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

27. September 2022

Universität Freiburg, Lehrstuhl für Betriebssysteme

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"	*L_Program*
		| "7" | "8" | "9"	
dig_with_0	::=	"0" | *dig_no_0*	
num	::=	"0" | *dig_no_0* *dig_with_0*$*$ | "$-$" *dig_no_0*$*$	
letter	::=	"*a*" ... "*Z*"	
name	::=	*letter*(*letter* | *dig_with_0* | _)$*$	
reg	::=	"*ACC*" | "*IN*1" | "*IN*2" | "*PC*" | "*SP*"	
		| "*BAF*" | "*CS*" | "*DS*"	
arg	::=	*reg* | *num*	
rel	::=	"==" | "!=" | "<" | "<=" | ">"	
		| ">=" | "*_NOP*"	

Grammatik 1: Grammatik des Lexers für die Sprache L_{RETI} in EBNF.

RETI-Architektur

Grammatik

instr	::=	*"ADD" reg arg* | *"ADDI" reg num* | *"SUB" reg arg*	*L_Program*
	|	*"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 2: 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 .

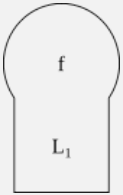
Weitere Definitionen

Interpreter

- führt Anweisungen „direkt“ aus.

T-Diagramm Programm

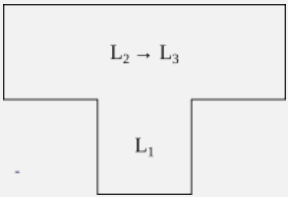
- in der Sprache L_1 geschrieben und berechnet Funktion f.



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"	*L_Lex*
ADD_OP	::=	"+"		
MUL_OP	::=	"∗"		

Grammatik 3: Grammatik des Lexers

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

Abbildung 1: Aus Eingabewort Tokens generieren.

Lexikalische Analyse

Definitionen

Token

- Tupel (T, V), wobei:
  - Tokentyp $T \mathrel{\hat{=}}$
    - 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 $V \mathrel{\hat{=}}$ 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 4: 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: $lex : \Sigma^* \to (T \times V)^*, w \mapsto (t_1, v_1) \dots (t_n, v_n)$.

Syntaktische Analyse

Ausgelassene Zwischenschritte

NUM	::=	"4" | "2"	*L_Lex*
ADD_OP	::=	"+"	
MUL_OP	::=	"∗"	
mul	::=	*mul MUL_OP NUM* | *NUM*	*L_Parse*
add	::=	*add ADD_OP mul* | *mul*	

Grammatik 5: Grammatik des Parsers unten und Grammatik des Lexers oben

- Tokentypen T sind in der Grammatik des Parsers Terminalsymbole

$$add \Rightarrow mul \Rightarrow mul\ \ MUL\_OP\ \ NUM \Rightarrow NUM\ \ MUL\_OP\ \ NUM \Rightarrow^* \text{"4"}\ \ \text{"}*\text{"}\ \ \text{"2"}$$

Syntaktische Analyse

Ausgelassene Zwischenschritte

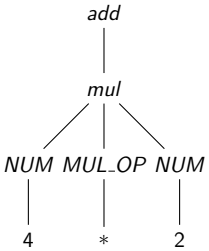


Abbildung 2: Formaler Ableitungsbaum

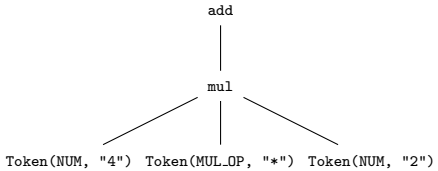
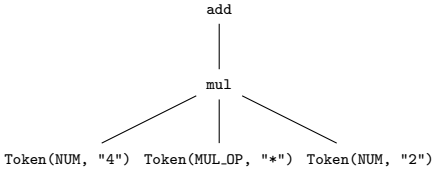


Abbildung 3: Compilerinterner Ableitungsbaum

Syntaktische Analyse

Ausgelassene Zwischenschritte

"4 * 2"



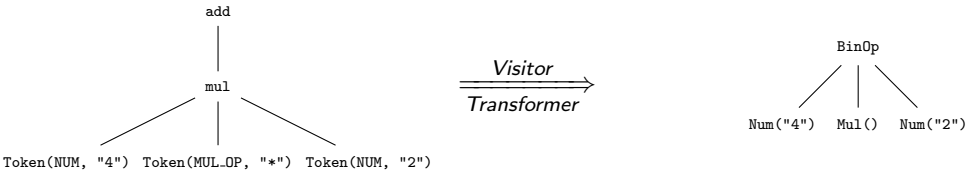
- Lexer ist Teil des Parsers.

Syntaktische Analyse

Ausgelassene Zwischenschritte

bin_op	::=	$Add() \mid Mul()$
exp	::=	$BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle) \mid Num(\langle str \rangle)$

Grammatik 6: Produktionen für Abstrakten Syntaxbaum



Syntaktische Analyse

Lark Parsing Toolkit

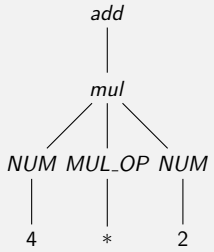
- erleichtert Syntaktische Analyse.
- Grammatik spezifizieren nach der Lark ein Eingabewort parst und einen Ableitungsbaum generiert.
- Earley Parser implementiert.
- Implementierung von Visitor und Transformer.
- Quellcode: `https://github.com/lark-parser/lark`.
- Dokomentation: `https://lark-parser.readthedocs.io/en/latest/index.html`.

Syntaktische Analyse

Definitionen

Formaler Ableitungsbaum

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

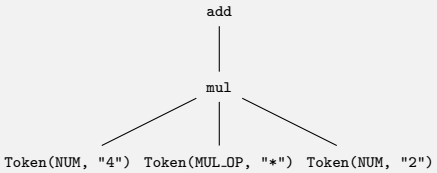


Syntaktische Analyse

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



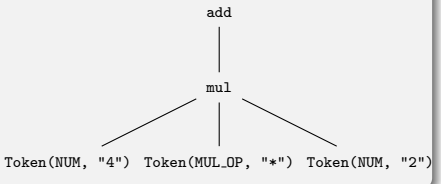
Syntaktische Analyse

Definitionen

Parser

- generiert aus Eingabewort einen compilerinternen Ableitungsbaum
- beeinhaltet 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

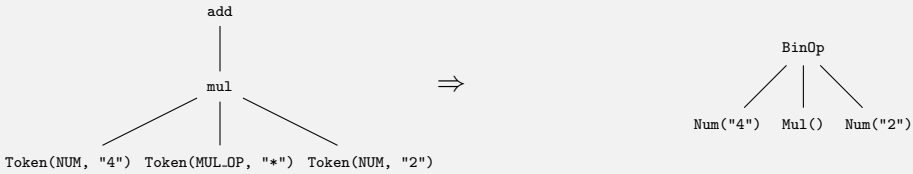
- von unten-nach-oben nach Prinzip der Breitensuche über Ableitungsbaum.
- generiert Abstrakten Syntaxbaum, indem beim Antreffen bestimmter Knoten des Ableitungsbaumes, diese durch Knoten des Abstrakten Syntaxbaumes ersetzt werden.

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 erlaut es schnell herauszufinden aus welchen Bestandteilen der Sprache mit unterscheidbarer Semantik diese zusammengesetzt ist.



Code Generierung

Definitionen

Konkrete Syntax

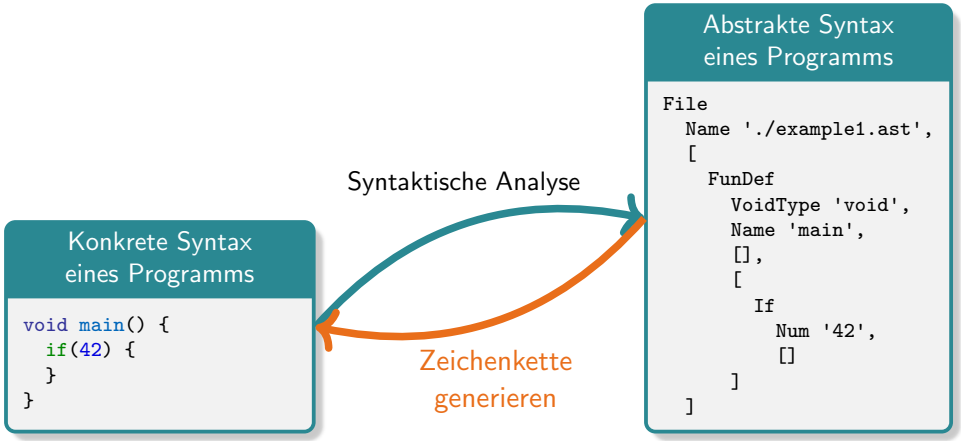
- bezeichnet den Aufbau von Programmen, wie man sie in eine Textdatei schreibt, um sie kompilieren zu lassen.

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 `*(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)`.
- Derefenzierung `*(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)` mithilfe von `Block(name, stmts_instrs-`, `GoTo(lable)`- 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

- $sym : \{my\_var, my\_fun, \ldots\} \rightarrow \{Adresse, Datentyp, \ldots\}^n,\ Bezeichner \mapsto (Information_1, \ldots, Information_n)$.
- um während dem Kompiliervorang Informationen zu speichern, die später nicht mehr so einfach zugänglich sind.

Code Generierung

PicoC-ANF Pass

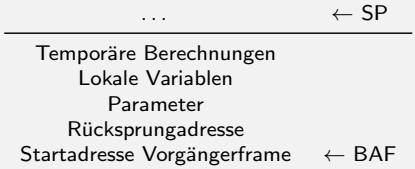
- alle Funktionen, außer der `main`-Funktion besitzen einen Stackframe für Lokale Variablen und Parameter.
- Globale Statische Daten sind Globale Variablen, sowie Lokale Variablen und Parameter der `main`-Funktion.

Code Generierung

PicoC-ANF Pass

Stackframe

- Datenstruktur, um Zustand einer Funktion zur Laufzeit zu „konservieren“.
- in einem Stack übereinander gestappelt 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.

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

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 Monadischer 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

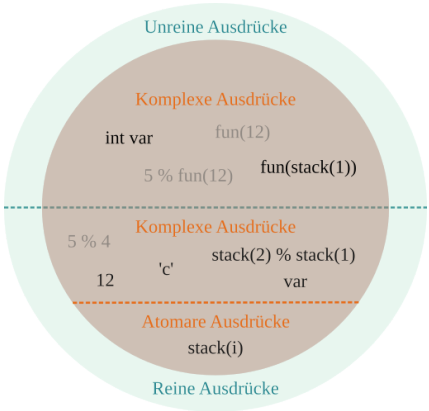


Abbildung 4: Überblick über Komplexe, Atomare, Unreine und Reine Ausdrücke.

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 `Im(str)` in Befehlen $< -(2^{21})$ oder $> 2^{21} - 1$.
    - Bitshiften und Anwenden von Bitweise ODER.
    - Immediate $< -(2^{31})$ oder $> 2^{31} - 1 \Rightarrow$ `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 duch einen Immediate mit passender Distanz / Adresse oder einen Sprungbefehl mit passender Distanz `Jump(Always(), Im(str(distance)))` ersetzt.

Code Generierung

RETI Pass

- $adr_{danach} = \#Bef_{vor\ akt.\ Bl.} + idx + 4$
- $Dist_{Zielbl.} = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases}$

Code Generierung

RETI Pass

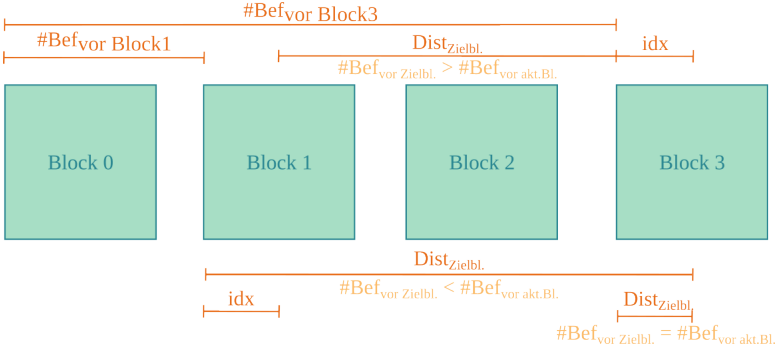


Abbildung 5: Veranschaulichung Distanzberechnung.

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. `var / 0`).

Tabelle 2: Fehlerarten, die zur Laufzeit auftreten.

Fehlermeldungen

Kategorien

Fehlerkategorie	Beschreibung
`UnknownIdentifier`	Es wird ein Zugriff auf einen Bezeichner gemacht (z.B. `unknown_var + 1`), der noch nicht deklariert und ist daher nicht in der Symboltabelle aufgefunden werden kann.
`UnknownAttribute`	Der Verbundstyp (z.B. `struct st {int attr1; int attr2;}`) auf dessen Attribut im momentanen Kontext zugegriffen wird (z.B. `var[3].unknown_attr`) besitzt das Attribut (z.B. `unknown_attr`) auf das zugegriffen werden soll nicht.
`ReDeclarationOrDefinition`	Ein Bezeichner von z.B. einer Funktion oder Variable, der bereits deklariert oder definiert ist (z.B. `int var`) wird erneut deklariert oder definiert (z.B. `int var[2]`). 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 intialisierten Konstante (z.B. `const int const_var = 42`) ein Wert zugewiesen wird (z.B. `const_var = 41`). Der einzige Weg, wie eine Konstante einen Wert erhält ist bei ihrere Initialisierung.
`PrototypeMismatch`	Der Prototyp einer deklarierten Funktion (z.B. `int fun(int arg1, int arg2[3])`) stimmt nicht mit dem Prototyp der späteren Definition dieser Funktion (z.B. `void fun(int arg1[2], int arg2) { }`) überein.
`ArgumentMismatch`	Wenn die Argumente eines Funktionsaufrufs (z.B. `fun(42, 314)`) nicht mit dem Prototyp der Funktion die aufgerufen werden soll (z.B. `void fun(int arg[2]) { }`) 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

Kommandozeilenargumente `<cli-options>`

Kommandozeilen-option	Beschreibung	Standard-wert
`-i`, `--intermediate-stages`	Gibt Zwischenstufen der Kompilierung in Form der verschiedenen Tokens, Ableitungsbäume, Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateiendungen aber gleichem Basisnamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal.	`false`, most_used: `true`
`-p`, `--print`	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

Kommandozeilenargumente `<cli-options>`, Teil 2

Kommandozeilen-option	Beschreibung	Standard-wert
`-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 aktivert 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 zustätzliche 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

Kommandozeilenargumente `<cli-options>`, Teil 3

Kommandozeilen-option	Beschreibung	Standard-wert
`-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

Kommandozeilenargumente <cli-options>, Teil 4

Kommandozeilen-option	Beschreibung	Standard-wert
-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 Zustsand 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 zusätlich 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

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

Shell-Mode

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

Bedienung

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

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

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.	12
`error`	Fehlermeldungen testen. Keine Verifikation wird ausgeführt.	67
`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

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).

Online