Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2020

- 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 TISS für die Lehrveranstaltung an. Nach Ende der Anmeldefrist (siehe Homepage) wird ein Account für Sie auf unserer Übungsmaschine `g0.complang.tuwien.ac.at` eingerichtet, der Accountname ist `u` gefolgt von der Matrikelnummer, z.B. u99999999. Ein Passwort für diesen Account erhalten Sie per Email. Bitte ändern Sie das Passwort möglichst bald, das zugesandte wird aus Sicherheitsgründen nach kurzer Zeit gesperrt.

2 Rechner

Die Übungsmaschine ist `g0.complang.tuwien.ac.at`; sollte sie längerfristig ausfallen, steht als Ersatzmaschine `g2.complang.tuwien.ac.at` zur Verfügung (Sie können sich aber vorerst nicht auf die Ersatzmaschine einloggen). Sie können sich aus dem Internet mit `ssh g0.complang.tuwien.ac.at` einloggen; falls Sie sich von einem Windows-Client aus einloggen wollen, können Sie das mit Putty tun.

Falls Sie Ihre Programme woanders entwickeln, müssen Sie Ihre Dateien für die Abgabe mit `scp` (eine ssh-Anwendung) auf unsere Rechner übertragen. Rufen Sie dann unbedingt das Test-Skript für das Beispiel *auf der Übungsmaschine* auf, damit Sie eventuelle Fehler mit katastrophalen Auswirkungen bemerken (wie z.B. das Kopieren in das falsche Verzeichnis).

Die in der Übung verwendeten Werkzeuge sind für verschiedene Plattformen auf `http://www.complang.tuwien.ac.at/ubvl/tools/` erhältlich (allerdings recht veraltet).

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).

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 Informatikforum (Details dazu siehe Übungshomepage).

Wenn Sie eine Frage zur Übung haben, stellen Sie sie am besten im Informatikforum (dann können auch andere antworten oder von der Antwort profitieren). Sie können auch den Leiter der Übung per Email fragen `anton@mips.complang.tuwien.ac.at`, oder in einer Sprechstunde (Termin per Email vereinbaren).

Technische Probleme wie Computerabstürze, falsche Permissions, oder vergessene Passwörter sind eine Sache für den Techniker. Wenden Sie sich direkt an ihn: email an Herbert Pohlai (`herbert@mips.complang.tuwien.ac.at`), Tel. (+43-1) 58801/18525.

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.

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
dotty	man dotty	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
lynx, firefox		WWW-Browser

Abbildung 1: Werkzeuge

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 $\boxed{\text{Ctrl}}x$ 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 `bison -y` (für den Fall, dass Sie Diskrepanzen zwischen diesem yacc und dem auf kommerziellen Unices bemerken). Mit `dotty` können Sie sich die Ausgabe von `bison -g` anschauen.

`mail` ist ein primitives Email-Werkzeug, `mutt` ist 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.

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. So sind in dieser Sprache zwar grundlegende Kontrollstrukturen vorhanden und es können Variablen definiert werden, aber es gibt z.B. keine Ein- und Ausgabe. Die fehlenden Elemente werden dadurch ergänzt, dass Programmteile in dieser Programmiersprache mit in C geschriebenen Programmteilen zusammengelinkt werden, die die fehlenden Funktionen durchführen. Dadurch erlernen Sie auch, 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.

---

[1] `mutt` ist 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` wird weitermachen.

In dieser Sprache kann man, wie in manchen anderen 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. März 2020, 14 Uhr.

7.1.2 Angabe

Gegeben ist folgende C-Funktion, die in einem 16-Zeichen-Array ASCII-Kleinbuchstaben in Großbuchstaben umwandelt:

```
unsigned char *asma(unsigned char *s)
{
  int i;
  for (i=0; i<16; 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 `pcmpgtb`. Dabei ist folgende Äquivalenz hilfreich:

```
(c>='a' && c<='z') ? 'A'-'a' : 0;
```

ist (bei Verwendung von Überlauf-Arithmetik) äquivalent zu

```
min('z'+1+min_t-'a' > c+min_t-'a' ? 0xff : 0, 'A'-'a')
```

wobei `min_t` der minimale Wert des Datentyps ist, den der Vergleich behandelt (bei `pcmpgtb` also -128). Zusätzlich zu dem oben genannten dürften die Befehle `pminub`, `paddb`, und `psubb` nützlich sein.

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

```
unsigned char *asma(unsigned char *s)
{
  return s;
}
```

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.

Am einfachsten tun Sie sich dabei wahrscheinlich, wenn Sie eine vereinfachte Variante der ursprünglichen Funktion (z.B. nur mit dem then-Zweig) mit z.B. `gcc -O -S` in Assembler übersetzen und sie dann verändern. Dann stimmt schon das ganze Drumherum.

7.1.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version korrekt ist.

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. März 2020, 14 Uhr.

7.2.2 Angabe

Gegeben ist folgende C-Funktion:

```
#include <string.h>
unsigned char *asmb(unsigned char *s)
{
  unsigned long i;
```

```
  for (i=0; s[i]; 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 `pcmpgtb`. Neben den schon genannten können Ihnen die Befehle `pcmpeqb`, und `pmovmskb` dabei helfen. Sie dürfen dabei annehmen, dass hinter dem letzten Zeichen von s noch 16 Bytes zugreifbar sind, und Sie dürfen bis zu 15 Zeichen hinter dem Ende von s beliebig verändern.

Für besonders effiziente Lösungen (gemessen an der Anzahl der *ausgeführten* Maschinenbefehle; wird ein Befehl n mal ausgeführt, zählt er n-fach) gibt es Bonuspunkte.

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 1. April 2020, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `end return if then else loop break cont var not and`. Weiters soll er auch noch folgende Lexeme erkennen: `( ) , ; : := * - + <= #`

Identifier bestehen aus Buchstaben, und Ziffern, und `_`, dürfen aber nicht mit Ziffern beginnen. Zahlen sind entweder Hexadezimalzahlen oder Dezimalzahlen. Hexadezimalzahlen beginnen mit `0x`, gefolgt von einer oder mehr Hexadezimalziffern, wobei Hex-Ziffern sowohl groß als auch klein geschrieben sein dürfen. Dezimalzahlen bestehen aus einer oder mehr Dezimalziffern. 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, `end39` ist also ein Identifier (longest input match), `39end` ist die Zahl `39` gefolgt vom Schlüsselwort `end`.

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 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, `End` ist also kein Schlüsselwort.

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. April 2020, 14 Uhr.

7.4.2 Angabe

Abbildung 2 zeigt die Grammatik (in `yacc/bison`-artiger EBNF) einer Programmiersprache. 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`.

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 beabsichtigten Art.

Links- oder Rechtsrekursion? Also: Soll das rekursive Vorkommen eines Nonterminals 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 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.

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

Funcdef: id '(' Pars ')' Stats end  /* Funktionsdefinition */
       ;

Pars: { id ',' } [ id ]     /* Parameterdefinition */
    ;

Stats: { Stat ';' }
     ;

Stat: return Expr
    | if Expr then Stats [ else Stats ] end
    | id ':' loop Stats end  /* Schleife */
    | break id
    | cont id
    | var id ':=' Expr  /* Variablendefinition */
    | Lexpr ':=' Expr   /* Zuweisung */
    | Expr              /* Ausdrucksanweisung */
    ;

Lexpr: id            /* Schreibender Variablenzugriff */
     | '*' Term      /* Schreibender Speicherzugriff */
     ;

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

Term: '(' Expr ')'
    | num
    | id                     /* lesender Variablenzugriff */
    | id '(' { Expr ',' } [ Expr ] ')'  /* Funktionsaufruf */
    ;
```

Abbildung 2: Grammatik

7.5 Attributierte Grammatik

7.5.1 Termin

Abgabe spätestens am 6. Mai 2020, 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.

7.5.2.1 Namen. Die folgenden Dinge haben Namen: Funktionen, Variablen, und Labels.

An einer Stelle darf nur eine Variable oder ein Label mit dem selben Namen sichtbar sein.

Alle Namen (`id`s), die in einer *Parameterdefinition* oder in einer *Variablendefinition* vorkommen, sind Variablennamen. Variablen, die in einer Parameterdefinition definiert wurden, sind in der ganzen Funktion sichtbar. Variablen, die in einer Variablendefinition definiert wurden, sind in allen folgenden Statements der unmittelbar umgebenden `Stats` sichtbar, und nirgendwo sonst. In der Definition selbst ist die Variable noch nicht sichtbar.

Bei einem *Variablenzugriff* muss eine Variable mit dem Namen sichtbar sein.

Die `id` am Anfang einer *Schleife* ist ein Label. Ein Label ist in der gesamten Schleife sichtbar, an deren Anfang es steht. Labels werden in `break`- und `cont`-Anweisungen verwendet, und sie müssen dort sichtbar sein.

Eine Funktion wird im *Funktionsaufruf* verwendet und in der *Funktionsdefinition* definiert. Verwendete Funktionen müssen nicht definiert werden und können nicht deklariert[2] werden. Funktionen dürfen, soweit es den Compiler betrifft, doppelt definiert werden und dürfen den gleichen Namen wie Variablen oder Labels haben; daher muss der Compiler Funktionsnamen nicht in einer Symboltabelle verwalten. Auch die Übereinstimmung der Anzahl der Argumente soll (und, im allgemeinen, kann) der Compiler nicht überprüfen.

Ihre statische Analyse soll überprüfen, dass alle verwendeten Variablen- und Label-Namen sichtbar sind und der richtigen Klasse (Label oder Variable) angehören, und dass zwei Definitionen des gleichen Label- oder Variablen-Namens keine überlappenden Sichtbarkeitsbereiche haben (auch nicht mit einem Namen der anderen Klasse).

[2] Im Sinne von C: Die Definition einer Funktion enthält den vollständigen Code. Die Deklaration enthält nur die Informationen, die der Compiler braucht, um eine Typüberprüfung des Aufrufs durchzuführen (in C auch bekannt als Prototyp, in anderen Sprachen oft als Signatur).

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. Heuer sind diese Möglichkeiten zwar für das AG-Beispiel wohl nicht nötig, aber behalten Sie sie für spätere Beispiele im Hinterkopf.

Verwenden Sie keine globalen Variablen oder Funktionen mit Seiteneffekten (z.B. Funktionen, die übergebene Datenstrukturen ändern) bei der Attributberechnung! `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. Mai 2020, 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 aus `Stat` nur `return`-Anweisungen abgeleitet werden, in denen aber kein *Funktionsaufruf* 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. Diese Programmiersprache kennt nur einen Datentyp: das 64-bit-Wort, das als vorzeichenbehaftete Zahl oder als Speicheradresse verwendet werden kann. Weder der Compiler noch das Laufzeitsystem soll eine Typüberprüfung vornehmen. Der Programmierer (der Anwender des Compilers) muss wissen, was er tut, der Compiler soll (und kann) das nicht überprüfen. Unsere Testprogramme führen keine Zugriffe auf ungültige Adressen aus.

7.6.2.2 Bedeutung der Operatoren. `+`, `-` und das binäre `*` haben ihre übliche Bedeutung (ein etwaiger Überlauf soll ignoriert werden).

Das unäre `*` betrachtet den Term als Adresse und liest einen 64-bit-Wert von dieser Adresse (bzw. schreibt beim *schreibenden Speicherzugriff* den Wert an diese Adresse).

`not` und `and` führen ihre Operationen bitweise auf ihre Operanden aus.

`<=` und `#` (ungleich) liefern -1 für wahr und 0 für falsch.

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

7.6.2.4 Erzeugter Code. Ihr Compiler soll AMD64-Assemblercode ausgeben. Jede Funktion im Programm verhält sich gemäß der Aufrufkonvention. Der erzeugte Code wird nach dem Assemblieren und Linken von C-Funktionen aufgerufen. Beispiel: Die Funktion `foo(a,b) ... end;` kann von C aus mit `foo(x,y)` aufgerufen werden, wobei `a` den Wert von `x` bekommt und `b` den von `y`.

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[3] ist $\leq 6 - v$, wobei v die Anzahl der sichtbaren Variablen ist. Die im Quellprogramm vorkommenden Zahlen und konstanten Ausdrücke sind $\geq -2^{30}$ und $< 2^{30}$; das gilt aber nicht für Ergebnisse von Berechnungen zur Laufzeit.

Der erzeugte Code soll korrekt sein und möglichst wenige Befehle ausführen (da es hier 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`). 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

[3]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. Juni 2020, 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 der Parser keinen *Funktionsaufruf* 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:

Die *Variablendefinition* speichert den Wert von `Expr` unter dem Namen der Variable.

Bei einer *Zuweisung* wird `Expr` ausgewertet und das Resultat in die Variable (bei einem *schreibenden Variablenzugriff*) oder in die Speicherstelle (*schreibender Speicherzugriff*) geschrieben.

Eine *Ausdrucks-Anweisung* wertet den Term aus und macht mit dem Ergebnis nichts (in diesem Beispiel gibt es keine Funktionsaufrufe, daher macht diese Anweisung hier gar nichts).

Eine `if`-Anweisung wertet `Expr` aus. Ist das Ergebnis ungerade, wird der then-Zweig ausgeführt, ansonsten der `else`-Zweig bzw. (falls es keinen `else`-Zweig gibt) nichts.

Die *Schleife* führt die enthaltenen Anweisungen aus, und beginnt dann von vorne.

Die `break`-Anweisung springt unmittelbar hinter das Ende der genannten Schleife.

Die `cont`-Anweisung springt an den Anfang der genannten Schleife.

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).

7.7.2.1 Erzeugter Code.

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

7.7.3 Hinweis

Es bringt nichts, für `iburg` Bäume zu bauen, die mehr als eine einfache Anweisung oder den Teil einer `if`-Anweisung bis zum bedingten Sprung umfassen: die Möglichkeit, durch die Baumgrammatik Knoten zusammenzufassen und so zu optimieren, kann nur auf der Ebene von Ausdrücken und einfachen Anweisungen genutzt werden (ausser man würde die Zwischendarstellung in einer Weise umformen, die zuviel Aufwand für diese LVA ist).

Auf höherer Ebene ist einfacher, für jede einfache Anweisung einen Baum zu bauen und dann in einem Traversal für jeden dieser Bäume den Labeler und den Reducer aufzurufen.

7.7.4 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.

7.8 Gesamtbeispiel

7.8.1 Termin

Abgabe spätestens am 17. Juni 2020, 14 Uhr.

Es gibt nur einen Nachtermin.

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:

Der *Funktionsaufruf* wertet alle `Expr`s aus und ruft dann die Funktion `id` auf, mit den Ergebnissen der `Expr`s als Parameter. Der von der Funktion zurückgegebene Wert ist der Wert des Funktionsaufrufs.

7.8.2.1 Erzeugter Code. Der erzeugte Code ruft Funktionen entsprechend den Aufrufkonventionen auf. Ansonsten gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, wobei ein Funktionsaufruf mit n Parametern bei der Berechnung der Tiefe mit dem Wert

$\max(0, n-1)$ (zuzüglich der maximalen Tiefe der Berechnungen der Parameter) 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.

7.8.3 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. auf den Stack zu kopieren, damit man beim Berechnen der Parameter einer anderen Funktion problemlos auf sie zugreifen kann. Diese Strategie mag zwar nicht zum optimalen Code führen, aber eine gute Regel beim Programmieren lautet: "First make it work, then make it fast".

7.8.4 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. Der ausgegebene Code muss vom Assembler verarbeitet werden können.