Student/in:		Unterschrift:	
DHBW Duale Hochschule Baden-Württemberg Stuttgart **KLAUSURDECKBLATT**		Fakultät Studiengang: Jahrgang / Kurs : Studienhalbjahr:	**Technik** **Angewandte Informatik** **2013 / 13B** **4. Semester**
Datum:	**29. 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	10	
3	12	
4	10	
5	9	
6	7	
7	6	
Summe	60	

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 jeweils kurz die Funktion eines Compilers und die Funktion eines Interpreters. Welche wesentlichen Teile sind beiden gemeinsam? Was sind die wesentlichen Unterschiede?

Lösung:

- Compiler: Liest Source Code und wandelt ihn (direkt oder indirekt) in ein direkt ausführbares Programm um. Der Compiler läuft zur Entwicklungszeit, das kompilierte Programm läuft dann ohne den Compiler, aber nur auf der festen Zielplattform.
- Interpreter: Liest Source Code und führt ihn (direkt) aus. Der Interpreter läuft zur Ausführzeit und ist für jede Ausführung des Programs notwendig. Das Programm läuft auf jeder Platform, auf der der Interpreter verfügbar ist.
- Sowohl Compiler als auch Interpreter benötigen Scanner und Parser, um die syntaktische Struktur des Programs zu erkennen.

Aufgabe 2 (10 Punkte)
Das folgende *nanoLang*-Programm enthält 5 verschiedene Fehler. Identifizieren Sie die einzelnen Fehler und beschreiben Sie, welche Phase des Compilers den Fehler jeweils identifiziert.

```
String root(Integer n)
{
   Integer square(Integer n)
   {
      return n*n;
   }

   Integer guess;
   guess := n;

   while(square(guess)!=n)
   {
      if(square(guess)>n)
      {
         guess := guess/2;
      }
      if(square(guess)<n)
      {
         guess := guess+1;
      }
   }
   return guess;
}

Integer main(String arg)
{
   print root(arg)+"\n";

   return 0;
}
```

Lösung:

- Geschachtelte Funktion `square()` ist in *nanoLang* nicht erlaubt. Parser/Syntaxanalyse.
- `root()` soll String zurückliefern, aber `guess` ist Integer. Semantische Prüfung.
- Zuweisungen in *nanoLang* verwenden `=`. Der Doppelpunkt wird schon vom Lexer/Scanner als Fehler erkannt.
- Die Argumente für das `+` in `main()` haben den falschen Typ. Semantische Prüfung.
- `root()` erwartet Integer-Parameter, bekommt aber String. Semantische Prüfung.

Aufgabe 3 (4+4+4 Punkte)
Betrachten Sie die folgende formale Sprache von *Typen*: `int` und `char` sind Typen. Wenn `t1` und `t2` Typen sind, so auch `t1[]`, `t1->t2`, und `(t1)`. Der Pfeil `->` ist rechtsassoziativ. Die Array-Klammern `[]` sind linksassoziativ und binden stärker als der Pfeil `->`. Beispiele für korrekte *Typen* und Gegenbeispiele sind:

Korrekte Typen	**Keine korrekten Typen**
`char -> int`	`int()`
`int[][][]`	`->char`
`(int -> int)[]`	`real->hans`
`(int -> (int -> int))`	`char,int`
`int[] -> char -> char[]`	`int->[]char`

a) Geben Sie eine Kontext-freie Grammatik $G = \langle V_N, V_T, P, S\rangle$ an, mit der die Sprache der Typen beschrieben wird. Sie können `int`, `char`, `->` und `[]` als Terminalsymbole voraussetzen.

b) Leiten Sie mit Ihrer Grammatik folgende Worte ab:

1. `int[][][]`
2. `int[] -> char -> char[]`

c) Geben Sie zu folgenden Worten einen Parse-Tree mit Ihrer Grammatik an.

1. `char -> int`
2. `int[] -> char -> char[]`

Lösung:

a) $V_N = \{T, A\}$, $V_T = \{$`->`, `[]`, $(,)$, `int`, `char`$\}$, Startsymbol T, P siehe unten.

1. $T \to A$ `->` $T \mid A$
2. $A \to A$`[]` $\mid (T) \mid$ `int` $\mid$ `char`

b) 1: $T \Longrightarrow A$ `->` $T \Longrightarrow T$ `->` $T \Longrightarrow$ `char ->` $T \Longrightarrow$ `char -> int`

5: $T \Longrightarrow A$ `->` $T \Longrightarrow A$ `->` A `->` $T \Longrightarrow A$ `->` A `->` $A \Longrightarrow A$`[]` `->` A `->` $A \Longrightarrow A$`[]` `->` A `->` A`[]` $\Longrightarrow$ `int[] ->` A `->` A`[]` $\Longrightarrow$ `int[] -> char ->` A`[]` $\Longrightarrow$ `int[] -> char -> char[]`

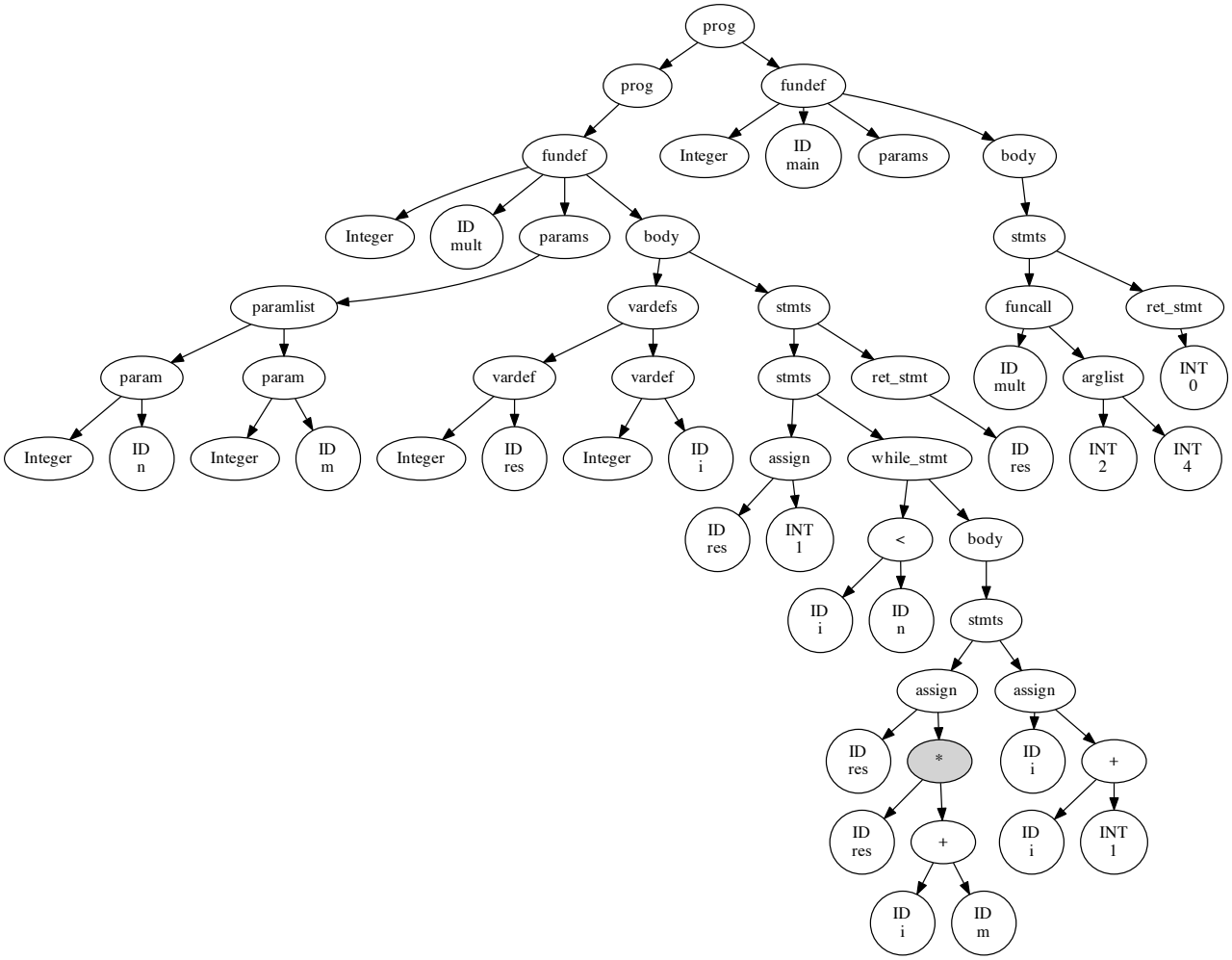
c) 2: $T \Longrightarrow A \Longrightarrow A$`[]` $\Longrightarrow A$`[][]` $\Longrightarrow A$`[][][]` $\Longrightarrow$ `int[][][]`

5: $T \Longrightarrow A$ `->` $T \Longrightarrow A$ `->` A `->` $T \Longrightarrow A$ `->` A `->` $A \Longrightarrow A$`[]` `->` A `->` $A \Longrightarrow A$`[]` `->` A `->` A`[]` $\Longrightarrow$ `int[] ->` A `->` A`[]` $\Longrightarrow$ `int[] -> char ->` A`[]` $\Longrightarrow$ `int[] -> char -> char[]`

Aufgabe 4 (6+4 Punkte)
Betrachten Sie den *abstract syntax tree* (AST) auf der nächsten Seite.

a) Generieren Sie aus dem AST ein äquivalentes *nanoLang*-Programm. Hinweis: Das Programm sollte bei normaler Schriftgröße problemlos unter den AST passen.

b) Nehmen Sie an, dass das Programm vollständig geparst ist und alle Symbole in die Symboltabellen eingetragen sind. Welche Symboltabellen sind an dem grau hinterlegten Knoten `*` gültig? Geben Sie die Hierarchie und die Einträge mit Namen und Typ an. Geben Sie nur die Namen an, die auch im Programm vorkommen, keine Library-Funktionen.



Lösung:
a) Programm

```
Integer mult(Integer n, Integer m)
{
    Integer res;
    Integer i;
    res=1;
    while(i<n)
    {
        res = res*(i+m);
        i=i+1;
    }
    return res;
}

Integer main()
{
    mult(2,4);
   return 0;
}
```

b) Symboltabellen:

- Global
  - `main():()->Integer`
  - `mult():(Integer, Integer)->Integer`
- `mult()` Header
  - `n: Integer`
  - `m: Integer`
- `mult() Body`
  - `res: Integer`
  - `i: Integer`
- Body `while`
  - Leer

Aufgabe 5 (9 Punkte)
Benennen Sie 3 wichtige Phasen eines Compilers. Beschreiben Sie jeweils kurz die Funktion und die eingesetzten Konzepte. Gehört die jeweilige Phase zum Front-End oder zum Back-End?

Lösung (3 von z.B. 4):

1. Der *Scanner* oder *Lexer* ist eine Phase des Front-End. zerlegt die Eingabe in einzelne Lexeme und assoziierte Token. Token sind typischerweise Worte einer regulären Sprache und durch reguläre Ausdrücke (oder eine reguläre Grammatik) beschrieben. Scanner sind oft in Form von endlichen Automaten beschrieben.

2. Die *Syntaxanalyse* ist Teil des Frontends. Sie überprüft die syntaktische Korrektheit der Eingabe und baut aus einer Folge von Token einen Parse-Baum oder gleich einen *abstrakten Syntax-Baum*. Grundlage ist eine kontext-freie Grammatik, die über einen Top-Down oder Bottom-Up Parser erkannt wird.

3. Die *semantische Analyse* ist typischerweise die letzte Phase des Front-End. bestimmt für jeden Identifier und jeden Teilausdruck den Typ. Sie überprüft semantisch Anforderungen. Grundlage sind Symboltabellen und Typinferenz.

4. Die Die Codegenerierung erzeugt aus einer maschinenunabhängien Zwischenform (z.B. dem abstrakten Syntaxbaum) das Programm in der Zielsprache.

Aufgabe 6 (7 Punkte)
Betrachten Sie die folgende Grammatik: $G = \langle V_N, V_T, P, E\rangle$ mit $V_N = \{E\}$, $V_T = \{v, n, *, +, (,)\}$, und folgenden Produktionen in P:

- $E \to (E)$
- $E \to E + E$
- $E \to E * E$
- $E \to v$
- $E \to n$

Geben Sie eine Grammatik G' mit $L(G') = L(G)$ an, die nur Parse-Trees ermöglicht, die $*$ und $+$ links-assoziativ interpretieren, und die "Punktrechnung vor Strichrechnung" respektieren.

Lösung: $G' = \langle V'_N, V_T, P, E\rangle$ mit $V'_N = \{E, T, F\}$ und P wie folgt:

- $E \to E + T$
- $E \to T$
- $T \to T * F$
- $T \to F$
- $F \to (E)|v|n$

Fortsetzung

Aufgabe 7 (6 Punkte)
Betrachten Sie das folgende Programm. Wie können Sie die Funktion `root()` optimieren, ohne ihr Ergebnis zu verändern? Geben Sie die Optimierungen und die endgültige Funktion an.

```
Integer square(Integer n)
{
   return n*n;
}

Integer root(Integer n)
{
   Integer guess;
   Integer upper;

   guess = n;
   upper = guess;

   while(square(guess)!=n)
   {
      if(square(guess)>n)
      {
         upper = guess;
         guess = guess/2;
      }
      if(square(guess)<n)
      {
         guess = guess+1;
      }
   }
   return guess;
}

Integer main(String arg)
{
   print root(StrToInt(arg));
   print "\n";

   return 0;
}
```

Lösung:

- `upper` wird nie sinnvoll verwendet, streichen!
- `square(guess)` muss nur berechnet werden, wenn sich `guess` aendert. werden - *common subexpression elimination.*

```
Integer root(Integer n)
{
   Integer guess;
   Integer sq;

   guess = n;
   sq = square(guess);

   while(sq!=n)
   {
      if(sq>n)
      {
         guess = guess/2;
         sq = square(guess);
      }
      if(sq<n)
```

```
        {
            guess = guess+1;
            sq = square(guess);
        }
    }
    return guess;
}
```

Fortsetzung

– Ende der Klausur –