

Assignment 2

CSE 755

Due: October 13, by 11:00 am

1. (5 pts)

Consider the type-checking example from the discussion of attribute grammars. Suppose that nonterminal $\langle formallist \rangle$ is defined as

$$\begin{aligned} \langle formallist \rangle &::= \langle formal \rangle \mid \langle formal \rangle , \langle formallist \rangle \\ \langle formal \rangle &::= \text{int id} \mid \text{bool id} \end{aligned}$$

In the lecture notes, there is an informal description of an attribute $types$ for nonterminal $\langle formallist \rangle$. Describe the value domain of $types$, and write the attribute grammar rules for computing this attribute.

2. (5 pts)

Consider the type-checking example from the discussion of attribute grammars. Suppose that nonterminal $\langle actualslist \rangle$ is defined as

$$\begin{aligned} \langle actualslist \rangle &::= \langle actual \rangle \mid \langle actual \rangle , \langle actualslist \rangle \\ \langle actual \rangle &::= \langle intexp \rangle \mid \langle boolexp \rangle \end{aligned}$$

In the lecture notes, there is an informal description of an attribute $expTypes$ (“expected types”) for $\langle actualslist \rangle$. At production $\langle intexp \rangle ::= \text{id} (\langle actualslist \rangle)$ the value of this attribute for the $\langle actualslist \rangle$ node is obtained from information about function `id`, as provided by attribute $alltbl$ (i.e., as provided by the stack of symbol tables); helper function `paramtypes` looks up this information.

Describe the value domain of $expTypes$, taking into account your solution for Problem 1. Describe informally how this attribute should be used to perform type checking of the actuals at the call site. As we discussed in class, the basic idea is to make sure that the expected type of each actual is indeed correct (e.g., if the expected type is `INT`, the actual parameter should be an $\langle intexp \rangle$). Based on your approach, write all rules for this attribute, as well as for all other attributes of nodes of type $\langle actual \rangle$ (if necessary). Do *not* introduce additional attributes for $\langle actualslist \rangle$.

3. (10 pts)

As discussed in class, the type-checking attribute grammar will accept

```

begin
  fun f (int i) : int =
  begin
    ... g(5) ...
  end;
  fun g (int j) : int =
  begin
    ...
  end;
  ...
end

```

Suppose we did not want to allow such “forward” references to declarations that appear later in the code—that is, within a $\langle declist \rangle$, the body of a function can refer only to variables or functions that are declared (1) earlier in the same $\langle declist \rangle$, or (2) in some surrounding outer block.

Change the attribute grammar discussed in class to achieve this goal. Do *not* add any new attributes; just change the evaluation rules. If you define any helper functions, describe them in detail. Do *not* change the underlying context-free grammar. You do *not* need to write down the entire attribute grammar; show only the new/changed parts of the grammar.

4. (20 pts)

Consider the following context-free grammar with a starting non-terminal $\langle prog \rangle$:

$$\begin{aligned}
 \langle prog \rangle &::= \langle seq \rangle \\
 \langle seq \rangle_1 &::= \langle stmt \rangle \mid \langle stmt \rangle ; \langle seq \rangle_2 \\
 \langle stmt \rangle &::= \langle assign \rangle \\
 \langle assign \rangle &::= id := \langle rhs \rangle \\
 \langle rhs \rangle &::= id + id \mid const
 \end{aligned}$$

This language allows straight-line code with simple assignments. Assume that the language semantics is the “standard” semantics discussed in class. Assume that `id` has an attribute *lexval* that gives you the string name of the identifier; this attribute is already provided and you do not need to define rules to compute it.

Part 1. Define an attribute grammar that identifies certain *redundancies*. Eliminating such redundancies may improve performance. Specifically, consider a statement s_1 of the form $x:=y+z$. Suppose that later in the code there is another statement s_2 of the form $w:=y+z$. Under certain conditions, s_2 can be replaced by $w:=x$, which will eliminate a redundant re-computation of $y+z$. For this homework problem, the condition we will consider is the following: *the statements that appear between s_1 and s_2 do not modify the values of x , y , and z* . If this condition holds, s_2 can be optimized.

For $\langle assign \rangle$ you need to define a boolean attribute *opt* whose value is true if the statement can be optimized according to the criterion described above. In cases

where the right-hand side of the assignment is a constant, the value of this attribute will be “false” (i.e., we cannot optimize $w:=5$). You can assume that each variable is initialized before its first use.

Show an attribute grammar for computing the value of $\langle assign \rangle.opt$. If you define any other attributes or helper functions, describe them in detail. Do not change the underlying context-free grammar.

Part 2. Suppose we enrich the language to include conditional behavior:

$$\begin{aligned} \langle stmt \rangle &::= \langle assign \rangle \mid \langle if \rangle \\ \langle if \rangle &::= \text{if } (..) \langle assign \rangle \text{ fi} \end{aligned}$$

Assume that you do not have any compile-time information about the conditions at ifs, and therefore you have to consider both outcomes (true and false) as possible. Also assume that these conditions do not write any variables in the program. With these changes, we could have multiple possible run-time executions of a program. We could generalize the optimization from above as follows: if, for all possible run-time executions, the values of x , y , and z are guaranteed to stay the same between s_1 and s_2 , then s_2 can be optimized.

Describe all necessary changes to the attribute grammar from Part 1 to accommodate the new language and the new definition of $\langle assign \rangle.opt$.

5. (10 pts)

Consider the language from the code-generation example discussed in class. Suppose we enrich the language to include the following two kinds of statements:

$$\begin{aligned} \langle c \rangle_1 &::= \dots \mid \text{repeat } \langle c \rangle_2 \text{ until } \langle be \rangle \\ &\quad \mid \text{if } \langle be \rangle_1 \text{ then } \langle c \rangle_2 \text{ elsif } \langle be \rangle_2 \text{ then } \langle c \rangle_3 \text{ else } \langle c \rangle_4 \end{aligned}$$

The first new statement is a *repeat-until* loop, in which the condition is evaluated at the end of the loop body; if this condition is false, the loop continues with the next iteration. (Similar loop construct exists in the Pascal programming language.) The second statement is a generalized if-statement: if $\langle be \rangle_1$ evaluates to true, $\langle c \rangle_2$ is executed. Otherwise, $\langle be \rangle_2$ is evaluated; depending on the outcome, either $\langle c \rangle_3$ or $\langle c \rangle_4$ is executed. (Fortran and PHP have similar kinds of statements.)

Describe in detail all necessary changes to the code-generation attribute grammar discussed in class in order to handle these new kinds of statements.