Praktikum: Compilerbau

Betreuer: André-Marcel Hellmund
4. Semester - Sommersemester 2012

1 Organisatorisches

Die Bearbeitung der Aufgabe erfolgt in Gruppenarbeit mit maximal 3 Gruppenmitgliedern. Die Aufgaben sollten idealerweise unter den Gruppenmitgliedern gleich verteilt werden - es wird aber nur die komplette Gruppe bewertet, nicht die einzelnen Gruppenmitglieder.

Die Bewertung des Projektes erfolgt binär: **bestanden** oder **nicht bestanden**. Bei Nichtbestehen des Projektes wird das Praktikum mit anderer Aufgabenstellung im nächsten Semester wiederholt.

Der späteste Abgabetermin ist: **12.06.2012**. Zwischenabgaben des Projektes erfolgen in Abständen von 4 Wochen. Weitere Details zu den Zwischenabgaben werden in der Einführungsveranstaltung bekannt gegeben.

Bei Fragen: jederzeit per eMail an **amh@hp.com**

2 Aufgabenstellung

Die Aufgabe in diesem Praktikum besteht darin, einen Compiler für eine Untermenge der Programmiersprache ANSI C zu entwickeln. Als Ausgabe soll Assembler-Code für die MIPS32-Architektur (RISC) erzeugt werden. Als Implementierungssprache ist C vorgesehen.

Bei der Umsetzung des Projektes sollen die Studenten die wesentlichsten Komponenten eines Compilers parallel zur Vorlesung Compilerbau praktisch vertiefen. Dazu zählen:

- Implementierung des Scanners in GNU Flex
- Anpassung einer vorgegebenen Grammatik in GNU Bison
- Implementierung einer geeigneten (2-stufigen) Symboltabelle
- Implementierung einer Zwischensprache (3-Adress-Code)
- Implementierung eines einfachen Typsystems
- Implementierung der MIPS32-Assembler Codeerzeugung

2.1 Nicht-funktionale Rahmenbedingung

Neben den funktionalen Aspekten der Compilerentwicklung solltet ihr versuchen auch die folgenden nicht-funktionalen Aspekte der Programmierung zu berücksichtigen.

- sicherer Code (überprüft die Rückgabewerte von Bibliotheks-/Systemaufrufsfunktionen auf Fehlerwerte)
- angemessene Fehlerbehandlung (der Compiler sollte dem Benutzer aussagekräftige Fehlermeldung mitteilen)

- Komplexität von Funktionen (Länge der Funktionen und Verschachtelungstiefe möglichst gering halten)
- sprechende Funktions- und Variablennamen (welches Schema ihr dabei wählt, spielt keine Rolle; Konsistenz des gewählten Schemas ist wichtiger) - i, j, k, etc. sind durchaus gängige Namen für Schleifen(zähler)variablen
- sinnvolle Kommentare im Code
- Beschreibungsblock für jede definierte Funktion mit Informationen über (a) Funktionsname, (b) kurze Funktionsbeschreibung, (c) Übergabeparameter und deren Bedeutung, (d) Rückgabewert und seine Bedeutung, (e) Fehlercodes und ihre Bedeutung und (f) sonstige Hinweise.
- Empfohlen sind die Programmier-Richtlinien für den Linux-Kernel. [KH02]

2.1.1 Doxygen

Da dieses Projekt in der Gruppe bearbeitet wird, sollte jede Funktion mit einem entsprechenden Header versehen werden. Wir haben uns dabei für das freie Werkzeug Doxygen (http://www.doxygen.org) entschieden, das Dokumentation aus den Kommentaren im Quelltext automatisch erzeugt. Der beigefügte Code ist bereits mit Doxygen-Kommentaren versehen. Ansonsten bietet die Webseite des Projektes ausreichend Dokumentation zu weiteren (Kommentar)befehlen.

2.1.2 Valgrind

Valgrind (http://valgrind.org) ist ein freies Tool, das einem Entwickler erlaubt sein Programm auf Speicherlecks zu überprüfen. Valgrind bietet noch weitere Tests an, aber die Speicherüberprüfung ist die prominenteste Funktionalität. Es gibt sicherlich verschiedene Argumente ob die Forderung nach speicherleckfreien Programm eine funktionale oder nicht-funktionale Anforderung ist. Wir haben diese Anforderung unter den nicht-funktionalen aufgelistet - immer vor dem Hintergrund, dass ein Speicherleck zu einem beliebigen Ausführungszeitpunkt zu einem funktionalen Problem führen kann: wenn kein Speicher mehr zur Verfügung steht. Im Allgemeinen kann ein Compiler bei großen Programmen einen hohen Speicherverbrauch für Datenstrukturen zur Optimierung haben; deshalb ist es wichtig, dass der Compiler an keiner Stelle Speicherlecks aufweist. Wir haben in die Projekt-Makefile ein Ziel zur Überprüfung mit Valgrind eingefügt. Der finale Compiler sollte idealerweise ohne Speicherlecks ausgeführt werden - auch im Fehlerfall.

2.2 Definition des Sprachumfangs

Bevor wir die einzelnen Aufgaben detaillierter beschreiben, wollen wir zunächst den Sprachumfang, d.h. die geforderten Sprachkonstrukte, des Compilers festlegen. Der zu entwickelnde Compiler soll dabei lediglich eine stark reduzierte Untermenge von ANSI C (ISO C99 Standard) verstehen und übersetzen können.

- Deklarationen (Prototypen) und Definitionen von Funktionen mit Rückgabewert und Übergabeparametern
- Definitionen von globalen und lokalen Variablen. In weiteren Unterblöcken (z.B. Schleifenrumpf) definierte Variablen werden als lokale Variablen betrachtet (weitere Details zu Gültigkeitsbereichen folgen weiter unten).
- Standard-Datentypen: *int* und *void*
- Erweiterte Datentypen: eindimensionale, statische Arrays

- binäre, arithmetische Operatoren: Addition (+), Subtraktion (−), Multiplikation (∗), Shift-Left (<<), Shift-Right (>>)
- binäre Vergleichsoperatoren: größer (>), größer gleich (>=), kleiner (<), kleiner gleich (<=), gleich (==), ungleich (! =)
- binäre, logische Operatoren: UND (&&), ODER (||)
- unäre, arithmetische Operatoren: Minus (−)
- unäre, logische Operatoren: logisches NICHT (!)
- unäre Feldoperatoren: Array-Zugriff([]). Als Einschränkung werden beim Array-Zugriff nur Konstanten und Integer-Variablen als Index zugelassen, aber keine beliebigen Ausdrücke.
- Runde-Klammern, (), werden in Ausdrücken zur Festlegung der Ausführungsreihenfolge genutzt
- Schleifen: while, do-while
- Verzweigungen: if-then, if-then-else
- Funktionsaufrufe - ohne variable Anzahl von Übergabeparametern
- Rückgabewerte aus Funktionen (return)

Hinweis: Alle anderen Konstrukte von ANSI C werden nicht unterstützt. Beispielsweise wird keine explizite Typumwandlung durch eingeklammerte Typen vor einem Variablen-Bezeichner, wie in ANSI C üblich, unterstützt. Ebenfalls wird der Sequenz-Operator , nicht unterstützt.

Entsprechende Testprogramme werden im Laufe des Praktikums zur Verfügung gestellt mit denen ihr testen könnt, ob euer Compiler die geforderten Sprachkonstrukte korrekt behandelt. Bitte testet aber nicht nur die zur Verfügung gestellten Testprogramme, sondern auch leichte Veränderungen dieser Programme. Wir könnten auf die selbe Idee kommen. ☺

Assoziativität und Priorität von Operatoren in C

Wenn mehrere Operatoren ohne explizite Klammerung in einem Ausdruck (z.B. $a = b + c + d * e << f$) verwendet werden, spielen die Assoziativität (Links- oder Rechtsassoziativ) und auch die Priorität (Ausführungsreihenfolge) eine besondere Rolle. Hier findet ihr als Referenz die Assoziativitäts- und Prioritätstabelle von ANSI C99 für die oben aufgelisteten Operatoren:

Priorität	Operator	Assoziativität
1	[]	Links
2	(unär) −, !	Rechts
3	∗	Links
4	+, −,	Links
5	<<, >>	Links
6	<, <=, >, >=	Links
7	==, ! =	Links
8	&&	Links
9	||	Links

Tabelle 1: Prioritäts- und Assoziativitätstabelle

2.3 Scanner-Implementierung in GNU Flex

In diesem Aufgabenteil sollt ihr einen kompletten Scanner in GNU Flex implementieren. Die dazu benötigten Token (Begriffsdefinition: siehe unten) werden nicht vorgegeben, sondern sollen von Euch aus der vorgegebenen Parserimplementierung extrahiert werden. Als Vorgabe jedoch die lexikalische Beschreibung von Bezeichnern, ganzen Zahlen und Kommentaren:

- **Bezeichner:** ein Buchstabe oder Unterstrich gefolgt von einer beliebigen Anzahl von Buchstaben, Zahlen und Unterstrichen.
- **ganze (Dezimal)Zahlen:** entweder eine Null oder aber eine 1 gefolgt von einer beliebigen Anzahl von Zahlen zwischen Null und Neun. **Hinweis:** eine Null gefolgt von einer beliebigen Anzahl von Zahlen (ungleich Null) wird in C als Oktalzahl (Basis 8) interpretiert. Diese Notation soll aber bereits durch den Scanner ausgeschlossen werden. Wir haben in der Definition von ganzen Zahlen bewusst das Vorzeichen ausgelassen. Negative Zahlen werden über den Parser und die unäre Minus-Operation realisiert.
- **Kommentar:** ein einzeiliger Kommentar beginnt mit einem Doppelslash (//) und endet am Ende der entsprechenden Zeile. Kommentare sollten ignoriert und nicht an den Parser übermittelt werden. **Hinweis:** Mehrzeilige Kommentare sind nicht Teil unserer Sprache.

Als Referenz werden hier einige Begrifflichkeiten festgehalten, die im Kontext der Scannerimplementierung wichtig sind [ALSU07]:

- **Token:** Als Kommunikationselement zwischen Scanner und Parser werden sogenannte Tokens definiert, die im Allgemeinen aus einem Tokennamen und gegebenenfalls mehreren Attributen bestehen. Ein Token dient der Abstraktion einer (beliebig großen) Menge von lexikalischen Einheiten zu einem syntaktischen Grundbaustein der Programmiersprache. Häufig vorkommende Grundbausteine (bzw. Token) sind z.B. Bezeichner, dezimale Ganzzahlen oder auch arithmetische Operatoren.
- **Muster:** Die zu einem Token gehörende Menge von lexikalischen Einheiten (Zeichenketten) wird durch ein sogenanntes Muster (Pattern) beschrieben. Das Muster für einen Bezeichner in C würde in Prosa wie oben beschrieben lauten. In GNU Flex werden die Muster mit Hilfe von regulären Ausdrücken definiert.
- **Lexem:** Als Lexem wird die zu einem Muster passende Zeichensequenz der Eingabedatei bezeichnet. Zu dem oben definierten Muster eines Bezeichners sind zum Beispiel 'a_123' und '_abc' gültige Lexeme.

Zu GNU Flex gibt es in der Einführungsveranstaltung eine kurze Einleitung zur Syntax und Semantik. Die aktuelle Dokumentation zu GNU Flex (http://www.gnu.org/software/flex/) ist im Projektarchiv beigefügt.

2.4 Parser-Erweiterung in GNU Bison

Die Grammatik für den definierten Sprachumfang ist bereits vorgegeben. Aktuell erzeugt die Grammatik aber noch eine Vielzahl von sogenannten Shift/Reduce - Konflikten, die durch entsprechende Definition von Assoziativität und Priorität der Operatoren gelöst werden können.

Die Hauptaufgabe in diesem Bereich besteht aber darin, die Grammatik um geeignete semantische Aktion für die einzelnen Produktionsregeln zu erweitern. Die semantischen Aktionen sollten dabei das Quellprogramm in eine Zwischensprache (siehe unten) übersetzen.

Zu GNU Bison gibt es in der Einführungsveranstaltung eine kurze Einleitung zur Syntax und Semantik. Die aktuelle Dokumentation zu GNU Bison (http://www.gnu.org/software/bison/) ist im Projektarchiv beigefügt.

2.5 Symboltabellenimplementierung

Die Aufgabe der Symboltabelle ist alle wichtigen Informationen zu Bezeichnern und Funktionen zu speichern, die während des Übersetzungsvorgangs vom Compiler gefunden werden. Zu Funktionen könnten zum Beispiel folgende Daten gespeichert werden:

- Funktionsname
- Rückgabetyp
- Anzahl Parameter
- Name, Typ und Reihenfolge der Übergabeparameter
- Referenz auf den Zwischencode der Funktion

Zu einem Bezeichner/Variable könnten unter anderem folgende Informationen abgespeichert werden:

- Variablenname
- Variablentyp (z.B. *int* Array)
- Speichergröße der Variable - im Fall von *int* sind dies 4 Byte auf unserer Zielplattform (folgt unten)
- Offset. Variablen werden entweder in einem globalen Datensegment (für globale Variablen) oder auf dem Stack einer Funktion (lokale Variablen) abgelegt. Dabei steht häufig nur ein Zeiger auf den Anfangsbereich des Segments/Stacks zur Verfügung. Um die finale Adresse einer Variablen zu ermitteln, muss die Startadresse mit dem gespeicherten Offset addiert werden.

In der Auflistung der Aufgabenstellung wird eine 2-stufige Symboltabelle erwähnt. Dahinter verbirgt sich lediglich ein Implementierungshinweis für die Symboltabelle. Es gibt aber keinerlei Vorgaben für Eure Umsetzung der Symboltabelle. **Hintergrundinformation:** In C ist es durchaus erlaubt zwei oder mehrere Variablen mit dem selben Namen anzulegen, sofern die Variablen in unterschiedlichen Gültigkeitsbereichen (scopes) angelegt werden. Mögliche Gültigkeitsbereiche in C99 sind: global (file scope), innerhalb einer Funktion (function scope), innerhalb einer Funktion innerhalb eines verschachtelten Blocks (block scope). In C99 wird noch ein vierter Gültigkeitsbereich genannt, der sogenannte Funktionsprototyp (function prototype scope) Gültigkeitsbereich. Im Kontext dieses Praktikums soll lediglich der globale Gültigkeitsbereich und der Funktionsgültigkeitsbereich (keine verschachtelten Blöcke) betrachtet werden. Neu definierte Variablen innerhalb von verschachtelten Blöcken sowie Funktionsparameter werden zum Funktionsgültigkeitsbereich gezählt.

Hinweis: wenn eine Variable innerhalb eines Gültigkeitsbereiches mehrfach definiert wird, sollte der Compiler eine Fehlermeldung an den Benutzer schicken und die Übersetzung abbrechen.

Implementierungshinweis: Bei der Implementierung der Symboltabelle werdet ihr mit sehr hoher Wahrscheinlichkeit auf eine Hash- und/oder Listenimplementierung zurückgreifen wollen. Als eine mögliche Implementierung können wir euch die freie Implementierung des UTHash-Projektes empfehlen. Die benötigten Header-Dateien sind im Projektarchiv enthalten und in der Make-Datei bereits implementiert. Dokumentation und Beispielcode sind auf der Projektseite (http://uthash.sourceforge.net/) vorhanden.

2.6 Zwischencodeerzeugung

Eine Vielzahl der heute entwickelten und verwendeten Compiler verwendet eine oder mehrere Zwischensprachen als interne Darstellung des eingelesenen Quellprogramms. Die Konvertierung des Quellprogramms in eine Zwischensprache wird unter anderem aus folgenden Gründen durchgeführt:

- Optimierungen lassen sich i.d.R. einfacher und effizienter auf geeigneten Zwischensprachen ausführen (z.B. auf der Static Single Assignment (SSA)-Form einer Zwischensprachen)
- eine geeignete Zwischensprache erlaubt die Abstraktion von den Spezifika der Eingabesprache. Ein Compiler könnte derart gestaltet werden, dass eine Optimierungs- und eine Codeerzeugungseinheit, aber mehrere Spracheinheiten (C, C++, Python) vorhanden sind. In diesem Fall würde jede dieser Spracheinheiten Zwischencode erzeugen und die Optimierungs- und Codeerzeugungsarbeit könnte dann unabhängig von der Eingabesprache implementiert werden.

Der zu implementierende Compiler soll auch mit einer Zwischensprache ausgestattet werden. Auf Basis dieser Zwischensprache soll auch die letztendliche Codeerzeugung implementiert werden. Wir wollen Euch eine mögliche Zwischensprache auf Basis von 3-Adress-Code vorstellen. Die Zwischensprache ist angelehnt an [ALSU07] und [Muc97].

2.6.1 3-Adress-Code

3-Adress-Code ist eine Form der Tupel-basierten Zwischensprache. Eine Funktion besteht dabei aus einer Sequenz von Tupeln, wobei jedes Tupel eine Operation, z.B. Addition, auf (Variablen)operanden/symbolen repräsentiert. Als bedeutende Einschränkung besteht dabei jedes Tupel/Operation aus maximal 3 Operanden, z.B. einem Zieloperand und zwei Eingabeoperanden bei einer Addition. Die folgende Tabelle listet mögliche Zwischencode-Operationen auf, wobei wir lediglich die ASCII-Repräsentation und keine binäre/implementierungsbezogene Repräsentation angeben.

Opcode	**# Operanden**	**Semantik**
ASSIGN	3	R = S1
ADD	3	R = S1 + S2
SUB	3	R = S1 - S2
MUL	3	R = S1 * S2
DIV	3	R = S1 / S2
MINUS	2	R = - S1
IFEQ	3	IF S1 == S2 GOTO L
IFNE	3	IF S1 != S2 GOTO L
IFGT	3	IF S1 > S2 GOTO L
IFGE	3	IF S1 >= S2 GOTO L
IFLT	3	IF S1 < S2 GOTO L
IFLE	3	IF S1 <= S2 GOTO L
GOTO	1	GOTO L
RETURN	1	RETURN S
RETURN	0	RETURN
CALL	3	R = CALL FUNC, (LISTE)
CALL	2	CALL FUNC, (LISTE)
MEM_LD	2	R = [M]
MEM_ST	2	[M] = S
ADDR	2	R = ADDR(A)

Tabelle 2: Zwischencode-Repräsentation

Der erste Abschnitt beschreibt die arithmetischen Operationen. Dabei stellen die Parameter $S_{1,2}$ (Quell)variablen oder Konstanten dar. (Quell)variablen können in diesem Kontext Integer-Variablen im globalen oder loka-

len Variablen-Scope in der Symboltabelle, oder auch temporäre Integer-Variablen sein. Nur ganze Zahlen sind als Konstanten zugelassen. Der Parameter R stellt eine (Ziel)variable dar, d.h. Integer-Variablen im globalen oder lokalen Variablen-Scope in der Symboltabelle, oder auch neue oder bereits existierende temporäre Integer-Variablen. Temporäre Variablen werden dabei durch Verwendung auf der linken Seite der Zuweisung implizit erzeugt - sie können, aber müssen nicht in der Symboltabelle notiert werden.

Im Kontext von (Ziel)variablen ist wichtig zu beachten, dass wir uns auf der Ebene des Zwischencodes befinden, und, dass dieser eine abstrakte Ebene (unabhängig von der Zielarchitektur) beschreibt. In unserem Zwischencode existiert daher kein beschränkter Registersatz (register file) oder Speicher (main memory). Auf der Ebene unseren Zwischencodes können wir daher mit einer unbeschränkten Menge von (virtuellen) temporären Variablen, neben den Variablen in der Symboltabelle, arbeiten. Im Prinzip könnt ihr mit beliebigen Bezeichnern für temporäre Variablen im Zwichencode arbeiten, solange die Namen nicht in der Symboltabelle erschienen können. Als einfacher Ansatz sind Variablen fortlaufend mit $\#V_i : i \in \mathbb{N}^+$ zu bezeichnen.

Der zweite Abschnitt listet die bedingten und unbedingten Sprünge auf. Dabei stellt der Parameter L ein Label (bzw. eine Zeilennummer des Zwischencodes) dar. Die Parameter $S_{1,2}$ sind wie im ersten Abschnitt zu interpretieren. Die bedingten und unbedingten Sprüngen können für Verzweigungen und Schleifen verwendet werden.

Im dritten Abschnitt werden die funktionsbezogenen Operationen gezeigt. Der letzte Parameter des Funktionsaufrufs ist dabei eine Liste von Funktionsparametern, wobei, wie im ersten Abschnitt, ein Funktionsparameter entweder eine (Quell)variable, temporäre Integer-Variable oder aber eine Konstante darstellt.

Im letzten Abschnitt wiederum werden die speicherbezogenen Operationen aufgezeigt. Der Parameter M stellt hier eine (Quell- oder Ziel)variable, oder eine Konstante dar. Als Konstanten sind, wie im ersten Abschnitt, nur ganze Zahlen zugelassen. Als (Quell- oder Ziel)variablen sind hier allerdings nur temporäre Integer-Variablen zugelassen. Der Inhalt dieser Variablen wird von den Operationen MEM_LD und MEM_ST als Speicheradresse interpretiert. Die Adresse einer lokalen oder globalen Integer-Array-Variablen in der Symboltabelle wird mittels der ADDR()-Operation bestimmt. Daraus ist auch erkennbar, dass der Parameter A nur lokale oder globale Integer-Array-Variablen-Bezeichner in der Symboltabelle darstellt. Die Zwischencode-Tabelle listet keine extra Operationen für Array-Zugriffe auf, deshalb müssen Array-Zugriffe (lesen/schreiben) über die Speicheroperationen implementiert werden.

Ein Beispiel für die hier vorgestellte Zwischensprache ist im Projektarchiv angehangen.

Hintergrundinformation: Zwischencode-Repräsentation können grundsätzlich in drei Kategorien eingeteilt werden: high-level, middle-level, low-level. Die high-level Repräsentation ist sehr stark an den Eingabesprachen orientiert, d.h. beinhaltet Operationen wie Array-Zugriff, Verzweigungen und Schleifen als explizite Operationen. In der middle-level Repräsentation können diese explizite Operationen in Operationen umgewandelt werden, die sich näher an den Instruktionen von Zielarchitekturen orientieren, d.h. Schleifen werden durch bedingte und unbedingte Sprünge realisiert. Die low-level Repräsentation wiederum ist nochmal stärker an den Zielarchitekturen angelehnt. In dieser Repräsentation können z.B. explizit Register und Speicherinstruktionen implementiert werden. Die oben vorgestellte Zwischensprache kann am ehesten zu den middle-level Repräsentationen gezählt werden.

2.7 Typsystem

Ziel des Typsystems einer Programmiersprache ist die Erkennung und/oder Verhinderung von Laufzeitfehlern eines Programms, welche durch die Verwendung von inkorrekten Variablen-Typen in einem Kontext ausgelöst werden würden – auch als Erkennung und Verhinderung von Typverletzungen/-fehlern durch Typüberprüfung (type checking) bezeichnet. Im Allgemeinen definiert das Typsystem der Programmiersprache alle in der Sprache vorkommenden Variablen-Typen und es stellt sicher, wie erwähnt, dass Variablen zur Laufzeit korrekt verwendet werden, d.h., dass Variablen entweder nur in gültigen Kontexten verwendet werden, inkorrekte Verwendungen zuverlässig erkannt werden, oder inkorrekte Verwendungen erfolgreich verhindert werden.

Dadurch kann beispielsweise verhindert werden, dass eine Funktion zur Laufzeit einen String übergeben bekommt, obwohl ein Integer erwartet worden wäre, was, je nach Programmiersprache, entweder zu einem erkannten Fehler zur Compile-Zeit, einem erkannten Fehler zur Laufzeit (z.B. eine Exception in Java), einer automatischen Typumwandlung (implicit type cast), oder einem unentdeckten Rechenfehler führt. Der letzte Fall ist natürlich der schlechteste, aber in C durchaus möglich.

Neben der Menge gültiger Variablen-Typen der Programmiersprache, schreibt das Typsystem also auch einen Satz von Regeln für den Umgang mit Typfehlern vor. Außerdem definiert es die Mechanismen für die Erkennung von Typfehlern. Eine Sprache, deren Typsystem alle Typfehler spätestens zur Laufzeit zuverlässig erkennt (wie beispielsweise Java), wird auch als typsicher bezeichnet. Bei ANSI C handelt es sich hingegen nicht um eine typsichere Sprache, da beispielsweise ein Fehler in einer expliziten Typumwandlung (explicit type cast) zur Laufzeit nicht erkannt wird, im Gegensatz zu Java.

Unsere Sprachdefinition sieht insgesamt zwei Typen vor: *int* und *int*[ ].

Allgemein sollten die folgenden Regeln im Typsystem umgesetzt werden:

- einer Variablen vom Typ *int* dürfen Werte vom Typ *int*[ ] zugewiesen werden. In diesem Fall wird die Adresse des Arrays zugewiesen.
- einer Variablen vom Typ *int*[ ] darf kein Wert zugewiesen werden, und es ist ein Compiler-Fehler auszulösen.
- einer Variablen vom Typ *void* darf kein Wert zugewiesen werden, und es ist ein Compiler-Fehler auszulösen.

Da unsere Sprache nur diesen eingeschränkten Satz von Variablen-Typen bietet, ist diese Liste von Regeln zur Behandlung von Typfehlern recht kurz. Im Unterschied zu ANSI C, handelt es sich bei unserer Sprache allerdings um eine typsichere Sprache, da alle Typfehler spätestens bereits zur Compile-Zeit erkannt werden können.

2.8 Codeerzeugung für MIPS32-Architektur

MIPS32 ist eine registerbasierte RISC-Architektur, d.h. eine sogenannte Load/Store Registermaschine, mit einer Wortbreite von 32-bit. Alle Register der MIPS32-Maschine haben eine Breite von 32-bit. Dies bedeutet für eine Implementierung der Programmiersprache C auf der MIPS32-Plattform in der Regel, dass der Integer-Datentyp eine Breite von 32-bit hat. Das soll wie in der Übersicht über den Sprachumfang angedeutet für unseren C-Dialekt gelten.

Eines der kennzeichnenden Merkmale dieser RISC-Architektur ist, dass alle Operationen nur auf Registern durchgeführt werden, was bedeutet dass alle Operanden zunächst entweder aus dem Hauptspeicher

oder aus Konstanten (auch Immediates genannt) in Register geladen werden müssen.

Die MIPS32-Architektur erlaubt generell den Speicherzugriff auf 1-, 2- und 4-byte Speicherbereiche. Die folgenden Erläuterungen beziehen sich hingegen nur auf die 4-byte Speicherbereiche. Die MIP32-Architektur kann 4-byte Speicherzugriffe nur an Wortgrenzen durchführen (word aligned memory access), und ein Zugriff erfasst immer ein gesamtes Wort. Da Speicheradressen für unsere Maschine byte-genau angegeben werden müssen, ist es für den Speicherzugriff an Wortgrenzen wichtig, dass eine Speicheradresse ein Vielfaches von 4 ist. Sollte dies nicht der Fall sein für eine Speicheradresse, löst der Prozessor eine Exception aus und bricht die Programmausführung ab. **Hinweis:** Da wir Speicheradressen byte-genau für den Prozessor angeben müssen, und alle (Adress)register der Maschine eine Breite von 32-bit haben, kann maximal ein Speicher von 4 GiB Größe adressiert werden. Dadurch können wir auch problemlos die Speicheradresse eines Arrays in einer Integer-Variablen speichern, wie im Abschnitt über das Typsystem beschrieben.

Weiterhin zeichnet sich die MIPS-Architektur durch einen Speicherstack aus, der zur Kommunikation zwischen Funktionen, z.B. Funktionsparameterübergabe, genutzt werden kann. Eine allgemeine Übersicht über die MIPS32-Architektur inklusive aller unterstützen Assembler-Befehle sowie einige konkrete Assembler-Programme sind im Projektarchiv angehangen.

2.8.1 Register

Die MIPS32-Architektur verfügt insgesamt über 32 Integer-Register (Gleitkommaregister existieren auch, spielen aber für diesen Integer-basierten Compiler keine Rolle), denen jeweils unterschiedliche Rollen zugeordnet werden. So wird z.B. eine Menge von Registern angegeben, die für die Parameterübergabe benutzt werden sollen oder eine Menge von Registern, die von aufgerufenen Funktionen definitiv gesichert werden müssen bevor sie benutzt werden. Wir werden zur Vereinfachung die Restriktionen aufweichen und den Registern folgende Bedeutungen zuordnen:

- Register 0: dieses Register liefert immer den Wert 0. Diese Register kann, aber muss nicht verwendet werden
- Register 2: dieses Register wird als Rückgabewert einer Funktion verwendet
- Register 4 bis 25: diese Register stehen für Berechnungen zur freien Verfügung
- Register 29 (sp): dieses Register wird als Stackpointer verwendet
- Register 30 (fp): dieses Register wird als Framepointer verwendet
- Register 31: dieses Register beinhaltet die Rücksprungadresse aus einer Funktion

Im Assembler-Code werden die einzelnen Register über ihre Nummer mit vorangestelltem $-Zeichen angesprochen. Der Stackpointer und der Framepointer sollten über $sp bzw. $fp angesprochen werden.

2.8.2 Parameterübergabe / Aufrufkonvention

Eine Aufrufkonvention (calling convention) legt fest, wie die Funktionsparameter an Funktionen übergeben werden. Laut der beigefügten Übersicht über die MIPS32-Architektur sieht die MIPS-Architektur eine Konvention vor, wobei die ersten 4 Parameter in Registern und weitere Parameter über den Stack übergeben werden. Zur Vereinfachung werden wir uns an der 32-bit x86-Architektur orientieren und alle Parameter über den Stack übergeben. Dazu werden die Parameter gemäß C-typischer Konvention in **umgekehrter** Reihenfolge auf den nach unten wachsenden Stack geschrieben bevor der Funktionsaufruf

ausgeführt wird. Der Rückgabewert der aufgerufenen Funktion (callee) wird wie oben beschrieben im Register 2 an die aufrufende Funktion (caller) zurückgegeben.

2.8.3 Funktionsprolog und -epilog

In der Vorlesung wird zu einem späteren Zeitpunkt noch das Konzept von Laufzeitaktivierungsblöcken (frames) von Funktionen vorgestellt, in denen u.a. die lokalen Variablen einer Funktion abgelegt werden. In der MIPS-Architektur wird solch ein Aktivierungsblock auch auf dem Speicherstack angelegt. Zur Kennzeichnung dieses Blocks dient der bereits eingeführte Framepointer. Der Framepointer zeigt dabei immer auf den Beginn der lokalen Daten (siehe Bild 1).

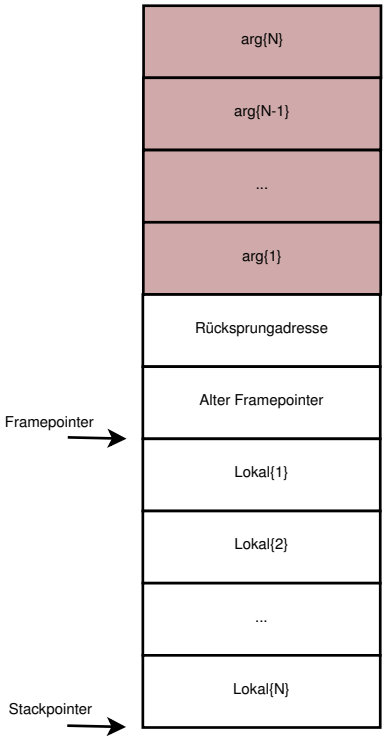

Abbildung 1: Speicherstack

Um Funktionsaufrufe, und speziell rekursive Aufrufe, korrekt zu behandeln muss dieser Framepointer über Funktionsaufrufe hinweg gesichert werden um die jeweiligen Aktivierungsblöcke wiederherzustellen. Bei uns ist dafür immer die aufgerufene Funktion (callee) zuständig. Hinzu kommt die Tatsache, dass die Rücksprungadresse aus einer Funktion über ein Register, Register 31, übergeben wird. Auch dieses Register muss gespeichert werden bevor weitere Funktionsaufrufe durchgeführt werden. Beide Werte, der Framepointer und die Rücksprungadresse, der aufrufendenen Funktion (caller) werden in der aufgerufenen Funktion (callee) auf dem Speicherstack zu Beginn einer Funktion gesichert und zum Ende der Funktion wiederhergestellt bevor die Kontrolle an die aufrufende Funktion wieder zurückgegeben wird.

2.8.4 Lokale Variablen

Wie im vorherigen Abschnitt beschrieben gibt es zu jeder Funktion einen Aktivierungsblock, in dem die lokalen Variablen abgelegt werden. Der Speicherplatz für die lokalen Variablen muss auf dem Speicherstack angelegt werden. Dieser Speicherplatz muss am Ende einer Funktion - bevor die Kontrolle wieder zurückgegeben wird - wieder freigegeben werden. Das Speichermanagement auf dem Stack erfolgt über die Veränderung des Stackpointers: bei der Allokation wird der Stackpointer reduziert, bei der Deallokation wird der Stackpointer wieder auf seinen vorherigen Wert durch eine Addition zurückgesetzt. **Hinweis:**

Der auf dem Stack für einen Funktionsaufruf benötigte Speicher (für Funktionsparameter, Rücksprungadresse, Alter Framepointer und lokale Variablen) ist bereits zur Übersetzungszeit bekannt. Daher kann die Erniedrigung (im Funktionsprolog) und Erhöhung (im Funktionsepilog) des Stackpointers über eine konstante Zahl erreicht werden, und es muss auf dem Stack kein alter Stackpointer gesichert werden.

2.9 Zugriff auf Parameter und lokale Variablen

Auf die Parameter einer Funktion sollte über den Framepointer zugegriffen werden. Auf die lokalen Variablen kann entweder über den Framepointer oder aber den Stackpointer zugegriffen werden. Wenn der Stackpointer verwendet wird, sollte aber bedacht werden, dass der Stackpointer u.U. im Laufe einer Funktion temporär, z.B. für Funktionsaufrufe - erweitert wird.

Gemäß Bild 1 hat der erste Übergabeparameter einen Offset von 8 vom Framepointer, der zweite Parameter einen Offset von 12, usw. Die erste lokale Variable hat bei Verwendung des Framepointers einen Offset von -4, die zweite lokale Variable einen Offset von -8, usw.

2.10 Instruktionsauswahl

Der Schritt Instruktionsauswahl ist ein sehr wichtiger Bestandteil eines Compilers, in dem aus einer Menge von Instruktionsfolgen mit gleicher Semantik die optimalste Instruktionsfolge ausgewählt werden muss. Um die Aufgabe zu vereinfachen werden wir die Auswahl der Instruktionen auf ein Minimum beschränken. Mit Referenz auf das beigefügte Übersichtsblatt sollten zumindest folgende Assembler-Operatoren verwendet werden:

- ADD: Additionen
- ADDI: zum Ausrichten des Stackpointers über Konstanten (Immediates)
- SUB: Subtraktion
- MOVE: zum Verschieben eines Wertes aus einem Register in ein anderes Register (z.B. zum Setzen eines neuen Framepointers)
- SSLV: für die Shift-Left Operation
- SSRV: für die Shift-Right Operation
- MUL: Multiplikation
- BEQZ: bedingter Sprung zu einem Label, wenn Wert gleich 0
- BGEZ: bedingter Sprung zu einem Label, wenn Wert größer oder gleich 0
- BGTZ: bedingter Sprung zu einem Label, wenn Wert größer 0
- BLEZ: bedingter Sprung zu einem Label, wenn Wert kleiner oder gleich 0
- BLTZ: Sprung wenn Wert kleiner 0
- J: unbedingter Sprung zu einem Label
- JAL: Funktionsaufruf (unter Verwendung von Register 31)
- JR: Return aus einer Funktion
- LA: zum Laden einer Adresse eines Arrays
- LI: Laden einer Konstante (Immediate) in ein Register

- LW: Laden eines 4Byte Wertes aus dem Hauptspeicher. Diese Instruktion verlangt immer ein Offset zu einer Basisadresse in einem Register. Der Offset kann auf 0 sein.
- SW: Speichern eines 4Byte Wertes in den Hauptspeicher. Diese Instruktion verlangt immer ein Offset zu einer Basisadresse in einem Register. Der Offset kann auf 0 sein.

2.11 Registerallokation

Die Registerallokation ist ein weiterer wichtiger Bestandteil eines Compilers. Register sind die schnellsten Speicherplätze in einem Computer, aber leider auch die am geringsten vorkommenden Speicherplätze. Um trotz der vorhandenen Registerengpässe (für alle vorkommenden Variablen) einen performanten Code zu erzeugen, muss der Compiler entscheiden welche Variablen zu welcher Zeit in welchem Register vorgehalten werden um so wenig Hauptspeicheroperationen (sehr sehr langsam) wie möglich auszuführen. Da jedoch Registerallokation zu den kompliziertesten Aufgaben eines Compilers gehört, werden wir auf eine effiziente Registerallokation verzichten. Stattdessen werden wir die Registerallokationen naiv durchführen, so dass:

- ein lesender Zugriff **immer** ein Lesen aus dem Hauptspeicher zur Folge hat
- ein schreibender Zugriff **immer** ein Schreiben in den Hauptspeicher zur Folge hat
- jede Konstante stets neu in ein Register geladen wird
- ein Register, dass für eine (temporäre) Variable im Zwischencode verwendet wird, sofort wieder freigegeben wird - d.h. wiederverwendet werden kann - sobald die Variable nicht mehr benötigt wird. Jede temporäre bzw. neu geladene Variable sollte pro Zwischencode-Operation nur 1mal verwendet werden. Somit kann die Registerallokation über eine einfache Register-Variablenzuordnung implementiert werden, die nach jeder Zwischencode-Operation aktualisiert wird

Dieses Schema führt zu einer Reihe von unnötigen Speicherzugriffen und demnach zu nicht-optimaler Performance der Applikation, aber dies wollen wir zur Vereinfachung der Code-Erzeugung durchaus vernachlässigen. Ein Beispiel für die Verwendung der Register für die Berechnungen ist im Projektarchiv angehangen.

2.12 Symboltabelle

Die globale Symboltabelle für den MIPS-Assembler ist sehr einfach aufgebaut. Es wird für jede globale Variable ein Label mit dem Namen der Variablen angelegt. Nach dem Variablennamen folgt dann die Spezifikation der Variablen: der Speicherinhalt der Variablen durch die `.word` Direktive. Im Fall der .word Direktive gibt es zwei Varianten, einerseits `.word <Val>` und andererseits `.word <Val> : <Anzahl>` um einen Speicherbereich zu initialisieren. Die zweite Form kann z.B. für Arrays verwendet werden. Alle globale Variablen sollten mit dem Wert 0 initialisiert werden.

2.13 Assembler-Format

In einer MIPS-Assembler-Datei gibt es zwei Hauptbereiche: einen Datenbereich, d.h. die globale Symboltabelle, und einen Textbereich, d.h. der ausführbare Code. Der Datenbereich wird durch eine `.data` Direktive und der Textbereich durch eine `.text` Direktive eingeleitet. Wie schon angedeutet, Variablen- und Funktionsnamen werden durch Labels aufgeführt.

Neben den Labels für Funktionen und Variablen gibt es auch noch Labels für Sprungziele, z.B. als Ziel eines If-Then-Else-Blocks. Die Syntax für diese Labels unterscheidet sich nicht von den Labels für Funktionen und Variablen. Hier sollte ein einheitliches Schema, wie z.B. _L#, gewählt werden. Dass der

Labelname mit einem Variablen- oder Funktionsnamen kollidieren könnte, kann vernachlässigt werden. Wir stützen uns hierbei auf den C99-Standard, der Bezeichner als reserviert deklariert, die mit einem Unterstrich gefolgt von einem Großbuchstaben anfangen (7.1.3 Reserved Identifiers).

3 Allgemeine Projekthinweise

3.1 Projektstruktur

Wir haben ein Projektarchiv mit Source-Code und Dokumentationen bereitgestellt. Den einzelnen Ordnern kommt dabei folgende Bedeutung zu:

- *bin*: während des Übersetzungsprozesses werden hier alle erzeugten Dateien, d.h. Objektdateien, generierte C-Dateien und ausführbare Dateien, abgelegt.
- *docs*: hier liegen weiterführende Dokumentation, wie Manuals oder Beispiel-Assembler-Dateien für MIPS32.
- *doxygen*: in diesem Verzeichnis wird die Doxygen-Dokumentation erzeugt. Die darin enthaltenen *config*-Datei sollte niemals gelöscht werden.
- *src*: hier soll der Quellcode des Compilers abgelegt werden.
- *test*: dieses Verzeichnis stellt eine kleine Testumgebung für euren Compiler bereit. In dem Unterverzeichnis *rt* liegt eine minimale C-Bibliothek (siehe unten) und eine Make-Datei, die einen minimalen Compilertreiber mit Linker implementiert. Um euren Compiler zu testen, wechselt in das *test* Verzeichnis und ruft **make 'file'.asm** auf wobei 'file' mit dem Namen (ohne Endung) der Quelldatei entspricht. Wenn alles funktioniert, sollte eine 'file'.asm Datei im Unterverzeichnis *out* erzeugt werden. Diese Datei kann dann mit dem MARS-Simulator (*Mars.jar*) geladen und getestet werden.

3.2 Optionen

Der Compiler enthält als einzige Eingabe die zu übersetzende Eingabedatei. Als Ausgabe sollen eine Datei erzeugt werden mit der .s Endung. Diese Datei soll sowohl die globale Symboltabelle wie auch den Textbereich enthalten.

Des weiteren kann als Debugging-Hilfe eine Option zum Ausgeben des Zwischencodes inkl. Symboltabelle (-p) mitgegeben werden. Der Compiler soll den Zwischencode und/oder die Symboltabelle dann in eine Datei mit Endung .ir schreiben.

3.3 Entwicklungswerkzeuge

Ein Compiler sollte immer als Teil einer ganzen Kette von Entwicklungswerkzeugen betrachtet werden. Neben dem Compiler wichtige Werkzeuge sind weiterhin der Linker und der Assembler. Die Aufgabe des Assemblers ist den Assemblercode, der in der Regel im ASCII-Format vorliegt, in den entsprechenden Binärcode des Prozessors zu transformieren. In unserer Umgebung wird der Assembler durch den MIPS-Simulator MARS dargestellt, der den Assemblercode in ASCII-Format ausführen kann.

Die Aufgabe des Linkers ist, mehrere ausführbare Module, in der Regel mehrere Objekt-Dateien (.o), zu einer ausführbaren Datei zusammenzuführen. Dies dient letztendlich der Modularisierung des Entwicklungsprozesses, d.h. für eine erneute Übersetzung der Applikation müssen nur diejenigen Module neu übersetzt werden, die sich seit dem letzten Erstellen der Applikation verändert haben.

3.3.1 Compilertreiber

In der Compilerpraxis ist es üblich neben dem eigentlich Compiler noch ein weiteres Werkzeug bereitzustellen, den sogenannten Compilertreiber. Der Compilertreiber ist ein Werkzeug, dass dem Benutzer eine einheitliche Schnittstelle für den gesamten Entwicklungsprozess zur Verfügung stellt. Der Compilertreiber erfüllt dabei folgende Aufgaben:

- Aufruf des eigentlichen Compiler mit Eingabedatei. Für jede übergebenen Eingabedatei wird der Compiler in der Regel ein Mal aufgerufen.
- Aufruf des Assemblers mit den Ausgabedateien des Compilers
- Aufruf des Linkers mit allen Ausgabedateien des Assemblers

3.3.2 Bibliotheken

Neben den Entwicklungswerkzeugen spielen auch die für eine Programmiersprache verfügbaren Bibliotheken eine außerordentliche Rolle um effizient sinnvolle Programme zu entwickeln. Der allgemeinen C-Bibliothek (C library) kommt dabei eine sehr zentrale Bedeutung zu. Wir haben eine äußerst minimale Bibliothek (unter *test/rt/libc.s*) mit 3 Funktionen für die Kommunikation mit dem Betriebssystem bereitgestellt:

- **print**: Ausgabe eines einzelnen int-Wertes
- **scan**: Einlesen eines einzelnen int-Wertes
- **exit**: Beenden des Programms mit entsprechendem Exit-Code

Gewöhnlicherweise werden die Signaturen der C-Bibliotheksfunktionen in Header-Dateien ausgeliefert und dann in das Programm eingebunden. Da unser Compiler keine include-Direktiven unterstützt, müssen die Prototypen dieser Funktion manuell zu jedem Programm hinzugefügt werden. Für ein Beispiele, siehe *test/condsum.c*.

Literatur

[ALSU07] Aho, Lam, Sethi, and Ullman. *Compilers - Principles, Techniques, & Tools.* Addison Wesley, 2007.

[KH02] Greg Kroah-Hartman. Documentation/codingstyle and beyond. In *In Proceedings of the Ottawa Linux Symposium*, 2002.

[Muc97] Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.