Übersetzerbau VL
Skriptum

Anton Ertl
Andreas Krall
Markus Schordan

2007

- Vorlesungsskriptum
- Übung: 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/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 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/ubvl/newsgroup.html` werden verschiedene Zugriffsmöglichkeiten erläutert. Stellen Sie Fragen, deren Antwort vermutlich auch andere Übungsteilnehmer interessiert, 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/ubvl/` Informationen zur Übung.

Während der Übung stehen zu gewissen Zeiten in den Übungsräumen Tutoren bereit (siehe `http://www.complang.tuwien.ac.at/ubvl/stundenplan.txt`). Wenn die Tutoren Ihre Frage nicht beantworten können, erreichen Sie unter anton@mips.complang.tuwien.ac.at einen betreuenden Assistenten. Wenn Sie Ihre Fragen gerne persönlich stellen, kommen Sie am Montag von 10h-11h zu Anton Ertl oder Markus Schordan 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“ bis „Gesamtbeispiel“. 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/ubvl/`).

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

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 lesen, indem sie in Emacs `C-h i` tippen. In der Dokumentation für Emacs bedeutet `C-`*x* $\boxed{\text{Ctrl}}x$ und `M-`*x* $\boxed{\text{Meta}}x$ (auf den Übungsgeräten also $\boxed{\text{Alt}}x$).

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). Mit `xvcg` können Sie sich die Ausgabe von `bison -g` anschauen.

`mail` ist ein primitives Email-Werkzeug, `elm` und `mutt` sind etwas beque-

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
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.
elm, mutt, mail	man elm, man mutt, man mail	Email
xrn	man xrn	Newsreader
lynx, mozilla firefox		WWW-Browser

Abbildung 1: Werkzeuge

mer[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 acht Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen einiger grundlegender Befehle der Alpha-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. 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 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 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, z.B. Lesen von einer nicht initialisierten Variable. 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.

6.1 Assembler A

6.1.1 Termin

Abgabe spätestens am 14. März 2007, 14 Uhr.

6.1.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned long asma(unsigned long l)
{
  unsigned long r;
  unsigned char *s = (unsigned char *)&l;
  unsigned char *t = (unsigned char *)&r;
  int i;
  for (i=0; i<sizeof(l); i++) {
    unsigned char c=s[i];
    c += (c>='A' && c<='Z') ? 'a'-'A' : 0;
    t[i] = c;
  }
  return r;
}
```

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

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

```
unsigned 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.

6.1.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version `cmpbge` und `zap` bzw. `zapnot` 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 21. März 2007, 14 Uhr.

6.2.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned char *asmb(unsigned char *s, unsigned long n)
{
  unsigned long i;
  for (i=0; i<n; i++) {
    unsigned char c=s[i];
    c += (c>='A' && c<='Z') ? 'a'-'A' : 0;
    s[i] = c;
  }
  return s;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `cmpbge` und `zap` oder `zapnot`. Sie dürfen dabei annehmen, dass hinter dem letzten Zeichen von s noch 8 Bytes 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` und `ustq` werden 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 weniger als mehr.

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 28. März 2007, 14 Uhr.

6.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `func end goto if return not and`. Weiters soll er auch noch folgende Lexeme erkennen: `; ( ) : = - + * < [ ] { }`.

Identifier bestehen aus Buchstaben und Ziffern, dürfen aber nur mit Buchstaben 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 Zeilenende 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 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 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 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).

6.3.4 Hinweis

Beachten Sie, neben `$` (ein Zeilenende, auf das ein Newline folgt) auch noch das Ende der Eingabe die Zeile beenden kann. 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.

6.4 Parser

6.4.1 Termin

Abgabe spätestens am 18. April 2007, 14 Uhr.

6.4.2 Angabe

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

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

Funcdef: func id '{' Ids '}' Stats end
       ;

Ids: { id }      /* Parameterdefinition */
   ;

Stats: { { Label } Stat ';' }
     ;

Label: id ':'     /* Labeldefinition */
     ;
```

```
Stat: Lexpr '=' Expr   /* Zuweisung */
    | goto id
    | if Expr goto id
    | Call
    | return Expr
    ;

Lexpr: id       /* schreibender Variablenzugriff */
     | Array    /* schreibender Arrayzugriff */
     ;

Expr: [ not ] Term
    | Term '-' Term
    | Term { '+' Term }
    | Term { '*' Term }
    | Term { and Term }
    | Term ( '<' | '=' ) Term
    ;

Term: '(' Expr ')'
    | id       /* lesender Variablenzugriff */
    | num
    | Call
    | Array    /* lesender Arrayzugriff */
    ;

Array: Term '[' Expr ']'
     ;

Call: id '{' { Term } '}'  /* Funktionsaufruf */
    ;
```

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, aber nicht unbedingt in der gewünschten Art.

6.5 Attributierte Grammatik

6.5.1 Termin

Abgabe spätestens am 2. Mai 2007, 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*), Labels (in der *Labeldefinition* und hinter `goto`) und Variablennamen (in der *Parameterdefinition* und in Variablenzugriffen). Diese drei Arten von Namen sind in verschiedenen Namensräumen, es kann also in einer Funktion ein Label, eine Variable, und eine Funktion vorkommen, die alle den selben Namen haben.

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

In der Parameterdefinition darf jeder Name nur einmal vorkommen. Ansonsten dürfen Variablennamen beliebig verwendet werden, soweit es die statische Analyse betrifft. Variablen müssen also nicht speziell definiert werden, um verwendet werden zu dürfen; es soll auch keine Überprüfung stattfinden, ob eine Variable geschrieben wird, bevor sie gelesen wird (allgemein ist das für diese Sprache auch ein unentscheidbares Problem).

Ein Label wird in einer Labeldefinition definiert und ist in der gesamten Funktion sichtbar (also auch vor der Definition), aber nicht darüber hinaus. In einer Funktion darf nur ein Label mit dem gleichen Namen definiert werden.

Der Identifier hinter einem `goto` muss ein definierter und sichtbarer Label sein.

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 Codeerzeugung A

6.6.1 Termin

Abgabe spätestens am 16. Mai 2007, 14 Uhr.

6.6.2 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 Alpha-Assemblercode übersetzt: alle Programme, in denen aus `Stats` nur genau ein `return`-Statement und keine anderen Statements abgeleitet wird, und in denen `Call` nicht abgeleitet wird. Programme, die statisch korrekt sind, aber dieser Einschränkung nicht entsprechen, werden bei diesem Beispiel nicht als Testeingaben vorkommen.

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

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 soll 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 (`Expr`s) 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 Operatoren `and` und `not` führen die Operation bitweise auf ihren Operanden durch. Die Operatoren `<` und `=` vergleichen ihre Operanden als vorzeichenbehaftete ganze Zahlen und liefern 1 für wahr und 0 für falsch.

Bei einem `Array` ist `Term` die Anfangsadresse des Arrays und `Expr` der Index, der Zugriff erfolgt auf die Adresse Term+8*Expr; z.B. greift `a[1]` auf die gleiche Adresse zu wie `(a+8)[0]`. Der lesende Arrayzugriff liefert als Resultat den 64-bit-Ausdruck an dieser Adresse.

Die `return`-Anweisung beendet die Funktion und liefert das Resultat von `Expr` als Ergebnis des Aufrufs der Funktion.

6.6.2.1 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 aufgerufen.

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.

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 maximale Tiefe eines Ausdrucks ist ≤ 6 (Tiefe eines Ausdrucks: Anzahl der Ableitungen von `Expr` zwischen einem Blatt des Syntaxbaums und dem nächsten `Stat`). 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`).

Der erzeugte Code soll korrekt sein und möglichst wenige Befehle ausführen (da es hier keine Verzweigungen gibt, ist das gleichbedeutend mit „wenig 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`). 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.

6.6.3 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 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.

6.7 Codeerzeugung B

6.7.1 Termin

Abgabe spätestens am 30. Mai 2007, 14 Uhr.

6.7.2 Angabe

Erweitern Sie den Compiler aus dem vorigen Beispiel so, dass er folgende Untermenge der statisch korrekten Programme in Alpha-Assemblercode übersetzt: Alle Programme, in denen der Parser `Call` nicht ableitet. 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:

Bei einer *Zuweisung* wird der Wert der `Expr` in die Variable bzw. in die vom `Array` bestimmte Adresse geschrieben.

Eine `goto`-Anweisung springt zum angegebenen Label.

Eine `if`-Anweisung wertet die `Expr` aus. Ist das niedrigste Bit des Ergebnisses gesetzt, erfolgt ein Sprung zum angegebenen Label, ansonsten passiert nichts weiter.

Kommt die Ausführung zum Ende der Funktion, ist das Ergebnis undefiniert (der erzeugte Code darf dann also beliebiges machen; der Compiler muss nicht überprüfen, ob das Programm vorher ein `return` ausführt).

6.7.2.1 Erzeugter Code. Es gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, und zusätzlich: Jede Funktion hat maximal 6 Variablen.

6.7.3 Hinweis

Es bringt nichts, für `iburg` Bäume zu bauen, die mehr als ein Statement umfassen: die Möglichkeit, durch die Baumgrammatik Knoten zusammenzufassen und so zu optimieren, kann nur auf Statement-Ebene und darunter genutzt werden. Es ist einfacher, für jedes Statement einen Baum zu bauen und dann in einem Traversal für jeden dieser Bäume den Labeler und den Reducer aufzurufen.

6.7.4 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.

6.8 Gesamtbeispiel

6.8.1 Termin

Abgabe spätestens am 13. Juni 2007, 14 Uhr.

6.8.2 Angabe

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

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

Der *Funktionsaufruf* wertet alle `Term`e aus und ruft dann die Funktion `id` auf, mit den Ergebnissen der Terme als Parameter. Wenn der Funktionsaufruf eine Anweisung ist, wird der Rückgabewert ignoriert. Wenn der Funktionsaufruf ein `Term` ist, wird der Rückgabewert zum Wert des Terms.

6.8.2.1 Erzeugter Code. Der erzeugte Code ruft Funktionen entsprechend den Alpha-Aufrufkonventionen auf. Ansonsten gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, wobei ein `Call` mit n Parametern bei der Berechnung der Tiefe mit dem Wert $\max(0, n-1)$ eingeht.

Wichtigstes Kriterium ist wie immer die Korrektheit, für gute Codeerzeugung gibt es aber wieder Sonderpunkte. Wir empfehlen, nur Optimierungen durchzuführen, die mit den verwendeten Werkzeugen einfach möglich sind. Bei diesem Beispiel kommt es mehr auf gute Registerbelegung an als auf die Optimierung von Ausdrücken.

6.8.3 Hinweise

Bei der Registerbelegung gibt es sowohl ein großes Optimierungspotential als auch ein großes Fehlerpotential, besonders im Zusammenhang mit (verschachtelten) Funktionsaufrufen.

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.

6.8.4 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.