Praktikum 6 Compilerbau WS14/15 Testat bis 13.01.2015

Ziele:
- Übersetzung von Mini-Java-FunProc-Programmen in M32-Assembler.

Aufgabe: (Übersetzen von Mini-Java-FunProc-Programmen in M32-Assembler)
Implementieren Sie die Übersetzung von Mini-Java-FunProc-Programmen in M32-Assemblercode. Beachten Sie dabei, dass das Auslösen einer Ausnahme zum Abbruch führt und nicht abgefangen wird.

Sie müssen sich sonst an keine Vorgaben halten.
Zur Unterstützung werden jedoch einige Tipps gegeben. Dabei werden einige Vorschläge zur Lösung von Problemen gemäß der Implementierung des Compilers gegeben, die auf dem Netz verfügbar ist.

Tipps:
1.) Symboltabellen
Es werden analog zur Vorlesung folgende Symboltabellen implementiert:

- eine globale Symboltabelle für das Hauptprogramm,
  die im jj-File als globale Variable allen Übersetzungsfunktionen zur Verfügung gestellt wird.

  Diese enthält neben den Einträgen für Konstanten und Variablen, jetzt auch noch Einträge für Funktionen und Prozeduren, die neben einem Tag, das für Funktionen und Prozeduren verschieden sein muss, noch die Anzahl der Parameter enthält.
  Im Gegensatz zur Vorlesung wird die Startadresse nicht abgespeichert, da sie sich symbolisch aus dem Namen der Funktion bzw. Prozedur ergibt.
  Außerdem sind die Initialwerte der lokalen Variablen nicht in der Symboltabelle vorhanden, da diese von der Prozedur selbst auf den Stack gelegt werden.
- eine lokale Symboltabelle pro Funktion bzw. Prozedur,
  die als Parameter der Übersetzungsfunktionen weitergegeben wird.

  Diese enthält neben den Einträgen für Konstanten noch Einträge für lokale Variablen und Parameter. Dabei wird jetzt jedoch nicht die absolute Adresse im Hauptspeicher abgelegt, sondern die Position im aktuellen Block auf dem Stack.

Um mehrmalige Implementierungen von Übersetzungsfunktionen zu vermeiden, kann man ggf. auch die globale Symboltabelle als Parameter übergeben, wenn man sich im Hauptprogramm befindet. Hierzu ist es notwendig, eine Oberklasse oder ein Interface für die beiden Symboltabellenklassen zu definieren, die dann als Parameter übergeben werden.
Im Folgenden ist ein mögliches Interface angegeben, das die Vereinigung aller Methoden der beiden Symboltabellen-Klassen ist.
D.h., dass manche Methoden in beiden Klassen implementiert sein können. Andere jedoch nur in einer der beiden Klassen gebraucht werden.

```
/**
 * Dieses Interface stellt für eine Symboltabelle notwendige Methoden zur
 * Verfügung, die jedoch abhängig von der Symboltabelle (lokale oder globale)
 * unterschiedlich implementiert werden müssen. <br>
 * <br>
 * <i>Anm.: </i>
 * <ul>
 * <li>Die Methoden addProcedure/addFunction müssen nur von der globalen
 * Symboltabelle implementiert werden.
 * <li>Die Methode addParam nur von der lokalen.
 * </ul>
*
 * @author Oliver Vesper
 */
```

Praktikum 6 Compilerbau WS14/15 Testat bis 13.01.2015

```
public interface InterST {
  /**
   * Liefert den Wert Y eines Symboltabellen-Eintrags (X,Y) zurück
   *
   * @param id
   *          Bezeichner
   * @return Wert Y
   * @throws UnknownSymbolException
   */
  public String getSymbol(String id) throws UnknownSymbolException;

  /**
   * Trägt eine Konstante in die Symboltabelle ein
   *
   * @param id
   *          Bezeichner
   * @param wert
   *          Wert der Konstanten
   * @throws SymbolAlreadyDefinedException
   */
  public void addConstant(String id, String wert)
      throws SymbolAlreadyDefinedException;

  /**
   * Trägt eine Variable in die Symboltabelle ein
   *
   * @param id
   *          Bezeichner
   * @param wert
   *          Initialisierungs-Wert der Variablen
   * @throws SymbolAlreadyDefinedException
   */
  public void addVariable(String id, String wert)
      throws SymbolAlreadyDefinedException;

  /**
   * Trägt einen Parameter in die Symboltabelle ein <br>
   * <i>Anm.: </i> Muss nur für lokale Symboltabelle implementiert werden
   *
   * @param id
   *          Bezeichner
   * @throws SymbolAlreadyDefinedException
   */
  public void addParam(String id) throws SymbolAlreadyDefinedException;

  /**
   * Trägt eine Prozedur inkl. der zu erwartenden Anzahl von Parametern in die
   * Symboltabelle ein <br>
   * <i>Anm.: </i> Muss nur für globale Symboltabelle implementiert werden
   * werden
   *
   * @param name
   *          Name der Prozedur
   * @param anzParam
   *          Anzahl der erwarteten Parameter
```

Praktikum 6 Compilerbau WS14/15 Testat bis 13.01.2015

```
   * @throws SymbolAlreadyDefinedException
   */
  public void addProcedure(String name, int anzParam)
      throws SymbolAlreadyDefinedException;

  /**
   * Trägt eine Funktion inkl. der zu erwartenden Anzahl von Parametern in die
   * Symboltabelle ein <br>
   * <i>Anm.: </i> Muss nur für globale Symboltabelle implementiert werden
   *
   * @param name
   *          Name der Funktion
   * @param anzParam
   *          Anzahl der erwarteten Parameter
   * @throws SymbolAlreadyDefinedException
   */
  public void addFunction(String name, int anzParam)
      throws SymbolAlreadyDefinedException;

  /**
   * Überprüft, ob es sich bei dem Bezeichner um eine Konstante handelt
   *
   * @param id
   *          Bezeichner
   * @return <code>true</code>, wenn Bezeichner existiert und eine Konstante
   *         ist; sonst <code>false</code>
   */
  public boolean isConstant(String id);

  /**
   * @param id
   *          Bezeichner
   * @return <code>true</code>, wenn Bezeichner existiert und eine Variable
   *         ist; sonst <code>false</code>
   */
  public boolean isVariable(String id);
}
```

2.) zusätzliche Exceptions

- RWertException
  falls es sich bei einem Bezeichner auf der rechten Seite einer Zuweisung nicht um den Bezeichner einer Variablen, einer Konstanten oder einer Funktion handelt.
- WrongParametersException
  falls es sich um die falsche Anzahl der Parameter einer Funktion oder Prozedur handelt.

3.) Übersetzung von Funktionen und Prozeduren

Funktionen und Prozeduren können so übersetzt werden, dass das aufrufende Programm die Parameterauswertung auf dem Stack durchführt und die aufgerufene Prozedur dann die Initialwerte der lokalen Variablen auf den Stack legt und mit der Bearbeitung beginnt.

4.) Verwendung eines einzigen Stacks als Prozedur- und Datenkeller

Da der M32-Assembler nur einen Stack anbietet, ist es sinnvoll den Prozedur- und den Datenkeller zusammenzulegen. Ein möglicher Ansatz dazu findet sich in Übungsaufgabe 52.

Praktikum 6 Compilerbau WS14/15 Testat bis 13.01.2015

Da zwischen zwei Blöcken noch Elemente des Datenkellers liegen können, wird dabei kein dynamic link verwendet, der die Anzahl der Elemente des Blockes enthält, sondern die absolute Adresse des Kopfes des darunter liegenden Blockes in einem Aktivierungsblock auf dem Stack abgespeichert. Diese Adressen und auch die Rücksprungadressen müssen ggf. in Registern zwischengespeichert werden.

5.) Verwendung der M32-Befehle CALL & RET

Sie können die M32-Befehle CALL & RET verwenden. Dabei ist zu berücksichtigen, dass die Rücksprungadresse automatisch auf den Stack gelegt wird, bzw. vom Stack in den Befehlszähler geschrieben wird.