Lab Session 6
Static Typing of a Fragment of C Language

Interpretation and Compilation of Languages

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of April 14, 2025

1 Introduction

The goal is to build a typechecker for a tiny fragment of the C language, called Mini C in the following. it contains integers and pointers to structures. It is fully compatible with C. This means a C compiler such as `gcc` can be used as a reference.

Differences wrt C. Any Mini C program is a legal C program. Yet, Mini C has limitations wrt C. Here are some of them:

- There is no variable initialization. To initialize a variable, one has to use an assignment;
- the only types are integers (Basic signed 32 integer type `int`), pointers to the structures (`struct` *id* `*`), and void pointer type, `void *`, (e.g. used for the return type of `malloc`);
- There is no pointer arithmetic (and no memory deallocation);
- Mini C has fewer keywords than C.

Predefined Functions. The following functions are predefined:

```
int putchar(int c);
void *malloc(int n);
```

(But there is no need for `#include` in Mini Cfor testing.)

2 Syntax

We use the following notations in grammars:

$\langle rule \rangle^\star$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero)
$\langle rule \rangle^\star_t$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero), with separator t
$\langle rule \rangle^+$	repeats $\langle rule \rangle$ at least once
$\langle rule \rangle^+_t$	repeats $\langle rule \rangle$ at least once, with separator t
$\langle rule \rangle?$	use $\langle rule \rangle$ optionally
($\langle rule \rangle$)	grouping

Be careful not to confuse "$\star$" and "$+$" with "`*`" and "`+`" that are C symbols. Similarly, do not confuse grammar parentheses with terminal symbols `(` and `)`.

2.1 Lexical Conventions

Spaces, tabs, and newlines are blanks. Comments are of two kinds:

- from `/*` to `*/` and not nested;
- from `//` to the end of the line.

Identifiers follow the regular expression $\langle ident \rangle$:

$$
\begin{array}{lcl}
\langle \mathit{digit} \rangle & ::= & \texttt{0--9} \\
\langle \mathit{alpha} \rangle & ::= & \texttt{a--z} \mid \texttt{A--Z} \\
\langle \mathit{ident} \rangle & ::= & (\langle \mathit{alpha} \rangle \mid \texttt{\_})\ (\langle \mathit{alpha} \rangle \mid \langle \mathit{digit} \rangle \mid \texttt{\_})^\star
\end{array}
$$

The following identifiers are keywords:

```
int struct if else while return sizeof
```

Last, integer literals follow the regular expression $\langle integer \rangle$:

$$
\begin{array}{lcl}
\langle \mathit{integer} \rangle & ::= & \texttt{0} \\
 & \mid & \texttt{1--9}\ \langle \mathit{digit} \rangle^\star \\
 & \mid & \texttt{0}\ \langle \mathit{digit\text{-}octal} \rangle^+ \\
 & \mid & \texttt{0x}\ \langle \mathit{digit\text{-}hexa} \rangle^+ \\
 & \mid & \texttt{'}\ \langle \mathit{character} \rangle\ \texttt{'} \\
\langle \mathit{digit\text{-}octal} \rangle & ::= & \texttt{0--7} \\
\langle \mathit{digit\text{-}hexa} \rangle & ::= & \texttt{0--9} \mid \texttt{a--f} \mid \texttt{A--F} \\
\langle \mathit{character} \rangle & ::= & \text{any ASCII character with a code in } [32, 127], \\
 & & \text{other than } \texttt{\textbackslash},\ \texttt{'}, \text{ and } \texttt{"} \\
 & \mid & \texttt{\textbackslash\textbackslash} \mid \texttt{\textbackslash'} \mid \texttt{\textbackslash"} \\
 & \mid & \texttt{\textbackslash x}\ \langle \mathit{digit\text{-}hexa} \rangle\ \langle \mathit{digit\text{-}hexa} \rangle
\end{array}
$$

2.2 Syntax

The grammar of source files is given in Fig. 1. The entry point is $\langle file \rangle$. Associativity and priorities are given below, from lowest to strongest priority.

operation	associativity	priority
`=`	right	lowest
`||`	left	
`&&`	left	
`==` `!=`	left	
`<` `<=` `>` `>=`	left	↓
`+` `-`	left	
`*` `/`	left	
`!` `-` (unary)	right	
`->`	left	strongest

$$
\begin{array}{lcl}
\langle \textit{file} \rangle & ::= & \langle \textit{decl} \rangle^\star \ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \langle \textit{decl\_typ} \rangle \ \mid \ \langle \textit{decl\_fct} \rangle \\
\langle \textit{decl\_vars} \rangle & ::= & \texttt{int} \ \langle \textit{ident} \rangle^+_{,} \ \texttt{;} \\
 & \mid & \texttt{struct} \ \langle \textit{ident} \rangle \ (\texttt{*} \ \langle \textit{ident} \rangle)^+_{,} \ \texttt{;} \\
\langle \textit{decl\_typ} \rangle & ::= & \texttt{struct} \ \langle \textit{ident} \rangle \ \texttt{\{} \ \langle \textit{decl\_vars} \rangle^\star \ \texttt{\}} \ \texttt{;} \\
\langle \textit{decl\_fct} \rangle & ::= & \texttt{int} \ \langle \textit{ident} \rangle \ \texttt{(} \ \langle \textit{param} \rangle^\star_{,} \ \texttt{)} \ \langle \textit{bloc} \rangle \\
 & \mid & \texttt{struct} \ \langle \textit{ident} \rangle \ \texttt{*} \ \langle \textit{ident} \rangle \ \texttt{(} \ \langle \textit{param} \rangle^\star_{,} \ \texttt{)} \ \langle \textit{bloc} \rangle \\
\langle \textit{param} \rangle & ::= & \texttt{int} \ \langle \textit{ident} \rangle \ \mid \ \texttt{struct} \ \langle \textit{ident} \rangle \ \texttt{*} \ \langle \textit{ident} \rangle \\
\langle \textit{expr} \rangle & ::= & \langle \textit{integer} \rangle \\
 & \mid & \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{expr} \rangle \ \texttt{->} \ \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{ident} \rangle \ \texttt{(} \ \langle \textit{expr} \rangle^\star_{,} \ \texttt{)} \\
 & \mid & \texttt{!} \ \langle \textit{expr} \rangle \ \mid \ \texttt{-} \ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle \ \langle \textit{binop} \rangle \ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{ident} \rangle \ \texttt{=} \ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle \ \texttt{->} \ \langle \textit{ident} \rangle \ \texttt{=} \ \langle \textit{expr} \rangle \\
 & \mid & \texttt{sizeof} \ \texttt{(} \ \texttt{struct} \ \langle \textit{ident} \rangle \ \texttt{)} \\
 & \mid & \texttt{malloc} \ \texttt{(} \ \texttt{struct} \ \langle \textit{ident} \rangle \ \texttt{)} \\
 & \mid & \texttt{(} \ \langle \textit{expr} \rangle \ \texttt{)} \\
\langle \textit{binop} \rangle & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle \textit{stmt} \rangle & ::= & \texttt{;} \\
 & \mid & \langle \textit{expr} \rangle \ \texttt{;} \\
 & \mid & \texttt{if} \ \texttt{(} \ \langle \textit{expr} \rangle \ \texttt{)} \ \langle \textit{stmt} \rangle \\
 & \mid & \texttt{if} \ \texttt{(} \ \langle \textit{expr} \rangle \ \texttt{)} \ \langle \textit{stmt} \rangle \ \texttt{else} \ \langle \textit{stmt} \rangle \\
 & \mid & \texttt{while} \ \texttt{(} \ \langle \textit{expr} \rangle \ \texttt{)} \ \langle \textit{stmt} \rangle \\
 & \mid & \langle \textit{bloc} \rangle \\
 & \mid & \texttt{return} \ \langle \textit{expr} \rangle \ \texttt{;} \\
\langle \textit{bloc} \rangle & ::= & \texttt{\{} \ \langle \textit{decl\_vars} \rangle^\star \ \langle \textit{stmt} \rangle^\star \ \texttt{\}}
\end{array}
$$

Figure 1: Grammar of Mini C.

3 Static Typing

Once parsing phase is completed (provided in the lab assignment), we explain how to perform static typing of Mini C.

Types and Typing Environments. Expressions have types τ with the following abstract syntax

$$\tau ::= \texttt{int} \mid \texttt{struct}\ id\ \texttt{*} \mid \texttt{void*}$$

where *id* stands for a structure name. We introduce the relation $\equiv$ over types as the smallest reflexive and symmetric relation that additionally satisfies the equation $\texttt{void*} \equiv \texttt{struct}\ id\ \texttt{*}$.

A typing environment Γ is a sequence of variable declarations $\tau\ x$, structure declarations $\texttt{struct}\ S\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}}$ and function declarations $\tau\ f(\tau_1, \dots, \tau_n)$. We write $\texttt{struct}\ S\ \texttt{\{}\tau\ x\texttt{\}}$ to indicate that structure S has a field x with type τ.

We say that a type τ is *well-formed* in environment Γ, and we write $\Gamma \vdash \tau\ \mathsf{bf}$, if all structure names in τ correspond to structures declared in Γ.

Uniqueness Rules. In addition to the typing rules below (for structure declarations, expressions, statements and function declarations), we have to check for uniqueness

- of structure names over the whole file;
- of structure fields inside a *single* structure;
- of function parameters;
- of local variables inside a *single* block;
- of function names over the whole file.

3.1 Adding Structure Declarations to the typing environment

A file is a list of structure and function declarations (there are no global variables in Mini C). We first add structure declarations to the typing environment. To this end, we introduce the judgment $\Gamma \vdash d \to \Gamma'$ meaning "in environment Γ, declaration d is well-formed and outputs environment Γ'". It is defined as follows.

$$\frac{\forall i,\ \Gamma, \texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}} \vdash \tau_i\ \mathsf{bf}}{\Gamma \vdash \texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1\texttt{;} \cdots \tau_n\ x_n\texttt{;\}} \to \{\texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}}\} \cup \Gamma}$$

Note that types τ_i may only refer to the structure *id* via pointer types (including the case where structure definition is recursive).

3.2 Type-Checking Expressions

We introduce the typing judgment $\Gamma \vdash e : \tau$ meaning "in environment Γ, expression e is well-typed, with type τ". This judgment is defined as follows:

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{void*}} \qquad \frac{c \text{ integer constant}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{struct}\ S\ * \quad \texttt{struct}\ S\ \texttt{\{}\tau\ x\texttt{\}} \in \Gamma}{\Gamma \vdash e\texttt{->}x : \tau} \qquad \frac{\texttt{struct}\ S \in \Gamma}{\Gamma \vdash \texttt{sizeof(struct}\ S\texttt{)} : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 \texttt{ = } e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \texttt{int}}{\Gamma \vdash \texttt{-}\ e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{!}\ e : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad op \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad op \in \{\texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \texttt{int} \quad \tau_2 \equiv \texttt{int} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\tau\ f(\tau_1', \dots, \tau_n') \in \Gamma \quad \forall i,\ \Gamma \vdash e_i : \tau_i \quad \tau_i \equiv \tau_i'}{\Gamma \vdash f(e_1, \dots, e_n) : \tau}$$

3.3 Type-Checking Statements

We introduce the judgment $\Gamma \vdash^{\tau_0} s$ meaning "in environment Γ, statement s is well-typed, for a return type τ_0". Type τ_0 stands for the return type of the function in which statement s occurs. This judgment is defined as follows:

$$\frac{}{\Gamma \vdash^{\tau_0} \texttt{;}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash^{\tau_0} e\texttt{;}} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau_0}{\Gamma \vdash^{\tau_0} \texttt{return}\ e\texttt{;}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} s_1 \quad \Gamma \vdash^{\tau_0} s_2}{\Gamma \vdash^{\tau_0} \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} s}{\Gamma \vdash^{\tau_0} \texttt{while}(e)\ s}$$

$$\frac{\forall j,\ \Gamma \vdash \tau_j\ \mathsf{bf} \quad \forall j,\ \Gamma + \{\tau_1\ x_1, \dots, \tau_k\ x_k\} \vdash^{\tau_0} s_j}{\Gamma \vdash^{\tau_0} \texttt{\{}\tau_1\ x_1 \cdots \tau_k\ x_k\texttt{;}s_1 \cdots s_n\texttt{\}}}$$

The last rule means that, to type a block with k local variables and n statements, we first check that the variable declarations are well-formed and then we type-check each statement in the environment that is augmented with the new declarations.

3.4 Type-Checking Function Declarations and Files

Finally, we explain how to type check functions declarations and files.

Function Declarations.

$$\frac{\forall i,\ \Gamma \vdash \tau_i\ \mathsf{bf} \quad \{\tau_0\ f(\tau_1, \dots, \tau_n), \tau_1\ x_1, \dots, \tau_n\ x_n\} \cup \Gamma \vdash^{\tau_0} b}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \dots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \dots, \tau_n)\} \cup \Gamma}$$

Note that the prototype of function f is added to the environment before we type-check its body b, so that recursive functions are allowed.

Files. Finally, we introduce the judgment $\Gamma \vdash_f d_1 \cdots d_n$ meaning "in environment Γ, the file made of declarations d_1, …, d_n is well-formed". Type-checking a file consists in type-checking its declarations in sequence, the environment being augmented with each new declaration.

$$\frac{}{\Gamma \vdash_f \emptyset} \qquad \frac{\Gamma \vdash d_1 \to \Gamma' \quad \Gamma' \vdash_f d_2 \cdots d_n}{\Gamma \vdash_f d_1\ d_2 \cdots d_n}$$

Entry Point. Finally, we have to check for the existence of a `main` function with type

```
int main();
```