Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen, Gianluca Turin

INF 5110: Compiler construction

Spring 2021

Series 6

12. 3. 2021

Topic: Symbol tables and type checking (Chapter 6)

Issued: 12. 3. 2021

Exercise 1 (AG: collateral vs. sequential declarations) Extend the grammar of Table 1 into an AG to capture "*collateral*" (simultaneous) declarations.

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

Table 1: Expression grammar with declarations

As a starting point, use the grammar from the lecture, which is reproduced here. So: Rewrite the grammar from Table 2 on the next page to use *collateral* declarations instead of sequential ones.

Exercise 2 (AG for expression evaluation) Write an attribute grammar that computes the *value* of each expression for the expression grammar Table 1 (it's the same as in the previous exercise).

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

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

Assume you are dealing with two numerical types, for integers and for floats. 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.

Grammar Rule	Semantic Rules	
$S \to exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$	
$exp_1 \to exp_2$ **+** exp_3	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$	
$exp_1 \to$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$	
$exp \to$ **id**	$exp.err =$ **not** $isin(exp.symtab,$ **id** $.name)$	2
$exp \to$ **num**	$exp.err =$ **false**	
$exp_1 \to$ **let** $dec\text{-}list$ **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$	3
$dec\text{-}list_1 \to dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$	4
$dec\text{-}list \to decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$	4
$decl \to$ **id** **=** exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** $(decl.intab = errtab)$ **or** $exp.err$ **then** $errtab$ **else if** $(lookup(decl.intab,$ **id** $.name) =$ $decl.nestlevel)$ **then** $errtab$ **else** $insert(decl.intab,$ **id** $.name, decl.nestlevel)$	1

Table 2: Sequential declarations (from the lecture)

Exercise 4 (Type equality and type checking)

Consider the following grammar which in particular features procedure or function declarations (Table 3)

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

program → *var-decls* **;** *fun-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → **int** | **bool** | **array [num] of** *type-exp*
fun-decls → **fun id (** *var-decls* **) :** *type-exp* **;** *body*
body → *exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id :=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[** *exp* **]** | **id (** *exps* **)**
| **num** | **true** | **false** | **id**
exps → *exps* **,** *exp* | *exp*

Table 3: Grammar with function declarations

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

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

Similar to the rules in the slide "Type checking as semantic rules" in the type checking section of Chapter 7 in the slides.

Exercise 5 (Symbol table) Think about 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.