Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2015

- Allgemeines und Beispiele
- GNU Emacs Reference Card
- AMD64-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 Anmeldung

Melden Sie sich in unserem Anmeldesystem `https://www.complang.tuwien.ac.at/anmeldung/` für die Lehrveranstaltung an. Mit der Anmeldung wird ein Account für Sie auf unserer Übungsmaschine `g0.complang.tuwien.ac.at` eingerichtet, der Accountname ist `u` gefolgt von der Matrikelnummer, z.B. u9999999. Das Passwort für diesen Account geben Sie bei der Anmeldung ein.

2 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 g0; sollte sie längerfristig ausfallen, steht als Ersatzmaschine die g2 zur Verfügung (Sie können sich aber vorerst nicht auf die Ersatzmaschine einloggen). Sie können sich von auswärts mit `ssh g0.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 USB-Sticks o.ä. zu überspielen. Falls Sie zuhause arbeiten wollen, müssen Sie Ihre Dateien für die Abgabe 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/ubvl/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 manchmal zu großem Andrang in den Übungsräumen. Wir emp-

fehlen daher, möglichst zu anderen Zeiten zu kommen.

3 Betreuung, Information

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

Verlautbarungen zur Übung (z.B. Klarstellungen zur Angabe) gibt es im Übungsforum (Details dazu siehe Übungshomepage).

Wenn Sie eine Frage zur Übung haben, stellen Sie sie am besten im Übungsforum (dann können auch andere von der Antwort profitieren). Sie können auch den Leiter der Übung per Email fragen `anton@mips.complang.tuwien.ac.at`, oder in die Sprechstunde kommen (Montag 10h-11h).

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

4 Beispiele

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

5 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 den Leiter der Übung.

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/ubvl/`).

Beachten Sie, dass bei der Abgabe die Überprüfung mit wesentlich komplizierteren Testfällen erfolgt als denen, die wir Ihnen vorher zur Verfügung stellen (vor allem ab dem Scanner-Beispiel). 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 im LVA-Forum 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.

6 Weitere Dokumentation bzw. Werkzeuge

Abbildung 1 zeigt die zur Verfügung stehenden Werkzeuge.

Die mit „man“ gekennzeichnete Dokumentation können Sie lesen, indem sie auf der Kommandozeile `man ...` eintippen. Die mit „info“ gekennzeichnete Dokumentation können Sie mit dem Programm `info` lesen, oder 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 `bison -y` (für den Fall, dass Sie Diskrepanzen zwischen diesem yacc und dem auf kommerziellen Unices bemerken). Mit `xvcg` können Sie sich die Ausgabe von `bison -g` anschauen.

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

[1] `mutt` 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 `mutt`/`xrn` wird weitermachen.

Name	online Doku	Bemerkung
emacs, vi	info emacs, man vi	Editor
gcc	info as	Assembler
gcc	info gcc	C-Compiler
make	info make	baut Programme
flex	man flex	Scanner-Generator
yacc, bison	man yacc, info bison	Parser-Generator
xvcg	man xvcg	Graphenzeichnen
ox	man ox	AG-basierter
	xdvi /usr/ftp/pub/ubvl/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.
mutt, mail	man mutt, man mail	Email
xrn	man xrn	Newsreader
lynx,		WWW-Browser
mozilla		
firefox		

Abbildung 1: Werkzeuge

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.

7 Beispiele

Es sind insgesamt acht Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen einiger grundlegender Befehle der AMD64-Architektur. In den weiteren Beispielen wird eine Programmiersprache vollständig implementiert. Diese Beispiele bauen aufeinander auf, d.h. Fehler, die Sie in den ersten Sprachimplementierungsbeispielen machen, sollten Sie beheben, damit sie in späteren Abgaben die Beurteilung nicht verschlechtern. Bei der Implementierung der Sprache wird mit jedem Beispiel (ausgenommen die letzten) 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. Zum Beispiel gibt es keine direkte Möglichkeit, Daten ein- oder auszugeben; diese Funktionen werden durch eine C-Funktion übernommen, die Funktionen in der Sparache aufruft, oder durch Aufrufen von C-Funktionen von Funktionen in unserer Sprache.

Die Kenntnisse, die Sie bei den Assembler-Beispielen erlangen, werden Sie auch wieder bei der Codegenerierung der letzten Beispiele verwenden. Die Beispiele 3-8 können alle aufeinander aufbauend implementiert werden, d.h. wenn Sie Ihr Programm von Anfang an gut entwerfen, können Sie dieses ab dem Scanner-Beispiel bis zum Gesamtbeispiel 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.

In dieser Sprache kann man, wie in den meisten Programmiersprachen, auch Programme schreiben, deren Semantik nicht definiert ist, und die Ihr Compiler trotzdem nicht als fehlerhaft erkennen muss und darf. Bei solchen Programmen ist es egal, welchen Code Ihr Compiler produziert (Code aus solchen Testeingaben wird von unseren Abgabescripts ohnehin nicht ausgeführt); Ihr Compiler sollte aber für Programme mit definierter Semantik korrekten Code produzieren.

7.1 Assembler A

7.1.1 Termin

Abgabe spätestens am 18.3.2015, 14 Uhr.

7.1.2 Angabe

Gegeben ist folgende C-Funktion:

```
void asma(unsigned long x[], unsigned long y[], unsigned long r[])
{
  unsigned long borrow, r0;
  r0 = x[0]-y[0];
  borrow = r0>x[0];
  r[0] = r0;
  r[1] = x[1]-y[1]-borrow;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `sbbq` und ohne Verwendung eines Vergleichsbefehls.

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

```
void asma(unsigned long x[], unsigned long y[], unsigned long sum[])
{
  sum[0] = x[0]-y[0];
}
```

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.

7.1.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version `sbbq` 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`.

7.1.4 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 Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

7.2 Assembler B

7.2.1 Termin

Abgabe spätestens am 25.3.2015, 14 Uhr.

7.2.2 Angabe

Gegeben ist folgende C-Funktion:

```
#include <stddef.h>
/* x, y haben n Elemente, sum hat n+1 Elemente */
void asmb(unsigned long x[], unsigned long y[],
          unsigned long r[], size_t n)
{
  unsigned long borrow, d;
  size_t i;
```

```
  borrow = 0;
  for (i=0; i<n; i++) {
    d = x[i]-y[i]-borrow;
    borrow = borrow ? d>=x[i] : d>x[i];
    r[i] = d;
  }
  r[i] = -borrow;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `sbbq`.

Für besonders effiziente Lösungen (gemessen an der Anzahl der *ausgeführten* Maschinenbefehle; wird ein Befehl n mal ausgeführt, zählt er n-fach) gibt es Bonuspunkte. Dabei könnte Ihnen das Wissen helfen, dass `add` und `sub` das Carry/Borrow-Flag verändern, `inc`, `dec`, `lea` und `mov` dagegen nicht.

7.2.3 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 weniger als mehr.

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

7.2.4 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 Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

7.3 Scanner

7.3.1 Termin

Abgabe spätestens am 15.4.2015, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende keywords unterscheiden kann: `fun if then else let in not head tail`

`and end isnum islist isfun`. Weiters soll er auch noch folgende Lexeme erkennen: `; = + - * . < = ( ) ->`

Identifier bestehen aus Buchstaben und Ziffern, dürfen aber nicht mit Ziffern beginnen.

Zahlen beginnen mit einer Ziffer und bestehen entweder ausschließlich aus Dezimalziffern oder aus `$` gefolgt von Hexadezimalziffern; führende Nullen sind bei Dezimalzahlen erlaubt, und sowohl Groß- als auch Kleinbuchstaben als Hex-Ziffern).

Leerzeichen, Tabs und Newlines zwischen den Lexemen sind erlaubt und werden ignoriert, ebenso Kommentare, die mit `//` anfangen und bis zum Ende der Zeile gehen. 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 keyword `if`.

Der Scanner soll für jedes Lexem eine Zeile ausgeben: für keywords und Lexeme aus Sonderzeichen soll das Lexem ausgegeben werden, für Identifier `ident` gefolgt von einem Leerzeichen und dem String des Identifiers, für Zahlen `num` gefolgt von einem Leerzeichen und der Zahl in Dezimaldarstellung ohne führende Nullen. 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 keyword.

7.3.3 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).

7.3.4 Hinweis

Die lex-Notation `$` steht für ein ein Zeilenende, auf das ein Newline folgt; zusätzlich kann auch noch das Ende der Eingabe die Zeile (und damit einen Kommentar) beenden. Am einfachsten ist es, nur zu spezifizieren, was ein Kommentar ist, und es dem longest input match zu überlassen, den Kommentar nicht zu früh abzubrechen.

7.4 Parser

7.4.1 Termin

Abgabe spätestens am 22.4.2015, 14 Uhr.

7.4.2 Angabe

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

```
Program: { Def ';' }
       ;

Def: ident = Lambda
   ;

Lambda: fun ident '->' Expr end
      ;

Expr: if Expr then Expr else Expr end
    | Lambda
    | let ident '=' Expr in Expr end
    | Term
    | { not | head | tail | isnum | islist | isfun } Term
    | Term { '+' Term }
    | Term '-' Term
    | Term { '*' Term }
    | Term { and Term }
    | Term { '.' Term }
    | Term ( '<' | '=' ) Term
    | Expr Term               /* Funktionsaufruf */
    ;

Term: '(' Expr ')'
    | num
    | ident                   /* Variablenverwendung */
    ;
```

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

7.4.3 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.

7.4.4 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, aber nicht unbedingt in der von Ihnen gewünschten Art.

Links- oder Rechtsrekursion? Also: Soll das rekursive Vorkommen eines Nonterminals auf als erstes (links) oder als letztes (rechts) auf der rechten Seite der Regel stehen? Bei `yacc`/`bison` und anderen LR-basierten Parsergeneratoren funktioniert beides. Sie sollten sich daher in erster Linie danach richten, was leichter geht, z.B. weil es Konflikte vermeidet oder weil es (im nächsten Beispiel) einfachere Attributierungsregeln erlaubt. Z.B. kann man mittels Linksrekursion bei der Subtraktion einen Parse-Baum erzeugen, der auch dem Auswertungsbaum entspricht. Sollte es keine anderen Gründe geben, kann man der Linksrekursion den Vorzug geben, weil sie mit einer konstanten Tiefe des Parser-Stacks auskommt.

7.5 Attributierte Grammatik

7.5.1 Termin

Abgabe spätestens am 6.5.2015, 14 Uhr.

7.5.2 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.

Auf der obersten Ebene (direkt in einem `Def`) definierte Namen sind im gesamten Programm sichtbar (auch vor der Definition).

Bei einem Lambda-Ausdruck ist der `ident` in `Expr` sichtbar.

Bei einem let-Ausdruck, ist der `ident` in dem `Expr` nach dem `in` sichtbar.

7.5.2.1 Überprüfungen.

Zu überprüfen ist:

In einer `Variablenverwendung` dürfen nur sichtbare Namen verwendet werden.

Die Sichtbarkeitsbereiche von zwei gleichen Namen dürfen sich nicht überlappen. Z.B. ist `f=fun f->0;` nicht erlaubt.

7.5.3 Hinweise

Es ist empfehlenswert, die Grammatik so umzuformen, dass sie für die AG günstig ist: Fälle, die syntaktisch gleich ausschauen, aber bei den Attributierungsregeln verschieden behandelt werden müssen, sollten auf verschiedene Regeln aufgeteilt werden; umgekehrt sollten Duplizierungen, die in dem Bemühen vorgenommen wurden, Konflikte zu vermeiden, auf ihre Sinnhaftigkeit überprüft und ggf. rückgängig gemacht werden. Testen Sie Ihre Grammatikumformungen mit den Testfällen.

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 (z.B. Funktionen, die übergebene Datenstrukturen ändern)! `ox` macht globale Variablen einerseits unnötig, andererseits auch fast unbenutzbar, da die Ausführungsreihenfolge der Attributberechnung nicht vollständig festgelegt ist. Bei Traversals ist die Reihenfolge festgelegt, und Sie können globale Variablen verwenden; seien Sie aber trotzdem vorsichtig.

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.

7.5.4 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.

7.6 Codeerzeugung A

7.6.1 Termin

Abgabe spätestens am 20.5.2015, 14 Uhr.

7.6.2 Angabe

Erweitern Sie die statische Analyse aus dem AG-Beispiel mit Hilfe von `iburg` zu einem Compiler, der folgende Untermenge der statisch korrekten Programme in AMD64-Assemblercode übersetzt: alle Programme, in denen keine `if`-Ausdrücke, keine `let`-Ausdrücke, und keine *Funktionsaufruf*e vorkommen, und in denen `Lambda` nur von Def und nicht von Expr abgeleitet wird.

Programme, die statisch korrekt sind, aber dieser Einschränkung nicht entsprechen, werden bei diesem Beispiel nicht als Testeingaben vorkommen. Allerdings kommen Programme, die diesen Einschränkungen nicht entsprechen, aber statisch nicht korrekt sind, als Eingaben zum Testen der statischen Überprüfungen vor, genauso wie fehlerhafte Eingaben zum Testen des Parsers und des Scanners.

Ein Teil der Sprache wurde schon im Beispiel attributierte Grammatik erklärt, hier der für dieses Beispiel notwendige Zusatz:

7.6.2.1 Datendarstellung. Es gibt drei Typen, die durch dynamische Typüberprüfung unterschieden werden: ganze Zahlen, Listenzellen und Closures; Closures werden in diesem Beispiel weder erzeugt noch kommen sie in einer anderen Form vor, und daher werden sie erst in einer späteren Angabe erklärt. Listenzellen bestehen aus zwei Feldern: dem Kopf (head) und dem Rest (tail); in beiden Feldern können beliebige Daten unserer Programmiersprache gespeichert werden.

Für die dynamische Typüberprüfung müssen die Daten mit tags versehen werden, die den Typ angeben. Wir verwenden folgendes Schema:

Wenn das niederwertigste Bit eines Wortes gleich 0 ist, ist das Datum eine vorzeichenbehaftete 63-Bit-Zahl; den Wert der Zahl erhält man durch bitweises arithmetisches Verschieben nach rechts um ein Bit.

Wenn das niederwertigste Bit eines Wortes gleich 1 ist, und das nächste Bit gleich 0 ist, ist das Datum ein Zeiger auf eine Listenzelle. Den tag-losen Zeiger erhält man, indem man vom Datum eins subtrahiert (die Adresse einer Listenzelle ist auf jeden Fall durch 8 teilbar). Das Kopffeld der Listenzelle ist auf Offset 0 vom tag-losen Zeiger, das Restfeld ist auf Offset 8. Eine Listenzelle ist 16 Bytes groß.

In Listen und bei der Parameterübergabe sind die Daten immer mit tags versehen, bei der Bearbeitung können sie dargestellt werden, wie es gerade paßt. Im folgenden wird bei Integern der Zahlenwert genannt, nicht die Darstellung mit tag.

7.6.2.2 Bedeutung der Operatoren. `+`, `-` und `*` erwarten ganze Zahlen als Operanden und haben ihre übliche Bedeutung (ein etwaiger Überlauf soll ignoriert werden). `and` und `not` erwarten ganze Zahlen als Operanden; `and` arbeitet bitweise, `not` invertiert das niederwertigste Nicht-Tag-Bit (z.B.: `not 2` ergibt 3, `not 1` ergibt 0).

Die folgenden Operationen liefern 1, wenn die Bedingung zutrifft, und 0, wenn nicht.

`<` und `=` vergleichen ihre Operanden. `<` funktioniert dabei nur für ganze Zahen, `=` für beliebige Operanden. `=` vergleicht dabei nur die beiden 64-bit-Wörter, die die Daten repräsentieren. Bei Listen werden also nur die Zeiger verglichen, nicht die Elemente; d.h., bei zwei Listen mit gleichem Inhalt, die nicht in der selben Operation gebaut wurden, wird `=` also 0 liefern (z.B. `(0.0)=(0.0)` liefert 0).

`isnum` erwartet beliebige Operanden und liefert 1, wenn der Operand eine Zahl ist, ansonsten 0; entsprechend `islist` für Listen und `isfun` für Closures (Closures kommen in diesem Beispiel nicht vor, aber `isfun` kann auch andere Operanden verarbeiten).

x`.`y baut eine Listenzelle, und zwar mit x als Kopf und y als Rest, und liefert als Resultat die Adresse der Zelle (entsprechend getagt). Bei Ausdrücken mit mehreren Punkten in Folge wird implizit wie folgt geklammert: `x.y.z` ist äquivalent zu `x.(y.z)`, `w.x.y.z` zu `w.(x.(y.z))` („Rechtsassoziativ“).

`head` erwartet eine Liste und liefert den Inhalt des Kopffeldes; entsprechend `tail` für das Restfeld.

Ein Lamda-Ausdruck produziert eine Funktion mit einem Parameter.

7.6.2.3 Typprüfung Einige Operatoren überprüfen zur Laufzeit, ob ihre Operanden einen bestimmten Typ haben. Wenn nicht, soll das Programm zur Laufzeit mit einem Signal abbrechen.

7.6.2.4 Funktionsresultat Eine Funktion gibt als Ergebnis den Wert der Expr zurück.

7.6.2.5 Erzeugter Code. Ihr Compiler soll AMD64-Assemblercode ausgeben. Jede Funktion im Programm verhält sich gemäß der Aufrufkonvention, abgesehen vom Heap-Pointer (siehe unten). Der erzeugte Code wird nach dem Assemblieren und Linken von C-Funktionen aufgerufen. Beispiel: Die Funktion `foo = fun a -> ...;` kann von C aus mit `foo(x)` aufgerufen werden, wobei `a` den Wert von `x` bekommt.

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

Die Listenzellen können Sie auf dem Heap anlegen. Der Heap-Zeiger befindet sich im Register r15 und wird schon von der aufrufenden Funktion passend übergeben (inkl. Ausrichtung auf gerade Adressen); der Heap wächst nach oben. Um die Freigabe der Elemente brauchen Sie sich nicht kümmern, es ist genug Platz für die Listenelemente, die in dem Ausdruck vorkommen.

Im Fall eines Typfehlers soll ein Signal erzeugt werden, das den Prozess beendet. Sie müssen das für jeden Operanden selbst abfragen (z.B. mit `xor`, `test` und `jne`). Ihr Programm kann ein Signal erzeugen, indem es zum Label `raisesig` springt, das Ihnen unser Testframework zur Verfügung stellt.

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): Die maximale Tiefe eines Ausdrucks[2] ist 5. Die im Quellprogramm vorkommenden Zahlen und konstanten Ausdrücke sind $\geq -2^{31}$ und $< 2^{31}$; das gilt aber nicht für Ergebnisse von Berechnungen zur Laufzeit.

Der erzeugte Code soll korrekt sein und möglichst wenige Befehle ausführen (da es hier abgesehen von Typchecks keine Verzweigungen gibt, ist das gleichbedeutend mit „wenige Befehle enthalten“). Dabei ist nicht an eine zusätzliche Optimierung (wie z.B. common subexpression elimination) gedacht, sondern vor allem an die Dinge, die Sie mit `iburg` tun können, also eine gute Codeauswahl (besonders bezüglich konstanter Operanden) und eventuell einige algebraische Optimierungen (siehe z.B. `http://www.complang.tuwien.ac.at/papers/ertl00dagstuhl.ps.gz`); insbesondere können Sie in diesem Beispiel die Tag-Checks und das Taggen optimieren, indem Sie für getagte und enttagte Werte Nonterminals in der Baumgrammatik einführen. Für besonders effizienten erzeugten Code gibt es Sonderpunkte.

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

7.6.3 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/codea`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `codea` erzeugen, das von der Standardeingabe liest und den generierten Code auf

[2]Tiefe eines Ausdrucks: Anzahl der Ableitungen von `Expr` zwischen einem Blatt des Syntaxbaums und dem nächsten `Statement`.

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 darf die Ausgabe beliebig sein.

7.7 Codeerzeugung B

7.7.1 Termin

Abgabe spätestens am 3.6.2015, 14 Uhr.

7.7.2 Angabe

Erweitern Sie den Compiler aus dem vorigen Beispiel so, dass er folgende Untermenge der statisch korrekten Programme in AMD64-Assemblercode übersetzt: Alle Programme, in denen *Funktionsaufruf*e nur als Tail-calls vorkommen, und in denen `Lambda` nur von Def und nicht von Expr abgeleitet wird. Ein Funktionsaufruf is ein Tail-call, wenn sein Ergebnis als Ergebnis der gesamten Funktion zurückgegeben wird; in unserem Fall also, wenn im Ableitungsbaum zwischen Def und dem Funktionsaufruf nur Lambda, `let`-Ausdrücke, und `if`-Ausdrücke vorkommen, wobei bei let und if der Funktionsaufruf nicht über das jeweils erste Expr abgeleitet werden darf. Programme, die statisch korrekt sind, aber dieser Einschränkung nicht entsprechen, werden bei diesem Beispiel nicht als Testeingaben vorkommen.

Ein Teil der Sprache wurde schon erklärt, hier der für dieses Beispiel notwendige Zusatz:

Ein `if`-Ausdruck wertet zunächst die erste Expr aus; wenn das Ergebnis etwas anderes ist als 0, wird die Expr aus dem `then`-Zweig ausgewertet und ihr Ergebnis ist das Ergebnis des `if`-Ausdrucks, ansonsten die Expr aus dem `else`-Zweig.

Ein `let`-Ausdruck wertet die erste Expr aus und bindet das Ergebnis an die Variable ident. Dann wird der zweite Ausdruck ausgewertet und sein Ergebnis ist das Ergebnis des `let`-Ausdrucks.

Ein Funktionsaufruf wertet Expr aus; das Ergebnis muß ein Lambda sein (in diesem Beispiel ist das nur der Fall, wenn Expr der Name einer Funktion ist, die in einem Def definiert wurde). Der Aufruf wertet auch Term aus und ruft die Funktion auf, wobei das Ergebnis des Terms als Argument übergeben wird. Der von der Funktion zurückgegebene Wert ist der Wert des Funktionsaufrufs.

7.7.2.1 Erzeugter Code. Es gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel.

Durch die Einschränkung auf Tail-Calls können Sie Funktionsaufrufe in diesem Beispiel als Sprung zur entsprechenden Funktion implementieren, wobei ggf. noch vorher Speicher auf dem Stack freigeben werden muss, wenn die aufrufende Funktion dort Speicher reserviert hat; die Rücksprungadresse dürfen Sie nicht entfernen.

7.7.3 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/codeb`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `codeb` 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 darf die Ausgabe beliebig sein.

7.8 Gesamtbeispiel

7.8.1 Termin

Abgabe spätestens am 17.6.2015, 14 Uhr.

Es gibt nur einen Nachtermin. Wenn Sie sich für ein Abschlussgespräch vor dem Nachtermin anmelden, wird für die Note nur das Ergebnis des ersten Abgabetermins berücksichtigt.

7.8.2 Angabe

Erweitern Sie den Compiler aus dem vorigen Beispiel so, dass er alle statisch korrekten Programme in AMD64-Assemblercode übersetzt.

Ein Teil der Sprache wurde schon erklärt, hier der für dieses Beispiel notwendige Zusatz:

Ein Lambda liefert eine Funktion mit einem Parameter, in der Variablen vorkommen können, die außerhalb des Lambda definiert wurden. Diese Funktionen-mit-Umgebung nennt man auch Closures. Sie können überall hin gereicht werden, auch an Stellen, wo die Variablen nicht sichtbar sind, und können von dort aufgerufen werden. Z.B.:

```
plusn = fun x -> fun y -> x+y end end;
plus3 = fun z -> (plusn 3) z end;
```

Wenn wir `plus3` mit dem Parameter 5 aufrufen, erzeugt der Aufruf von `plusn 3` eine Closure mit der Umgebung x=3, die dann wiederum mit dem Argument 5 aufgerufen wird, und letztendlich das Ergebnis 8 liefert.

Closures werden in unsere Testfällen nicht von ausserhalb des Programms hineingereicht oder als Rückgabewert des Aufrufs von aussen vorkommen.

7.8.2.1 Erzeugter Code. Die in Def direkt definierten Funktionen werden weiterhin entsprechend den Aufrufkonventionen aufgerufen.

Da Closures nur innerhalb des Programmes vorkommen, können Sie sich Ihre Implementierung aussuchen; der Aufruf einer Closure muss entsprechend dazu passen. Es folgt ein Vorschlag für den Aufbau der Closures:

Ausgangspunkt ist die Addressierung nicht-lokaler Variablen mit statischen Ketten (Abschnitt 10.3 im Vorlesungsskriptum). Da unsere Closures im Gegensatz zu den Prozeduren in Pascal als Rückgabewerte verwendet werden können und damit die Prozedur, in der die Variablen definiert werden, überleben können, muss zumindest ein Teil der Daten auf dem Heap gespeichert werden, und zwar die Variablen und der Zeiger auf den statischen Vorgänger. Die Rücksprungadresse wird nach der Rückkehr nicht mehr gebraucht und kann daher auf dem Stack bleiben; eventuell abzuspeichernde Zwischenergebnisse, z.B. bei einem verschachtelten Aufruf, kann man auch auf den Stack legen.

Klassischerweise wird eine Closure als Paar repräsentiert: Ein Zeiger auf den Anfang des Codes der Funktion, und ein Zeiger auf das Environment (also den statischen Pointer). Das würde aber zwei Maschinenworte brauchen und nicht in die Variablen passen, die wir bisher implementiert haben. Daher erzeugen wir auf dem Heap eine Struktur mit zwei Feldern für die Closure und repräsentieren die Closure mit einem getagten Zeiger auf diese Struktur (Dieses Verpacken in einer Struktur, auf die man mit einem (polymorphen) Zeiger zugreift, nennt man auch Boxing). Das Tag für Closure-Zeiger ist, wie schon früher beschrieben, 1 auf dem niederwertigsten Bit und 1 auf dem nächsten Bit. Den Anfang der Closure-Struktur erhält man, indem man vom getagten Zeiger 3 subtrahiert.

Abbildung 2 zeigt ein Beispiel, wie die Daten zu einem bestimmten Zeitpunkt der Laufzeit ausschauen könnten.

Es gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, wobei ein Lambda-Ausdruck bei der Berechnung der Tiefe mit dem Wert 1 eingeht; die Tiefe der Expr in einem Lambda-Ausdruck ist auch nach der Regel begrenzt, wird aber bei der Berechnung der Tiefe des Ausdrucks, der das Lambda enthält, nicht berücksichtigt; die Idee ist, dass jedes Lambda seine eigene Registerbelegung hat, unabhängig von der Registerbelegung des umgebenden Ausdrucks.

Bei diesem Beispiel gibt es keine Sonderpunkte für guten Code. Das Beispiel korrekt hinzukriegen ist ohnehin schwer genug.

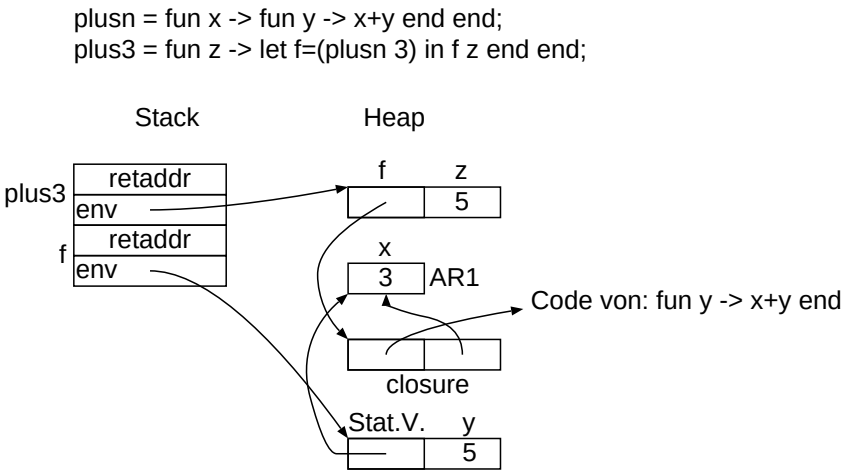

Abbildung 2: Stack und Environments (auf dem Heap) für den Aufruf `plus3 5` zu dem Zeitpunkt, wenn das `+` durchgeführt wird. Der Stack wächst nach unten. Der Aufruf von plus3 legt einen Activation Record mit den Elementen f und z auf den Heap. Ein Zeiger auf den statischen Vorgänger ist hier nicht gezeigt, da plus3 keine umgebende Funktion hat und daher auch keinen statischen Vorgänger braucht (es kann aber u.U. die Implementierung vereinfachen, wenn man ein Feld dafür trotzdem vorsieht). Vor dem gezeigten Zeitpunkt erfolgt der Aufruf `plusn 3`, der den Activation Record AR1 mit dem Element x auf dem Heap anlegt, und als Rückgabewert eine Closure für die Funktion `fun y -> x+y end` mit dem von AR1 beschriebenen Environment zurückgibt. Nach der Rückkehr von plusn (das wir daher nicht mehr auf dem Stack sehen) wird diese Closure aufgerufen und dabei wird ein Activation Record mit dem Element y und mit AR1 als statischem Vorgänger angelegt. Auf dem Stack sind zu dem Zeitpunkt Daten von zwei Aufrufen zu sehen: von plus3 und f (der von `plusn 3` erzeugten Closure). Für jeden Aufruf ist eine Rücksprungadresse zu sehen, damit am Schluss zum Aufrufer zurückgekehrt werden kann, und ein Zeiger auf den Activation Record, damit man den nach der Rückkehr wieder findet. Dynamische Vorgänger werden hier nicht gespeichert, da die Daten auf dem Stack eine dem Compiler bekannte Größe haben.

7.8.3 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 2, bei anderen Fehlern 3.