
5 int ar[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
6 int len = 10;
7
8 void primes(int *primes, int n) {
9     int i = 3;
10    int j;
11    int idx = 1;
12    char undividable = 1;
13    if (n <= 1)
14        return;
15    primes[0] = 2;
16
17    while (i <= n) {
18        j = 0;
19        while (j < idx) {
20            if (i % primes[j] == 0) {
21                undividable = 0;
22            }
23            j = j + 1;
24        }
```

Codebeispiel 6

Primzahlen bis Zahl n, Teil 2

```
1     if (undividable) {
2         primes[idx] = i;
3         idx = idx + 1;
4     }
5     undividable = 1;
6     i = i + 1;
7 }
8 }
9
10 void main() {
11     int i = 0;
12     primes(ar, input());
13     while (i < len) {
14         print(ar[i]);
15         i = i + 1;
16     }
17 }
```

Bedienung

Tutorials und Dokumentation



Abbildung 13: README.md des PicoC-Compilers Repositories.

- https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/getting_started.md.

Appendix

Schwerpunkte

- ▶ **Syntax** und **Semantik** der Sprache L_{PicoC} **identisch** zur Sprache L_C .
 - ▶ außer bei Kommandozeilenoptionen, Fehlermeldungen usw. **kein Unterschied** zu z.B. dem **GCC**.
- ▶ möglichst die **RETI-Codeschnipsel** aus der Vorlesung Scholl, „Betriebssysteme“, Kapitel 3 Übersetzung höherer Programmiersprachen in Maschinensprache.
 - ▶ bei **Inkonsistenzen** und **Umstimmigkeiten** angepasst.

PicoC

Grammatik

- ▶ https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/src/concrete_syntax_picoc.lark

RETI-Architektur

Grammatik

dig_no_0	::=	"1" "2" "3" "4" "5" "6"	<i>L_Program</i>
		"7" "8" "9"	
dig_with_0	::=	"0" <i>dig_no_0</i>	
num	::=	"0" <i>dig_no_0</i> <i>dig_with_0</i> * "—" <i>dig_no_0</i> *	
letter	::=	"a" ... "Z"	
name	::=	<i>letter</i> (<i>letter</i> <i>dig_with_0</i> _)*	
reg	::=	"ACC" "IN1" "IN2" "PC" "SP"	
		"BAF" "CS" "DS"	
arg	::=	<i>reg</i> <i>num</i>	
rel	::=	"==" "!=" "<" "<=" ">"	
		">=" "_NOP"	

Grammatik 2: Grammatik des Lexers für die Sprache L_{RETI} in EBNF.

RETI-Architektur

Grammatik

instr	::=	"ADD" reg arg "ADDI" reg num "SUB" reg arg	<i>L_Program</i>
		"SUBI" reg num "MULT" reg arg "MULTI" reg num	
		"DIV" reg arg "DIVI" reg num "MOD" reg arg	
		"MODI" reg num "OPLUS" reg arg "OPLUSI" reg num	
		"OR" reg arg "ORI" reg num	
		"AND" reg arg "ANDI" reg num	
		"LOAD" reg num "LOADIN" arg arg num	
		"LOADI" reg num	
		"STORE" reg num "STOREIN" arg argnum	
		"MOVE" reg reg	
		"JUMP" rel num INT num RTI	
		"CALL" "INPUT" reg "CALL" "PRINT" reg	
program	::=	(instr";")*	

Grammatik 3: Grammatik des Parsers für die Sprache L_{RETI} in EBNF.

PicoC

Definitionen

Call-by-Value

- ▶ **Kopie** des **Arguments** wird im Stackframe der aufgerufenen Funktion an **Parameter** gebunden.
- ▶ **Argument** bleibt bei **Änderungen** am entsprechenden **Parameter** in der aufgerufenen Funktion in der aufrufenden Funktion **unverändert**.

Call-by-Reference

- ▶ **Referenz** des **Arguments** wird im Stackframe der aufgerufenen Funktion an **Parameter** gebunden.
- ▶ **Argument** ändert sich bei **Änderungen** am entsprechenden **Parameter** in der aufgerufenen Funktion auch in der aufrufenden Funktion .

Weitere Definitionen

Interpreter

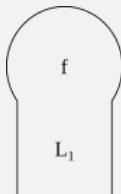


- ▶ führt Anweisungen „direkt“ aus.

T-Diagramm Programm



- ▶ in der Sprache L_1 geschrieben und berechnet Funktion f .

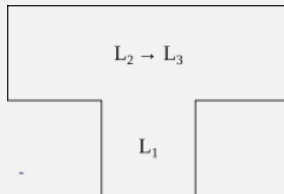


Weitere Definitionen

T-Diagramm Übersetzer



- ▶ in der Sprache L_1 geschrieben und übersetzt von Sprache L_2 in die Sprache L_3 .
- ▶ gleiche Semantik.
- ▶ Kompilieren ist immer Übersetzen, aber Übersetzen ist nicht immer Kompilieren.



Lexikalische Analyse

Aufgabe

NUM	::=	"4" "2"	<i>L_Lex</i>
ADD_OP	::=	"+"	
MUL_OP	::=	"*"	

Grammatik 4: Grammatik des Lexers

"4 * 2"	$\xrightarrow[\text{Lexer}]{\text{LexikalischeAnalyse}}$	(Token(NUM, "4"), Token(MUL_OP, "*"), Token(NUM, "2"))
---------	--	--

Abbildung 14: Aus Eingabewort Tokens generieren.

Lexikalische Analyse

Definitionen

Token



- ▶ Tupel (T, V) , wobei:
 - ▶ Tokentyp $T \hat{=}$
 - ▶ bestimmtes Nicht-Terminalsymbol auf der linken Seite des $::=$ -Symbols in der Grammatik des Lexers.
 - ▶ Überbegriff für möglicherweise unendliche Menge von Tokenwerten, die sich aus einem bestimmten Nicht-Terminalsymbol ableiten lassen.
 - ▶ in der Grammatik des Parsers ein Terminalsymbol.
 - ▶ Tokenwert $V \hat{=}$ aus einem bestimmten Nicht-Terminalsymbol ableitbares Wort in der Grammatik des Lexers.

Lexikalische Analyse

Definitionen

NUM	::=	"4" "2"	L_{Lex}
ADD_OP	::=	"+"	
MUL_OP	::=	"*"	

Grammatik 5: Grammatik des Lexers in EBNF

Lexer

- ▶ bildet **Eingabewort** $w \in \Sigma^*$ auf **Folge von Tokens** $(t_1, v_n) \dots (t_n, v_n) \in (T \times V)^*$ ab: $lex : \Sigma^* \rightarrow (T \times V)^*, w \mapsto (t_1, v_1) \dots (t_n, v_n)$.

Syntaktische Analyse

Ausgelassene Zwischenschritte

NUM	::=	"4" "2"	<i>L_Lex</i>
ADD_OP	::=	"+"	
MUL_OP	::=	"*"	
mul	::=	<i>mul MUL_OP NUM</i> <i>NUM</i>	<i>L_Parse</i>
add	::=	<i>add ADD_OP mul</i> <i>mul</i>	

Grammatik 6: Grammatik des Parsers unten und Grammatik des Lexers oben

► Tokentypen *T* sind in der Grammatik des Parsers **Terminalsymbole**

add \Rightarrow *mul* \Rightarrow *mul MUL_OP NUM* \Rightarrow *NUM MUL_OP NUM* \Rightarrow * "4" "*" "2"

Syntaktische Analyse

Ausgelassene Zwischenschritte

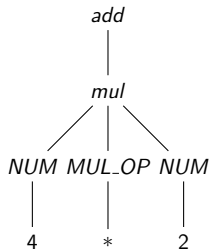


Abbildung 15: Formaler Ableitungsbaum

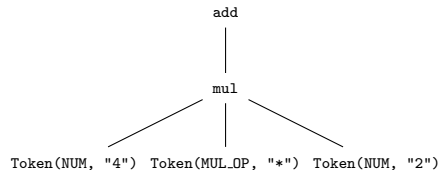


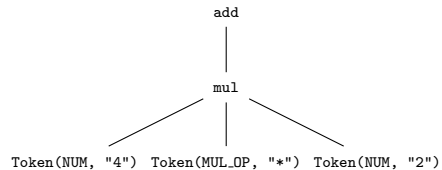
Abbildung 16: Compilerinterner Ableitungsbaum

Syntaktische Analyse

Ausgelassene Zwischenschritte

"4 * 2"

Parser
→
Lexer



- **Lexer** ist Teil des **Parsers**.

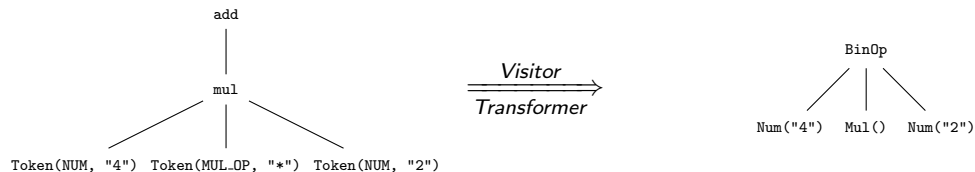
Syntaktische Analyse

Ausgelassene Zwischenschritte

```

bin_op ::= Add() | Mul()
exp    ::= BinOp(<exp>, <bin_op>, <exp>) | Num(<str>)
  
```

Grammatik 7: Produktionen für Abstrakten Syntaxbaum



Syntaktische Analyse

Lark Parsing Toolkit

- ▶ erleichtert Syntaktische Analyse.
- ▶ Grammatik spezifizieren nach der Lark ein Eingabewort parst und einen Ableitungsbaum generiert.
- ▶ Earley Parser implementiert.
- ▶ Implementierung von Visitor und Transformer.
- ▶ Quellcode: <https://github.com/lark-parser/lark>.
- ▶ Dokumentation:
<https://lark-parser.readthedocs.io/en/latest/index.html>.

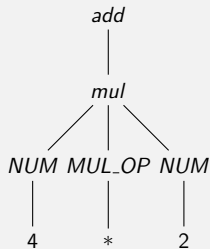
Syntaktische Analyse

Definitionen

Formaler Ableitungsbaum



- ▶ Darstellung einer **Ableitung** als **Baum**.
- ▶ Innere Knoten $\hat{=}$ Nicht-Terminalsymbole.
- ▶ Blätter $\hat{=}$ Terminalsymbole oder das **leere Wort** ε .



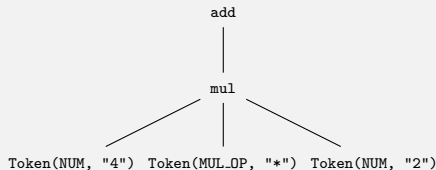
Syntaktische Analyse

Definitionen

(Compilerinterner) Ableitungsbaum



- ▶ compilerinterne Datenstruktur für Formalen Ableitungsbaum
- ▶ Innere Knoten $\hat{=}$ Nicht-Terminalsymbolen N der Grammatik des Parsers
 $G = \langle N, \Sigma, P, S \rangle$
- ▶ Blätter $\hat{=}$ Tokens (T, V) , Grammatik des Lexers interessiert nicht mehr



Syntaktische Analyse

Definitionen

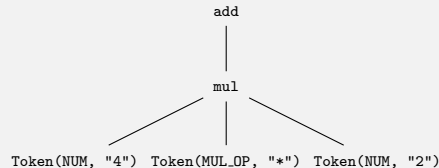
Parser



- ▶ generiert aus Eingabewort einen compilerinternen Ableitungsbaum
- ▶ beinhaltet Lexer

"4 * 2"

⇒



Syntaktische Analyse

Definitionen

Visitor



- ▶ von **unten-nach-oben** nach Prinzip der **Breitensuche** über **Ableitungsbaum**.
- ▶ **manipuliert** Knoten oder **tauscht** Knoten **in-place** mit anderen Knoten des Ableitungsbaumes, indem beim Antreffen bestimmter Knoten des Ableitungsbaumes bestimmte **Aktionen** ausgeführt werden.

Transformer



- ▶ von **unten-nach-oben** nach Prinzip der **Breitensuche** über **Ableitungsbaum**.
- ▶ generiert **Abstrakten Syntaxbaum**, indem beim Antreffen bestimmter Knoten des Ableitungsbaumes, diese durch Knoten des Abstrakten Syntaxbaumes **ersetzt** werden.

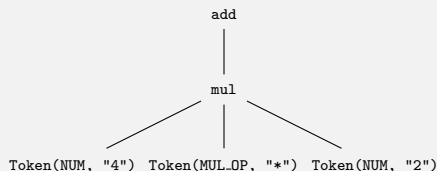
Syntaktische Analyse

Definitionen

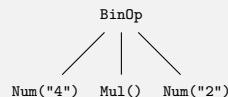
Abstrakter Syntaxbaum



- ▶ **compilerinterne** Datenstruktur
- ▶ **Abstraktion** eines Ableitungsbaumes, Knoten für z.B. Präzedenz sind weg.
- ▶ leichter **Zugriff** und **Weiterverarbeitbarkeit**
- ▶ setzt **Funktionalität einer Sprache** um und erlaubt es schnell herauszufinden aus welchen **Bestandteilen** der Sprache mit **unterscheidbarer Semantik** diese zusammengesetzt ist.



⇒



Code Generierung

Definitionen

Konkrete Syntax

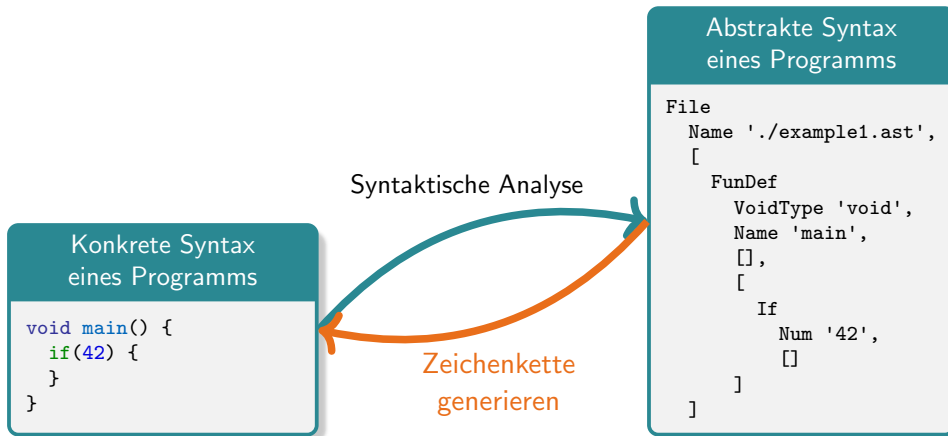
- ▶ bezeichnet den **Aufbau** von Programmen, wie man sie in eine **Textdatei** schreibt, um sie **kompilieren** zu lassen.

Abstrakte Syntax

- ▶ bezeichnet den **Aufbau** von **Abstrakten Syntaxbäumen**.
- ▶ nur bestimmte **Kompositionen** von Knoten sind **erlaubt**.

Code Generierung

Definitionen



Code Generierung

PicoC-Shrink Pass

- ▶ gleiche Semantik des Dereferenzierungsoperators $*(\text{pntr_or_ar} + i)$ und des Operators für Indexzugriff auf ein Feld $\text{pntr_or_ar}[i]$, sind austauschbar.
- ▶ Ersetzen von $\text{Deref}(\text{exp}, i)$ durch $\text{Subscr}(\text{exp}, i)$.
- ▶ Dereferenzierung $*(\text{pntr_or_ar} + i)$ wird von den Routinen für einen Indexzugriff auf ein Feld $\text{pntr_or_ar}[i]$ übernommen \Rightarrow kein redundanter Code.

Code Generierung

PicoC-Blocks Pass

- ▶ `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs-`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)` umgesetzt.
 - ▶ für **Bedingungen** und **Branches** ist jeweils ein eigener **Block** zuständig.
 - ▶ `IfElse(exp, stmts1, stmts2)` wird zur Umsetzung von **Bedingungen** verwendet.
 - ▶ für beide Fälle, wenn die Bedingung **wahr** oder **falsch** ist, wird mithilfe von `GoTo(label)` in einen von zwei **alternativen Blöcken** gesprungen oder ein Block **erneut aufgerufen** usw.

Code Generierung

PicoC-Blocks Pass

- ▶ jede **Funktion** erhält **eigenen Block**, der alle Anweisungen bis zum ersten Auftauchen oder Nicht-Auftauchen eines `If(exp, stmts)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)` enthält.

Code Generierung

PicoC-ANF Pass

- ▶ formt **Abstrakten Syntaxbaum** um, sodass er die **Syntax** der Sprache L_{PicoC_ANF} erfüllt, deren Grammatik in **A-Normalform** ist.
- ▶ **Funktionen** mit ihren **Lokalen Variablen**, **Parametern** und **Sichtbarkeitsbereichen**, sowie **Verbunstypen** mit ihren **Verbundsattributen** werden mithilfe einer **Symboltabelle** aufgelöst.

Symboltabelle



- ▶ $sym : \{my_var, my_fun, \dots\} \rightarrow \{Adresse, Datentyp, \dots\}^n, Bezeichner \mapsto (Information_1, \dots, Information_n).$
- ▶ um **während** dem **Kompilervorang** Informationen zu speichern, die später **nicht** mehr so einfach **zugänglich** sind.

Code Generierung

PicoC-ANF Pass

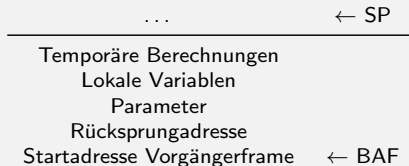
- ▶ alle Funktionen, **außer** der `main`-Funktion besitzen einen **Stackframe** für **Lokale Variablen** und **Parameter**.
- ▶ **Globale Statische Daten** sind **Globale Variablen**, sowie **Lokale Variablen** und **Parameter** der `main`-Funktion.

Code Generierung

PicoC-ANF Pass

Stackframe

- ▶ Datenstruktur, um **Zustand** einer Funktion zur Laufzeit zu „konservieren“.
- ▶ in einem Stack **übereinander gestapelt** und in die **entgegengesetzte Richtung** wieder abgebaut.
- ▶ die **Startadresse** des **Vorgängerframes** und die **Rücksprungadresse** beide im Stackframe der **aufgerufenen** Funktion.



Code Generierung

PicoC-ANF Pass

- **Zweck:** Maschinenbefehlen annähern, die meist nur **eine Aktion** ausführen, indem eine **Anweisung**, die **mehreren Aktionen** entspricht, aufgespalten wird und **Nebeneffekte** vorgeschoben werden.

Code

```
void main() {  
    int x = 1 - 5 * 4;  
}
```

ziehe **Komplexe Ausdrücke** aus
Anweisungen und Ausdrücken **vor**

Code in A-Normalform

```
void main() {  
    int x; // allocate at address  
    ↪ 0 relative to DS Register  
    1;  
    5;  
    4;  
    stack(2) * stack(1);  
    stack(2) - stack(1);  
    global(0) = stack(1);  
}
```

PicoC-ANF Pass

A-Normalform

Unreiner Ausdruck



- ▶ Ausdruck mit Nebeneffekt.

Reiner Ausdruck



- ▶ Ausdruck ohne Nebeneffekt.

PicoC-ANF Pass

A-Normalform

Monadische Normalform

- ▶ alle Anweisungen enthalten keine Unreinen Ausdrücke.
- ▶ Reine und Unreine Ausdrücke voneinander getrennt.

Code

```
void main() {  
    int var = 5 % 4;  
}
```

ziehe Unreine Ausdrücke
aus Anweisungen vor

Code in Monadi- scher Normalform

```
void main() {  
    int var;  
    var = 5 % 4;  
}
```

PicoC-ANF Pass

A-Normalform

Atomarer Ausdruck



- ▶ übersetzt sich in **keinen** kompletten Maschinenbefehl und **keine** Folge von Maschinenbefehlen.
- ▶ **legt** z.B. einen **Immediate** in einem Maschinenbefehl **fest**.

Komplexer Ausdruck



- ▶ ein Ausdruck, der **nicht atomar** ist.
- ▶ lässt sich in einen **Maschinenbefehl** oder eine **Folge von Maschinenbefehlen** übersetzen.

PicoC-ANF Pass

A-Normalform

A-Normalform

- ▶ ist bereits in **Monadischer Normalform**.
- ▶ alle **Anweisungen** und **Ausdrücke** enthalten ausschließlich **Atomare Ausdrücke**.

ziehe **Komplexe Ausdrücke** aus
Anweisungen und Ausdrücken **vor**

Code

```
void main() {  
    int x = 1 - 5 * 4;  
}
```

Code in A-Normalform

```
void main() {  
    int x; // allocate at address  
    ↪ 0 relative to DS Register  
    1;  
    5;  
    4;  
    stack(2) * stack(1);  
    stack(2) - stack(1);  
    global(0) = stack(1);  
}
```

PicoC-ANF Pass

A-Normalform

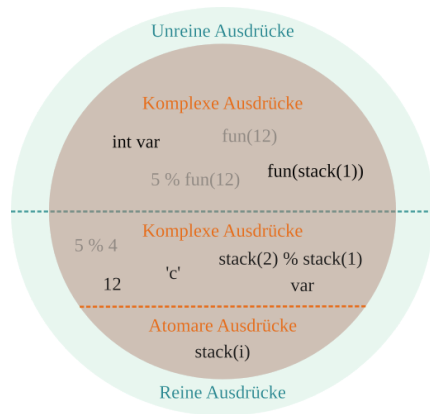


Abbildung 17: Überblick über Komplexe, Atomare, Unreine und Reine Ausdrücke.

Code Generierung

RETI-Blocks Pass

- ▶ **PicoC-Knoten**, die **Anweisungen** darstellen, werden durch **semantisch** entsprechende **RETI-Knoten**, die **Befehle** darstellen ersetzt.

Code Generierung

RETI-Patch Pass

- ▶ **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes** durch:
 - ▶ **Einfügen** eines `GoTo(Name('main'))` in den `global.<number>`-Block, wenn `main`-Funktion **nicht** die **erste Funktion** ist.
 - ▶ **Entfernen** von `GoTo()`s, deren Sprung nur **eine** Adresse weiterspringt.
 - ▶ RETI-Code **vor** jede Division, der prüft, ob durch 0 geteilt wird.
 - ▶ Fehlercode 1 in ACC-Register für `DivisionByZero`.
 - ▶ Ausführung wird **beendet**.

Code Generierung

RETI-Patch Pass

- ▶ **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes** durch:
 - ▶ Überprüfen, ob **Immediates** $\text{Im}(\text{str})$ in Befehlen $< -(2^{21})$ oder $> 2^{21} - 1$.
 - ▶ **Bitshiften** und Anwenden von **Bitweise ODER**.
 - ▶ **Immediate** $< -(2^{31})$ oder $> 2^{31} - 1 \Rightarrow \text{TooLargeLiteral}$.

Code Generierung

RETI Pass

- ▶ letzte verbliebene **PicoC-Knoten** werden durch entsprechende **RETI-Knoten** ersetzt:
 - ▶ **keine Blöcke** mehr, Knoten genauso **zusammengefügt**, wie sie in entfernten Blöcken **angeordnet** waren.
 - ▶ `GoTo(Name(str))` werden durch einen **Immediate** mit passender **Distanz** / **Adresse** oder einen **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))` ersetzt.

Code Generierung

RETI Pass

$$\triangleright \text{adr}_{\text{danach}} = \#Bef_{\text{vor akt. Bl.}} + \text{idx} + 4$$

$$\triangleright \text{Dist}_{\text{Zielbl.}} = \begin{cases} \#Bef_{\text{vor Zielbl.}} - \#Bef_{\text{vor akt. Bl.}} - \text{idx} & \#Bef_{\text{vor Zielbl.}} \neq \#Bef_{\text{vor akt. Bl.}} \\ -\text{idx} & \#Bef_{\text{vor Zielbl.}} = \#Bef_{\text{vor akt. Bl.}} \end{cases}$$

Code Generierung

RETI Pass

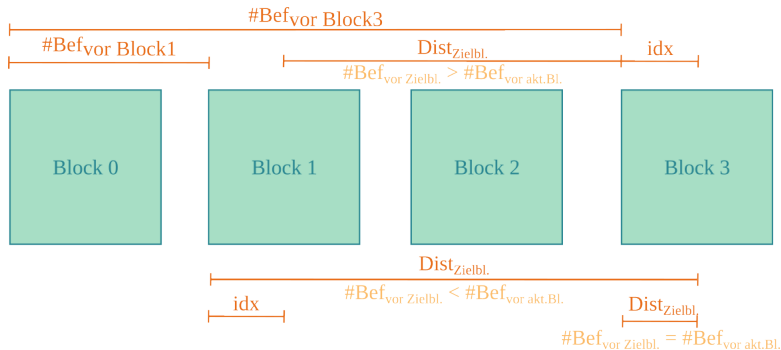


Abbildung 18: Veranschaulichung Distanzberechnung.

Fehlermeldungen

Kategorien

Fehlerkategorie	Beschreibung
UnexpectedCharacter	Der Lexer ist auf eine unerwartete Zeichenfolge gestossen, die in der Grammatik des Lexers nicht abgeleitet werden kann.
UnexpectedToken	Der Parser hat ein unerwartetes Token erhalten, das in dem Kontext in dem es sich befand in der Grammatik des Parsers nicht abgeleitet werden kann.
UnexpectedEOF	Der Parser hat in dem Kontext in dem er sich befand bestimmte Tokens erwartet , aber die Eingabe endete abrupt.

Tabelle 1: Fehlerarten in der Lexikalischen und Syntaktischen Analyse.

Fehlerkategorie	Beschreibung
DivisionByZero	Wenn bei einer Division durch 0 geteilt wird (z.B. <code>var / 0</code>).

Tabelle 2: Fehlerarten, die zur Laufzeit auftreten.

Fehlermeldungen

Kategorien

Fehlerkategorie	Beschreibung
UnknownIdentifier	Es wird ein Zugriff auf einen Bezeichner gemacht (z.B. <code>unknown_var + 1</code>), der noch nicht deklariert und ist daher nicht in der Symboltabelle aufgefunden werden kann.
UnknownAttribute	Der Verbundstyp (z.B. <code>struct st {int attr1; int attr2;}</code>) auf dessen Attribut im momentanen Kontext zugegriffen wird (z.B. <code>var[3].unknown_attr</code>) besitzt das Attribut (z.B. <code>unknown_attr</code>) auf das zugegriffen werden soll nicht .
ReDeclarationOrDefinition	Ein Bezeichner von z.B. einer Funktion oder Variable , der bereits deklariert oder definiert ist (z.B. <code>int var</code>) wird erneut deklariert oder definiert (z.B. <code>int var[2]</code>). Dieser Fehler ist leicht festzustellen, indem geprüft wird ob das Assoziative Feld durch welches die Symboltabelle umgesetzt ist diesen Bezeichner bereits als Schlüssel besitzt.
TooLargeLiteral	Der Wert eines Literals ist größer als $2^{31} - 1$ oder kleiner als -2^{31} .
NoMainFunction	Das Programm besitzt keine oder mehr als eine main-Funktion.

Tabelle 3: Fehlerarten in den Passes.

Fehlermeldungen

Kategorien

Fehlerkategorie	Beschreibung
ConstAssign	Wenn einer initialisierten Konstante (z.B. <code>const int const_var = 42</code>) ein Wert zugewiesen wird (z.B. <code>const_var = 41</code>). Der einzigste Weg , wie eine Konstante einen Wert erhält ist bei ihrer Initialisierung .
PrototypeMismatch	Der Prototyp einer deklarierten Funktion (z.B. <code>int fun(int arg1, int arg2[3])</code>) stimmt nicht mit dem Prototyp der späteren Definition dieser Funktion (z.B. <code>void fun(int arg1[2], int arg2) { }</code>) überein.
ArgumentMismatch	Wenn die Argumente eines Funktionsaufrufs (z.B. <code>fun(42, 314)</code>) nicht mit dem Prototyp der Funktion die aufgerufen werden soll (z.B. <code>void fun(int arg[2]) { }</code>) nach Datentypen oder Anzahl Argumente bzw. Parameter übereinstimmt.
WrongReturnType	Wenn eine Funktion, die ihrem Prototyp zufolge einen Rückgabewert hat, der nicht mit dem dem Datentyp übereinstimmt, der von einer return-Anweisung zurückgegeben wird.

Tabelle 4: Fehlerarten in den Passes, Teil 2.

Bedienung des PicoC-Compilers

Kommandozeilenargumente <cli-options>

Kommandozeilen-option	Beschreibung	Standardwert
<code>-i, --intermediate-stages</code>	Gibt Zwischenstufen der Kompilierung in Form der verschiedenen Tokens , Ableitungsbäume , Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateieindungen aber gleichem Basisnamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal .	false , most_used: true
<code>-p, --print</code>	Gibt alle Dateiausgaben auch im Terminal aus. Diese Option ist im Shell-Mode dauerhaft aktiviert.	false (true im Shell-Mode und für den most_used-Befehl)

Bedienung des PicoC-Compilers

Kommandozeilenargumente <cli-options>, Teil 2

Kommandozeilen-option	Beschreibung	Standardwert
-v, --verbose	Fügt den verschiedenen Zwischenschritten der Kompilierung , unter anderem auch dem finalen RETI-Code Kommentare hinzu. Diese Kommentare beinhalten eine Anweisung oder einen Befehl aus einem vorherigen Pass , der durch die darunterliegenden Anweisungen oder Befehle ersetzt wurde. Wenn die --run-Option aktiviert ist, wird der Zustand der virtuellen RETI-CPU vor und nach jedem Befehl angezeigt.	false
-vv, --double-verbose	Hat dieselben Effekte , wie die --verbose-Option, aber bewirkt zusätzlich weitere Effekte . PicoC-Knoten erhalten bei der Ausgabe als zusammenhängende Abstrakte Syntaxbäume zusätzliche runde Klammern , sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In Fehlermeldungen werden mehr Tokens angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der --intermediate_stages-Option werden in den dadurch ausgegebenen Abstrakten Syntaxbäumen zusätzlich versteckte Attribute angezeigt, die Informationen zu Datentypen und Informationen für Fehlermeldungen beinhalten.	false

Bedienung des PicoC-Compilers

Kommandozeilenargumente <cli-options>, Teil 3

Kommandozeilen-option	Beschreibung	Standardwert
-h, --help	Zeigt die Dokumentation , welche ebenfalls unter Link gefunden werden kann im Terminal an. Mit der --color-Option kann die Dokumentation mit farblicher Hervorhebung im Terminal angezeigt werden.	false
-l, --lines	Es lässt sich einstellen, wieviele Zeilen rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	2
-c, --color	Aktiviert farbige Ausgabe .	false, most_used: true
-t, --thesis	Filtert für die Codebeispiele in der schriftlichen Ausarbeitung der Bachelorarbeit bestimmte Kommentare in den Abstrakten Syntaxbäumen heraus, damit alles übersichtlich bleibt.	false

Bedienung des PicoC-Compilers

Kommandozeilenargumente <cli-options>, Teil 4

Kommandozeilen-option	Beschreibung	Standardwert
-R, --run	Führt die RETI-Befehle , die das Ergebnis der Kompilierung sind mit einer virtuellen RETI-CPU aus. Wenn die --intermediate_stages -Option aktiviert ist, wird eine Datei <basename>.reti_states erstellt, welche den Zustand der RETI-CPU nach dem letzten ausgeführten RETI-Befehl enthält. Wenn die --verbose- oder --double_verbose -Option aktiviert ist, wird der Zustand der RETI-CPU vor und nach jedem Befehl auch noch zusätzlich in die Datei <basename>.reti_states ausgegeben.	false , most_used: true
-B, --process-begin	Setzt die relative Adresse , wo der Prozess bzw. das Codesegment für das ausgeführte Programm beginnt .	3
-D, --datasegment-size	Setzt die Größe des Datensegments . Diese Option muss mit Vorsicht gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die Globalen Statischen Daten und der Stack miteinander kollidieren .	32

Bedienung des PicoC-Compilers

Shell-Mode

- ▶ Starten: `> picoc_compiler`.
- ▶ Kompilieren: `> compile <cli-options> "<seq-of-stmts>" (cpl)`.
 - ▶ automatisch in main-Funktion eingefügt: `void main(){<seq-of-stmts>}`.
- ▶ Kompilieren, meist genutzt:
`> must_used <cli-options> "<seq-of-stmts>" (mu)`.
- ▶ Beenden: `> quit`.
- ▶ Dokumentation: `> help (?)`.

Bedienung des PicoC-Compilers

Shell-Mode

- ▶ Multiline-Command: weitere Zeile mit `↵` und mit `;` terminieren.
- ▶ Farben toggeln: `> color_toggle (ct)`.
- ▶ Cursor bewegen: `←`, `→`.
- ▶ Befehlshistorie: `↑`, `↓`.
- ▶ Autovervollständigung: `Tab`.

Bedienung des PicoC-Compilers

Shell-Mode

- ▶ Befehlshistorie anzeigen: `> history` .
- ▶ Aktion mit Befehlshistorie ausführen `> history <opt>` .
 - ▶ Befehl erneut ausführen: `-r <cmd-nr>` .
 - ▶ Befehl editieren `-e <cmd-nr>` (Editor durch **Environment Variable** `$EDITOR` bestimmt).
 - ▶ Befehlshistorie leeren: `-c` .
 - ▶ Befehl suchen: `ctrl + r` .

Bedienung des PicoC-Compilers

Show-Mode

- ▶ Starten für bestimmtes Programm:

```
> make show FILEPATH=<path-to-file> <more-options> .
```

- ▶ Starten für bestimmten Test in /tests:

```
> make test-show TESTNAME=<testname> <more-options> .
```

- ▶ Zustände vor / nach Befehl ansehen: `Tab`, `↑ — Tab` .

- ▶ Beenden: `q`, `Esc` .

- ▶ Spezielle Einstellungen: `/interp_showcase.vim` .

- ▶ Neovim: `:help`, `:Tutor` .

Bedienung des PicoC-Compilers

Show-Mode

- ▶ Fenster minimieren / maximieren: **m**, **M**.
- ▶ Alle Fenster gleich aufteilen: **E**.
- ▶ Kommentare toggeln: **C**.
- ▶ (Relative) Zeilennummern toggeln: **N**, **R**.
- ▶ Zeile farbig markieren: **1**, ..., **9**.
- ▶ Farbig markierte Zeilen verstecken / wieder einblenden: **H**.
- ▶ Farbig markierte Zeilen entfernen **D**.
- ▶ Weiteres Fenster öffnen: **S**.

Tests

Bedienung

- ▶ Tests in /tests verifizieren und ausführen: `> make test <more-options> .`
 - ▶ `/run_tests.sh`, welches **zuerst** `/extract_input_and_expected.sh`, `/convert_to_c.py` und `/verify_tests.sh` ausführt.
- ▶ Tests vom GCC verifizieren lassen: `> make verify TESTNAME=<testname> .`
 - ▶ **vorher** `/extract_input_and_expected.sh`, `/convert_to_c.py` ausgeführt.
 - ▶ `/verify_tests.sh`.

Tests

Bedienung

► Informationen aus Tests extrahieren:

```
> make extract TESTNAME=<testname> .
```

► **Eingabe** // in:<space-sep-values> in <program>.in, **Ausgabe** // expected:<space-sep-values> in <program>.out_expected, **Datensegmentgröße** // datasegment:<size> **optional** in <program>.datasegment_size.

► `/extract_input_and_expected.sh` .

Tests

Bedienung

- ▶ Testdatei erstellen, die vom GCC kompiliert werden kann:

```
> make convert TESTNAME=<testname> .
```

- ▶ `input()`s werden durch **Eingaben** in `<program>.in` ersetzt.
- ▶ `print(exp)`s werden durch `#include<stdio.h>` und `printf("%d", exp)` ersetzt.
- ▶ `/convert_to_c.py` .

Tests

Makefile Optionen <more-options>

Kommandozeilenoption	Beschreibung	Standardwert
FILEPATH	Pfad zur Datei, die im Show-Mode angezeigt werden soll.	∅
TESTNAME	Name des Tests. Alles andere als der Basisname , wie die Dateiendung wird abgeschnitten.	∅
EXTENSION	Dateiendung , die an TESTNAME angehängt werden soll, damit daraus z.B. ./tests/TESTNAME.EXTENSION wird.	reti_states
NUM_WINDOWS	Anzahl Fenster auf die ein Dateiinhalt verteilt werden soll.	5
VERBOSE	Möglichkeit für eine ausführlichere Ausgabe die Kommandozeilenoption -v oder -vv zu aktivieren.	∅ bzw. -v für test-show
DEBUG	Möglichkeit die Kommandozeilenoption -d zu aktivieren, um bei make test-show TESTNAME=<testname> den Debugger für den entsprechenden Test <testname> zu starten.	∅

Tests

Testkategorien

Testkategorie	Beschreibung	Anzahl
basic	Grundlegende Funktionalitäten des PicoC-Compilers.	23
advanced	Spezialfälle und Kombinationen verschiedener Funktionalitäten des PicoC-Compilers.	20
hard	Lange und komplexe Tests, für welche die Funktionalitäten des PicoC-Compilers in perfekter Harmonie miteinander funktionieren müssen.	8
example	Bekannte Algorithmen, die als gutes, repräsentatives Beispiel für die Funktionsfähigkeit des PicoC-Compilers dienen.	24
error	Fehlermeldungen testen. Keine Verifikation wird ausgeführt.	69
exclude	Aufgrund vielfältiger Gründe soll keine Verifikation ausgeführt werden.	7
thesis	Codebeispiele der schriftlichen Ausarbeitung der Bachelorarbeit, die etwas umgeschrieben wurden, damit nicht nur das Durchlaufen dieser Tests getestet wird.	28
tobias	Vom Betreuer dieser Bachelorarbeit, M.Sc. Tobias Seufert geschrieben.	1

Codebeispiel

PicoC-Code

```
1 // in:1
2 // expected:42
3 // datasegment:36
4
5 struct stt {int attr1; int attr2[2];};
6
7 struct stt ar_of_sts[3][2];
8
9 int fun(struct stt (*param)[3][2]){
10     ((*param+2))[1].attr2[input()] = 42;
11     return 1;
12 }
13
14 void main() {
15     struct stt (*pntr_on_ar_of_sts)[3][2] = &ar_of_sts;
16     int res = fun(pntr_on_ar_of_sts);
17     if (res) {
18         print(((*pntr_on_ar_of_sts+2))[1].attr2[1]);
19     }
20 }
```

Lexikalische Analyse

Tokens generieren

PicoC-Code

```
// ...
struct stt (*pntr_on_ar_of_sts)[3][2]
↪ = &ar_of_sts;
// ...
```

Lexer

Tokenfolge

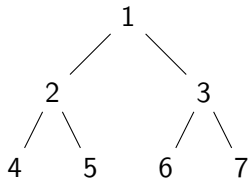
```
... Token('STRUCT', 'struct'),
↪ Token('NAME', 'stt'), Token('LPAR',
↪ '(', Token('MUL_DEREF_PNTR', '*'),
↪ Token('NAME', 'pntr_on_ar_of_sts'),
↪ Token('RPAR', ')'), Token('LSQB',
↪ '['), Token('NUM', '3'),
↪ Token('RSQB', ']'), Token('LSQB',
↪ '['), Token('NUM', '2'),
↪ Token('RSQB', ']'), Token('EQUAL',
↪ '='), Token('REF_AND', '&'),
↪ Token('NAME', 'ar_of_sts'),
↪ Token('SEMICOLON', ';'), ...
```

- ▶ ein „Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs stt“.

- ▶ *Clockwise/Spiral Rule*

Syntaktische Analyse

Darstellung von Bäumen



⇒

Baum in der Darstellung
des PicoC-Compilers

```
1
 2
  4
  5
 3
  6
  7
```

- ▶ wächst von links-nach-rechts und alle Kinderknoten sind unter dem Elternknoten.

Syntaktische Analyse

Ableitungsbaum generieren

PicoC-Code

```
// ...
struct stt (*pntr_on_ar_of_sts)[3][2] =
↪ &ar_of_sts;
// ...
```

- ▶ ein „Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs stt“.

- ▶ *Clockwise/Spiral Rule*

Parser
& Lexer

Ableitungsbaum

```
...
init_stmt
  alloc
    type_spec
      struct_spec
        name      stt
      pntr_decl
        pntr_deg
          array_decl
            pntr_decl
              pntr_deg      *
              array_decl
                name pntr_on_ar_of_sts
                array_dims
                  array_dims
                    3
                    2
              initializer
            ...
```

Syntaktische Analyse

Ableitungsbaum vereinfachen

Ableitungsbaum

```

...
init_stmt
  alloc
    type_spec
      struct_spec
        name      stt
      ptr_decl
        ptr_deg
        array_decl
          ptr_decl
            ptr_deg      *
            array_decl
              name ptr_on_ar_of_sts
              array_dims
                3
                2
            initializer
            ...

```

Visitor

Vereinfachter Ableitungsbaum

```

...
init_stmt
  alloc
    ptr_decl
      ptr_deg
      array_decl
        array_dims
          3
          2
      ptr_decl
        ptr_deg      *
        array_decl
          array_dims
          type_spec
            struct_spec
              name      stt
            name ptr_on_ar_of_sts
          initializer
          ...

```

Syntaktische Analyse

Abstrakten Syntaxbaum generieren

Vereinfachter Ableitungsbaum

```

...
init_stmt
  alloc
    pntr_decl
      pntr_deg
        array_decl
          array_dims
            3
            2
          pntr_decl
            pntr_deg *
            array_decl
              array_dims
              type_spec
              struct_spec
                name      stt
            name      pntr_on_ar_of_sts
          initializer
          ...

```

Transformer
ohne Umdrehen

Abstrakter Syntaxbaum

```

...
Assign
  Alloc
    Writeable,
    ArrayDecl
      [
        Num '3',
        Num '2'
      ],
    PntrDecl
      Num '1',
      StructSpec
        Name 'stt',
      Name 'pntr_on_ar_of_sts',
    Ref
      Name 'ar_of_sts',
  ...

```

Syntaktische Analyse

Abstrakten Syntaxbaum generieren

Vereinfachter Ableitungsbaum

```

...
init_stmt
  alloc
    pntr_decl
      pntr_deg
        array_decl
          array_dims
            3
            2
          pntr_decl
            pntr_deg *
            array_decl
              array_dims
              type_spec
                struct_spec
                  name      stt
            name      pntr_on_ar_of_sts
          name      pntr_on_ar_of_sts
        initializer
        ...

```

Transformer
mit Umdrehen

Abstrakter Syntaxbaum

```

...
Assign
  Alloc
    Writeable,
    PntrDecl
      Num '1',
      ArrayDecl
        [
          Num '3',
          Num '2'
        ],
      StructSpec
        Name 'stt',
        Name 'pntr_on_ar_of_sts',
      Ref
        Name 'ar_of_sts',
    ...

```

Syntaktische Analyse

Abstrakten Syntaxbaum generieren

Tokenfolge

```
... Token('STRUCT', 'struct'), Token('NAME',
↳ 'stt'), Token('LPAR', '('),
↳ Token('MUL_DEREF_PNTR', '*'),
↳ Token('NAME', 'pntr_on_ar_of_sts'),
↳ Token('RPAR', ')'), Token('LSQB', '['),
↳ Token('NUM', '3'), Token('RSQB', ']'),
↳ Token('LSQB', '['), Token('NUM', '2'),
↳ Token('RSQB', ']'), Token('EQUAL', '='),
↳ Token('REF_AND', '&'), Token('NAME',
↳ 'ar_of_sts'), Token('SEMICOLON', ';'),
↳ ...
```

Syntaktische Analyse

Abstrakter Syntaxbaum

```
...
Assign
  Alloc
    Writeable,
    PntrDecl
      Num '1',
      ArrayDecl
        [
          Num '3',
          Num '2'
        ],
      StructSpec
        Name 'stt',
        Name 'pntr_on_ar_of_sts',
      Ref
        Name 'ar_of_sts',
    ...
```

Code Generierung

Abstrakter Syntaxbaum nach Syntaktischer Analyse

```
1 File
2   Name './example_presentation.ast',
3   [
4     StructDecl
5       Name 'stt',
6       [
7         Alloc(Writeable(), IntType('int'), Name('attr1'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9       ],
10    Exp
11      Alloc
12        Writeable,
13        ArrayDecl
14          [
15            Num '3',
16            Num '2'
17          ],
18        StructSpec
19          Name 'stt',
20          Name 'ar_of_sts',
```

Code Generierung

Abstrakter Syntaxbaum nach Syntaktischer Analyse, Teil 2

```
21 FunDef
22   IntType 'int',
23   Name 'fun',
24   [
25     Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))),
26     ↪ Name('param'))
27   ],
28   [
29     Assign(Subscr(Attr(Subscr(Deref(Deref(Name('param')), Num('0')), Num('2')), Num('1')), Name('attr2')),
30     ↪ Call(Name('input'), [])), Num('42'))
31     Return(Num('1'))
32   ],
33 ]
```

Code Generierung

Abstrakter Syntaxbaum nach Syntaktischer Analyse, Teil 3

```

31  FunDef
32      VoidType 'void',
33      Name 'main',
34      [],
35      [
36          Assign(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))),
37              ↪ Name('pntr_on_ar_of_sts')), Ref(Name('ar_of_sts')))
38          Assign(Name('res'), Call(Name('fun'), [Name('pntr_on_ar_of_sts')])),
39          If
40              Name 'res',
41              [
42                  Exp(Call(Name('print'), [Subscr(Attr(Subscr(Deref(Deref(Name('pntr_on_ar_of_sts')), Num('0')),
43                      ↪ Num('2')), Num('1')), Name('attr2')), Num('1'))]))
44              ]
45      ]

```


Code Generierung

PicoC-Shrink Pass

```
1 File
2   Name './example_presentation.picoc_shrink',
3   [
4     StructDecl
5       Name 'stt',
6       [
7         Alloc(Writeable(), IntType('int'), Name('attr1'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9       ],
10    Exp
11    Alloc
12    Writeable,
13    ArrayDecl
14    [
15      Num '3',
16      Num '2'
17    ],
18    StructSpec
19      Name 'stt',
20      Name 'ar_of_sts',
```

Code Generierung

PicoC-Shrink Pass, Teil 2

```
21 FunDef
22   IntType 'int',
23   Name 'fun',
24   [
25     Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))),
26     ↪ Name('param'))
27   ],
28   [
29     Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')), Name('attr2')),
30     ↪ Call(Name('input'), [])), Num('42'))
31     Return(Num('1'))
32   ],
```

Code Generierung

PicoC-Shrink Pass, Teil 3

```

31  FunDef
32      VoidType 'void',
33      Name 'main',
34      [],
35      [
36          Assign(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))),
37              ↪ Name('ptr_on_ar_of_sts')), Ref(Name('ar_of_sts')))
38          Assign(Name('res'), Call(Name('fun'), [Name('ptr_on_ar_of_sts')])),
39          If
40              Name 'res',
41              [
42                  Exp(Call(Name('print'), [Subscr(Attr(Subscr(Subscr(Name('ptr_on_ar_of_sts'), Num('0')),
43                      ↪ Num('2')), Num('1')), Name('attr2')), Num('1')))]))
44              ]
45      ]

```

Code Generierung

PicoC-Blocks Pass

```
1 File
2   Name './example_presentation.picoc_blocks',
3   [
4     StructDecl
5       Name 'stt',
6       [
7         Alloc(Writeable(), IntType('int'), Name('attr1'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9       ],
10    Exp
11    Alloc
12    Writeable,
13    ArrayDecl
14    [
15      Num '3',
16      Num '2'
17    ],
18    StructSpec
19      Name 'stt',
20      Name 'ar_of_sts',
```

Code Generierung

PicoC-Blocks Pass, Teil 2

```
21 FunDef
22   IntType 'int',
23   Name 'fun',
24   [
25     Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))),
26     ↪ Name('param'))
27   ],
28   [
29     Block
30     Name 'fun.3',
31     [
32       Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')),
33       ↪ Name('attr2')), Call(Name('input'), [])), Num('42'))
34       Return(Num('1'))
35     ]
36   ],
```

Code Generierung

PicoC-Blocks Pass, Teil 3

```

35  FunDef
36      VoidType 'void',
37      Name 'main',
38      [],
39      [
40          Block
41              Name 'main.2',
42              [
43                  Assign(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
44                      ↪ StructSpec(Name('stt')))), Name('ptr_on_ar_of_sts')), Ref(Name('ar_of_sts')))
45                  Assign(Alloc(Writable(), IntType('int'), Name('res')), Call(Name('fun'),
46                      ↪ [Name('ptr_on_ar_of_sts')]))
47                  // If(Name('res'), [],),
48                  IfElse
49                      Name 'res',
50                      [
51                          GoTo(Name('if.1'))
52                      ],
53                      [
54                          GoTo(Name('if_else_after.0'))
55                      ]
56              ],
57      ],

```

Code Generierung

PicoC-Blocks Pass, Teil 4

```
55     Block
56     Name 'if.1',
57     [
58         Exp(Call(Name('print'), [Subscr(Attr(Subscr(Subscr(Subscr(Name('pntr_on_ar_of_sts'), Num('0')),
59             ↪ Num('2')), Num('1')), Name('attr2')), Num('1'))]))
60         GoTo(Name('if_else_after.0'))
61     ],
62     Block
63     Name 'if_else_after.0',
64     []
65 ]
```

Code Generierung

PicoC-ANF Pass - Symboltabelle

```
1 SymbolTable
2 [
3     Symbol
4     {
5         type qualifier:      Empty()
6         datatype:            IntType('int')
7         name:                 Name('attr1@stt')
8         value or address:     Empty()
9         position:             Pos(Num('5'), Num('16'))
10        size:                 Num('1')
11    },
12    Symbol
13    {
14        type qualifier:      Empty()
15        datatype:            ArrayDecl([Num('2')], IntType('int'))
16        name:                 Name('attr2@stt')
17        value or address:     Empty()
18        position:             Pos(Num('5'), Num('27'))
19        size:                 Num('2')
20    },
```


Code Generierung

PicoC-ANF Pass - Symboltabelle, Teil 2

```

21 Symbol
22 {
23     type qualifier:      Empty()
24     datatype:            StructDecl(Name('stt'), [Alloc(Writable(), IntType('int'),
25 ↪ Name('attr1'))Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))])
26     name:                Name('stt')
27     value or address:    [Name('attr1@stt'), Name('attr2@stt')]
28     position:            Pos(Num('5'), Num('7'))
29     size:                Num('3')
30 },
31 Symbol
32 {
33     type qualifier:      Writable()
34     datatype:            ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))
35     name:                Name('ar_of_sts@global!')
36     value or address:    Num('0')
37     position:            Pos(Num('7'), Num('11'))
38     size:                Num('18')
39 },

```

Code Generierung

PicoC-ANF Pass - Symboltabelle, Teil 3

```

39 Symbol
40 {
41   type qualifier:      Empty()
42   datatype:            FunDecl(IntType('int'), Name('fun'), [Alloc(Writeable(), PntrDecl(Num('1'),
43   ↪   ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt')))), Name('param'))])
44   name:                Name('fun')
45   value or address:    Empty()
46   position:            Pos(Num('9'), Num('4'))
47   size:                Empty()
48 },
49 Symbol
50 {
51   type qualifier:      Writeable()
52   datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt'))))
53   name:                Name('param@fun')
54   value or address:    Num('0')
55   position:            Pos(Num('9'), Num('21'))
56   size:                Num('1')
57 },

```

Code Generierung

PicoC-ANF Pass - Symboltabelle, Teil 4

```
57 Symbol
58 {
59     type qualifier:      Empty()
60     datatype:            FunDecl(VoidType('void'), Name('main'), [])
61     name:                 Name('main')
62     value or address:     Empty()
63     position:             Pos(Num('14'), Num('5'))
64     size:                 Empty()
65 },
66 Symbol
67 {
68     type qualifier:      Writeable()
69     datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], StructSpec(Name('stt'))))
70     name:                 Name('pntr_on_ar_of_sts@main')
71     value or address:     Num('18')
72     position:             Pos(Num('15'), Num('15'))
73     size:                 Num('1')
74 },
```

Code Generierung

PicoC-ANF Pass - Symboltabelle, Teil 5

```
75 Symbol
76 {
77     type qualifier:    Writeable()
78     datatype:          IntType('int')
79     name:               Name('res@main')
80     value or address:   Num('19')
81     position:           Pos(Num('16'), Num('6'))
82     size:               Num('1')
83 }
84 ]
```

Code Generierung

PicoC-ANF Pass

```

1 File
2   Name './example_presentation.picoc_mon',
3   [
4     Block
5       Name 'global.4',
6       [],
7     Block
8       Name 'fun.3',
9       [
10        // Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')),
11        ↪ Name('attr2')), Call(Name('input'), [])), Num('42'))
12        Exp(Num('42'))
13        Ref(Stackframe(Num('0')))
14        Exp(Num('0'))
15        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
16        Exp(Num('2'))
17        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18        Exp(Num('1'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Ref(Attr(Stack(Num('1')), Name('attr2')))
21        Exp(Call(Name('input'), []))

```

Code Generierung

PicoC-ANF Pass, Teil 2

```

21     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22     Assign(Stack(Num('1')), Stack(Num('2')))
23     // Return(Num('1'))
24     Exp(Num('1'))
25     Return(Stack(Num('1')))
26 ],
27 Block
28   Name 'main.2',
29   [
30     // Assign(Name('pntr_on_ar_of_sts'), Ref(Name('ar_of_sts')))
31     Ref(Global(Num('0')))
32     Assign(Global(Num('18')), Stack(Num('1')))
33     // Assign(Name('res'), Call(Name('fun'), [Name('pntr_on_ar_of_sts')]))
34     StackMalloc(Num('2'))
35     Exp(Global(Num('18')))
36     NewStackframe(Name('fun.3'), GoTo(Name('addr@next_instr')))
37     Exp(GoTo(Name('fun.3')))
38     RemoveStackframe()
39     Exp(ACC)
40     Assign(Global(Num('19')), Stack(Num('1')))

```

Code Generierung

PicoC-ANF Pass, Teil 3

```
41 // If(Name('res'), [])
42 // IfElse(Name('res'), [], [])
43 Exp(Global(Num('19'))),
44 IfElse
45   Stack
46     Num '1',
47     [
48       GoTo(Name('if.1'))
49     ],
50     [
51       GoTo(Name('if_else_after.0'))
52     ]
53 ],
54 Block
55   Name 'if.1',
56   [
57     Ref(Global(Num('18')))
58     Exp(Num('0'))
59     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
```

Code Generierung

PicoC-ANF Pass, Teil 4

```
60     Exp(Num('2'))
61     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
62     Exp(Num('1'))
63     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
64     Ref(Attr(Stack(Num('1')), Name('attr2')))
65     Exp(Num('1'))
66     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
67     Exp(Stack(Num('1')))
68     Exp(Call(Name('print'), [Stack(Num('1'))]))
69     Exp(GoTo(Name('if_else_after.0')))
70 ],
71 Block
72   Name 'if_else_after.0',
73   [
74     Return(Empty())
75   ]
76 ]
```


Code Generierung

RETI-Blocks Pass

```
1 File
2   Name './example_presentation.reti_blocks',
3   [
4     Block
5       Name 'global.4',
6       [],
7     Block
8       Name 'fun.3',
9       [
10        # // Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')),
11        ↪ Name('attr2')), Call(Name('input'), [])), Num('42'))
12        # Exp(Num('42'))
13        SUBI SP 1;
14        LOADI ACC 42;
15        STOREIN SP ACC 1;
16        # Ref(Stackframe(Num('0')))
17        SUBI SP 1;
18        MOVE BAF IN1;
19        SUBI IN1 2;
20        STOREIN SP IN1 1;
21        # Exp(Num('0'))
22        SUBI SP 1;
```

Code Generierung

RETI-Blocks Pass, Teil 2

```
22     LOADI ACC 0;
23     STOREIN SP ACC 1;
24     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25     LOADIN SP IN2 2;
26     LOADIN IN2 IN1 0;
27     LOADIN SP IN2 1;
28     MULTI IN2 18;
29     ADD IN1 IN2;
30     ADDI SP 1;
31     STOREIN SP IN1 1;
32     # Exp(Num('2'))
33     SUBI SP 1;
34     LOADI ACC 2;
35     STOREIN SP ACC 1;
36     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
37     LOADIN SP IN1 2;
38     LOADIN SP IN2 1;
39     MULTI IN2 6;
40     ADD IN1 IN2;
41     ADDI SP 1;
42     STOREIN SP IN1 1;
```

Code Generierung

RETI-Blocks Pass, Teil 3

```
43      # Exp(Num('1'))
44      SUBI SP 1;
45      LOADI ACC 1;
46      STOREIN SP ACC 1;
47      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
48      LOADIN SP IN1 2;
49      LOADIN SP IN2 1;
50      MULTI IN2 3;
51      ADD IN1 IN2;
52      ADDI SP 1;
53      STOREIN SP IN1 1;
54      # Ref(Attr(Stack(Num('1')), Name('attr2')))
55      LOADIN SP IN1 1;
56      ADDI IN1 1;
57      STOREIN SP IN1 1;
58      CALL INPUT ACC;
59      SUBI SP 1;
60      STOREIN SP ACC 1;
61      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
62      LOADIN SP IN1 2;
63      LOADIN SP IN2 1;
```

Code Generierung

RETI-Blocks Pass, Teil 4

```
64     MULTI IN2 1;
65     ADD IN1 IN2;
66     ADDI SP 1;
67     STOREIN SP IN1 1;
68     # Assign(Stack(Num('1')), Stack(Num('2')))
69     LOADIN SP IN1 1;
70     LOADIN SP ACC 2;
71     ADDI SP 2;
72     STOREIN IN1 ACC 0;
73     # // Return(Num('1'))
74     # Exp(Num('1'))
75     SUBI SP 1;
76     LOADI ACC 1;
77     STOREIN SP ACC 1;
78     # Return(Stack(Num('1')))
79     LOADIN SP ACC 1;
80     ADDI SP 1;
81     LOADIN BAF PC -1;
82 ],
83 Block
84     Name 'main.2',
```

Code Generierung

RETI-Blocks Pass, Teil 5

```
85  [
86    # // Assign(Name('pntr_on_ar_of_sts'), Ref(Name('ar_of_sts')))
87    # Ref(Global(Num('0')))
88    SUBI SP 1;
89    LOADI IN1 0;
90    ADD IN1 DS;
91    STOREIN SP IN1 1;
92    # Assign(Global(Num('18')), Stack(Num('1')))
93    LOADIN SP ACC 1;
94    STOREIN DS ACC 18;
95    ADDI SP 1;
96    # // Assign(Name('res'), Call(Name('fun'), [Name('pntr_on_ar_of_sts')]))
97    # StackMalloc(Num('2'))
98    SUBI SP 2;
99    # Exp(Global(Num('18')))
100   SUBI SP 1;
101   LOADIN DS ACC 18;
102   STOREIN SP ACC 1;
103   # NewStackframe(Name('fun.3'), GoTo(Name('addr@next_instr')))
104   MOVE BAF ACC;
105   ADDI SP 3;
```

Code Generierung

RETI-Blocks Pass, Teil 6

```
106     MOVE SP BAF;  
107     SUBI SP 3;  
108     STOREIN BAF ACC 0;  
109     LOADI ACC GoTo(Name('addr@next_instr'));  
110     ADD ACC CS;  
111     STOREIN BAF ACC -1;  
112     # Exp(GoTo(Name('fun.3')))  
113     Exp(GoTo(Name('fun.3')))  
114     # RemoveStackframe()  
115     MOVE BAF IN1;  
116     LOADIN IN1 BAF 0;  
117     MOVE IN1 SP;  
118     # Exp(ACC)  
119     SUBI SP 1;  
120     STOREIN SP ACC 1;  
121     # Assign(Global(Num('19')), Stack(Num('1')))  
122     LOADIN SP ACC 1;  
123     STOREIN DS ACC 19;  
124     ADDI SP 1;  
125     # // If(Name('res'), [])  
126     # // IfElse(Name('res'), [], [])
```

Code Generierung

RETI-Blocks Pass, Teil 7

```
127      # Exp(Global(Num('19')))  
128      SUBI SP 1;  
129      LOADIN DS ACC 19;  
130      STOREIN SP ACC 1;  
131      # IfElse(Stack(Num('1')), [], [])  
132      LOADIN SP ACC 1;  
133      ADDI SP 1;  
134      JUMP== GoTo(Name('if_else_after.0'));  
135      Exp(GoTo(Name('if.1')))  
136  ],  
137  Block  
138      Name 'if.1',  
139      [  
140          # Ref(Global(Num('18')))  
141          SUBI SP 1;  
142          LOADI IN1 18;  
143          ADD IN1 DS;  
144          STOREIN SP IN1 1;  
145          # Exp(Num('0'))  
146          SUBI SP 1;  
147          LOADI ACC 0;
```

Code Generierung

RETI-Blocks Pass, Teil 8

```
148     STOREIN SP ACC 1;
149     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
150     LOADIN SP IN2 2;
151     LOADIN IN2 IN1 0;
152     LOADIN SP IN2 1;
153     MULTI IN2 18;
154     ADD IN1 IN2;
155     ADDI SP 1;
156     STOREIN SP IN1 1;
157     # Exp(Num('2'))
158     SUBI SP 1;
159     LOADI ACC 2;
160     STOREIN SP ACC 1;
161     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
162     LOADIN SP IN1 2;
163     LOADIN SP IN2 1;
164     MULTI IN2 6;
165     ADD IN1 IN2;
166     ADDI SP 1;
167     STOREIN SP IN1 1;
168     # Exp(Num('1'))
```


Code Generierung

RETI-Blocks Pass, Teil 9

```
169     SUBI SP 1;
170     LOADI ACC 1;
171     STOREIN SP ACC 1;
172     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
173     LOADIN SP IN1 2;
174     LOADIN SP IN2 1;
175     MULTI IN2 3;
176     ADD IN1 IN2;
177     ADDI SP 1;
178     STOREIN SP IN1 1;
179     # Ref(Attr(Stack(Num('1')), Name('attr2')))
180     LOADIN SP IN1 1;
181     ADDI IN1 1;
182     STOREIN SP IN1 1;
183     # Exp(Num('1'))
184     SUBI SP 1;
185     LOADI ACC 1;
186     STOREIN SP ACC 1;
187     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
188     LOADIN SP IN1 2;
189     LOADIN SP IN2 1;
```

Code Generierung

RETI-Blocks Pass, Teil 10

```
190     MULTI IN2 1;
191     ADD IN1 IN2;
192     ADDI SP 1;
193     STOREIN SP IN1 1;
194     # Exp(Stack(Num('1')))
195     LOADIN SP IN1 1;
196     LOADIN IN1 ACC 0;
197     STOREIN SP ACC 1;
198     LOADIN SP ACC 1;
199     ADDI SP 1;
200     CALL PRINT ACC;
201     # Exp(GoTo(Name('if_else_after.0')))
202     Exp(GoTo(Name('if_else_after.0')))
203 ],
204 Block
205   Name 'if_else_after.0',
206   [
207     # Return(Empty())
208     LOADIN BAF PC -1;
209   ]
210 ]
```

Code Generierung

RETI-Patch Pass

```
1 File
2   Name './example_presentation.reti_patch',
3   [
4     Block
5       Name 'global.4',
6       [
7         # // Exp(GoTo(Name('main.2')))
8         Exp(GoTo(Name('main.2')))
9       ],
10    Block
11      Name 'fun.3',
12      [
13        # // Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')),
14        ↪ Name('attr2')), Call(Name('input'), [])), Num('42'))
15        # Exp(Num('42'))
16        SUBI SP 1;
17        LOADI ACC 42;
18        STOREIN SP ACC 1;
19        # Ref(Stackframe(Num('0')))
20        SUBI SP 1;
21        MOVE BAF IN1;
22        SUBI IN1 2;
23        STOREIN SP IN1 1;
```

Code Generierung

RETI-Patch Pass, Teil 2

```
23      # Exp(Num('0'))
24      SUBI SP 1;
25      LOADI ACC 0;
26      STOREIN SP ACC 1;
27      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
28      LOADIN SP IN2 2;
29      LOADIN IN2 IN1 0;
30      LOADIN SP IN2 1;
31      MULTI IN2 18;
32      ADD IN1 IN2;
33      ADDI SP 1;
34      STOREIN SP IN1 1;
35      # Exp(Num('2'))
36      SUBI SP 1;
37      LOADI ACC 2;
38      STOREIN SP ACC 1;
39      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
40      LOADIN SP IN1 2;
41      LOADIN SP IN2 1;
42      MULTI IN2 6;
43      ADD IN1 IN2;
44      ADDI SP 1;
```

Code Generierung

RETI-Patch Pass, Teil 3

```
45     STOREIN SP IN1 1;  
46     # Exp(Num('1'))  
47     SUBI SP 1;  
48     LOADI ACC 1;  
49     STOREIN SP ACC 1;  
50     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))  
51     LOADIN SP IN1 2;  
52     LOADIN SP IN2 1;  
53     MULTI IN2 3;  
54     ADD IN1 IN2;  
55     ADDI SP 1;  
56     STOREIN SP IN1 1;  
57     # Ref(Attr(Stack(Num('1')), Name('attr2')))  
58     LOADIN SP IN1 1;  
59     ADDI IN1 1;  
60     STOREIN SP IN1 1;  
61     CALL INPUT ACC;  
62     SUBI SP 1;  
63     STOREIN SP ACC 1;  
64     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))  
65     LOADIN SP IN1 2;  
66     LOADIN SP IN2 1;
```

Code Generierung

RETI-Patch Pass, Teil 4

```
67     MULTI IN2 1;
68     ADD IN1 IN2;
69     ADDI SP 1;
70     STOREIN SP IN1 1;
71     # Assign(Stack(Num('1')), Stack(Num('2')))
72     LOADIN SP IN1 1;
73     LOADIN SP ACC 2;
74     ADDI SP 2;
75     STOREIN IN1 ACC 0;
76     # // Return(Num('1'))
77     # Exp(Num('1'))
78     SUBI SP 1;
79     LOADI ACC 1;
80     STOREIN SP ACC 1;
81     # Return(Stack(Num('1')))
82     LOADIN SP ACC 1;
83     ADDI SP 1;
84     LOADIN BAF PC -1;
85 ],
86 Block
87   Name 'main.2',
88   [
```

Code Generierung

RETI-Patch Pass, Teil 5

```
89      # // Assign(Name('pntr_on_ar_of_sts'), Ref(Name('ar_of_sts'))))
90      # Ref(Global(Num('0'))))
91      SUBI SP 1;
92      LOADI IN1 0;
93      ADD IN1 DS;
94      STOREIN SP IN1 1;
95      # Assign(Global(Num('18')), Stack(Num('1'))))
96      LOADIN SP ACC 1;
97      STOREIN DS ACC 18;
98      ADDI SP 1;
99      # // Assign(Name('res'), Call(Name('fun'), [Name('pntr_on_ar_of_sts')]))
100     # StackMalloc(Num('2'))
101     SUBI SP 2;
102     # Exp(Global(Num('18'))))
103     SUBI SP 1;
104     LOADIN DS ACC 18;
105     STOREIN SP ACC 1;
106     # NewStackframe(Name('fun.3'), GoTo(Name('addr@next_instr'))))
107     MOVE BAF ACC;
108     ADDI SP 3;
109     MOVE SP BAF;
110     SUBI SP 3;
```

Code Generierung

RETI-Patch Pass, Teil 6

```
111     STOREIN BAF ACC 0;
112     LOADI ACC GoTo(Name('addr@next_instr'));
113     ADD ACC CS;
114     STOREIN BAF ACC -1;
115     # Exp(GoTo(Name('fun.3')))
116     Exp(GoTo(Name('fun.3')))
117     # RemoveStackframe()
118     MOVE BAF IN1;
119     LOADIN IN1 BAF 0;
120     MOVE IN1 SP;
121     # Exp(ACC)
122     SUBI SP 1;
123     STOREIN SP ACC 1;
124     # Assign(Global(Num('19')), Stack(Num('1')))
125     LOADIN SP ACC 1;
126     STOREIN DS ACC 19;
127     ADDI SP 1;
128     # // If(Name('res'), [])
129     # // IfElse(Name('res'), [], [])
130     # Exp(Global(Num('19')))
131     SUBI SP 1;
132     LOADIN DS ACC 19;
```


Code Generierung

RETI-Patch Pass, Teil 7

```
133     STOREIN SP ACC 1;
134     # IfElse(Stack(Num('1')), [], [])
135     LOADIN SP ACC 1;
136     ADDI SP 1;
137     JUMP== GoTo(Name('if_else_after.0'));
138     # // not included Exp(GoTo(Name('if.1')))
139 ],
140 Block
141   Name 'if.1',
142   [
143     # Ref(Global(Num('18')))
144     SUBI SP 1;
145     LOADI IN1 18;
146     ADD IN1 DS;
147     STOREIN SP IN1 1;
148     # Exp(Num('0'))
149     SUBI SP 1;
150     LOADI ACC 0;
151     STOREIN SP ACC 1;
152     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
153     LOADIN SP IN2 2;
154     LOADIN IN2 IN1 0;
```

Code Generierung

RETI-Patch Pass, Teil 8

```
155     LOADIN SP IN2 1;
156     MULTI IN2 18;
157     ADD IN1 IN2;
158     ADDI SP 1;
159     STOREIN SP IN1 1;
160     # Exp(Num('2'))
161     SUBI SP 1;
162     LOADI ACC 2;
163     STOREIN SP ACC 1;
164     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
165     LOADIN SP IN1 2;
166     LOADIN SP IN2 1;
167     MULTI IN2 6;
168     ADD IN1 IN2;
169     ADDI SP 1;
170     STOREIN SP IN1 1;
171     # Exp(Num('1'))
172     SUBI SP 1;
173     LOADI ACC 1;
174     STOREIN SP ACC 1;
175     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
176     LOADIN SP IN1 2;
```

Code Generierung

RETI-Patch Pass, Teil 9

```
177     LOADIN SP IN2 1;
178     MULTI IN2 3;
179     ADD IN1 IN2;
180     ADDI SP 1;
181     STOREIN SP IN1 1;
182     # Ref(Attr(Stack(Num('1')), Name('attr2')))
183     LOADIN SP IN1 1;
184     ADDI IN1 1;
185     STOREIN SP IN1 1;
186     # Exp(Num('1'))
187     SUBI SP 1;
188     LOADI ACC 1;
189     STOREIN SP ACC 1;
190     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
191     LOADIN SP IN1 2;
192     LOADIN SP IN2 1;
193     MULTI IN2 1;
194     ADD IN1 IN2;
195     ADDI SP 1;
196     STOREIN SP IN1 1;
197     # Exp(Stack(Num('1')))
198     LOADIN SP IN1 1;
```

Code Generierung

RETI-Patch Pass, Teil 10

```
199     LOADIN IN1 ACC 0;
200     STOREIN SP ACC 1;
201     LOADIN SP ACC 1;
202     ADDI SP 1;
203     CALL PRINT ACC;
204     # Exp(GoTo(Name('if_else_after.0')))
205     # // not included Exp(GoTo(Name('if_else_after.0')))
206 ],
207 Block
208   Name 'if_else_after.0',
209   [
210     # Return(Empty())
211     LOADIN BAF PC -1;
212   ]
213 ]
```

Code Generierung

RETI Pass

```
1 # // Exp(GoTo(Name('main.2')))  
2 JUMP 58;  
3 # // Assign(Subscr(Attr(Subscr(Subscr(Subscr(Name('param'), Num('0')), Num('2')), Num('1')), Name('attr2')),  
↪ Call(Name('input'), [])), Num('42'))  
4 # Exp(Num('42'))  
5 SUBI SP 1;  
6 LOADI ACC 42;  
7 STOREIN SP ACC 1;  
8 # Ref(Stackframe(Num('0')))  
9 SUBI SP 1;  
10 MOVE BAF IN1;  
11 SUBI IN1 2;  
12 STOREIN SP IN1 1;  
13 # Exp(Num('0'))  
14 SUBI SP 1;  
15 LOADI ACC 0;  
16 STOREIN SP ACC 1;  
17 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))  
18 LOADIN SP IN2 2;  
19 LOADIN IN2 IN1 0;  
20 LOADIN SP IN2 1;  
21 MULTI IN2 18;
```

Code Generierung

RETI Pass, Teil 2

```
22 ADD IN1 IN2;
23 ADDI SP 1;
24 STOREIN SP IN1 1;
25 # Exp(Num('2'))
26 SUBI SP 1;
27 LOADI ACC 2;
28 STOREIN SP ACC 1;
29 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30 LOADIN SP IN1 2;
31 LOADIN SP IN2 1;
32 MULTI IN2 6;
33 ADD IN1 IN2;
34 ADDI SP 1;
35 STOREIN SP IN1 1;
36 # Exp(Num('1'))
37 SUBI SP 1;
38 LOADI ACC 1;
39 STOREIN SP ACC 1;
40 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41 LOADIN SP IN1 2;
42 LOADIN SP IN2 1;
```

Code Generierung

RETI Pass, Teil 3

```
43 MULTI IN2 3;
44 ADD IN1 IN2;
45 ADDI SP 1;
46 STOREIN SP IN1 1;
47 # Ref(Attr(Stack(Num('1')), Name('attr2')))
48 LOADIN SP IN1 1;
49 ADDI IN1 1;
50 STOREIN SP IN1 1;
51 CALL INPUT ACC;
52 SUBI SP 1;
53 STOREIN SP ACC 1;
54 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
55 LOADIN SP IN1 2;
56 LOADIN SP IN2 1;
57 MULTI IN2 1;
58 ADD IN1 IN2;
59 ADDI SP 1;
60 STOREIN SP IN1 1;
61 # Assign(Stack(Num('1')), Stack(Num('2')))
62 LOADIN SP IN1 1;
63 LOADIN SP ACC 2;
```

Code Generierung

RETI Pass, Teil 4

```
64 ADDI SP 2;
65 STOREIN IN1 ACC 0;
66 # // Return(Num('1'))
67 # Exp(Num('1'))
68 SUBI SP 1;
69 LOADI ACC 1;
70 STOREIN SP ACC 1;
71 # Return(Stack(Num('1')))
72 LOADIN SP ACC 1;
73 ADDI SP 1;
74 LOADIN BAF PC -1;
75 # // Assign(Name('pntr_on_ar_of_sts'), Ref(Name('ar_of_sts')))
76 # Ref(Global(Num('0')))
77 SUBI SP 1;
78 LOADI IN1 0;
79 ADD IN1 DS;
80 STOREIN SP IN1 1;
81 # Assign(Global(Num('18')), Stack(Num('1')))
82 LOADIN SP ACC 1;
83 STOREIN DS ACC 18;
84 ADDI SP 1;
```


Code Generierung

RETI Pass, Teil 5

```
85 # // Assign(Name('res'), Call(Name('fun'), [Name('pntr_on_ar_of_sts'))])
86 # StackMalloc(Num('2'))
87 SUBI SP 2;
88 # Exp(Global(Num('18'))))
89 SUBI SP 1;
90 LOADIN DS ACC 18;
91 STOREIN SP ACC 1;
92 # NewStackframe(Name('fun.3'), GoTo(Name('addr@next_instr'))))
93 MOVE BAF ACC;
94 ADDI SP 3;
95 MOVE SP BAF;
96 SUBI SP 3;
97 STOREIN BAF ACC 0;
98 LOADI ACC 78;
99 ADD ACC CS;
100 STOREIN BAF ACC -1;
101 # Exp(GoTo(Name('fun.3'))))
102 JUMP -76;
103 # RemoveStackframe()
104 MOVE BAF IN1;
105 LOADIN IN1 BAF 0;
```

Code Generierung

RETI Pass, Teil 6

```
106 MOVE IN1 SP;
107 # Exp(ACC)
108 SUBI SP 1;
109 STOREIN SP ACC 1;
110 # Assign(Global(Num('19')), Stack(Num('1')))
111 LOADIN SP ACC 1;
112 STOREIN DS ACC 19;
113 ADDI SP 1;
114 # // If(Name('res'), [])
115 # // IfElse(Name('res'), [], [])
116 # Exp(Global(Num('19')))
117 SUBI SP 1;
118 LOADIN DS ACC 19;
119 STOREIN SP ACC 1;
120 # IfElse(Stack(Num('1')), [], [])
121 LOADIN SP ACC 1;
122 ADDI SP 1;
123 JUMP== 51;
124 # // not included Exp(GoTo(Name('if.1')))
125 # Ref(Global(Num('18')))
126 SUBI SP 1;
```

Code Generierung

RETI Pass, Teil 7

```
127 LOADI IN1 18;
128 ADD IN1 DS;
129 STOREIN SP IN1 1;
130 # Exp(Num('0'))
131 SUBI SP 1;
132 LOADI ACC 0;
133 STOREIN SP ACC 1;
134 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
135 LOADIN SP IN2 2;
136 LOADIN IN2 IN1 0;
137 LOADIN SP IN2 1;
138 MULTI IN2 18;
139 ADD IN1 IN2;
140 ADDI SP 1;
141 STOREIN SP IN1 1;
142 # Exp(Num('2'))
143 SUBI SP 1;
144 LOADI ACC 2;
145 STOREIN SP ACC 1;
146 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
147 LOADIN SP IN1 2;
```

Code Generierung

RETI Pass, Teil 8

```
148 LOADIN SP IN2 1;
149 MULTI IN2 6;
150 ADD IN1 IN2;
151 ADDI SP 1;
152 STOREIN SP IN1 1;
153 # Exp(Num('1'))
154 SUBI SP 1;
155 LOADI ACC 1;
156 STOREIN SP ACC 1;
157 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
158 LOADIN SP IN1 2;
159 LOADIN SP IN2 1;
160 MULTI IN2 3;
161 ADD IN1 IN2;
162 ADDI SP 1;
163 STOREIN SP IN1 1;
164 # Ref(Attr(Stack(Num('1')), Name('attr2')))
165 LOADIN SP IN1 1;
166 ADDI IN1 1;
167 STOREIN SP IN1 1;
168 # Exp(Num('1'))
```

Code Generierung

RETI Pass, Teil 9

```
169 SUBI SP 1;
170 LOADI ACC 1;
171 STOREIN SP ACC 1;
172 # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
173 LOADIN SP IN1 2;
174 LOADIN SP IN2 1;
175 MULTI IN2 1;
176 ADD IN1 IN2;
177 ADDI SP 1;
178 STOREIN SP IN1 1;
179 # Exp(Stack(Num('1')))
180 LOADIN SP IN1 1;
181 LOADIN IN1 ACC 0;
182 STOREIN SP ACC 1;
183 LOADIN SP ACC 1;
184 ADDI SP 1;
185 CALL PRINT ACC;
186 # Exp(GoTo(Name('if_else_after.0')))
187 # // not included Exp(GoTo(Name('if_else_after.0')))
188 # Return(Empty())
189 LOADIN BAF PC -1;
```

Literatur

Vorlesungen



Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL:

https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157
(besucht am 09.07.2022).



— „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03.08.2022).

Online



Clockwise/Spiral Rule. URL:

<https://c-faq.com/decl/spiral.anderson.html> (besucht am 29.07.2022).