# Exercises on Compilers

**Jordi Cortadella**

February 7, 2022

## Semantic Analysis

1. We look at a simple language with an exception mechanism:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{throw id} \\
S & \rightarrow & S \ \texttt{catch id} \Rightarrow S \\
S & \rightarrow & S \ \texttt{or} \ S \\
S & \rightarrow & \textit{other}
\end{array}
$$

A `throw` statement throws a named exception. This is caught by the nearest enclosing `catch` statement (i.e., where the throw statement is in the left sub-statement of the catch statement) using the same name, whereby the statement after the arrow in the catch statement is executed. An `or` statement is a non-deterministic choice between the two statements, so either one can be executed. *other* is a statement that does not throw any exceptions. We want the type checker to ensure that all possible exceptions are caught and that no catch statement is superfluous, i.e., that the exception it catches can, in fact, be thrown by its left sub-statement. Write type-check functions that implement these checks.

**Hint:** Let the type of a statement be the set of possible exceptions it can throw.

**Solution:**
We define a new synthesized attribute for $S$ called $S.except$ that represents the set of uncaught exceptions of the statements executed by $S$. The attribute is calculated as follows:

$$
\begin{array}{ll}
S \rightarrow \texttt{throw id} & \{S.except = \{\texttt{id}\}\} \\
S \rightarrow S_1 \ \texttt{catch} \ \{\text{id}\} \Rightarrow S_2 & \{S.except = (S_1.except \setminus \texttt{id}) \cup S_2.except\} \\
S \rightarrow S_1 \ \texttt{or} \ S_2 & \{S.except = S_1.except \cup S_2.except\} \\
S \rightarrow \textit{other} & \{S.except = \emptyset\}
\end{array}
$$

An error exists if $S.except \neq \emptyset$ at the topmost statement of a function.
A `catch` statement is redundant if $\texttt{id} \notin S_1$ in the second production, meaning that the statement is prepared to catch `id` but $S_1$ never throws it.

2. [From Aho-Sethi-Ullman's book.]

Let us consider the grammar of a simple language, represented by the nonterminal $P$, consisting of a sequence of declarations $D$ followed by a single expression $E$.

$$
\begin{array}{lcl}
P & \rightarrow & D \ ; \ E \\
D & \rightarrow & D \ ; \ D \mid \texttt{id} \ : \ T \\
T & \rightarrow & \texttt{char} \mid \texttt{integer} \mid \texttt{array [num] of} \ T \mid * T \\
E & \rightarrow & \texttt{literal} \mid \texttt{num} \mid \texttt{id} \mid E \ \texttt{mod} \ E \mid E \ \texttt{[} \ E \ \texttt{]} \mid E*
\end{array}
$$

One program generated by the grammar is:

```
key: integer;
key mod 1999
```

The language has two basic types, *char* and *integer*; a third basic type *type_error* is used to signal errors. For arrays, the number inside the square brackets represents it size. For exmaple,

```
array [256] of char
```

leads to the type expression *array*(256, *char*). The prefix operator $*$ in declarations builds a pointer type, so `*integer` leads to the type expression *pointer(integer)*.

The action associated with the production $D \rightarrow \texttt{id} \ : \ T$ saves a type in the symbol-table entry for an identifier. The action *addtype(id.entry, T.type)* is applied to the synthesized attribute *entry* pointing to the symbol-table entry for `id` and a type expression represented by the synthesized attribute *entry* of the nonterminal $T$.

If $T$ generates `char` or `integer`, then $T.type$ is defined to be *char* or *integer*, respectively. The upper bound of an array is obtained from the attribute *val* of token `num` that gives the integer represented by `num`. Since the declarations appear before the expression, we can be sure that the types of all declared indentifiers will be saved in the symbol table before the expression is checked.

Next, part of the semantic actions that save the type of an identifier in the symbol table is shown:

$$
\begin{array}{ll}
P \rightarrow D \ ; \ E & \\
D \rightarrow D \ ; \ D & \\
D \rightarrow \texttt{id} \ : \ T & \{ \ \mathit{addtype}(\texttt{id}.\mathit{entry}, \ \mathrm{T}.\mathit{type}) \ \} \\
T \rightarrow \texttt{char} & \{ \ T.\mathit{type} \ := \ \mathit{char} \ \} \\
T \rightarrow \texttt{integer} & \{ \ T.\mathit{type} \ := \ \mathit{integer} \ \} \\
T \rightarrow *T_1 & \{ \ T.\mathit{type} \ := \ \mathit{pointer}(T_1.\mathit{type}) \ \} \\
T \rightarrow \texttt{array [num] of} \ T_1 & \{ \ T.\mathit{type} \ := \ \mathit{array}(\texttt{num}.\mathit{val}, \ T_1.\mathit{type}) \ \}
\end{array}
$$

The tokens `literal` and `num` have type *char* and *integer*, respectively. The expressions formed by applying the `mod` operator to two subexpressions of type *integer* has type *integer*; otherwise, its type is *type_error*. In an array referece $E_1[E_2]$, the index expression $E_2$ must have type *integer*. The type of $E*$ is the type $t$ of the object pointed to by the pointer $E$.

Add the type checking actions of the grammar to synthesize the attribute $E.type$ for the expression. We can use the function *lookup*(`id`.*entry*) to fetch the type saved in the symbol table for the entry corresponding to the identifier `id`.

**Solution:**

$E \rightarrow$ `literal` $\quad \{\ E.type\ :=\ char\ \}$

$E \rightarrow$ `num` $\quad \{\ E.type\ :=\ integer\ \}$

$E \rightarrow$ `id` $\quad \{\ E.type\ :=\ lookup(\texttt{id}.entry)\ \}$

$E \rightarrow E_1$ `mod` $E_2 \quad \{\ E.type\ :=\ (E_1.type = integer\ \texttt{and}\ E_2.type = integer)\ ?\ integer\ :\ type\_error\ \}$

$E \rightarrow E_1[E_2] \quad \{\ E.type\ :=\ (E_2.type = integer\ \texttt{and}\ E_1.type = array(s,t))\ ?\ t\ :\ type\_error\ \}$

$E \rightarrow E_1* \quad \{\ E.type\ :=\ E_1.type = pointer(t)\ ?\ t\ :\ type\_error\ \}$

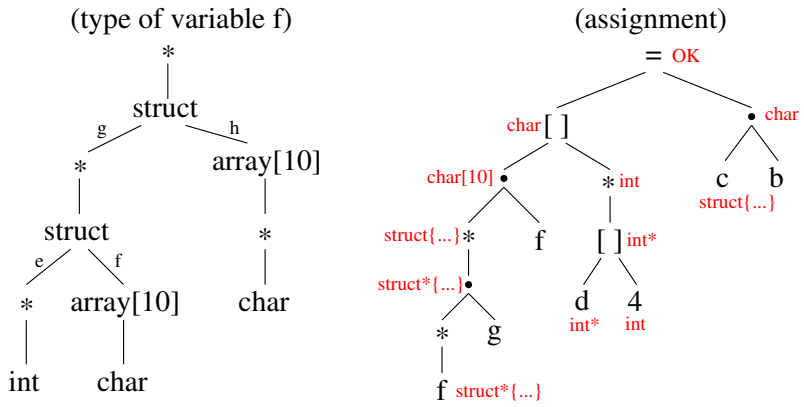
3. Let us consider the following fragment of C code:

```
struct {
    int a;
    char b;
} c;

(int *) d[10];

struct {
    struct {
        int *e;
        char f[10];
    } *g;
    (char *) h[10];
} e, *f;
```

```
1: (*((*f).g)).f[*(d[4])] = c.b;
2: e.h[11] = &(c.b);
3: *(e.g + int) = *((*f).g);
4: d[12] = &(c.a+1);
5: c.a = e.2[h];
6: (*f)..h[1];
```

- Give an AST for the type of variable `f`. Use the label `struct` to represent the root node of a struct's AST and the label `array[`$n$`]` to represent the root node of an array of $n$ elements. When representing a `struct`, annotate the label of each field at the tree edge associated to the field.
- Give an AST representing the assignment in line 1 and infer the type associated to each node of the AST. Represent the access to a `struct` (`x.y`) as a binary tree `(.,x,y)` and the access to an array (`a[i]`) as a binary tree `([],a,i)`. Is it a correct assignment?
- Detect all possible errors in the lines of code 1-6. Indicate when the error is detected (lexical, syntax, semantic, runtime).
- Indicate when the following errors are detected (lexical, syntax, semantic, runtime):
  - `&` can only be used for an addressable memory location.
  - Missing semicolon at the end of a statement.
  - The LHS and RHS types of an assignment are not compatible.
  - A character does not belong to the alphabet of the language.
  - An array index is out of bounds.
  - An array index must be integer.
  - Unclosed parenthesis in an expression.
  - Division by zero.
  - Unclosed quotes (`"`) of a string.
  - Only integer values can be added to pointers.

**Solution:**
Below are the ASTs of the type of variable `f` and the assignment. In red, the type of the nodes of the assignment AST.




Errors in lines 1-6:

- Line 1: Only runtime errors could be produced, e.g. access using a NULL pointer, or access to array `f` with an index out of bounds.
- Line 2: Access to array with index out of bounds. This error is typically produced at runtime, although a compiler with powerful static analysis could also detect it during semantic analysis.
- Line 3: `int` is a keyword and the presence in the expression would produce a syntax error. Obtaining a point from an expression (+) is a semantic error.
- Line 4: Access to array with index out of bounds. This error is typically produced at runtime, although a compiler with powerful static analysis could also detect it during semantic analysis. Moreover, the expression `c.a+1` is not addressable, i.e., no address can be obtained from it.
- Line 5: syntax error. An identifier must follow the dot that precedes the field of a structure.
- Line 6: syntax error. An identifier must follow the dot that precedes the field of a structure.

Errors in the code:

| | |
|---|---|
| `&` can only be used for an addressable memory location | Semantic |
| Missing semicolon at the end of a statement | Syntax |
| The LHS and RHS types of an assignment are not compatible | Semantic |
| A character does not belong to the alphabet of the language | Lexical |
| An array index is out of bounds | Runtime |
| An array index must be integer | Semantic |
| Unclosed parenthesis in an expression | Syntax |
| Division by zero | Runtime |
| Unclosed quotes (") in a string | Lexical |
| Only integer values can be added to pointers | Semantic |