Project 1: Code Generation for a Register Machine

The goal of this project is to turn the concepts learned in this course into a practical compiler. The project covers

- Symbol table managament
- Code generation for register machines
- Use of a compiler generator

The project is optional, but without the project the best possible mark for this course will be 2 (gut). If you submit a correct project (and if you achieve at least 40 points in the written exam) the mark from the exam will be automatically raised by 1. The submission date for the project can be found on the web page of this course. The implementation language for the project can be Java, C# or C/C++.

The Programming Language SL (simple language)

Syntax:

```
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 = "*" | "/" | "%".
```

Data types

INTEGER	4 bytes; constants like 123
CHAR	1 byte; constants like 'x'

Comments delimited by /* and */

Bult-in procedures

Put(e)	prints the expression *e* which is of type CHAR to the console
PutLn	starts a new line
ORD(ch)	converts the character *ch* to an integer
CHR(i)	converts the integer *i* to a character

Sample Program in SL

```
PROGRAM

    VAR i: INTEGER;

    PROCEDURE PutInt (x: INTEGER); /* largest printable number = 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 /* print odd numbers */
    i := 1;
    WHILE i < 100 DO
        PutInt(i); PutLn();
        i := i + 2
    END
END.
```

If the implementation of the full compiler is too much for you, you can drop the implementation of procedures. But be ambitious and try to implement everything.

a) Symbol Table Management

Write an attributed grammar *SL.ATG* that can be processed by the compiler generator Coco/R. The user manual, the executable and the sources of Coco/R (for Java, C# and C++) can be found at http://ssw.jku.at/Coco/ .

The C# version of Coco/R, for example, generates a scanner (*Scanner.cs*) and a parser (*Parser.cs*) from *SL.ATG*. You have to write a main program SL.cs which should look like this (for details see the user manual of 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");
    }

}
```

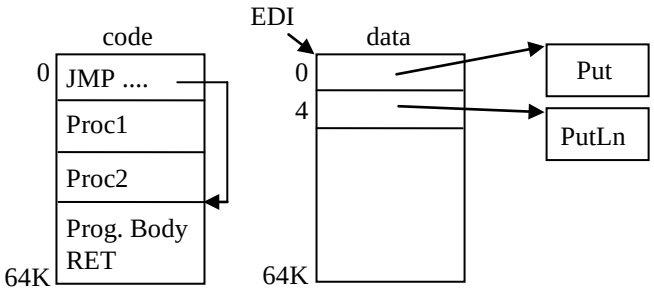
Add semantic actions to *SL.ATG* to perform the following tasks:

- Build a symbol table with *Obj* and *Struct* nodes. Initialize the universe with predefined *Obj* and *Struct* nodes for INTEGER and CHAR as well as for the built-in procedures *Put*, *PutLn*, *ORD* and *CHR*.
- Check the necessary context conditions in declarations and statements (e.g., all names must be declared, there must not be double declarations, assignments and parameter passing must obey the usual type compatibility rules).

b) Code Generation

Extend SL.ATG such that IA32 code is generated during parsing. The generated code should simply be written into a file (without header or linker information). The first instruction, however, must be a jump to the program's body where execution should start.

Use the mini loader (downloadable from the web page of this course) to execute the generated machine code. The loader allocates memory for the code and the data according to the figure below. After loading the code it does a CALL to address 0 of the loaded code. At the end of the program's body there should be a RET instruction that causes the program to return to the loader.



Please note also the following:

- *Addresses*: The code will be executed in 32-bit mode. Thus, all addresses are 4 bytes in size. Also, you do not have to care for segment registers.
- *Global (static) data*: The loader allocates a global data area (64 KB) and loads its address into register EDI. The first 8 bytes of the global data area hold the addresses of the built-in procedures *Put* and *PutLn*. Therefore, the actual global data start at [EDI+8]. Make sure that EDI is not used otherwise in your code. Operands of kind *Abs* should be treated like operands of kind *RegRel* with EDI as the base register.
- *Addressing*: Global data are accessed relative to EDI, local data relative to EBP. Jumps should always be done with relative jump distances.
- *Stack*: The loader allocates the stack and initializes the stack pointer ESP. The stack grows towards lower addresses.
- *Output*: The loader provides the built-in procedures *Put* and *PutLn*. Their addresses are installed into the first two words of the global data area so that they can be invoked in the following way:

  - Put(ch): `CALL [EDI]` `0xFF 0x17`
  - PutLn(): `CALL [EDI + 4]` `0xFF 0x57 0x04`

  Note, that both procedures might modify registers. Thus, you have to save at least EDI across such invocations (e.g., with PUSH EDI, ... POP EDI).

In order to debug the generated code you can use Visual Studio or some other IDE with disassembly functionality.