Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2025

- 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 Forum (Details dazu siehe Übungshomepage).

Wenn Sie eine Frage zur Übung haben, stellen Sie sie am besten im Forum (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, code	info emacs, man vi	Editor
gcc	info as	Assembler
gcc	info gcc	C-Compiler
make	info make	baut Programme
flex	info flex	Scanner-Generator
bison, yacc	info bison, man yacc	Parser-Generator
dotty	`https://www.graphviz.org/Documentation.php`	Graphenzeichnen
ox, ox-1.04	`https://www.complang.tuwien.ac.at/ubvl/tools/doc/`	AG-basierter Compilergenerator
iburg, burg	`https://www.complang.tuwien.ac.at/ubvl/tools/doc/burg.pdf`	Baumparser-Generator
bfe	Skriptum	Präprozessor für burg
gdb	info gdb	Debugger
objdump	info objdump	Disassembler etc.

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.

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 „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 $\boxed{\text{Alt}}x$). Neben den genannten Dokumentationsquellen erfahren Sie mit `man` *`P`* für die meisten Programme, wie Sie *`P`* aufrufen können.

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

Das auf den Übungsgeräten unter `yacc` aufrufbare Programm ist `bison -y`. Mit `dotty` können Sie sich die Ausgabe von `bison -g` anschauen. `ox-1.04` ist eine ältere Version von `ox`; sollte `ox` nicht so funktionieren wie dokumentiert, probieren Sie, ob `ox-1.04` stattdessen funktioniert (und wenn ja, ist Ihr Code interessant, um einen Bug report an den aktuellen Maintainer von Ox zu schicken).

7 Beispiele

Es sind insgesamt acht Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen von Konzepten von Computerarchitekturen am Beispiel 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 Speicher für Datenstrukturen kann nicht innerhalb dieser Sprache angefordert werden. Sprachkonstrukte für Speicherzugriffe 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 erlernen Sie auch, wie verschiedene Sprachen miteinander kombiniert werden können. Weiters gibt es keine direkte Möglichkeit, Daten ein- oder auszugeben; diese Funktionen werden durch eine C-Funktion übernommen, die Funktionen in der Sprache aufruft, oder durch Aufrufen von C-Funktionen aus Funktionen in unserer Sprache.

Die Kenntnisse, die Sie bei den Assembler-Beispielen erlangen, werden Sie auch wieder bei der Codegenerierung der letzten Beispiele verwenden. Die Beispiele 3-8 können alle aufeinander aufbauend implementiert werden, d.h. wenn Sie Ihr Programm von Anfang an gut entwerfen, können Sie dieses ab dem Scanner-Beispiel bis zum Gesamtbeispiel stets wiederverwenden und erweitern. Beachten Sie jedoch, dass bei jeder Abgabe stets das gesamte Quellprogramm im Abgabeverzeichnis vorhanden sein muss (und zwar nicht in Form von symbolic links).

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

In dieser Sprache kann man, wie in vielen 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 19. März 2025, 14 Uhr.

7.1.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned long asma(unsigned char s[])
{
  unsigned long r=0;
  long i;
  for (i=0; i<64; i++)
    if (s[i]>='0' && s[i]<='9')
      r++;
  return r;
}
```

Schreiben Sie eine Version dieser Funktion in Assembler, ohne einen Kontrolltransferbefehl ausser `ret` zu verwenden. Verwenden Sie die Befehle `vpcmpltub`, `kmovq`, und `popcnt`; und viel mehr Befehle braucht eine Lösung für das Beispiel nicht.

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

```
unsigned long asma(unsigned char s[])
{
   return s[0];
}
```

mit z.B. `gcc -O -S` in Assembler übersetzen und sie dann erweitern. Dann stimmt schon das ganze Drumherum.

Sie können auch die Originalfunktion auf der Übungsmaschine `g0` mit `gcc -O3 -march=native -S` übersetzen, um zu sehen, wie der Compiler die AVX-512-Befehle einsetzt, auch wenn das Resultat die Anforderungen nicht erfüllt. Andererseits kann Sie die Ausgabe von gcc auch in die Irre führen, da gcc nicht erkennt, dass `popcnt` verwendet werden kann und daher aufwendigen Ersatzcode erzeugt.

7.1.3 Hinweis

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

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

7.1.4 Abgabe

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

7.2 Assembler B

7.2.1 Termin

Abgabe spätestens am 26. März 2025, 14 Uhr.

7.2.2 Angabe

Gegeben ist folgende C-Funktion:

```
unsigned long asmb(unsigned char s[], size_t n)
{
  unsigned long r=0;
  long i;
  for (i=0; i<n; i++)
    if (s[i]>='0' && s[i]<='9')
      r++;
  return r;
}
```

Schreiben Sie eine Version dieser Funktion in Assembler (diesmal mit Kontrollflussbefehlen). Verwenden Sie die Befehle `vpcmpltub`, `kmovq`, und `popcnt`.

Beachten Sie, dass auf die Adressen hinter `s[n-1]` (und vor `s[0]`) möglicherweise nicht zugegriffen werden darf, und vermeiden Sie solche Zugriffe. Bei maskierten Befehlen mit Speicherzugriffen (z.B. `vpcmpltub (%r10), %zmm0, %k0 {%k1}`) wird nur auf die Bytes zugegriffen, für die in der Maske (in diesem Fall in `%k1`) Bits gesetzt sind, und daher kann man solche Probleme damit vermeiden.

Eventuell wollen Sie für das Erzeugen der Maske für den oben genannten Zweck einen Shift-Befehl wie `shrq` verwenden. Beachten Sie, dass ein Shift-Betrag von 64 dabei wie ein Shift-Betrag von 0 wirkt (ähnliche Eigentümlichkeiten gibt es bei anderen Shift-Befehlen).

Für besonders effiziente Lösungen (gemessen an der Anzahl der *ausgeführten* (nicht statischen) 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 die geforderten Befehle verwendet und 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 2. April 2025, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `end array of int return if then else while do var or not`. Weiters soll er auch noch folgende Lexeme erkennen: `; ( ) , : := = < [ ] + * -`

Identifier bestehen aus Buchstaben, Ziffern und `_`, dürfen aber nur mit Buchstaben beginnen. Zahlen sind entweder Dezimalzahlen oder Hexadezimalzahlen. Hexadezimalzahlen beginnen mit `0x` gefolgt von einer oder mehr Hexadezimalziffern; Hexadezimalziffern dürfen sowohl groß als auch klein geschrieben werden. 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 nächsten `*/` gehen; Kommentare können also nicht geschachtelt werden. Alles andere sind lexikalische Fehler. Es soll jeweils das längste mögliche Lexem erkannt werden, `if39` ist also ein Identifier (longest input match), `39if` ist die Zahl `39` 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 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 9. April 2025, 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`

```
Program: { ( Funcdef | Funcdec ) ';' }
       ;

Funcdef: Funcdec Stats end  /* Funktionsdefinition */
       ;

Funcdec: id '(' Pars ')' ':' Type ;  /* Funktionsdeklaration */

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

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

Type: { array of } int ;            /* Typdeklaration */

Stats: { Stat ';' } ;

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

Cond: { Cterm or } [ not ] Cterm ;

Cterm: '(' Cond ')'
     | Expr '=' Expr
     | Expr '<' Expr
     ;

Lexpr: id                /* schreibender Variablenzugriff */
     | Term '[' Expr ']' /* schreibender Arrayzugriff */
     ;

Expr: Term { '+' Term }
    | Term { '*' Term }
    | Term '-' Term
    ;

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

Abbildung 2: Grammatik; Terminale sind klein geschrieben bzw. mit Hochkomma umschlossen; Nonterminale sind groß geschrieben.

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

Beseitigen Sie alle Konflikte in der Grammatik durch Umformen der Grammatik! `yacc` löst solche Konflikte zwar auf, aber nicht unbedingt in der von Ihnen gewünschten Art. Um solche Konflikte zu verstehen, rufen Sie `yacc/bison` mit `-Wcounterexample` auf, oder mit `-v` und schauen Sie sich die `.output`-Datei an.

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

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 7. Mai 2025, 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 (`id`): Funktionen und Variablen.

Eine Funktion wird in der *Funktionsdeklaration* deklariert. Der Name einer Funktion ist im ganzen Programm sichtbar, auch vor der Deklaration.

Bei einem *Funktionsaufruf* muss der Name der Funktion sichtbar sein, die Funktion muss also irgendwo im Programm deklariert werden. Weiters muss die Anzahl und die Typen der Parameter zwischen Aufruf und Deklaration übereinstimmen, siehe unten.

Ein Name, der in einer *Parameterdefinition* oder in einer *Variablendefinition* vorkommt, ist ein Variablenname. Variablen, die in einer Parameterdefinition definiert wurden, sind in der ganzen Funktion sichtbar. Variablen, die einer Variablendefinition definiert wurden, sind im unmittelbar umgebenden `Stats` ab dem nächsten `Stat` sichtbar, aber nicht in der Variablendefinition selbst, davor, oder nachdem das unmittelbar umgebende `Stats` zu Ende ist.

Bei einem *Variablenzugriff* (schreibend oder lesend) muss eine Variable oder ein Parameter mit dem Namen sichtbar sein.

Für jede Stelle gilt: Eine dort sichtbare Variable oder Funktion darf nicht den gleichen Namen haben wie eine andere dort sichtbare Variable oder Funktion.

Ihre statische Analyse soll alle diese Bedingungen prüfen, also dass ein Name in seinem Sichtbarkeitsbereich nur einmal deklariert bzw. definiert wird, dass in Funktionsaufrufen der Name einer Funktion vorkommt und in Variablenzugriffen der Name einer sichtbaren Variable.

7.5.2.2 Typen. Variablen und Ausdrücke (`Expr`) u.ä. (`Lexpr`, `Term`; nicht `Cond` und `Cterm`) haben Typen. Der Typ einer Variable wird bei der Definition der Variable (bzw. des Parameters) deklariert.

Bei einer *Variablendefinition* muss der Ausdruck den gleichen Typ haben wie die Variable.

Bei einer Zuweisung müssen die beiden Seiten den gleichen Typ haben.

Der Typ eines (schreibenden oder lesenden) *Variablenzugriffs* ist der Typ der Variable.

Bei einem (schreibenden oder lesenden) *Arrayzugriff* s muss der `Term` einen Typ `array of` x haben und die `Expr` den Typ `int`. Der Arrayzugriff hat dann den Typ x.

Bei einem `=`-Vergleich müssen die beiden Ausdrücke den gleichen Typ haben.

Bei einem `<`-Vergleich müssen die beiden Ausdrücke den Typ `int` haben.

Ein Ausdruck, der nur aus einem `Term` besteht, hat den Typ des Terms; genauso ist der Typ eines Terms, der aus `'(' Expr ')'` besteht, der Typ des Ausdrucks.

Bei Verwendung der Operatoren `+`, `*` und `-` müssen alle Operanden den Typ `int` haben und der Ausdruck hat den Typ `int`.

Der Typ von `num` ist `int`.

Der Typ eines Funktionsaufrufs ist im `Type` der Funktionsdeklaration dieser Funktion angegeben. Die Anzahl der Argumente des Funktionsaufrufs muss mit der Anzahl der Parameter in der Funktionsdeklaration übereinstimmen, und jedes Argument muss den gleichen Typ haben wie der entsprechende Parameter.

Zu überprüfen ist also, dass an jeder Stelle, wo ein bestimmter Typ erwartet wird, der Ausdruck o.ä. den Typ auch tatsächlich hat.

7.5.3 Hinweise

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

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

Verwenden Sie keine globalen Variablen oder Funktionen mit Seiteneffekten (z.B. Funktionen, die übergebene Datenstrukturen ändern) bei der Attributberechnung (Decoration)! `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 21. Mai 2025, 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; statisch inkorrekte Programme müssen weiterhin als solche erkannt werden.

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 folgende Datentypen:

- `int` ist eine Vorzeichenbehaftete 64-bit-Zahl (entspricht also `long` in C).
- Arrays werden durch ihre Anfangsadresse repräsentiert (ebenfalls ein 64-bit-Wert). Die gegebene Programmiersprache kann nur auf ein Array zugreifen, das in einer anderen Programmiersprachen erzeugt wird und als Funktionsargument (später auch als Resultat eine Funktionsaufrufs) an Programme der gegebenen Programmiersprache übergeben wird. Bei *Arrayzugriffen* soll weder der Compiler noch das Laufzeitsystem überprüfen, ob der Zugriff innerhalb der Arraygrenze liegt, es liegen auch nicht die dafür nötigen Informationen vor. Unsere Testprogramme führen keine Zugriffe ausserhalb der Arraygrenzen aus.

Das Laufzeitsystem soll keine Typprüfung vornehmen.

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

Bei einem (schreibenden oder lesenden) *Arrayzugriff* ist der vom `Term` (auf der rechten Seite der Regel) berechnete Wert die Addresse von Element

0 des Arrays, und der von `Expr` berechnete Wert der Index des Elements, auf das zugegriffen wird. Der Zugriff erfolgt dann auf die Adresse Term+8*Expr. Der *lesende Arrayzugriff* liefert als Resultat den 64-bit-Wert an dieser Adresse, der je nach Typ eine ganze Zahl oder die Anfangsadresse eines Arrays ist. Der *schreibende Arrayzugriff* überschreibt das 64-bit-Wort an dieser Stelle (ab Beispiel Codeerzeugung B).

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[1] ist $\leq 6 - v$, wobei v die Anzahl der sichtbaren Parameter und Variablen ist. 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 Ausnutzung der Adressierungsarten) und, wenn Sie besonders ehrgeizig sind, 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.

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

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

7.7 Codeerzeugung B

7.7.1 Termin

Abgabe spätestens am 4. Juni 2025, 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:

Eine `if`-Anweisung wertet `Cond` nach der Kontrollflussmethode aus. Wenn die Bedingung zutrifft, werden die `Stats` hinter dem `then` ausgeführt, sonst die `Stats` hinter dem `else`, bzw., falls kein `else`-Zweig vorhanden ist, gar nichts.

Eine `while`-Anweisung wertet `Cond` nach der Kontrollflussmethode aus. Wenn die Bedingung zutrifft, werden die `Stats` hinter dem `do` ausgeführt und danach die `while`-Anweisung von vorne begonnen. Wenn die Bedingung nicht zutrifft, macht die `while`-Anweisung nichts weiter, und die Ausführung wird dahinter fortgesetzt.

Die Bedingung kann auch ein `or` von mehreren Bedingungen (`Cterms`) sein. In diesem Fall verursacht die erste zutreffende Bedingung, dass der `then`-Zweig ausgeführt wird bzw. die Schleife fortgesetzt wird, ohne die weiteren Bedingungen zu prüfen (Kontrollflussmethode).

Die Bedingung kann auch ein `not` einer anderen Bedingung `Cterm` sein. In diesem Fall springt `Cterm` im Erfolgsfall zum `else`-Zweig bzw. hinter die `while`-Schleife, und im Misserfolgsfall zum `then`-Zweig bzw. die `while`-Schleife wird fortgesetzt, die beiden Sprungziele werden also vertauscht (Kontrollflussmethode).

`or` und `not` können auch verschachtelt angewendet werden, dann muss die Kontrollflussmethode entsprechend allgemeiner angewendet werden, siehe Vorlesungsskriptum.

Eine *Variablendefinition* wertet die `Expr` aus und weist das Ergebnis der definierten Variable zu.

Eine *Zuweisung* schreibt den Wert der `Expr` in die durch `Lexpr` angegebene Variable bzw. das Arrayelement.

Eine `Term`-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).

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 umfassen bzw. ein `Cterm`: die Möglichkeit, durch die Baumgrammatik Knoten zusammenzufassen und so zu optimieren, kann nur auf der Ebene von Ausdrücken und einfachen Anweisungen genutzt werden, solange man in der Zwischendarstellung nahe beim abstrakten Syntaxbaum bleibt.

Auf höherer Ebene ist es einfacher, für jede einfache Anweisung bzw. `CTerm` 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 18. Juni 2025, 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.