Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2021

- 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 17. März 2021, 14 Uhr.

7.1.2 Angabe

Gegeben ist folgende C-Funktion:

```
void asma(unsigned long x[], unsigned long y[], unsigned long sum[])
{
  unsigned long carry, sum0;
  sum0 = x[0]+y[0];
  carry = sum0<x[0];
  sum[0] = sum0;
  sum[1] = x[1]+y[1]+carry;
}
```

Schreiben Sie diese Funktion in Assembler unter Verwendung von `adcq` 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 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 24. März 2021, 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 sum[], size_t n)
{
  unsigned long carry, s;
  size_t i;
  carry = 0;
  for (i=0; i<n; i++) {
    s = x[i]+y[i]+carry;
    carry = carry ? s<=x[i] : s<x[i];
    sum[i] = s;
  }
  sum[i] = carry;
}
```

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

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 14. April 2021, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `interface end class implements var method int return if then else while do not and new this null`. Weiters soll er auch noch folgende Lexeme erkennen: `; : ( ) . , := + * - < =`

Identifier bestehen aus Buchstaben und Ziffern, dürfen aber nicht mit Ziffern beginnen. Zahlen sind entweder Hexadezimalzahlen oder Dezimalzahlen. Hexadezimalzahlen beginnen mit einer Dezimalziffer, gefolgt von null oder mehr Hexadezimalziffern (wobei alphabetische Hexadezimal-Ziffern sowohl groß als auch klein geschrieben sein dürfen), gefolgt von `H`. 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 mit dem nächsten `*/` enden; Kommentare können also nicht geschachtelt werden. 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).

7.4 Parser

7.4.1 Termin

Abgabe spätestens am 21. April 2021, 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.

```
Program: { ( Interface | Class ) ';' }
       ;

Interface: interface id ':' /* Interfacedefinition */
            { id '(' [ { Type ',' } Type ] ')' ':' Type } /* abstrakte Methode */
           end
         ;

Class: class id /* Klassendefinition */
       implements { id } ':' /* Interfaceverwendung */
         { Member ';' }
       end
     ;

Member: var id ':' Type  /* Objektvariablendefinition */
      | method id '(' Pars ')' Stats end /* Methodenimplementierung */
      ;

Type: int
    | id /* Interfaceverwendung */
    ;

Pars: [ { Par ',' } Par ]
    ;

Par: id ':' Type  /* Parameterdefinition */
   ;

Stats: { Stat ';' }
     ;

Stat: return Expr
    | if Expr then Stats [ else Stats ] end
    | while Expr do Stats end
    | var id ':' Type ':=' Expr /* Variablendefinition */
    | id ':=' Expr /* Zuweisung */
    | Expr         /* Ausdrucksanweisung */
    ;

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

Term: '(' Expr ')'
    | num
    | this
    | null id
    | id /* lesender Zugriff */
    | Term '.' id '(' [ { Expr ',' } Expr ] ')' /* Aufruf */
    ;
```

Abbildung 2: Grammatik

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.

7.5 Attributierte Grammatik

7.5.1 Termin

Abgabe spätestens am 5. Mai 2021, 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 Typen und Nichtüberprüfungen. Die Syntax dieser Sprache ist zwar auf Typüberprüfung ausgelegt, aber um die Lösung der Aufgabe nicht noch komplizierter zu machen, müssen Sie keine Typüberprüfung implementieren. Die Testeingaben, die keine Fehler enthalten, die Ihr Compiler melden soll, enthalten auch keine Typfehler, aber Sie sollten trotzdem keine Typüberprüfung einbauen (zumindest keine, die in einem `exit` mündet): diese Angabe enthält keine Typüberprüfungsregeln und es kommt möglicherweise doch zu falsch-positiven Resultaten, wenn Sie selbstausgedachte Typregeln überprüfen.

Genausowenig sollen Sie die Übereinstimmung der Parameteranzahlen zwischen abstrakten Methoden, Methodenimplementierung, und Aufrufen überprüfen.

Weiters soll Ihr Compiler nicht überprüfen, ob eine Klasse Implementierungen für alle abstrakten Methoden bereitstellt, die in den Interfaces enthalten sind, die die Klasse implementiert.

Sie müssen allerdings die Typnamen entsprechend der unten angegebenen Regeln überprüfen.

7.5.2.2 Namen. Die folgenden Dinge haben Namen: Interfaces, Klassen, abstrakte Methoden, Objektvariablen, Parameter und Variablen. Sie alle teilen einen Namensraum. In diesem Abschnitt beziehen sich die *hervorgehobenen* Begriffe auf den Namen, der durch `id` in der Zeile des Kommentars repräsentiert wird.

Eine *Interfacedefinition* definiert einen Interface-Namen. Eine *Klassendefinition* definiert einen Klassennamen. Eine *abstrakte Methode* definiert den Namen einer abstrakten Methode. Eine *Objektvariablendefinition* definiert den Namen einer Objektvariable. Eine *Parameterdefinition* definiert den Namen eines Parameters. Eine *Variablendefinition* definiert den Namen einer Variable.

An einer Stelle darf nur ein Ding mit demselben Namen sichtbar sein (selbst wenn sie von verschiedener Art sind).

Interfaces, Klassen, und abstrakte Methoden sind von ihrer Definition an bis zum Ende des Programms sichtbar. Objektvariablen sind von ihrer Definition bis zum Ende der Klasse sichtbar. Die Sichtbarkeit von Parametern erstreckt sich über die gesamte Methode. Die Sichtbarkeit einer Variable beginnt hinter der Variablendefinition und geht bis zum Ende des `Stats`, das die Variablendefinition unmittelbar enthält.

Bei der Verwendung eines Namens muss dieser Name sichtbar sein und natürlich von der richtigen Art sein: Bei einer Interfaceverwendung darf man nicht einen Klassennamen verwenden und umgekehrt.

Interfacenamen werden in *Interfaceverwendung*en (2 Vorkommen in der Grammatik) und im `null`-Term verwendet.

Der `new`-Ausdruck verwendet einen Klassennamen.

Abstrakte Methoden werden in der *Methodenimplementierung* und im *Aufruf* verwendet.

Die *Zuweisung* und der *lesende Zugriff* sind Verwendungen des Namens einer Objektvariable, eines Parameters oder einer Variable (alle drei Arten von Namen sind erlaubt).

Ihre statische Analyse soll überprüfen, dass alle verwendeten Namen sichtbar und von der richtigen Art sind, und dass zwei Definitionen des gleichen Namens (auch verschiedener Arten) keine überlappenden Sichtbarkeitsbereiche haben.

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 19. Mai 2021, 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 *Aufruf* und kein `new`-Ausdruck 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 Klassen. Eine Klasse wird durch eine statische Tabelle aller abstrakter Methoden repräsentiert. Für jede abstrakte Methode im Programm (also über alle Interfaces hinweg, auch für Methoden von Interfaces, die die Klasse gar nicht implementiert) sind dort 8 Bytes vorgesehen; die erste abstrakte Methode hat den Offset 0, die zweite den Offset 8, usw. Für nicht implementierte Methoden ist 0 in der Tabelle gespeichert, für implementierte Methoden die Adresse des Einsprungspunkts der Methode.[2]

Da in unseren korrekten Testprogrammen nur implementierte Methoden aufgerufen werden, müssen Sie die anderen Einträge nicht mit 0 belegen, ja, sie müssen sie nicht einmal anlegen. Es kann aber hilfreich sein, um Fehler in den Testprogrammen zu erkennen.

Für eine Klasse gibt es einen extern sichtbaren Label mit dem Namen der Klasse, dessen Wert die Anfangsadresse der Tabelle ist.

In diesem Beispiel brauchen wir die Klassenrepräsentation nur, um darüber von außen eine Methodenimplementation aufrufen zu können. Im Gesamtbeispiel wird sie auch für die Implementierung des *Aufruf*s verwendet.

[2]Diese Darstellung zeichnet sich durch ihre Einfachkeit aus. Für große objektorientierte Programme braucht diese Darstellung relativ viel Speicher, und es gibt reichlich Literatur über speichereffizientere Implementierungen von Klassen und method dispatch.

7.6.2.2 Objekte. Ein Objekt enthält zunächst die 8 Byte grosse Adresse der Klassentabelle (für die Klasse des Objekts); darauf folgen die Objektvariablen, auch jeweils 8 Bytes groß. Die erste Objektvariable hat also Offset 8 vom Anfang des Objekts, die zweite 16, usw.

7.6.2.3 Daten (Parameter, Objektvariablen, Ausdrücke). Diese Programmiersprache kennt vorzeichenbehaftete 64-bit-Zahlen und Referenzen auf Objekte; eine Referenz ist die Anfangsadresse des Objekts. Ein `null`-Term liefert eine Referenz, die kompatibel mit Variablen vom angegebenen Typ ist, aber kein Objekt referenziert; Diese Null-Referenz wird durch 0 repräsentiert.

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 Aufrufe auf Null-Referenzen oder ganze Zahlen aus.

`this` ist die Referenz auf das aktuelle Objekt; der Aufrufer der aktuellen Methode hat das aktuelle Objekt festgelegt.

Lesende Zugriffe auf Parameter liefern den Wert des Parameters. *Lesende Zugriffe* auf Objektvariablen liefern den Wert dieser Objektvariablen des aktuellen Objekts.

7.6.2.4 Bedeutung der Operatoren. `+`, `-` und `*` arbeiten auf Zahlen und haben ihre übliche Bedeutung (ein etwaiger Überlauf soll ignoriert werden).

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

`<` und `=` liefern -1 für wahr und 0 für falsch. Die Operanden müssen beides Zahlen sein, oder beides Objektreferenzen (u.U. auf Objekte verschiedener Klassen). Bei `<` auf Objektreferenzen wird die Referenz genauso behandelt wie eine Zahl, und bildet eine strenge Totalordnung über die Objekte; der Compiler kann also den gleichen Code erzeugen, egal welchen Typ die Operanden von `<` haben.

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

7.6.2.6 Erzeugter Code. Ihr Compiler soll AMD64-Assemblercode ausgeben. Insbesondere müssen Sie die Klassentabellen erzeugen und die Methodenimplementationen.

Der erzeugte Code wird nach dem Assemblieren und Linken von C-Funktionen aufgerufen.

Jede Methode im Programm verhält sich gemäß der Aufrufkonvention, mit folgender Abweichung: im ersten Parameter wird `this` übergeben, der

Parameter n der Methode entspricht daher Parameter $n + 1$ der Aufrufkonvention. Beispiel: eine Methode `foo(a:int)` wird von C aus mit `foo(x,3)` aufgerufen, wobei `x` zu `this` wird und 3 zu `a`. Wobei das noch zu vereinfacht war: Tatsächlich können wir von C aus Methoden nur über die Klassentabelle aufrufen.

Wie erwähnt, sollen die Klassennamen 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 5 Parameter zusätzlich zu `this`. Die maximale Tiefe eines Ausdrucks[3] ist $\leq 5 - v$, wobei v die Anzahl der sichtbaren Variablen ist (`this` wird dabei nicht mitgezählt). 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 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 die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen

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

Fehlern der Fehlerstatus 3. Im Fall eines Fehlers darf die Ausgabe beliebig sein.

7.7 Codeerzeugung B

7.7.1 Termin

Abgabe spätestens am 2. Juni 2021, 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:

`new id` erzeugt ein Objekt der Klasse mit dem gegebenen Namen. Die Adresse der Klassentabelle ist initialisiert, keine der Objektvariablen ist initialisiert. Die Objekte kann Ihr Compiler auf dem Heap anlegen. Der Heap-Zeiger befindet sich im Register r15 und wird schon von der aufrufenden Funktion passend übergeben; der Heap wächst in Richtung größere Adressen. Um die Freigabe der Objekte brauchen Sie sich nicht kümmern, es ist genug Platz für alle Objekte, die in unseren Testprogrammen erzeugt werden.

Die *Variablendefinition* speichert den Wert von `Expr` unter dem Namen der Variable. Ein *lesender Zugriff* auf eine Variable liefert den Wert der Variable.

Bei einer *Zuweisung* wird `Expr` ausgewertet und das Resultat in die Variable, den Parameter, oder die Objektvariable 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 negativ, wird der then-Zweig ausgeführt, ansonsten der `else`-Zweig bzw. (falls es keinen `else`-Zweig gibt) nichts.

Die *while*-Anweisung wertet `Expr` aus. Ist das Ergebnis negativ, werden die `Stats` ausgeführt, und danach die *while*-Anweisung noch einmal von vorne. Ist das Ergebnis nicht negativ, wird die Ausführung hinter der *while*-Anweisung fortgesetzt.

Kommt die Ausführung zum Ende der Methode, 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, mit folgender Änderung: r15 ist jetzt kein callee-saved Register, sonder der Heap-Zeiger.

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 16. Juni 2021, 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 *Aufruf* wertet den `Term` aus, der eine Referenz auf ein Objekt O ergibt. Weiters wertet der Aufruf alle `Expr`s aus. Danach wird die Methodenimplementation aufgerufen, wobei die Referenz auf O als erstes Argument (für die aufgerufene Methode also als `this`) und die Exprs (von links beginnend) als weitere Argumente übergeben werden. Die Methodenimplementation wird gefunden, indem zum Anfang Klassentabelle der Klasse von O der Offset der abstrakten Methode mit dem Namen `id` addiert wird; an der sich ergebenden Stelle liegt dann die Startadresse der Methodenimplementation.

7.8.2.1 Erzeugter Code. Der erzeugte Code ruft Funktionen entsprechend den Aufrufkonventionen auf (mit Ausnahme der Behandlung von r15). Ansonsten gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, wobei ein Funktionsaufruf mit n Exprs bei der Berechnung der Tiefe mit dem Wert n (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.