FH-OÖ Hagenberg/ESD
Compilerbau 3, WS 2015
Projekt Codeerzeugung

FH-HAGENBERG

Name: Abgabetermin: 31.01.2015

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. Übergeben Sie dem MIECCompiler eine oder mehrere Quelldatei(en) (`FileName.miec`) über die Kommandozeile, prüfen Sie die Datei(en) auf die entsprechende Dateierweiterung `.miec` und rufen Sie den Scanner und Parser auf.
4. Der Compiler erzeugt eine Ergebnisdatei `MIECCompiler_result` 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 Comilers (entspricht der exe-Datei) eingetragen und für jede übersetzte Datei im Fehlerfall die Anzahl der Fehler. Folgendes Beispiel zeigt das genaue Dateiformat:

   ```
   MiniIEC.exe
   ..\.\failed\test2_sch.miec: FAILED: 1 error(s) detected
   MiniIEC.exe
   ..\.\crashed_files\test_mayer.miec: OK
   MiniIEC.exe
   ..\.\ok\test_huber.miec: OK
   ..\.\ok\test3.miec: OK
   ```

5. Schreiben Sie verschiedene MIEC-Testprogramme und testen Sie den MIECCompiler ausführlich!

Übung2: Semantikanschluss und Symbollistenverwaltung

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 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::string const& fileName)` 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

```
1  ////////////////////////////////////////////////////////////////////////
2  // Multiplication by shift
3  //
4  //    result = 0
5  //    while (multiplier != 0)
6  //    {
7  //       multiplier = multiplier >> 1
8  //       if (carry != 0)
9  //       {
10 //          result += multiplikand
11 //       }
12 //       multiplikand = multiplikand << 1
13 //    }
14 //
15 ////////////////////////////////////////////////////////////////////////
16 void CodeGenerator::OperationMultiply(DACEntry* apDacEntry)
17 {
18    RegNr regA = mpRegAdmin->GetRegister(apDacEntry->GetArgument1()); // multiplicand
19    RegNr regB = mpRegAdmin->GetRegister(apDacEntry->GetArgument2()); // multiplier
20    RegNr regJmp = mpRegAdmin->GetRegister(); // used for jumps
21    RegNr regResult = mpRegAdmin->GetRegister();  // will contain result
22    mpGenProl16->LoadI(regResult, 0);  //init
23    WORD codePosStart = mpGenProl16->GetCodePosition(); //start of loop begin
24    RegNr helpReg = mpRegAdmin->GetRegister();
25    mpGenProl16->LoadI(helpReg, 0);
26    mpGenProl16->Comp(regB, helpReg);  //compare: multiplier != 0
27    WORD jumpData1 = mpGenProl16->LoadI(regJmp, 0); //stores jump address of while-statement
28    mpGenProl16->JumpZ(regJmp); //jump if zero bit is set (multiplier = 0)
29    mpGenProl16->ShR(regB);  //multiplier = mulitiplier >> 1
30
31    WORD jumpData2 = mpGenProl16->LoadI(regJmp, 0); //stores jump address of if-statement
32
33    //jump if carry bit is set ( multiplier = 3 -> 011 >> 1 = 001 (carry bit = 1) )
34    mpGenProl16->JumpC(regJmp);
35    mpGenProl16->ShL(regA);  //multiplicand = multiplicand << 1
36    mpGenProl16->LoadI(regJmp, codePosStart); //prepeare jump address
37    mpGenProl16->Jump(regJmp); //jump to begin of while-statement
38    //sets jump address for if-statement
39    mpGenProl16->SetAddress(jumpData2, mpGenProl16->GetCodePosition());
40    mpGenProl16->Add(regResult, regA); //adds multiplicand to result register
41
42    mpGenProl16->ShL(regA);  //multiplicand = multiplicand << 1
43    mpGenProl16->LoadI(regJmp, codePosStart);  //prepeare jump address
44    mpGenProl16->Jump(regJmp);  //jump to begin of while-statement
45
46    //sets jump address for end of while-statement
47    mpGenProl16->SetAddress(jumpData1, mpGenProl16->GetCodePosition());
48
49    // regResult contains result of multiplication -> assign to DAC-Entry
50    mpRegAdmin->AssignRegister(regResult, apDacEntry);
51
52    // free all other registers
53    mpRegAdmin->FreeRegister(regA);
54    mpRegAdmin->FreeRegister(regB);
55    mpRegAdmin->FreeRegister(regJmp);
56    mpRegAdmin->FreeRegister(helpReg);
57 }
```

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
0002: loadi r2, 0x0000
0004: store r1, r2
0005: loadi r2, 0x0002
0007: store r1, r2
0008: loadi r2, 0x0004
000A: store r1, r2
000B: loadi r3, 0x0005
000D: loadi r4, 0x0003
000F: loadi r6, 0x0000
0011: loadi r0, 0x0000
0013: comp  r4, r0
0014: loadi r5, 0x0024
0016: jumpz r5
0017: shr   r4
0018: loadi r5, 0x001f
001A: jumpc r5
001B: shl   r3
001C: loadi r5, 0x0011
001E: jump  r5
001F: add   r6, r3
0020: shl   r3
0021: loadi r5, 0x0011
0023: jump  r5
0024: loadi r0, 0x0000
0026: store r6, r0
0027: loadi r3, 0x0000
0029: load  r3, r3
002A: print r3
002B: sleep
```

Implementierung des Divisionsoperators

```
1  ////////////////////////////////////////////////////////////////////////////
2  // Division by shift
3  //
4  //    remainder = 0
5  //    bits = 16
6  //    do
7  //    {
8  //       [Logic shift left] dividend
9  //       [Shift left] remainder
10 //       if ((carry != 0) || (remainder >= divisor))
11 //       {
12 //          remainder -= divisor
13 //          dividend |= 0x01
14 //       }
15 //       bits--
16 //    } while (bits > 0)
17 //
18 ////////////////////////////////////////////////////////////////////////////
19 void CodeGenerator::OperationDivide(DACEntry* apDacEntry)
20 {
21    RegNr regA = mpRegAdmin->GetRegister(apDacEntry->GetArgument1()); // dividend
22    RegNr regB = mpRegAdmin->GetRegister(apDacEntry->GetArgument2()); // divisor
23    RegNr regJmp = mpRegAdmin->GetRegister(); // used for jumps
24    RegNr regRemainder = mpRegAdmin->GetRegister(); // remainder
25    mpGenProl16->LoadI(regRemainder, 0);
26    RegNr regBits = mpRegAdmin->GetRegister(); // bit counter
27    mpGenProl16->LoadI(regBits, 16);
28    WORD codePosStart = mpGenProl16->GetCodePosition();
29    mpGenProl16->ShL(regA);
30    mpGenProl16->ShLC(regRemainder);
31    WORD jumpData1 = mpGenProl16->LoadI(regJmp, 0);
32    mpGenProl16->JumpC(regJmp);
33    mpGenProl16->Comp(regB, regRemainder);
34    mpGenProl16->JumpC(regJmp);
35    mpGenProl16->JumpZ(regJmp);
36    WORD jumpData2 = mpGenProl16->LoadI(regJmp, 0);
37    mpGenProl16->Jump(regJmp);
38    mpGenProl16->SetAddress(jumpData1, mpGenProl16->GetCodePosition());
39    mpGenProl16->Sub(regRemainder, regB);
40    RegNr helpReg = mpRegAdmin->GetRegister();
41    mpGenProl16->LoadI(helpReg, 1);
42    mpGenProl16->Or(regA, helpReg);
43    mpGenProl16->SetAddress(jumpData2, mpGenProl16->GetCodePosition());
44    mpGenProl16->Dec(regBits);
45    WORD jumpData3 = mpGenProl16->LoadI(regJmp, 0);
46    mpGenProl16->JumpZ(regJmp);
47    mpGenProl16->LoadI(regJmp, codePosStart);
48    mpGenProl16->Jump(regJmp);
49    mpGenProl16->SetAddress(jumpData3, mpGenProl16->GetCodePosition());
50
51    // regA contains result of division -> assign to DAC-Entry
52    mpRegAdmin->AssignRegister(regA, apDacEntry);
53
54    // free all other registers
55    mpRegAdmin->FreeRegister(regB);
56    mpRegAdmin->FreeRegister(regJmp);
57    mpRegAdmin->FreeRegister(regRemainder);
58    mpRegAdmin->FreeRegister(regBits);
59    mpRegAdmin->FreeRegister(helpReg);
60 }
```

Beispiel: Einfache Division

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

Quellcode:

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

PROL16 Maschinencode:

```
0000: loadi r1, 0x0000
0002: loadi r2, 0x0000
0004: store r1, r2
0005: loadi r3, 0x000f
0007: loadi r4, 0x0003
0009: loadi r6, 0x0000
000B: loadi r7, 0x0010
000D: shl   r3
000E: shlc  r6
000F: loadi r5, 0x0018
0011: jumpc r5
0012: comp  r4, r6
0013: jumpc r5
0014: jumpz r5
0015: loadi r5, 0x001c
0017: jump  r5
0018: sub   r6, r4
0019: loadi r0, 0x0001
001B: or    r3, r0
001C: dec   r7
001D: loadi r5, 0x0023
001F: jumpz r5
0020: loadi r5, 0x000d
0022: jump  r5
0023: loadi r0, 0x0000
0025: store r3, r0
0026: loadi r3, 0x0000
0028: load  r3, r3
0029: print r3
002A: sleep
```

Abgabe:

- Geben Sie den gesammten Quellcode als VS2015-Projekt ab.
- 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.