FILS

Course: Compiler Techniques

Homework #7

Type Checking and Intermediate Code Generation

To do *type checking* a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.
Type checking can take on two forms: synthesis and inference. Type synthesis builds up the type of an expression from the types of its sub-expressions. It requires names to be declared before they are used.
Ex. of synthesis rule:

```
if f has type s -> t and x has type s, then expression f (x) has type t
```

Type inference determines the type of a language construct from the way it is used.
Ex. of inference rule:

```
if f(x) is an expression, then for some α and β, f has type α -> β and x has
type α
```

The type of an expression *E1* **op** *E2* is determined by the operator **op** and the types of *E1* and *E2.* A *coercion* is an implicit type conversion, such as from *integer* to *float.*
A declaration specifies the type of a name. The width of a type is the amount of storage needed for a name with that type. Using widths, the relative address of a name at run time can be computed as an offset from the start of a data area. The type and relative address of a name are put into the symbol table due to a declaration, so the translator can subsequently get them when the name appears in an expression.

An *intermediate representation* is typically some combination of a graphical notation and three-address code. As in syntax trees, a node in a graphical notation represents a construct; the children of a node represent its sub-constructs. Three address code takes its name from instructions of the form x := y **op** z, with at most one operator per instruction. There are additional instructions for control flow.
Expressions with built-up operations can be unwound into a sequence of individual operations by attaching actions to each production of the form *E:= E1* **op** *E2.* The action either creates a node for *E* with the nodes for *E1* and *E2* as children (if the intermediate form is a syntax three), or it generates an instruction in an intermediate code language (for instance, three-address instructions) that applies **op** to the addresses for *E1* and *E2* and puts the result into a new temporary name, which becomes the address for E.

Exercise 1. Type checking

Below is a grammar used by a calculator of arithmetic expressions involving operator * and integer or floating-point constant numbers. Floating-point numbers are distinguished by having a decimal point.
The calculated values are assigned to corresponding-as-type variables. The variables are gathered with their attributes in a symbol table.

```
L-> L ; S | S
S-> id = E
E-> E * F | F
F -> num.num | num
```

a) Give an SDD to determine the type of each factor F and expression E and update variables in the symbol table with type attributes.
b) Extend your SDD of (a) to translate expressions into three-address instructions. Use the unary operator **intToFloat** to turn an integer into an equivalent float.

Exercise 2. Type checking

Determine the types and relative addresses for the identifiers in the following sequence of declarations:

```
float x;
record (float x; float y; ) p;
record (int tag; float x; float y; ) q;
```

Exercise 3. Code generation

Translate the arithmetic expressions:

```
   a + -(b + c)
   a=b[i]+c[j]
   a[i]=b*c - b*d
```

into:

a) A syntax tree.

b) Quadruples.

c) Triples.

Exercise 4. Code generation

Give the intermediate code generated for the following statements in Java:

```
if (x<100 || x>200 && x!=y ) x=0;
if (a==b && (c==d || e==f)) a=b=c*d + e*f;
repeat x=x+5 until x*x>y;
```

For all examples, optimize the code by avoiding redundant gotos;