Compiler Theory CS447b: Quiz 7. **UWO, March 23, 2004.**

Student ID number:

Guidelines. The quiz is closed book and all notes are forbidden. The duration of the quiz is 30 minutes. Exercise 1 is worth 4 points whereas each of the other two is worth 8 points. All answers should be written in the *answer boxes*. No justifications for the answers are needed **unless** explicitly required. You are expected to do this exam on your own without assistance from anyone else in the class. If possible, please avoid pencils and use pens with dark ink. Thank you.

Exercise 1. We consider the following translation scheme.

S	$\longmapsto$	$A\ \{C.i := A.s\}\ C$
S	$\longmapsto$	$AB\ \{C.i := A.s\}\ C$
A	$\longmapsto$	**int** $\{A.s :=$ **int**$.val\}$
B	$\longmapsto$	**id** $\{\}$
C	$\longmapsto$	**int** $\{C.s := C.i +$ **int**$.val\}$

where S, A, B, C are non-terminals, **int** and **id** are terminals, $C.i$ is an inherited attribute, $A.s$ and **int**$.val$ are synthesized attributes. Write a YACC program for this translation scheme.

Answer 1

```
S      : A C
       | A B M C
       ;
A      : TOK_INT {$$ = $1; }
       ;
B      : TOK_ID
       ;
M      :   /* empty */  {$$ = $-2; }
       ;
C      :  TOK_INT {$$ = $-1 + $1; }
       ;
```

Exercise 2. We consider the following grammar. As in the course, the nonterminals P, D, S, T, E stand for *Program*, *Declaration*, *Statement*, *Type*, *Expression*. The terminals **id**, **boolean**, **integer**, **literal**, **num** stand for *identifier*, *boolean* (as a type), *integer* (as a type), *boolean literal* (as a value), *integer number* (as a value).

$$
\begin{array}{|l|}
\hline
P \longmapsto D;\ S \\
D \longmapsto D;\ D \\
D \longmapsto \mathbf{id} bf : T \\
T \longmapsto \mathbf{boolean} \\
T \longmapsto \mathbf{integer} \\
E \longmapsto \mathbf{literal} \\
E \longmapsto \mathbf{num} \\
E \longmapsto \mathbf{id} \\
E \longmapsto E_1 + E_2 \\
E \longmapsto E_1 = E_2 \\
S \longmapsto S_1;\ S_2 \\
S \longmapsto E \\
S \longmapsto \mathbf{id}{:=}\, S_1 \\
S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ \{S_1\} \\
S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\} \\
\hline
\end{array}
$$

There's one s light difference w.r.t. the grammar of the course in the *Type checking* chapter: each valid statement has a value and thus a type (rather than no value and type `void`). The following rules compute the value of a valid statement.

- if S_1 and S_2 are valid statements then the value of $\boxed{S_1;\ S_2}$ is that of S_2.
- if $\boxed{\mathbf{id}{:=}\, S_1}$ is a valid statement then its value is that of S_1.
- if $\boxed{\mathbf{if}\ E\ \mathbf{then}\ \{S_1\}}$ is a valid statement then its value is that of S_1.
- if $\boxed{\mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}}$ is a valid statement then its value is that of S_1.

We associate attributes $T.type$, $E.type$, $S.type$, to the grammar symbols T, E, S. Each of these attributes may be **boolean**, **integer** or **type_error**. Then, for instance, the statement $\boxed{\mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}}$ is valid as soon as $E.type = \mathbf{boolean}$ and $S_1.type \neq \mathbf{type\_error}$. The attribute $\mathbf{id}.entry$ refers to the entry of **id** in the symbol table.

Question. Complete the following type checker for the above grammar.

Answer 2

Some of the actions below can be *simplified*. But I prefer to give a *clear* correction rather than *too much elegant*.

$D \longmapsto \textbf{id}{:}\,T$		$\{$ addtype($\textbf{id}.entry$, $T.type$) $\}$
$T \longmapsto \textbf{boolean}$	$\{\ T.type :=$	$\textbf{boolean}\ \}$
$T \longmapsto \textbf{integer}$	$\{\ T.type :=$	$\textbf{integer}\ \}$
$E \longmapsto \textbf{literal}$	$\{\ E.type :=$	$\textbf{boolean}\ \}$
$E \longmapsto \textbf{num}$	$\{\ E.type :=$	$\textbf{integer}\ \}$
$E \longmapsto \textbf{id}$	$\{\ E.type :=$	lookup($\textbf{id}.entry$) $\}$
$E \longmapsto E_1 + E_2$	$\{\ E.type :=$	if $E_1.type = \textbf{integer}$ and $E_2.type = \textbf{integer}$ then $\textbf{integer}$ else $\textbf{type\_error}\ \}$
$E \longmapsto E_1 = E_2$	$\{\ E.type :=$	if $E_1.type = E_2.type$ and $E_1.type \neq \textbf{type\_error}$ then $E_1.type$ else $\textbf{type\_error}\ \}$
$S \longmapsto S_1;\ S_2$	$\{\ S.type :=$	if $S_1.type \neq \textbf{type\_error}$ and $S_2.type \neq \textbf{type\_error}$ then $S_2.type$ else $\textbf{type\_error}\ \}$
$S \longmapsto E$	$\{\ S.type :=$	if $E.type \neq \textbf{type\_error}$ then $E.type$ else $\textbf{type\_error}\ \}$
$S \longmapsto \textbf{id}{:=}S_1$	$\{\ S.type :=$	if $S_1.type = \textbf{id}.type$ then $S_1.type$ else $\textbf{type\_error}\ \}$
$S \longmapsto \textbf{if}\ E\ \textbf{then}\ \{S_1\}$	$\{\ S.type :=$	if $E.type = \textbf{boolean}$ then $S_1.type$ else $\textbf{type\_error}\ \}$
$S \longmapsto \textbf{while}\ E\ \textbf{repeat}\ \{S_1\}$	$\{\ S.type :=$	if $E.type = \textbf{boolean}$ then $S_1.type$ else $\textbf{type\_error}\ \}$

Exercise 3. We extend the grammar of Exercise 2 in order to handle unary functions with a single returned value. Here's a program showing these new features.

```
f: integer -> integer;
f(n: integer): integer == {
  if n = 0 then {1} else {n * f(n-1)};
}
```

To do so, we add productions for generating the type of a function, the definition of a function, and a call to a function. Here's the enhanced grammar.

$$
\begin{array}{|l|}
\hline
P \longmapsto D;\ S \\
D \longmapsto D;\ D \\
D \longmapsto \mathbf{id} : T \\
T \longmapsto A \\
A \longmapsto \mathbf{boolean} \\
A \longmapsto \mathbf{integer} \\
T \longmapsto A_1 \to A_2 \\
E \longmapsto \mathbf{literal} \\
E \longmapsto \mathbf{num} \\
E \longmapsto \mathbf{id} \\
E \longmapsto E_1 + E_2 \\
E \longmapsto E_1 = E_2 \\
E \longmapsto \mathbf{id}(E_1) \\
S \longmapsto S_1;\ S_2 \\
S \longmapsto E \\
S \longmapsto \mathbf{id}{:=}\,S_1 \\
S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ \{S_1\}\mathbf{else}\ \{S_2\} \\
S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\} \\
S \longmapsto \mathbf{id}\,(\mathbf{id} :\ A_1){:}A_2 == \{S_1\} \\
\hline
\end{array}
$$

To keep things simple, observe that we impose the following constraints.

(C_1) The input type and the output type of a function can only be *primitive types*, that is **boolean** or **integer**. This appears in the above grammar.

(C_2) In our language, there is **only one** scope for variables and functions. In other words, every variable or function is global.

(C_3) In addition, each formal parameter of a function is viewed as a variable.

These constraints imply that **only one** symbol table is needed, as in the previous exercise. To denote the type of a function you can use a record of the form $[in : T_1, out : T_2]$. So, coming back to the above example

- parsing the first line would create an entry like $\boxed{f \mid [in:\ integer, out:\ integer]}$ in the symbol table,
- parsing the function definition would create an entry like $\boxed{n \mid integer}$.

Finally, since in this language every statement has a value (see the previous exercise), a function definition must have a value too. This value is simply the defined function. So, coming back again to the above example, the type of the statement corresponding to the function definition is just $[in:\ integer, out:\ integer]$.
Question. Complete the type checker for the following rules. You can use any pseudo-code style that you like or even write sentences like *if there exists an entry e with type T then T.*

Answer 3

$T \longmapsto A_1 \rightarrow A_2$	{ $T.type$:=	$[in:\ A_1.type, out:\ A_2.type]$ }
$E \longmapsto \mathbf{id}(E_1)$	{ $E.type$:=	**if** there exists a declared function called $\mathbf{id}.name$ and with type $[in:\ E_1.type, out:\ E_2.type]$ **then** $E_2.type$ else **type_error** }
$S \longmapsto \mathbf{id}_1\ (\mathbf{id}_2 :\ A_1){:}A_2 == \{S_1\}$	{ $S.type$:=	addtype($\mathbf{id}_2.entry$, $A_1.type$) **if** there exists a declared function called $\mathbf{id}_1.name$ and with type $[in:\ A_1.type, out:\ A_2.type]$ **and** $A_2.type = S_1.type$ **then** $[in:\ A_1.type, out:\ A_2.type]$ else **type_error** }

Exercise 4. This is a **bonus exercise**. You will receive 4 extra marks if you can explain how to modify the grammar and the type checker of Exercise 3 such that the revised language has

- functions returning not only primitive types but also functions,
- local definition-declarations in function bodies.

As a hint, we give the revised rule for generating functions types

$$\boxed{T \longmapsto A_1 {\rightarrow} (T_2)}$$

Here's a program that must be accepted by the revised grammar and type checker.

```
f: integer -> (integer -> (integer));
f(n: integer): integer -> (integer) == {
    g: integer -> (integer);
    g(m: integer): integer == { n + m; };
}
```

Only a brief explanation is needed, not a translation scheme.

Answer 4

One needs one symbol table for the global variables and functions and one symbol table per function body. For this to happen a statement must become a program. That is we must add a rule like

$$\boxed{S \longmapsto P}$$

and state that a type of a program is the type of its statement sequence.
The mechanisms for declaring, defining and calling a function remain essentially the same. The rules for declaring and defining just need to become

$$\boxed{\begin{array}{l} T \longmapsto A_1 {\rightarrow} (T_2) \\ S \longmapsto \mathbf{id}_1\,(\mathbf{id}_2 :\ A_1){:}T_2 == \{S_1\} \end{array}}$$