Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2022 — **Series 6** — 1. 4. 2022

Topic: Symbol tables and type checking (Chapter 6) (Exercises with hints for solution)

Issued: 1. 4. 2022

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

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

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.

Solution: The original definition of the attribute grammar for *sequential declarations* from the book and the lecture is repeated here in Table 2.

The AG looks complex, but actually part of it may be due to the one particular requirement on AGs which we also was mentioned variously in the exercises and the lecture: AGs are a *declarative* or *functional* formalism! That is slightly different from the "action part" in a parser generator such as `cup` or `yacc`, where the action part (which correspond to the semantic rules of an AG) can have *side effects*, one can destructively update a "*state*". That's *not* allowed in the semantic rules of an AG. Of course a semantic rule like

$$exp_2\texttt{.nestlevel} = exp_1\texttt{.nestlevel}$$

can be interpreted (in an imperative programming language) as

the field `nestlevel` in object/record exp_2 is assigned to the value of the field exp_1`.nestlevel`,

or in (alternative) programming notation

```
exp2.nestlevel := exp1.nestlevel .
```

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)

I use here `:=` for *assignment*, which is something different from *equations*.[1] That assignments are not really the same as equations can be seen already from the fact that we know that $x = 3$ is the same equation than $3 = x$, but the same swapping of sides ("commutativity") cannot be done for assignments $x := 3$ ...

[1] In the tradition of C and similar, many languages use `=` for assignments. But in many others, it's `:=` or similar for assignments.

Another way of seeing it is: we can view an equation of the form $x = 3$ or exp_2 `.nestlevel` = exp_1 `.nestlevel` as assignment, but what is *not* allowed is to assign to the left-hand side *again later!* In the terminology of progamming languages (and/or compilers), it a *single-assignment* variable. So the difference between equations $x = 3$ (which correspond to single-assignment variables in programming languages) and assignments $x := 3$ is: the first is meant as "x **is** 3", the second one means something more complex, namely

> "whatever the value of x was **before** the assignment, **afterwards** it's 3 (up until it possibly is changed again)".

Attributes in the AG formalism correspond to variables, but they are of the "mathematical" kind (i.e., single-assignment).

Now, what consequence does all that have for the AG of sequential variable type declarations? In principle, sequential type declarations using symbol tables are pretty simple: there is a sequence of type declarations (see the non-terminal *dec-list*). Then the type checker goes through the individual declarations (see the production $decl \rightarrow \mathbf{id} = exp$) one by one. Each time, it adds the type declaration as binding to the symbol table. In a typical *imperative*[2] implementation of symbol tables (for instance using hash tables), it means updating the symbol table (by some procedure `insert` or similar).

Doing so is fairly obvious, but when it comes to the attribute grammar specification, being a declarative formalism, *we cannot destructively update a symbol table.* It violates the fact that the specification is declarative (resp. that variables are "single-assignment").

The solution to that is simple (once one has seen it): one distinguishes between the symbol table *before* the declaration and the symbol table afterwards. So instead of an imperative view like "add the binding to the symbol table", the view is "calculate the new symbol table from the old symbol table by inserting some binding". So, the fact that we are not allowed to *update* an attribute `symtab` leads to the formulation that there are attributes `intab` and `outtab`, representing the symbol table before the addition and the symbol table after! The way that the symbol tables are "handed down" in lists of declarations is "left-to-right". Now in the AST, "left-to-right" here refers to relationship between *siblings*, not to parents-children. If we have a strict interpretation of what inherited and synthesized attributes are, then the attributes may be neither. Nonetheless, we can classify the `intabl` and `outtab` as follows (using the more relaxed definitions of inherited and synthesized attributes):

symbol	attributes	kind
exp	`symtab`	inherited
	`nestlevel`	inherited
	err	synthesized
dec-list, *decl*	`intab`	inherited
	`outtab`	synthesized
	`nestlevel`	inherited
id	`name`	injected by scanner

The second piece that needs consideration may be the (inherited) attribute `nestlevel`. Basically, the declaration list of a let-construct has a nesting level increased by one compared to the "surrounding context" (represented by exp_1 in the corresponding semantic rule). The same nesting level is for exp_2 in the let-construct. In the semantic action, the nesting level for exp_2 is inherited from *dec-list*. This is another instance of "sibling-inheritance". Alternatively, one could have inherited the nesting level from exp_1 to exp_2 (but then increasing the level by one, of course).

[2] Imperative means, with side-effects. Note that there *are* efficient functional implementations of symbol tables, for instance using red-black trees or similar data structures, even if the dominant and classical implementation uses imperative hash tables.

The nesting level is instrumental in the way the symbol tables are used here (for sequential declarations).

In the particular AG (which is not a full type checker), the nesting level is *not* used in the case of variables, when one consults the symbol table, looking up the type of the variable. The simplified presentation here just checks if the variable is "contained" in the symbol table, but does not bother to find the type (and consequently does not make use of the nesting level, to find the *proper* binding, for instance under lexical scoping). What *is* done in this exercise is that the nesting level is used to make sure that are *not two declarations* in the same scope (i.e., nesting level).[3] In the AG here, the check is done in one of the semantic rules for $decl \to \mathbf{id} = exp$: if the variable is already stored in the ST *with the same nesting level, this results in an error.*[4] With this perspective, the sequential declarations from Table 2 are fairly simple (and discussed in the lecture as well).

Now, what needs to be *changed* in order to achieve a "collateral" form of declaration? We have to achieve that the declararations of a let-construct are added somehow "at once". Since the interface of the symbol table does not support "bulk addition" of bindings, we have to achieve that ourselves. Intuitively, the declarations in the declaration list are checked *all* with the same *symbol-table* (namely the one which is available at the beginning of the let construct). In absence of "bulk-addition" to the symbol table, we nonetheless need to incrementally add the bindings of the declaration list to the symbol table. This means intuitively, we need *two copies* of the symbol table: the one before the declaration lists starts, and the one we incrementally extend when walking down the declaration list (provided no error occurs).[5] So, in a practical imperative implementation, we need a copy of the symbol table in the state before walking down the declaration list, and symbol table updated while walking down that list.[6] In the declarative framework of the attribute grammar, this means we need *three attributes* representing various states of the symbol table.

They are called in the solution `addintab`, `addouttab` and `readintab`.

The first two attributes capture the pre- and post-states of the symbol table while walking down the declaration list, and `readintab` is the memorized state of the symbol-table *at the beginning* of the declaration list.

With this in mind, the semantic rules from Table 3 capture the desired behavior. Note how the `readintab` attribute is simply handed over unchanged when walking down the declaration list. In contrast, for the "additive" version of that symbol table, the "in-tab" of the rest gets its value from the "out-tab" of the declarations before (as was done in the original, sequential style of declaration).

Important is also the treatment of individual declarations, i.e., the treatment of the production $decl \to \mathbf{id} = exp$. In particular, the symbol table for the expression is `readintab`, the "memorized" symbol table. The rest is basically unchanged (modulo the fact that what what was called `intab` is now called `addintab` and analogous for attribute `outtab`. Note the typo in the corresponding actions in Table 3: `addintype` is meant as `addintab`.

□

Exercise 2 (AG for expression evaluation) Write an attribute grammar that computes the

[3] There may be alternative ways to achieve this, for instance using chained symbol tables, where the symbol table(s) itself takes care that not two entries are added. Instead of increasing a nesting level, the semantic action my involve creating a new symbol table in the "chain" of symbol tables. Still alternative designs are possible.

[4] This would be one point which might well be delegated to the symbol table: the ST itself makes sure that no declarations with the same nesting level are entered. That may be achieved by chaining, or by making the nesting level part of the key. That design may actually be a better choice, leading to a cleaner attribute grammar.

[5] The situation corresponds now more directly to a "functional" treatment of symbol tables to start with.

[6] That may not be the most (memory-)efficient way of doing it. One may choose other ways of achieving the same short of a full copy.

Grammar Rule	Semantic Rule
$S \rightarrow exp$	As before
$exp_1 \rightarrow exp_2 + exp_3$	As before
$exp_1 \rightarrow$ **(** exp_2 **)**	As before
$exp \rightarrow id$	As before
$exp \rightarrow num$	As before
$exp_1 \rightarrow$ **let** $dec\text{-}list$ **in** exp_2	$dec\text{-}list.addintab = exp_1.symtab$ $dec\text{-}list.readintab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel+1$ $exp_2.symtab = dec\text{-}list_2.addouttab$ $exp_2.nestlevel = exp_1.nestlevel+1$
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$, $decl$	$dec\text{-}list_2.addintab = dec\text{-}list_1.addintab$ $dec\text{-}list_2.readintab = dec\text{-}list_1.readintab$ $decl.addintab = dec\text{-}list_2.addouttab$ $decl.readintab = dec\text{-}list_1.readintab$ $dec\text{-}list_1.addouttab = decl.addouttab$

Grammar Rule	Semantic Rule
$dec\text{-}list \rightarrow decl$	$decl.addintab = dec\text{-}list.addintab$ $decl.readintab = dec\text{-}list.readintab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.addouttab = decl.addouttab$
$decl \rightarrow$ **id** = exp	$exp.symbtab = decl.readintab$ $decl.addouttab =$ **if** $((decl.addintype = errtab)$ **or** $exp.err)$ **then** $errtab$ **else if** $(lookup(decl.addintype,$**id**$.name) =$ $decl.nestlevel)$ **then** $errtab$ **else** $insert(decl.addintab,$**id**$.name,decl.nestlevel)$

Table 3: Collateral declarations

value of each expression for the expression grammar Table 1 (it's the same as in the previous exercise).

Solution: As discussed in the lecture and as the topic of the previous exercise: declarations can be interpreted *sequential* or as "*collateral*"/simultaneous. That does not just influence which (syntactically correct) declarations are type correct, it also influences the *evaluation.* The difference between the interpretations (sequential vs. collateral) is non-syntactic; both are based on the same context-free grammar. The difference is in the context-sensitive part (here the AG). The exercise asks to evaluate expressions as given in the lecture (and not as in the previous exercise) , which means we assume that the declarations are meant as *sequential* declarations (it's easier to capture anyhow).

Now, what makes this evaluation different from the previous evaluations of expressions in the lecture is that we have *variables* and their declarations/definitions! In the same way that type declaration can "fix" the type for a variable, the let-declaration now can fix the value (via an expression) of a variable. Note that the language is "single-assignment", it's a declarative language (resp. language fragment): variables cannot be "updated" imperatively.

In the solution, we assume that *errors* can occur. That happens if one needs the value of a

variable which has not been declared/defined.[7] This means, an expression can have a "standard" value, which in the grammar is "numerical", and an exceptional one (here "error"). An error is injected into the evaluation when looking up a variable in the symbol table for which no value us found. Once, an error occurs (in that a value becomes `error` or a value of a symbol table becomes `errtab`, the error state will propagate so that the overall expression is erroneous.[8]

But the rule of declarations of *values* are the same as for declaration of *types!* Therefore the rules very much resemble the ones for type declarations (in the sequential setting).

The solution is shown in Table 4.

Grammar Rule	Semantic Rule
$exp_1 \rightarrow exp_2 + exp_3$	exp_1.val = **if** (exp_2.val = **error**) **or** (exp_3.val = **error**) **then error** **else** exp_2.val + exp_3.val
$exp_1 \rightarrow (exp_2)$	exp_1.val = exp_2.val
$exp \rightarrow$ ***id***	exp.val = lookupVal(exp.symtab, id.name)
$exp \rightarrow$ ***num***	exp.val = ***num***.val
$exp_1 \rightarrow$ ***let*** dec-list ***in*** exp_2	exp_1.val = **if** (decl-list.outtab = errtab) **then error** **else** exp_2.val
decl → ***id*** = exp	decl.outtab = **if** (decl.intab = errtab) **then** errtab **else** **if** (lookupLevel(decl.intab,id.name) = decl.nestlevel) **then** errtab **else** insert(decl.intab,id.name, decl.nestlevel, exp.val)

Table 4: Evaluation

□

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

$$\begin{aligned} exp \quad \rightarrow \quad & exp + exp \mid exp - exp \mid exp * exp \mid exp \,/\, exp \\ \mid \quad & (\, exp \,) \mid \textbf{num} \mid \textbf{num} \,.\, \textbf{num} \end{aligned}$$

[7] If we assumed that the expression would not have undefined/undeclared variables, the AG would be slightly simpler. It would not change much in principle as far as the treatment of the attributes/symbol table is concerned. Basically, we assume there are "standard" values and "error" values.

[8] That corresponds to exceptions: once an exception is raised (but not caught), it will propagate to the top-level of the program/expression). We may also not the following: later in the lecture, for code generation, there will be a section on how to generate code for booleans. At that point the code will often use something like "short-circuit evaluation". What we are doing here with evaluating expression via the AG is "non-short-circuiting".

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 some languages: 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.

Solution: The key is to introduce information, i.e., an attribute, keeping track if a numerical expression is a float or not. That can be interpreted as *type* information.[9] Here, since we have only two cases —a floating point number or an integer— it is captured by a *boolean* attribute `isFloat`. Apart from that: since we are requested to evaluate the expression, we also add an attribute `val`. In this exercise, there are no declarations and variables, so the attribute is purely synthesized (as `isFloat`). In the evaluation, we have to *convert* sometimes integer values to floats, which is done with `FLOAT`. In the rule for division, a case distinction is done based on the "type" of the subexpressions, as required by the exercise.

Note that the "type" exp_1 `.isFloat` is determined *first* and the (sibling) attribute *exp* `.val` *depends* on it.

The solution is given in Table 5. □

Exercise 4 (Type equality and type checking)

Consider the following grammar which in particular features procedure or function declarations (Table 6)

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

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

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

Solution: The grammar is given in Table 7. It's a extension of grammars given in the lecture.

The clauses for function declaration could be represented as tree as shown in Table 8.

Table 9 shows a possible AG for type checking the language, concentrating on the new language elements. One new element is that the grammar has a clause for *exps*, i.e., for tuples of expression. That's needed for the arguments of functions. The way that it's dealt with here is using an attribute `types`, representing *lists* (i.e., tuples) of types.

That's not the only conceivable solution. Instead of introducing

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.

[9] Oftentimes, at least in simple settings, the result of the type checker is an (updated) AST, where the nodes contain the type of the corresponding syntactic element (typically in a field of the node, and the fields can be seen directly as concrete implementation of attributes. In the task at hand, we have a restricted settting insofar that we concentrate on numerical expressions only and that we are consequently interested in two types only (floats and ints).

Grammar Rule	Semantic Rule
$exp_1 \rightarrow exp_2 + exp_3$	exp_1.isFloat = exp_2.isFloat **or** exp_3.isFloat exp_1.val = **if** exp_1.isFloat **then** floatAdd(**if** not exp_2.isFloat **then** FLOAT(exp_2.val) **else** exp_2.val, **if** not exp_3.isFloat **then** FLOAT(exp_3.val) **else** exp_3.val) **else** intAdd(exp_2.val, exp_3.val)
$exp_1 \rightarrow exp_2 / exp_3$	exp_1.val = **if** (**not** exp_2.isFloat **and** **not** exp_3.isFloat) **then** exp_2.val **div** exp_3.val **else** exp_2.val / exp_3.val
$exp_1 \rightarrow (exp_2)$	exp_1.val = exp_2.val
$exp \rightarrow$ **num**	exp.isFloat = false exp.val = **num**.val
$exp \rightarrow$ **num.num**	exp.isFloat = true exp.val = **num**.val

Table 5: Evaluation/typing of floating point expression

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.

Solution: This one is a bit of a discussion question. However, the situation is not "nice", the situation is not a sign of a well-thought-of language design. What is "un-nice" is that the same surface syntax is interpreted differently depending on the context. In other words, the compiler needs "context-sensitive" information in order to do the parsing actually. An alternative would be to make a node in the AST which covers both interpretations, which is afterwards made unambiguous by the semantic analysis. Also pragmatically it's not desirable that it's hard for a human to understand "syntax" in such sitations. In any way, the *syntactical* aspects of a language (captured by context-free grammars) should not be mixed and messed up by context-sensitive information. On the "conceptual" side: especially attribute grammars can *obviously* not help the parser in that it assists in disambuation while parsing: *semantic rule part cannot influence which rule to take.* The AG framework only *adds* semantic rules to existing grammar production, but does not influence the parsing itself.

In the parser: Under parsing the parser can consult the symbol-table, looking up `A`. The symbol table then needs to be designed to contain both names for ordinary variables and names for types.[10][11] Based on that information, the parser can create an AST node/tree representing a type cast or else a node/tree represeting a substraction expression.

[10] One can also view names for types as type variables, especially if they are used as type synonyms.

[11] Alternatively, one may in practice have 2 symbol tables, one for ordinary names and one for names standing for types, as perhaps the two name spaces (for type names and for variable names) follow different rules.

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 6: Grammar with function declarations

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*] : *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 7: Statements with function declarations

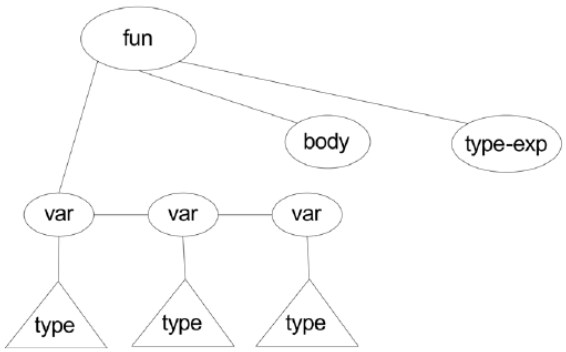


Table 8: Syntax tree for types function declarations

In the lexer: Basically, the situation is similar, except that now already the scanner must consult the symbol table (the parser as well, as the parser is still responsible to add bindings to the table, like information about `A` at the point where `A` is declared, and that information can be seen then as being "inherited" in the AG way of thinking). Since the

Grammar Rule	Semantic Rule
fun-decls → **fun id (***var-decls***):** *type-exp ; body*	*fun-decls.type = makeTypeNode (fun, var-decls.types, type-exp.type)*
$var\text{-}decls_1$ → $var\text{-}decls_2$;*var-decl*	$var\text{-}decls_1.types$ = $var\text{-}decls_2.types$ + *var-decl.type*
var-decls → *var-decl*	*var-decls.types = var-decl.type*
exp → **id(***exps***)**	**if** *isFunType(lookup(***id***.name))* **and** *exps.types = parameterTypesOf(***id***.name)* **then** *exp.type = lookup(***id***.name)* **else** *type-error*
$exps_1$ → $exps_2$, *exp*	$exps_1.types = exps_2.types$ + *exp.type*
exps → *exp*	*exps.types = exp.type*

Table 9: Attribute grammar for typing function declarations

scanner now makes the decision, it would generate `two different tokens`, depending on the context-information from the symbol table. Then, the parser can directly generate the proper syntax tree.