UNIVERSITETET I OSLO
Institutt for Informatikk

PMA, PSE

Axelsen, Krogdahl, Møller-Pedersen, Steffen

INF 5110: Compiler construction

Spring 2017

Series 6

20. 4. 2017

Topic: Symbol tables and type checking (Chapter 6)

Issued: 20. 4. 2017

Exercise 1 (AG: collateral vs. sequential declarations) [1] Rewrite the grammar from Table 6.9 from [1] to use *collateral* declarations instead of sequential ones.

The *underlying* grammar is given in Table 1.

$$
\begin{array}{rcl}
S & \rightarrow & \mathit{exp} \\
\mathit{exp} & \rightarrow & (\,\mathit{exp}\,) \mid \mathit{exp} + \mathit{exp} \mid \mathbf{id} \mid \mathit{num} \mid \mathbf{let}\ \mathit{dec\text{-}list}\ \mathbf{in}\ \mathit{exp} \\
\mathit{dec\text{-}list} & \rightarrow & \mathit{dec\text{-}list}\,,\ \mathit{decl} \mid \mathit{decl} \\
\mathit{decl} & \rightarrow & \mathbf{id} = \mathit{exp}
\end{array}
$$

Table 1: Expression grammar with declarations

Exercise 2 (AG for expression evaluation) [2] Write an attribute grammar that computes the *value* of each expression for the expression grammar of [1, Section 6.3.5]. The grammar is repeated in Table 1 (it's the same as in the previous exercise).

Exercise 3 (AG: type conversion resp. evaluation) [3] Consider the following (ambiguous) expression grammar.

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{exp} + \mathit{exp} \mid \mathit{exp} - \mathit{exp} \mid \mathit{exp} * \mathit{exp} \mid \mathit{exp}\,/\,\mathit{exp} \\
& \mid & (\,\mathit{exp}\,) \mid \mathbf{num} \mid \mathbf{num}\,.\,\mathbf{num}
\end{array}
$$

Suppose that the rules of C are followed in computing the *value* of such expressions:

> If two subexpressions are of *mixed type,* then the integer subexpression is *converted* to floating point, and the floating-point operator is applied.

Write an attribute grammar that will convert such expressions in expressions that are legal in Modula-2: conversions from integer to floating point are expressed by applying the `FLOAT` function, and the division operator `/` is considered to be `div` if both operands are integers.

That was the task as in [1]. In the lecture: let's use an AG to *evaluate* such expressions (instead of converting them to Modula-2's conventions).

[1] The task corresponds to [1, Exercise 6.17.]
[2] The task corresponds to [1, Exercise 6.18.]
[3] The task corresponds to [1, Exercise 6.20.]

Exercise 4 (Type equality and type checking) [4]

1. Devise a suitable tree structure for the new function type structures, and write a *typeEqual* function for two function types.

2. Write semantic rules for the type checking of function declarations and function calls, represented by a rule

$$exp \rightarrow \textbf{id} \,\textbf{(}\; exp \;\textbf{)} \;,$$

   similar to the rules of [1, Table 6.10, page 330].

Exercise 5 (Symbol table) [5] Consider the following ambiguity in C expressions. Consider the expression `(A)-x`. If `x` is an integer variable and `A` is defined in a `typedef` as equivalent to `double`, then this expression *casts* the value of `-x` to `double`. On the other hand, if `A` is an integer variable, then this *computes* the integer difference of the two variables.

1. Describe how the *parser* might use the *symbol table* to disambiguate the two interpretations.

2. Describe how the *scanner* might use the symbol table disambiguate the two interpretations.

References

[1] K. Louden. *Compiler Construction, Principles and Practice.* PWS Publishing, 1997.

[4] The task corresponds to [1, Exercise 6.21.]
[5] The task corresponds to [1, Exercise 6.22.]