Fachbereich für Computerwissenschaften
Universität Salzburg

Compiler Construction - Projekt Dokumentation - SS 2011

mimimiC

Michael KLEBER E-Mail: michael.kleber@sbg.ac.at
Yusuf ÖZBEK E-Mail: yusuf.oezbek@student.uibk.ac.at

Betreuer: Prof. Christoph Kirsch

Inhaltsverzeichnis

1. Einleitung

Der in dieser Dokumentation beschriebene Compiler „mimimiC“ ist als Projektaufgabe im Rahmen der VP „Grundlagen Compilersysteme“ an der Universität Salzburg von Prof. Kirsch entstanden. Das Ziel dieser LVU war es, einen Compiler in einer beliebigen Sprache von Grund auf zu entwickeln und zu implementieren. Daher wird es in dieser Ausarbeitung Schritt für Schritt beschrieben, wie wir unseren Compiler entwickelt bzw. implementiert haben.

2. Unterstützte Sprachen

2.1. Schlüsselwörter

Die vom mimimiC Compiler unterstützte Schlüsselwörter sind folgendermaßen vordefiniert:

```
1   #define IDENTIFIER          // Variablen- und Funktionsbezeichner
2   #define INT                 // int
3   #define CHAR                // char
4   #define STRUCT              // struct
5   #define STRUCTCHARACTER     // struct symbol
6   #define ANDOPERATOR         // &
7   #define CHARACTER           // a|...z , A| ...|Z
8   #define NUMERIC             // 0|...|9
9   #define STRING              // string
10  #define IF                  // if
11  #define ELSE                // else
12  #define WHILE               // while
13  #define TYPEDEF             // typedef
14  #define INCLUDE             // include
15  #define RETURN              // return
16  #define ERROR               // error
17  #define ENDOFFILE           // Ende der Datei
18  #define HASH                // #
19  #define ADDITION            // +
20  #define SUBTRACTION         // -
21  #define DIVISION            // /
22  #define MULTIPLICATION      // *
23  #define GREATER             // >
24  #define LESSER              // <
```

```
25  #define LOGICALAND          // &&
26  #define LOGICALOR           // ||
27  #define MODULO              // %
28  #define DOT                 // .
29  #define COMMA               // ,
30  #define SEMICOLON           // ;
31  #define LOGICALNOT          // !
32  #define LEFTPARANTHESIS     // (
33  #define RIGHTPARANTHESIS    // )
34  #define LEFTBRACKET         // [
35  #define RIGHTBRACKET        // ]
36  #define LEFTBRACE           // {
37  #define RIGHTBRACE          // }
38  #define SINGLEQUOTE         // '
39  #define DOUBLEQUOTE         // „
40  #define ASSIGNMENT          // =
41  #define BACKSLASH           // \
42  #define NULLCHARACTER       // null
```

2.2. Input-Sprache

Unser Compiler ist vollständig in C geschrieben und akzeptiert eine Sprache, die stark an C angelehnt ist. Die Eingabesprache unseres Compilers ist ein Subset von C. Dabei werden die wichtigsten Features von C unterstützt. mimimiC ist ein nativer Multi-Pass Compiler und erzeugt RISC-Code für unsere Virtuelle Maschine..

2.3. EBNF

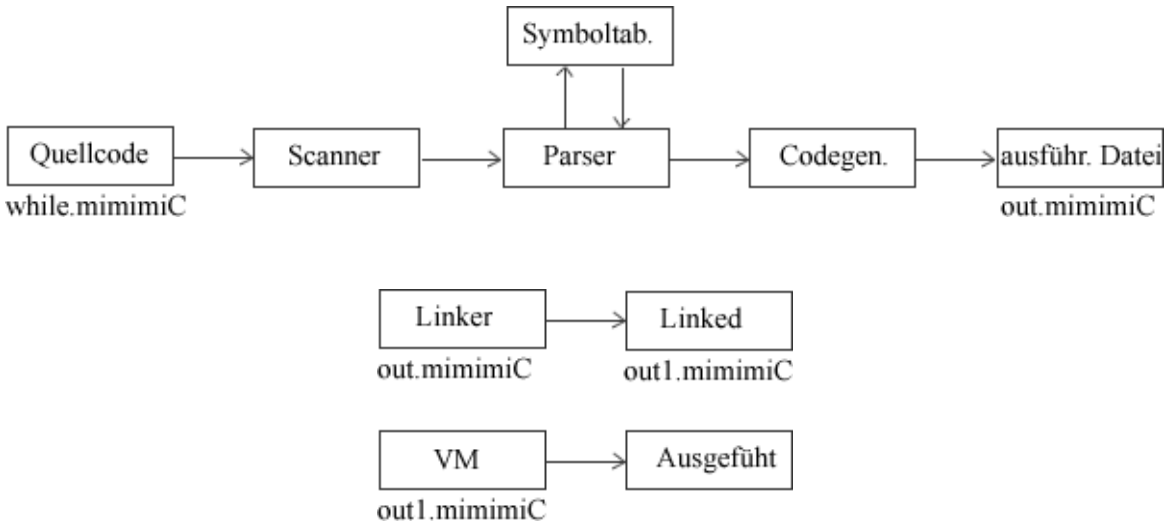
Die EBNF unseres Compilers wurde folgendermaßen definiert:

```
Scanner:
sign= '(' | ')' | '{' | '}' | '=' | '<' | '>' | '\' | '&' | '-' | ',' | '.' |
      '[' | ']' | '*' | '/' | '|' |'%'| ' | '#' | ';' | '\"' | '\\'.
digit= '0' | '1' | ... | '9'.
letter= 'a'..'z'A'..'Z'.
space= " " | "\n" | "\t " .
number= digit {digit} .
identifier= letter{letter|digit}.
character= letter | digit | sign | space .
string= doublequotes { character } doublequotes.

keywords= "int" | "char" | "struct" | "if" | "else" | "while" |  "typedef" |
          "include" | "return" | "null".
```

```
structSymbol= "struct".
intSymbol= "int".
charSymbol= "char".


Parser:

lesser= "<".
greater= ">".
and= "&&".
not= "!".
or= "||".

begin= {include} module.
relation= "==" | lesser |greater | "<=" | ">=" | "!=".
value= number | character | string | null.
module= {statement}.
statement= (typeDefinition) |(type identifier (variableDeclaration | function)).
variableDeclaration= {"[" ["NUMERIC"] "]"} ["=" expression] ";".

function= "(" [type identifier["["[expression]"]"] {"," type
          identifier["["[expression]"]"]}] ")" (codeunit | ";").

include= "#" "include" ( string | (lesser identifier ["/"identifier] "."
identifier greater) ).

variableDefinition= type variable.
variable= identifier{"[" [expression] "]"}.
typeDefinition= "typedef" type variable ";".
type= (simpleType | identifier).
simpleType= ( intSymbol | charSymbol | structSymbol ).
struct= structSymbol /*[identifier]*/ [ "{" { vardefinition ";" } "}"].
expression= andExpression {or andExpression}.
andExpression= [not] relationalExpression {and [not] relationalExpression}.
relationalExpression= ["-"] addSubExpression [relation ["-"] addSubExpression].
addSubExpression= multdivExpression {("-"|"+") multdivExpression}.
multDivExpression= term {("*"|"/"|"%") term }.

term= value | (identifier (functioncall | ({"."identifier}["[" [expression]"]"])
      ) ) | "(" expression ")".

functionCall= "("[expression] {","expression} ")".
codeunit= "{" {code} "}".

code= (simpleType variable [assignment]";") | controlStructure |(identifier
      ((identifier["[" [expression] "]"] [assignment])|({"."identifier}["["
      [expression] "]"] assignment) | functionCall ) ";") .

assignment= "=" expression.
controlStructure= ifStructure | whileStructure |"return" [expression] ";".
ifStructure= "if" "(" expression ")" codeUnit ["else" (ifStructure | codeUnit)].
whileStructure= "while" "(" expression ")" codeUnit.
```

3. Aufbau

Die gesamte Aufbau unseres Compilers sieht graphisch folgendermaßen aus:



3.1. Scanner

Unser Scanner liest Zeichen einzeln (mit Readahead von eins, falls benötigt) von einer vorgegebenen Quelldatei, in der ein Programmcode mit der Programmierungssprache C geschrieben wird, ein und fügt bzw. substituiert nach den vordefinierten Regeln/Restriktionen die Zeichen zu einem Token zusammen. Die allgemeine Aufgabe des Scanners ist, eine lexikalische Analyse der eingelesenen Zeichen durchzuführen. Nach dem Programmstart wird der Scanner vom Parser aufgerufen und die vom Scanner erzeugten Tokens, an den Parser weitergeleitet.

Die Methode `nextToken` wird vom Parser aufgerufen und liest das erste Zeichen vom Quelltext ein. Wenn die Datei leer ist, wird als Rückgabe `ENDOFFILE` ausgegeben. Danach wird die Methode `readChar` und `setCode` aufgerufen. In der `readChar` Methode wird das erste Zeichen vom Quellcode eingelesen und in den Tokens hinzugefügt Beim Lesen des Zeichens werden `isSpace`, `isAlphabet`, `isSymbol` und `isNumber` Methoden aufgerufen. In der `isSpace` Methode wird es bestimmt, ob das eingelesenes Zeichen ein blank, ein tabulator, oder eine newline ist. In der `isAlphabet` Methode wird es bestimmt, ob das eingelesene Zeichen von Buchstaben `a...z`, `A...Z` wie z.B. `int`, `if`, `while`, etc. entsteht. ist. In der `isSymbol` Methode wird es bestimmt, ob das eingelesene Zeichen ein Symbol `<`, `(`, `#` etc. ist. In der `isNumber` Methode wird es bestimmt, ob das eingelesene Zeichen eine Nummer `0...9` ist. In der `setCode` Methode werden die eingelesenen Wörter mit den vordefinierten Wörtern und Symbole identifiziert und ausgegeben. Dabei werden auch auf die Kommentare `//`, `/* */` in der Quelldatei beachtet.

Beispiel: befindet sich im Ordner: `tests/comment.mimimiC`

```
int main() /* long comment 1 */
{
  int x= 122+ 5; // short comment 1
  int y= x* 7;
  int z= y/ 2;  /* long comment 1 */
  int i= z% 30; // short comment 2

 // printf("x: %d \ny: %d \nz: %d \ni: %d\n ", x, y ,z ,i);

} //end main
```

Damit wir bestimmte Informationen über eingelesene Tokens zu haben werden die Informationen als `struct` mit den folgenden Werten in behandelt:

```
typedef struct
{
  int codeNumber; // numerische Darstellung des Token-Types
  char value[64]; // enthält einen 64 Byte char-Array, das speichert die
                  // gelesene Wert
  int line;       // Zeileninformation
  int column;     // Spalteninformation
  int length;     // Längeninformation
} Token;
```

3.2. Parser

Der Parser ist im allgemein eine Umsetzung der EBNF und führt die syntaktische Analyse der Quelldatei durch. Dies geschieht indem der Parser, die vom Scanner erhaltenen Tokens zu Sätzen zusammenbaut und diese anhand der vordefinierten Grammatikregeln auf Richtigkeit überprüft. Weiters hat der Parser die Aufgaben die Symboltabelle/Objekttabelle zu erstellen und die möglichen aufgetretenen Fehler zu behandeln. Die syntaktische Analyse erfolgt, nachdem der in der EBNF angegebenen Grammatikregeln für die Produktion. Jeder dieser Produktion entspricht einer Funktion im Parser.

Im Parser wird jedes Token entlang der EBNF verarbeitet, bis er die richtige Beschreibung gefunden hat. Dabei können wir den gesamten Quellcode analysieren, um Fehler für jeden falschen Code zu bestimmen. Wird vom Parser ein Fehler gefunden, dann erfolgt eine Ausgabe auf die Konsole mit der entsprechenden Fehlermeldung. Der Parser versucht weiterhin die restlichen Tokens auf ihre Richtigkeit zu überprüfen.

3.3. Symboltabelle

Die Symboltabelle erstellt eine Liste von Structs, um die Symboltabelle darzustellen. Jedes Symbol, wie `struct`, `function`, etc. werden gelesen und dann in die Symboltabelle hinzugefügt bzw. gespeichert. Wenn es zum Beispiel eine Variable zugewiesen wird, überprüft die grammatische Definition die Symboltabelle auf der reservierten Adresse und speichert dann den Wert zu dieser Adresse. Im Falle einer `function` oder eines `struct`, wird die relative Adresse in dem reservierten Speicher gespeichert. Dabei wird es entschieden ob die Variable local oder global ist.

Das Struct, das die einzelnen Symboltabelleneinträge repräsentiert, sieht folgendermaßen aus:

```
#define SIZE_int        1  // Länge des Datentypes int in Byte
#define SIZE_char       1  // Länge des Datentypes char in Byte
#define SIZE_adr        1  // Länge des Referenzes adr in Byte
#define SIZE_TYPE       20 // Länge des Referenzes type in Byte
#define SIZE_OBJECT     20 // Länge des Referenzes object in Byte
#define T_INT           1  // Typendefinition für int
#define T_CHAR          2  // Typendefinition für char
#define T_ARRAY         3  // Typendefinition für array
#define T_STRUCT        4  // Typendefinition für struct
#define T_FUNCTION      5  // Typendefinition für Funktion
#define CLASS_VARIABLE  1  // Klasse für Variablen
#define CLASS_FUNCTION  2  // Klasse für Funktionen
#define CLASS_TYPE      3  // Klasse für Typen
#define GLOBAL          0  // Bezeichner für globale Variable
#define MODE_VAR        1  // Variable
#define MODE_CONST      2  // Konstante
#define MODE_REG        3  // Register
#define MODE_COND       4  // Kondition
#define MODE_POINTER    5  // Pointer
#define MODE_PARAMETER  6  // Parameter

typedef struct _Type Type;
typedef struct _Object Object;
typedef struct _SymbolItem symbolItem;
typedef struct _SymbolType symbolType;
typedef struct _SymbolEntry symbolEntry;


struct _Type
{
  int form;       // Ein array, record oder simple type.
  int number;     // Eindeutige Nummer für jede Typ
  Type *base;     // Bezeichnet den Typ der Elemente, falls ein array
  int len;        // Die Länge, falls ein array.
  Object *fields; // Eine Liste von Objekten, falls ein record
};

struct _Object
{
  symbolItem *item;//
  Ident *name;    // Der Name von Variablen, Funktionen usw.
  Type *otype;    // Ein Pointer auf den Typ der Variable
  Object *nextO;  // Das nächste Element in der Symboltabelle
  int value;      // Der Wert
  int level;      //
```

```
  int class;      // Compiler unterscheidet zwischen folgenden Klassen:
                  // CLASS_VARIABLE: Variablen
                  // CLASS_FUNCTION: Funktionen
                  // CLASS_TYPE: Struktdefinitionen
};


struct _SymbolItem
{
  int mode;       //
  int a;          // offset a
  int b;          // offset b
  int r;          // register
  int c;          // condition <, <=, etc.
  Type *type;     //
  symbolEntry *nextEntry; //
};


struct _SymbolEntry {

};
```

3.4. Code Generierung

```
0000:0020 41 44 44 49 01 00 00 00 00 00 00 00 e4 f5 04 08 ADDI.
0000:0030 53 55 42 49 1d 00 00 00 1d 00 00 00 01 00 00 00 SUBI.
0000:0040 20 53 54 57 01 00 00 00 1e 00 00 00 ff ff ff ff  STW.
0000:0050 41 44 44 49 01 00 00 00 00 00 00 00 e4 f5 04 08 ADDI.
0000:0060 53 55 42 49 1d 00 00 00 1d 00 00 00 01 00 00 00 SUBI.
0000:0070 20 53 54 57 01 00 00 00 1e 00 00 00 fe ff ff ff  STW.
0000:0080 20 4c 44 57 01 00 00 00 1e 00 00 00 ff ff ff ff  LDW.
0000:0090 41 44 44 49 02 00 00 00 00 00 00 00 00 00 00 00 ADDI.
0000:00a0 41 44 44 49 03 00 00 00 00 00 00 00 e4 f5 04 08 ADDI.
0000:00b0 20 43 4d 50 02 00 00 00 02 00 00 00 03 00 00 00  CMP.
0000:00c0 20 42 4c 54 02 00 00 00 00 00 00 00 03 00 00 00  BLT.
0000:00d0 41 44 44 49 02 00 00 00 00 00 00 00 00 00 00 00 ADDI.
0000:00e0 20 42 45 51 02 00 00 00 00 00 00 00 02 00 00 00  BEQ.
0000:00f0 41 44 44 49 02 00 00 00 00 00 00 00 01 00 00 00 ADDI.
0000:0100 20 42 45 51 02 00 00 00 00 00 00 00 08 00 00 00  BEQ.
0000:0110 41 44 44 49 03 00 00 00 00 00 00 00 e4 f5 04 08 ADDI.
0000:0120 20 4c 44 57 04 00 00 00 1e 00 00 00 ff ff ff ff  LDW.
0000:0130 41 44 44 49 05 00 00 00 00 00 00 00 e4 f5 04 08 ADDI.
0000:0140 20 53 55 42 04 00 00 00 04 00 00 00 05 00 00 00  SUB.
0000:0150 20 41 44 44 03 00 00 00 03 00 00 00 04 00 00 00  ADD.
0000:0160 20 53 54 57 03 00 00 00 1e 00 00 00 ff ff ff ff  STW.
0000:0170 20 42 45 51 00 00 00 00 00 00 00 00 f1 ff ff ff  BEQ.
```

4. Features

4.1. Primitive Datentypen und Komplexe Datentypen

Unser Compiler unterstützt bei der Input-Sprache primitive Datentypen wie `int` und `char`. Unser Compiler unterstützt bei der Input-Sprache auch komplexe Datentypen wie `struct, und array (int, char).`

4.2. Kontrollstrukturen

Unser Compiler unterstützt Kontrollstrukturen wie `if`, `else` und `while`. Dabei können `if`, `else` und `while` Anweisungen beliebig viel verschachtelt werden. Bei der Anweisungen soll man beachten, dass die geschweifte Klammern `{,}` immer angelegt werden müssen. `for` und `do while` Anweisungen werden nicht unterstützt. Außerdem wird die Lazy-Evaluation nicht implementiert.

Beispiel: befindet sich im Ordner: `tests/ifelse.mimimiC`

```
int main()
{
   int a = 4;
   int i = 1;
   int j = 2;
   int x = 0;

if( ((a == i) && (a > 0))||
    ((a == j) && (a > 6))||
    ((a == 12)&& (j != x)))
{
      j= j%4;
      x= 6+1;
}

   if((a< 5) || (a>=5))
   {
     a = a * 6;
   } //end if

   else
   {
     a = 2;

   } //end else
   if (i >= 0)
   {
  if (j > i)
  {
     x = x - j;
  } //end if
  } //end if

  else
  {
    x = x * i;

  } //end else
} //end main
```

Beispiel: befindet sich im Ordner: `tests/while.mimimiC`

```
int main()
{
  int x= 0;
  int y= 19;

    while(x < 11)
    {
      x= 1 + x;
    } //end while
      x= x * 5;

    while (y > 3)
    {
      y= 2 * 2;

       while(y > 2)
       {
         y= y * x;
       } //end while

       y= y + x;

    } //end while
} //end main
```

4.3. Typendefinitionen/Structs

Für Datentypenvereinbarungen mittels `typedef` gibt es zwei Möglichkeiten: Festlegung eines Aliasnamens zu einem existierenden benannten Datentyps `typedef typname aliasname;` und Festlegung eines Namens zu einem zu vereinbarenden aggregierten Datentyp `typedef struct { int re, int im; } complx;` Bei unserem Compiler funktioniert derzeit typedef und struct leider nicht! (Bug)

`Beispiel:befindet sich im Ordner: tests/struct.mimimiC und tests/typedef.mimimiC`

```
struct point
{
   int x;
   int y;

}; //end struct
int main()
{
   point.x= 45;
   point.y= 33;

   if (point.x != point.y)
   {

   } //end if
 } //end main
```

```
typedef struct
{
  int i;

} ident; //end typedef

int main()
{
   ident ip= 10;
    return 0;

} //end main
```

4.4. Arithmetische und Logische Operatoren

Unser Compiler unterstützt arithmetische und logische Operatoren bzw. Ausdrücke auf Integer-Werten. Es können arithmetische Operationen „+, -, *, /, %“ und logische Operationen „==, !=, >, < , >=, <= ||, &&“ und Klammern beliebig nach Infix-Schreibweise kombiniert werden. Dabei um die generierten Instruktionen für arithmetische Ausdrücke zu reduzieren bzw. die Operationen bei der Kompilierungszeit auszuwerten, wird die Konstantenfaltung nicht implementiert.

Beispiel: befindet sich im Ordner: `tests/arithmetic.mimimiC und tests/logical.mimimiC`

```
int main()
{
  int x= 122+ 5;
  int y= x* 7;
  int z= y/ 2;
  int i= z% 30;

//printf("x: %d \ny: %d \nz: %d \ni:
%d\n ", x, y ,z ,i);

} //end main
```

```
int main()
{
  int a = 5;

  if (a > 4 && a < 8)
  {
    //printf("a: 5, 6 or 7n \n");
  } //end if

  else if (a <= 2 || a >= 9)
  {
    //printf("a: 1, 2, 9 or 10n \n");
  } //end else if

  else
  {
    //printf("a: 3, 4 or 8n \n");
  } //end else

    //printf("a: %d \n", a);

} //end main
```

4.5. Fehlerbehandlung

Syntax-Fehler sollten erkannt werden und eine Fehlermeldung zur Folge haben. Bei Fehlern wird die Stelle im Code angegeben und der Kompilierungsvorgang fortgesetzt. Dies ist jedoch nur bei schwachen Symbolen, wie fehlenden Semikolons oder geschlossenen Klammern möglich. Fehlende so genannte „Starken-Symbole“ wie „if“ oder geöffnete Klammern können nicht einfach für den Kompilierungsvorgang korrigiert werden da ein „Kontext-logisches Ergänzen“ bei diesen Symbolen nicht möglich ist. Falls es vom Parser einen syntaktischem Fehler gefunden wird, erfolgt dann eine Ausgabe auf die Konsole mit der entsprechenden Fehlermeldung (Die Zeile- und Token-Nummer, die fehlt, und was erwartet wird). Der Parser versucht weiterhin die restlichen Tokens auf ihre Richtigkeit zu überprüfen.

Beispiel: befindet sich im Ordner: tests/error.mimimiC

```
1 #include <stdio.h>
2
3 int main()
4 {
5 int= 122+ 5;
6 int y= x* 7;
7 int z= y/ 2;
8 int i= z% 30;
9
10 } //end main
```

```
parsing: tests/error.mimimiC
Setting File: tests/error.mimimiC
Token: # (17)
Token: include (16)
Token: < (26)
Token: stdio (1)
Token: . (30)
Token: h (1)
Token: > (25)
include found in line 1

Token: int (2)
Statement: found in line 3
Token: main (1)
type found in line 3
Identifier In The Statement found in line 3
Token: ( (34)
Token: ) (35)
Token: { (38)
CodeUnit: object exist, function found

Token: int (2)
simpletype INT found in line 5
Token: = (42)
Error in line 5 by Token =,42! variable expected!
Error in line 5 by Token =,42! } expected!
module found in line 5
```

4.6. Arrays

Derzeit nur 1-Dimensional

Beispiel: befindet sich im Ordner: `tests/array.mimimiC`

```
int function2();
int function3();

int main()
{
   function2();
   function3();
} //end main

int function2()
{

  int i[4];
      i[0]= 23;
      i[1]= 34;
      i[2]= 65;
      i[3]= 74;
} //end function2
```

```
int function3()
{
     int i[34];

      i[1]= 6;

} //end function3
```

4.7. Pointer

Mit dem Adress-Operator „&“erhält man die Adresse einer Variablen im Speicher. Das wird vor allem verwendet, um Zeiger auf bestimmte Variablen verweisen zu lassen.

Beispiel: befindet sich im Ordner: `tests/pointer.mimimiC`

```
int main()
{
    int x= 4;
    int *y;

    y= &x;
    *y= *y+ 3;

} //end main
```

4.8. Funktionen

Beispiel: befindet sich im Ordner: `tests/functioncall.mimimiC`

`Call-by-Value und Call-by-Reference funktionieren. (mi Pointern)`

```
int add(int x, int y)
{
    int r;
    r= x+ y;
} //end add

int sub(int x, int y)
{
    int r;
    r= x- y;
} //end sub

int div(int x, int y)
{
    int r;
    r= x/ y;
} //end div


int mul(int x, int y)
{
    int r;
    r= x* y;
} //ent mul

int mod(int x, int y)
{
    int r;
    r= x% y;
} //end mod


int callByValue(x1, y1)
```

```
int global = 10;

int function2()
{
    return function1()+ 15;
} //end  function2

int function1()
{
  global = global + 3;

    if(global < 60)
    {
      int b = function2();
      return 50 + b;
    } //end if

      return 50;
} // end  function1




int main()
{
  int result;

  result = add(5, 3);
  result = sub(34, 2);
  result = div(11, 6);
  result = mul(9, 9);
  result= mod(324, 5);
```

```
{
    int temp;
    temp= x1;
    x1= y1;
    y1= temp;

} //end callByValue


int callByReference(int *x2, int *y2)
{
    int temp2;
    temp2= *x2;
    *x2= *y2;
    *y2= temp2;
} // end  callByReference
```

```
  int x1= 50, y1= 70;
  callByValue(x1, y1);

  int x2= 3, y2= 5;
  callByReference(&x2, &y2);

  int a = function1();
  global = global + a;

  return 0;
} //end main
```

Beispiel: befindet sich im Ordner: tests/functionprototype.mimimiC

```
int fac(int n);

int main()
{
   //int y = fac(3);

     return 0;
} //end main

int fac(int n)
{
   if (n == 0)
   {
     return 1;
   } //end if
   else
   {
      return n* fac(n- 1);
   } //end else
} //end fac
```

4.9. File I/O

Wird über die VM realisiert, dazu gibt es die `FOP (open), FCL (close), FRD (read), FWR (write).` Darüber dann Funktionen wie `printf(char *, int length)` oder `printInt(int n).` Diese Funktionen über die `stdincl.h`-Datei benützbar.

4.10. Separate Kompilierung

Parser erzeugt für jede Eingabedatei eine RISC-Datei, die Dateien werden dann durch den Linker in eine einzelne RISC-Datei gelinkt. Dabei liest der Linker alle Dateien ein, berechnet für jede Datei die Code-, Globalerspeichergröße sowie Headervariablen und verwendet diese als Adressoffsets von Sprüngen, globale Variablen, Heap sowie zur Adressberechnung von Funktionsaufrufen, die über eine Eingabedatei hinausgehen oder in der `stdincl.h` stehen. Dabei sind lokale und globale Variablen sowie Heap und Headervariablen Quadbyte-Adresiert und Sprünge Instruktionszeilen-Adressiert (4*4 Byte). Im Gegensatz zu lokalen und globalen Variablen, können Headervariablen keinen Initalisierungswert haben. Kann der Linker nicht alle Abhängigkeiten auflösen, generiert er wieder ein Objekt-File.

5. Heap

5.1. Spezifikationen

Heap ist als Bumppointer implementiert. Dieser liegt immer an der Speicheradresse 3 (Quadbyte adressiert) und wird durch Aufruf von `malloc(int size)` um die mitgegebene Größe erhöht und der Anfangswert zurückgegeben.

6. Linker

```
0000:0000 38 38 38 20 31 37 20 35 33 20 35 37 31 32 32 39 888 17 53 571229
0000:0010 32 31 30 36 38 39 20 31 20 30 20 31 33 34 35 34 210689 1 0 13454
0000:0020 32 38 32 30 31 32 32 39 30 38 33 39 38 37 20 32 28201229083987 2
0000:0030 39 20 32 39 20 31 31 34 36 35 31 34 34 30 39 36 9 29 11465144096
0000:0040 20 31 20 33 30 20 2d 31 31 32 32 39 32 31 30 36  1 30 -112292106
0000:0050 38 39 20 31 20 30 20 31 33 34 35 34 32 38 32 30 89 1 0 134542820
0000:0060 31 32 32 39 30 38 33 39 38 37 20 32 39 20 32 39 1229083987 29 29
0000:0070 20 31 31 34 36 35 31 34 34 30 39 36 20 31 20 33  11465144096 1 3
0000:0080 30 20 2d 32 31 34 36 34 30 39 33 37 32 38 20 31 0 -21464093728 1
0000:0090 20 33 30 20 2d 31 31 32 32 39 32 31 30 36 38 39  30 -11229210689
0000:00a0 20 32 20 30 20 30 31 32 32 39 32 31 30 36 38 39  2 0 01229210689
0000:00b0 20 33 20 30 20 31 33 34 35 34 32 38 32 30 31 33  3 0 13454282013
0000:00c0 34 37 32 34 30 37 33 36 20 32 20 32 20 33 31 34 47240736 2 2 314
0000:00d0 31 34 32 38 33 38 30 38 20 32 20 30 20 33 31 32 14283808 2 0 312
0000:00e0 32 39 32 31 30 36 38 39 20 32 20 30 20 30 31 33 29210689 2 0 013
0000:00f0 36 33 34 39 33 34 30 38 20 32 20 30 20 32 31 32 63493408 2 0 212
0000:0100 32 39 32 31 30 36 38 39 20 32 20 30 20 31 31 33 29210689 2 0 113
0000:0110 36 33 34 39 33 34 30 38 20 32 20 30 20 38 31 32 63493408 2 0 812
0000:0120 32 39 32 31 30 36 38 39 20 33 20 30 20 31 33 34 29210689 3 0 134
0000:0130 35 34 32 38 32 30 31 34 36 34 30 39 33 37 32 38 5428201464093728
0000:0140 20 34 20 33 30 20 2d 31 31 32 32 39 32 31 30 36  4 30 -112292106
0000:0150 38 39 20 35 20 30 20 31 33 34 35 34 32 38 32 30 89 5 0 134542820
0000:0160 31 31 31 32 38 38 38 30 39 36 20 34 20 34 20 35 1112888096 4 4 5
0000:0170 31 31 34 35 33 32 33 38 30 38 20 33 20 33 20 34 1145323808 3 3 4
```

6.1. Spezifikationen

6.2. Object File

Im Objekt-File sind neben den RISC-Code noch Annotationen . Es steht vor jedem Funktionsbeginn ein Marker mit dem Funktionsnamen, für Headervariablen gibt es die speziellen Linker-Befehle `LDWH` und `STWH`, die aus dem Zielregister und den Variablennamen bestehen. Für Befehle die gefixt werden müssen, wird das erste Quellregister auf 99 gesetzt und der Intermediate wird dann vom Linker angepasst.

7. Virtuelle Maschine

7.1. Spezifikationen

Befehl	Nr	Bedeutung	Befehl	Nr	Bedeutung

ADD	1	Addition
		`R[a]=R[b]+R[c];`
			PSH	15	Push On Stack
					`R[b]=R[b]-c;`
					`int col=(R[b])%4;`
					`memory[R[b]/4][col]=R[a];`

SUB	2	Subtraction `R[a]=R[b]-R[c];`	BEQ	16	Branch If Equal `if(R[a]!=0){nxt=PC+c;}`
MUL	3	Multiplication `R[a]=R[b]*R[c];`	BNE	17	Branch Not Equal `if(R[a]!=0){nxt=PC+c;}`
DIV	4	Division `if(R[c]!=0){R[a]=R[b]/R[c];}`	BLT	18	Branch If Less Than `if(R[a]<0){nxt=PC+c;}`
MOD	5	Modulo `R[a]=R[b]%R[c];`	BGE	19	Branch If Greater Than Or Equal `if(R[a]<=0){nxt=PC+c;}`

CMP	6	Compare
		`R[a]=R[b]-R[c];`
			BLE	20	Branch If Less Than Or Equal
					`if(R[a]==0||R[a]<=0)`
					`{nxt=PC+c}`

OR	7	Logical Or `R[a]=(R[b]||R[c]);`	BGT	21	Branch If Greater Than `if(R[a]>0){nxt=PC+c;}`
AND	8	Logical And `R[a]=(R[b]&&R[c]);`	JSR	22	Jump To Subroutine `R[31]=PC+1; nxt= c;`
ADDI	9	Immediate Addition `R[a]=R[b]+c;`	RET	23	Return `nxt=R[c];`
SUBI	10	Immediate Subtraction `R[a]=R[b]-c;`	FOP	24	File Open
ANDI	11	Immediate And `R[a]=(R[b]&&c);`	FCL	25	File Close

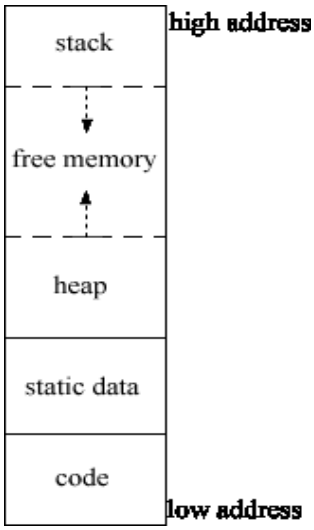
LDW	12	Load Word
		`int col=((R[b]+c))%4;`
		`R[a= memory[(R[b]+c)/4][col];`
			FRD	26	File Read

POP	13	Pop On Stack `int col=(R[b])%4;`	FWR	27	File Write

		`R[a= memory[R[b]/4][col];` `R[b=R[b]+c;`			

STW	14	Store Word
		`int col=((R[b]+c))%4;`
		`memory[(R[b]+c)/4][col]=R[a];`			

7.2. Speicher Management



Literaturverzeichnis

[1] “Grundlagen und Techniken des Compilerbaus”, Niklaus Wirth, 1. Auflage, Addison-Wesley (1996).