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
};
```