

Projekt Codeerzeugung

Ziel dieses Projekts ist die praktische Umsetzung und Vertiefung der in der Vorlesung vermittelten Kenntnisse auf den Gebieten

- Symbolistenverwaltung
- Codeerzeugung für die Intel-Architektur
- Umgang mit dem Compiler-Generator Coco/R

Die Bearbeitung dieses Projekts ist freiwillig und kann auch während der Sommerferien erfolgen. Wer das Projekt bearbeitet, verbessert damit automatisch seine Note in der Abschlussklausur um einen Grad (die beste Note in der Abschlussklausur ist "gut"). Der Abgabetermin und die Abgabemodalitäten stehen auf der Webseite dieser Lehrveranstaltung. Als Programmiersprache kann C#, Java oder C/C++ verwendet werden.

Programmiersprache SL

Gegeben sei eine einfache Sprache namens SL (simple language):

```
SL= "PROGRAM" {Declaration} "BEGIN" StatSeq "END" ". ".
Declaration = VarDecl | ProcDecl.
VarDecl = "VAR" {IdList ":" Type ","}.
IdList = ident {" "," ident}.
Type = ident.
ProcDecl = "PROCEDURE" ident [Parameters] ";" {VarDecl} ["BEGIN" StatSeq] "END" ident ";".
Parameters = "(" [Param {" "," Param}] ")" [ ":" Type].
Param = ["VAR"] IdList ":" Type.
StatSeq = Statement {" "," Statement}.
Statement =
    [ ident (" :=" Expression | ActParameters )
      | "IF" Condition "THEN" StatSeq {"ELSIF" Condition "THEN" StatSeq} ["ELSE" StatSeq] "END"
      | "WHILE" Condition "DO" StatSeq "END"
      | "RETURN" [Expression]
    ].
Condition = Expression Relop Expression.
Expression = [Addop] Term {Addop Term}.
Term = Factor {Mulop Factor}.
Factor = ident [ActParameters] | number | charCon | "(" Expression)".
ActParameters = "(" [Expression {" "," Expression}])".
Relop = "=" | "#" | "<" | ">" | ">=" | "<="".
Addop = "+" | "-".
Mulop = "*" | "/" | "%".
```

Als Datentypen sind nur die vordefinierten Typen INTEGER (4 Byte) und CHAR (1 Byte) erlaubt. CHAR-Konstanten werden durch ein Zeichen unter Hochkommas (z.B. "x") ausgedrückt. Kommentare werden zwischen die Klammern /* und */ geschrieben. Die Standardprozedur *Put(expr)* gibt den Ausdruck *expr* vom Typ CHAR auf der Konsole aus, die Standardprozedur *PutLn* bewirkt einen Zeilenvorschub. Die Standardfunktion *ORD(ch)* wandelt das Zeichen *ch* in einen INTEGER-Wert um, *CHR(i)* wandelt die Zahl *i* in einen CHAR-Wert um.

Beispielprogramm in SL

```
PROGRAM

  VAR i: INTEGER;

  PROCEDURE PutInt(x: INTEGER); /* groesste ausgebbare Zahl = 9999 */
    VAR c0, c1, c2, c3: CHAR;
  BEGIN
    c3 := CHR(48 + x % 10); x := x / 10;
    c2 := CHR(48 + x % 10); x := x / 10;
    c1 := CHR(48 + x % 10); x := x / 10;
    c0 := CHR(48 + x % 10);
    IF c0 > "0" THEN Put(c0); Put(c1); Put(c2)
    ELSIF c1 > "0" THEN Put(c1); Put(c2)
    ELSIF c2 > "0" THEN Put(c2)
    END;
    Put(c3)
  END PutInt;

  BEGIN /* ungerade Zahlen ausgeben */
    i := 1;
    WHILE i < 100 DO
      PutInt(i); PutLn;
      i := i + 2
    END
  END.
END.
```

Falls Ihnen der Aufwand zu groß wird, können Sie die Behandlung von Prozeduren aus der Sprache SL streichen. Seien Sie aber ehrgeizig und versuchen Sie auch die Prozeduren zu bewältigen.

a) Symbollistenverwaltung

Schreiben Sie eine attributierte Grammatik *SL.ATG*, die mit dem Compiler-Generator Coco/R verarbeitet werden kann. Das Benutzerhandbuch und die Quellen von Coco/R (für Java, C# und C++) finden Sie unter:

<http://www.ssw.uni-linz.ac.at/Coco/>

Coco/R erzeugt in der C#-Version aus *SL.ATG* einen Parser (*Parser.cs*) und einen Scanner (*Scanner.cs*). Schreiben Sie ein Hauptprogramm *SL.cs*, das etwa wie folgt aussehen soll (Details siehe Beschreibung von Coco/R):

```
public class SL {  
  
    public static void Main (string[] args) {  
        string file = ... source file name ...;  
        Scanner scanner = new Scanner(file);  
        Parser parser = new Parser(scanner);  
        parser.Parse();  
        Console.WriteLine(parser.errors.count + " errors detected");  
    }  
  
}
```

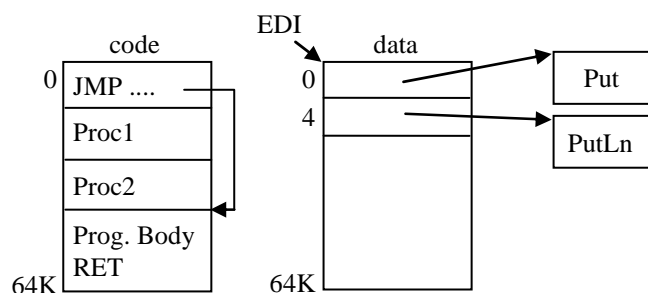
Fügen Sie in *SL.ATG* semantische Aktionen ein, die unter anderem folgendes leisten sollen:

- Aufbau einer Symbolliste mit Objekt- und Strukturknoten. Erzeugen Sie vordefinierte Objekt- und Strukturknoten für INTEGER und CHAR sowie für die Standardprozeduren Put, PutLn, ORD und CHR.
- Prüfung der nötigen Kontextbedingungen in Deklarationen und Anweisungen (z.B. keine Doppeldeklarationen, alle verwendeten Namen deklariert, Zuweisungskompatibilität, etc.).

b) Codeerzeugung

Erweitern Sie SL.ATG so, dass Schritt haltend mit der Syntaxanalyse Maschinencode erzeugt wird. Der Maschinencode soll ohne zusätzliche Daten (wie z.B. Vorspann oder Linker-Informationen) in eine Datei gespeichert werden. Die erste Instruktion muss allerdings ein Sprung auf den Programmrumpf sein.

Benutzen Sie den Lader, den Sie auf der Homepage als Quellcode und als Executable vorfinden, um den Maschinencode auszuführen. Der Lader legt Speicherbereiche für Code und Daten gemäß folgender Skizze an, lädt den Programmcode und springt mittels CALL-Instruktion auf Adresse 0 des geladenen Codes. Am Ende des Programms soll die Programmausführung mittels RET wieder zum Lader zurückkehren.



Ferner ist folgendes zu beachten:

- *Adressgröße:* Der Maschinencode wird im 32-Bit-Modus ausgeführt, alle Adressen und Stack-Slots sind 4 Byte groß. Sie brauchen sich nicht um die Segmentregister zu kümmern.
- *Globale (statische) Daten:* Der Lader legt den globalen Datenbereich (64 KB) an und lädt seine Adresse ins Register EDI. Die ersten 8 Byte des globalen Datenbereichs enthalten die Adressen von *Put* und *PutLn*, die eigentlichen globalen Daten beginnen deshalb bei [EDI + 8]. Stellen Sie sicher, dass das Register EDI nicht anderweitig verwendet wird!
- *Adressierung:* Globale Daten werden relativ zu EDI adressiert, lokale Daten relativ zu EBP. Sprünge sollten immer relativ erfolgen.
- *Stack:* Sie können davon ausgehen dass der Stack und ESP bereits vom Lader korrekt initialisiert sind. Der Stack wächst in Richtung niedrigere Adressen.
- *Ausgaben:* Der Lader stellt die Standardprozeduren *Put* und *PutLn* zur Verfügung. Ihre Adressen stehen in den ersten beiden Doppelworten des globalen Datenbereichs, sodass die Prozeduren folgendermaßen aufgerufen werden können:
 - *Put*(ch): CALL [EDI] 0xFF 0x17
 - *PutLn*() CALL [EDI + 4] 0xFF 0x57 0x04

Beachten Sie, dass die beiden Prozeduren sämtliche Register ändern können. Sie müssen sie also zumindest EDI über den Aufruf hinweg retten, z.B. mit (PUSH EDI, ..., POP EDI).

Um den erzeugten Code zu debuggen, verwenden Sie z.B. Visual Studio oder eine andere Entwicklungsumgebung mit Disassembly-Anzeige und gehen Sie im Single-Step-Modus durch die Instruktionen.