FH-OÖ Hagenberg/ESD
Compiler Engineering, WS 2021
Projekt Codeerzeugung

Name: Abgabetermin: 31.01.2022

Mat.Nr: Punkte:

Ziel dieses Projektes ist die praktische Umsetzung und Vertiefung der in der Theorie vermittelten Kenntnisse auf den Gebieten

- Lexikalische Analyse (Scanner)
- Syntaktische Analyse (Parser)
- Attributierte Grammatiken
- Semantische Analyse
- Symbollistenverwaltung
- Zwischencodeerzeugung
- Codeerzeugung
- Umgang mit dem Compiler-Generator Coco/R.

Programmiersprache MiniIEC

Gegeben sei eine einfache Sprache namens MIEC in folgender Darstellung:

MIEC → **PROGRAM** *ident*
(*VarDecl*)?
BEGIN
Statements
END

VarDecl → **BEGIN_VAR**
ident **: Integer ;** (*ident* **: Integer ;**)*
END_VAR

Statements → *Stat* (*Stat*)*
Stat → *ident* **:=** *Expr* **;**
| **print** (*Expr*) **;**

| **WHILE** *Condition* **DO** *Statements* **END**
| **IF** *Condition* **THEN** *Statements* **END**
| **IF** *Condition* **THEN** *Statements* **ELSE** *Statements* **END**

Expr → *Term* (**+** *Term*)*
Expr → *Term* (**-** *Term*)*

Term → *Fact* (***** *Fact*)*
Term → *Fact* (**/** *Fact*)*

Fact → *ident*
Fact → *number*
Fact → **(***Expr***)**

Condition → *Expr Relop Expr*
Relop → **=**
Relop → **<=**
Relop → **>=**
Relop → **!=**
Relop → **<**
Relop → **>**

Als Datentyp ist in der ersten Ausbaustufe nur der Typ Ganzzahl (Integer mit 2 Byte) erlaubt. Kommentare werden zwischen (* und *) geschrieben. Die Standardprozedur `print` gibt den Ausdruck `expr` auf der Konsole aus und bewirkt einen Zeilenvorschub. Die Namen im Fettdruck sind sogenannte Schlüsselwörter.

Beispielprogramm in MIEC

Jedes MIEC-Programm befindet sich in einer separaten Datei mit der Erweiterung `*.miec`.

```
1  PROGRAM Hello
2    BEGIN_VAR
3      a: Integer;
4      b: Integer;
5    END_VAR
6  BEGIN
7    a := 3 * 6 + (2 * 3);
8    WHILE b < a DO
9      b := b + 1;
10   END
11   IF a > b THEN
12     print(a);
13   ELSE
14     print(b);
15   END
16 END
```

Übung1: Scanner und Parser

Das Ziel der ersten Übung besteht darin, einen Scanner (Lexer) und einen Parser soweit vorzubereiten und zu generieren, dass dieser MIEC-Programme übersetzen kann und syntaktische Fehler erkennt. Als Compiler-Generator wird Coco/R verwendet, machen Sie sich dazu mit der Dokumentation auf der Kommunikationsplattform vertraut (*Übersicht_CocoR.pdf* und *CocoR_Tutorial*). Zusätzlich steht ein Beispiel `Taste.zip` zur Verfügung. Führen Sie anschließend folgende Implementierungsschritte durch:

1. Installation des *MS Visual Studio*-MIECCompiler-Projektes.
2. Schreiben Sie eine attributierte Grammatik `MIEC.atg` entsprechend der Programmiersprache MIEC, die mit dem Compiler-Generator Coco/R verarbeitet werden kann. Coco erzeugt in der C++-Version einen Parser (`Parser.h`, `Parser.cpp`) und einen Scanner (`Scanner.h`, `Scanner.cpp`).

   Aufruf von Coco/R:

   ```
   Coco.exe <ATGFilename.atg> -o <DirOfGeneratedFiles> -namespace <CompilerNamespace> -frames <DirOfFrameFiles>
   ```

3. **Aufruf des Compilers:**

   ```
   MIECCompiler.exe -in <file.miec> -out <file.iex>
   ```

   Die Aufrufschnittstelle ist unbedingt einzuhalten, um ein Zusammenspiel mit der automatischen Testplattform (ACOTEP) zu gewährleisten. Übergeben Sie dem MIECCompiler den Pfad der Quelldatei (`FileName.miec`) und der Zieldatei (`FileName.iex`) über die Kommandozeile. Prüfen Sie die Dateien auf die entsprechende Dateierweiterung `.miec` bzw. `.iex` und rufen Sie im Anschluss den Scanner und Parser auf.
4. **Ergebnisdatei:** Der Compiler erzeugt eine Datei `MIECCompiler.report` im ASCII-Format, die Informationen über den Compiliervorgang speichert. Existiert die Datei nicht, wird sie erzeugt, existiert sie, wird das Ergebnis an die bestehende Datei angehängt! Für einen Compiliervorgang wird der Name des Compilers (entspricht der exe-Datei) eingetragen und für jede übersetzte Datei im Fehlerfall die Anzahl der Fehler. Folgendes Beispiel zeigt das genaue Dateiformat:

   ```
   Thu Sep 12 12:31:17 2019 => ..\SourceFiles\xfail_Fehler_noVar.miec: FAILED: 1 error(s) detected
   Thu Sep 12 13:01:04 2019 => ..\SourceFiles\Mul.miec: OK
   Thu Sep 12 13:06:35 2019 => D:\fh-hagenberg\Projekt_Codeerzeugung\src\WhileTest.miec: OK
   ```

5. Schreiben Sie verschiedene MIEC-Testprogramme und testen Sie den MIECCompiler ausführlich! Testdateien, die bei der Übersetzung einen Fehler liefern sollen, müssen mit dem Präfix "xfail" beginnen (Grund: ACOTEP).

Übung2: Semantikanschluss und Symboltabelle

Das Ziel der zweiten Übung ist, den MIECCompiler so weit zu erweitern, dass für alle Deklarationen entsprechende Symbole und Typen erzeugt werden, die miteinander so verkettet sind, dass keine Informationen verloren gehen. Die Symboltabelle stellt die Basis für die Zwischencodeerzeugung dar. Dazu fügen Sie in die `MIEC.atg` Attribute und semantische Aktionen ein und führen folgende Implementierungsschritte durch:

1. Erzeugung von Symbol- und Typknoten für alle Variablen und Typen.
2. Abbildung von numerischen Konstanten.
3. Aufbau einer Symboltabelle, die alle deklarierten Variablen, Typen und Konstanten speichert.
4. Prüfung der nötigen Kontextbedingungen in Deklarationen und Anweisungen:
   - Doppeldeklarationen sind nicht erlaubt.
   - Alle verwendeten Namen (Variablen) müssen deklariert sein.
   - Zuweisungskompatibilität: Typprüfung bei Zuweisung oder Vergleich von Variablen und Konstanten
5. Führen Sie eine Offsetberechung für die deklarierten Variablen durch. Jede Variable speichert ihren Offset den sie später im Datensegment einnehmen wird.

Hinweis: Die Symboltabelle wird in der ATG folgendermaßen inkludiert und deklariert:

```
#include "SymbolTable.h"

COMPILER MIEC

   SymbolTable mSymTab;

   // helper methods
   // ...

CHARACTERS
  ...
TOKENS
  ...
```

Durch diese Deklaration wird die Symboltabelle als Attribut in der Klasse `Parser` erzeugt, und somit kann direkt in den semantischen Aktionen auf die Symboltabelle zugegriffen werden.

Übung3: Aufbau einer Zwischendarstellung

Das Ziel der dritten Übung ist, die Grammatik mit semantischen Aktionen zu versehen, so dass für alle Anweisungen des Quelltextes eine entsprechende Zwischendarstellung (Drei-Adress-Code-Konstrukte) im Speicher aufgebaut wird, die als Basis für die Maschinen-Codeerzeugung dient. Die DAC-Konstrukte werden in Form einer *Tripel-Darstellung* (siehe Folienskript) gespeichert. Führen Sie dazu folgende Implementierungsschritte durch:

1. Als Schnittstelle für die Zwischencodegenerierung soll eine Klasse `DACGenerator` dienen. Sie stellt Methoden zur Verfügung, die DAC-Anweisungen erzeugen. Als Parameter dienen Operatoren und Symbole die entsprechend verknüpft werden und so im Speicher eine Abbildung der Anweisungen des Quelltextes darstellen.

2. Eine einzelne DAC-Anweisung wird durch eine Klasse `DACEntry` abgebildet und besteht aus einem Operator und zwei Argumenten, die wieder durch Symbole dargestellt werden.

3. Sprünge in einer Schleifen- oder Bedingungsanweisung können durch einen Verweis auf die entsprechende Zielanweisung abgebildet werden. Die Zielanweisung ist jene Anweisung, die abhängig von der Bedingung ausgeführt wird.

4. Erweiterung der ATG um semantische Aktionen die DAC-Anweisungen mit Hilfe des DAC-Generators erzeugen und in einem entsprechenden Container speichern.

Hinweis: Der `DACGenerator` wird in der ATG folgendermaßen inkludiert und deklariert:

```
#include "DACGenerator.h"

COMPILER MIEC

   DACGenerator mDACGen;

CHARACTERS
  ...
TOKENS
  ...
```

Durch diese Deklaration wird der `DACGenerator` als Attribut in der Klasse `Parser` erzeugt, und kann direkt in den semantischen Aktionen der ATG verwendet werden und den DAC-Code entsprechend erzeugen.

Die Operatoren im `DACEntry` können durch folgende Enumeration abgebildet werden:

```
enum class OpKind {
      eAdd, eSubtract, eMultiply, eDivide, eIsEqual, eIsLessEqual, eIsGreaterEqual,
      eIsNotEqual, eIsLess, eIsGreater, eAssign, eJump, eIfJump, eIfFalseJump, ePrint,
      eExit
};
```

Übung4: Codeerzeugung

Erzeugen Sie aus der Zwischendarstellung Maschinencode für den PROL16. Der Maschinencode wird ohne zusätzliche Daten in eine Datei (`*.iex`) gespeichert, da in MIEC nur temporäre Daten möglich sind, die nicht initialisierbar sind.

Der Generator für den Maschinencode PROL16 (`CodeGenProl16.h` und `CodeGenProl16.cpp`) wird zur Verfügung gestellt. Für jeden Befehl des PROL16-Befehlssatzes bietet der Generator eine entsprechende Methode, die den zugehörigen Operationscode erzeugt und in ein Byte-Feld schreibt. Am Ende wird der gesammte Maschinencode mit Hilfe einer Methode `WriteIex(std::ostream& iexFile)` in eine ausführbare Datei geschrieben.

Um aus der Zwischendarstellung entsprechenden Maschinecode erzeugen zu können, sind folgende Implementierungsschritte durchzuführen:

1. Implementierung einer Klasse `CodeGenerator`, die eine Liste der einzelnen DAC-Konstrukte (Zwischendarstellung) speichert und den `CodeGenProl16` verwaltet und verwendet.
2. Eine Methode `CodeGenerator::GenerateCode()` durchläuft die DAC-Konstrukte und erzeugt für jedes Konstrukt mit Hilfe des `CodeGenProl16` den entsprechenden PROL16-Code. Die Variablen am Datenspeicher werden nicht automatisch vorinitialisiert, dafür ist entsprechender Maschinencode zu erzeugen (siehe Beispiel Maschinencode für Multiplikation und Division).
3. Schreiben Sie einen Register-Administrator der die Register des PROL16 verwaltet. Wahlweise stellt der PROL16 8 oder 16 Register zur Verfügung. Wir beschränken uns in unserer Implementierung auf 8 Register.

Ein Register-Administrator verwaltet die Register des PROL16. Er speichert in einer Liste die Register die während der Codeerzeugung benutzt werden bzw. frei sind. Da nur eine bestimmte Anzahl an Registern zur Verfügung steht, kann nur eine bestimmte Anzahl an temporären Zwischenergebnissen in Registern gespeichert werden. Wird die Anzahl an möglichen Zwischenergebnissen während der Codeerzeugung überschritten, so müssen die temporären Werte auf den Registerspeicher (Stack) ausgelagert werden, um Register für weitere Berechnungen frei zu bekommen. Werden die ausgelagerten Werte in Folgeanweisungen wieder benötigt, so sind sie wieder in ein entsprechendes Register zu laden.

Die Werte von Variablen werden in der ersten Ausbaustufe nicht am Registerspeicher zwischengesichert!

Virtuelle Maschine für den PROL 16

Benutzen Sie die virtuelle Maschine VMProl16, die auf der Kommunikationsplattform zur Verfügung gestellt wird, um den Maschinencode auszuführen. Die VMProl16 legt Speicherbereiche für Code und Daten fest, lädt den Programmcode (iex-Datei) und führt ihn entsprechend aus.

Aufruf der VM:

```
VMProl16 ExecuteableFile.iex
```

Implementierung des Code-Generators

Aus zeitlichen Gründen werden die Schnittstelle des Code-Generators und die Implementierung der Division und der Multiplikation zur Verfügung gestellt. Der PROL16 stellt keine Instruktionen für Multiplikation und Division bereit, deshalb müssen diese beiden Operationen via Additions-, Subtraktions- und Shift-Operationen implementiert werden.

Die Methode `GenerateCode(...)` bekommt die ausführbare Datei und durchläuft den gesamten Zwischencode. Abhängig von der Zwischencode-Anweisung ist der entsprechende Maschinencode mithilfe des Maschinencode-Generators (`CodeGenProl16`) zu erzeugen. Die Codeerzeugung für die einzelnen Operationen ist in privaten Methoden gekapselt, deren erster Parameter immer der entsprechende `DACEntry` des Zwischencodes ist.

```
1   class CodeGenerator {
2   public:
3      ...
4
5      void GenerateCode(std::ostream& os);
6
7   private:
8        typedef std::list<std::pair<WORD, DACEntry const*> > TUnresolvedJumps;
9
10     void OperationAdd(DACEntry* apDacEntry);
11     void OperationSubtract(DACEntry* apDacEntry);
12     void OperationMultiply(DACEntry* apDacEntry);
13     void OperationDivide(DACEntry* apDacEntry);
14     void OperationAssign(DACEntry* apDacEntry);
15     void OperationJump(DACEntry* apDacEntry, TUnresolvedJumps& arUnresolvedJumps);
16     void OperationConditionalJump(DACEntry* apDacEntry, TUnresolvedJumps& arUnresolvedJumps);
17     void OperationPrint(DACEntry* apDacEntry);
18
19     //private members
20     CodeGenProl16* mpGenProl16;
21     RegisterAdmin* mpRegAdmin;
22     ...
23  };
```

Implementierung des Multiplikationsoperators

```
////////////////////////////////////////////////////////////////////////
// Multiplication by shift
//
//    result = 0
//    while (multiplier != 0)
//    {
//       multiplier = multiplier >> 1
//       if (carry != 0)
//       {
//          result += multiplikand
//       }
//       multiplikand = multiplikand << 1
//    }
//
////////////////////////////////////////////////////////////////////////
 void CodeGenerator::OperationMultiply(DACEntry* pDacSym) {

   //prepare multiplicand und multiplier
   //multiplicand -> loads value to register
   RegNr regA = mpRegAdmin->GetRegister(pDacSym->GetFirstOperand());
   RegNr regB;
   if (pDacSym->GetFirstOperand() == pDacSym->GetSecondOperand()) { //multiplicand = multiplier
      regB = mpRegAdmin->GetRegister();
      mpGenProl16->Move(regB,regA);
   }
   else {
      //multiplier -> loads value to register
      regB = mpRegAdmin->GetRegister(pDacSym->GetSecondOperand());
   }

   //generate code for jump of while-statement
   RegNr regJmp = mpRegAdmin->GetRegister();     //used for jumps
   RegNr regResult = mpRegAdmin->GetRegister();  //will contain result
   mpGenProl16->LoadI(regResult, 0);             //init result register

   //stores acutal codeposition (begin of while)
   WORD codePosStart = mpGenProl16->GetCodePosition();
   //init general purpose register für compare
   mpGenProl16->LoadI(mpRegAdmin->cGeneralPurposeRegister, 0);
   //compare multiplier with 0
   mpGenProl16->Comp(regB, mpRegAdmin->cGeneralPurposeRegister);

   //load value 0 to jump register for now and store code-position in jumpData1
   //-> jump address is resolved later (end of while-statement)
   WORD jumpData1 = mpGenProl16->LoadI(regJmp, 0);
   mpGenProl16->JumpZ(regJmp);  //jump to address in jump-register, if zero-flag is set

   mpGenProl16->ShR(regB);  //code for operation -> multiplier = multiplier >> 1

   //generate code for if-statement
   //load value 0 to jump register for now and store code-position in jumpData2
   //-> jump address is resolved later (if-statement => false)
   WORD jumpData2 = mpGenProl16->LoadI(regJmp, 0);
   mpGenProl16->JumpC(regJmp);  //jump to address in jump-register, if carry-flag is set

   mpGenProl16->ShL(regA);  //mulitplicand = multiplicand << 1

   mpGenProl16->LoadI(regJmp, codePosStart);  //load jump address -> begin of while
   mpGenProl16->Jump(regJmp);                 //jump to begin of while-statement

   //end of if -> resolve and set jump address to codeposition jumpData2
   mpGenProl16->SetAddress(jumpData2, mpGenProl16->GetCodePosition());

```

```
64        mpGenProl16->Add(regResult, regA);          //result += multiplicand
65        mpGenProl16->ShL(regA);                     //mulitplicand = multiplicand << 1
66
67        mpGenProl16->LoadI(regJmp, codePosStart);  //load jump address -> begin of while
68        mpGenProl16->Jump(regJmp);                  //jump to begin of while-statement
69
70        //end of while -> resolve and set jump address to codeposition jumpData1
71        mpGenProl16->SetAddress(jumpData1, mpGenProl16->GetCodePosition());
72
73        //regResult contains result of multiplication -> assign to DAC-symbol
74        mpRegAdmin->AssignRegister(regResult, pDacSym);
75
76        // free all other registers
77        mpRegAdmin->FreeRegister(regA);
78        mpRegAdmin->FreeRegister(regB);
79        mpRegAdmin->FreeRegister(regJmp);
80    }
```

Beispiel: Einfache Multiplikation

Das folgende Beispiel zeigt den Quellcode für eine einfache Multiplikation und den generierten Maschinencode:

Quellcode:

```
PROGRAM Hello
  BEGIN_VAR
    a: Integer;
     b: Integer;
     c: Integer;
  END_VAR
BEGIN
  a := 5 * 3;
  print(a);
END
```

PROL16 Maschinencode:

```
0000: loadi r1, 0x0000        // lade den Initialisierungswert 0 nach r1
0002: loadi r2, 0x0000        // lade die Adresse der Variablen a nach r2
0004: store r1, r2            // initialisiere die Variable a mit dem Wert 0

0005: loadi r1, 0x0005        // lade die Konstante 5 nach r1
0007: loadi r2, 0x0003        // lade die Konstante 3 nach r2
0009: loadi r4, 0x0000        // initialisiere das Ergebnis-Register

000B: loadi r0, 0x0000        // initialisiere r0 (general purpose register)
000D: comp  r2, r0            // multiplier wird mit 0 verglichen
000E: loadi r3, 0x001e        // lade die Sprungadresse ins Sprungregister
0010: jumpz r3                // springe, aus der while-Schleife, wenn zero flag gestzt ist

0011: shr   r2                // multiplier = multiplier >> 1

0012: loadi r3, 0x0019        // lade die Sprungadresse ins Sprungregister
0014: jumpc r3                // springe, in das if-Statement, wenn carry flag gesetzt ist
0015: shl   r1                // multiplicand = multiplicand << 1

0016: loadi r3, 0x000b        // lade die Sprungadresse
0018: jump  r3                // springe zum Beginn der while-Schleife

0019: add   r4, r1            // result += multiplicand

001A: shl   r1                // multiplicand = multiplicand << 1

001B: loadi r3, 0x000b        // lade die Sprungadresse
001D: jump  r3                // springe zum Beginn der while-Schleife

001E: loadi r0, 0x0000        // lade die Adresse der Variablen a
0020: store r4, r0            // speichere das Ergebnis in a

0021: loadi r1, 0x0000        // lade die Adresse der Variablen a nach r1
0023: load  r1, r1            // lade den Wert von a nach r1
0024: print r1                // ausgeben von a auf der Konsole

0025: sleep                   // Programm-Ende
```

Implementierung des Divisionsoperators

```
////////////////////////////////////////////////////////////////////////////
// Division by shift
//
//    remainder = 0
//    bits = 16
//    do
//    {
//       dividend = dividend << 1
//       remainder = remainder << 1 (with carry bit)
//       if ((carry != 0) || (remainder >= divisor))
//       {
//          remainder -= divisor
//          dividend |= 0x01
//       }
//       bits--
//    } while (bits > 0)
//
////////////////////////////////////////////////////////////////////////////
 void CodeGenerator::OperationDivide(DACEntry* pDacSym) {

   //prepare multiplicand und multiplier
   //dividend -> loads value to register
   RegNr regA = mpRegAdmin->GetRegister(pDacSym->GetFirstOperand());
   //divisor -> loads value to register
   RegNr regB = mpRegAdmin->GetRegister(pDacSym->GetSecondOperand());

   //prepare help variables
   RegNr regJmp = mpRegAdmin->GetRegister();       //used for jumps
   RegNr regRemainder = mpRegAdmin->GetRegister(); //get register for remainder
   mpGenProl16->LoadI(regRemainder, 0);            //initialize remainder
   RegNr regBits = mpRegAdmin->GetRegister();      //bit counter
   mpGenProl16->LoadI(regBits, 16);                //initialize bit counter with 16

   //stores acutal codeposition (begin of do-while)
   WORD codePosStart = mpGenProl16->GetCodePosition();
   mpGenProl16->ShL(regA);                    //dividend = dividend << 1
   mpGenProl16->ShLC(regRemainder);           //remainder = remainder << 1 (with carry bit)

   WORD jumpData1 = mpGenProl16->LoadI(regJmp, 0); //jump address -> end if
   //jump to address, if carry bit is set => carry != 0
   mpGenProl16->JumpC(regJmp);
   //compare remainder and divisor => remainder >= divisor
   mpGenProl16->Comp(regB, regRemainder);
   mpGenProl16->JumpC(regJmp);                //jump to address, if carry bit is set
   mpGenProl16->JumpZ(regJmp);                //jump to address, if zero flag is set

   WORD jumpData2 = mpGenProl16->LoadI(regJmp, 0); //jump address -> while-statement
   mpGenProl16->Jump(regJmp);
   mpGenProl16->SetAddress(jumpData1, mpGenProl16->GetCodePosition());

   //remainder -= divisor
   mpGenProl16->Sub(regRemainder, regB);

   //dividend |= 0x01
   mpGenProl16->LoadI(mpRegAdmin->cGeneralPurposeRegister, 1);
   mpGenProl16->Or(regA, mpRegAdmin->cGeneralPurposeRegister);

   //set jump address to while-statement
   mpGenProl16->SetAddress(jumpData2, mpGenProl16->GetCodePosition());

   //bits--
   mpGenProl16->Dec(regBits);

```

```
64        //jump address for end of do-while
65        WORD jumpData3 = mpGenProl16->LoadI(regJmp, 0);
66
67        //bits >= 0
68        mpGenProl16->JumpZ(regJmp);
69
70        //load jump address for begin of do-while
71        mpGenProl16->LoadI(regJmp, codePosStart);
72        //jump to  begin of do-while
73        mpGenProl16->Jump(regJmp);
74        //set jump address -> end of do-while
75        mpGenProl16->SetAddress(jumpData3, mpGenProl16->GetCodePosition());
76
77        //regA contains result of division -> assign to DAC-symbol
78        mpRegAdmin->AssignRegister(regA, pDacSym);
79
80        //free all other registers
81        mpRegAdmin->FreeRegister(regB);
82        mpRegAdmin->FreeRegister(regJmp);
83        mpRegAdmin->FreeRegister(regRemainder);
84        mpRegAdmin->FreeRegister(regBits);
85    }
```

Beispiel: Einfache Division

Das folgende Beispiel zeigt den Quellcode für eine einfache Division und den generierten Maschinencode:

Quellcode:

```
PROGRAM Hello
  BEGIN_VAR
    a: Integer;
  END_VAR
BEGIN
  a := 15 / 3;
  print(a);
END
```

PROL16 Maschinencode:

```
0000: loadi r1, 0x0000    // Variable a mit 0 initialisieren
0002: loadi r2, 0x0000
0004: store r1, r2

0005: loadi r1, 0x000f    // Konstante mit Wert 15 laden
0007: loadi r2, 0x0003    // Konstante mit Wert 3 laden
0009: loadi r4, 0x0000    // remainder = 0
000B: loadi r5, 0x0010    // bits = 16

                                  // Beginn der do-while
000D: shl   r1            // dividend = dividend << 1
000E: shlc  r4            // remainder = remainder << 1 (with carry bit)
000F: loadi r3, 0x0018    // Sprungadresse fuer if
0011: jumpc r3            // springe ueber das if-statement, wenn carry gesetzt
0012: comp  r2, r4        // remainder >= divisor
0013: jumpc r3            // springe ins if-statement, wenn carry gesetzt
0014: jumpz r3            // springe ins if-statement, wenn zero gesetzt
0015: loadi r3, 0x001c    // Sprungadresse -> Ende if-Statement
0017: jump  r3            // springe ueber das if-Statement
0018: sub   r4, r2        // remainder -= divisor
                          // Ende if-Statement

0019: loadi r0, 0x0001    // r0 = 0x01
001B: or    r1, r0        // dividend |= 0x01
001C: dec   r5            // bits--
001E: loadi r3, 0x0023    // Sprungadresse -> while
001F: jumpz r3            // springe aus dem do-while, wenn bits == 0

0020: loadi r3, 0x000d    // lade Sprungadresse
0022: jump  r3            // springe zu Beginn von do-while

0023: loadi r0, 0x0000    // Adresse von a laden
0025: store r1, r0        // Ergebnis nach a speichern

0026: loadi r1, 0x0000    // Adresse von a laden
0028: load  r1, r1        // Wert von a nach r1 laden
0029: print r1            // den Wert von a auf der Konsole ausgeben

002A: sleep               // Programm-Ende
```

Hinweise zur Abgabe:

- Geben Sie den gesamten Quellcode als VS2019-Projekt ab.
- Das gesammte Projekt ist als MIECCompiler_name1_name2.zip abzugeben. Das zip muss zusätzlich die Realease-Version des MIECCompilers enthalten.
- Kommentieren Sie den Quellcode entsprechend!
- Schreiben Sie entsprechende MIEC-Testdateien und geben Sie diese mit ab.
- Die VM und MIECDevelop sind nicht mitabzugeben.
- Das vollständige Klassendiagramm (Reverse Engineering mit EA) ist als pdf mitabzugeben.
- Die Ausgabe des Compilers (std::cout) beschränkt sich auf folgendes Format:

  ```
  ..\testfiles\ok\mixed\Test2.miec: OK
  ```

  bzw. im Fehlerfall (std::cerr)

  ```
  -- line 14 col 7: invalid Factor
  ..\testfiles\failed\TestSubstract.miec: FAILED: 1 error(s) detected
  ```

- Prüfen Sie die Vollständigkeit der Kommandozeilenparameter (siehe Übung1).
- Prüfen Sie das richtige Format der MIECCompiler.report Datei (siehe Übung1).
- Testen Sie Ihre Abgabe(zip-Datei) mit der automatischen Testplattform ACOTEP.
- Achten Sie auf die Einhaltung folgender Namen:
  - MIECCompiler.exe
  - MIECCompiler_name1_name2.zip
  - MIECCompiler.report
  - xfail_Datei.miec