Einführung

Im Modul Compilerbau werden Sie für die prozedurale Programmiersprache SPL einen Compiler entwickeln. Die Zielplattform des Compilers ist der Eco32, ein RISC-Prozessor, der von Prof. Geisse für die Lehre entwickelt wurde. Das bedeutet, dass die Programme, die Ihr Compiler erzeugt nicht einfach so auf Linux, Windows oder MacOS ausgeführt werden können. Stattdessen wird das übersetzte Programm mit dem Eco32-Simulator ausgeführt, der mit einer virtuellen Maschine wie z.B. der NJVM aus dem Modul 'Konzepte systemnaher Programmierung' vergleichbar ist. Einen ähnlichen Ansatz verfolgt zum Beispiel auch der Java-Compiler, der Code für die Java Virtual Machine (JVM) erzeugt.

Dieser Ordner enthält den Eco32-Simulator sowie weitere Werkzeuge, die Sie für die Ausführung von SPL-Programmen benötigen. Außerdem ist ein Referenzcompiler enthalten, den Sie benutzen können, um SPL-Programme zu übersetzen. Ihr Ziel des Semesters ist es, einen eigenen Compiler mit dem gleichen Funktionsumfang des Referenzcompilers zu entwickeln.

Falls sie einfach nur ein SPL-Programm ausführen wollen, lesen Sie den Abschnitt zum vereinfachten Bauen und Ausführen. Etwas detaillierter werden die enthaltenen Werkzeuge und wie sie benutzt werden im Abschnitt Werkzeuge beschrieben. Enthalten sind außerdem die Quelldateien für die SPL-Laufzeitbibliothek. Falls Sie sich dafür interessieren, finden sie im letzten Abschnitt dieser Datei mehr Informationen. Um den Compiler zu verstehen, den Sie im Laufe des Semesters entwickeln werden, empfehlen wir mindestens das Lesen des Abschnitts über den Referenzcompiler. Außerdem sollten Sie die folgenden Sätze zu den Systemvorraussetzungen lesen.

Systemvorraussetzungen

Die Werkzeuge dieses Ordners können auf Linux- und MacOS-Systemen benutzt werden. Windows-Nutzer können das Subsystem für Linux installieren und verwenden: Anleitung

MacOS

Die Programme liegen als universelle Binaries vor, sodass sie sowohl auf älteren Macs als auch auf aktuellen M1-Modellen ausführbar sein sollten.

Es kann dazu kommen, dass die Ausführung der Werkzeuge durch Ihr Betriebssystem blockiert wird. Die Programme sind nicht von Apple zertifiziert, sodass sie standardmäßig blockiert werden. Um die Programme verwenden zu können, müssen Sie sie in den Einstellungen freigeben.

TL/DR: Vereinfachtes Bauen und Ausführen

Um ein SPL-Programm soweit zu übersetzen, dass es mit dem Eco32-Simulator ausgeführt werden kann, sind mehrere Schritte notwendig. Damit Sie diese Schritte nicht immer wieder manuell durchführen müssen, stehen mehrere Hilfsskripte zur Verfügung. Das sind Bash-Skripte, die Sie über die Kommandozeile ausführen können.

Bauen mit `compile.sh`

Mit dem beiligenden Skript `compile.sh` können SPL-Programme übersetzt werden. Beim Aufruf des Skripts wird der Pfad zu einem SPL-Programm angegeben: `./compile.sh prog.spl` Auch relative sind Pfade möglich: `./compile.sh ../tests/prog.spl`

Das Skript ruft den Referenzcompiler, den Assembler, den Linker und den Lader auf. Die Ergebnisse der einzelnen Schritte werden unter den folgenden Dateiendungen im Ordner des SPL-Programms abgelegt:

Zwischenergebnis	Erklärung
`prog.s`	Der vom Compiler erzeugte Assemblercode in einem menschenlesbaren Textformat.

Zwischenergebnis	Erklärung
`prog.o`	Der vom Assembler erzeugte Objektcode in einem Binärformat.
`prog.x`	Das vom Linker erzeugte Executable inklusive der Bibliotheksprozeduren und dem Startupcode. Ebenfalls ein Binärformat.
`prog.bin`	Das vom Lader erzeugte Binärformat, das mit dem Simulator ausgeführt werden kann.

Argumente an `compile.sh` werden an den Referenzcompiler weitergegeben, sodass Sie auch dessen Optionen verwenden können.

Ausführen mit `run.sh`

Das Skript `run.sh` führt eine angegebene, ausführbare Binärdatei mit dem Eco32-Simulator aus. Dafür muss die die `.bin`-Datei als Kommandozeilenargument angegeben werden!

Der Aufruf dieses Skripts für das SPL-Programm `prog.spl` lautet `./run.sh prog.bin`. Dafür muss das Programm vorher mit `compile.sh` übersetzt worden sein. Denken Sie auch daran, die Übersetzung nach jeder Änderung am Programm zu wiederholen.

All-in-One Lösung `splrun.sh`

Häufig wollen Sie ein Programm nach einer Änderung kompilieren und direkt ausführen. Dafür liegt das Skript `splrun.sh` bei, das die beiden vorherigen Skripte kombiniert. Auch diesem Skript geben Sie den Pfad eines SPL-Programms als Parameter an: `./splrun.sh prog.spl` Zusätzliche Argumente werden ähnlich wie bei `compile.sh` an den Referenzcompiler weitergegeben.

Hinweis: Wenn Sie gegen Ende des Semesters Ihren selbstentwickelten Compiler testen wollen, indem Sie SPL-Programme übersetzen und ausführen, könnte es sich lohnen eine Kopie von `compile.sh` (und `splrun.sh`) anzulegen, in der der Aufruf des Referenzcompilers durch einen Aufruf Ihres eigenen Compilers ersetzt ist.

Werkzeuge

Die mitgelieferten Werkzeuge sind der Referenzcompiler, ein Assembler, ein Linker, ein Loader, sowie der Eco32-Simulator. Die Programme liegen im Unterordner `bin`. Dieser Abschnitt liefert Hintergrundinformationen zu den Werkzeugen, die zwar interessant sein dürften, für das Compilerbaupraktikum aber nicht unbedingt relevant sind. Auch wenn Sie den Rest überspringen sollten Sie zumindest den ersten Abschnitt über den Referenzcompiler lesen. Das wird Ihnen bei der Entwicklung Ihres eigenen Compilers enorm helfen.

Die angegeben Befehle zum Aufrufen der Werkzeuge gehen davon aus, dass das Arbeitsverzeichnis der Ordner ist, in der auch dieses ReadMe liegt. Wollen Sie die Tools aus einem anderen Verzeichnis heraus aufrufen, müssen Sie den Aufrufpfad entsprechend anpassen.

Referenzcompiler

Der SPL-Compiler heißt `refspl`. Seine Aufgabe ist es, SPL-Programme zu Eco32-Assemblercode zu übersetzen. Im Laufe des Semesters werden Sie einen eigenen SPL-Compiler schreiben, der ebenfalls Eco32-Assemblercode erzeugt. Ihr Ziel ist es, das Verhalten des Referenzcompilers nachzubilden.

Der generelle Aufruf des Compilers sieht in der Kommandozeile so aus: `./bin/refspl prog.spl prog.s`. Der erste angegebene Dateiname ist eine `.spl`-Eingabedatei. Sie enthält das SPL-Programm, das der Compiler übersetzen soll.

Der zweite Dateiname bestimmt die Ausgabedatei, in die der Compiler den erzeugten Assemblercode schreibt. Üblich ist für Assemblercodedateien die Dateiendung `.s`. Assemblercode ist ein menschenlesbares Textformat. Das bedeutet, dass

Sie die erzeugte Datei mit einem Texteditor öffenen und anschauen können.

Für beide Dateien können relative oder absolute Dateipfade angegeben werden, z.B. `../tests/prog.spl` oder `~/prog.s`. Die Ausgabedatei kann auch weggelassen werden. In diesem Fall wird der erzeugte Assemblercode über die Standardausgabe auf die Kommandozeile ausgegeben.

Der Referenzcompiler kann die Zwischenergebnisse nach den einzelnen Compilerphasen ausgeben. Dies können Sie nutzen, um Ihren eigenen Compiler zu testen, indem Sie die Ausgaben des Referenzcompilers mit den Ausgaben Ihres eigenen Compilers vergleichen Die Grundgerüste, die Ihnen für die Entwicklung des Compilers zur Verfügung gestellt werden, verfügen über dieselben Ausgabefunktionen. Wenn Ihr Compiler bei Verwendung der gleichen Optionen mit mehreren geeigneten Testprogrammen die identische Ausgabe wie der Referenzcompiler erzeugt, können Sie sicher sein, die Aufgabe korrekt gelöst zu haben.

Die Optionen für die einzelnen Phasen sind in dieser Tabelle aufgeführt:

Option	Bedeutung
`--tokens`	Führt die lexikalische Analyse mit dem Scanner aus (PA1). Die erkannten Tokens werden zeilenweise mit Positionsangaben ausgegeben.
`--parse`	Führt die syntaktische Analyse mit dem Parser aus (PA2). Es wird lediglich ausgegeben, ob die Eingabe ein syntaktisch korrektes SPL-Programm ist.
`--absyn`	Führt ebenfalls die syntaktische Analyse aus. Der intern erzeugte abstrakte Syntaxbaum (PA3) wird ausgegeben.
`--tables`	Führt Teil 1 der Semantikanalyse aus, in der eine Symboltabelle aufgebaut wird (PA4.a). Die Symboltabelle wird ein einer von Menschen lesbaren Form ausgegeben.
`--semant`	Führt die vollständige Semantikanalyse aus (PA4.b). Gibt aus, ob das SPL-Programm frei von Semantikfehler wie z.B. Typfehlern ist.
`--vars`	Führt die Variablenallokation aus (PA5). Für jede Prozedur werden mit ASCII-Art eine grafische Darstellung des Stack-Layouts und andere Eckdaten des Stackframes ausgegeben.

Bei einem Aufruf des Compilers kann maximal eine dieser Optionen verwendet werden. Der Compiler führt dann alle Phasen inklusive der Phase aus, dessen Textausgabe angefordert wurde. Soll der gesamte Compiler inklusive Codegenerierung ausgeführt werden, geben Sie keine dieser Optionen an.

Assembler

Der Eco32-Assembler übersetzt menschenlesbaren Eco32-Assemblercode in ein binäres Objektcodeformat. Objektcodedateien enden üblicherweise mit der Dateiendung `.o`. Der ausführbare Assembler heißt `as`.

Der Aufruf der Assemblers sieht so aus: `./bin/as prog.s prog.o`. Die angegebenen Dateien stellen Ein- und Ausgabedatei dar. Auch hier können beliebige Dateipfade verwendet werden, sodass Ihre Testdateien auch in einem anderen Ordner liegen können.

Linker

Ein Linker bindet eine oder mehrere Objektcodedateien zu einem kompletten Executable. Er wird deswegen auch gelegentlich Binder genannt. Linker ermöglichen es, Ihren Code auf mehr als eine Datei aufzuteilen oder vorkompilierte Bibliotheken zu Ihrem Programm dazuzubinden. Dafür werden Referenzen auf die Definitionen externer Module in Objektcodedateien durch sogenannte symbolische Adressen dargestellt, die der Linker durch eine echte Adresse ersetzt.

In SPL ist die Aufteilung eines Programms in mehrere Module nicht möglich. Für den Eco32 muss dennoch ein Linker verwendet werden. Neben dem durch den SPL-Compiler erzeugten Code, müssen zwei weitere Teile dazugebunden werden:

1. Der sogenannte Startupcode, der den echten Einstiegspunkt des Programms enthält. Seine Aufgabe ist es, zur Laufzeit den Stackpointer und den Framepointer einzurichten und anschließend die Main-Prozedur des SPL-Programms aufzurufen. Außerdem ist hier der Code enthalten, der die sogenannten Interrupts des Eco32 behandelt. Falls Sie mit dem Begriff Interrupt noch nichts anfangen können, reicht es sich das ganze wie eine Art Exception auf Hardwareebene vorzustellen. Der Startupcode liegt in Form einer bereits mit dem Assembler übersetzten Objektcodedatei `lib/start.o` vor. Der menschenlesbare Assemblercode des Startupcodes kann in der Datei `lib/sources/start.s` eingesehen werden.
2. Die Implementierungen der Bibliotheksprozeduren von SPL. Für die vordefinierten Prozeduren von SPL (`readi`, `printi` ...) muss natürlich ebenfalls Code vorhanden sein. Er wird in Form der ebenfalls bereits übersetzten Bibliotheksdatei `lib/libsplrts.a` zur Verfügung gestellt. Auch diese Bibliothek liegt als Assemblercode in den Dateien mit der `.s`-Endung in `lib/sources` vor.

Der Eco32-Linker liegt als Datei `./bin/ld` vor. Inklusive der beiden genannten Dinge lautet der Linkeraufruf für ein eigenes SPL-Programm so:

```
./bin/ld -s lib/stdalone.lnk -Llib -o prog.x lib/start.o prog.o -lsplrts
```

Eine Erklärung der einzelnen Bestandteile des Aufrufs zeigt diese Tabelle:

Bestandteil	Bedeutung
`-s lib/stdalone.lnk`	Gibt dem Linker eine Datei an, in der Anweisungen für den Bindeprozess zu finden sind. In diesem Fall sind Anweisungen dafür enthalten, ein vollständiges ausführbares Executable zu bauen. Die angegebene Datei ist ein menschenlesbares Textformat. Wie genau ihr Inhalt zu verstehen ist, ist jedoch auch dem Autor dieser Erklärung unbekannt und für Sie als Compilerbauer vermutlich auch zweitrangig.
`-Llib`	Gibt dem Linker das Verzeichnis `lib` als Bibliotheksverzeichnis an, in dem er Bibliotheken sucht.
`-o prog.x`	Legt `prog.x` als Ausgabedatei fest.
`lib/start.o prog.o`	Eine Auflistung aller Objektcodedateien, die der Linker zusammenbinden soll.
`-lsplrts`	Weist den Linker an, die Bibliothek mit dem Namen `splrts` als Bibliothek zum erzeugten Programm dazuzubinden. Der Linker findet die Bibliothek im Unterverzeichnis `lib`, da dieses Verzeichnis zuvor mit `-Llib` als Bibliotheksverzeichnis angegeben wurde.

Loader

Der Loader (Lader) ist der letzte Schritt in der Übersetzung eines SPL-Programms zu einem ausführbaren Programm. Das Laden eines Programms ist normalerweise Aufgabe des Betriebssystems und nimmt einige letzte Anpassungen des Adressraums eines Programms vor. SPL-Programme werden bei uns jedoch auf dem Simulator ohne Betriebssystem ausgeführt, sodass dieser Schritt von einem zusätzlichen separaten Werkzeug durchgeführt wird. Dieses Werkzeug liegt als `bin/load` vor.

Ein Aufruf des Loaders sieht so aus: `./bin/load prog.x prog.bin`

Die erste Datei ist die Eingabedatei und ist ein vom Linker erzeugtes Executable. Die zweite Datei ist die Ausgabedatei, die mit dem Eco32-Simulator ausgeführt werden kann. Die übliche Dateiendung für ausführbare Eco32-Dateien ist `.bin`.

Simulator

Das letzte Werkzeug in der Kette ist der Eco32-Simulator. Er wird verwendet, um fertig übersetzte, gebundene und geladene Programme auszuführen. Der Simulator simuliert den vollständigen Eco32-Prozessor sowie seine verfügbaren Hardwarekomponenten. Er tut damit viel mehr als wir im Compilerbau benötigen.

Ein einfacher Aufruf zur Ausführung eines SPL-Programms sieht so aus: `./bin/sim -l prog.bin -x -s 1 -stdt 0`

Bestandteil	Erklärung
`-l prog.bin`	Gibt dem Simulator an, welche Binärdatei er in den Speicher laden soll bevor die Ausführung beginnt. Hier geben Sie also die vom Lader erzeugte Datei (normalerweise `.bin`) an.
`-x`	Sorgt dafür, dass der Simulator nach Ausführung des SPL-Programms (also nach Ende von `main`) ordnungsgemäß beendet wird.
`-s 1`	Installiert für den Hardwaresimulator genau eine serielle Schnittstelle. Einfach ausgedrückt bedeutet dies, dass das Programm damit Ein- und Ausgaben verarbeiten kann.
`-stdt 0`	Verbindet die zuvor angelegte serielle Schnittstelle mit der Standardeingabe und Standardausgabe Ihres Terminals. Vereinfacht gesagt: Damit können Sie Eingaben für z.B. `readi` und `readc` machen sowie die Ausgaben von `printi` oder `printc` auch wirklich in der Kommandozeile sehen.

Normalerweise verfügt der Simulator außerdem über eine Grafikausgabe. Aufgrund von Kompatibilitätsproblemen ist dies jedoch derzeit deaktiviert. Das bedeutet, dass die SPL-Grafikfunktionen (`clearAll`, `setPixel`, `drawLine` und `drawCircle`) derzeit nicht verwendet werden können.

Archiver

Ein weiteres Tool, das in der normalen Übersetzungs-Ausführungskette jedoch nicht verwendet wird, ist der Archiver (`ar`). Der Archiver kann aus mehreren Objektcodedateien (`.o`) eine sogenannte statische Bibliothek bauen. Sie hat üblicherweise die Dateiendung `.a` und ist ein nicht-menschenlesbares Binärformat. Eine statische Bibliothek enthält ähnlich wie Objektcodedateien Maschinencode, der zu einem Programm durch den Linker dazugebunden werden kann.

Im Unterordner `lib` liegt ein Beispiel für eine solche Datei (`libsplrts.a`). Sie enthält die Definitionen der vordefinierten SPL-Prozeduren. Das Skript `lib/sources/build.sh` kann verwendet werden, um die statische Bibliothek aus den Assemblercodedateien zu bauen.

Die SPL Laufzeitbibliothek

Neben der vorgestellten Werkzeugen ist auch die SPL-Laufzeitbibliothek in diesem Ordner enthalten. Sie liegt im Unterordner `lib`. Die Laufzeitbibliothek enthält die in SPL verfügbaren vordefinierten Prozeduren, wie z.B. `readi`, `printi`, `setPixel` oder `clearAll`.

Eine bereits mit dem Archiver als statische Bibliothek gebaute Fassung der Laufzeitbibliothek ist die Datei `libsplrts.a`. Außerdem liegen die menschenlesbaren Assemblercodequellen im Unterorder `lib/sources` vor. Sie zu Verstehen erfodert ein bisschen Übung im Umgang mit Eco32-Assemblercode, kann aber auch selbst eine ganz gute Übung sein. Schauen Sie gerne in die Quelldateien rein und schauen sich ein bisschen um.

Die Datei `lib/sources/start.s` enthält den sogenannten Startupcode für Ihr SPL-Programm. Der übersetzte Objektcode dieser Datei ist nicht Teil der statischen Laufzeitbibliothek, sondern wird vom Linker als Einstiegspunkt in das Programm verwendet. Unter dem Label `start` finden Sie dort den Code, der den Laufzeitstack initialisiert, eine Startnachricht auf der Kommandozeile ausgibt, und anschließend die `main`-Prozedur des Programms aufruft.