Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2023

- 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	`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* [Ctrl]*x* und `M-`*x* [Meta]*x* (auf den Übungsgeräten also [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.

7 Beispiele

Es sind insgesamt acht Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen von Assembler, im speziellen 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.

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 15. März 2023, 14 Uhr.

7.1.2 Angabe

Gegeben ist folgende C-Funktion:

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

Schreiben Sie eine Version dieser Funktion, die nur für $1 \leq$`n`≤ 64 funktionieren muss in Assembler, ohne einen Kontrolltransferbefehl ausser `ret` zu verwenden. Da Ihre Funktion nur auf die selben Speicherstellen zugreifen darf wie die Originalfunktion, müssen Sie mindestens die Speicherzugriffe über AVX-512-Befehle mit Masken bewerkstelligen (und der Großteil des Rests geht dann auch am einfachsten mit AVX-512-Befehlen und Maskenbefehlen).

Tipp: Aus `n` (im gegebenen Bereich) kann man in C wie folgt eine Maske machen, in der die untersten `n` bits gesetzt (1) sind und die anderen 0:

```
0xffffffffffffffffUL>>(64-n)
```

Wie Sie das in Assembler schreiben, ist natürlich Teil der Aufgabe.

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

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

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.

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 22. März 2023, 14 Uhr.

7.2.2 Angabe

Gegeben ist folgende C-Funktion:

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

(Ja, abgesehen vom Namen die gleiche Funktion wie im vorigen Beispiel).

Diesmal müssen Sie die Funktion für beliebige `n` in Assembler implementieren, dürfen dafür aber statisch (maximal) zwei Kontrolltransferbefehle neben `ret` verwenden, darunter kein Aufruf.

Tipp: Dafür werden Sie vielleicht auch eine Maske brauchen, bei der 0 Bits gesetzt sind. Allerdings geht die oben gezeigte Berechnung dafuer mit `shrq` nicht, weil ein Shift-Betrag von 64 wie ein Shift-Betrag von 0 wirkt.

Auswege: Die Notwendigkeit so einer Maske oder einer mit 64 gesetzten Bits vermeiden, den Extra-Fall mit `cmov` behandeln.

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 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 29. März 2023, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `object int class end return cond continue break not or new null`. Weiters soll er auch noch folgende Lexeme erkennen: `; ( , ) <- -> - + * > #`

Identifier bestehen aus Buchstaben, Ziffern und `_`, dürfen aber nur mit Buchstaben beginnen. Zahlen sind entweder Hexadezimalzahlen oder Dezimalzahlen. Eine Dezimalzahl beginnt mit einer Ziffer, gefolgt von beliebig vielen Ziffern und `_`. Eine Hexadezimalzahl beginnt mit `0x` gefolgt von beinahe beliebig vielen Hexademzimalziffern und `_`, wobei mindestens eine Hexadezimalziffer vorkommen muss; Hexadezimalziffern dürfen groß oder klein geschrieben werden.

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 und ohne `_`. 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 19. April 2023, 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 erzeug-

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

Selector: Type id '(' object { ',' Type } ')'
        ;

Type: int
    | object
    ;

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

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

Pars: Par { ',' Par }
    ;

Par: Type id  /* Parameterdefinition */
   ;

Stats: { Stat ';' }
     ;

Stat: Return
    | Cond
    | Type id '<-' Expr /* Variablendefinition */
    | id '<-' Expr /* Zuweisung */
    | Expr         /* Ausdrucksanweisung */
    ;

Return: return Expr
      ;

Cond: cond { Guarded ';' } end
      ;

Guarded: [ Expr ] '->' Stats ( continue | break )
       ;

Expr: { not | '-' } Term
    | Term { '+' Term }
    | Term { '*' Term }
    | Term { or Term }
    | Term ( '>' | '#' ) Term
    | new id
    ;

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

ten 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.

7.5 Attributierte Grammatik

7.5.1 Termin

Abgabe spätestens am 3. Mai 2023, 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: Klassen, Methodenselektoren, Objektvariablen, Parameter, Variablen. Sie alle teilen sich einen Namensraum. In diesem Abschnitt beziehen sich die *hervorgehobenen* Begriffe auf den Namen, der durch `id` in der Zeile des Kommentars repräsentiert wird.

Eine *Klassendefinition* definiert einen Klassennamen. Ein `Selector` definiert einen Methodenselektor (in Java-Terminologie: eine abstrakte Methode). Eine *Objektvariablendefinition* definiert eine Objektvariable. Eine *Parameterdefinition* definiert einen Parameter. Eine *Variablendefinition* definiert eine (lokale) Variable.

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

Klassen und Methodenselektoren sind im ganzen Programm sichtbar, auch vor ihrer Definition. Objektvariablen sind in der gesamten Klasse sichtbar, auch vor ihrer Definition. Parameter sind in der gesamten Methodenimplementierung 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 aus dem `Stats` abgeleiteten Programmteils, das die Variablendefinition unmittelbar enthält; bei dem *Stats* in der *Methodenimplementierung* geht der Sichtbarkeitsbereich noch weiter und umfasst auch noch das `Return`.

Bei der Verwendung eines Namens muss dieser Name sichtbar sein und von der richtigen Art sein: Bei einer *Methodenimplementierung* oder einem *Aufruf* muss der Name eines Methodenselektors verwendet werden, bei einer *Zuweisung* oder einem *lesenden Zugriff* der Name einer Objektvariablen, eines Parameters, oder einer Variablen (alle drei Arten von Namen sind erlaubt), bei einem `new`-Ausdruck der Name einer Klasse.

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.

7.5.2.2 Typen. Diese Sprache hat zwei grundlegende Typen: `int` und `object` (eine Objektreferenz). Ihr Compiler soll eine Typüberprüfung entsprechend folgender Regeln durchführen:

Bei einer Methodenimplementierung mit dem Namen *id* muss die Anzahl und die Typen der Parameter und der Typ des Rückgabewerts (links der `id`) mit den Entsprechungen in der `Selector`-Definition mit dem selben Namen übereinstimmen.

Bei einer `return`-Anweisung muss der Typ des Ausdrucks `Expr` mit dem Typ des Rückgabewerts der umgebenden Methode übereinstimmen.

Nach einer Variablendefinition hat die Variable `id` nachher den angegebenen Typ. Der Typ der `Expr` muss der gleiche sein.

Bei einer Zuweisung muss der Typ der `id` und der `Expr` der gleiche sein.

Bei einer `Guarded`-Klausel muss der Ausdruck vor dem `->` (so vorhanden) ein `int` sein.

`not - + * or >` erwarten `int`-Terme und ihr Ergebnis ist ein `int`.

Bei `#` müssen die beiden Terme den gleichen Typ haben; das Ergebnis ist ein `int`.

`new id` liefert ein `object`.

Ein geklammerter Ausdruck hat den gleichen Typ wie ein ungeklammerter.

`num` ist ein `int`.

`null` ist ein `object`.

Das Ergebnis eines *lesenden Zugriffs* hat den Typ des `id`s (der in der *Objektvariablendefinition*, *Parameterdefinition*, oder *Variablendefinition* angegebene Typ).

Das Ergebnis eines *Aufruf*s hat den Typ des Rückgabewerts der Methode `id`.

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! `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 17. Mai 2023, 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 Methodenselektoren repräsentiert. Für jeden Methodenselektor im Programm sind dort 8 Bytes vorgesehen; der erste Methodenselektor 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.

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 (wenn dahinter kein Eintrag einer implementierten Methode kommt). Es kann aber hilfreich sein, um Fehler in den Testprogrammen zu erkennen.

```
object m0(object);
object m1(object,int);
class c
  object next
  int val
  object m1(object o, int v)
    next = new c;
    val = v;
    return o;
  end;
end;
```

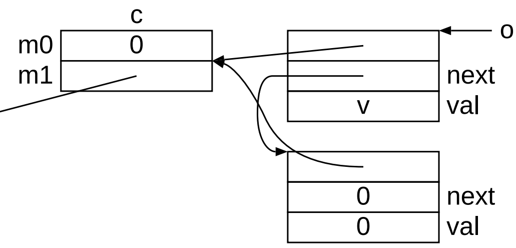

Abbildung 3: Tabelle der Methodenselektoren der Klasse `c` (vom Compiler statisch erzeugt) und zwei Objekte dieser Klasse unmittelbar vor dem `return`: das als `o` übergebene Objekt (anderswo erzeugt) und das mit `new c` zur Laufzeit Merzeugte. Der Pfeil auf den Quellcode der Methodenimplementierung deutet an, dass in diesem Tabelleneintrag eigentlich die Adresse des *Maschinen*codes der Methodenimplementierung steht. Das Programm verwendet auch Teile der Sprache, die erst in Codeerzeugung B erklärt werden.

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.

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.

Abbildung 3 zeigt ein Beispiel für die Datenstrukturen von Klassen und Objekten.

7.6.2.3 `ints` und `objects`. Ein `int` ist eine vorzeichenbehaftete 64-bit-Zahl. Ein `object` (Objektreferenz) wird durch die Anfangsadresse des Objekts repräsentiert.

Das Laufzeitsystem soll keine Überprüfung jedweder Art vornehmen. Unsere Testprogramme führen keine Aufrufe auf `null`-Objekte oder von Methoden aus, die für die Klasse nicht implementiert sind.

Lesende Zugriffe auf Parameter liefern den Wert des Parameters. *Lesende Zugriffe* auf Objektvariablen liefern den Wert dieser Objektvariablen in dem Objekt, das der Methodenimplementierung als erster Parameter übergeben wurde.

Sie wundern sich vielleicht, warum die Methodenimplementierung auf Objektvariablen der aktuellen Klasse zugreifen darf, und wieso der erste Parameter so eine spezielle Rolle einnimmt: Weil in dieser Programmiersprache der erste Parameter verwendet wird, um die Methodenimplementierung für den Methodenselektor auszuwählen: Wenn eine Methodenimplementierung einer Klasse C aufgerufen wurde, ist der erste Parameter daher ein Objekt der Klasse C. Der erste Parameter nimmt also die Rolle ein, die in einem Java-Aufruf `o.m(...)` das Objekt `o` einnimmt, bzw. innerhalb der Methodenimplementierung entspricht der erste Parameter dem `this` in Java.

`null` liefert den Wert 0 (als Objektreferenz).

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

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

`>` und `#` liefern -1 für wahr und 0 für falsch. Dabei prüft `>`, ob der linke Operand größer ist als der rechte, und `#`, ob die beiden Operanden ungleich sind.

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.

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 6 Parameter. Die maximale Tiefe eines Ausdrucks[1] ist $\leq 6 - v$, wobei v die Anzahl der sichtbaren 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,

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

die Sie mit `iburg` tun können, also eine gute Codeauswahl (besonders bezüglich konstanter Operanden) und, wenn Sie sehr engagiert 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. 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 31. Mai 2023, 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, das wie folgt initialisiert wird: das Feld bei Offset 0 zeigt auf die Klassentabelle von `id`; alle Objektvariablen werden mit 0 initialisiert.[2] Die Objekte kann Ihr Compiler auf dem Heap anlegen. Der Heap-Zeiger befindet sich im Register

[2] Für Objektvariablen vom Typ `object` heisst das, dass sie mit `null` initialisiert werden.

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).

Bei einem `Cond` wird das Expr des ersten enthaltenen `Guarded` ausgewertet; wenn das Resultat ungleich 0 ist, werden die enthaltenen `Stats` ausgeführt, ansonsten wird mit dem nächsten `Guarded` weitergemacht. Wenn das `Guarded` keine `Expr` hat, werden die `Stats` auf jeden Fall ausgeführt. Wenn es kein nächstes `Guarded` gibt, wird hinter dem `Cond` weitergemacht. Wenn in einem `Guarded` das Ende der `Stats` erreicht wird, springt `continue` an den Anfang des `Conds`, `break` an das Ende.

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, sondern der Heap-Zeiger.

7.7.3 Hinweis

Es bringt nichts, für `iburg` Bäume zu bauen, die mehr als eine einfache Anweisung oder den Teil eines `Guarded` 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 14. Juni 2023, 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 alle `Expr`s aus und ruft dann die passende Methodenimplementierung entsprechend der Aufrufkonvention (mit der Abweichung bezüglich `r15`) auf. Der von der Methode zurückgegebene Wert ist der Wert des Aufrufs. Die passende Methodenimplementierung wird gefunden, indem aus der Klassentabelle des Objekts o im ersten Parameter der Eintrag für den Methodenselektor `id` gelesen wird.

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 $\max(0, n-1)$ (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.