Student/in:		Unterschrift:	
DHBW Duale Hochschule Baden-Württemberg Stuttgart **ÜBUNGSKLAUSUR**		Fakultät Studiengang: Jahrgang / Kurs : Studienhalbjahr:	**Technik** **Angewandte Informatik** **2013 / 13B** **4. Semester**
Datum:	**20. Mai 2015**	Bearbeitungszeit:	**90 Minuten**
Modul: Unit:	**TINF2002.3** **Compilerbau**	Dozent:	**Stephan Schulz**
Hilfsmittel:	**Vorlesungsskript, eigene Notizen**		
Punkte:		Note:	

Aufgabe	erreichbar	erreicht
1	6	
2	6	
3	10	
4	10	
5	8	
6	7	
7	7	
Summe	54	

1. Sind Sie gesund und prüfungsfähig?
2. Sind Ihre Taschen und sämtliche Unterlagen, insbesondere alle nicht erlaubten Hilfsmittel, seitlich an der Wand zum Gang hin abgestellt und nicht in Reichweite des Arbeitsplatzes?
3. Haben Sie auch außerhalb des Klausurraumes im Gebäude keine unerlaubten Hilfsmittel oder ähnliche Unterlagen liegen lassen?
4. Haben Sie Ihr Handy ausgeschaltet und abgegeben?

(Falls Ziff. 2 oder 3 nicht erfüllt sind, liegt ein Täuschungsversuch vor, der die Note „nicht ausreichend“ zur Folge hat.)

Aufgabe 1 (6 Punkte)
Beschreiben Sie kurz 3 wesentliche Punkte, in denen eine Hochsprache sich von einer maschinennahen Sprache unterscheidet.

Lösung (z.B):

- Portabilität - Hochsprachen sind weitgehend unabhängig von der unterliegenden Hardware
- Expressivität - Hochsprachen erlauben es, komplexe Sachverhalte kompakt darzustellen
- Robustheit und Typsystem - Hochsprachen erlauben bessere Abstraktionen und bessere Fehlererkennung

Aufgabe 2 (6 Punkte)
Nennen Sie 3 Aspekte der Syntax einer Programmiersprache, die üblicherweise nicht über die Grammatik beschrieben werden, weil sie kontext-sensitiv sind. Wie werden diese Aspekte üblicherweise behandelt?

Lösung (3 aus 4+X):

- Variablen vor Verwendung definieren
- Typkonsistenz bei Funktionsaufrufen
- Typkonsistenz bei Operatoren
- Konsistenz von Funktiondefinition und Rückgabewert

Behandlung jeweils jeweils per Führen einer Symboltabelle mit Typinformationen und Typüberprüfung während der Semantischen Prüfung.

Aufgabe 3 (10 Punkte)
Das folgende *nanoLang*-Programm enthält 5 Fehler. Identifizieren Sie die einzelnen Fehler und beschreiben Sie, welche Phase des Compilers den Fehler jeweils identifiziert.

```
Integer StrMult(Integer x, String str)
{
   String res;

   res  = '';

   while(x>"0")
   {
      res = StrCat(res, str));
      x = x-+-1;
   }
   return res;
}

Integer main(String count, String str)
{
   print StrMult(StrToInt(count),str);
   print "\n";
   return 0;
}
```

Lösung:

1. `StrMult()` hat Rückgabewert `Integer`, gibt aber String zurück. Semantische Analyse.
2. `''` ist in `nanoLang` illegal. Scanner.
3. Vergleich `x>"0"`. Typüberprüfung in der semantischen Analyse.
4. Extra Klammer hinter `StrCat`. Syntaxanalyse.
5. `-+-` ist keine sinvolle/erlaubte Sequenz. Syntaxanalyse.

Aufgabe 4 (2+4+4 Punkte)
Betrachten Sie die folgende Grammatik: $G = \langle V_N, V_T, P, E\rangle$ mit $V_N = \{E, V, N, R\}$, $V_T = \{x, y, z, 0, 1, *, +, (,)\}$, und folgenden Produktionen in P:

- $E \to (E)$
- $E \to E + E$
- $E \to E * E$
- $E \to V$
- $E \to N$
- $V \to xR|yR|zR|x|y|z$
- $R \to xR|yR|zR|0R|1R|x|y|z|0|1$
- $N \to 0N|1N|0|1$

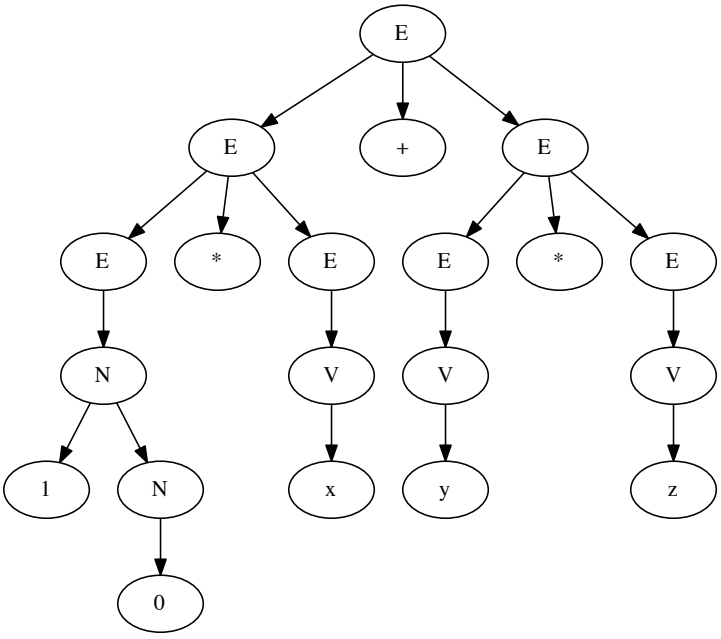
a) Geben Sie eine Rechtsableitung für das Wort $(x0 + 10) * xy1$ an.

b) Generieren Sie eine Ableitung und einen Parse-Tree für das Wort $10 * x + y * z$, der "Punktrechnung vor Strichrechnung" respektiert.

c) Welche Teile der Grammatik sind kontext-frei und welche sind regulär? Wenn Sie einen Parser für G schreiben müssten, welche Token würde Ihr Parser verarbeiten?

Lösung:

a) $E \Longrightarrow E * E$
$\Longrightarrow E * V$
$\Longrightarrow E * xR$
$\Longrightarrow E * xyR$
$\Longrightarrow E * xy1$
$\Longrightarrow (E) * xy1$
$\Longrightarrow (E + E) * xy1$
$\Longrightarrow (E + N) * xy1$
$\Longrightarrow (E + 1N) * xy1$
$\Longrightarrow (E + 10) * xy1$
$\Longrightarrow (V + 10) * xy1$
$\Longrightarrow (xR + 10) * xy1$
$\Longrightarrow (x0 + 10) * xy1$



$E \Longrightarrow E + E$
$\Longrightarrow E * E + E * E$
$\overset{4}{\Longrightarrow} N * V + V * V$
$\overset{3}{\Longrightarrow} N * x + y * z$
$\Longrightarrow 1N * x + y * z$

b) $\Longrightarrow 10 * x + y * z$

c) Regeln 1-5 sind kontext-frei (und nicht regulär). Regeln 6-8 sind regulär und würden vom Scanner vorverarbeitet werden. Token für den Parser wäre also z.B. $*, +, (,), n, v$, wobei n Ziffernfolgen und v Variablennamen repräsentieren würden.

Aufgabe 5 (4+4 Punkte)
Betrachten Sie das folgende *nanoLang*-Program.

a) Welche Symboltabellen sind an der Stelle `XXX` angelegt? Geben Sie die Hierarchie und die Einträge an.

b) *nanoLang* hat statisches Scoping. Welche Variablennamen sind sichtbar? Geben Sie jeweils Name und Typ an.

```
String gstr;
Integer str;

String StrMult(Integer x, String str)
{
   String res;

   res = "";
   str = gstr;

   while(x>0)
   {
      # XXXX
      res = StrCat(res, str);
      x = x-1;
   }
   return res;
}

Integer main(String count, String str)
{
   gstr = str;
   print StrMult(StrToInt(count), "");
   print "\n";
   return 0;
}
```

Lösung:

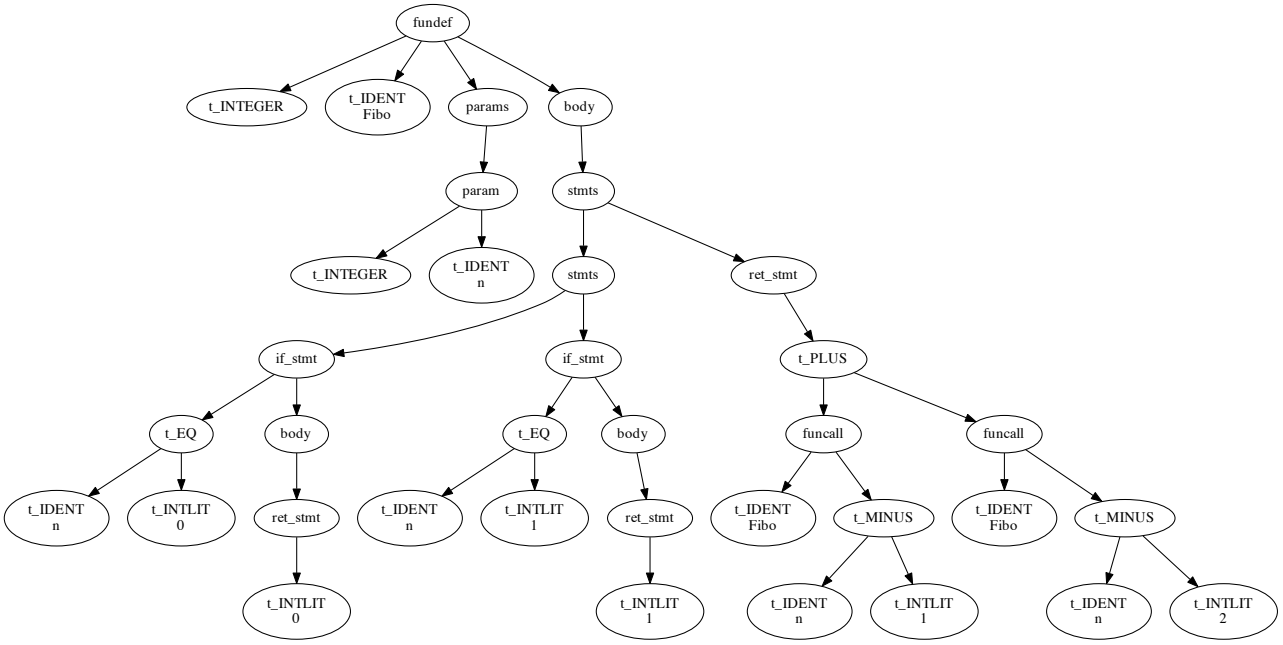
- Symboltabellen:
  - Global: gstr, str, StrMult, main
  - Function StrMult: x, str
  - Body StrMult: res
  - Body while: -
- Namen:
  - String res
  - Integer x
  - String Str
  - String gstr

Fortsetzung

Aufgabe 6 (7 Punkte)
Betrachten Sie das folgende Programmsegment. Geben Sie einen abstrakten Syntax-Baum an, der alle wesentlichen Eigenschaften des Segments widergibt. Sie können Knoten mit Operatoren oder Befehlsworten mit ebendiesen beschriften.

```
Integer Fibo(Integer n)
{
    if(n=0)
    {
        return 0;
    }
    if(n=1)
    {
        return 1;
    }
    return (Fibo(n-1)+Fibo(n-2));
}
```

Lösung:



Fortsetzung

Aufgabe 7 (7 Punkte)
Betrachten Sie die folgende Funktion. Wie können Sie die Funktion optimieren, ohne ihr Ergebnis zu verändern? Geben Sie die Optimierungen und das endgültige Programm an.

```
Integer optFun(Integer limit)
{
    Integer sum1;
    Integer sum2;
    Integer i;
    Integer j;
    Integer a;
    Integer b;
    sum1 = 0;
    sum2 = 0;
    i=0;

    while(i<=limit)
    {
        j=0;
        while(j<=limit)
        {
            Integer c;
            a = i*i*j;
            sum1 = sum1+a;
            b=i*i;
            sum2 = sum2+b;
            c = a+b;
            j=j+1;
        }
        i=i+1;
    }
    return sum1/sum2;
}
```

Lösung (z.B.):

```
Integer optFun(Integer limit)
{
    Integer sum1;
    Integer sum2;
    Integer i;
    Integer j;
    Integer a;
    Integer b;
    sum1 = 0;
    sum2 = 0;
    i=0;

    while(i<=limit)
    {
        j=0;
        b=i*i;  # Aus der Schleife geschoben
        while(j<=limit)
        {
            # Integer c; # Nicht verwendet
            a = b*j; # Common subexpression
            sum1 = sum1+a;
            sum2 = sum2+b;
            # c = a+b; Nicht verwendet
            j=j+1;
        }
        i=i+1;
    }
    return sum1/sum2;
}
```

Fortsetzung

– Ende der Klausur –