PCCC

Pseudocode C Compiler

Mario Preishuber 1120643, Thomas Hütter 1120239

Introduction to Compiler Contruction

Univ.-Prof. Dr. Christoph Kirsch
Sommersemester 2013

Fachbereich Computerwissenschaften

UNIVERSITÄT SALZBURG

Mario Preishuber, 1120643
Thomas Hütter, 1120239

Fachbereich Computerwissenschaften
Universität Salzburg

Inhaltsverzeichnis

Mario Preishuber, 1120643
Thomas Hütter, 1120239

Fachbereich Computerwissenschaften
Universität Salzburg

Allgemein

Dies ist die Dokumentation des Pseudocode C Compilers (PCCC), welcher im Rahmen der Lehrveranstaltung „Grundlagen Compilersysteme“, gehalten von Univ.-Prof. Dr. Christoph Kirsch, von Mario Preishuber und Thomas Hütter erstellt wurde.
Bei diesem Compiler handelt es sich um einen Single Pass Compiler, welcher für ein Subset von C entwickelt wurde. Implementiert ist der PCCC in genau diesem Subset von C und erzeugt wird ein Bytecode für einen selbst in C geschriebenen DLX Emulator.
In dieser Dokumentation erwähnt sind Details, Notizen und Erklärungen, die über den Inhalt der Lehrveranstaltung hinausgehen.

Features

Enthaltene Funktionen

- Datentypen (inklusive Typprüfung)
  - Integer, Character, Boolean
  - Arrays, Strings, Structs
  - typedef
- Conditional
  - if / else
- Loops
  - while
- Prozeduren
  - Beliebige Rückgabewerte
  - Beliebige Parameter
- Bool'sche Operationen (lazy evaluation)
  - <, >, <=, >=, ==, !=, &&, ||
- Arithmetische Operationen (konstante Ausdrücke werden gleich ausgewertet)
  - +, -, *, /
- Datei Operationen
  - open, close, read, write
- Sonstige Funktionen
  - printf
  - malloc

Nicht enthaltene Funktionen

- Seperate Compilation
- Mehrdimensionale Arrays

Einschränkungen

- Beim Aufbau eines Programms müssen einige Regeln beachtet werden:
  - Includes (nicht implementiert) müssen am Anfang stehen, anschließend alle Struct-Deklarationen und darauf können nun globale Variablen oder Prozeduren folgen.

Mario Preishuber, 1120643
Thomas Hütter, 1120239
Fachbereich Computerwissenschaften
Universität Salzburg

    - Auch in einer Prozedur müssen alle lokalen Variablen zu Beginn deklariert werden.
- Übergang zwischen Deklarationen und Statements in Prozeduren nicht eindeutig, daraus würde sich eine Fehlermeldung bei der Typprüfung am ersten Statement ergeben.
- Die erste globale Variable, nach den Struct Definitionen, darf keine Struct Variable sein.
- Vergleichsoperatoren müssen geklammert werden.

Implementierung

Scanner

Im Scanner gibt es keine Besonderheiten, es wird Zeichen für Zeichen gelesen und ausgewertet. Trifft eine Zeichenfolge auf ein gewünschtes Muster zu, wird ein Token erstellt. In der folgenden Liste sind die verwendeteten Tokens, samt ID, aufgelistet. Diese sind als globale int Variablen definiert.

```
ERROR     = -1;
INIT      =  0;
LSQBR     =  1;  /* [ */
RSQBR     =  2;  /* ] */
LPAR      =  3;  /* ( */
RPAR      =  4;  /* ) */
LCUBR     =  5;  /* { */
RCUBR     =  6;  /* } */

SEMCOL    =  7;  /* ; */
COMMA     =  8;  /* , */
DQUOTE    =  9;  /* " */
QUOTE     = 10;  /* ' */
EQSIGN    = 11;  /* = */
PLUS      = 12;  /* + */
MINUS     = 13;  /* - */
TIMES     = 14;  /* * */
DIV       = 15;  /* / */

LT        = 16;  /* < */
GT        = 17;  /* > */
EQ        = 18;  /* == */
NEQ       = 19;  /* != */
LET       = 20;  /* <= */
GET       = 21;  /* >= */
AND       = 22;  /* && */
OR        = 23;  /* || */
```

```
INT           = 30;  /* key-word: int    */
CHAR          = 31;  /* key-word: char   */
VOID          = 32;  /* key-word: void   */
STRUCT        = 33;  /* key-word: struct */
TYPEDEF       = 34;  /* key-word: typedef*/
BOOL          = 35;  /* boolean value    */
IDENT         = 36;  /* identifier       */
NUMBER        = 37;  /* number value     */
STRING        = 38;  /* string value     */
CHARACTER     = 39;  /* character value  */

IF            = 40;  /* key-word: if     */
ELSE          = 41;  /* key-word: else   */
WHILE         = 42;  /* key-word: while  */
RETURN        = 43;  /* key-word: return */

COMMENT       = 50;  /* comment (this)   */
DOT           = 51;  /* .                */
INCLUDE       = 52;  /* #include         */
LF            = 53;  /* \n               */
END           = 54;  /* last token       */
ARROW         = 55;  /* ->               */
SIZEOF        = 56;  /* size of type     */
MALLOC        = 57;  /* allocate memory  */
PRINTF        = 58;  /* write to output  */

OPEN          = 60;  /* open file        */
CLOSE         = 61;  /* close file       */
READ          = 62;  /* read file        */
WRITE         = 63;  /* write file       */
```

Parser

Beim Parser des PCCC handelt es sich um einen recursive decent parser mit einem look ahead von 1. Die Codegenerierung erfolgt nach dem Single Pass Prinzip, also zum ehest möglichen Zeitpunkt.

Die zugrunde liegende EBNF des umgesetzten Subsets folgt hier:

```
programm     = {include} {structDec} {globalDec} .

GlobalDec    = {typedefDec} [struct] typeSpec ["*"]
               ( procImpl | identifier ";" ) .

procImpl     = identifier formalParams
               ( ";" | ( "{" varDecSeq statementSeq "}" ) ) .
formalParams = "(" [ formalParam { "," formalParam } ] ")" .
formalParam  = basicArrType identifier .
BasicArrType = typeSpec ["*"] .

statementSeq = { (ifCmd | whileLoop | printf | expression | procRet) ";" } .
block        = "{" statementSeq "}" .

typedefDec   = "typedef" ["struct"] typeSpec ["*"] identifier ";" .
structDec    = "struct" identifier "{" varDecSeq {varDecSeq} "}" ";" .
varDecSeq    = { [typedefDec]["struct"] typeSpec ["*"] identifier ";" } .

procCall     = identifier actParams .
ActParams    = "(" [expression { "," expression }] ")" .

ifCmd        = "if" "(" expression ")" block ["else" block] .
WhileLoop    = "while" "(" expression ")" block .
ProcRet      = "return" expression .

FileFct      = fileOpen | fileClose | fileRead | fileWrite .
FileOpen     = "open" "(" expression "," expression "," expression ")" .
fileClose    = "close" "(" expression ")" .
fileRead     = "read" "(" expression "," expression "," expression ")" .
fileWrite    = "write" "(" expression "," expression "," expression ")" .

selector     = ("->" identifier | "[" expression "]") .
printf       = "printf" "(" (string | expression) ")" .
malloc       = "malloc" expression .
Sizeof       = "sizeof" "(" ["struct"] typeSpec ")" .
include      = "include" ["<" | """] identifier "." identifier [">" | """] .

expression   = arithExp { compOp arithExp } .
arithExp     = [ "-" ] term { ( "+" | "-" | "||" ) term } .
term         = factor { ( "*" | "/" | "&&" ) factor } .
factor       = number | character | string | ( "(" expression ")" ) | sizeof |
               malloc | fileFct | ( ["*"] identifier [ (selector |
               ("=" expression) | op |
               ("(" procCall ")") ) (";" | "," | "}" | "->" | ")") ] ).

typeSpec     = "int" | "char" | "void" | identifier .

Op           = "+" | "-" | "*" | "/" | "&&" | "||" .
compOp       = "==" | "!=" | "<=" | ">=" | "<" | ">" .

identifier   = letter {letter | digit | "_"} .
number       = ["-"] digit {digit} .
String       = """ { digit | letter } """ .
character    = "'" ["\"](digit | letter) "'".
digit        = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
letter       = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
               "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
               "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" |
               "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
               "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" .
```

Codegenerierung

Eine zwingende Bedingung an ein Programm, das vom PCCC kompiliert wird, ist das Enthalten einer Main Methode, was folgende Auswirkungen auf den erzeugten Code hat. Wird die Main Methode gefunden, merkt sich der Compiler die Stelle im Programm und fügt als ersten Befehl, nach den Metadaten, einen Sprung zu Main ein. Weiters wird automatisch ein Trap Befehl sowohl am Ende der Main Methode, sowie als letzter Befehl eingefügt.

Symbol Table

Die Symbol Table wurde als einfach verkettete Liste implementiert, die die Objekte über ein Next Element verbindet. Als mögliche Funktionen stehen diese zur Verfügung:

- `int insert(struct object_t head, struct object_t object)`
- `struct object_t lookUp(struct object_t head, char *name)`
- `struct object_t findProcedureObject(struct object_t head, char *name)`

Die Methoden sind selbsterklärend, `insert` fügt ein Element am Ende der mit `head` übergebenen Liste an. `lookUp` sucht in der mit `head` übergebenen Tabelle nach `name` und gibt das gefundene Object oder `0` zurück. Die Methode `findProcedureObject` ist nur eine spezialisierte Form von `lookUp`, die als zusätzliche Suchbedingung nur nach Prozeduren sucht.

Im Parser ist die Symbol Table durch 2 Instanzen davon eingebunden, eine für die lokalen Objekte und eine für die globalen. Die lokale wird bei jedem Auftreten einer Prozedur neu initialisiert und die globale bleibt während der vollen Programmausführung erhalten.

Target Machine

Bei der Target Machine handelt es sich um einen selbstgeschriebenen erweiterten DLX Emulator. Mit erweitert ist gemeint, dass folgende zusätzliche Kommandos zur Verfügung stehen:

Metadaten

- Anzahl der Kommandos: cs
- Benötigter Speicher für Globale Variablen: gp
- Benötigter Speicher für Strings: sp

Dateioperationen

- Öffnen: flo(int a, int b, int c);
- Schliessen: flc(int c);
- Lesen: rdc(int a, int b, int c);
- Schreiben: wrc(int a, int b, int c);

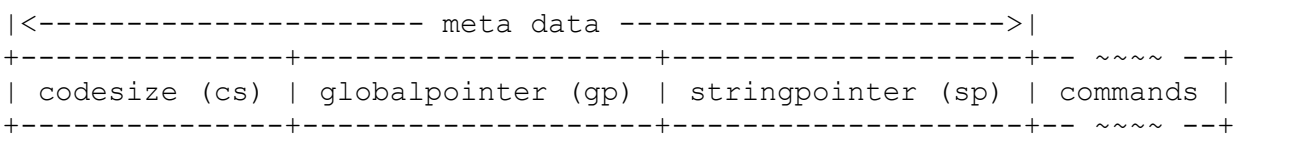
Malloc

- mal(int a, int b, int c);

Ausgabe

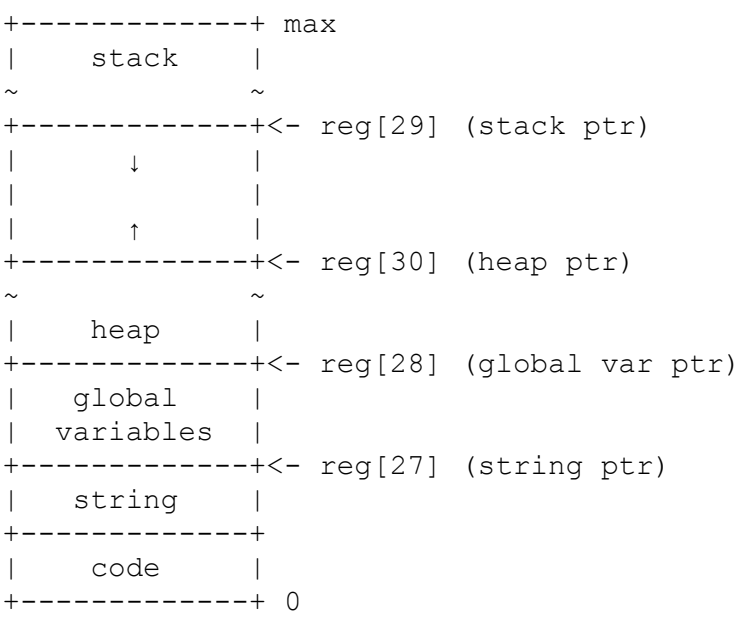
- Ausgabe einer Zahl: prn(int a);
- Ausgabe eines Zeichens: prc(int a);

Die Target Machine verarbeitet Binär Dateien, welche wie in der Vorlesung erklärt, kodiert sind. Damit eine Datei korrekt ausgeführt werden kann, muss das untenstehende Format einhalten werden.



Wie die Grafik zeigt sind einige Metadaten nötig. Das erste Kommando gibt die Anzahl der Kommandos des enthaltenen Programms an (ohne die Metadaten). Als zweites folgt der Globalpointer, welcher angibt wieviel Speicher für globale Variablen benötigt wird. Der dritte und letzte Wert ist der Stringpointer, dieser gibt den, für Strings, benötigten Speicher an.

Speicherstruktur



Die oben stehende Grafik zeigt die Struktur des DLX Emulators. Am unteren Ende stehen die Kommandos es folgen die Strings und globalen Variablen. Diese drei Bereiche sind konstant, deren Größe wird durch die Metadaten bestimmt und kann nicht während der Laufzeit vergrößert oder verkleinert werden. Im Anschluss an die globalen Variablen folgt der Heap. Dieser enthält alle während der Laufzeit, durch ein `malloc`, erzeugten Variablen und wächst nach oben. Theoretisch kann der Heap durch die Verwendung des `free`-Kommandos auch schrumpfen, jedoch wurde kein `free` implementiert. Am Ende des Speichers befindet sich der Stack. Dieser wächst dem Heap entgegen, seine Verwendung dient Prozeduraufrufen, Übergabeparametern und lokalen Variablen.

Registerbelegung

Register	Registerbedeutung
0	hat immer den Wert 0
25	Return Register
26	Frame Pointer
27	String Pointer
28	Global Pointer
29	Stack Pointer
30	Heap Pointer
31	Link

Selfcompilation

Anmerkungen

Zur Selfcompilation wird vom PCCC eine Datei `selfcomp.c` als Input verwendet, die sich geringfügig vom Compiler selbst in folgenden Punkten unterscheidet:

- `printf`: Die Ausgabefunktion hat in C eine andere Syntax als hier. Printf ist in den zwei Optionen `printf(`*`String`*`)` und `printf(`*`Variable`*`)` enthalten.
- `bool`: Da in C der Typ `bool` nicht existiert, wurde dieser in `pccc.c` als `typedef` implementiert.

Die erzeugte Datei `out_selfcomp` enthält den Compiler der die Datei `selftest.c` compiliert. Mit der Target Machine kann nun die erzeugte Datei `out_selftest` ausgeführt werden.

Ergebnis

Der erzeugte Compiler funktioniert leider nur sehr eingeschränkt. Die größten Defizite sind hier vor allem im Speichermanagement hinzunehmen. Während Prozeduren, Structs, If/Else, lokale und globale Variablen funktionieren, verhindern Schleifen, Arrays oder Strings die Erzeugung des Programms. Die zur Verfügung stehenden Funktionen sind in einem kleinen Testfile `selfTest.c` zusammengefasst.

Quellen

[1] Niklaus Wirth. Compiler Construction. Addison-Wesley, 1996
[2] Univ.-Prof. Dr. Christoph Kirsch. Notes: Introduction to Compiler Construction. 2013

Mario Preishuber, 1120643
Thomas Hütter, 1120239
Fachbereich Computerwissenschaften
Universität Salzburg