

Programmieren für Übungsaufgaben in MMIX

zur Informatik II – 2003 (Prof. Loos)

Holger Gast, WSI, Universität Tübingen
email: gast@informatik.uni-tuebingen.de

27. Juni 2003

Es ist viel darüber geschrieben worden, wie man große Programme in immer kleinere Unteraufgaben aufteilt, bis man schließlich eine Größe erreicht, in der man den Programmcode aufschreiben kann (siehe auch Faszikel, S. 63). Wahrscheinlich um die Leser nicht abzuschrecken lassen die Autoren jedoch die „unschönen“ Seiten des Programmierens zumeist beiseite und geben nur generelle Anweisungen. Aufgrund einiger direkter Anfragen Ihrer Kommilitonen und Rückmeldungen aus der Tutorenbesprechung unternehme ich hier also den Versuch, diese Lücke zu stopfen: Ich lasse Sie detailliert an der Erstellung einer Musterlösung teilhaben. Die Textform ist dabei vielleicht etwas freier, und ich muß Ihnen mehr Zwischenergebnisse und Fehlversuche eingestehen, als das sonst in der Literatur zu geschehen pflegt; ich habe aber das Gefühl, daß für Sie gerade das interessant sein kann: Sie analysieren beim Lesen nicht das fertige Produkt, sondern den Produktionsprozess.

Bevor ich Sie mit dem weiteren Text allein lasse, möchte ich Sie ermuntern, mir Mail zu schreiben, um Anregungen, Kritik, Hinweise auf Ungenauigkeiten und Auslassungen oder was auch immer weiterzugeben. Außerdem weise ich Sie darauf hin, daß Sie den Text bitte *nicht* weitergeben sollen; dafür habe ich mich entschieden, damit nachfolgende Leser immer eine aktuelle Version erhalten und ich jederzeit Änderungen vornehmen und verteilen kann. Eine aktuelle Version werde ich unter <http://www-pu.informatik.uni-tuebingen.de/users/gast/docs/progmix.pdf> bereithalten.

1 Fehleinschätzungen

Es gibt einige Einschätzungen, die ziemlich sicher dazu führen, daß man eine Übungsaufgabe *nicht* lösen wird; mit diesen sollte man zuallererst einmal aufräumen.

1. Aufgaben sind entweder verständlich oder unverständlich.
2. Aufgaben müssen so sein, daß man sie sofort versteht und lösen kann.
3. Aufgaben wiederholen den Stoff der Vorlesung und üben ihn ein.
4. Programme schreibt man von vorn nach hinten einfach auf.
5. Programme laufen beim ersten Versuch fehlerfrei ab.
6. Programmieren macht immer Spaß.
7. Ich bin der einzige, dem die Lösung nicht einfällt.
8. Ich muß die Aufgabe ganz allein lösen.

Wenn Ihnen klar ist, daß diese Aussagen alle falsch sind, ärgern Sie sich bitte nicht, vielleicht hilft der Abschnitt ja jemand anderem. Leider ist hier kein Platz, alle diese Verneinungen wirklich zu diskutieren, aber ich hoffe, daß der Rest des Textes wenigstens eine implizite Rechtfertigung enthält.

2 Grundzyklus der Problemlösung

Ich kann Ihnen bestimmt kein Rezept geben, mit dem Sie sicher alle Programmieraufgaben lösen können; gerade beim Programmieren hilft einfach nichts als Üben und Erfahrung. Aber vielleicht hilft Ihnen die Leitidee eines „Grundzyklus“,¹ die der Beispielaufgabe im Rest des Textes zugrunde liegt. Sie formuliert eine konstruktive Variante der „Fehleinschätzungen“, indem sie jede einzelne vermeidet:

Verstehen – Planen – Ausführen – Testen

Zu lesen ist diese Abfolge eigentlich rückwärts: Man kann schlecht testen, bevor man nicht verstanden hat, was das erwartete Ergebnis ist, man sollte nichts anfangen zu tun, bevor man nicht einen Plan gemacht hat, wo man hin will. Die Schritte des „Grundzyklus“ sind zwar offensichtlich notwendig und Sie fragen, warum ich sie extra aufschreibe: Gerade wenn man nicht weiter kommt, helfen sie vielleicht, die Orientierungsrichtung „vorwärts“ wiederzufinden.

Der Grundzyklus muß zur Aufgabenlösung mehrfach wiederholt werden: Beim Lesen der Aufgabe, beim Erstellen des Pseudo-Codes, beim Erstellen des MMIX Programms und bei der Fehlerbeseitigung. Dabei nimmt von mal zu mal der Detaillierungsgrad, in dem Sie die Lösung formulieren zu. Leider ist es unvermeidlich, daß der Fehlerteufel gerade im Detail steckt: Oft werden Sie erst beim Debuggen feststellen, daß Ihr Pseudo-Code Fehler hatte – lassen Sie sich dadurch nicht aus der Ruhe bringen.

3 Die Beispielaufgabe

Als durchgehendes Beispiel möchte ich Aufgabe 18, die Klammerpaare, verwenden: Der Algorithmus ist nicht ganz offensichtlich, aber klein genug, um ihn hier zu erläutern; außerdem kann ich mich noch an meine Programmierfehler erinnern, so daß ich für das Debugging keine neuen einzubauen brauche. Schließlich haben Sie die Möglichkeit, die publizierte Musterlösung mit der folgenden Schilderung, wie ich dazu kam, zu vergleichen. Vor allem lohnt sich ein Vergleich mit Abschnitt 5, Teil **Ausführen** – dort steht in Kurzform

¹Die Parallelität mit Fetch, Decode, Execute, Exception hatte ich nicht intendiert.

die Musterlösung, den gesamten Rest müssen Sie beim Lesen selbst denken.

Zur Erinnerung der Aufgabentext:

Gegeben ist eine String, in dem unter anderem Klammern ' (' und ') ' vorkommen. Überprüfen Sie, ob der String ein wohlgeklammerter Ausdruck ist. Ein String w ist wohlgeklammert, wenn er eine der folgenden Bedingungen erfüllt:

1. w leer ist oder ein einzelnes Zeichen außer den Klammern.
2. w ist die Konkatenation $w = w_1w_2$ zweier wohlgeklammerter Strings w_1, w_2 .
3. w hat die Form $w = (w_1)$, wobei w_1 ein wohlgeklammerter Ausdruck ist.

Hinweis: Durchlaufen Sie den String von vorn bis zur abschließenden 0 und zählen Sie mit, wieviele öffnende Klammern Sie gesehen haben. Für jede schließende Klammer erniedrigen Sie den Zähler. Alle Zeichen außer den Klammern dürfen Sie ignorieren. Den Eingabestring s können Sie in einer Konstanten speichern oder gemäß Aufgabe 17 von der Kommandozeile lesen.

1. [1] Listen Sie alle möglichen Fehlersituationen in obigen Algorithmus auf, die anzeigen, daß der String nicht wohlgeklammert ist. Erfinden Sie Fehlermeldungen dazu.
2. [1] Schreiben Sie den Algorithmus in Pseudo-Code auf.
3. [4] Programmieren Sie den Algorithmus in MMIX. Geben Sie in Fehlersituationen die Meldungen aus 1. aus.

Hinweis: Die ersten Teilaufgaben können Sie auch in den Kommentaren zu Ihrem Programm abgeben.

Bei der Bearbeitung der Aufgabe werde ich, anders als in den Musterlösungen, den Grundzyklus *mehrfach* durchlaufen. Aus Platzgründen, und damit ein klareres Bild von der Lösung entsteht, schreibe ich jedoch für jeden Schritt die Durchläufe direkt hintereinander, getrennt durch fettgedruckte Ziffern. Wenn Sie also im Text sehen

Verstehen: 1: V_1 2: V_2
Planen: 1: P_1 2: P_2
Ausführen: 1: A_1 2: A_2
Testen: 1: T_1 2: T_2

dann habe ich eigentlich die Textstücke in der Reihenfolge $V_1, P_1, A_1, T_1, V_2, P_2, A_2, T_2$ verfaßt; das gilt auf für die Programmstücke, die *nicht* aus einem lauffähigen Programm herausgeschnitten wurden. Ich möchte Sie ermuntern, den Text *zweimal* zu lesen: Einmal in der gerade angegebenen Reihenfolge, d.h. indem Sie zwischen den fetten „Marken“ springen, das zweite Mal, indem Sie die Marken einfach überlesen. Der erste Durchlauf entspricht dem, wie Sie auf die Lösung kommen, der zweite entspricht ihrer Abgabe.

4 Aufgabe verstehen und Pseudocode

Zum Verständnis möchte ich eigentlich nur insofern etwas sagen, als es das Erstellen des Programms direkt betrifft; für den Umgang mit dem Aufgabentext selbst können Sie jederzeit Ihren Tutor ansprechen. Die Bezeichnungen unseres

Grundzyklus müssen wir schon etwas dehnen, um hier noch hinzukommen.

Der Pseudo-Code hilft Ihnen beim Verstehen, indem Sie ihn knapp hinschreiben können, ohne auf syntaktische Korrektheit zu achten: Es ist nur wichtig, daß *Sie* (und ihr Tutor) verstehen, was gemeint ist. Benutzen Sie ihn, um sich selbst klar zu machen, wie Sie die Eingaben verarbeiten wollen, um das gewünschte Ergebnis zu erhalten. Im Grunde ist dieser Schritt der schwierigste: Sie müssen mit den Schablonen für Schleifen und Verzweigungen, die Sie aus der Vorlesung kennen, so ein Puzzle konstruieren, daß am Ende das Richtige herauskommt. Deshalb teilen Sie am besten diesen Schritt in zwei Teilschritte ein: Zunächst erklären Sie (sich selbst oder einem Kommilitonen) die Lösung in eigenen Worten, sehr grob; wenn es Ihnen Schwierigkeiten macht, die Erklärung zu behalten, schreiben Sie sie auf. Wenn das geschafft ist, sind Sie schon über den Berg: Jetzt formulieren Sie Ihre Sätze so um, daß nur noch „solange . . . , mache . . .“ und „für den Fall, daß . . . , mach . . .“ vorkommen.

Verstehen Zunächst einmal sollte man sich Zeit nehmen, den Text gründlich zu lesen; wenn Worte oder Ideen vorkommen, die man nicht kennt, ist das nicht so schlimm: Dafür gibt es Tutoren und Assistenten. *Ziel:* Machen Sie sich klar, *was* Sie anfertigen wollen. Was sind die Daten, auf denen Sie operieren, welche Form hat Ihre Abgabe?

1: Es sind die drei am Ende aufgezählten Dinge verlangt, zwei davon in (ziemlich) freiem Text, eines ist ein MMIX-Programm. Sie alle beziehen sich auf Strings (d.h. Zeichen/Bytes, die hintereinander im Speicher stehen und von einer 0 abgeschlossen werden). Die Besonderheit der Eingaben sind die Klammern, alle anderen Zeichen interessieren nicht. Glücklicherweise ist der zu schreibende Algorithmus schon als Text formuliert (*Hinweis*). Was aber bedeuten die drei Bedingungen an die Strings? Dem Text nach drücken sie aus, daß ein String „wohlgeklammert“ ist; außerdem soll angeblich der *Hinweis* eine Algorithmen-Form dieses Begriffs enthalten.

2: Ich beschließe, aber sagen Sie das nicht weiter, die kryptische 1-2-3 Erklärung einfach zu ignorieren. Vielleicht hat da jemand sehr genau sein wollen und den Text unverständlich gemacht.² Es fehlen mir also noch Pseudo-Code und MMIX-Programm.

3: Ok, die Fehlerbedingungen habe ich jetzt auch verstanden (s.u.) und schreibe sie in einen Kommentar in meine MMIX-Quelle. Außerdem sehe ich den Fehler bei ') (' : Wenn ich c verkleinere und dann wieder vergrößere, kann es sein, daß mir ein Fehler durch die Lappen geht.

Planen Machen Sie sich klar, *wie* Sie es anfertigen wollen.

1: Ich schreibe ein **Main** Programm, das auf irgendeine Art den *Hinweis* umsetzt. Es arbeitet auf einer Zeichenkette s , ich werde also auch einen Index i in die Zeichenkette benötigen. Ohne nachzudenken lohnt es immer, auch gleich `t IS $255` zu deklarieren (Standardkonvention im Faszikel).

2: Ich werde einfach den Hinweis ausprogrammieren; das mit den Fehlerfällen habe ich nicht verstanden. Vielleicht fange ich mal mit dem Pseudo-Code an.

3: Ich werde beim Herunterzählen noch einen Test einbauen müssen.

²„Like all people who try to exhaust a subject, he exhausted his listeners.“ (O. Wilde)

Ausführen Schreiben Sie alles auf, was Sie schon haben – löschen können Sie immer noch. Notieren Sie z.B. ruhig schon einmal das Schema für Unterprogramme (PREFIX, das Einsprunglabel, die IS-Deklarationen für die Eingaben). Das gibt ein beruhigendes Gefühl: Jetzt braucht man nur noch Löcher auszufüllen. Ganz wichtig an dieser Stelle: Erfinden Sie im ersten Durchlauf Testeingaben und Testaufrufe!

1: `s` muß irgendein 'wohlgeklammerter' String sein – aber das hatte ich ja nicht verstanden. Beispiele, die man dann auch als Tests verwenden kann, helfen. Da die Aufgabe sagt, ich kann alle nicht-Klammern gleich ignorieren, schreibe ich sie erst gar nicht hin. Erst mal benutze ich die Intuition: Klammern müssen immer paarweise auftreten. Ich würde folgende durchgehen lassen: `()`, `((()()))`, `((()))`, aber diese ablehnen: `(,)`, `((,))`, `((()))()`.

Ach ja, und dann hatte ich auch schon verstanden, wie mein MMIX-Programm am Ende aussehen soll:

```
"par0.mms" 3a ≡
                LOC #100
s                IS $0
i                IS $1
t                IS $255
<weitere Register 4b, ... >
Main            SWYM 0
<meine Loesung 4a>
                TRAP 0,Halt,0
<zum Ende 5j>◇
```

2: Also gut, Hinweis in Pseudo-Code. Nehmen Sie einen Zähler: `c=0`. Ok, hab ich. Laufen Sie durch den String; wie geht das? Index verwenden! Gut, los:

```
<Schleife PC 3b> ≡
    for i=0 to n (wobei s[n]=0) ...◇
Macro never referenced.
```

Nein, so geht's nicht, das `wobei` ist nicht erlaubt

```
<Schleife 2 3c> ≡
    i=0;
    while (s[i]!=0) {
        <Rumpf Pseudocode 3e>
        ++i
    }◇
```

Macro never referenced.

Ok, das sieht schon besser aus – nur noch erlaubte Operationen. (*Zwischengedanke*: Das kann ich auch gleich in MMIX hinschreiben – ach nee, nachher muß ich noch was ändern und dann muß ich's an zwei Stellen tun). Also weiter: Zähler erhöhen und erniedrigen, je nachdem, auf welche Klammer ich stoße. Je-nachdem-Fallunterscheidung! –If!

```
<Rumpf Pseudocode 1 3d> ≡
    if (s[i]='(') then c=c+1
    if (s[i]=')') then c=c-1◇
```

Macro never referenced.

Hmm, das war einfacher als ich dachte.

3: Wegen des entdeckten Fehlers `()(` wird akzeptiert!) ersetzen wir das durch:

```
<Rumpf Pseudocode 3e> ≡
    if (s[i]='(') then c=c+1
    if (s[i]=')') then
        c=c-1
    if c<0 then error "syntax error"◇
```

Macro referenced in 3c.

Testen Ziel: „Wenn ich das so abgebe, habe ich dann die Aufgabe gelöst?“ Die Antwort ist nicht automatisch „ja“. Nachdem Sie den Plan haben, können Sie darauf schauen, ob er auch den *Details* der Aufgabenstellung gerecht wird. Erfüllen die Testeingaben alle Anforderungen? Haben Sie Spezialfälle ausgetüzt, die in der Aufgabe nicht so erwähnt sind?

1: Stimmt meine Intuition von 'wohlgeklammert' überhaupt mit dem Begriff der Aufgabe überein? Die 1,2,3-Beschreibung verstehe ich nicht, vielleicht komme ich mit dem *Hinweis* durch? Mit einem Blatt Papier schreibe ich über jede Klammer in den Beispielen die Zählerstände.³ Prima, der *Hinweis* scheint zu funktionieren: Er stimmt mit meiner Einschätzung, was gute und schlechte Strings sind überein.

2: Mein Pseudocode ist erst mal fertig. Mal sehen, ob er funktioniert. Ich stecke einen Beispielstring hinein und mach mir eine Tabelle mit den `c, i` bei jedem Schleifendurchlauf. Kann ich nachher gut zum Debuggen benutzen.

Dabei fällt mir auf: Immer wenn ich einen fehlerhaften String bearbeite, wird mein Zähler irgendwann `< 0`, oder aber am Ende bleibt der Zähler `> 0` stehen. Das kann nicht gut sein. Fehler! Mein Pseudocode läuft einfach weiter, selbst wenn irgendwann der Zähler mal unter 0 ging! Beispiel: `) (` ist anscheinend wohlgeklammert!

3: Ok, jetzt wird `) (` zurückgewiesen.

5 MMIX-Programm

In den Musterlösungen finden Sie oft Bemerkungen wie „Diesen Pseudocode können wir direkt in MMIX umsetzen“. Das geschieht nicht aus Faulheit, sondern weil man die Konstrukte des Pseudo-Codes direkt mit den Schablonen der Vorlesung in MMIX ausdrücken kann. In diesem Abschnitt gilt also: Denken Sie *nicht* über die Lösung nach! Wenn Ihr Pseudo-Code korrekt war und Sie die Schablonen genau befolgen, haben Sie ziemlich gute Chancen, dass auch Ihr MMIX-Programm im großen und ganzen korrekt ist.

Verstehen Lesen Sie Ihren Pseudo-Code nochmals durch, aber diesmal nicht unter dem Gesichtspunkt, was er tut, sondern welche Konstrukte (Schleifen, Fallunterscheidungen, Variablen) er enthält. Machen Sie sich klar, wie diese untereinander in Verbindung stehen: Welche Konstrukte folgen aufeinander, welche sind ineinander geschachtelt? Ziel: Sie haben die *Struktur* Ihres Pseudo-Codes verstanden.

1: Ich verwende drei Variablen `s, i, c` und eine ist die Eingabe; `s` ist eine Adresse, `i, c` kleine ganze Zahlen. Ich werde für jede ein Register reservieren. Unterprogramme muß ich keine aufrufen und ich bin auch nicht in einem Unterprogramm, sonst müßte ich noch an `rJ` und die Register für den Funktionsaufruf denken. Alle meine arithmetischen Operationen und Vergleiche gibt es direkt in MMIX.

2: Ich habe eine große Schleife, in der zwei Fallunterscheidungen geschachtelt sind. Eine Detailfrage ist, wo ich eigentlich den String `s` hernehme.

³Ich erinnere mich noch an die Osterferien, in denen ich meinen ersten Parser programmierte; eine Teilaufgabe sah ich darin, die tiefste Klammerebene zu finden und begann damit, daß ich eine Formel hinschrieb; man kommt ziemlich schnell darauf, daß man zählt, wieviele Klammern 'noch offen' sind – bald war die Formel mit lauter kleinen bunten Zahlen versehen.

3: Ich sollte das `s[i]` nicht in einem temporären Register haben, ich brauche es ja doch gleich wieder!

4: Jetzt mal zu den zwei Fallunterscheidungen. Sie haben einen gemeinsamen Aufbau. Wohin lege ich aber die Konstanten '(' und ')'?

5: Jetzt fehlen mir nur noch die beiden Aktionen, eine davon hat wieder eine Fallunterscheidung.

6: Die Testeingabe muß in den Speicher kommen und dessen Anfangsadresse muß in Register `s` zu liegen kommen. Ausserdem muss nach dem Verlassen der Schleife das Ergebnis ausgegeben werden. Wie erkenne ich nach dem Verlassen der Schleife, ob der String korrekt ist? Die durchgerechneten Beispiele zeigen mir, daß der String wohlgeklammert ist, wenn am Ende der Schleife `c = 0` ist.

7: Das Programm ist fertig. Jetzt müssen wir es durch `mmixal` bringen. Zu verstehen sind dabei vor allem die Fehlermeldungen, auf die ich hier aber nicht eingehe.

Planen Überlegen Sie sich, wie sie die *einzelnen* Konstrukte sowie ihre *Beziehungen* in MMIX nachbilden. Benutzen Sie dabei soweit als möglich Schablonen der Vorlesung. Ein Tipp: Markieren Sie die Stellen mit Kommentaren, bei denen Sie aus irgendeinem Grund aufgehört haben zu programmieren. Schreiben Sie auf, was an diesen Stellen noch fehlt. Ich mache das in diesem Text mit `nuweb`, einem Literate-Programming Tool. Verstehen Sie also in meinem Code die spitzigen Klammern als Kommentare, was ich hier später noch einfügen will.

1: Erst mal schreib ich den Rahmen hin; ich weiss noch nicht, wie ich meine Testeingabe herkrige, habe aber zwei Ideen: 1) Ich lege mir ein `BYTE` Feld an und hole die Adresse. 2) Ich lese das erste Kommandozeilenargument (Aufgabe 17!). Woher kriege ich das Ergebnis? Ich denke, ich gebe es lieber aus.

2: Gut, jetzt also an die äußere Schleife. Dazu brauche ich eine Marke, einen Test und einen Rücksprung. Ausserdem muß ich den Ausdruck im Test herunterbrechen.

3: Ich hole mir `s[i]` in ein neues Register und lösche die unsinnige `CMP` Instruktion.

4: Hmm, das ist gut! Die Konstanten '(' und ')' sind nur ein Byte groß, sie passen also in den *immediate*-Teil der `CMP` Befehle, die ich ohne hin zum Vergleichen brauche. Außerdem mache ich es wohl so, daß ich den jeweils nächsten Befehl nach der Fallunterscheidung mit einem `OH` bezeichne: Falls der Test schief geht, kann ich dorthin verzweigen und einfach die Änderungen an `c` überspringen.

5: Das Ende des inneren `if` ist das Ende der umgebenden `if` Anweisung, also das `OH` bei der `INCL i,1`. Die `error`-Anweisung realisieren wir durch eine Meldung und Programmabbruch.

6: Als Testeingabe benutze ich das erste Kommandozeilenargument, als Ausgabe 'ok' oder die Fehlermeldungen. Der Test auf `c = 0` ist einfach.

7: Ich lasse `mmixal` über das Programm laufen. Wenn Fehler darin sind, korrigiere ich sie im Quelltext. Das kann durchaus schon mal eine halbe Stunde dauern, wenn man wirklich viel Code auf einmal geschrieben hat.

Ausführen Schreiben Sie auf, was Sie neu verstanden haben. Fangen Sie unbedingt an, Code zu schreiben, *ohne* gleich den Programmtext von vorne nach hinten zu entwerfen – schließlich haben Sie einen Rechner und es macht keine Probleme, nachher Zeilen einzufügen und zu löschen.

1: Mein MMIX-Programm muß irgendwie so aussehen.

```
<meine Loesung 4a> ≡
  <Testeingabe für s erzeugen 5i>
  <die Klammerpruefung 4c>
  <Ergebnis ausgeben 4e, ... >
    TRAP 0, Halt, 0◇
```

Macro referenced in 3a.

```
<weitere Register 4b> ≡
  c      IS $2
  ◇
```

Macro defined by 4bf, 5h.
Macro referenced in 3a.

2: Als Marke bietet es sich an, die Bedeutung der Schleife zu erfassen, oder aber eine lokale Marke zu verwenden. Ich verwende einmal `loop_c`.

```
<die Klammerpruefung 4c> ≡
  SET i, 0
  loop_c SWYM 0
  <Test der Schleife 4g>
  <Rumpf der Schleife 4h>
    JMP loop_c◇
```

Macro referenced in 4a.

Der Test lautet im Pseudo-Code `s[i] != 0`. Diesen Ausdruck muß ich wohl zerlegen. Ich versuche erst mal, so hinzukommen

```
<Versuchs-Test 1 4d> ≡
  LDBU t, s, i
  CMPT t, t, 0
  BZ t, end_loop◇
```

Macro never referenced.

Die Marke `end_loop` steht beim Ergebnis

```
<Ergebnis ausgeben 4e> ≡
  end_loop <Ergebnis ausgeben 2 ?>◇
```

Macro defined by 4e, 5f.
Macro referenced in 4a.

3: Erst mal brauche ich ein Register

```
<weitere Register 4f> ≡
  si     IS $4
  ◇
```

Macro defined by 4bf, 5h.
Macro referenced in 3a.

Damit ist der Testausdruck sehr einfach geworden.

```
<Test der Schleife 4g> ≡
```

```
  LDBU si, s, i
  BZ si, end_loop◇
```

Macro referenced in 4c.

4: Der Rumpf der Schleife sieht dann so aus

```
<Rumpf der Schleife 4h> ≡
  OH      <Test '(' 5a>
          <c++ 5c>
  OH      <Test ')' 5b>
          <c- 5d>
  OH      INCL i, 1◇
```

Macro referenced in 4c.

Die beiden Tests sind ziemlich symmetrisch

```
<Test '(' 5a) ≡  
    CMP t,si,'(''  
    BNZ t,0F◇
```

Macro referenced in 4h.

```
<Test ')' 5b) ≡  
    CMP t,si,')'  
    BNZ t,0F◇
```

Macro referenced in 4h.

5: Erst mal das Einfache:

```
<c++ 5c) ≡  
    INCL c,1◇
```

Macro referenced in 4h.

Jetzt etwas komplizierter

```
<c- 5d) ≡  
    SUBU c,c,1  
    BNN c,0F  
    GETA t,err1  
    TRAP 0,Fputs,StdErr  
    TRAP 0,Halt,0 ◇
```

Macro referenced in 4h.

Und hier ist noch die Konstante für die Fehlermeldung: ⁴

```
<Fehlermeldungen 5e) ≡  
    LOC (@+3)&-4  
    err1 BYTE "unerwartete schliessende Klammer",#a,0  
    ◇
```

Macro defined by 5eg.
Macro referenced in 5j.

6: Zum Schluß gebe ich noch das Ergebnis aus wie geplant.

```
<Ergebnis ausgeben 5f) ≡  
    BZ c,ok  
    GETA t,err2  
    TRAP 0,Fputs,StdErr  
    TRAP 0,Halt,0  
    ok GETA t,okstr  
    TRAP 0,Fputs,StdErr  
    TRAP 0,Halt,0◇
```

Macro defined by 4e, 5f.
Macro referenced in 4a.

```
<Fehlermeldungen 5g) ≡  
    LOC (@+3)&-4  
    okstr BYTE "ok",#a,0  
    LOC (@+3)&-4  
    err2 BYTE "Klammern offen am Ende",#a,0  
    ◇
```

Macro defined by 5eg.
Macro referenced in 5j.

Die Testeingabe hole ich über die Kommandozeile:

```
<weitere Register 5h) ≡  
    argv IS $1◇
```

Macro defined by 4bf, 5h.
Macro referenced in 3a.

⁴An dieser Stelle pfusche ich etwas: Eigentlich habe ich erst beim Assemblieren gemerkt, daß die Adresse kein Vielfaches von 4 ist und ich das LOC brauchte.

```
<Testeingabe für s erzeugen 5i) ≡  
    LDOU s,argv◇
```

Macro referenced in 4a.

Alle Fehlermeldungen zusammen schreibe ich ganz an das Ende des Programms:

```
<zum Ende 5j) ≡  
    <Fehlermeldungen 5e, ... >  
    ◇
```

Macro referenced in 3a.

7: `mmixal` bedienen – ok.

Testen Überprüfen Sie, ob Ihr MMIX-Code noch mit dem Pseudo-Code übereinstimmt. Verschieben Sie dies nicht erst ans Ende, wenn alles fertig ist, denn im Moment wissen Sie noch, *warum* Sie das MMIX-Programm so geschrieben haben und welche Entscheidungen notwendig waren. Nur so können Sie diese auch gleich mit prüfen.

1: Der Grobentwurf Eingabe-Verarbeitung-Ausgabe sieht brauchbar aus. Ich habe auch daran gedacht, das Programm ab #100 abzulegen.

2: Ich habe offenbar eine Schleife geschrieben. Der Test stimmt offensichtlich; aber die zweite Anweisung ist unnützlich: `t` ist nachher 0, genau dann wenn es vorher 0 ist.

3: Jetzt ist der Code kurz und ich habe auch schon `s[i]` für später in einem Register. Prima.

4: Ok, ich habe die beiden Tests hingeschrieben, sie scheinen das Richtige zu tun: Wenn `si = '('`, dann wird das erste `CMP` ein 0 liefern und wir springen *nicht* über `c++`. Für den zweiten Test gilt das Gleiche.

5: sieht gut aus: Wenn wir eine öffnende Klammer haben, führen wir `c=c+1` aus, bei einer schließenden `c=c-1` und stoppen mit einem Fehler falls dann `c < 0`.

Beim Durchlesen fehlen noch die Erzeugung der Testeingaben und die Ausgabe des Ergebnisses.

6: Das Ergebnis sollte ich jetzt korrekt ausgeben können; ein Beispiel durchschauen (Tabelle der `c, i` von vorher benutzen!). Achtung: Ich hatte ein neues Register `argv` eingeführt und mit dem `i` überlagert; entweder ich ändere die Zuordnung von `i` (`argv` liegt fest), oder überzeuge mich, daß die beiden Register nicht gleichzeitig benutzt werden.

7: Ok, Assemblierung war erfolgreich.

6 Debugging

Jedes Programm enthält nach dem ersten Schreiben noch Fehler und unterschiedliche Programmierer machen unterschiedliche Fehler. Dementsprechend muß auch jeder seine Strategie finden, wie er die Fehler wieder los wird. Manche lesen lieber ihren Code 20 Mal, manche beweisen, daß er korrekt ist (das ist selten), die meisten lassen ihn einmal laufen und schauen, ob das Ergebnis so aussieht wie erwartet.

Verstehen Die Sichtweise auf ihr Programm ändert sich erneut: Sie haben im Hinterkopf, was Sie erreichen wollen (Aufgabe verstanden) und wie sie es erreichen wollen (Pseudo-Code). Nun schauen Sie nach, was Ihr MMIX-Code *wirklich* tut, ohne dabei gleich vorauszusetzen, was er tun sollte. Dazu benötigen Sie ein Werkzeug, mit dem Sie schrittweise beobachten können, wie die Instruktionen ausgeführt werden: `mmix -i`.

1: Ich lese nochmals mein gesamtes Programm und stelle befriedigt fest, daß ich noch bei jeder Instruktion weiß, warum sie dort ist und was ich erwarte, daß sie tut.

```
"par1.mms" 6 ≡
                                LOC #100
s      IS $0
i      IS $1
t      IS $255
c      IS $2
si     IS $4
argv   IS $1
Main   SWYM 0
      LDOU s,argv
      SET i,0
loop_c SWYM 0
      LDBU si,s,i
      BZ  si,end_loop
OH     CMP t,si,'('
      BNZ t,OF
      INCL c,1
OH     CMP t,si,')'
      BNZ t,OF
      SUBU c,c,1
      BNN c,OF
      GETA t,err1
      TRAP 0,Fputs,StdErr
      TRAP 0,Halt,0
OH     INCL i,1
      JMP loop_c
end_loop BZ c,ok
      GETA t,err2
      TRAP 0,Fputs,StdErr
      TRAP 0,Halt,0
ok     GETA t,okstr
      TRAP 0,Fputs,StdErr
      TRAP 0,Halt,0
      TRAP 0,Halt,0
      TRAP 0,Halt,0
      LOC (@+3)&-4
err1   BYTE "unerwartete schliessende Klammer",#a,0
      LOC (@+3)&-4
okstr  BYTE "ok",#a,0
      LOC (@+3)&-4
err2   BYTE "Klammern offen am Ende",#a,0◊
```

- 2: Ich verstehe nichts mehr: Ich bekommen immer ok.
- 3: Offensichtlich ist die Initialisierung nicht ganz richtig: Ich will nicht das erste, sondern das zweite `argv` laden.
- 4: Sollte mein Programm korrekt sein?
- 5: Welche Spezialfälle kann es geben? Die Eingabe kann leer sein!
- 6: Das Problem besteht also in einem 0-Zeiger zu Beginn. Diesen sollten wir explizit abfangen, der Rest des Codes ist nicht für 0-Zeiger gemacht.

Planen Planung bezieht sich hier auf zwei Hauptaufgaben: Das Auffinden und Beseitigen von Fehlern. Für beide gibt es Strategien, die wichtigste Grundregel aber ist, daß sie zuerst etwas einsehen sollten, bevor Sie in irgendeiner Form aktiv werden. Es nützt nichts, auf gut Glück Befehle zu ändern, besser ist es zu testen, bis man den ersten Befehl gefunden hat, der *nachweislich* ein anderes als das erwartete Ergebnis produziert, und genau *diesen und keinen anderen gleichzeitig* abzuändern. Überlegen Sie bei jedem Plan, ob er wahrscheinlich zum Erfolg führt, d.h. wie er das Ergebnis der falschen

Instruktion zum besseren verändern wird (ein Grundzyklus im Kleinen!).

- 1: Ich werde mal das Programm laufen lassen.
- 2: Werde den Debugger anwerfen und nachsehen.
- 3: Ich ändere also die erste Instruktion zu `LDOU s,argv,8`.
- 4: Ich lasse das Programm mit allen meinen Beispielen laufen.
- 5: Eine leere Eingabe erzeuge ich mit `mmix par`. Was ist dann aber im `argv` gespeichert? Dazu lese ich das Faszikel: Es wird 0 als letztes Element des `argv` Feldes abgelegt. Wenn ich also `LDOU s,argv,8` für `mmix par` aufrufe, müßte `s=0` sein – und diesen Zeiger darf ich nicht dereferenzieren.
- 6: Ich schreibe eine Fallunterscheidung an den Anfang. Bei einem leeren String muß ich laut Aufgabe ok ausgeben.

Ausführen Führen Sie genau die geplanten Änderungen durch; lassen Sie sich nicht beirren, wenn Sie auf weitere Ungereimtheiten stoßen – ändern Sie eine Sache nach der anderen, notieren Sie sich auf einem Blatt Papier lieber die Dinge, die *später* noch zu tun bleiben.

- 1: Ich rufe `mmix par '(')` auf, dann `mmix par '(('`, dann `mmix par '(('?`.
- 2: Ich trace die einzelnen Instruktionen.
- 3: gemacht.
- 4: Aufrufe: `mmix par bsp` laufen
- 5: Ich versuche, die Faszikel-Erklärung nachzuvollziehen. Wirklich: Beim Trace wird `s=0` gesetzt, d.h. `s` weist nicht auf einen gültigen String. Aber warum kommt am Ende trotzdem ok heraus?
- 6: Ich füge die Instruktion `BZ s,ok` noch ein.

Testen Es gibt nun zwei *Ziele*: 1) Weisen Sie nach, daß Änderungen nun das richtige Ergebnis liefern. 2) Die Ergebnisse der vorher schon bestandenen Testfälle dürfen sich nicht ändern.

- 1: Leider wird immer ok ausgegeben. Das kann ja nicht richtig sein.
- 2: Zunächst sehe ich, daß die Initialisierung von `s` aus `argv` geklappt hat und das LDBU am Schleifenkopf liest Zeichen – allerdings sind es nicht die erwarteten ASCII-Codes der Klammern. Was ist es dann? Ein `M400000000000020`" (die Adresse habe ich aus dem Trace) schafft Klarheit: `M8[#400000000000020]="par",0,0,0,0,0`
- 3: Die Änderung hatte den gewünschten Effekt: Nach der Ausführung der neuen Anweisung haben wir `$0=#400000000000028` und `M8[#400000000000028]="((",0,0,0,0,0,0`. Gut. Ich drücke `c` für *continue* und erhalte die erwartete Antwort `Klammern offen am Ende`.
- 4: Scheint alles ok zu sein. Wie ist es mit Spezialfällen?
- 5: Bei der leeren Eingabe verhält sich noch eine Instruktion etwas unschön: (LDBU) `rL=5, $4=1[4] = M1[#0+#0] = #0`, d.h. nach `s=0` folgt `si=0` – aber nur, weil der `mmix` Simulator die eigentlich nicht erlaubte Dereferenz zufällig ausführt und wir keinen TRIP-Handler eingetragen haben.
- 6: Ein erneuter Trace bestätigt, daß der Spezialfall „leere Eingabe“ jetzt korrekt behandelt wird.

7 Tipps zum Debuggen

- Testen Sie zuerst den 'Normalfall', dann den 'Fehlerfall', dann den 'Spezialfall'.
- Schreiben Sie ein Hauptprogramm, das ihren Code mit Eingaben versieht *bevor* Sie mit dem Code selbst beginnen.
- Isolieren Sie den Fehler, bevor Sie versuchen ihn zu beheben.
- Weisen Sie den Fehler nach: Erzeugen Sie eine Eingabe, die einen fehlerhaften Lauf erzeugt.
- Ändern Sie genau das Code-Stück, bei dem Sie den Fehler nachgewiesen haben.
- Ändern Sie nichts, ohne einen Fehler beobachtet zu haben – Spekulationen führen nur zu unnötigen Änderungen.
- Ändern Sie nie zwei Dinge auf einmal.
- Überlegen Sie bei jeder Änderung, oder besser testen Sie, daß die alten Beispiele noch durchlaufen.
- Knuth schlägt vor, jede Instruktion bei den ersten beiden Ausführungen zu *tracen* (also zu beobachten): `mmix -t2`
- Benutzen Sie Breakpoints. In `mmix -i` erhalten Sie sie mit *bx*adresse; die Adresse bekommen Sie aus dem Listing. Mit *c* (*continue*) starten Sie die Ausführung nach einer Unterbrechung bzw. am Anfang, sie läuft bis zum nächsten Breakpoint oder *trap*.
- Gute Kandidaten für Breakpoints sind:
 - Startmarken von Schleifen und Unterprogrammen
 - Die POP Befehle von Unterprogrammen
 - Stellen, an denen Sie wissen, daß das Ergebnis noch korrekt ist, obwohl es kurz danach falsch wird.
 - Der Anfang der Codesequenz, die ein beobachtetes falsches Ergebnis erzeugt (auch wenn man noch nicht weiß, ob deren Eingabe richtig ist).
 - In der *Mitte* einer Anweisungsfolge, in der irgendwo ein Fehler auftritt (Divide and conquer!)
 - Instruktionen, von denen Sie vermuten, daß sie einen Fehler verursachen.
- Halten Sie für die möglichen Breakpoints schon beim Entwurf des Programms Testdaten (d.h. Eingaben und erwartet Ergebnisse) bereit.
- Lesen Sie die für Fehler in Frage kommenden Code-Stücke nochmals sorgfältig durch – nach langem Debuggen neigt man dazu, nur noch mit dem Debugger die Fehler zu suchen anstatt das Denken wieder einzuschalten.
- Schauen Sie sich für Spezialfälle genau an, wie diese im Code behandelt werden. Stellen Sie sicher, daß sie explizit behandelt werden und nicht durch womöglich maschinenabhängige Zufälle korrekt sind.

8 Das fertige Programm

```
"par.mmas" 7 ≡
                LOC #100
s                IS $0
i                IS $1
t                IS $255
c                IS $2
si              IS $4
argv            IS $1
Main            SWYM 0
                LDOU s,argv,8
                BZ  s,ok
                SET i,0
loop_c          SWYM 0
                LDBU si,s,i
                BZ  si,end_loop
OH              CMP t,si,'('
                BNZ t,OF
                INCL c,1
OH              CMP t,si,')'
                BNZ t,OF
                SUBU c,c,1
                BNN c,OF
                GETA t,err1
                TRAP 0,Fputs,StdErr
                TRAP 0,Halt,0
OH              INCL i,1
                JMP loop_c
end_loop        BZ  c,ok
                GETA t,err2
                TRAP 0,Fputs,StdErr
                TRAP 0,Halt,0
ok              GETA t,okstr
                TRAP 0,Fputs,StdErr
                TRAP 0,Halt,0
                TRAP 0,Halt,0
                TRAP 0,Halt,0
                LOC (@+3)&-4
err1            BYTE "unerwartete schliessende Klammer",#a,0
                LOC (@+3)&-4
okstr           BYTE "ok",#a,0
                LOC (@+3)&-4
err2            BYTE "Klammern offen am Ende",#a,0◊
```