Übersetzerbau LU
Skriptum

Anton Ertl
Andreas Krall
Markus Schordan

2006

- Allgemeines und Beispiele
- GNU Emacs Reference Card
- Alpha Assembler Handbuch
- make: A Program for Maintaining Programs
- lex — a Lexical Analyzer Generator
- yacc — Yet Another Compiler-Compiler
- Ox: Tutorial Introduction
- `burg`, `iburg` und `bfe`

1 Rechner

In den Übungsräumen in der Argentinierstraße 8, Erdgeschoß stehen Ihnen ca. 25 X-Terminals als Arbeitsplätze zur Verfügung. Die offiziellen Öffnungszeiten des Labors sind Montag bis Freitag 9h-17h, jedoch sind die Übungsräume normalerweise wochentags bis 22h und samstags bis 17h zugänglich (es kommt aber vor, dass die Eingangstür schon früher versperrt wird). Die Übungsrechner sind rund um die Uhr in Betrieb, sodass Sie sich von auswärts (z.B. von den Benutzerräumen des ZID) auch zu anderen Zeiten einloggen können. Sollte es allerdings außerhalb der offiziellen Öffnungszeiten zu einem technischen Problem (z.B. Absturz) kommen, wird das Problem erst am nächsten Arbeitstag behoben.

Auf den X-Terminals können Sie Verbindungen zu verschiedenen Computern auswählen. Die Übungsmaschine ist die b1; sollte sie längerfristig ausfallen, steht als Ersatzmaschine die b2 zur Verfügung (Sie können sich aber vorerst nicht auf die Ersatzmaschine einloggen). Sie können sich von auswärts mit `ssh b1.complang.tuwien.ac.at` einloggen.

Vor dem Einloggen sollten Sie einen Doppelklick auf das *Ende*-Icon machen oder zweimal Ctrl Alt Backspace drücken (X-server reset, verbessert die Stabilität). Nach dem Einloggen erscheint ein Emacs-Fenster und einige andere. Sie können die Session beenden, indem Sie einen X-Server-Reset auslösen (z.B. per Doppelklick auf das *Ende*-Icon).

Auf allen Arbeitsplätzen liegt die Meta-Taste auf Alt.

Wir haben keine Möglichkeit, Dateien von oder auf Diskette o.ä. zu überspielen. Falls Sie zuhause arbeiten wollen, müssen Sie Ihre Dateien mit `scp` (eine ssh-Anwendung) auf unsere Rechner übertragen.

Die in der Übung verwendeten Werkzeuge sind für verschiedene Plattformen auf `http://www.complang.tuwien.ac.at/ublu/tools/` erhältlich.

Wenn Sie selbst ein `.forward`-File einrichten oder ändern, testen Sie es unbedingt! Wenn es nicht funktioniert, haben wir keine Möglichkeit, Sie zu erreichen (z.B. um Ihnen die Ergebnisse der Abgabe mitzuteilen).

Nach den Erfahrungen der letzten Jahre kommt es kurz vor den Abgabeterminen zu großem Andrang in den Übungsräumen. Wir empfehlen daher, möglichst zu anderen, von Tutoren betreuten Zeiten zu kommen.

2 Betreuung, Information

Verlautbarungen zur Übung (z.B. Klarstellungen zur Angabe) gibt es in der Newsgroup tuwien.lva.uebersetzerbau-lu; derzeit ist sie nur auf wenigen Newsservern vorhanden; auf `http://www.complang.tuwien.ac.at/ublu/newsgroup.html` werden verschiedene Zugriffsmöglichkeiten erläutert. Stellen Sie Fragen, deren Antwort vermutlich auch andere Übungsteilnehmer in-

teressiert, bitte dort. Wir bemühen uns, Fragen innerhalb eines Arbeitstages zu beantworten. Bitte schauen Sie zuerst, ob die Frage nicht schon gestellt und beantwortet wurde. Wenn Ihre Frage schon früher beantwortet wurde, wird die Antwort auf Ihre Frage meist die Message-Id der alten Antwort sein. Mit der Message-Id kommen Sie auf folgende Weise an den Artikel, wenn Ihr Newsreader diese Funktion nicht bietet:

```
telnet news.tuwien.ac.at nntp
article <Message-Id>
```

Im WWW finden Sie unter `http://www.complang.tuwien.ac.at/ublu/` Informationen zur Übung.

Während der Übung stehen zu gewissen Zeiten in den Übungsräumen Tutoren bereit (siehe `http://www.complang.tuwien.ac.at/ublu/stundenplan.txt`). Wenn die Tutoren Ihre Frage nicht beantworten können, erreichen Sie unter ublu@mips.complang.tuwien.ac.at die betreuenden Assistenten; Email zu Übungsthemen an andere Accounts wird von den Empfängern möglicherweise ignoriert. Wenn Sie Ihre Fragen gerne persönlich stellen, kommen Sie am Montag von 10h-11h zu Anton Ertl in die Sprechstunde.

Technische Probleme wie Computerabstürze, Druckerprobleme, falsche Permissions sind eine Sache für den Techniker. Wenden Sie sich direkt an ihn: email an Herbert Pohlai (`root@mips.complang.tuwien.ac.at`), Tel. 18525.

3 Beispiele

Die Beispiele finden Sie weiter hinten im Skriptum. Beachten Sie, dass die ersten Beispiele erfahrungsgemäß wesentlich leichter sind als die Beispiele „Attributierte Grammatik“, „Gesamtbeispiel“ und „Optimierende Codeerzeugung“. Versuchen Sie, mit den ersten Beispielen möglichst rasch fertig zu werden, um genügend Zeit für die schwierigeren zu haben.

4 Beurteilung

Ihre Note wird aufgrund der Qualität der von Ihnen abgegebenen Programme ermittelt. Das Hauptkriterium ist dabei die Korrektheit. Sie wird mechanisch überprüft, Sie erhalten per Email das Ergebnis der Prüfung. Wenn Sie meinen, dass sich das Prüfprogramm geirrt hat, wenden Sie sich an einen Tutor.

Die Prüfprogramme sind relativ einfach, dumm und kaum fehlertolerant. Damit Sie prüfen können, ob Ihr Programm im richtigen Format ausgibt und ähnliche wichtige Kleinigkeiten, stehen Ihnen die Testprogramme und

einige einfache Testeingaben und -resultate zur Verfügung; Sie können die Testprogramme auch benutzen, um Ihre Programme mit eigenen Testfällen zu prüfen (siehe `http://www.complang.tuwien.ac.at/ublu/`).

Beachten Sie, dass bei der Abgabe die Überprüfung mit wesentlich komplizierteren Testfällen erfolgt als denen, die wir Ihnen vorher zur Verfügung stellen. Ein erfolgreiches Absolvieren der Ihnen vorher zur Verfügung stehenden Tests heißt also noch lange nicht, dass Ihr Programm korrekt ist. Sie müssen sich selbst weitere Testfälle überlegen (wie auch im Berufsleben).

Ihre Programme werden zu den angegebenen Terminen kopiert und später überprüft. Ändern Sie zu den Abgabeterminen zwischen 14h und 15h nichts im Abgabeverzeichnis, damit es nicht zu inkonsistenten Abgaben kommt.

Ein paar Tage nach der Abgabe erhalten Sie das Ergebnis per Email. Das Ausschicken der Ergebnisse wird auch in `at.tuwien.lva.uebersetzerbau-lu` verkündet, Sie brauchen also nicht nachfragen, wenn Sie dort noch nichts gesehen haben. Eine Arbeitswoche nach der ersten Abgabe werden Ihre (eventuell von Ihnen verbesserten) Programme erneut kopiert und überprüft. Diese Version wird mit 70% der Punkte eines rechtzeitig abgegebenen Programms gewertet. Das ganze wiederholt sich zwei Arbeitswochen nach dem ersten Abgabetermin (30% der Punkte). Sie erhalten für das Beispiel das Maximum der drei Ergebnisse.

Sollten Sie versuchen, durch Kopieren oder Abschreiben von Programmen eine Leistung vorzutäuschen, die Sie nicht erbracht haben, erhalten Sie keine positive Note. Die Kontrolle erfolgt in einem Gespräch am Ende des Semesters, in dem überprüft wird, ob Sie auch verstehen, was Sie abgegeben haben. Weitere Maßnahmen behalten wir uns vor.

Ihr Account ist nur für Sie lesbar. Bringen Sie andere nicht durch Ändern der Permissions in Versuchung, zu schummeln.

5 Weitere Dokumentation bzw. Werkzeuge

Es stehen folgende Werkzeuge zur Verfügung:

Name	online Doku	Bemerkung
emacs, vi	info emacs, man vi	Editor
gcc	info as	Assembler
gcc, ccc	info gcc, man ccc	C-Compiler
make	info make	baut Programme
flex	man flex	Scanner-Generator
yacc, bison	man yacc, info bison	Parser-Generator
ox	man ox	AG-basierter
	xdvi /usr/ftp/pub/ublu/oxURM.dvi	Compilergenerator
burg, iburg	man iburg, man burg	Baumparser-Generator
bfe	Skriptum	Präprozessor für burg
gdb	info gdb	Debugger
objdump	info objdump	Disassembler etc.
elm, mutt, mail	man elm, man mutt, man mail	Email
xrn	man xrn	Newsreader
lynx, mozilla firefox		WWW-Browser

Die mit „man“ gekennzeichnete Dokumentation können Sie lesen, indem sie auf der Kommandozeile `man ...` eintippen. Die mit „info“ gekennzeichnete Dokumentation können Sie lesen, indem sie in Emacs `C-h i` tippen. In der Dokumentation für Emacs bedeutet `C-`x Ctrl x und `M-`x Meta x (auf den Übungsgeräten also Alt x).

Alle Werkzeuge rufen Sie von der Shell-Kommandozeile aus auf, indem Sie ihren Namen tippen.

Mit flex erzeugte Scanner müssen normalerweise mit `-lfl` gelinkt werden.

Das auf den Übungsgeräten unter `yacc` aufrufbare Programm ist Berkeley yacc und ist näher mit Bison verwandt als mit AT&T yacc (für den Fall, dass Sie Diskrepanzen zwischen diesem yacc und dem auf kommerziellen Unices bemerken).

`mail` ist ein primitives Email-Werkzeug, `elm` und `mutt` sind etwas bequemer[1].

Das Ox User Reference Manual ist nicht in diesem Skriptum abgedruckt, sondern steht nur on-line zur Verfügung, da es relativ umfangreich ist und nur ein Teil der enthaltenen Information in dieser Übung nützlich ist.

[1] `elm` und `xrn` sind so vor-eingestellt, dass der schon laufende Emacs als Editor verwendet wird. Wenn Sie mit dem Editieren des Buffers fertig sind, tippen Sie `C-x #` und `elm` wird weitermachen.

6 Beispiele

Es sind insgesamt sieben Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen einiger grundlegender Befehle der Alpha-Architektur. In den weiteren fünf Beispielen wird eine Programmiersprache vollständig implementiert. Diese Beispiele bauen aufeinander auf, d.h. Fehler, die in den ersten Beispielen der Sprachimplementierung gemacht werden, sollten behoben werden, um in späteren Abgaben nicht mehr die Beurteilung zu verschlechtern. Bei der Implementierung der Sprache wird mit jedem Beispiel (ausgenommen das Gesamtbeispiel) auch ein neues Werkzeug eingeführt, das nach Einarbeitung in die Verwendungsweise des Werkzeugs die Arbeit erleichtert.

Die zu implementierende Sprache ist eingeschränkt, um den Arbeitsaufwand nicht zu groß werden zu lassen. So sind in dieser Sprache zwar grundlegende Kontrollstrukturen vorhanden und es können Variablen definiert werden, aber Datenstrukturen können innerhalb dieser Sprache nicht erzeugt werden. Sprachkonstrukte für Speicherzugriff und Adressarithmetik sind jedoch vorhanden. Daher müssen bei den letzten beiden Beispielen, um die Codegenerierung testen zu können, Datenstrukturen in einem C-Programm erzeugt werden und dann mit dem von Ihnen generierten Code gelinkt werden. Dadurch wird auch ein tieferes Verständnis vermittelt, wie verschiedene Sprachen miteinander kombiniert werden können. Die Kenntnisse, die Sie bei den ersten beiden Assembler-Beispielen erlangen, werden Sie somit auch wieder bei der Codegenerierung der Beispiele 6 und 7 verwenden können. Die Beispiele 3-7 können alle aufeinander aufbauend implementiert werden, d.h. wenn Sie Ihr Programm von Anfang an gut entwerfen, können Sie dieses ab Beispiel 3 bis zum Gesamtbeispiel 7 stets wiederverwenden und erweitern. Beachten Sie jedoch, dass bei jeder Abgabe stets das gesamte Quellprogramm im Abgabeverzeichnis vorhanden sein muss (und zwar nicht in Form von symbolic links).

In den folgenden Abschnitten finden Sie die Angaben und Erklärungen für die Modalitäten der Beispielabgaben. Von der Sprache wird in jedem Abschnitt immer nur soviel erklärt, wie für das jeweilige Beispiel notwendig ist. Wenn Sie einen Überblick über die gesamte Sprache haben wollen, sollten Sie sich gleich am Anfang alle Angaben durchlesen.

6.1 Assembler A

6.1.1 Termin

Abgabe spätestens am 15. März 2006, 14 Uhr.

6.1.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned long asma_ref(unsigned long l)
{
  unsigned char *s = (unsigned char *)&l;
  int i;
  unsigned long r=0;
  for (i=0; i<sizeof(l); i++)
    r += s[i];
  return r;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `perr` (viel mehr ist übrigens auch nicht nötig, insbesondere keine Schleife).

Am einfachsten tun Sie sich dabei wahrscheinlich, wenn Sie eine einfache C-Funktion wie

```
long asma(unsigned long a)
{
  return a;
}
```

mit z.B. `gcc -O -S` in Assembler übersetzen und sie dann erweitern. Dann stimmt schon das ganze Drumherum. Die Originalfunktion auf diese Weise zu übersetzen ist auch recht lehrreich, aber vor allem, um zu sehen, wie man es nicht machen soll. Um `perr` zu assemblieren, muss der Prozessortyp mit `.arch ev6` deklariert werden.

6.1.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version `perr` verwendet und korrekt ist, also bei gleicher (zulässiger) Eingabe das gleiche Resultat liefert wie das Original.

Zum Assemblieren und Linken verwendet man am besten `gcc`, der Compiler-Treiber kümmert sich dann um die richtigen Optionen für `as` und `ld`.

6.1.4 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/asma` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und `make` soll eine Datei `asma.o` erzeugen. Diese Datei soll nur die Funktion `asma` enthalten, keinesfalls `main`. Diese Funktion soll den Alpha-Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

6.2 Assembler B

6.2.1 Termin

Abgabe spätestens am 22. März 2006, 14 Uhr.

6.2.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned long asmb_ref(unsigned char *s, unsigned long n)
{
  unsigned long i;
  unsigned long r=0;
  for (i=0; i<n; i++)
    r += s[i];
  return r;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `perr`. Sie dürfen dabei annehmen, dass hinter dem letzten Zeichen von s noch 8 Zeichen zugreifbar sind.

Für besonders effiziente Lösungen (gemessen an der Anzahl der *ausgeführten* Maschinenbefehle) gibt es Bonuspunkte. Allerdings dürfen dabei keine Alignment-Fehler auftreten (die Korrektur solcher Fehler durch Linux wird bei den Testläufen ausgeschaltet). `uldq` wird in mehrere Maschinenbefehle übersetzt; die tatsächlichen Maschinenbefehle sehen Sie mit `objdump -d`.

6.2.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version korrekt ist, also bei jeder zulässigen Eingabe das gleiche Resultat liefert wie das Original. Dadurch können Sie viel mehr verlieren als Sie durch Optimierung gewinnen können, also optimieren Sie im Zweifelsfall lieber nicht.

Die Vertrautheit mit dem Assembler müssen Sie beim Gespräch am Ende des Semesters beweisen, indem Sie Fragen zum abgegebenen Code beantworten.

6.2.4 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/asmb` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und `make` soll eine Datei `asmb.o` erzeugen. Diese Datei soll nur die Funktion `asmb` enthalten, keinesfalls `main`. Diese Funktion

soll den Alpha-Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

6.3 Scanner

6.3.1 Termin

Abgabe spätestens am 29. März 2006, 14 Uhr.

6.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `func end if then else exit start repeat not and`; weiters soll er auch noch folgende Lexeme erkennen: `; ( ) , * + -> - > =`.

Identifier bestehen aus Buchstaben, Ziffern und _, dürfen aber nur mit Buchstaben und _ beginnen. Zahlen bestehen ausschließlich aus Dezimalziffern (führende Nullen sind erlaubt). Leerzeichen, Tabs und Newlines zwischen den Lexemen sind erlaubt und werden ignoriert, ebenso Kommentare, die mit `/*` anfangen und bis zum nächsten `*/` gehen (Kommentare können also nicht geschachtelt werden). Alles andere sind lexikalische Fehler. Es soll jeweils das längste mögliche Lexem erkannt werden, `if39` ist also ein Identifier (longest input match), `39if` ist eine Zahl gefolgt vom Schlüsselwort `if`.

Der Scanner soll für jedes Lexem eine Zeile ausgeben: für Schlüsselwörter und Lexeme aus Sonderzeichen soll das Lexem ausgegeben werden, für Identifier „`id` “ gefolgt von dem String des Identifiers, für Zahlen „`num` “ gefolgt von der Zahl in Hexadezimaldarstellung ohne führende Nullen und ohne 0x-Prefix. Für Leerzeichen, Tabs, Newlines und Kommentare soll nichts ausgegeben werden (auch keine Leerzeile).

Der Scanner soll zwischen Groß- und Kleinbuchstaben unterscheiden, `If` ist also kein Schlüsselwort.

6.3.3 Abgabe

Legen Sie ein Verzeichnis `~/abgabe/scanner` an, in das Sie die maßgeblichen Dateien stellen. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können (auch den ausführbaren Scanner) und mittels `make` ein Programm namens `scanner` erzeugen, das von der Standardeingabe liest und auf die Standardausgabe ausgibt. Korrekte Eingaben sollen akzeptiert werden (Ausstieg mit Status 0, z.B. mit `exit(0)`), bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden. Bei einem lexikalischen Fehler darf der Scanner Beliebiges ausgeben (eine sinnvolle Fehlermeldung hilft bei der Fehlersuche).

6.4 Parser

6.4.1 Termin

Abgabe spätestens am 5. April 2006, 14 Uhr.

6.4.2 Angabe

Gegeben ist die Grammatik (in `yacc`/`bison`-artiger EBNF):

```
Program: { Funcdef ';' }
       ;

Funcdef: func id '(' Ids ')' Prodtrans end
       ;

Ids: id { ',' id }      /* Parameterdefinition */
   ;

Prod: ( Exprs | If ) '->'
    ;

Prodtrans: Prod { Tran }
         ;

Tran: Loop '->'
    | Consumers ';' { Ifexit ';' } Prod
    ;

If: if Bool then Prodtrans else Prodtrans end
  ;

Ifexit: if Bool then Prodtrans exit
      ;

Loop: start '->' { Tran } repeat /* Schleife */
    ;

Consumers: Consumer { ',' Consumer }
         ;

Consumer: id          /* Variablendefinition */
        | '*' Term    /* schreibender Speicherzugriff */
        ;
```

```
Exprs: Expr { ',' Expr }
     ;

Expr: '-' Term
    | '*' Term                  /* lesender Speicherzugriff */
    | Term { '+' Term }
    | Term { '*' Term }
    ;

Term: '(' Expr ')'
    | num
    | id                        /* Variablenverwendung */
    | id '(' Exprs ')'          /* Funktionsaufruf */
    ;

Bool: Bterm { and Bterm }
    | [ not ] Bterm
    ;

Bterm: Expr ( '>' | '=' ) Expr
     | '(' Bool ')'
     ;
```

Schreiben Sie einen Parser für diese Sprache mit `flex` und `yacc`/`bison`. Die Lexeme sind die gleichen wie im Scanner-Beispiel (`id` steht für einen Identifier, `num` für eine Zahl). Das Startsymbol ist `Program`.

6.4.3 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/parser` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `parser` erzeugen, das von der Standardeingabe liest. Korrekte Programme sollen akzeptiert werden (Ausstieg mit Status 0, z.B. mit `exit(0)`), bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei Syntaxfehlern der Fehlerstatus 2. Das Programm darf auch etwas ausgeben (auch bei korrekter Eingabe), z.B. damit Sie sich beim Debugging leichter tun.

6.4.4 Hinweis

Die Verwendung von Präzedenzdeklarationen von `yacc` kann leicht zu Fehlern führen, die man nicht so schnell bemerkt (bei dieser Grammatik sind sie

sowieso sinnlos). Konflikte in der Grammatik sollten Sie durch Umformen der Grammatik beseitigen; `yacc` löst den Konflikt zwar irgendwie, aber nicht unbedingt in der gewünschten Art.

6.5 Attributierte Grammatik

6.5.1 Termin

Abgabe spätestens am 10. Mai 2006, 14 Uhr.

6.5.2 Angabe

Erweitern Sie den Parser aus dem letzten Beispiel mit Hilfe von `ox` um eine Symboltabelle und eine statische Analyse.

Die *hervorgehobenen* Begriffe beziehen sich auf Kommentare in der Grammatik.

6.5.2.1 Namen. Es gibt Funktionsnamen (unmittelbar hinter `func` und im *Funktionsaufruf*) und Variablennamen (alle anderen).

Funktionsnamen dürfen, soweit es die statische Analyse betrifft, beliebig verwendet werden, die statische Analyse braucht sie also überhaupt nicht behandeln.

Variablen werden in der *Parameterdefinition* oder in der *Variablendefinition* definiert. Ihr Sichtbarkeitsbereich beginnt mit der ersten ')' bzw. dem ersten ';' nach der Definition und endet am Ende eines der folgenden Konstrukte: der Funktion, einer *Schleife*, oder eines `Prodtrans`; und zwar endet der Sichtbarkeitsbereich am Ende desjenigen Konstrukts, das die Definition unmittelbar umschließt.

Variablen, die gleichzeitig sichtbar sind, dürfen nicht den gleichen Namen haben.

6.5.2.2 Kontrollstruktur. Ein `Ifexit` darf nur innerhalb einer *Schleife* vorkommen. Das `Ifexit` erlaubt das Verlassen der *Schleife*, die das `Ifexit` unmittelbar umschließt. Jede Schleife muß mindestens ein `Ifexit` enthalten, das das Verlassen dieser Schleife erlaubt.

6.5.2.3 Datenfluss. In dieser Sprache fließen an den `->` ein oder mehrere Werte von links nach rechts. Dabei müssen links genausoviele Daten produziert werden wie rechts konsumiert werden, und das soll von der statischen Analyse geprüft werden. Produziert werden die Daten letztendlich immer in `Exprs`, konsumiert in `Consumers` oder am Ende der Funktion. Und zwar produziert `Exprs` soviele Werte, wie `Exprs` direkt vorkommen, und `Consumers`

konsumiert so viele, wie `Consumer`s vorkommen; das Funktionsende konsumiert genau einen Wert.

Bei einem Kontrollfluss-Konstrukt (z.B.: `else exit repeat`) folgt der Datenfluss dem Kontrollfluss. Die genauen Regeln dafür lauten:

- Die beiden `Prodtrans` in einem `If` müssen gleich viele Werte produzieren, und das `If` produziert dann auch so viele Werte.
- Die `Prodtrans`s aller `IfExit`s, die die gleiche *Schleife* verlassen können, müssen gleich viele Werte produzieren. Die *Schleife* produziert dann auch so viele Werte.
- In eine *Schleife* (`start`) müssen genauso viele Werte fließen wie vom Ende der *Schleife* (`repeat`) wieder zum Anfang zurückfließen. Diese Werte fließen auch vom Anfang der *Schleife* in das erste `Tran`.

6.5.2.4 Beispiel. Hier ein Beispiel für die Kombination aus Datenfluss und Kontrollfluss: Eine Schleife, in die drei Werte fließen und die (per `Ifexit`) einen Wert produziert; und ein `If`, das zwei Werte produziert:

```
func x(a,b)
  1,2,3 -> start -> d,e,f;
    if a>b then
      d,e ->
    else
      e,d ->
    end -> g,h;
    if g>h then
      f -> exit;
    e,f,d -> repeat
  -> /* 1 Wert */ end;
```

6.5.3 Hinweise

Offenbar übersehen viele Leute, dass attributierte Grammatiken Information auch von rechts nach links (im Ableitungsbaum) weitergeben können. Sie denken sich dann recht komplizierte Lösungen aus. Dabei reichen die von `ox` zur Verfügung gestellten Möglichkeiten vollkommen aus, um zu einer relativ einfachen Lösung zu kommen.

Verwenden Sie keine globalen Variablen oder Funktionen mit Seiteneffekten bei der Attributberechnung! `ox` macht globale Variablen einerseits unnötig, andererseits auch fast unbenutzbar, da die Ausführungsreihenfolge nicht vollständig festgelegt ist.

Sie brauchen angeforderten Speicher (z.B. für Symboltabellen-Einträge oder Typinformation) nicht freigeben, die Testprogramme sind nicht so groß, dass der Speicher ausgeht (zumindest wenn Sie's nicht übertreiben).

Das Werkzeug Torero (http://www.complang.tuwien.ac.at/torero/) ist dazu gedacht, bei der Erstellung von attributierten Grammatiken zu helfen.

6.5.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/ag`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `ag` erzeugen, das von der Standardeingabe liest. Korrekte Programme sollen akzeptiert werden, bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei Syntaxfehlern der Fehlerstatus 2, bei anderen Fehlern (z.B. Verwendung eines nicht sichtbaren Namens) der Fehlerstatus 3. Die Ausgabe kann beliebig sein, auch bei korrekter Eingabe.

6.6 Gesamtbeispiel

6.6.1 Termin

Abgabe spätestens am 31. Mai 2006, 14 Uhr.

6.6.2 Angabe

Erweitern Sie die statische Analyse aus dem AG-Beispiel zu einem Compiler, der, die die Programme in Alpha-Assemblercode übersetzt. Sie dürfen dabei burg/iburg verwenden, müssen es aber nicht. Ein Teil der Sprache wurde schon im Beispiel attributierte Grammatik erklärt, hier der Rest:

6.6.2.1 Ausdrücke und `Consumers`. Diese Programmiersprache kennt nur einen Datentyp: das 64-bit-Wort, das als vorzeichenbehaftete Zahl oder als Adresse verwendet werden kann. Weder der Compiler noch das Laufzeitsystem sollen eine Typüberprüfung vornehmen. Bei Speicherzugriffen muss der Programmierer (der Anwender des Compilers) wissen, was er tut, der Compiler soll (und kann) das nicht überprüfen. Unsere Testprogramme führen keine Zugriffe auf ungültige Adressen aus.

Ausdrücke (`Exprs`) arbeiten auf 64-bit-Worten und liefern solche Werte als Ergebnis. `+`, `-` und das binäre `*` haben ihre übliche Bedeutung (ein etwaiger Überlauf soll ignoriert werden).

Die *Variablendefinition* speichert den hereinkommenden 64-bit-Wert unter dem Namen der Variable. Die *Variablenverwendung* hat als Ergebnis den Wert, der in der genannten Variable gespeichert ist.

Bei einem Speicherzugriff ist der von `Term` berechnete Wert die Adresse der Speicherstelle. Der lesende Speicherzugriff liefert als Resultat den 64-bit-Ausdruck an dieser Adresse. Beim schreibenden Speicherzugriff wird der zu schreibende 64-Bit-Wert an diese Adresse gespeichert.

Der *Funktionsaufruf* wertet alle Ausdrücke aus und ruft dann die Funktion `id` auf, mit den Ergebnissen der Ausdrücke als Parameter. Das Ergebnis des Ausdrucks in der Funktion ist der Rückgabewert der Funktion. Dabei soll die Alpha-Aufrufkonvention verwendet werden.

Die Reihenfolge von Produzenten und Konsumenten ist gleich. Bei `1,2 -> a,b` hat also `a` den Wert 1 und `b` den Wert 2.

6.6.2.2 Kontrollfluss. `Bool`s werden nach der Kontrollflussmethode ausgewertet.

`>` und `=` vergleichen die beiden Ausdrücke als vorzeichenbehaftete Zahlen. `not` hat seine übliche Bedeutung.

`and` wertet den ersten `Bterm` aus. Ist das Ergebnis „falsch“, ist der gesamte Ausdruck „falsch“. Ist das Ergebnis „wahr“, dann wird der zweite Ausdruck ausgewertet, und sein Ergebnis ist das Ergebnis des gesamten Ausdrucks.

`If` und `Ifexit` werten zunächst den `Bool` aus. `If` führt dann, je nach Ergebnis, entweder den `then`- oder den `else`-Zweig aus; danach geht der Kontrollfluss hinter dem `end` weiter. Bei `Ifexit` geht im Falle einer wahren Bedingung nach Ausführung des `then`-Zweiges der Kontrollfluss hinter dem `repeat` jener *Schleife* weiter, die vom `Ifexit` verlassen wird. Ergibt die Bedingung falsch”, geht der Kontrollfluss hinter dem `exit` weiter (ohne Ausführung des `then`-Zweiges).

Erreicht der Kontrollfluss in einer *Schleife* das Ende (`repeat`), springt sie wieder an den Anfang.

Der Datenfluss folgt dem Kontrollfluss, also erhält in dem Fragment

```
1 -> start -> a;
  if a=2 then
    3 -> exit;
  2 -> repeat
-> b;
```

in der ersten Iteration `a` den Wert 1, in der zweiten Iteration den Wert 2, dann wird die *Schleife* verlassen, und `b` erhält den Wert 3.

Wenn der Kontrollfluss das Ende der Funktion erreicht, dann erfolgt ein Return aus der Funktion, und der Rückgabewert ist der Wert, der das Funktionsende erreicht hat.

6.6.2.3 Erzeugter Code. Ihr Compiler soll Alpha-Assemblercode ausgeben. Jede Funktion im Programm verhält sich gemäß der Alpha-Aufrufkonvention. Der erzeugte Code wird nach dem Assemblieren und Linken von C-Funktionen oder Funktionen, die Ihr Compiler compiliert hat, aufgerufen. Und Ihr Code ruft wiederum C-Funktionen auf oder Funktionen, die Ihr Compiler compiliert hat.

Der Name einer Funktion soll als Label am Anfang des erzeugten Codes verwendet werden und das Symbol soll exportiert werden; andere Symbole soll Ihr Code nicht exportieren.

Folgende Einschränkungen sind dazu gedacht, Ihnen gewisse Probleme zu ersparen, die reale Compiler bei der Codeauswahl und Registerbelegung haben. Sie brauchen diese Einschränkungen nicht überprüfen, unsere Testeingaben halten sich an diese Einschränkungen (eine Überprüfung könnte Ihnen allerdings beim Debuggen Ihrer eigenen Testeingaben helfen): Funktionen haben maximal 6 Parameter. Die Anzahl der gleichzeitig sichtbaren Variablen einer Funktion ist maximal 6. Die maximale Tiefe eines Ausdrucks plus die Anzahl der Ausdrücke, die unmittelbar in `Exprs`, `Consumers` oder `Bterm` vorkommen, ist ≤ 6 (Tiefe eines Ausdrucks: Anzahl der Ableitungen von `Expr` zwischen einem Blatt des Syntaxbaums und dem nächsten `Exprs`, `Consumers` oder `Bterm`). Die im Quellprogramm vorkommenden Zahlen sind kleiner als 256; allerdings geben wir für konstante Ausdrücke keine solche Garantie (z.B. kann folgendes vorkommen: `255*255`).

6.6.3 Hinweise

Eine einfache Strategie bezüglich der Parameter der aktuellen Funktion ist, sie nicht in den Argumentregistern zu lassen, sondern sie z.B. in gesicherte Register zu kopieren, damit man beim Berechnen der Parameter einer anderen Funktion problemlos auf sie zugreifen kann.

6.6.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/gesamt`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `gesamt` erzeugen, das von der Standardeingabe liest und auf die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen Fehlern der Fehlerstatus 3. Im Fall eines Fehlers kann die Ausgabe beliebig sein. Verwenden Sie keine globalen Variablen. Der ausgegebene Code muss vom Assembler verarbeitet werden können.

6.7 Optimierende Codeerzeugung

6.7.1 Termin

Abgabe spätestens am 14. Juni 2006, 14 Uhr. Achtung, bei diesem Beispiel gibt es nur einen Nachtermin!

6.7.2 Angabe

Erweitern Sie Ihren Compiler mit Hilfe von burg/iburg so, dass er besseren Code erzeugt.

Der erzeugte Code soll korrekt sein und möglichst wenige Befehle ausführen. Dabei ist nicht an eine zusätzliche Optimierung (wie z.B. common subexpression elimination) gedacht, sondern vor allem an die Dinge, die Sie mit `burg/iburg` tun können, also eine gute Codeauswahl (z.B. Zusammenfassen von `*(a+8)` zu einem Load- oder Store-Befehl) und einige algebraische Optimierungen[2]. Dafür gibt es dann besondere Testfälle, in denen Code, der mit solchen Optimierungen verbessert werden kann, in relativ grossem Maß vorkommt; diese Testfälle werden dann für die Bewertung herangezogen.

In diesem Beispiel erhalten Sie Punkte dafür, dass Ihr Compiler schnelleren Code produziert als ein nicht allzu schlechter Compiler, der keine solchen Optimierungen vornimmt. Allerdings erhalten Sie wieder Punkteabzüge für nicht bestandene Testfälle. Wenn Sie also nur einen korrekten, aber langsamen Compiler abgeben, gibt es genauso keine Punkte, wie wenn sie einen Compiler abgeben, der in einigen Fällen sehr schnellen Code erzeugt, in vielen anderen aber falschen Code. Wenn Ihr Compiler aber besonders guten Code erzeugt und korrekt ist, sind auch mehr als 100% bei diesem Beispiel möglich.

6.7.3 Hinweis

Bei der Registerbelegung gibt es sowohl ein großes Optimierungspotential als auch ein großes Fehlerpotential, besonders im Zusammenhang mit (verschachtelten) Funktionsaufrufen. Allerdings wird dieser Aspekt von unseren Benchmark-Testfällen nicht betont (weil das keine Optimierung ist, bei der `burg` hilft).

Beachten Sie, dass es leicht ist, durch eine falsche Optimierungsregel alle Punkte zu verlieren, die Sie durch Optimierung überhaupt gewinnen können. Testen Sie daher ihre Optimierungen besonders gut. Überlegen Sie sich, welche Optimierungen es wohl wirklich bringen (welche Fälle also tatsächlich vorkommen), und lassen Sie die anderen weg.

[2]Einige Beispiele für solche Optimierungen finden Sie auf `http://www.complang.tuwien.ac.at/papers/ertl00dagstuhl.ps.gz`.

6.7.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/code`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `code` erzeugen, das von der Standardeingabe liest und den generierten Code auf die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen Fehlern der Fehlerstatus 3. Im Fall eines Fehlers kann die Ausgabe beliebig sein.