Matrikelnummer:			
DHBW Duale Hochschule Baden-Württemberg Stuttgart **ÜBUNGSKLAUSUR**		Fakultät Studiengang: Jahrgang / Kurs : Studienhalbjahr:	**Technik** **Angewandte Informatik** **2016 / 16B** **4. Semester**
Datum:	**16. Mai 2018**	Bearbeitungszeit:	**60 Minuten**
Modul: Unit:	**TINF2002.3** **Compilerbau**	Dozent:	**Stephan Schulz**
Hilfsmittel:	**Zwei Texte, z.B. Vorlesungsskript, eigene Notizen**		
Punkte:		Note:	

Aufgabe	erreichbar	erreicht
1	6	
2	10	
3	9	
4	10	
5	10	
6	9	
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)
Nennen Sie mindestens 3 nicht-triviale Aufgaben, die in der semantischen Analysephase eines Compilers für eine statisch getypte Sprache bearbeitet werden und beschreiben Sie diese kurz.

Lösung (3 von X):

- Namensverwaltung - alle Identifier müssen in ihren Scope genau ein mal definiert sein.
- Typinferenz: Für jeden Ausdruck/Teilausdruck muss der Typ ermittelt werden.
- Typüberprüfung: Argumente von Funktionen und Operatoren müssen mit der Signatur übereinstimmen.
- Funktionen müssen Werte des erklärten Rückgabetyps zurückgeben

Aufgabe 2 (10 Punkte)
Das folgende *nanoLang*-Programm enthält 5 Typen von Fehlern. Markieren Sie mindestes einen Fehler jeden Typs und beschreiben Sie, welche Phase des Compilers den Fehler jeweils identifiziert.

```
Integer not(Integer val)
{
   if(val == 0)
   {
      return 1;
   }
   return 0;
}

Integer Euklid(Integer a, Integer b)
{
   while(not(a + - b))
   {
      if(a > b)
      {
         a := a- - -b;
      }
      else
      {
         b := b - a;
      }
   }
   return a;
}


Integer main(String arg1, String arg2)
{
   print StrCat(IntToStr(Euklid(arg1, arg2)), '\n');
   return 0;
}
```

Lösung:

- `if(val == 0)` - == ist kein Operator. Parser (für den Lexer zwei = und somit ok.).
- `while(not(a+-b))`: `while` braucht in *nanoLang* echten Vergleich, keinen Integer Wert. Parser. Kein Fehler:`+-b`!.
- `:=` ist kein Token (2 mal), `:` kein legales Zeichen. Lexer.
- Argumente zu `Euklid()` sind vom Typ `String`, sollten `Integer` sein. Semantische Analyse.
- `'` ist kein Begrenzer für Strings, und klein legales Zeichen. Lexer.

Aufgabe 3 (2+2+5 Punkte)
Betrachten Sie die Grammatik $G = \langle\{S, L\}, \{(,), a\}, P, S\rangle\}$, wobei P die folgenden Regeln enthält:

1. $S \to a$
2. $S \to (L)$
3. $L \to LL$
4. $L \to \varepsilon$
5. $L \to S$

a) Geben Sie ein Linksableitung (*leftmost derivation*) für das Wort $((aa(a))a)$ an.

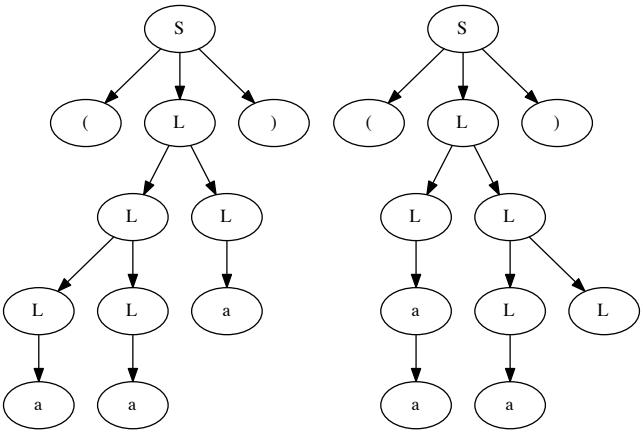
b) Geben Sie ein Rechtsableitung (*rightmost derivation*) für das Wort $((((a)a)a)a)$ an.

c) Ist G eindeutig oder mehrdeutig (*ambiguous*)? Falls eindeutig: Geben Sie ein Wort w mit $|w| \geq 6$ an, für das es nur einen einzigen Parsebaum (*parse tree*) gibt. Falls mehrdeutig (*ambiguous*): Geben Sie zwei verschiedene Parsebäume (*parse trees*) für ein von Ihnen gewähltes Wort an.

a)
$$\begin{aligned} S &\Rightarrow (L) \\ &\Rightarrow (LL) \\ &\Rightarrow (SL) \\ &\Rightarrow ((L)L) \\ &\Rightarrow ((LL)L) \\ &\Rightarrow ((LLL)L) \\ &\Rightarrow ((aLL)L) \\ &\Rightarrow ((aaL)L) \\ &\Rightarrow ((aaS)L) \\ &\Rightarrow ((aa(L))L) \\ &\Rightarrow ((aa(a))L) \\ &\Rightarrow ((aa(a))a) \end{aligned}$$

b)
$$\begin{aligned} S &\Rightarrow (L) \\ &\Rightarrow (LL) \\ &\Rightarrow (La) \\ &\Rightarrow (Sa) \\ &\Rightarrow ((L)a) \\ &\Rightarrow ((LL)a) \\ &\Rightarrow ((La)a) \\ &\Rightarrow ((Sa)a) \\ &\Rightarrow (((L)a)a) \\ &\Rightarrow (((LL)a)a) \\ &\Rightarrow (((La)a)a) \\ &\Rightarrow (((Sa)a)a) \\ &\Rightarrow ((((L)a)a)a) \\ &\Rightarrow ((((a)a)a)a) \end{aligned}$$

c) G ist mehrdeutig. Parse-Trees für (aaa):



Fortsetzung

Aufgabe 4 (2+4+4 Punkte)
Betrachten Sie das folgende *nanoLang*-Programm.

a) Geben Sie die Sequenz der ersten 10 Lexeme und die dazugehörigen Token des Programms an (z.B. im Format (("(", OPENPAR), (""Hallo"", STRINGLIT) ...). Es kommt nicht auf die exakten Token-Bezeichnungen an, diese sollten aber sinnvoll und eindeutig sein.

b) Nehmen Sie an, dass das Programm vollständig geparst ist und alle Symbole in die Symboltabellen eingetragen sind. Welche Symboltabellen sind an der Stelle XXX im Programm gültig? Geben Sie die Hierarchie der Symboltabellen und für jede Symboltabelle die Einträge mit Namen und Typ an. Geben Sie alle Namen an, die auch im Programm vorkommen, aber keine nicht verwendeten Library-Funktionen.

c) Generieren Sie einen abstrakten Syntaxbaum für die `while()`-Schleife in der Funktion `Convoluter()` (einschließlich des Codes im Rumpf der Schleife). Das Programm finden Sie auf der nächsten Seite noch einmal.

```
Integer y;

String Convoluter(Integer x)
{
   Integer res;
   res = 0;
   while(x > 0)
   {
      Integer z;
      res = res+y*((res+(x+y)));
      # XXX Symboltabellen hier!
      x = x-1;
   }
   return IntToStr(res);
}

Integer main(String arg)
{
   y = 2;
   print Convoluter(StrToInt(arg));
   print "\n";
   return 0;
}
```

Lösung:

a)
```
< "Integer"   ,  INTEGER >
< "y"         ,  IDENT >
< ";"         ,  SEMICOLON >
< "String"    ,  STRING >
< "Convoluter",  IDENT >
< "("         ,  OPENPAR >
< "Integer"   ,  INTEGER >
< "x"         ,  IDENT >
< ")"         ,  CLOSEPAR >
< "{"         ,  OPENCURLY >
```

Fortsetzung

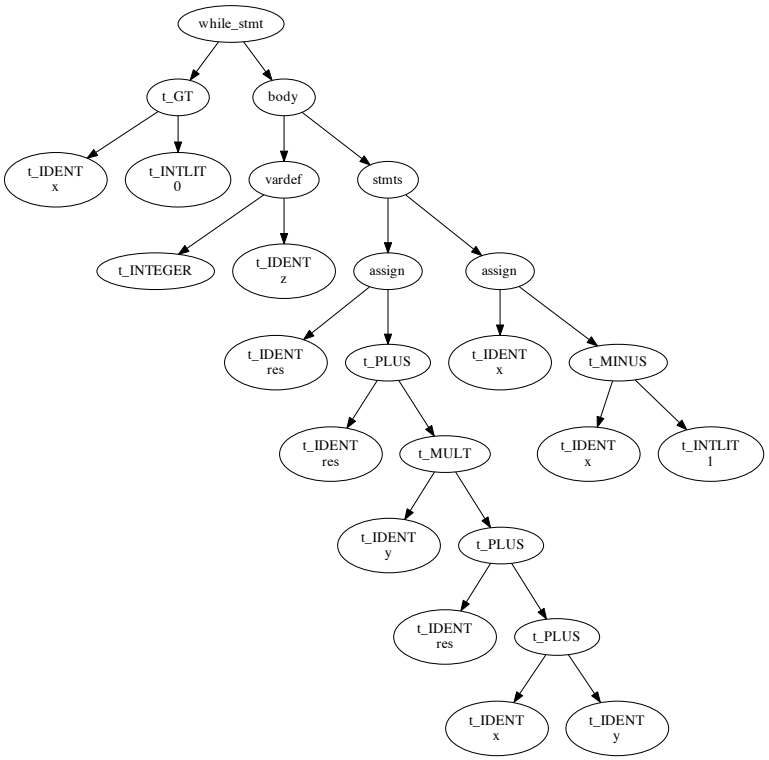
```
Integer y;

String Convoluter(Integer x)
{
   Integer res;
   res = 0;
   while(x > 0)
   {
      Integer z;
      res = res+y*((res+(x+y)));
      # XXX Symboltabellen hier!
      x = x-1;
   }
   return IntToStr(res);
}

Integer main(String arg)
{
   y = 2;
   print Convoluter(StrToInt(arg));
   print "\n";
   return 0;
}
```

Lösung:

b)
- Global
  - `IntToStr : (Integer) -> String`
  - `StrToInt : (String) -> Integer`
  - `y : Integer`
  - `Convoluter : (Integer) -> String`
  - `main : (String) -> Integer`
- `Convolutor()` Header
  - `x : Integer`
- `Convolutor()` Rumpf
  - `res : Integer`
- `while` Rumpf
  - `z : Integer`



c)

Aufgabe 5 (4+6 Punkte)
Betrachten Sie die folgende Sprache: Identifier sind nicht-leere (endliche) Sequenzen aus den Buchstaben `a`,`b`, `c`, `d`. Zahlen sind nicht-leere (endliche) Sequenzen aus den Ziffern `0`, `1`. Weitere Zeichen sind die Klammern `(` und `)`, und die Operatoren `*` und `+`.

Zahlen und Identifier sind Ausdrücke. Wenn e, f Ausdrücke sind, dann sind auch `(`e`)`, e`+`f und e`*`f Ausdrücke. Nur Worte, die nach diesen Regeln konstruiert werden, sind Ausdrücke. Es gilt: `*` bindet stärker als `+`, und beide Operatoren sind links-assoziativ.

a) Identifizien Sie ein sinnvolle Klasse von Token für diese Sprache und geben Sie für jeden Tokentyp einen regulären Ausdruck an, der diesen beschreibt. Sie können davon ausgehen, dass ein Zeichen in Hochkommas keine besondere Bedeutung hat (z.B. ist `'*'` das einfache Multiplikations-Zeichen, `*` (ohne Hochkommas) der Kleene-Stern)

b) Geben Sie eine kontext-freie Grammatik $G = \langle V_N, V_T, S, P\rangle$ an, die die Sprache der oben definierten Ausdrücke erzeugt. Die Grammatik sollte nur Parse-Trees erlauben, die die angegebene Präzedenz und die Assoziativitäten berücksichtigen.

a)
- Identifier: $(a|b|c|d)(a|b|c|d)*$
- Zahlen: $(0|1)(0|1)*$
- Klammer-auf: $'('$
- Klammer-zu: $')'$
- Plus: $'+'$
- Mal: $'*'$

b) $G = \langle\{A, P, F, I, Z\}, \{a, b, c, d, 0, 1,),), *, +\}, A, P\rangle$ mit folgenden Regeln in P:
- $A \to A + P$
- $A \to P$
- $P \to P * F$
- $P \to F$
- $F \to (A)$
- $F \to aI|bI|cI|dI$
- $I \to aI|bI|cI|dI|\varepsilon$
- $F \to 0Z|1Z$
- $Z \to 0Z|1Z|\varepsilon$

Fortsetzung

Aufgabe 6 (1+1+1+6 Punkte)
Betrachten Sie das folgende Programmsegment.

a) Bestimmen Sie den Rückgabewert von `sumfunc()` für folgende Eingabewerte:

i) 1

ii) 3

iii) 4

b) Wie können Sie die Funktion `sumfunc()` optimieren, ohne ihr Ergebnis zu verändern? Geben Sie die Optimierungen und die endgültige Funktion an.

```
Integer even(Integer num)
{
    Integer ret;
    ret = 0;

    if(num = 0)
    {
        return 1;
    }
    if(num = 1)
    {
        return 0;
    }
    return even(num-2);
}

Integer sumfunc(Integer num)
{
    Integer sum;
    sum = 0;
    while(num > 0)
    {
        if(even(num)>0)
        {
            sum = sum + (num*num);
            num = num-1;
        }
        else
        {
            if(even(num)<=0)
            {
                num = num-1;
            }
        }
    }
    print sum;
    print "\n";
    return sum;
}
```

Fortsetzung
Lösung:

a) $1 \rightarrow 0$; $3 \rightarrow 4$; $4 \rightarrow 20$

b)
- if(even(num)...) ist immer wahr im else Zweig: Bedingung und else streichen.
- num = num-1 wird im if und else Zweig ausgeführt und kann herausgezogen werden.
- ret in `even()` wird nicht benötigt: streichen

```
Integer even(Integer num)
{
    if(num = 0)
    {
        return 1;
    }
    if(num = 1)
    {
        return 0;
    }
    return even(num-2);
}

Integer sumfunc(Integer num)
{
    Integer sum;
    sum = 0;
    while(num > 0)
    {
        if(even(num)>0)
        {
            sum = sum + (num*num);
        }
        num = num-1;
    }
    print sum;
    print "\n";
    return sum;
}
```

– Ende der Klausur –