Compilers

Spring 2009

Homework 4

Due Tuesday, May 5, 2009 at 2.30 PM in class

Please label all pages you turn in with your name and student number.

Problem 1: Attributive Grammar and Syntax-Directed Translation [40 points]

In class we described a SDT translation scheme for declarations in a PASCAL-like language where the identifiers precede the type declaration itself. In this exercise you are asked to develop an attributive grammar and syntax-directed definition that performs type checking for integer and real values.

This type checking means that your syntax-directed definition should first identify the value of expressions based on the declared types and then should check (hence the name type-checking) that on binary operations such as addition or multiplications (to make matters simple here) both operands are of the same type. To accomplish this you have a partial grammar below for which you need to:

a) [20 points] Define the attributes for each non-terminal symbol and the corresponding semantics rules.
b) [10 points] Determine the order in which the attributes need to be evaluated.
c) [10 points] Show an example of the attribute values in the code example: "s = a * b + 1.0" where "s" and "a" are declared as integer and "b" is declared as real.

Note that in the case of a type error, or mismatch the semantic action must be able to propagate an undefined or erroneous value to other expressions or identifiers to be later used by the compiler error reporting functions.

Decl → **id** List　　　　　　　　　　　Assign → **id** = Expr
List　→ ',' **id** List | ':' Type　　　　Expr → Expr + Expr
Type → **integer** | **real**　　　　　　Expr → Expr * Expr
　　　　　　　　　　　　　　　　　　Expr → **id**
Stmt → Decl | Assign　　　　　　　　Expr → Const
StmtList → Stmt ; StmtList | ε　　　　Const → **intconst** | **realconst**

Problem 2: Intermediate Code Generation [30 points]

For the assignment instruction below perform the following:

```
x = ( a + ( b * 2)) + 1
```

a) [10 points] Augment the SDT scheme with a rule corresponding to the production $E \rightarrow const$ and using a "value" attribute for the constant with its numeric value.

b) [05 points] Generate three-address instructions using the SDT scheme described in class (skipping over the rules for parenthesis to simplify) and without any minimization of temporary variables.

c) [10 points] Redo the code generation above but now reusing temporaries using the method described in class.

d) [05 points] Argue that the solution found in c) is optimal.

Problem 3: Back-patching of Loop Constructs [30 points]

In class we saw the actions for a Syntax-Directed Translation scheme to generated code using the back-patching technique for a *while* loop construct. In this exercise you will develop a similar scheme for the *do-while* construct using the production below and also taking into account *continue* and *break* statements. Argue that your solution works for the case of nested loops and *break* and *continue* statements at different nesting levels.

> (1) S → **do** L **while** E;
> (2) S → **continue;**
> (3) S → **break;**
> (4) L → S ; L
> (4) L → S

Do not forget to show the augmented production with the marker non-terminal symbols, M and possibly N along with the corresponding rules for the additional symbols and productions. Argue for the correctness of your solution without necessarily having to show an example.