Principia Softwarica: The Plan 9 C Compiler `5c`

ARM (32 bits) edition

OCaml edition

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson and Yoann Padioleau

March 10, 2026

Contents

1 Introduction **11**
- 1.1 Motivations . . . 11
- 1.2 The ARM Plan 9 C compiler: `5c` . . . 11
- 1.3 Other compilers . . . 11
- 1.4 Getting started . . . 12
- 1.5 Requirements . . . 12
- 1.6 About this document . . . 12
- 1.7 Copyright . . . 12
- 1.8 Acknowledgments . . . 12

2 Overview **13**
- 2.1 Compiler principles . . . 14
  - 2.1.1 Frontend . . . 14
  - 2.1.2 Scope resolution . . . 14
  - 2.1.3 Use/def checking . . . 14
  - 2.1.4 Type checking . . . 14
  - 2.1.5 Backend . . . 14
  - 2.1.6 Labelization . . . 14
  - 2.1.7 Linearization . . . 14
  - 2.1.8 Frameization . . . 14
  - 2.1.9 Optimizations . . . 14
- 2.2 `5c` command-line interface . . . 14
- 2.3 `helloworld.c` . . . 14
- 2.4 Input C language . . . 14
  - 2.4.1 C innovations . . . 14
- 2.5 Output object format . . . 14
- 2.6 The ARM architecture . . . 14
- 2.7 Code organization . . . 14
- 2.8 Software architecture . . . 14
- 2.9 Book structure . . . 14

3 Core Data Structures **15**
- 3.1 Tokens . . . 15
- 3.2 Abstract syntax tree . . . 16
  - 3.2.1 Location . . . 16
  - 3.2.2 Names and namespaces . . . 16
  - 3.2.3 Types . . . 17
  - 3.2.4 Expressions . . . 17
  - 3.2.5 Statements . . . 19
  - 3.2.6 Declarations and definitions . . . 20

3.3 Type system . . . 22
3.4 Storage class . . . 23
3.5 Frontend result: typed program . . . 23
3.6 Backend result: assembly instructions . . . 24

4 Main functions **25**
4.1 `main()` . . . 25
4.1.1 `main()` skeleton . . . 25
4.1.2 Main code path . . . 26
4.1.3 Arguments processing . . . 27
4.1.4 Error management . . . 27
4.2 `compile()` . . . 28
4.3 `frontend()` . . . 28
4.4 `backend()` . . . 28

5 Lexing **29**
5.1 Overview . . . 29
5.2 Comments and spaces . . . 31
5.3 Keywords and identifiers . . . 31
5.3.1 Unicode identifiers . . . 32
5.3.2 Typedef trick, part 1 . . . 32
5.4 Operators . . . 32
5.5 Punctuations . . . 33
5.6 Numbers . . . 33
5.6.1 Decimals . . . 33
5.6.2 Octals and hexadecimals . . . 34
5.6.3 Unsigned and long numbers . . . 34
5.6.4 Floats . . . 34
5.7 Characters . . . 35
5.7.1 Escaping characters . . . 35
5.7.2 Unicode characters . . . 36
5.8 Strings . . . 36
5.8.1 Unicode strings . . . 36

6 Parsing **37**
6.1 Overview . . . 37
6.2 Parsing environment and scope management . . . 38
6.3 Lifting nested definitions to the toplevel . . . 40
6.4 Naming anonymous definitions . . . 40
6.5 Grammar overview . . . 40
6.6 Declarations and definitions, part one . . . 42
6.6.1 No-declarator declarations . . . 43
6.6.2 Declarator-based declarations . . . 43
6.6.3 Entity declarations, the declarator . . . 43
6.6.4 Function definitions . . . 44
6.7 Statements . . . 45
6.7.1 Blocks . . . 45
6.7.2 Conditionals . . . 46
6.7.3 Loops . . . 46
6.7.4 Control flow jumps . . . 47

6.7.5 Labels and goto . . . 47
6.7.6 Switch . . . 47
6.8 Expressions . . . 47
6.8.1 Numeric constants . . . 48
6.8.2 String constants . . . 48
6.8.3 Entity uses . . . 48
6.8.4 Arithmetic expressions . . . 49
6.8.5 Boolean expressions . . . 49
6.8.6 Assignments . . . 49
6.8.7 Pointers . . . 49
6.8.8 Array accesses . . . 50
6.8.9 Field accesses . . . 50
6.8.10 Function calls . . . 50
6.8.11 Cast . . . 50
6.8.12 Ternary expressions . . . 50
6.8.13 Prefix/postfix . . . 50
6.8.14 `sizeof()` . . . 50
6.9 Initializers and designators . . . 51
6.10 Types and storage classes . . . 51
6.10.1 Basic types . . . 52
6.10.2 Storage classes . . . 52
6.10.3 Qualifiers . . . 53
6.10.4 Structures and unions . . . 53
6.10.5 Enums . . . 54
6.10.6 Pointer and array types . . . 55
6.10.7 Function types, parameter types . . . 55
6.10.8 Typedefs . . . 56
6.11 Declarations and definitions, part two . . . 56
6.11.1 Globals, external declarator . . . 56
6.11.2 Function parameters . . . 56
6.11.3 Locals, automatic declarator . . . 56
6.11.4 Types, abstract declarator . . . 57

7 Checking **58**
7.1 Overview . . . 58
7.2 Printing warnings: `5c -w` and `5c -W` . . . 58
7.3 Checking environment . . . 59
7.4 Checking a program . . . 59
7.4.1 Checking toplevel definitions . . . 60
7.4.2 Checking statements . . . 61
7.4.3 Checking expressions . . . 63
7.4.4 Checking types . . . 64
7.5 Checks . . . 65
7.5.1 Inconsistent or redefined tag . . . 65
7.5.2 Inconsistent or redefined identifier . . . 66
7.5.3 Unused variables . . . 66
7.5.4 Used before set . . . 67
7.5.5 Used but not defined tags . . . 67
7.5.6 False positives silencing: `SET()/USED()` . . . 67
7.5.7 Use of undeclared labels or unused labels . . . 67

7.5.8 Unreachable code . . . 68
7.5.9 Missing return . . . 68
7.5.10 Unused expression . . . 68
7.5.11 Constant if: `5c -c` . . . 68
7.5.12 Out of range shifting . . . 68
7.5.13 Useless comparisons . . . 68

8 Typechecking **69**
8.1 Overview . . . 69
8.2 Typechecking environment . . . 70
8.3 Typechecking helpers . . . 71
8.3.1 Type equality . . . 71
8.3.2 Type compatibility . . . 71
8.3.3 Type merge . . . 71
8.3.4 generic `void*` pointer conversions: `5c -V` . . . 71
8.3.5 Compatibility policies . . . 71
8.3.6 Array to pointer conversions . . . 71
8.4 Expressions . . . 72
8.4.1 Numeric constants . . . 72
8.4.2 String constants . . . 72
8.4.3 Entity uses . . . 73
8.4.4 Arithmetic and boolean expressions . . . 73
8.4.5 Arithmetic conversions . . . 75
8.4.6 Pointer arithmetic . . . 76
8.4.7 Assignments . . . 76
8.4.8 Pointers . . . 77
8.4.9 Array accesses . . . 78
8.4.10 Field accesses . . . 78
8.4.11 Function calls . . . 80
8.4.12 Cast . . . 81
8.4.13 Ternary expressions . . . 81
8.4.14 Prefix/postfix . . . 81
8.4.15 `sizeof()` . . . 82
8.5 Statements . . . 82
8.5.1 `if` . . . 83
8.5.2 `switch` . . . 84
8.5.3 `return` . . . 84
8.6 Declarations . . . 84
8.6.1 Structures and unions . . . 84
8.6.2 Enumerations . . . 85
8.6.3 Typedef expansions . . . 85
8.6.4 Variables . . . 86
8.6.5 Functions . . . 88
8.7 Const checking and evaluation . . . 91

9 Assembly Generation **93**
9.1 Overview . . . 93
9.2 Arch-specific settings . . . 93
9.3 Code generation environment . . . 94
9.4 `codegen()` skeleton . . . 95

9.5 Functions . . . 96
9.6 Register allocation . . . 98
9.7 Operand part 1 . . . 99
9.8 Statements . . . 99
9.8.1 Local variables . . . 99
9.8.2 Blocks, sequences . . . 100
9.8.3 Conditionals . . . 100
9.8.4 Switch . . . 101
9.8.5 Labels and goto . . . 101
9.8.6 Loops . . . 102
9.8.7 Control flow jumps . . . 103
9.9 Basic expressions . . . 104
9.9.1 Operand part 2 . . . 104
9.9.2 Numeric constants . . . 105
9.9.3 Entity uses . . . 105
9.9.4 Pointers part1 . . . 105
9.9.5 `gmove()` . . . 106
9.10 Complex expressions . . . 107
9.10.1 Complexity . . . 107
9.10.2 String constants . . . 108
9.10.3 Arithmetic expressions . . . 108
9.10.4 Boolean expressions . . . 109
9.10.5 Assignments part 1 . . . 109
9.10.6 Pointers part2 . . . 110
9.10.7 Assignments part 2 . . . 112
9.10.8 Assignments and arithmetic . . . 112
9.10.9 Function calls . . . 112
9.10.10 Array accesses . . . 112
9.10.11 Field accesses . . . 112
9.10.12 Cast . . . 112
9.10.13 Ternary expressions . . . 112
9.10.14 Prefix/postfix . . . 112
9.10.15 `sizeof()` . . . 112
9.11 Boolean expressions . . . 112
9.12 Other topics . . . 112
9.12.1 Sizes . . . 112
9.12.2 First argument . . . 112
9.12.3 Alignment . . . 112
9.12.4 Toplevel globals . . . 112
9.12.5 Initializers . . . 112
9.13 Advanced assembly generation . . . 112
9.13.1 Function calls . . . 112
9.13.2 Switch . . . 112

10 Object File Generation **113**
10.1 Object format . . . 113
10.2 Instruction output . . . 113

11 AST-level Optimizations **114**
11.1 AST simplifications . . . 114
11.1.1 General rewrites . . . 114
11.1.2 Constant evaluation . . . 114
11.2 Comma hoisting . . . 114
11.3 Bitshifting opportunities . . . 114
11.4 And opportunities . . . 114
11.5 Immediate operands . . . 114
11.5.1 Substraction . . . 114
11.5.2 Addition, or, etc. . . . 114
11.5.3 Multiplication . . . 114
11.6 Associative-commutative arithmetic optimisations . . . 114

12 Assembly-level Optimisations **115**
12.1 Nop detection . . . 115
12.2 ARM special instructions . . . 115
12.3 Register passing argument: `REGARG` . . . 115
12.4 Register allocation optimisations . . . 115
12.5 Peephole optimizer . . . 115
12.6 Dominators . . . 115

13 Linking Support **116**
13.1 `#pragma lib` and automagic linking . . . 116
13.2 Safe linking with type signatures: `5c -T` . . . 116
13.2.1 `sign()` . . . 116
13.2.2 `#pragma incomplete` . . . 116

14 Debugging Support **117**
14.1 General debugging metadata . . . 117
14.2 Acid debugger metadata: `5c -a` . . . 117
14.2.1 Entities . . . 117
14.2.2 Structure/union definitions . . . 117
14.3 Pickle: `5c -Z` . . . 117

15 Profiling Support **118**
15.1 Text attributes . . . 118
15.2 `#pragma profile` . . . 118

16 Preprocessing **119**
16.1 Overview . . . 119
16.2 Core data structures . . . 119
16.2.1 Preprocessor configuration . . . 119
16.2.2 Directives AST . . . 120
16.2.3 Location and line history . . . 120
16.2.4 Parser hook . . . 121
16.3 Lexing . . . 122
16.4 Parsing . . . 130
16.5 `#include` . . . 134
16.5.1 Include paths, `-I` . . . 134
16.5.2 Tracing origin . . . 134

16.5.3 `#include` . . . 134
16.6 `#define` . . . 134
16.6.1 `-D` . . . 134
16.6.2 `#define` . . . 135
16.6.3 Macro Expansion . . . 135
16.7 `#undef` . . . 136
16.8 `#ifdef` . . . 136
16.9 `#pragma` . . . 136
16.10 `#line` . . . 136

17 Advanced Topics TODO **137**
17.1 Advanced C features . . . 138
17.1.1 Function and function pointer automatic conversions . . . 138
17.1.2 Array and pointer automatic conversions . . . 138
17.1.3 Local `static` . . . 138
17.1.4 Local `volatile` . . . 138
17.1.5 Bitfields . . . 138
17.1.6 Old style prototypes . . . 138
17.1.7 `_Noreturn()` . . . 138
17.2 C extensions . . . 138
17.2.1 Typing extensions . . . 138
17.2.2 Unnamed structure elements . . . 138
17.2.3 Struct constructors . . . 138
17.2.4 Per-processor storage: `extern register` . . . 138
17.2.5 Type reflection: `typestr()` . . . 138
17.2.6 Signature reflection: `signof()` . . . 138
17.2.7 Format's arguments checking . . . 138
17.3 Floats . . . 138
17.4 Big values . . . 138
17.4.1 Complex types . . . 138
17.4.2 Complex return value . . . 138
17.4.3 Complex argument . . . 138
17.4.4 Complex expression . . . 138
17.4.5 `vlong` constant . . . 138
17.5 64 bits operations . . . 138
17.6 Endianness . . . 138
17.7 Other optimizations . . . 138
17.7.1 Packing: `#pragma pack` . . . 138

18 Conclusion **139**

A Debugging **140**
A.1 Dumpers . . . 140
A.1.1 AST dumper: `5c -x` . . . 140
A.2 `5c -v`, verbose mode . . . 141
A.3 Preprocessing debugging . . . 142
A.3.1 Macro debugging: `5c -m` . . . 142
A.3.2 File inclusion debugging: `5c -e` . . . 142
A.3.3 Line information debugging: `5c -f` . . . 142
A.4 Parsing debugging . . . 142

A.4.1 Printing names: `5c -L` . . . 142
A.4.2 Printing declarations: `5c -d` . . . 142
A.5 Typing debugging . . . 142
A.5.1 Printing expression type trees: `5c -t` . . . 143
A.6 Code generation debugging . . . 143
A.6.1 Printing processed opcodes: `5c -G` . . . 144
A.6.2 Printing code generation information: `5c -g` . . . 144
A.6.3 Printing generated assembly: `5c -S` . . . 144
A.7 Optimization debugging . . . 144
A.7.1 Printing arithmetic trees: `5c -m` . . . 144
A.7.2 Printing registerization: `5c -r` . . . 144
A.8 Printing initializations: `5c -i` . . . 144

B Error Management **145**
B.1 `errorexit()` . . . 145
B.2 Lexing error . . . 145
B.3 Parsing error . . . 145
B.4 Typechecking error . . . 145
B.5 Checking error . . . 145
B.6 Code generation error . . . 146
B.7 Backtraces . . . 146
B.8 Error location . . . 146

C Extra Code **147**
C.1 `Arch_compiler.ml` . . . 147
C.2 `Arch5.mli` . . . 147
C.3 `Arch5.ml` . . . 147
C.4 `Ast.ml` . . . 149
C.5 `Check.mli` . . . 151
C.6 `Check.ml` . . . 151
C.7 `CLI.mli` . . . 153
C.8 `CLI.ml` . . . 153
C.9 `Codegen.mli` . . . 154
C.10 `Codegen.ml` . . . 154
C.11 `Dumper_.mli` . . . 158
C.12 `Dumper_.ml` . . . 158
C.13 `Error.mli` . . . 158
C.14 `Error.ml` . . . 158
C.15 `Eval_const.mli` . . . 158
C.16 `Eval_const.ml` . . . 159
C.17 `Flags.ml` . . . 159
C.18 `Globals.ml` . . . 160
C.19 `Lexer.mll` . . . 160
C.20 `Main.ml` . . . 160
C.21 `Parse.mli` . . . 160
C.22 `Parse.ml` . . . 160
C.23 `Parser.mly` . . . 161
C.24 `Rewrite.mli` . . . 161
C.25 `Rewrite.ml` . . . 161
C.26 `Storage.ml` . . . 162

C.27 Type.ml . . . 162
C.28 Typecheck.mli . . . 162
C.29 Typecheck.ml . . . 163
C.30 macroprocessor/Ast_cpp.ml . . . 165
C.31 macroprocessor/Flags_cpp.ml . . . 165
C.32 macroprocessor/Lexer_cpp.mll . . . 166
C.33 macroprocessor/Location_cpp.mli . . . 166
C.34 macroprocessor/Location_cpp.ml . . . 166
C.35 macroprocessor/Parse_cpp.mli . . . 167
C.36 macroprocessor/Parse_cpp.ml . . . 167
C.37 macroprocessor/Preprocessor.ml . . . 168

Glossary **170**

Indexes **171**

References **171**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a C compiler.

1.1 Motivations

Why a C compiler? Because I think you are a better programmer if you fully understand how things work under the hood, and understanding how a compiler translates high-level C code in low-level assembly, and ultimately machine code, is essential.

Here are a few questions I hope this book will answer:

- How does a compiler parse a complex programming language?
- How does a compiler manage the scope of variables, structures, enumerations, or typedefs?
- How does a typechecker work?
- How does a compiler translate conditionals, loops, and other control-flow constructs in basic assembly jumps?
- How does a compiler translate a switch?
- How does a compiler convert arbitrary complex expressions into simpler assembly instructions using a small set of registers?
- How does a function pass arguments to another function? Are registers used? Is the stack used? How does a function communicate its return value to the caller?

1.2 The ARM Plan 9 C compiler: `5c`

1.3 Other compilers

Here are a few compilers that I considered for this book, but which I ultimately discarded:

- gcc
- clang
- lcc
- compcert

- bcc
- fbcc and tcc

1.4 Getting started

1.5 Requirements

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `5c`, Ken Thompson, who wrote in some sense most of this book.

Chapter 2

Overview

2.1 Compiler principles

2.1.1 Frontend

2.1.2 Scope resolution

2.1.3 Use/def checking

2.1.4 Type checking

2.1.5 Backend

2.1.6 Labelization

2.1.7 Linearization

2.1.8 Frameization

2.1.9 Optimizations

2.2 `5c` command-line interface

2.3 `helloworld.c`

2.4 Input C language

2.4.1 C innovations

2.5 Output object format

2.6 The ARM architecture

2.7 Code organization

2.8 Software architecture

2.9 Book structure

Chapter 3

Core Data Structures

3.1 Tokens

⟨*Parser tokens* 15a⟩≡ (37d) 15b▷

```
/*(*----------------------------------------*)*/
/*(*2 Constants and identifiers *)*/
/*(*----------------------------------------*)*/
%token <Ast.loc * string * Type.integer_type> TIConst
%token <Ast.loc * string * Type.float_type> TFConst
%token <Ast.loc * string * Type.t> TString
%token <Ast.loc * string> TName TTypeName
```

⟨*Parser tokens* 15b⟩+≡ (37d) ◁15a 15c▷

```
/*(*----------------------------------------*)*/
/*(*2 Keywords *)*/
/*(*----------------------------------------*)*/
%token <Ast.loc> Tvoid  Tchar Tshort Tint Tlong  Tdouble Tfloat
%token <Ast.loc> Tsigned Tunsigned
%token <Ast.loc> Tstruct Tunion Tenum
%token <Ast.loc> Ttypedef
%token <Ast.loc> Tconst Tvolatile  Trestrict Tinline
%token <Ast.loc> Tauto Tstatic Textern Tregister
%token <Ast.loc> Tif Telse  Twhile Tdo  Tfor  Tbreak Tcontinue  Treturn Tgoto
%token <Ast.loc> Tswitch Tcase Tdefault
%token <Ast.loc> Tsizeof
```

⟨*Parser tokens* 15c⟩+≡ (37d) ◁15b 15d▷

```
/*(*----------------------------------------*)*/
/*(*2 Operators *)*/
/*(*----------------------------------------*)*/
%token <Ast.loc> TPlus TMinus  TMul TDiv TMod
%token <Ast.loc> TEq TEqEq TBang TBangEq
%token <Ast.loc> TAnd TOr TXor TAndAnd TOrOr
%token <Ast.loc> TTilde
%token <Ast.loc> TPlusPlus TMinusMinus
%token <Ast.loc> TInf TSup  TInfEq TSupEq
%token <Ast.loc> TInfInf TSupSup
%token <Ast.loc * Ast.arithOp> TOpEq
```

⟨*Parser tokens* 15d⟩+≡ (37d) ◁15c 16a▷

```
/*(*----------------------------------------*)*/
/*(*2 Punctuation *)*/
/*(*----------------------------------------*)*/
%token <Ast.loc> TOPar TCPar  TOBrace TCBrace TOBra TCBra
%token <Ast.loc> TComma TSemicolon
%token <Ast.loc> TArrow TDot TQuestion TColon
```

⟨*Parser tokens* 16a⟩+≡ (37d) ◁15d
```
/*(*----------------------------------------*)*/
/*(*2 Misc *)*/
/*(*----------------------------------------*)*/
%token TSharp
%token EOF
```

3.2 Abstract syntax tree

3.2.1 Location

⟨*type* Ast.loc 16b⟩≡ (149)
```
(* global linenumber after preprocessing *)
type loc = Location_cpp.loc
```

3.2.2 Names and namespaces

⟨*type* Ast.name 16c⟩≡ (149)
```
type name = string
```

⟨*type* Ast.blockid 16d⟩≡ (149)
```
(* for scope *)
type blockid = int (* same than Type_.blockid, repeated here for clarity *)
```

⟨*type* Ast.fullname 16e⟩≡ (149)
```
(* A fully resolved and scoped name.
 *
 * 5c: uses a reference to a symbol in a symbol table to fully qualify a name.
 * Instead, I use a unique blockid and an external hash or environment that
 * maps this fullname to the appropriate information.
 * I think it offers a better separation of concerns.
 *
 * 'name' below can be a gensym'ed name for anonymous struct/union/enum.
 *)
type fullname = name * blockid (* same than Type_.fullname *)
```

⟨*type* Ast.idkind 16f⟩≡ (149)
```
(* Used in Globals.ml/Lexer.mll/Parser.mly to recognize typedef identifiers.
 * alt: could be moved in a separate Naming.ml, but not worth it for just two types.
 *)
type idkind =
  | IdIdent
  | IdTypedef
  | IdEnumConstant
```

⟨*global* Globals.hids 16g⟩≡ (160a)
```
(* to recognize typedefs in the lexer *)
let hids: (string, Ast.idkind) Hashtbl.t =
  Hashtbl.create 101
```

⟨*type* Ast.tagkind 16h⟩≡ (149)
```
(* to manage the scope of tags *)
type tagkind =
  | TagStruct
  | TagUnion
  | TagEnum
```

3.2.3 Types

⟨*type* Ast.typ 17a⟩≡ (149)
```
(* What are the differences between typ below and Type.t?
 * - typedef expansion is not done here
 * - constant expressions are not resolved yet
 *  (those expressions can involve enum constants which will be resolved later).
 * Again, I think it offers a better separation of concerns.
 *
 * Note that 'type_' and 'expr' are mutually recursive (because of const_expr).
 * todo: qualifier type
 *)
type typ = {
  t: type_bis;
  t_loc: loc;
}
```

⟨*type* Ast.type_bis 17b⟩≡ (149)
```
  and type_bis =
  | TBase of Type.t (* only the basic stuff *)
  | TPointer of typ
  | TArray of const_expr option * typ
  | TFunction of function_type

  (* no StructDef here; they are lifted up; just StructName *)
  | TStructName of Type.struct_kind * fullname
  (* In C an enum is really like an int. However, we could do
   * extended checks at some point to do more strict type checking!
   *)
  | TEnumName of fullname
  | TTypeName of fullname
```

⟨*type* Type.struct_kind 17c⟩≡ (162b)
```
 and struct_kind = Struct | Union
```

⟨*function* Ast.tagkind_of_su 17d⟩≡ (149)
```
let tagkind_of_su = function
  | Type.Struct -> TagStruct
  | Type.Union -> TagUnion
```

3.2.4 Expressions

⟨*type* Ast.expr 17e⟩≡ (149)
```
and expr = {
  e: expr_bis;
  e_loc: loc;
  ⟨Ast.expr other fields 22c⟩
}
```

⟨*type* Ast.expr_bis 17f⟩≡ (149)
```
  and expr_bis =
  (* Note that characters are transformed in Int at parsing time; no need Char *)
  | Int of string * Type.integer_type
  | Float of string * Type.float_type
  (* codegen: converted to Id after typechecking *)
  | String of string * Type.t (* always array of chars for now, no unicode *)

  (* Global, local, parameter, enum constant (can be scoped), function.
   * Not that the storage, type, usage of ids is computed later and stored
```

```
     * in external hashtbl.
     * codegen: Id converted to Int when the fullname refers to a enum constant
     *)
    | Id of fullname

    | Call of expr * argument list

    (* should be a statement really *)
    | Assign of assignOp * expr * expr

    (* codegen: converted to pointer arithmetic, *(x+y) *)
    | ArrayAccess of expr * expr (* x[y] *)
    (* codegen: converted to pointer offset access *)
    | RecordAccess of expr * name (* x.y *)
    (* codegen: converted to RecordAccess, ( *x ).y *)
    | RecordPtAccess of expr * name (* x->y,  and not x.y!! *)

    (* less: bool (* explicit cast (xcast) *) *)
    | Cast of typ * expr

    | Postfix of expr * fixOp
    | Prefix of fixOp * expr
    (* contains GetRef and Deref!! pointers!  *)
    | Unary of unaryOp * expr
    | Binary of expr * binaryOp * expr

    | CondExpr of expr * expr * expr
    (* 'x, y', but really should be a statement, and could be removed.
     * I think mostly used in 'for(...;...;...)'
     *)
    | Sequence of expr * expr

    (* codegen: converted to Int *)
    | SizeOf of (expr, typ) Either_.t

    ⟨Ast.expr initialiser cases 20g⟩
    ⟨Ast.expr extension cases 20h⟩

⟨type Ast.argument 18a⟩≡                                              (149)
  and argument = expr

⟨type Ast.const_expr 18b⟩≡                                            (149)
  (* Because we call the preprocessor first, the remaining cases
   * where const_expr is not a constant are basic arithmetic expressions
   * like 2 < < 3, or enum constants.
   *)
  and const_expr = expr

⟨type Ast.unaryOp 18c⟩≡                                               (149)
    and unaryOp  =
      (* less: could be lifted up; those are really important operators *)
      | GetRef | DeRef
      (* codegen: converted to binary operation with 0 (-x => 0-x) *)
      | UnPlus |  UnMinus
      (* codegen: converted to -1 ^ x *)
      | Tilde
      | Not

⟨type Ast.assignOp 18d⟩≡                                              (149)
    and assignOp = Eq_ | OpAssign of arithOp
```

⟨*type* Ast.fixOp 19a⟩≡ (149)
```
    and fixOp    = Dec | Inc
```

⟨*type* Ast.binaryOp 19b⟩≡ (149)
```
    and binaryOp = Arith of arithOp | Logical of logicalOp
```

⟨*type* Ast.arithOp 19c⟩≡ (149)
```
        and arithOp   =
          | Plus | Minus
          | Mul | Div | Mod
          | ShiftLeft | ShiftRight
          | And | Or | Xor
```

⟨*type* Ast.logicalOp 19d⟩≡ (149)
```
        and logicalOp =
          | Inf | Sup | InfEq | SupEq
          | Eq | NotEq
          | AndLog | OrLog
```

3.2.5 Statements

⟨*type* Ast.stmt 19e⟩≡ (149)
```
  type stmt = {
    s: stmt_bis;
    s_loc: loc;
  }
```

⟨*type* Ast.stmt_bis 19f⟩≡ (149)
```
    and stmt_bis =
    | ExprSt of expr
    (* empty statement is simply Block [] *)
    | Block of stmt list

    (* else is optional and represented as an empty Block *)
    | If of expr * stmt * stmt

    (* expr must have an integer type; it can not be a pointer like in a If *)
    | Switch of expr * case_list

    | While of expr * stmt
    | DoWhile of stmt * expr
    | For of (expr option, var_decl list) Either_.t *
             expr option *
             expr option *
             stmt

    | Return of expr option
    (* no argument to continue or break as in PHP *)
    | Continue | Break

    (* labels have a function scope, so no need to use 'fullname' here *)
    | Label of name * stmt
    | Goto of name

    (* should occur only in Switch *)
    | Case of expr * stmt
    | Default of stmt

    | Var of var_decl
```

⟨*type* Ast.case_list 20a⟩≡ (149)
```
(* Can we have a specific case type? It is hard in C because they mix labels
 * and 'case' a lot (see the code in the lexer of 5c).
 *)
and case_list = stmt
```

3.2.6 Declarations and definitions

⟨*type* Ast.toplevel 20b⟩≡ (149)
```
type toplevel =
  | StructDef of struct_def
  | TypeDef of type_def
  | EnumDef of enum_def
  (* globals, but also extern decls and prototypes *)
  | VarDecl of var_decl
  | FuncDef of func_def
```

⟨*type* Ast.toplevels 20c⟩≡ (149)
```
type toplevels = toplevel list
```

⟨*type* Ast.program 20d⟩≡ (149)
```
type program = toplevels * Location_cpp.location_history list
```

Variables

⟨*type* Ast.var_decl 20e⟩≡ (149)
```
and var_decl = {
  v_name: fullname;
  v_loc: loc;
  v_storage: Storage.t option;
  v_type: typ;
  v_init: initialiser option;
}
```

⟨*type* Ast.initialiser 20f⟩≡ (149)
```
 (* can have ArrayInit and RecordInit here in addition to other expr *)
 and initialiser = expr
```

⟨Ast.expr *initialiser cases* 20g⟩≡ (17f)
```
(* should appear only in a variable initializer, or after GccConstructor *)
| ArrayInit of (const_expr option * expr) list
| RecordInit of (name * expr) list
```

⟨Ast.expr *extension cases* 20h⟩≡ (17f)
```
(* gccext: kenccext: *)
| GccConstructor  of typ * expr (* always an ArrayInit (or RecordInit?) *)
```

Functions

⟨*type* Ast.func_def 20i⟩≡ (149)
```
type func_def = {
  (* functions have a global scope; no need for fullname here *)
  f_name: name;
  f_loc: loc;
  (* everything except Param or Auto *)
  f_storage: Storage.t option;
  f_type: function_type;
  (* always a Block *)
  f_body: stmt;
}
```

⟨*type* Ast.function_type 21a⟩≡ (149)
```
  and function_type = (typ * (parameter list * bool (* var args '...' *)))
```

⟨*type* Ast.parameter 21b⟩≡ (149)
```
    and parameter = {
      (* When part of a prototype, the name is not always mentionned, hence
       * the option below.
       *
       * I use 'fullname' here for consistency; parameters are treated like,
       * locals, so we can have below simply 'Id of fullname' and have
       * no differences between accessing a local or a parameter.
       *)
      p_name: fullname option;
      p_loc: loc;

      p_type: typ;
    }
```

Structures and unions

⟨*type* Ast.struct_def 21c⟩≡ (149)
```
  (* struct and union *)
  type struct_def = {
    su_name: fullname;
    su_loc: loc;
    su_kind: Type.struct_kind;
    (* todo: bitfield annotation *)
    su_flds: field_def list;
  }
```

⟨*type* Ast.field_def 21d⟩≡ (149)
```
    (* Not the same than var_decl; fields have no storage and can have bitflds.*)
    and field_def = {
     (* kenccext: anonymous structure element get an artificial field name
      * (see is_gensymed())
      *)
      fld_name: name;
      fld_loc: loc;
      fld_type: typ;
    }
```

Enumerations

⟨*type* Ast.enum_def 21e⟩≡ (149)
```
  type enum_def = {
    (* this name is rarely used; C programmers rarely write 'enum Foo x;' *)
    enum_name: fullname;
    enum_loc: loc;
    enum_constants: enum_constant list;
  }
```

⟨*type* Ast.enum_constant 21f⟩≡ (149)
```
    and enum_constant = {
    (* we also need to use 'fullname' for constants, to scope them *)
      ecst_name: fullname;
      ecst_loc: loc;
      ecst_value: const_expr option;
    }
```

Type aliases

⟨*type* `Ast.type_def` 22a⟩≡ (149)
```
type type_def = {
  typedef_name: fullname;
  typedef_loc: loc;
  typedef_type: typ;
}
[@@deriving show { with_path = false } ]
```

3.3 Type system

⟨*type* `Type.t` 22b⟩≡ (162b)
```
(* The C type system.
 *
 * There is no Typedef below. The typechecker expands typedefs.
 * There is no Enum either because variables using enum
 * (as in 'enum Foo x;') gets their type expanded to an integer type.
 *
 * less: put qualifier here?
 * todoext: Bool! with strict bool checking.
 * todoext: Enum of fullname with stricter checking.
 *)
type t =
  (* basic types *)
  | Void
  | I of integer_type
  | F of float_type

  (* composite types *)
  | Pointer of t
  (* Why not unsugar Array to Pointer? Because the type system checks
   * for some array incompatibilities. int[2] != int[3].
   * However, the Array type usually gets converted to Pointer
   * during typechecking (see array_to_pointer())
   *)
  | Array of int option * t
  | Func of t * t list * bool (* varargs '...' *)
  | StructName of struct_kind * fullname
```

⟨`Ast.expr` *other fields* 22c⟩≡ (17e)
```
(* properly set during typechecking in typecheck.ml *)
e_type: Type.t;
```

⟨*type* `Type.integer_type` 22d⟩≡ (162b)
```
  and integer_type = integer_kind * sign
```

⟨*type* `Type.integer_kind` 22e⟩≡ (162b)
```
    and integer_kind =
    | Char
    | Short
    | Int
    | Long
    | VLong
```

⟨*type* `Type.sign` 22f⟩≡ (162b)
```
    and sign = Signed | Unsigned
```

⟨*type* Type.float_type 23a⟩≡ (162b)
```
and float_type =
| Float
| Double
```

⟨*type* Type.qualifier 23b⟩≡ (162b)
```
type qualifier =
  | Volatile
  | Const
  (* less: unsupported: | Restrict | Inline *)
```

⟨*constant* Type.int 23c⟩≡ (162b)
```
let int = I (Int, Signed)
```

⟨*constant* Type.long 23d⟩≡ (162b)
```
let long = I (Long, Signed)
```

⟨*type* Type.blockid 23e⟩≡ (162b)
```
type blockid = int
```

⟨*type* Type.fullname 23f⟩≡ (162b)
```
type fullname = string * blockid
```

3.4 Storage class

⟨*type* Storage.t 23g⟩≡ (162a)
```
(* No Typedef here, because a typedef is not a storage! *)
type t =
  | Local  (* local (a.k.a., Auto) *)
  | Param  (* parameter *)

  | Extern (* public global defined elsewhere *)
  | Global (* public global defined here *)
  | Static (* Private global less: could rename Private *)

  (* less:  | Inline? | Register? | ExternRegister? *)
```

3.5 Frontend result: typed program

⟨*type* Typecheck.typed_program 23h⟩≡ (163b 162c)
```
type typed_program = {
  (* resolved type and storage information for identifiers and tags *)
  ids: (Ast.fullname, idinfo) Hashtbl_.t;

  (* resolved struct definitions *)
  structs: (Ast.fullname, Type.struct_kind * Type.structdef) Hashtbl_.t;

  (* functions annotated with types for each expression nodes
   * (so you can more easily generate code later).
   *
   * The enum constants should also be internally resolved and replaced
   * with constants and some constant expressions (e.g., for
   * array size) should also be resolved (and evaluated).
   *)
  funcs: Ast.func_def list;
}
```

⟨*type* `Typecheck.idinfo` 24a⟩≡ (163b 162c)

```
type idinfo = {
    typ: Type.t;
    sto: Storage.t;
    loc: Location_cpp.loc;
    (* typed initialisers (fake expression for function definitions) *)
    ini: Ast.initialiser option;
  }
```

⟨*type* `Type.structdef` 24b⟩≡ (162b)

```
(* Note that the field can be gensym'ed for anonymous struct/union elements.
 * Note also structdef is not part of Type.t above; structdef is used
 * instead in Typecheck.typed_program
 * todo: bitfield
 *)
type structdef = (string * t) list
```

3.6 Backend result: assembly instructions

Chapter 4

Main functions

4.1 main()

⟨*toplevel* Main._1 25a⟩≡ (160c)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps (Exit.catch (fun () ->
         CLI.main caps argv))
  )
```

⟨*signature* CLI.main 25b⟩≡ (153a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; ..> ->
  string array -> Exit.t
```

⟨*type* CLI.caps 25c⟩≡ (153)
```
(* Need:
 * - open_in: for argv derived file but also for #include'd files
 *   because 5c does its own macropreprocessing
 * - open_out for -o object file or 5.argv[0]
 * - env: for INCLUDE (for cpp)
 *)
type caps = < Cap.open_in; Cap.open_out; Cap.env >
```

4.1.1 main() skeleton

⟨*function* CLI.main 25d⟩≡ (153b)
```
let main (caps : <caps; ..>) (argv : string array) : Exit.t =
  ⟨CLI.main() set arch and thechar 26d⟩
  let usage = spf "usage: %s [-options] file.c" argv.(0) in

  (* in *)
  let args = ref [] in
  (* out *)
  let outfile = ref "" in

  ⟨CLI.main() macroprocessor locals 134a⟩
  ⟨CLI.main() other locals 120b⟩

  let options = [
    ⟨CLI.main() options elements 26a⟩
  ] |> Arg.align
  in
```

```
⟨CLI.main() parse argv and set args and flags 27a⟩
⟨CLI.main() logging setup 142b⟩
(* less: process the old style -Dname=val and -Idir attached *)
⟨CLI.main() action management 141d⟩

try
  ⟨CLI.main() matching args and outfile 27b⟩
with exn ->
  if Sys.file_exists !outfile
  then begin
     Logs.info (fun m -> m "removing %s because of error" !outfile);
     FS.remove caps (Fpath.v !outfile);
  end;
  ⟨CLI.main() when exn 27c⟩
```

⟨CLI.main() options *elements* 26a⟩≡ (25d) 58i▷

```
"-o", Arg.Set_string outfile,
" <file> place output (an object) in file";
```

4.1.2 Main code path

⟨CLI.main() *main code path with* cfile *and* outfile 26b⟩≡ (27b)

```
let outfile : Fpath.t =
  ⟨CLI.main() main code path, define outfile 26e⟩
in

⟨CLI.main() main code path, define macropreprocessor conf 119c⟩
outfile |> FS.with_open_out caps (fun chan ->
  compile caps conf arch (Fpath.v cfile) chan
);
```

⟨*signature* CLI.compile 26c⟩≡ (153a)

```
(* main algorithm; works by side effect on outfile *)
val compile: < Cap.open_in; .. > ->
  Preprocessor.conf -> Arch.t -> Fpath.t (* infile *) -> Chan.o (* outfile *) ->
  unit
```

⟨CLI.main() *set* arch *and* thechar 26d⟩≡ (25d)

```
let arch : Arch.t =
  match Filename.basename argv.(0) with
  | "o5c" -> Arch.Arm
  | "ovc" -> Arch.Mips
  | s -> failwith (spf "arch could not detected from argv0 %s" s)
in

let thechar = Arch.thechar arch in
let thestring = Arch.thestring arch in
```

⟨CLI.main() *main code path, define* outfile 26e⟩≡ (26b)

```
let base = Filename.basename cfile in
(if outstr = ""
then begin
  let res =
    if base =~ "\\(.*\\)\\.c"
    then Regexp_.matched1 base ^ (spf ".o%c" thechar)
    else base ^ (spf ".o%c" thechar)
  in
  outfile := res;
  res
```

```
  end
  else outstr
  ) |> Fpath.v
```

4.1.3 Arguments processing

⟨CLI.main() *parse* argv *and set* args *and flags* 27a⟩≡ (25d)
```
  (try
    Arg.parse_argv argv options (fun t ->
      args := t::!args
    ) usage;
  with
  | Arg.Bad msg -> UConsole.eprint msg; raise (Exit.ExitCode 2)
  | Arg.Help msg -> UConsole.print msg; raise (Exit.ExitCode 0)
  );
```

⟨CLI.main() *matching* args *and* outfile 27b⟩≡ (25d)
```
  (match !args, !outfile with
  | [], "" ->
      Arg.usage options usage;
      Exit.Code 1
  | [cfile], outstr ->
      ⟨CLI.main() main code path with cfile and outfile 26b⟩
      Exit.OK
  | _ ->
    (* stricter: *)
      failwith
        "compiling multiple files at the same time is not supported; use mk"
  )
```

4.1.4 Error management

⟨CLI.main() *when* exn 27c⟩≡ (25d)
```
  ⟨CLI.main() when exn if backtrace 146f⟩
  else
    ⟨CLI.main() match exn 27d⟩
```

⟨CLI.main() *match* exn 27d⟩≡ (27c)
```
  (match exn with
  | Failure s ->
      (* useful to indicate that error comes from 5c? *)
      Error.errorexit (spf "%cc: %s" thechar s)

  ⟨CLI.main() match exn other cases 27e⟩

  | _ -> raise exn
  )
```

⟨CLI.main() *match* exn *other cases* 27e⟩≡ (27d)
```
  | Location_cpp.Error (s, loc) ->
      (* less: could use final_loc_and_includers_of_loc loc *)
      let (file, line) = Location_cpp.final_loc_of_loc loc in
      Error.errorexit (spf "%s:%d %s" !!file line s)
  (* ocaml-light: | Check.Error err | Typecheck.Error err | ...  *)
  | Check.Error err -> Error.errorexit (Check.string_of_error err)
  | Typecheck.Error err -> Error.errorexit (Check.string_of_error err)
  | Eval_const.Error err -> Error.errorexit (Check.string_of_error err)
  | Codegen.Error err -> Error.errorexit (Check.string_of_error err)
```

4.2 compile()

⟨*function* CLI.compile 28a⟩≡ (153b)
```
let compile (caps : < Cap.open_in; ..>) (conf : Preprocessor.conf) (arch : Arch.t)
    (infile : Fpath.t) (outfile : Chan.o) :
    unit =
  let tast : Typecheck.typed_program = frontend caps conf infile in
  let asm_prog : 'a Ast_asm.program = backend arch tast in
  Object_file.save arch asm_prog outfile
```

⟨*function* CLI.compile5 28b⟩≡ (153b)

4.3 frontend()

⟨*function* CLI.frontend 28c⟩≡ (153b)
```
let frontend (caps : < Cap.open_in; .. >) (conf : Preprocessor.conf)
     (infile : Fpath.t) :
    Typecheck.typed_program =

  let ast = Parse.parse caps conf infile in
  ⟨CLI.frontend() if dump_ast 140e⟩

  (* use/def checking, unused entity, redefinitions, etc. *)
  Check.check_program ast;
  (* typedef expansion, type and storage resolution, etc. *)
  let tp : Typecheck.typed_program =
    Typecheck.check_and_annotate_program ast
  in
  ⟨CLI.frontend() if dump_typed_ast 143c⟩
  tp
```

4.4 backend()

⟨*function* CLI.backend5 28d⟩≡ (153b)
```
let backend (arch : Arch.t) (tast : Typecheck.typed_program) :
    'a Ast_asm.program =

  let tast = Rewrite.rewrite tast in
  let (asm, _locs) = Codegen.codegen arch tast in
    ⟨CLI.backend5() if dump_asm 143e⟩
```

⟨*signature* Codegen.codegen 28e⟩≡ (154a)
```
(* can raise Error *)
val codegen:
  Arch.t -> Typecheck.typed_program ->  'i Ast_asm.program
```

Chapter 5

Lexing

5.1 Overview

⟨Lexer.mll 29⟩≡

```
{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Regexp_.Operators

open Parser
module A = Ast
module L = Location_cpp
module T = Type

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Limitations compared to 5c:
 *  - no L"" and L'' unicode strings or unicode characters
 *  - no \x hexadecimal escape sequence in strings or characters
 *  - no unicode identifier
 *  - no typestr
 *    (seems dead extension anyway)
 *  - no signof
 *
 * todo: handle big numbers here? or let later phases do that?
 *  check for overflow, truncation, sign-extended character constant, etc
 *  see yyerror and warn in Compiler.nw
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨Lexer helpers 160b⟩
}

(*****************************************************************************)
(* Regexps aliases *)
(*****************************************************************************)
⟨constant Lexer.space 31d⟩
⟨constant Lexer.letter 31i⟩
⟨constant Lexer.digit 31j⟩
⟨constant Lexer.oct 34a⟩
⟨constant Lexer.hex 34c⟩
```

```
(******************************************************************************)
(* Main rule *)
(******************************************************************************)
⟨rule Lexer.token 30a⟩

(******************************************************************************)
(* String rule *)
(******************************************************************************)
⟨rule Lexer.string 36b⟩

(******************************************************************************)
(* Character rule *)
(******************************************************************************)
⟨rule Lexer.char 35c⟩

(******************************************************************************)
(* Comment rule *)
(******************************************************************************)
(* dup: lexer_asm5.mll *)
⟨rule Lexer.comment 31g⟩
```

⟨rule Lexer.token 30a⟩≡ (29)

```
rule token = parse
  (* ------------------------------------------------------------------------ *)
  (* Spacing/comments *)
  (* ------------------------------------------------------------------------ *)
  ⟨Lexer.token space/comment cases 31e⟩

  (* ------------------------------------------------------------------------ *)
  (* Symbols *)
  (* ------------------------------------------------------------------------ *)
  ⟨Lexer.token symbol cases 32c⟩

  (* ------------------------------------------------------------------------ *)
  (* Numbers *)
  (* ------------------------------------------------------------------------ *)
  (* dup: lexer_asm5.mll *)
  ⟨Lexer.token number cases 33b⟩

  (* ------------------------------------------------------------------------ *)
  (* Chars/Strings *)
  (* ------------------------------------------------------------------------ *)
  ⟨Lexer.token chars/strings cases 35b⟩

  (* ------------------------------------------------------------------------ *)
  (* Keywords and identifiers *)
  (* ------------------------------------------------------------------------ *)
  ⟨Lexer.token keywords/identifiers cases 32a⟩

  (* ------------------------------------------------------------------------ *)
  (* CPP *)
  (* ------------------------------------------------------------------------ *)
  ⟨Lexer.token cpp cases 31c⟩

  (* ------------------------------------------------------------------------ *)
  | eof { EOF }
  | _ (*as c*)   { let c = char_ lexbuf in
                   error (spf "unrecognized character: '%c'" c) }
```

⟨function Lexer.error 30b⟩≡ (160b)

```
  let error s =
    raise (L.Error (spf "Lexical error: %s" s, !L.line))
```

⟨*function* Lexer.loc 31a⟩≡ (160b)
```
  let loc () = !L.line
```

⟨*function* Lexer.char_ 31b⟩≡ (160b)
```
  (* needed only because of ocamllex limitations in ocaml-light
   * which does not support the 'as' feature.
   *)
  let char_ lexbuf =
    let s = Lexing.lexeme lexbuf in
    String.get s 0
```

⟨Lexer.token *cpp cases* 31c⟩≡ (30a)
```
  (* See ../macroprocessor/lexer_cpp.mll *)
  | "#" { TSharp }
```

5.2 Comments and spaces

⟨*constant* Lexer.space 31d⟩≡ (29)
```
  let space = [' ''\t']
```

⟨Lexer.token *space/comment cases* 31e⟩≡ (30a) 31f▷
```
  | space+        { token lexbuf }
```

⟨Lexer.token *space/comment cases* 31f⟩+≡ (30a) ◁31e 31h▷
```
  | "//" [^'\n']* { token lexbuf }
  | "/*"          { comment lexbuf }
```

⟨*rule* Lexer.comment 31g⟩≡ (29)
```
  and comment = parse
    | "*/"          { token lexbuf }
    | [^ '*' '\n']+ { comment lexbuf }
    | '*'           { comment lexbuf }
    | '\n'          { incr Location_cpp.line; comment lexbuf }
    | eof           { error "eof in comment" }
```

⟨Lexer.token *space/comment cases* 31h⟩+≡ (30a) ◁31f
```
  | '\n'          { incr Location_cpp.line; token lexbuf }
```

5.3 Keywords and identifiers

⟨*constant* Lexer.letter 31i⟩≡ (29)
```
  let letter = ['a'-'z''A'-'Z']
```

⟨*constant* Lexer.digit 31j⟩≡ (29)
```
  let digit = ['0'-'9']
```

⟨Lexer.token *keywords/identifiers cases* 32a⟩≡ (30a)
```
  | (letter | '_') (letter | digit | '_')* {
      let s = Lexing.lexeme lexbuf in

      match s with
      | "static" -> Tstatic (loc())
      | "extern" -> Textern (loc())
      (* we could forbid those constructs; they are not really useful *)
      | "auto" -> Tauto (loc())
      | "register" -> Tregister (loc())

      | "const" -> Tconst (loc()) | "volatile" -> Tvolatile (loc())
      | "inline" -> Tinline (loc()) | "restrict" -> Trestrict (loc())

      | "void" -> Tvoid (loc())
      | "char" -> Tchar (loc()) | "short" -> Tshort (loc())
      | "int"  -> Tint  (loc()) | "long"  -> Tlong  (loc())
      | "float"  -> Tfloat  (loc()) | "double"   -> Tdouble   (loc())

      | "signed" -> Tsigned (loc()) | "unsigned" -> Tunsigned (loc())

      | "struct" -> Tstruct (loc()) | "union" -> Tunion (loc())
      | "enum" -> Tenum (loc())
      | "typedef" -> Ttypedef (loc())

      | "if" -> Tif (loc()) | "else" -> Telse (loc())
      | "while" -> Twhile (loc()) | "do" -> Tdo (loc()) | "for" -> Tfor (loc())
      | "break" -> Tbreak (loc()) | "continue" -> Tcontinue (loc())
      | "switch" -> Tswitch (loc())
      | "case" -> Tcase (loc()) | "default" -> Tdefault (loc())
      | "return" -> Treturn (loc()) | "goto" -> Tgoto (loc())

      | "sizeof" -> Tsizeof (loc())
      (* less: USED/SET here? or manage via symbol table *)

      | _ ->
        ⟨Lexer.token() in identifier case, typedef trick 32b⟩
        else TName (loc(), s)
  }
```

5.3.1 Unicode identifiers

5.3.2 Typedef trick, part 1

⟨Lexer.token() *in identifier case, typedef trick* 32b⟩≡ (32a)
```
  if Hashtbl.mem Globals.hids s
  then
    (* typedef trick, because ambiguity in C grammar *)
    (match Hashtbl.find Globals.hids s with
    | A.IdIdent | A.IdEnumConstant -> TName (loc(), s)
    | A.IdTypedef -> TTypeName (loc(), s)
    )
```

5.4 Operators

⟨Lexer.token *symbol cases* 32c⟩≡ (30a) 33a▷
```
  | "+" { TPlus (loc()) } | "-" { TMinus (loc()) }
```

```
  | "*" { TMul  (loc()) } | "/" { TDiv   (loc()) } | "%" { TMod (loc()) }

  | "&"  { TAnd  (loc()) }  | "|"  { TOr     (loc()) } | "^" { TXor (loc()) }
  | "<<" { TInfInf (* TLsh *) (loc()) } | ">>" { TSupSup (* TRsh *) (loc()) }
  | "~"  { TTilde  (loc()) }

  | "&&" { TAndAnd (loc()) } | "||" { TOrOr (loc()) }
  | "!"  { TBang   (loc()) }

  | "++" { TPlusPlus (loc()) } | "--" { TMinusMinus (loc()) }

  | "="  { TEq   (loc()) }
  | "==" { TEqEq (loc()) }  | "!=" { TBangEq (loc()) }

  | "+=" { TOpEq (loc(), A.Plus) } | "-=" { TOpEq (loc(), A.Minus) }
  | "*=" { TOpEq (loc(), A.Mul) }  | "/=" { TOpEq (loc(), A.Div) }
  | "%=" { TOpEq (loc(), A.Mod) }
  | "&=" { TOpEq (loc(), A.And) }        | "|="  { TOpEq (loc(), A.Or) }
  | "^=" { TOpEq (loc(), A.Xor) }
  | ">>="{ TOpEq (loc(), A.ShiftRight) } | "<<=" { TOpEq (loc(), A.ShiftLeft) }

  | "<"  { TInf   (loc()) } | ">"  { TSup   (loc()) }
  | "<=" { TInfEq (loc()) } | ">=" { TSupEq (loc()) }
```

5.5 Punctuations

⟨Lexer.token *symbol cases* 33a⟩+≡ (30a) ◁32c

```
  | "(" { TOPar   (loc()) } | ")" { TCPar   (loc()) }
  | "{" { TOBrace (loc()) } | "}" { TCBrace (loc()) }
  | "[" { TOBra   (loc()) } | "]" { TCBra   (loc()) }

  | "," { TComma (loc()) } | ";" { TSemicolon (loc()) }
  | "->" { TArrow (loc()) }
  | "." { TDot   (loc()) }
  | "?" { TQuestion (loc()) }
  | ":" { TColon    (loc()) }
```

5.6 Numbers

⟨Lexer.token *number cases* 33b⟩≡ (30a)

```
  ⟨Lexer.token octal case 34b⟩
  ⟨Lexer.token hexadecimal case 34d⟩
  | "0x" { error "malformed hex constant" }

  ⟨Lexer.token decimal case 33c⟩

  ⟨Lexer.token float case 34f⟩
  (* special regexp for better error message *)
  | (digit+ | digit* '.' digit+) ['e''E'] ('+' | '-')?
     { error "malformed fp constant exponent" }
```

5.6.1 Decimals

⟨Lexer.token *decimal case* 33c⟩≡ (33b)

```
  | (['0'-'9'] digit*) (*as s*) (['U''u']? (*as unsigned*)) (['L''l']* (*as long*))
```

```
    { let (s, unsigned, long) =
        let s = Lexing.lexeme lexbuf in
        s =~ "\\([0-9]+\\)\\([Uu]?\\)\\([Ll]*\\)" |> ignore;
        Regexp_.matched3 s
      in
      TIConst (loc(), s, inttype_of_suffix unsigned long)}
```

5.6.2 Octals and hexadecimals

⟨*constant* `Lexer.oct` 34a⟩≡ (29)
```
let oct = ['0'-'7']
```

⟨`Lexer.token` *octal case* 34b⟩≡ (33b)
```
| "0"  (oct+ (*as s*)) (['U''u']? (*as unsigned*)) (['L''l']* (*as long*))
    { let (s, unsigned, long) =
        let s = Lexing.lexeme lexbuf in
        s =~ "0\\([0-7]+\\)\\([Uu]?\\)\\([Ll]*\\)" |> ignore;
        Regexp_.matched3 s
      in
      TIConst(loc(), "0o" ^ s, inttype_of_suffix unsigned long)}
```

⟨*constant* `Lexer.hex` 34c⟩≡ (29)
```
let hex = (digit | ['A'-'F''a'-'f'])
```

⟨`Lexer.token` *hexadecimal case* 34d⟩≡ (33b)
```
| "0x" (hex+ (*as s*))  (['U''u']? (*as unsigned*)) (['L''l']* (*as long*))
    { let (s, unsigned, long) =
        let s = Lexing.lexeme lexbuf in
        s =~ "0x\\([0-9A-Fa-f]+\\)\\([Uu]?\\)\\([Ll]*\\)" |> ignore;
        Regexp_.matched3 s
      in
      TIConst(loc(), "0x" ^ s, inttype_of_suffix unsigned long)}
```

5.6.3 Unsigned and long numbers

⟨*function* `Lexer.inttype_of_suffix` 34e⟩≡ (160b)
```
let inttype_of_suffix sign size =
  let sign =
    match String.lowercase_ascii sign with
      | "" -> T.Signed
      | "u" -> T.Unsigned
      | s -> error (spf "Impossible: wrong sign suffix: %s" s)
  in
  match String.lowercase_ascii size with
  | "" -> T.Int, sign
  | "l" -> T.Long, sign
  | "ll" -> T.VLong, sign
  | s -> error (spf "Impossible: wrong int size suffix: %s" s)
```

5.6.4 Floats

⟨`Lexer.token` *float case* 34f⟩≡ (33b)
```
(* stricter: I impose some digit+ after '.' and after 'e' *)
| ((digit+ | digit* '.' digit+) (['e''E'] ('+' | '-')? digit+)?) (*as s*)
    (['F''f']* (*as float*))
   { let (s, float) =
       let s = Lexing.lexeme lexbuf in
```

```
        s =~ "\\([^Ff]+\\)\\([Ff]*\\)$" |> ignore;
        Regexp_.matched2 s
      in
      TFConst (loc(), s, floattype_of_suffix float) }
```

⟨*function* Lexer.floattype_of_suffix 35a⟩≡ (160b)
```
let floattype_of_suffix s =
  match String.lowercase_ascii s with
  | "" -> T.Double
  | "f" -> T.Float
  | s -> error (spf "Impossible: wrong float size suffix: %s" s)
```

5.7 Characters

⟨Lexer.token *chars/strings cases* 35b⟩≡ (30a) 36a ▷
```
(* converting characters in integers *)
| "'" { TIConst (loc(), spf "%d" (char lexbuf), (T.Char, T.Signed)) }
```

⟨*rule* Lexer.char 35c⟩≡ (29)
```
and char = parse
  | "''"                              { Char.code '\'' }
  (* less: 5c allows up to 8 octal number when in L'' mode *)
  | "\\" ((oct oct? oct?) (*as s*)) "'"
      { let s = Lexing.lexeme lexbuf |>
            String_.drop_prefix 1 |> String_.drop_suffix 1
        in
        int_of_string ("0o" ^ s) }
  | "\\" (['a'-'z' '\\' '\''] (*as c*)) "'"
      { let c = String.get (Lexing.lexeme lexbuf) 1 in
        code_of_escape_char c }
  | '\\' '\n' { char lexbuf }
  | [^ '\\' '\'' '\n'] (*as c*)  "'"
      { let c = String.get (Lexing.lexeme lexbuf) 0 in
        Char.code c }
  | '\n' { error "newline in character" }
  | eof  { error "end of file in character" }
  | _    { error "missing '" }
```

5.7.1 Escaping characters

⟨*function* Lexer.code_of_escape_char 35d⟩≡ (160b)
```
let code_of_escape_char c =
  match c with
  | 'n' -> Char.code '\n' | 'r' -> Char.code '\r'
  | 't' -> Char.code '\t' | 'b' -> Char.code '\b'

  (* compatibility with plan 9 C code? *)
  | 'f' -> Logs.err (fun m -> m "unknown \\f"); 0x00
  (* could be removed, special 5c escape char *)
  | 'a' -> 0x07 | 'v' -> 0x0b

  | '\\' | '\'' | '"' -> Char.code c
  (* stricter: we disallow \ with unknown character *)
  | _ -> error "unknown escape sequence"
```

5.7.2 Unicode characters

5.8 Strings

⟨Lexer.token *chars/strings cases* 36a⟩+≡ (30a) ◁35b
```
  | '"' { TString (loc(), string lexbuf, T.Array (None, T.I (T.Char,T.Signed)))}
```

⟨*rule* Lexer.string 36b⟩≡ (29)
```
  and string = parse
    | '"' { "" }
    | "\\" ((oct oct oct) (*as s*))
        { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
          let i = int_of_string ("0o" ^ s) in string_of_ascii i ^ string lexbuf }
    | "\\" (['a'-'z' '\\' '"'] (*as c*))
        { let c = String.get (Lexing.lexeme lexbuf) 1 in
          let i = code_of_escape_char c in string_of_ascii i ^ string lexbuf  }
    (* strings can contain newline! but they must be escaped before *)
    | '\\' '\n' { "\n" ^ string lexbuf }
    | [^ '\\' '"' '\n']+
        { let x = Lexing.lexeme lexbuf in x ^ string lexbuf }
    | '\n' { error "newline in string" }
    | eof  { error "end of file in string" }
    | _    { error "undefined character in string" }
```

⟨*function* Lexer.string_of_ascii 36c⟩≡ (160b)
```
  let string_of_ascii i =
    String.make 1 (Char.chr i)
```

5.8.1 Unicode strings

Chapter 6

Parsing

6.1 Overview

⟨*signature* Parse.parse 37a⟩≡ (160d)
```
(* will macropreprocess internally first *)
val parse:
  < Cap.open_in; .. > -> Preprocessor.conf -> Fpath.t -> Ast.program
```

⟨*signature* Parse.parse_no_cpp 37b⟩≡ (160d)
```
(* internals *)
val parse_no_cpp:
  Chan.i -> Ast.program
```

⟨*function* Parse.parse_no_cpp 37c⟩≡ (160e)
```
let parse_no_cpp (chan : Chan.i) =
  L.line := 1;
  let lexbuf = Lexing.from_channel chan.ic in
  (try
      Parser.prog Lexer.token lexbuf, []
    with Parsing.Parse_error ->
        failwith (spf "Syntax error: line %d" !L.line)
  )
```

⟨Parser.mly 37d⟩≡
```
%{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Either

open Ast
module T = Type
module L = Location_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The original 5c does many things during parsing; Instead, we
 * do the minimum here. We just return a very simple AST.
 *
 * Limitations compared to 5c (and sometimes also to ANSI C or C11):
 *  - no support for old style parameter declaration
 *    (obsolete practice anyway)
 *  - impose a certain order for the storage, qualifier, and type
 *    (everybody follow this convention anyway)
 *  - no implicit single 'signed' means 'signed int'.
```

```
 *    Signed has to have an explicit int-type after.
 *  - sure? forbid definitions (typedefs, struct, enum) not at toplevel
 *    (confusing anyway?)
 *    (but then would no need blockid for those, or just for nested struct def)
 *  - forbid typedefs inside forexpr
 *    (who uses that anyway?)
 *  - can not mix qualified and not qualified elements in initializers lists
 *
 * todo:
 *  - add qualifiers in AST
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨Parser helpers 161a⟩
%}

/*(*************************************************************************)*/
/*(*1 Tokens *)*/
/*(*************************************************************************)*/
⟨Parser tokens 15a⟩

/*(*************************************************************************)*/
/*(*1 Priorities *)*/
/*(*************************************************************************)*/
⟨Parser tokens priorities 42a⟩

/*(*************************************************************************)*/
/*(*1 Rules type declaration *)*/
/*(*************************************************************************)*/
⟨Parser type declarations 42b⟩
%start prog

%%
⟨Parser grammar 40d⟩
```

⟨*function* `Parser.error` 38a⟩≡ (161a)

```
let error s =
  raise (L.Error (spf "Syntax error: %s" s, !L.line))
```

6.2 Parsing environment and scope management

⟨*global* `Parser.block_counter` 38b⟩≡ (161a)

```
let block_counter : Ast.blockid ref = ref 0
```

⟨*type* `Parser.env` 38c⟩≡ (161a)

```
(* To manage scope (used notably to recognize typedefs in the lexer)
 *
 * alt: could have a recursive environment with 'parent: env;' and so
 * we could remove the need for those xxx_scope.
 * we would need then a lookup_id and lookup_tag that would
 * possibly look in the parent field.
 *)
type env = {
  (* there are mainly two namespaces in C: one for ids and one for tags *)
  ids:  (string, idkind * Ast.blockid) Hashtbl.t;
  tags: (string, tagkind * Ast.blockid) Hashtbl.t;
```

```
    mutable block: Ast.blockid;

    mutable ids_scope:  (string list) list;
    mutable tags_scope: (string list) list;
    mutable block_scope: Ast.blockid list;
  }
```

⟨*global* Parser.env 39a⟩≡ (161a)
```
  let env = {
    ids = Hashtbl_.create ();
    tags = Hashtbl_.create ();
    block = 0;

    ids_scope = [];
    tags_scope = [];
    block_scope = [];
  }
```

⟨*function* Parser.add_id 39b⟩≡ (161a)
```
  (* Should we warn if 'id' already declared? No, because at the toplevel
   * it is ok to redeclare the same variable or prototype.
   * So better to do those checks in another phase (in Check.ml and Typecheck.ml).
   *)
  let add_id env id idkind =
    Hashtbl.add Globals.hids id idkind;

    Hashtbl.add env.ids id (idkind, env.block);
    match env.ids_scope with
    | xs::xss -> env.ids_scope <- (id::xs)::xss
    | [] -> env.ids_scope <- [[id]]
```

⟨*function* Parser.add_tag 39c⟩≡ (161a)
```
  let add_tag env tag tagkind =
    Hashtbl.add env.tags tag (tagkind, env.block);
    match env.tags_scope with
    | xs::xss -> env.tags_scope <- (tag::xs)::xss
    | [] -> env.tags_scope <- [[tag]]
```

⟨*function* Parser.new_scope 39d⟩≡ (161a)
```
  let new_scope env =
    incr block_counter;
    env.block <- !block_counter;
    env.block_scope <- env.block :: env.block_scope;
    env.ids_scope <- []::env.ids_scope;
    env.tags_scope <- []::env.tags_scope;
    ()
```

⟨*function* Parser.pop_scope 39e⟩≡ (161a)
```
  let pop_scope env =
    (match env.block_scope, env.ids_scope, env.tags_scope with
    | x::xs, ys::yss, zs::zss ->
        env.block <- x;
        env.block_scope <- xs;
        ys |> List.iter (fun id ->
          Hashtbl.remove env.ids id;

          Hashtbl.remove Globals.hids id;
        );
        zs |> List.iter (fun tag ->
```

```
        Hashtbl.remove env.tags tag
      );
      env.ids_scope <- yss;
      env.tags_scope <- zss;
  | _ -> raise (Impossible "pop empty declaration stack, grammar wrong")
  )
  (* less: return an optional list of instructions at some point *)
```

6.3 Lifting nested definitions to the toplevel

⟨*global* Parser.defs 40a⟩≡ (161a)

```
(* 'defs' contains things we lift up in the AST (struct defs, enums, typedefs).
 * Note that we tag those defs with a blockid, so there is no escape-scope
 * problem.
 *)
let defs =
  ref []
```

⟨*function* Parser.get_and_reset 40b⟩≡ (161a)

```
let get_and_reset x =
  let v = !x in
  x := [];
  v
```

6.4 Naming anonymous definitions

⟨*function* Parser.gensym 40c⟩≡ (161a)

```
(* for anonymous struct/union/enum and structure elements *)
let gensym_counter = ref 0
let gensym () =
  incr gensym_counter;
  spf "|sym%d|" !gensym_counter
```

6.5 Grammar overview

⟨*Parser grammar* 40d⟩≡ (37d)

```
/*(*************************************************************************)*/
/*(*1 Program *)*/
/*(*************************************************************************)*/
⟨rule Parser.prog 42c⟩
⟨rule Parser.prog1 42d⟩

/*(*************************************************************************)*/
/*(*1 Declarations *)*/
/*(*************************************************************************)*/
/*(*----------------------------------------*)*/
/*(*2 External declarators *)*/
/*(*----------------------------------------*)*/
⟨rule Parser.xdecl 42e⟩
⟨rule Parser.xdlist 43c⟩
⟨rule Parser.storage_and_type_xdecor 44d⟩

/*(*----------------------------------------*)*/
/*(*2 Automatic declarators *)*/
/*(*----------------------------------------*)*/
```

⟨*rule* `Parser.adecl` 56e⟩
⟨*rule* `Parser.adlist` 57a⟩

```
/*(*************************************************************************)*/
/*(*1 Statements *)*/
/*(*************************************************************************)*/
```
⟨*rule* `Parser.block` 45g⟩
⟨*other block related rules* 45h⟩

⟨*rule* `Parser.stmnt` 45b⟩
⟨*rule* `Parser.ulstmnt` 45c⟩
⟨*other stmt related rules* 161b⟩

```
/*(*************************************************************************)*/
/*(*1 Expressions *)*/
/*(*************************************************************************)*/
```
⟨*rule* `Parser.expr` 47h⟩
⟨*other expr related rules* 47i⟩
⟨*other expr ebnf rules* 46e⟩

⟨*rule* `Parser.string` 48i⟩

```
/*(*************************************************************************)*/
/*(*1 Initializers *)*/
/*(*************************************************************************)*/
```
⟨*rule* `Parser.init` 51b⟩
⟨*other init related rules* 51c⟩

```
/*(*************************************************************************)*/
/*(*1 Types *)*/
/*(*************************************************************************)*/
/*(*---------------------------------------*)*/
/*(*2 Types part 1 (left part of a type) *)*/
/*(*---------------------------------------*)*/
```
⟨*rule* `Parser.type_` 51i⟩
⟨*rule* `Parser.simple_type` 52b⟩
⟨*rule* `Parser.complex_type` 52c⟩

⟨*other struct related rules* 53f⟩

```
/*(*---------------------------------------*)*/
/*(*2 Types part 2 (right part of a type) *)*/
/*(*---------------------------------------*)*/
```
⟨*rule* `Parser.xdecor` 43d⟩
⟨*other declarator related rules* 44a⟩

⟨*rule* `Parser.paramlist` 56d⟩
⟨*rule* `Parser.zparamlist` 56a⟩

```
/*(*---------------------------------------*)*/
/*(* abstract declarators *)*/
/*(*---------------------------------------*)*/
```
⟨*rule* `Parser.abdecor` 57b⟩
⟨*other abstract declarator related rules* 57c⟩

```
/*(*---------------------------------------*)*/
/*(*3 qualifiers *)*/
/*(*---------------------------------------*)*/
```
⟨*rule* `Parser.qualifier` 53a⟩
⟨*other qualifier related rules* 53b⟩

```
  /*(*************************************************************************)*/
  /*(*1 Struct/union/enum body *)*/
  /*(*************************************************************************)*/
  ⟨rule Parser.sbody 54b⟩
  ⟨other structure body related rules 54c⟩

  ⟨rule Parser.enum 55a⟩
  ⟨rule Parser.const_expr 55b⟩

  /*(*************************************************************************)*/
  /*(*1 Storage, qualifiers *)*/
  /*(*************************************************************************)*/
  ⟨rule Parser.storage 52e⟩
  ⟨rule Parser.storage_and_type 52f⟩
```

⟨*Parser tokens priorities* 42a⟩≡ (37d)

```
  /*(* must be at the top so that it has the lowest priority *)*/
  %nonassoc LOW_PRIORITY_RULE
  /*(* see conflicts.txt *)*/
  %nonassoc Telse

  /*(* in 5c but not in orig_c.mly *)*/
  %left   TSemicolon
  %left   TComma
  %right  TEq TOpEq
  %right  TQuestion TColon

  /*(* same than in orig_c.mly *)*/
  %left   TOrOr
  %left   TAndAnd
  %left   TOr
  %left   TXor
  %left   TAnd
  %left   TEqEq TBangEq
  %left   TInf TSup TInfEq TSupEq
  %left   TInfInf TSupSup
  %left   TPlus TMinus
  %left   TMul TDiv TMod

  /*(* in 5c but not in orig_c.mly *)*/
  %right  TMinusMinus TPlusPlus TArrow TDot TOBra TOPar
```

⟨*Parser type declarations* 42b⟩≡ (37d)

```
  %type <Ast.toplevels> prog
```

⟨*rule* Parser.prog 42c⟩≡ (40d)

```
  prog: prog1 EOF   { $1 }
```

⟨*rule* Parser.prog1 42d⟩≡ (40d)

```
  prog1:
   |  /*(*empty*)*/ { [] }
   | prog1 xdecl    { $1 @ $2 }
```

6.6 Declarations and definitions, part one

⟨*rule* Parser.xdecl 42e⟩≡ (40d)

```
  xdecl:
    ⟨Parser.xdecl cases 43a⟩
```

6.6.1 No-declarator declarations

⟨Parser.xdecl *cases* 43a⟩≡ (42e) 43b ▷
```
  | storage_and_type        TSemicolon
      { (* stricter: *)
        if !defs = []
        then error "declaration without any identifier";

        get_and_reset defs;
      }
```

6.6.2 Declarator-based declarations

⟨Parser.xdecl *cases* 43b⟩+≡ (42e) ◁43a 44c ▷
```
  | storage_and_type xdlist TSemicolon
      { (* less: could suggest to separate struct/enum def from vardecl *)
        get_and_reset defs @
        ($2 |> List.map (fun (((id, loc), typ2), v_init) ->
           let (sto_or_typedef, typ1) = $1 in
           let typ = typ2 typ1 in
           (match sto_or_typedef, v_init with
           | Left v_storage, _ ->
               add_id env id IdIdent;
               VarDecl { v_name = (id, env.block (* 0 *));
                         v_loc = loc; v_type = typ; v_storage; v_init; }
           (* stricter: clang also reports this, but not 5c *)
           | Right (), Some _ ->
               error "initializer with typedef"
           | Right (), None ->
               add_id env id IdTypedef;
               TypeDef { typedef_name = (id, env.block);
                         typedef_loc = loc; typedef_type = typ; }
           )
         ))
      }
```

⟨*rule* Parser.xdlist 43c⟩≡ (40d)
```
  xdlist:
   | xdecor          { [$1, None] }
   ⟨Parser.xdlist other cases 51a⟩
   | xdlist TComma xdlist { $1 @ $3 }
```

6.6.3 Entity declarations, the declarator

⟨*rule* Parser.xdecor 43d⟩≡ (40d)
```
  /*
  (* declarator return a couple:
   *  (name, partial type (a function to be applied to return type))
   *
   * Note that with 'int* foo(int)' we must return
   * TFunction(Pointer int, [int]) and not Pointer (Func(int,[int])).
   * The grammar parses the xdecor part  '*foo(int)' as:
   *         *
   *         |
   *         ()
   *       /   \
   *      foo     [ int]
   *
```

```
   * So when we return the partial function for TMul, we must
   * apply first TPointer to the return type 'x', and then apply
   * the function for xdecor.
   *
   *
   *
   * less: handle qualifiers
   *)*/
  xdecor:
   | xdecor2                { $1 }
   ⟨Parser.xdecor other cases 55c⟩
```

⟨*other declarator related rules* 44a⟩≡ (40d)

```
  /*(* use 'tag' here too, because you can have 'foo foo;' declarations *)*/
  xdecor2:
   | tag
       { (snd $1, fst $1), (fun x -> x) }
   | TOPar xdecor TCPar
       { $2 }
   ⟨Parser.xdecor2 other cases 55d⟩
```

⟨*rule* Parser.tag 44b⟩≡ (161b)

```
  tag:
   | TName     { $1 }
   | TTypeName { $1 }
```

6.6.4 Function definitions

⟨Parser.xdecl *cases* 44c⟩+≡ (42e) ◁43b

```
  | storage_and_type_xdecor block_no_new_scope
      { let ((id, f_loc), f_type, f_storage) = $1 in
        (* pop_scope from new_scope done in storage_and_type_xdecor *)
        pop_scope env;
        [ FuncDef { f_name = id; f_loc; f_type; f_body = $2; f_storage; } ]
      }
```

⟨*rule* Parser.storage_and_type_xdecor 44d⟩≡ (40d)

```
  storage_and_type_xdecor: storage_and_type xdecor
      { (* stricter: *)
        if !defs <> []
        then error "move struct or typedef definition outside the function";

        let ((id, loc), typ2) = $2 in
        let (sto_or_typedef, typ1) = $1 in
        let typ = typ2 typ1 in

        (match typ.t, sto_or_typedef with
        | TFunction (ret, (params, varargs)), Left sto ->
            (* add in global scope the function name *)
            add_id env id IdIdent;

            (* add in new scope the parameters *)
            new_scope env;
            let params = params |> List.map (fun p ->
              match p.p_name with
              | Some (id, _) ->
                  add_id env id IdIdent;
                  { p with p_name = Some (id, env.block) }
              | None -> p
```

```
          )
          in
          let ft = (ret, (params, varargs)) in
          ((id, loc), ft, sto)

      (* stricter: *)
      | TFunction _, Right _ ->
          error "a function definition can not be a type definition"
      (* stricter: it could be TTypeName that resolves to a TFunction, but
       * I resolve typedefs later so I have to forbid it here, and it is
       * confusing anyway as you can not see the parameter.
       *)
      | _, _ -> error "not a function type"
      )
    }
```

⟨*rule* Parser.block_no_new_scope 45a⟩≡ (45h)
```
block_no_new_scope: TOBrace slist TCBrace { mk_st (Block $2) $1 }
```

6.7 Statements

⟨*rule* Parser.stmnt 45b⟩≡ (40d)
```
stmnt:
 |        ulstmnt { $1 }
 | labels ulstmnt { $1 $2 }
```

⟨*rule* Parser.ulstmnt 45c⟩≡ (40d)
```
ulstmnt:
 ⟨Parser.ulstmnt cases 45d⟩
```

⟨Parser.ulstmnt *cases* 45d⟩≡ (45c) 45f▷
```
| cexpr TSemicolon { mk_st (ExprSt $1) $2 }
/*(* used when do for(...) ; to have an empty statement *)*/
|       TSemicolon { mk_st (Block []) $1 }
```

⟨*function* Parser.mk_st 45e⟩≡ (161a)
```
let mk_st st loc = { s = st; s_loc = loc }
```

6.7.1 Blocks

⟨Parser.ulstmnt *cases* 45f⟩+≡ (45c) ◁45d 46c▷
```
| block { $1 }
```

⟨*rule* Parser.block 45g⟩≡ (40d)
```
block: tobrace slist tcbrace { mk_st (Block $2) $1 }
```

⟨*other block related rules* 45h⟩≡ (40d)
```
⟨rule Parser.tobrace 46a⟩
⟨rule Parser.tcbrace 46b⟩
⟨rule Parser.slist 45i⟩

⟨rule Parser.block_no_new_scope 45a⟩
```

⟨*rule* Parser.slist 45i⟩≡ (45h)
```
slist:
 | /*(*empty*)*/ { [] }
 | slist adecl   { $1 @ $2 }
 | slist stmnt   { $1 @ [$2] }
```

⟨*rule* Parser.tobrace 46a⟩≡ (45h)
```
tobrace: TOBrace { new_scope env; $1 }
```

⟨*rule* Parser.tcbrace 46b⟩≡ (45h)
```
tcbrace: TCBrace { pop_scope env; $1 }
```

6.7.2 Conditionals

⟨Parser.ulstmnt *cases* 46c⟩+≡ (45c) ◁45f 46d▷
```
  | Tif TOPar cexpr TCPar stmnt %prec LOW_PRIORITY_RULE
      {
        if $5.s = Block []
        then Error.warn "empty if body" $5.s_loc;
        mk_st (If ($3, $5, mk_st (Block[]) $1)) $1
      }
  | Tif TOPar cexpr TCPar stmnt Telse stmnt
      {
        if $5.s = Block []
        then Error.warn "empty if body" $5.s_loc;
        if $7.s = Block []
        then Error.warn "empty else body" $7.s_loc;
        mk_st (If ($3, $5, $7)) $1
      }
```

6.7.3 Loops

⟨Parser.ulstmnt *cases* 46d⟩+≡ (45c) ◁46c 47a▷
```
  | Twhile TOPar cexpr TCPar stmnt                { mk_st (While ($3, $5)) $1 }
  | Tdo stmnt Twhile TOPar cexpr TCPar TSemicolon { mk_st (DoWhile ($2, $5)) $1 }
  | tfor TOPar forexpr TSemicolon zcexpr TSemicolon zcexpr TCPar stmnt
      { pop_scope env;
        mk_st (For ($3, $5, $7, $9)) $1
      }
```

⟨*other expr ebnf rules* 46e⟩≡ (40d) 50d▷
```
  zcexpr:
   | /*(*empty*)*/ { None }
   | cexpr         { Some $1 }
```

⟨*rule* Parser.tfor 46f⟩≡ (161b)
```
  tfor: Tfor { new_scope env; $1 }
```

⟨*rule* Parser.forexpr 46g⟩≡ (161b)
```
  forexpr:
   | zcexpr
       { Left $1 }
   | storage_and_type adlist
       { Right
           (* less: could factorize code with xdecl *)
           ($2 |> List.map (fun (((id, v_loc), typ2), v_init) ->
            let (sto_or_typedef, typ1) = $1 in
            let v_type = typ2 typ1 in
            (match sto_or_typedef with
            | Left v_storage ->
              add_id env id IdIdent;
              { v_name = (id, env.block); v_loc; v_type; v_storage; v_init; }
            (* stricter: *)
            | Right _ -> error "typedefs inside 'for' are forbidden"
```

```
        )
      ))
    }
```

6.7.4 Control flow jumps

⟨Parser.ulstmnt *cases* 47a⟩+≡ (45c) ◁46d 47b▷
```
  | Treturn zcexpr TSemicolon { mk_st (Return $2) $1 }
```

⟨Parser.ulstmnt *cases* 47b⟩+≡ (45c) ◁47a 47c▷
```
  | Tbreak TSemicolon         { mk_st Break $1 }
  | Tcontinue TSemicolon      { mk_st Continue $1 }
```

6.7.5 Labels and goto

⟨Parser.ulstmnt *cases* 47c⟩+≡ (45c) ◁47b 47f▷
```
  | Tgoto tag TSemicolon      { mk_st (Goto (snd $2)) $1 }
```

⟨*rule* Parser.labels 47d⟩≡ (161b)
```
  labels:
   | label        { (fun st -> $1 st)  }
   | labels label { (fun st -> $1 ($2 st)) }
```

⟨*rule* Parser.label 47e⟩≡ (161b)
```
  label:
   /*(* less: not tag here? can not conflict with typedef? *)*/
   | TName      TColon  { (fun st -> mk_st (Label (snd $1, st)) $2)  }
   ⟨Parser.label other cases 47g⟩
```

6.7.6 Switch

⟨Parser.ulstmnt *cases* 47f⟩+≡ (45c) ◁47c
```
  /*(* stricter: I impose a block, not any stmnt *)*/
  | Tswitch TOPar cexpr TCPar block
      { (* less: generate (0:int - (0:int - x)) *)
        mk_st (Switch ($3, $5)) $1
      }
```

⟨Parser.label *other cases* 47g⟩≡ (47e)
```
  | Tcase expr TColon  { (fun st -> mk_st (Case ($2, st)) $3) }
  | Tdefault   TColon  { (fun st -> mk_st (Default st) $2) }
```

6.8 Expressions

⟨*rule* Parser.expr 47h⟩≡ (40d)
```
  expr:
   | xuexpr { $1 }
   ⟨Parser.expr other cases 49b⟩
```

⟨*other expr related rules* 47i⟩≡ (40d)
```
  ⟨rule Parser.xuexpr 48a⟩
  ⟨rule Parser.uexpr 48b⟩
  ⟨rule Parser.pexpr 48c⟩
  ⟨rule Parser.cexpr 48e⟩
  ⟨rule Parser.lexpr 51h⟩
```

⟨*rule* `Parser.xuexpr` 48a⟩≡ (47i)
```
xuexpr:
 | uexpr { $1 }
 ⟨Parser.xuexpr other cases 50f⟩
```

⟨*rule* `Parser.uexpr` 48b⟩≡ (47i)
```
uexpr:
 | pexpr { $1 }
 ⟨Parser.uexpr other cases 49a⟩
```

⟨*rule* `Parser.pexpr` 48c⟩≡ (47i)
```
pexpr:
 ⟨Parser.pexpr cases 48d⟩
```

⟨`Parser.pexpr` *cases* 48d⟩≡ (48c) 48f▷
```
 | TOPar cexpr TCPar { $2 }
```

⟨*rule* `Parser.cexpr` 48e⟩≡ (47i)
```
cexpr:
 | expr { $1 }
 | cexpr TComma cexpr { mk_e (Sequence ($1, $3)) $2 }
```

6.8.1 Numeric constants

⟨`Parser.pexpr` *cases* 48f⟩+≡ (48c) ◁48d 48h▷
```
 | TIConst { let (loc, a,b) = $1 in mk_e (Int (a,b))    loc }
 | TFConst { let (loc, a,b) = $1 in mk_e (Float (a,b))  loc }
```

⟨*function* `Parser.mk_e` 48g⟩≡ (161a)
```
let mk_e e loc = { e = e; e_loc = loc; e_type = T.Void }
```

6.8.2 String constants

⟨`Parser.pexpr` *cases* 48h⟩+≡ (48c) ◁48f 48j▷
```
 | string  { let (loc, a,b) = $1 in mk_e (String (a,b)) loc }
```

⟨*rule* `Parser.string` 48i⟩≡ (40d)
```
string:
 | TString        { $1 }
 | string TString
     { let (loc1, s1,t1) = $1 in let (_loc2, s2,t2) = $2 in
       (* stricter: better error message, 5c just says "syntax error" *)
       if t1 <> t2
       then error "incompatible strings"
       else loc1, s1 ^ s2, t1
     }
```

6.8.3 Entity uses

⟨`Parser.pexpr` *cases* 48j⟩+≡ (48c) ◁48h 50a▷
```
 | TName
     { let (loc, id) = $1 in
       try
         let (idkind, blockid) = Hashtbl.find env.ids id in
         assert (idkind <> IdTypedef);
         mk_e (Id (id, blockid)) loc
       with Not_found ->
         (* stricter: if caller is Call, still forbid implicit decl of func! *)
         error (spf "name not declared: %s" id)
         (*Id ($1, 0)*)
     }
```

6.8.4 Arithmetic expressions

⟨Parser.uexpr *other cases* 49a⟩≡ (48b) 49d▷
```
  | TPlus xuexpr  { mk_e (Unary (UnPlus, $2)) $1 }
  | TMinus xuexpr { mk_e (Unary (UnMinus, $2)) $1 }
```

⟨Parser.expr *other cases* 49b⟩≡ (47h) 49c▷
```
  | expr TPlus expr   { mk_e (Binary ($1, Arith Plus, $3)) $2 }
  | expr TMinus expr  { mk_e (Binary ($1, Arith Minus, $3)) $2 }
  | expr TMul expr    { mk_e (Binary ($1, Arith Mul, $3)) $2 }
  | expr TDiv expr    { mk_e (Binary ($1, Arith Div, $3)) $2 }
  | expr TMod expr    { mk_e (Binary ($1, Arith Mod, $3)) $2 }
```

⟨Parser.expr *other cases* 49c⟩+≡ (47h) ◁49b 49e▷
```
  | expr TAnd expr    { mk_e (Binary ($1, Arith And, $3)) $2 }
  | expr TXor expr    { mk_e (Binary ($1, Arith Xor, $3)) $2 }
  | expr TOr expr     { mk_e (Binary ($1, Arith Or, $3)) $2 }
```

⟨Parser.uexpr *other cases* 49d⟩+≡ (48b) ◁49a 49g▷
```
  | TTilde xuexpr { mk_e (Unary (Tilde, $2)) $1 }
```

⟨Parser.expr *other cases* 49e⟩+≡ (47h) ◁49c 49f▷
```
  | expr TSupSup expr { mk_e (Binary ($1, Arith ShiftRight, $3)) $2 }
  | expr TInfInf expr { mk_e (Binary ($1, Arith ShiftLeft , $3)) $2 }
```

6.8.5 Boolean expressions

⟨Parser.expr *other cases* 49f⟩+≡ (47h) ◁49e 49h▷
```
  | expr TAndAnd expr { mk_e (Binary ($1, Logical AndLog, $3)) $2 }
  | expr TOrOr expr   { mk_e (Binary ($1, Logical OrLog, $3)) $2 }
```

⟨Parser.uexpr *other cases* 49g⟩+≡ (48b) ◁49d 49k▷
```
  | TBang xuexpr  { mk_e (Unary (Not, $2)) $1 }
```

⟨Parser.expr *other cases* 49h⟩+≡ (47h) ◁49f 49i▷
```
  | expr TEqEq expr   { mk_e (Binary ($1, Logical Eq, $3)) $2 }
  | expr TBangEq expr { mk_e (Binary ($1, Logical NotEq, $3)) $2 }
```

⟨Parser.expr *other cases* 49i⟩+≡ (47h) ◁49h 49j▷
```
  | expr TInf expr    { mk_e (Binary ($1, Logical Inf, $3)) $2 }
  | expr TSup expr    { mk_e (Binary ($1, Logical Sup, $3)) $2 }
  | expr TInfEq expr  { mk_e (Binary ($1, Logical InfEq, $3)) $2 }
  | expr TSupEq expr  { mk_e (Binary ($1, Logical SupEq, $3)) $2 }
```

6.8.6 Assignments

⟨Parser.expr *other cases* 49j⟩+≡ (47h) ◁49i 50g▷
```
  | expr TEq expr     { mk_e (Assign (SimpleAssign, $1, $3)) $2 }
  | expr TOpEq expr   { mk_e (Assign (OpAssign (snd $2), $1, $3)) (fst $2) }
```

6.8.7 Pointers

⟨Parser.uexpr *other cases* 49k⟩+≡ (48b) ◁49g 50h▷
```
  | TMul xuexpr  { mk_e (Unary (DeRef, $2)) $1 }
  | TAnd xuexpr  { mk_e (Unary (GetRef, $2)) $1 }
```

6.8.8 Array accesses

⟨Parser.pexpr *cases* 50a⟩+≡ (48c) ◁48j 50b▷
```
  /*(* stricter: was cexpr, but ugly to allow cexpr here *)*/
  | pexpr TOBra expr TCBra  { mk_e (ArrayAccess ($1, $3)) $2 }
```

6.8.9 Field accesses

⟨Parser.pexpr *cases* 50b⟩+≡ (48c) ◁50a 50c▷
```
  /*(* we could unsugar here RecordPtAccess; instead we do it in typecheck.ml*)*/
  | pexpr TDot tag   { mk_e (RecordAccess ($1, snd $3)) $2 }
  | pexpr TArrow tag { mk_e (RecordPtAccess ($1, snd $3)) $2 }
```

6.8.10 Function calls

⟨Parser.pexpr *cases* 50c⟩+≡ (48c) ◁50b 50i▷
```
  /*(* less: could do implicit declaration of unknown function *)*/
  | pexpr TOPar zelist TCPar { mk_e (Call ($1, $3)) $2 }
```

⟨*other expr ebnf rules* 50d⟩+≡ (40d) ◁46e 50e▷
```
  zelist:
   | /*(*empty*)*/ { [] }
   | elist         { $1 }
```

⟨*other expr ebnf rules* 50e⟩+≡ (40d) ◁50d 55e▷
```
  elist:
   | expr { [$1] }
   | elist TComma elist { $1 @ $3 }
```

6.8.11 Cast

⟨Parser.xuexpr *other cases* 50f⟩≡ (48a)
```
  | TOPar qualifier_and_type abdecor TCPar xuexpr  { mk_e (Cast ($3 $2, $5)) $1 }
```

6.8.12 Ternary expressions

⟨Parser.expr *other cases* 50g⟩+≡ (47h) ◁49j
```
  | expr TQuestion cexpr TColon expr { mk_e (CondExpr ($1, $3, $5)) $2 }
```

6.8.13 Prefix/postfix

⟨Parser.uexpr *other cases* 50h⟩+≡ (48b) ◁49k
```
  | TPlusPlus xuexpr   { mk_e (Prefix (Inc, $2)) $1 }
  | TMinusMinus xuexpr { mk_e (Prefix (Dec, $2)) $1 }
```

⟨Parser.pexpr *cases* 50i⟩+≡ (48c) ◁50c 50j▷
```
  | pexpr TPlusPlus   { mk_e (Postfix ($1, Inc)) $2 }
  | pexpr TMinusMinus { mk_e (Postfix ($1, Dec)) $2 }
```

6.8.14 `sizeof()`

⟨Parser.pexpr *cases* 50j⟩+≡ (48c) ◁50i
```
  | Tsizeof TOPar qualifier_and_type abdecor TCPar
      { mk_e (SizeOf (Right ($4 $3))) $1 }
  | Tsizeof uexpr
      { mk_e (SizeOf (Left $2)) $1 }
```

6.9 Initializers and designators

⟨Parser.xdlist *other cases* 51a⟩≡ (43c)
```
  | xdecor TEq init { [$1, Some $3] }
```

⟨*rule* Parser.init 51b⟩≡ (40d)
```
  init:
   | expr                  { $1 }
   | TOBrace ilist comma_opt TCBrace
       { match $2 with
         | [] -> raise (Impossible "grammar force at least one element")
         | (Left x)::xs ->
             mk_e (ArrayInit (x::(xs |> List.map (function
               | Left x -> x
               | Right _ -> error "mixing array and record initializer forbidden"
             )))) $1
         | (Right x)::xs ->
             mk_e (RecordInit (x::(xs |> List.map (function
               | Right x -> x
               | Left _ -> error "mixing array and record initializer forbidden"
             )))) $1
       }
```

⟨*other init related rules* 51c⟩≡ (40d)
```
  ⟨rule Parser.comma_opt 51e⟩
  ⟨rule Parser.ilist 51d⟩
  ⟨rule Parser.ilist2 51f⟩
  ⟨rule Parser.qual 51g⟩
```

⟨*rule* Parser.ilist 51d⟩≡ (51c)
```
  ilist:
   | init2              { [$1] }
   | ilist TComma init2 { $1 @ [$3] }
```

⟨*rule* Parser.comma_opt 51e⟩≡ (51c)
```
  comma_opt:
   | /*(*empty*)*/ { }
   | TComma { }
```

⟨*rule* Parser.ilist2 51f⟩≡ (51c)
```
  init2:
   | init          { Left (None, $1) }
   | qual TEq init { $1 $3 }
```

⟨*rule* Parser.qual 51g⟩≡ (51c)
```
  qual:
   | TOBra lexpr TCBra { (fun x -> Left (Some $2, x)) }
   | TDot tag          { (fun x -> Right (snd $2, x)) }
```

⟨*rule* Parser.lexpr 51h⟩≡ (47i)
```
  /*(*todo: (long expr) is wrapping expr in a (long) cast *)*/
  lexpr: expr { $1 }
```

6.10 Types and storage classes

⟨*rule* Parser.type_ 51i⟩≡ (40d)
```
  type_:
    | simple_type  { let (t, loc) = $1 in mk_t (Ast.TBase t) loc }
    | complex_type { $1 }
```

⟨*function* Parser.mk_t 52a⟩≡ (161a)
```
let mk_t t loc = { t = t; t_loc = loc }
```

⟨*rule* Parser.simple_type 52b⟩≡ (40d)
```
simple_type:
 ⟨Parser.simple_type cases 52d⟩
```

⟨*rule* Parser.complex_type 52c⟩≡ (40d)
```
complex_type:
 ⟨Parser.complex_type cases 53c⟩
```

6.10.1 Basic types

⟨Parser.simple_type *cases* 52d⟩≡ (52b)
```
| Tchar            { (T.I (T.Char, T.Signed), $1) }
/*(* meh, I should remove all Signed variants *)*/
| Tsigned Tchar    { (T.I (T.Char, T.Signed), $1) }
| Tunsigned Tchar  { (T.I (T.Char, T.Unsigned), $1) }

| Tshort           { (T.I (T.Short, T.Signed), $1) }
| Tunsigned Tshort { (T.I (T.Short, T.Unsigned), $1) }

| Tint             { (T.I (T.Int, T.Signed), $1) }
| Tunsigned Tint   { (T.I (T.Int, T.Unsigned), $1) }
/*(*bad: should be removed, but for compatibility with plan9 code I keep it*)*/
| Tunsigned        { (T.I (T.Int, T.Unsigned), $1) }

| Tlong            { (T.I (T.Long, T.Signed), $1) }
| Tunsigned Tlong  { (T.I (T.Long, T.Unsigned), $1) }

| Tlong Tlong      { (T.I (T.VLong, T.Signed), $1) }
| Tunsigned Tlong Tlong { (T.I (T.VLong, T.Unsigned), $1) }


| Tfloat  { (T.F (T.Float), $1) }
| Tdouble { (T.F (T.Double), $1) }

| Tvoid   { (T.Void, $1) }
/*(* less: allow more combinations, so better than just "syntax error"? *)*/
```

6.10.2 Storage classes

⟨*rule* Parser.storage 52e⟩≡ (40d)
```
/*(* less: allow some combinations? like extern register? *)*/
storage:
 | Tauto     { Storage.Local }
 | Tstatic   { Storage.Static }
 | Textern   { Storage.Extern }
 /*(* stricter: 5c just skips register declarations, I forbid them *)*/
 | Tregister { error "register not supported" }
 | Tinline   { error "inline not supported" }
/*(* less: allow more combinations, so better than just "syntax error"? *)*/
```

⟨*rule* Parser.storage_and_type 52f⟩≡ (40d)
```
/*(* stricter: I impose an order. c(lass) then g(arbage) then t(type). *)*/
storage_and_type:
 |          qualifiers type_ { Left None, $2 }
 | storage  qualifiers type_ { Left (Some $1), $3 }
 ⟨Parser.storage_and_type other cases 56c⟩
/*(* less: allow more combinations, so better than just "syntax error"? *)*/
```

6.10.3 Qualifiers

⟨*rule* `Parser.qualifier` 53a⟩≡ (40d)
```
qualifier:
 | Tconst    { T.Const }
 | Tvolatile { T.Volatile }
 | Trestrict { error "restrict not supported" }
```

⟨*other qualifier related rules* 53b⟩≡ (40d)
```
qualifier_and_type: qualifiers type_ { $2 }
/*(* less: allow storage here, so better than just "syntax error"? *)*/

qualifiers:
 | /*(*empty*)*/        { [] }
 | qualifiers qualifier { $1 @ [$2] }
```

6.10.4 Structures and unions

Structure/union uses

⟨`Parser.complex_type` *cases* 53c⟩≡ (52c) 53e▷
```
| su tag {
    let (su, loc) = $1 in
    let (_, id) = $2 in
    try
      let (_tagkind, bid) = Hashtbl.find env.tags id in
      (* assert takind = $1? let check.ml do this check *)
      let fullname = id, bid in
      mk_t (Ast.TStructName (su, fullname)) loc
    with Not_found ->
      (* will check in check.ml whether struct defined later *)
      let fullname = id, 0 (* less: or env.block? *) in
      mk_t (Ast.TStructName (su, fullname)) loc
}
```

⟨*rule* `Parser.su` 53d⟩≡ (53f)
```
su:
 | Tstruct { T.Struct, $1 }
 | Tunion  { T.Union, $1 }
```

Structure/union definitions

⟨`Parser.complex_type` *cases* 53e⟩+≡ (52c) ◁53c 54d▷
```
| su tag_opt sbody {
    let (su_kind, su_loc) = $1 in
    let id = $2 in
    let fullname = id, env.block in
    (* check if already defined in check.ml *)
    defs := (StructDef { su_name = fullname; su_loc; su_kind; su_flds = $3 })::!defs;
    add_tag env id (Ast.tagkind_of_su su_kind);
    mk_t (Ast.TStructName (su_kind, fullname)) loc
}
```

⟨*other struct related rules* 53f⟩≡ (40d)
```
⟨rule Parser.su 53d⟩
⟨rule Parser.tag_opt 54a⟩
```

⟨*rule* Parser.tag_opt 54a⟩≡ (53f)
```
tag_opt:
 | tag           { snd $1 }
 | /*(*empty*)*/ { gensym () }
```

⟨*rule* Parser.sbody 54b⟩≡ (40d)
```
/*(* note that a structure does not define a new scope *)*/
sbody: TOBrace edecl TCBrace { $2 }
```

⟨*other structure body related rules* 54c⟩≡ (40d)
```
edecl:
 |       edecl_elem TSemicolon { $1 }
 | edecl edecl_elem TSemicolon { $1 @ $2 }

edecl_elem:
 | qualifier_and_type edlist
   { $2 |> List.map (fun ((id, loc), typ2) ->
       (* note that this element can introduce a nested struct definition! *)
       let typ1 = $1 in
       let typ = typ2 typ1 in
       { fld_name = id; fld_loc = loc; fld_type = typ }
     )
   }
/*(* kenccext: c99ext? unnamed structure elt; used in u.h/regexp.h/bio.h/.. *)*/
 | qualifier_and_type
   { let s = gensym () in
    (* note that this anon elt can even be a nested anon struct definition! *)
     let typ = $1 in
     (* check that struct/union done later after typedef expansion *)
     [ { fld_name = s; fld_loc = typ.t_loc; fld_type = typ } ]
   }

/*(* todo: bitfield *)*/
edlist:
 | edecor               { [$1] }
 | edlist TComma edecor { $1 @ [$3] }

edecor: xdecor { $1 }
```

6.10.5 Enums

Enumeration type uses and definitions

⟨Parser.complex_type *cases* 54d⟩+≡ (52c) ◁53e 54e▷
```
| Tenum tag   {
    let (_, id) = $2 in
    try
      let (_tagkind, bid) = Hashtbl.find env.tags id in
      let fullname = id, bid in
      mk_t (Ast.TEnumName fullname) $1
    with Not_found ->
      let fullname = id, 0 (* less: or env.block? *) in
      mk_t (Ast.TEnumName fullname) $1
  }
```

⟨Parser.complex_type *cases* 54e⟩+≡ (52c) ◁54d 56b▷
```
| Tenum tag_opt TOBrace enum TCBrace {
    let id = $2 in
    let enum_name = id, env.block in
```

```
    defs := (EnumDef { enum_name; enum_loc = $1; enum_constants = $4 })::!defs;
    add_tag env id TagEnum;
    mk_t (Ast.TEnumName enum_name) $1
  }
```

Enumeration constant definitions

⟨*rule* `Parser.enum` 55a⟩≡ (40d)

```
  enum:
   | TName
       { let (loc, id) = $1 in
         add_id env id IdEnumConstant;
         [ { ecst_name = (id, env.block); ecst_loc = loc; ecst_value = None } ]
       }
   | TName TEq const_expr
       { let (loc, id) = $1 in
        (* note that const_expr can reference enum constants defined before *)
         add_id env id IdEnumConstant;
         [ { ecst_name = (id, env.block); ecst_loc = loc; ecst_value = Some $3 } ]
       }

   | enum TComma enum { $1 @ $3 }
   | enum TComma      { $1 }
```

⟨*rule* `Parser.const_expr` 55b⟩≡ (40d)

```
  const_expr: expr { $1 }
```

6.10.6 Pointer and array types

⟨`Parser.xdecor` *other cases* 55c⟩≡ (43d)

```
  | TMul qualifiers xdecor
      { let (id, f) = $3 in
        id, (fun x -> f (mk_t (TPointer x) $1))
      }
```

⟨`Parser.xdecor2` *other cases* 55d⟩≡ (44a) 55f▷

```
  | xdecor2 TOBra zexpr TCBra
      { let (id, f) = $1 in
        (* bug: not 'id, (fun x -> mk_t (TArray ($3, f x)) $2)' *)
        id, (fun x -> f (mk_t (TArray ($3, x)) $2))
      }
```

⟨*other expr ebnf rules* 55e⟩+≡ (40d) ◁50e

```
  zexpr:
   | /*(*empty*)*/ { None }
   | lexpr         { Some $1 }
```

6.10.7 Function types, parameter types

⟨`Parser.xdecor2` *other cases* 55f⟩+≡ (44a) ◁55d

```
  /*(* add parameters in scope in caller, when processing the function body *)*/
  | xdecor2 TOPar zparamlist TCPar
      { let (id, f) = $1 in
        (* bug: not 'id, (fun x -> mk_t (TFunction (f x, $3)) $2)' *)
        id, (fun x -> f (mk_t (TFunction (x, $3)) $2))
      }
```

⟨*rule* Parser.zparamlist 56a⟩≡ (40d)
```
zparamlist:
 | /*(*empty*)*/ { [], false }
 | paramlist     { $1 }
```

6.10.8 Typedefs

⟨Parser.complex_type *cases* 56b⟩+≡ (52c) ◁54e
```
 | TTypeName
     { let (loc, id) = $1 in
       try
         let (idkind, bid) = Hashtbl.find env.ids id in
         assert (idkind = IdTypedef);
         mk_t (Ast.TTypeName (id, bid)) loc
       with Not_found ->
         raise (Impossible (spf "could not find typedef for %s" id))
     }
```

⟨Parser.storage_and_type *other cases* 56c⟩≡ (52f)
```
 | Ttypedef qualifiers type_ { Right (), $3 }
```

6.11 Declarations and definitions, part two

6.11.1 Globals, external declarator

6.11.2 Function parameters

⟨*rule* Parser.paramlist 56d⟩≡ (40d)
```
paramlist:
 | qualifier_and_type xdecor
     { let ((id, p_loc), typ2) = $2 in
       (* the final blockid will be assigned when we create the scope of the
        * function body.
        *)
       [ { p_name = Some (id, -1); p_loc; p_type = typ2 $1 } ], false
     }
 | qualifier_and_type abdecor
     { [{ p_name = None; p_loc = $1.t_loc; p_type = $2 $1 }], false }

 | paramlist TComma paramlist
     { let (xs, isdot1) = $1 in
       let (ys, isdot2) = $3 in
       (* stricter: 5c does not report *)
       if isdot1
       then error "dots allowed only in last parameter position";

       xs @ ys, isdot2
     }
 | TDot TDot TDot { [], true }
```

6.11.3 Locals, automatic declarator

⟨*rule* Parser.adecl 56e⟩≡ (40d)
```
adecl:
 | storage_and_type        TSemicolon
     { if !defs = []
```

```
        then error "declaration without any identifier";

        (* stricter: *)
        error "move struct or typedef definitions to the toplevel";
        (* TODO? could allow at least structdef and just return [] here *)
      }

  | storage_and_type adlist TSemicolon
      { (* stricter: *)
        if !defs <> []
        then error "move struct or typedef definitions to the toplevel";

        (* less: could factorize code with xdecl *)
        ($2 |> List.map (fun (((id, v_loc), typ2), v_init) ->
           let (sto_or_typedef, typ1) = $1 in
           let v_type = typ2 typ1 in
           (match sto_or_typedef with
           | Left v_storage ->
               add_id env id IdIdent;
               mk_st (Var { v_name = (id, env.block);
                            v_loc; v_type; v_storage; v_init;}) loc
           (* stricter: *)
           | Right () -> error "typedefs not at the toplevel are forbidden"
           )
        ))
      }
```

⟨*rule* Parser.adlist 57a⟩≡ (40d)
```
adlist: xdlist { $1 }
```

6.11.4 Types, abstract declarator

⟨*rule* Parser.abdecor 57b⟩≡ (40d)
```
abdecor:
 | /*(*empty*)*/ { (fun x -> x) }
 | abdecor1      { $1 }
```

⟨*other abstract declarator related rules* 57c⟩≡ (40d)
```
abdecor1:
 | TMul qualifiers          { (fun x -> mk_t (TPointer x) $1) }
 | TMul qualifiers abdecor1 { (fun x -> $3 (mk_t (TPointer x) $1)) }
 | abdecor2  { $1 }

abdecor2:
 | abdecor3 { $1 }
 | abdecor2 TOPar zparamlist TCPar { (fun x -> $1 (mk_t (TFunction (x, $3))$2))}
 | abdecor2 TOBra zexpr TCBra      { (fun x -> $1 (mk_t (TArray ($3, x))$2))}

abdecor3:
 | TOPar TCPar          { (fun x -> mk_t (TFunction (x, ([], false))) $1)  }
 | TOBra zexpr TCBra    { (fun x -> mk_t (TArray ($2, x)) $1) }
 | TOPar abdecor1 TCPar { $2 }
```

Chapter 7

Checking

7.1 Overview

⟨*signature* Check.check_program 58a⟩≡ (151a)
```
(* can raise Error if failhard, otherwise print on stderr *)
val check_program: Ast.program -> unit
```

⟨*exception* Check.Error 58b⟩≡ (151)
```
exception Error of error
```

⟨*type* Check.error 58c⟩≡ (151)
```
type error =
  | Inconsistent of
      string * Location_cpp.loc * (* error here *)
      string * Location_cpp.loc   (* previous decl/def/whatever here *)
  | Misc of string * Location_cpp.loc
```

⟨*function* Check.error 58d⟩≡ (151b)
```
let error err =
  if !failhard
  then raise (Error err)
  else Logs.err (fun m -> m "%s" (string_of_error err))
```

⟨*signature* Check.failhard 58e⟩≡ (151a)
```
val failhard : bool ref
```

⟨*constant* Check.failhard 58f⟩≡ (151b)
```
let failhard = ref true
```

⟨*function* Check.check_program 58g⟩≡ (151b)
```
let check_program (prog : Ast.program) : unit =
  check_usedef prog
```

7.2 Printing warnings: `5c -w` and `5c -W`

⟨*global* Flags.warn 58h⟩≡ (159c)
```
let warn = ref false
```

⟨CLI.main() options *elements* 58i⟩+≡ (25d) ◁26a 59b▷
```
"-w", Arg.Set Flags.warn,
" enable warnings";
```

⟨*signature* Error.warn 58j⟩≡ (158c)
```
val warn: string -> Location_cpp.loc -> unit
```

⟨*global* `Flags.warnerror` 59a⟩≡ (159c)
```
let warnerror = ref false
```

⟨`CLI.main()` options *elements* 59b⟩+≡ (25d) ◁58i 120c▷
```
"-werror", Arg.Set Flags.warnerror,
" warnings generate error exceptions";
```

⟨*function* `Error.warn` 59c⟩≡ (158d)
```
let warn s loc =
  if !Flags.warn
  then
    if !Flags.warnerror
    then raise (Location_cpp.Error (spf "Warning: %s" s, loc))
    else
      let (file, line) = Location_cpp.final_loc_of_loc loc in
      Logs.warn (fun m -> m "%s:%d: %s" !!file line s)
```

7.3 Checking environment

⟨*type* `Check.usedef` 59d⟩≡ (151b)
```
type usedef = {
  mutable defined: Ast.loc option;
  mutable used: Ast.loc option;
}
```

⟨*type* `Check.env` 59e⟩≡ (151b)
```
type env = {
  ids:      (fullname, usedef * Ast.idkind) Hashtbl.t;
  tags:     (fullname, usedef * Ast.tagkind) Hashtbl.t;

  (* to reset after each function (because labels have a function scope) *)
  mutable labels:   (string, usedef) Hashtbl.t;

  (* block scope *)
  mutable local_ids: fullname list;

  (* todo: inbreakable: bool; incontinueable: bool *)
}
```

⟨`Check.check_usedef()` *set initial* `env` 59f⟩≡ (59g)
```
let env = {
  ids = Hashtbl.create 101;
  tags = Hashtbl.create 101;
  labels = Hashtbl.create 101;

  local_ids = [];
}
in
```

7.4 Checking a program

⟨*function* `Check.check_usedef` 59g⟩≡ (151b)
```
(* use of undefined, redefined, redeclared, unused, inconsistent tags, etc. *)
let check_usedef (program : Ast.program) : unit =

  ⟨function Check.check_usedef.toplevel 60a⟩
```

```
⟨function Check.check_usedef.stmt 61c⟩
⟨function Check.check_usedef.expr 63d⟩
⟨function Check.check_usedef.type_ 64c⟩

⟨Check.check_usedef() set initial env 59f⟩
fst program |> List.iter (toplevel env);

⟨Check.check_usedef() check used but not defined tags after program visit 67a⟩
(* less: could check unused static var decl? *)
()
```

7.4.1 Checking toplevel definitions

```
⟨function Check.check_usedef.toplevel 60a⟩≡ (59g)
  let rec toplevel env = function
    ⟨Check.check_usedef.toplevel() match cases 60b⟩
```

```
⟨Check.check_usedef.toplevel() match cases 60b⟩≡ (60a) 60c▷
  | StructDef { su_kind=su; su_name=fullname; su_loc=loc; su_flds=flds }->

    (* checking the tag *)
    let tagkind = Ast.tagkind_of_su su in
    check_inconsistent_or_redefined_tag env fullname tagkind loc;

    (* checking the fields *)
    let hflds = Hashtbl_.create () in
    flds |> List.iter
     (fun {fld_name = name; fld_loc = loc; fld_type = typ} ->
        (* stricter: 5c reports at use time, clang does immediately *)
        if Hashtbl.mem hflds name
        then error (Inconsistent (spf "duplicate member '%s'" name, loc,
                                  "previous declaration is here",
                                  Hashtbl.find hflds name));
        Hashtbl.add hflds name loc;
        type_ env typ
    )
```

```
⟨Check.check_usedef.toplevel() match cases 60c⟩+≡ (60a) ◁60b 60d▷
  | EnumDef { enum_name = fullname; enum_loc = loc; enum_constants = csts }->

    (* checking the tag *)
    let tagkind = TagEnum in
    check_inconsistent_or_redefined_tag env fullname tagkind loc;

    (* checking the constants *)
    csts |> List.iter
        (fun { ecst_name = fullname; ecst_loc = loc; ecst_value = eopt } ->
          check_inconsistent_or_redefined_id env fullname IdEnumConstant loc;
          eopt |> Option.iter (expr env)
        );
```

```
⟨Check.check_usedef.toplevel() match cases 60d⟩+≡ (60a) ◁60c 61a▷
  | TypeDef { typedef_name = fullname; typedef_loc = loc; typedef_type =typ}->
    check_inconsistent_or_redefined_id env fullname IdTypedef loc;
    type_ env typ
```

⟨Check.check_usedef.toplevel() *match cases* 61a⟩+≡ (60a) ◁60d 61b▷

```
  | VarDecl { v_name = fullname; v_loc; v_type = t; v_init = eopt; v_storage = _} ->

    (if Hashtbl.mem env.ids fullname &&
        snd (Hashtbl.find env.ids fullname) = IdIdent
    (* this can be ok, you can redeclare toplevel identifiers as you
     * can give a final storage. It depends on the situation,
     * see typecheck.ml
     *)
     then ()
     else check_inconsistent_or_redefined_id env fullname IdIdent v_loc
    );
    type_ env t;
    eopt |> Option.iter (expr env)
```

⟨Check.check_usedef.toplevel() *match cases* 61b⟩+≡ (60a) ◁61a

```
  (* todo: if use struct tags params, they must be complete at this point *)
  | FuncDef { f_name = name; f_loc = loc; f_type = ftyp; f_body = st; f_storage = _ } ->
    let fullname = name, 0 in

    (if Hashtbl.mem env.ids fullname &&
        snd (Hashtbl.find env.ids fullname) = IdIdent
    (* this can be ok, you can redeclare toplevel identifiers as you
     * can give a final storage. It depends on the situation,
     * see typecheck.ml
     *)
     then ()
     else check_inconsistent_or_redefined_id env fullname IdIdent loc
    );

    type_ env {t = TFunction ftyp; t_loc = loc };

    (* new function scope *)
    let env = { env with local_ids = []; labels = Hashtbl_.create () } in
    let (_tret, (tparams, _dots)) = ftyp in
    tparams |> List.iter (fun { p_name = fullnameopt; p_loc; p_type = _} ->
      fullnameopt |> Option.iter (fun fullname ->

        check_inconsistent_or_redefined_id env fullname IdIdent p_loc;
        env.local_ids <- fullname :: env.local_ids;
      );
    );

    (* We could match st to a Block and avoid new scope for it, but
     * it does not matter here. We do it correctly in parser.mly so
     * a parameter and local with the same name will have the same
     * blockid so we will detect if you redefine an entity even
     * if in different scope here.
     *)
    stmt env st;

    (* check function scope *)
    check_unused_locals env;
    check_labels env;
```

7.4.2 Checking statements

⟨*function* Check.check_usedef.stmt 61c⟩≡ (59g)

```
  and stmt env st0 =
```

```
    match st0.s with
    ⟨Check.check_usedef.stmt() match st0.s cases 62a⟩
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 62a⟩≡ (61c) 62b▷

```
  | ExprSt e -> expr env e
  | Block xs ->
      (* new block scope *)
      let env = { env with local_ids = [] } in
      List.iter (stmt env) xs;

      (* check block scope *)
      check_unused_locals env
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 62b⟩+≡ (61c) ◁62a 62c▷

```
  | Var { v_name = fullname; v_loc = loc; v_type = typ; v_init = eopt; v_storage = _ } ->
    (* less: before adding in environment? can have recursive use? *)
    eopt |> Option.iter (expr env);
    (* todo: if local VarDEcl, can actually have stuff nested like
     *  extern int i;  in which case we must go back to global
     *  scope for i! so rewrite AST? or just in typecheck.ml
     *  generate right storage for it.
     * can also be nested prototype (but I should forbid it
     *)
    check_inconsistent_or_redefined_id env fullname IdIdent loc;
    env.local_ids <- fullname :: env.local_ids;
    type_ env typ;
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 62c⟩+≡ (61c) ◁62b 62d▷

```
  | If (e, st1, st2) ->
      expr env e;
      stmt env st1;
      stmt env st2;
  (* ocaml-light: | While (e, st)  | Switch (e, st) | Case (e, st) *)
  | While (e, st) ->
      expr env e;
      stmt env st;
  | Switch (e, st) ->
      expr env e;
      stmt env st;
  | Case (e, st) ->
      expr env e;
      stmt env st;
  | DoWhile (st, e) ->
      stmt env st;
      expr env e
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 62d⟩+≡ (61c) ◁62c 63a▷

```
  | For (e1either, e2opt, e3opt, st) ->
      (* new block scope again *)
      let env = { env with local_ids = [] } in

      (match e1either with
      | Left e1opt -> e1opt |> Option.iter (expr env)
      | Right decls ->
          decls |> List.iter (fun decl ->
            stmt env ({s = Var decl; s_loc = decl.v_loc })
          )
      );
      e2opt |> Option.iter (expr env);
      e3opt |> Option.iter (expr env);
```

```
      stmt env st;

      (* check block scope *)
      check_unused_locals env
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 63a⟩+≡ (61c) ◁62d 63b▷

```
  | Return eopt -> eopt |> Option.iter (expr env)
  (* todo: check that inside something that be continue/break *)
  | Continue | Break -> ()
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 63b⟩+≡ (61c) ◁63a 63c▷

```
  (* checks on labels are done in FuncDef once we analyzed the whole body *)
  | Label (name, st) ->
      (try
         let usedef = Hashtbl.find env.labels name in
         usedef.defined |> Option.iter (fun locprev ->
           error (Inconsistent (spf "redefinition of label '%s'" name,
                                st0.s_loc,
                                "previous definition is here", locprev))
         );
         usedef.defined <- Some st0.s_loc;
       with Not_found ->
         Hashtbl.add env.labels name
           {defined = Some st0.s_loc; used = None}
      );
      stmt env st;

  | Goto name ->
      (try
         let usedef = Hashtbl.find env.labels name in
         usedef.used <- Some st0.s_loc
       with Not_found ->
         Hashtbl.add env.labels name { defined = None; used = Some st0.s_loc }
      )
```

⟨Check.check_usedef.stmt() *match* st0.s *cases* 63c⟩+≡ (61c) ◁63b

```
  | Default st -> stmt env st
```

7.4.3 Checking expressions

⟨*function* Check.check_usedef.expr 63d⟩≡ (59g)

```
  and expr env e0 =
    match e0.e with
    ⟨Check.check_usedef.expr() match e0.e cases 63e⟩
  and exprs env xs = xs |> List.iter (expr env)
```

⟨Check.check_usedef.expr() *match* e0.e *cases* 63e⟩≡ (63d) 63f▷

```
  | Int _ | Float _ | String _ -> ()
```

⟨Check.check_usedef.expr() *match* e0.e *cases* 63f⟩+≡ (63d) ◁63e 64b▷

```
  | Id fullname ->
      (try
         let (usedef, _idkind) = Hashtbl.find env.ids fullname in
         usedef.used <- Some e0.e_loc
       with Not_found ->
         (* todo: can be USED or SET *)
         raise (Impossible (spf "ids are always declared first: %s"
                              (unwrap fullname)))
      )
```

⟨*function* `Ast.unwrap` 64a⟩≡ (149)
```
let unwrap (name, _) = name
```

⟨`Check.check_usedef.expr()` *match* `e0.e` *cases* 64b⟩+≡ (63d) ◁63f
```
| Call (e, es) -> exprs env (e::es)
(* ocaml-light: | Assign (_, e1, e2) | Binary (e1, _, e2) | Sequence (e1, e2) | ArrayAccess (e1, e2) *)
| Assign (_, e1, e2) -> exprs env [e1; e2]
| Binary (e1, _, e2) -> exprs env [e1; e2]
| Sequence (e1, e2) -> exprs env [e1; e2]
| ArrayAccess (e1, e2) -> exprs env [e1; e2]
(* ocaml-light: | RecordAccess (e, _) | RecordPtAccess (e, _)
| Postfix (e, _) | Prefix (_, e) | Unary (_, e) *)
| RecordAccess (e, _) -> expr env e
| RecordPtAccess (e, _) -> expr env e
| Postfix (e, _) -> expr env e
| Prefix (_, e) -> expr env e
| Unary (_, e) -> expr env e
(* ocaml-light: | Cast (typ, e) | GccConstructor (typ, e) *)
| Cast (typ, e) ->
  type_ env typ;
  expr env e
| GccConstructor (typ, e) ->
  type_ env typ;
  expr env e
| CondExpr (e1, e2, e3) -> exprs env [e1; e2; e3]
| SizeOf either ->
    (match either with
    | Left e -> expr env e
    | Right t -> type_ env t
    )
| ArrayInit xs ->
  xs |> List.iter (fun (eopt, e) ->
    eopt |> Option.iter (expr env);
    expr env e
  )
| RecordInit xs -> xs |> List.iter (fun (_, e) -> expr env e)
```

7.4.4 Checking types

⟨*function* `Check.check_usedef.type_` 64c⟩≡ (59g)
```
and type_ env = fun typ ->
  match typ.t with
  ⟨Check.check_usedef.type_() match typ.t cases 64d⟩
in
```

⟨`Check.check_usedef.type_()` *match* `typ.t` *cases* 64d⟩≡ (64c) 65a▷
```
| TBase _ -> ()
| TPointer t -> type_ env t
| TArray (eopt, t) ->
  eopt |> Option.iter (expr env);
  type_ env t;
| TFunction (tret, (params, _dots)) ->
  type_ env tret;
  params |> List.iter (fun p ->
    (* nothing to do with p.p_name here *)
    type_ env p.p_type
  )
```

⟨Check.check_usedef.type_() *match* typ.t *cases* 65a⟩+≡ (64c) ◁64d 65b▷

```
  | TStructName (_, _) | TEnumName _ ->
    let tagkind, fullname =
      match typ.t with
      | TStructName (su, fullname) -> Ast.tagkind_of_su su, fullname
      | TEnumName fullname -> TagEnum, fullname
      | _ -> raise (Impossible "see pattern above")
    in
    (try
      let (usedef, oldtagkind) = Hashtbl.find env.tags fullname in
      if tagkind <> oldtagkind
      then inconsistent_tag fullname typ.t_loc usedef;
      usedef.used <- Some typ.t_loc;
    with Not_found ->
      (* forward decl *)
      Hashtbl.add env.tags fullname
        ({defined = None; used = Some typ.t_loc; }, tagkind)
    )
```

⟨Check.check_usedef.type_() *match* typ.t *cases* 65b⟩+≡ (64c) ◁65a

```
  | TTypeName fullname ->
      (try
         let (usedef, idkind) = Hashtbl.find env.ids fullname in
         if idkind <> IdTypedef
         then raise (Impossible "typename returned only if typedef in scope");
         usedef.used <- Some typ.t_loc
       with Not_found ->
         raise (Impossible "typename returned only if typedef in scope")
      )
```

7.5 Checks

7.5.1 Inconsistent or redefined tag

⟨*function* Check.check_inconsistent_or_redefined_tag 65c⟩≡ (151b)

```
  let check_inconsistent_or_redefined_tag env fullname tagkind loc =
    try
      let (usedef, oldtagkind) = Hashtbl.find env.tags fullname in
      if tagkind <> oldtagkind
      then inconsistent_tag fullname loc usedef;
      (* the tag may not have be defined, as in a previous 'struct Foo x;' *)
      usedef.defined |> Option.iter (fun locdef ->
        error (Inconsistent (spf "redefinition of '%s'" (unwrap fullname), loc,
                       "previous definition is here", locdef))
      );
      (* now it's defined *)
      usedef.defined <- Some loc;
    with Not_found ->
      Hashtbl.add env.tags fullname ({defined = Some loc; used = None;}, tagkind)
```

⟨*function* Check.inconsistent_tag 65d⟩≡ (151b)

```
  let inconsistent_tag fullname loc usedef =
    let locbefore =
      match usedef with
      (* ocaml-light: | { defined = Some loc; _ } | { used = Some loc; _ } *)
      | { defined = Some loc; used = _ } -> loc
      | { used = Some loc; defined = _ } -> loc
      | _ -> raise (Impossible "must have a def or a use")
```

```
    in
    error (Inconsistent (
      spf "use of '%s' with tag type that does not match previous declaration "
        (unwrap fullname), loc,
      "previous use is here", locbefore
    ))
```

7.5.2 Inconsistent or redefined identifier

⟨*function* Check.check_inconsistent_or_redefined_id 66a⟩≡ (151b)

```
  let check_inconsistent_or_redefined_id env fullname idkind loc =
    try
      let (usedef, oldidkind) = Hashtbl.find env.ids fullname in
      if idkind <> oldidkind
      then inconsistent_id fullname loc usedef
      else
        (match usedef.defined with
        | Some locdef ->
          (* stricter: 5c allows at least for same typedef; I do not. *)
          error (Inconsistent (spf "redefinition of '%s'" (unwrap fullname), loc,
                               "previous definition is here", locdef))
        (* the id must be defined, there is no forward use of ids
         * (enum constants, typedefs, variables)
         *)
        | None -> raise (Impossible "ids are always defined before being used")
        )
    with Not_found ->
      Hashtbl.add env.ids fullname ({defined = Some loc; used = None; }, idkind)
```

⟨*function* Check.inconsistent_id 66b⟩≡ (151b)

```
  let inconsistent_id fullname loc usedef =
    let locbefore =
      match usedef with
      | { defined = Some loc; used = _ } -> loc
      | { defined = None; used = _ } -> raise (Impossible "id always defined first")
    in
    error (Inconsistent (
      spf "redefinition of '%s' " (unwrap fullname), loc,
      "previous definition is here", locbefore
    ))
```

7.5.3 Unused variables

⟨*function* Check.check_unused_locals 66c⟩≡ (151b)

```
  let check_unused_locals (env : env) : unit =
    (* less: could also delete entries in env.ids *)
    env.local_ids |> List.iter (fun fullname ->
      let (usedef, idkind) = Hashtbl.find env.ids fullname in
      assert (idkind = IdIdent);
      match usedef with
      | { defined = Some loc; used = None } ->
          (* 5c says whether 'auto' or 'param' *)
          Error.warn
            (spf "variable declared and not used: '%s'" (unwrap fullname)) loc
      | { defined = Some _; used = Some _ } -> ()
      | { defined = None; used = _} ->
            raise (Impossible "locals are always defined")
    )
```

7.5.4 Used before set

7.5.5 Used but not defined tags

⟨Check.check_usedef() *check used but not defined tags after program visit* 67a⟩≡ (59g)
```
(* stricter: check if used but not defined tags (5c does not, clang does) *)
env.tags |> Hashtbl.iter (fun fullname (usedef, _idkind) ->
  match usedef with
  | { used = Some loc; defined = None } ->
    error (Misc (spf "use of tag '%s' that is never completed"
                     (unwrap fullname), loc))
  (* this can happen, header file are big and can cover multiple modules *)
  | { defined = Some _; used = None } -> ()
  (* perfect *)
  | { defined = Some _; used = Some _} -> ()
  | { defined = None; used = None } ->
      raise (Impossible "one or the other")
);
```

7.5.6 False positives silencing: SET()/USED()

⟨*toplevel* Parser._1 67b⟩≡
```
(* a few builtins.
 * less: 5c manages USED/SET at a lower level; they are lexical keywords
 * and they are in the AST with special nodes (OUSED/OSET)
 *)
let _ =
  add_id env "USED" IdIdent;
  add_id env "SET" IdIdent;
  ()
```

7.5.7 Use of undeclared labels or unused labels

⟨*function* Check.check_labels 67c⟩≡ (151b)
```
let check_labels env =
  env.labels |> Hashtbl.iter (fun name usedef ->
    match usedef with
    | { defined = Some _; used = Some _ } -> ()

    | { used = Some loc; defined = None } ->
        error (Misc (spf "use of undeclared label '%s'" name, loc))
    | { defined = Some loc; used = None } ->
        Error.warn (spf "label declared and not used '%s'" name) loc

    | { defined = None; used = None } ->
      raise (Impossible "at least one of used or defined")
  )
```

7.5.8 Unreachable code

7.5.9 Missing return

7.5.10 Unused expression

7.5.11 Constant if: `5c -c`

7.5.12 Out of range shifting

7.5.13 Useless comparisons

Chapter 8

Typechecking

8.1 Overview

⟨*signature* Typecheck.check_and_annotate_program 69a⟩≡ (162c)
```
(* Returns resolved type and storage information for identifiers and tags.
 * Annotate also with types each expression nodes in the returned functions
 * (so you can more easily generate code later).
 *
 * It also internally resolves enum constants and replaces them
 * with constants and evaluates some constant expressions (e.g., for
 * array size). It also does a few simple rewrites like +x => x + 0,
 * -x => 0 - x, add & before arrays in certain context, add some
 * explicit Cast operations, convert ArrayAccess in pointer arithmetic
 * operation, etc.
 *
 * can raise Error.
 *)
val check_and_annotate_program:
  Ast.program -> typed_program
```

⟨*exception* Typecheck.Error 69b⟩≡ (163b 162c)
```
exception Error of error
```

⟨*type* Typecheck.error 69c⟩≡ (163b 162c)
```
type error = Check.error
```

⟨*function* Typecheck.type_error 69d⟩≡ (163b)
```
let type_error _t loc =
  (* less: dump t? *)
  raise (Error (E.Misc ("incompatible type", loc)))
```

⟨*function* Typecheck.type_error2 69e⟩≡ (163b)
```
(* todo: for op xxx *)
let type_error2 t1 t2 loc =
  let s1 = Dumper_.s_of_any (FinalType t1) in
  let s2 = Dumper_.s_of_any (FinalType t2) in
  raise (Error (E.Misc (spf "incompatible types (%s and %s)" s1 s2, loc)))
```

⟨*function* Typecheck.check_and_annotate_program 69f⟩≡ (163b)
```
(* 5c: was called tcom *)
let check_and_annotate_program (prog: Ast.program) : typed_program =
  let (ast, _locs) = prog in

  let funcs = ref [] in
```

```
  ⟨function Typecheck.check_and_annotate_program.toplevel 84d⟩
  ⟨Typecheck.check_and_annotate_program() set initial env 70d⟩

  ast |> List.iter (toplevel env);

  { ids = env.ids_; structs = env.structs_; funcs = List.rev !funcs }
```

8.2 Typechecking environment

⟨type Typecheck.env 70a⟩≡ (163b)

```
(* Environment for typechecking *)
type env = {

  (* those 2 fields will be returned ultimately by check_and_annotate_program *)
  ids_:  (Ast.fullname, idinfo) Hashtbl.t;
  structs_: (Ast.fullname, Type.struct_kind * Type.structdef) Hashtbl.t;

  (* internal *)
  typedefs: (Ast.fullname, Type.t) Hashtbl.t;
  (* stricter: no float enum *)
  enums: (fullname, Type.integer_type) Hashtbl.t;

  (* stricter: no support for float enum constants either *)
  constants: (Ast.fullname, integer * Type.integer_type) Hashtbl.t;

  (* return type of function; used to typecheck Return *)
  return_type: Type.t;

  ⟨Typecheck.env other fields 70b⟩
}
```

⟨Typecheck.env other fields 70b⟩≡ (70a)

```
(* used to add some implicit GetRef for arrays and functions *)
expr_context: expr_context;
```

⟨type Typecheck.expr_context 70c⟩≡ (163b)

```
and expr_context =
  | CtxWantValue
  ⟨Typecheck.expr_context other cases 78a⟩
```

⟨Typecheck.check_and_annotate_program() set initial env 70d⟩≡ (69f)

```
let env = {
  ids_ = Hashtbl_.create ();
  structs_ = Hashtbl_.create ();

  typedefs = Hashtbl_.create ();
  enums = Hashtbl_.create ();
  constants = Hashtbl_.create ();

  return_type = T.Void;
  expr_context = CtxWantValue;
}
in
```

8.3 Typechecking helpers

8.3.1 Type equality

⟨*function* Typecheck.same_types 71a⟩≡ (163b)
```
(* if you declare multiple times the same global, we need to make sure
 * the types are the same. ex: 'extern int foo; ... int foo = 1;'
 * This is where we detect inconsistencies like 'int foo; void foo();'.
 *
 * Because we expand typedefs before calling same_types, and because
 * we do struct equality by name not fields, testing the equality of
 * two types is simple.
 *)
let same_types (t1 : Type.t) (t2 : Type.t) : bool =
  match t1, t2 with

  (* 'void*' can match any pointer! The generic trick of C
   * (but only when the pointer is at the top of the type).
   *)
  | T.Pointer T.Void, T.Pointer _      -> true
  | T.Pointer _,      T.Pointer T.Void -> true

  (* stricter: struct equality by name, not by fields *)
  | _ -> t1 =*= t2
```

8.3.2 Type compatibility

8.3.3 Type merge

⟨*function* Typecheck.merge_types 71b⟩≡ (163b)
```
(* if you declare multiple times the same global, we must merge types. *)
let merge_types t1 _t2 =
  t1
  (* TODO? what is doing 5c? *)
```

8.3.4 generic `void*` pointer conversions: `5c -V`

8.3.5 Compatibility policies

8.3.6 Array to pointer conversions

⟨*function* Typecheck.array_to_pointer 71c⟩≡ (163b)
```
(* When you mention an array in a context where you want to access the array
 * content, we prefix the array with a '&' and change its type from a T.Array
 * to a T.Pointer. This allows in turn to write typechecking rules
 * mentioning only Pointer (see for example check_compatible_binary).
 *)
let array_to_pointer (env : env) (e : expr) : expr =
  match e.e_type with
  | T.Array (_, t) ->

    (match env.expr_context with
    | CtxWantValue ->
      if not (lvalue (e))
      then raise (Error (E.Misc ("not an l-value", e.e_loc)));

      { e = Unary (GetRef, e); e_type = T.Pointer t; e_loc = e.e_loc }
```

```
    ⟨array_to_pointer() when Array case, match context cases 78b⟩
    )

  (* stricter? do something for Function? or force to put address? *)
  | T.Func _ ->
    (match env.expr_context with
    | CtxWantValue ->
      Error.warn "you should get the address of the function" e.e_loc;
      e
    | _ -> e
    )

  | _ -> e
```

8.4 Expressions

```
⟨function Typecheck.expr 72a⟩≡                                    (163b)
  let rec expr (env : env) (e0 : expr) : expr (* but type annotated *) =
    (* default env for recursive call *)
    let newenv = { env with expr_context = CtxWantValue } in

    match e0.e with
    ⟨Typecheck.expr() match e0.e cases 72c⟩

    | ArrayInit _
    | RecordInit _
    | GccConstructor _
        -> raise Todo
```

```
⟨function Typecheck.expropt 72b⟩≡                                 (163b)
  and expropt (env : env) (eopt : expr option) : expr option =
      match eopt with
      | None -> None
      | Some e -> Some (expr env e)
```

```
⟨Typecheck.expr() match e0.e cases 72c⟩≡                  (72a) 72d▷
  | Sequence (e1, e2) ->
    let e1 = expr newenv e1 in
    let e2 = expr newenv e2 in
    { e0 with e = Sequence (e1, e2); e_type = e2.e_type }
```

8.4.1 Numeric constants

```
⟨Typecheck.expr() match e0.e cases 72d⟩+≡          (72a) ◁72c 72e▷
  | Int    (_s, inttype)   -> { e0 with e_type = T.I inttype }
```

```
⟨Typecheck.expr() match e0.e cases 72e⟩+≡          (72a) ◁72d 72f▷
  | Float  (_s, floattype) -> { e0 with e_type = T.F floattype }
```

8.4.2 String constants

```
⟨Typecheck.expr() match e0.e cases 72f⟩+≡          (72a) ◁72e 73a▷
  (* less: transform in Id later? *)
  | String (_s, t)         -> { e0 with e_type = t } |> array_to_pointer env
```

8.4.3 Entity uses

⟨Typecheck.expr() *match* e0.e *cases* 73a⟩+≡ (72a) ◁72f 73b▷

```
  | Id fullname ->
     if Hashtbl.mem env.constants fullname
     then
       let (i, inttype) = Hashtbl.find env.constants fullname in
       { e0 with e = Int (spf "%d" i, inttype); e_type = T.I inttype }
     else
       let idinfo = Hashtbl.find env.ids_ fullname in
       { e0 with e_type = idinfo.typ } |> array_to_pointer env
```

8.4.4 Arithmetic and boolean expressions

⟨Typecheck.expr() *match* e0.e *cases* 73b⟩+≡ (72a) ◁73a 73c▷

```
  | Binary (e1, op, e2) ->
    let e1 = expr newenv e1 in
    let e2 = expr newenv e2 in

    check_compatible_binary op e1.e_type e2.e_type e0.e_loc;

    (* todo: add casts if left and right not the same types? or do it later? *)
    let finalt : Type.t =
      match op with

      | Arith Minus ->
        (match e1.e_type, e2.e_type with
        | T.Pointer _, T.Pointer _ -> T.long (* TODO? depend on Arch.t? *)
        | _ -> result_type_binary e1.e_type e2.e_type
        )

      | Arith (Plus | Mul | Div | Mod
               | And | Or | Xor
               (* todo: also add T.int cast when shl/shr on right operand *)
               | ShiftLeft | ShiftRight
               ) ->
        result_type_binary e1.e_type e2.e_type

      | Logical (Eq | NotEq
                 | Inf | Sup | InfEq | SupEq
                 | AndLog | OrLog
                 ) ->
        (* ugly: should be a T.Bool! C is ugly. *)
        T.int
    in
    (* TODO: add int cast for ShiftLeft, ShiftRight right operand
     * TODO: call arith() to do "usual arithmetic conversions"? or do that
     * alt: later in Codegen? on in Rewrite.ml?
     *)

    { e0 with e = Binary (e1, op, e2); e_type = finalt }
```

⟨Typecheck.expr() *match* e0.e *cases* 73c⟩+≡ (72a) ◁73b 76a▷

```
  | Unary (op, e) ->
    (match op with
    (* + E -~> 0 + E *)
    | UnPlus ->
      let e = Binary ({e0 with e = Int ("0",(T.Int,T.Signed))}, Arith Plus,e) in
```

```
      expr env { e0 with e = e }
    (* - E -~> 0 - E *)
    | UnMinus ->
      let e = Binary ({e0 with e = Int ("0",(T.Int,T.Signed))}, Arith Minus,e)in
      expr env { e0 with e = e }
    (* ~ E -~> -1 ^ E *)
    | Tilde ->
      let e = Binary ({e0 with e = Int ("-1",(T.Int,T.Signed))}, Arith Xor,e) in
      expr env { e0 with e = e }

    | Not ->
      let e = expr newenv e in
      (match e.e_type with
      (* what about T.Array? see array_to_pointer above *)
      | T.I _ | T.F _ | T.Pointer _ -> ()
      | _ -> type_error e.e_type e.e_loc
      );
      { e0 with e = Unary (Not, e); e_type = T.int }
     ⟨Typecheck.expr() in Unary, match op other cases 77b⟩
     )
```

⟨*function* Typecheck.check_compatible_binary 74⟩≡ (163b)

```
  (* when you apply an operation between two expressions, this
   * expression can be valid even if the types of those two expressions
   * are not the same. However, they must be "compatible".
   * The compatibility policy depends on the operation of the expression.
   *
   * less: better error messages, for instance when want to add 2 pointers.
   *)
  let check_compatible_binary op (t1 : Type.t) (t2 : Type.t) loc : unit =
    match op with
    | Arith Plus ->
      (match t1, t2 with
      | (T.I _ | T.F _), (T.I _ | T.F _)
      | T.Pointer _, T.I _
      | T.I _, T.Pointer _
        -> ()
      (* you can not add 2 pointers *)
      | T.Pointer _, T.Pointer _
      | _ -> type_error2 t1 t2 loc
      )
    | Arith Minus ->
      (match t1, t2 with
      | (T.I _ | T.F _), (T.I _ | T.F _)
      (* you can not sub a pointer to an int (but can sub an int to a pointer) *)
      | T.Pointer _, T.I _
        -> ()
      (* you can sub 2 pointers (if they have the same types, and the
       * result is a long). same_types() will allow a void* to match any pointer.
       *)
      | T.Pointer _, T.Pointer _ when same_types t1 t2 -> ()
      | _ -> type_error2 t1 t2 loc
      )
    | Arith (Mul | Div) ->
      (match t1, t2 with
      | (T.I _ | T.F _), (T.I _ | T.F _) -> ()
      | _ -> type_error2 t1 t2 loc
      )
    | Arith (Mod   | And | Or | Xor  | ShiftLeft | ShiftRight ) ->
      (match t1, t2 with
```

```
    | (T.I _), (T.I _ )
      -> ()
    (* stricter: I do not allow T.Int _ with T.F _, 5c does (clang does not) *)
    | _ -> type_error2 t1 t2 loc
    )

  | Logical (Eq | NotEq  | Inf | Sup | InfEq | SupEq) ->
    (match t1, t2 with
    | (T.I _ | T.F _), (T.I _ | T.F _) -> ()
    | T.Pointer _, T.Pointer _ when same_types t1 t2 -> ()
    (* you can not compare two structures! no deep equality (nor arrays) *)
    | _ -> type_error2 t1 t2 loc
    )
  | Logical (AndLog | OrLog) ->
    (* stricter? should impose Bool! *)
    (match t1 with
    | (T.I _ | T.F _ | T.Pointer _) ->
      (match t2 with
      | (T.I _ | T.F _ | T.Pointer _) -> ()
      | _ -> type_error t2 loc
      )
    | _ -> type_error t1 loc
    )
```

8.4.5 Arithmetic conversions

⟨*function* Typecheck.result_type_binary 75⟩≡ (163b)

```
let result_type_binary (t1 : Type.t) (t2 : Type.t) : Type.t =
  match t1, t2 with
  | T.I (T.Char, T.Signed), (T.I _ | T.F _ | T.Pointer _) -> t2

  | T.I (T.Char, T.Unsigned), T.I (x, _) -> T.I (x, T.Unsigned)
  | T.I (T.Char, T.Unsigned), (T.F _ | T.Pointer _) -> t2

  | T.I (T.Short, T.Signed), (T.I ((T.Char|T.Short), sign)) ->
      T.I (T.Short, sign)
  | T.I (T.Short, T.Signed), (T.I _ | T.F _ | T.Pointer _) -> t2

  | T.I (T.Short, T.Unsigned), (T.I ((T.Char|T.Short), _)) ->
      T.I (T.Short, T.Unsigned)
  | T.I (T.Short, T.Unsigned), T.I (x, _) -> T.I (x, T.Unsigned)
  | T.I (T.Short, T.Unsigned), (T.F _ | T.Pointer _) -> t2

  | T.I (T.Int, T.Signed), (T.I ((T.Char|T.Short|T.Int), sign)) ->
      T.I (T.Int, sign)
  | T.I (T.Int, T.Signed), (T.I _ | T.F _ | T.Pointer _) -> t2

  | T.I (T.Int, T.Unsigned), (T.I ((T.Char|T.Short|T.Int), _)) ->
      T.I (T.Int, T.Unsigned)
  | T.I (T.Int, T.Unsigned), T.I (x, _) -> T.I (x, T.Unsigned)
  | T.I (T.Int, T.Unsigned), (T.F _ | T.Pointer _) -> t2

  | T.I (T.Long, T.Signed), (T.I ((T.Char|T.Short|T.Int|T.Long), sign))->
      T.I (T.Long, sign)
  | T.I (T.Long, T.Signed), (T.I _ | T.F _ | T.Pointer _) -> t2

  | T.I (T.Long, T.Unsigned), (T.I ((T.Char|T.Short|T.Int|T.Long), _)) ->
      T.I (T.Long, T.Unsigned)
  | T.I (T.Long, T.Unsigned), T.I (x, _) -> T.I (x, T.Unsigned)
```

```
    | T.I (T.Long, T.Unsigned), (T.F _ | T.Pointer _) -> t2

    | T.I (T.VLong, T.Signed),(T.I ((T.Char|T.Short|T.Int|T.Long|T.VLong),sign))->
        T.I (T.VLong, sign)
    | T.I (T.VLong, T.Signed), (T.F _ | T.Pointer _) -> t2

    | T.F T.Float, (T.I _ | T.F T.Float) -> T.F T.Float
    | T.F T.Float, T.F T.Double -> T.F T.Double
    | T.F T.Float, T.Pointer _ -> t2

    | T.F T.Double, (T.I _ | T.F _) -> T.F T.Double
    | T.F T.Double, T.Pointer _ -> t2

    | T.Pointer _, (T.I _ | T.F _) -> t1

    (* see same_types special handling of void* pointers *)
    | T.Pointer T.Void, T.Pointer _ -> t2
    | T.Pointer _, T.Pointer T.Void -> t1

    | T.Pointer _, T.Pointer _ ->
      assert (t1 =*= t2);
      t1

    | _ -> raise (Impossible "case should be forbidden by compatibility policy")
```

8.4.6 Pointer arithmetic

8.4.7 Assignments

⟨Typecheck.expr() *match* e0.e *cases* 76a⟩+≡ (72a) ◁73c 78d▷

```
  | Assign (op, e1, e2) ->
    let e1 = expr newenv e1 in
    let e2 = expr newenv e2 in

    if not (lvalue (e1))
    then raise (Error (E.Misc ("not an l-value", e0.e_loc)));

    check_compatible_assign op e1.e_type e2.e_type e0.e_loc;

    (* todo: add cast on e2 if not same type,
     * todo: mixedasop thing?
     *)
    { e0 with e = Assign (op, e1, e2); e_type = e1.e_type }
```

⟨*function* Typecheck.lvalue 76b⟩≡ (163b)

```
  (* we assume the typechecker has called expr() on 'e0' before,
   * so Id of enum constants for example has been substituted to Int
   * and so are not considered an lvalue.
   * 5c: was cached in an 'addable' field
   *)
  let lvalue (e0 : expr) : bool =
    match e0.e with
    | Id _
    | Unary (DeRef, _)
    (* todo: lvalue only if leftpart is a lvalue. But when it can not be
     * a lvalue? if bitfield?
     *)
    | RecordAccess _
      -> true
```

```
    (* Strings are transformed at some point in Id.
     * We must consider them as an lvalue, because an Id is an lvalue
     * and because if a string is passed as an argument to a function, we want
     * to pass the address of this string (see array_to_pointer()).
     *)
    | String _ ->
      (* raise (Impossible "transformed before") *)
      true

    | Int _ | Float _
    | Binary _
    | Unary ((GetRef | UnPlus |  UnMinus | Tilde | Not), _)
      -> false
    | ArrayAccess _ | RecordPtAccess _ -> raise (Impossible "transformed before")
    (* TODO? what remains? should be just false no? *)
    | _ -> raise Todo
```

⟨*function* `Typecheck.check_compatible_assign` 77a⟩≡ (163b)

```
  (* less: could run typ_ext hooks here? and return a new node? for
   * unnamed_inheritance.c?
   *)
  let check_compatible_assign op (t1 : Type.t) (t2 : Type.t) loc : unit =
    match op with
    | Eq_ ->
      (match t1, t2 with
      | (T.I _ | T.F _), (T.I _ | T.F _) -> ()
      (* 'void*' special handling done in same_types() *)
      | T.Pointer _, T.Pointer _ when same_types t1 t2 -> ()
      | T.StructName (su1, name1), T.StructName (su2, name2)
        when su1 =*= su2 && name1 =*= name2 -> ()
      | _ -> type_error2 t1 t2 loc
      )
    (* not exactly the same rule than in check_compatible_binary *)
    | OpAssign op ->
      (match op with
      | (Plus | Minus) ->
        (match t1, t2 with
        | (T.I _ | T.F _), (T.I _ | T.F _) -> ()
        (* you can not x += y or x-=y when both x and y are pointers
         * even though you can do x - y because the result type is a long
         * (and so you can not assign than back into x).
         *)
        | T.Pointer _, T.I _ -> ()
        | _ -> type_error2 t1 t2 loc
        )
      | (Mul | Div)
      | (Mod   | And | Or | Xor  | ShiftLeft | ShiftRight )
        -> check_compatible_binary (Arith op) t1 t2 loc
      )
```

8.4.8 Pointers

⟨`Typecheck.expr()` *in* `Unary`, *match* `op` *other cases* 77b⟩≡ (73c) 78c▷

```
  | GetRef ->

      (* we dont want an additional '&' added before an array *)
      let e = expr { env with expr_context = CtxGetRef } e in
```

```
      if not (lvalue (e))
      then raise (Error (E.Misc ("not an l-value", e0.e_loc)));

      (* less: warn if take address of array or function, ADDROP *)
      { e0 with e = Unary (GetRef, e);
                e_type = T.Pointer (e.e_type) }
```

⟨Typecheck.expr_context *other cases* 78a⟩≡ (70c) 82b▷

```
  | CtxGetRef
```

⟨array_to_pointer() *when* Array *case, match context cases* 78b⟩≡ (71c) 82c▷

```
  | CtxGetRef ->
    Error.warn "address of array ignored" e.e_loc;
    e
```

⟨Typecheck.expr() *in* Unary, *match* op *other cases* 78c⟩+≡ (73c) ◁77b

```
  | DeRef ->
      let e = expr newenv e in

      (match e.e_type with
      | T.Pointer t ->
        { e0 with e = Unary (DeRef, e);
                  e_type = t } |> array_to_pointer env
      (* what about T.Array? no need, see array_to_pointer() *)
      | _ -> type_error e.e_type e.e_loc
      )
```

8.4.9 Array accesses

⟨Typecheck.expr() *match* e0.e *cases* 78d⟩+≡ (72a) ◁76a 78e▷

```
  (* x[y] --> *(x+y), pointer arithmetic power *)
  | ArrayAccess (e1, e2) ->
    let e = Unary (DeRef, { e0 with e = Binary (e1, Arith Plus, e2) }) in
    expr env { e0 with e = e }
```

8.4.10 Field accesses

⟨Typecheck.expr() *match* e0.e *cases* 78e⟩+≡ (72a) ◁78d 79a▷

```
  | RecordAccess (e, name) ->
    let e = expr newenv e in

    (match e.e_type with
    | T.StructName (_su, fullname) ->
      let (_su2, def) = Hashtbl.find env.structs_ fullname in
      (try
         let t = List.assoc name def in
         { e0 with e = RecordAccess (e, name);
                   e_type = t } |> array_to_pointer env
       with Not_found ->
         ⟨Typecheck.expr() when field name not found and gensymed field exn 79b⟩
         else
           raise (Error(E.Misc(spf "not a member of struct/union: %s" name,
                               e.e_loc)))
      )
    | _ -> type_error e.e_type e.e_loc
    )
```

⟨Typecheck.expr() *match* e0.e *cases* 79a⟩+≡ (72a) ◁78e 80a▷

```
  (* x->y --> ( *x).y *)
  | RecordPtAccess (e, name) ->
    let e = RecordAccess ({ e0 with e = Unary (DeRef, e)}, name) in
    expr env { e0 with e = e }
```

⟨Typecheck.expr() *when field name not found and gensymed field exn* 79b⟩≡ (78e)

```
  if def |> List.exists (fun (fld, _) -> Ast.is_gensymed fld)
  then
    try
      unsugar_anon_structure_element env e0 e name def
        |>array_to_pointer env
    with Not_found ->
      raise (Error(E.Misc(spf "not a member of struct/union: %s" name,
                          e.e_loc)))
```

⟨*function* Typecheck.unsugar_anon_structure_element 79c⟩≡ (163b)

```
  (* X.foo --> X.|sym42|.foo *)
  let rec unsugar_anon_structure_element (env : env) e0 e name def =

    let res = ref [] in

    def |> List.iter (fun (fldname, t) ->
      if fldname = name
      then res |> Stack_.push { e0 with e = RecordAccess (e, name); e_type = t }
      else
        if Ast.is_gensymed fldname
        then
          (match t with
          | T.StructName (_su, fullname) ->
            let (_su2, def) = Hashtbl.find env.structs_ fullname in
            (try
               let e =
                 unsugar_anon_structure_element env e0
                   ({ e with e = RecordAccess (e, fldname); e_type = t })
                   name def
               in
               res |> Stack_.push e
             with Not_found -> ()
            )
          | _ -> raise (Impossible "checked anon elements are struct/union")
          )
        else ()
    );
    (match !res with
    | [x] -> x
    | [] -> raise Not_found
    | _x::_y::_xs ->
      raise (Error(E.Misc(spf "ambiguous unnamed structure element %s" name,
                                e.e_loc)))
    )
```

⟨*function* Ast.is_gensymed 79d⟩≡ (149)

```
  (* see also Parser.gensym *)
  let is_gensymed str =
    str =~ "|sym[0-9]+|.*"
```

8.4.11 Function calls

⟨Typecheck.expr() *match* e0.e *cases* 80a⟩+≡ (72a) ◁79a 81a▷

```
  | Call (e, es) ->
    (* less: should disable implicit OADDR for function here in env *)
    let e = expr newenv e in

    (match e.e_type with
    | T.Func (tret, tparams, varargs) ->
       (* we enable GetRef for array here (and functions)
        * TODO? why not use newenv?
        *)
      let es = List.map (expr { env with expr_context = CtxWantValue }) es in

      check_args_vs_params es tparams varargs e0.e_loc;

      (* todo: add cast *)
      (* less: format checking *)
      { e0 with e = Call (e, es);
                e_type = tret }

    | T.Pointer (T.Func (_tret, _tparams, _varargs)) ->
      (* stricter?: we could forbid it, but annoying for my print in libc.h *)
      let e = { e with e = Unary (DeRef, e); } in
      expr newenv { e0 with e = Call (e, es) }

    | _ -> type_error e.e_type e.e_loc
    )
```

⟨*function* Typecheck.check_args_vs_params 80b⟩≡ (163b)

```
  (* 5c: was called tcoma() *)
  let rec check_args_vs_params (es : expr list) tparams (varargs : bool) loc =
    match es, tparams, varargs with

    (* stricter? confusing to have foo() and foo(void) *)
    | [], ([] | [T.Void]), _ -> ()

    | [], _, _ ->
      raise (Error (E.Misc ("not enough function arguments", loc)))

    | _e::_es, [], false ->
      raise (Error (E.Misc ("too many function arguments", loc)))

    | e::es, [], true ->
      (match e.e_type with
      (* ??? *)
      | T.I _ | T.F _ | T.Pointer _ | T.StructName _ -> ()
      (* TODO: enumerate possible remaining, and why type_error? *)
      | _ -> type_error e.e_type loc
      );
      check_args_vs_params es [] true loc

    | e::es, t::ts, _ ->
      (match e.e_type with
      | T.I _ | T.F _ | T.Pointer _ | T.StructName _ -> ()
      (* TODO: enumerate possible remaining, and why type_error? *)
      | _ -> type_error e.e_type loc
      );
      (try
         (* todo: convert to int small types? see tcoma *)
```

```
        check_compatible_assign Eq_ t e.e_type e.e_loc
      with Error _ ->
        (* TODO? actual error message of 5c? *)
        raise (Error (E.Misc ("argument prototype mismatch", e.e_loc)))
    );
    check_args_vs_params es ts varargs loc
```

8.4.12 Cast

⟨Typecheck.expr() *match* e0.e *cases* 81a⟩+≡ (72a) ◁80a 81b▷

```
  | Cast (typ, e) ->
    (* todo? set special env ADDROF|CASTOF? CASTOF seemed unused *)

    let t = type_ env typ in
    let e = expr newenv e in

    (match e.e_type, t with
    | T.I _, (T.I _ | T.F _ | T.Pointer _ | T.Void)
    | T.F _, (T.I _ | T.F _ | T.Void)
    | T.Pointer _, (T.I _ | T.Pointer _ | T.Void)
      -> ()
    (* less: seems pretty useless *)
    | T.Void, T.Void -> ()
    (* less: seems pretty useless *)
    | T.StructName (su1, _) , T.StructName (su2, _) when su1 =*= su2 -> ()
    | T.StructName _, T.Void -> ()
    | _ -> type_error2 e.e_type t e0.e_loc
    );
    { e0 with e = Cast (typ, e);
              e_type = t } (* |> array_to_pointer ? *)
```

8.4.13 Ternary expressions

⟨Typecheck.expr() *match* e0.e *cases* 81b⟩+≡ (72a) ◁81a 81c▷

```
  | CondExpr (e1, e2, e3) ->
    let e1 = expr newenv e1 in
    let e2 = expr newenv e2 in
    let e3 = expr newenv e3 in

    (* stricter? should enforce e1.e_type is a Bool *)
    check_compatible_binary (Logical Eq) e2.e_type e3.e_type e0.e_loc;

    (* todo: special nil handling? need? *)
    let finalt = result_type_binary e2.e_type e3.e_type in

    (* todo: add cast, and special nil handling *)
    { e0 with e = CondExpr (e1, e2, e3);
              e_type = finalt }
```

8.4.14 Prefix/postfix

⟨Typecheck.expr() *match* e0.e *cases* 81c⟩+≡ (72a) ◁81b 82a▷

```
  (* ocaml-light: | Postfix (e, op) | Prefix (op, e) *)
  | Postfix (_, _) | Prefix (_, _) ->
    let (e, op) =
        match e0.e with
        | Postfix (e, op) -> e, op
```

```
        | Prefix (op, e) -> e, op
        | _ -> raise (Impossible "pattern match only those cases")
    in
    let e = expr newenv e in

    if not (lvalue (e))
    then raise (Error (E.Misc ("not an l-value", e.e_loc)));

    check_compatible_binary (Arith Plus) e.e_type T.int e0.e_loc;

    (match e.e_type with
    | T.Pointer T.Void ->
      raise (Error (E.Misc ("inc/dec of a void pointer", e.e_loc)));
    | _ -> ()
    );

    { e0 with e =
        (match e0.e with
        | Postfix _ -> Postfix (e, op)
        | Prefix _ -> Prefix (op, e)
        | _ -> raise (Impossible "pattern match only those cases")
        );
        e_type = e.e_type }
```

8.4.15 sizeof()

⟨Typecheck.expr() *match* e0.e *cases* 82a⟩+≡ (72a) ◁81c

```
  | SizeOf(te) ->

    (match te with
    | Left e ->
      (* we pass a special context because if t2 mentions an array,
       * we want the size of the array, not the size of a pointer to an array
       *)
      let e = expr { env with expr_context = CtxSizeof  } e in
      { e0 with e = SizeOf (Left e);
                e_type = T.int }

    (* todo: build a fake expression but with the right expanded type
     * so the codegen later does not have to redo the job of expanding
     * typedefs.
     *)
    | Right typ ->
      { e0 with e = SizeOf (Right typ);
                e_type = T.int }
    )
```

⟨Typecheck.expr_context *other cases* 82b⟩+≡ (70c) ◁78a

```
  | CtxSizeof
```

⟨array_to_pointer() *when* Array *case, match context cases* 82c⟩+≡ (71c) ◁78b

```
  | CtxSizeof -> e
```

8.5 Statements

⟨*function* Typecheck.stmt 82d⟩≡ (163b)

```
  (* The code below is boilerplate, mostly.
```

```
   * expr() should not do any side effect on the environment, so we can
   * call recursively in any order stmt() and expr() (including the
   * reverse order of evaluation of OCaml for arguments).
   *)
  let rec stmt (env : env) (st0 : stmt) : stmt (* with exprs inside annotated *) =
    { st0 with s =
      (match st0.s with
      ⟨Typecheck.stmt() match st0.s cases 83a⟩
      )
    }
```

⟨Typecheck.stmt() *match* st0.s *cases* 83a⟩≡ (82d) 83b▷

```
  | ExprSt e -> ExprSt (expr env e)
  | Block xs -> Block (List.map (stmt env) xs)
```

⟨Typecheck.stmt() *match* st0.s *cases* 83b⟩+≡ (82d) ◁83a 83c▷

```
  (* stricter? should require Bool, not abuse pointer *)
  | While (e, st) ->
    While (expr env e, stmt env st)
  | DoWhile (st, e) ->
    DoWhile (stmt env st, expr env e)
```

⟨Typecheck.stmt() *match* st0.s *cases* 83c⟩+≡ (82d) ◁83b 83d▷

```
  | For (e1either, e2opt, e3opt, st) ->
    (* we may have to do side effects on the environment, so we process
     * e1either first
     *)
    let e1either =
      (match e1either with
      | Left e1opt -> Left (expropt env e1opt)
      | Right _decls -> raise Todo
      )
    in
    For (e1either, expropt env e2opt, expropt env e3opt, stmt env st)
```

⟨Typecheck.stmt() *match* st0.s *cases* 83d⟩+≡ (82d) ◁83c 83e▷

```
  | Continue -> Continue
  | Break -> Break
```

⟨Typecheck.stmt() *match* st0.s *cases* 83e⟩+≡ (82d) ◁83d 83f▷

```
  | Label (name, st) -> Label (name, stmt env st)
  | Goto name -> Goto name
```

8.5.1 if

⟨Typecheck.stmt() *match* st0.s *cases* 83f⟩+≡ (82d) ◁83e 84a▷

```
  | If (e, st1, st2) ->
    let e = expr env e in

    (match e.e_type with
    (* ugly: no real bool type in C; abuse int, float, and worse pointers *)
    | T.I _ | T.F _ | T.Pointer _ -> ()
    (* stricter: error when does not typecheck, not just set null type on e *)
    (* TODO: list remaining cases explicitely *)
    | _ -> type_error e.e_type e.e_loc
    );
    If (e, stmt env st1, stmt env st2)
```

8.5.2 switch

⟨Typecheck.stmt() *match* st0.s *cases* 84a⟩+≡ (82d) ◁83f 84b▷
```
  | Switch (e, xs) ->
    let e = expr env e in

    (* ensure e is a number! not a pointer
     * TODO? I originally accepted T.F _  but I think that was a bug as
     * 5c does not seem to allow it.
     *)
    (match e.e_type with
    | T.I _ -> ()
    | _ -> type_error e.e_type e.e_loc
    );
    (* TODO: do the 0:int - (0:int - x) rewrite? *)
    Switch (e, stmt env xs)
```

⟨Typecheck.stmt() *match* st0.s *cases* 84b⟩+≡ (82d) ◁84a 84c▷
```
  (* less: should enforce int expr? *)
  | Case (e, st) -> Case (expr env e, stmt env st)
  | Default st -> Default (stmt env st)
```

8.5.3 return

⟨Typecheck.stmt() *match* st0.s *cases* 84c⟩+≡ (82d) ◁84b 86b▷
```
  | Return eopt ->
    Return
      (match eopt with
      | None ->
        if env.return_type =*= T.Void
        then None
        (* stricter: error, not warn *)
        else raise (Error (E.Misc ("null return of a typed function",
                                   st0.s_loc)))
      | Some e ->
        let e = expr env e in
        check_compatible_assign Eq_ env.return_type e.e_type e.e_loc;
        (* todo: add cast *)
        Some e
      )
```

8.6 Declarations

⟨*function* Typecheck.check_and_annotate_program.toplevel 84d⟩≡ (69f)
```
  let toplevel (env : env) = function
    ⟨Typecheck.check_and_annotate_program.toplevel() cases 84e⟩
  in
```

8.6.1 Structures and unions

⟨Typecheck.check_and_annotate_program.toplevel() *cases* 84e⟩≡ (84d) 85b▷
```
  | StructDef { su_kind=su; su_name=fullname; su_loc=loc; su_flds=flds }->

    Hashtbl.add env.structs_ fullname
      (su, flds |> List.map
          (fun {fld_name = name; fld_loc=_; fld_type = typ } ->
```

```
            let t = type_ env typ in

            ⟨Typecheck.check_and_annotate_program() if gensymed name 85a⟩
            (name, t)
          )
      )
```

⟨Typecheck.check_and_annotate_program() *if gensymed name* 85a⟩≡ (84e)

```
  (* kenccext: c99ext?:
   * less: if there are multiple anon structure elements, we
   * could check eagerly if no ambiguous fields instead
   * of checking it when you use a field.
   *)
  (match Ast.is_gensymed name, t with
  | false, _ -> ()
  | true, T.StructName _ -> ()
  | true, _ ->
    raise (Error (E.Misc
           ("unnamed structure element must be struct/union", loc)))
  );
```

8.6.2 Enumerations

⟨Typecheck.check_and_annotate_program.toplevel() *cases* 85b⟩+≡ (84d) ◁84e 85c▷

```
  | EnumDef { enum_name = fullname; enum_loc = _loc; enum_constants = csts }
    ->
    (* stricter: no support for float enum constants *)
    let lastvalue = ref 0 in
    let maxt = ref (T.Int, T.Signed) in
    csts |> List.iter (fun
      { ecst_name = fullname; ecst_loc = loc; ecst_value = eopt } ->
        (match eopt with
        | Some e ->
          (try
             (* less: should also return an integer type *)
             let i = Eval_const.eval env.constants e in
             let t = (T.Int, T.Signed) in
             (* todo: maxt := max_types !maxt t; *)
             Hashtbl.add env.constants fullname (i, t);
             lastvalue := i;
           with Eval_const.NotAConstant ->
             raise (Error (E.Misc (spf "enum not a constant: %s"
                                       (unwrap fullname), loc)))
          )
        | None ->
          (* todo: curt *)
          let t = (T.Int ,T.Signed) in
          Hashtbl.add env.constants fullname (!lastvalue, t);
        );
        incr lastvalue
    );
    Hashtbl.add env.enums fullname !maxt
```

8.6.3 Typedef expansions

⟨Typecheck.check_and_annotate_program.toplevel() *cases* 85c⟩+≡ (84d) ◁85b 87▷

```
  | TypeDef { typedef_name = fullname; typedef_loc = _loc; typedef_type =typ}->
    Hashtbl.add env.typedefs fullname (type_ env typ)
```

⟨*function* Typecheck.type_ 86a⟩≡ (163b)

```
  (* Expand typedefs and resolve constant expressions. *)
  let rec type_ (env : env) (typ0 : typ) : Type.t =
    match typ0.t with
    | TBase t -> t
    | TPointer typ -> T.Pointer (type_ env typ)
    | TArray (eopt, typ) ->
        (match eopt with
        | None -> T.Array (None, type_ env typ)
        | Some e ->
          (try
             let i = Eval_const.eval env.constants e in
             T.Array (Some i, type_ env typ)
           with Eval_const.NotAConstant ->
             raise (Error
                    (E.Misc("array size must be a positive constant",typ0.t_loc)))
          )
        )
    | TFunction (tret, (tparams, tdots)) ->
        T.Func (type_ env tret,
                tparams |> List.map (fun p ->
                  let t = type_ env p.p_type in
                  (match t with
                  (* libc.h has a 'typedef long jmp_buf[2]' and then functions
                   * like 'int setjmp(jmp_buf)' so we need to support that.
                   * less: could warn?
                   *)
                  | T.Array (_, t) -> T.Pointer t
                  (* stricter: could transform T.Func in pointer *)
                  | T.Func _ -> type_error t p.p_loc
                  | _ -> t
                  )
                ), tdots)
    | TStructName (su, fullname) -> T.StructName (su, fullname)
    (* expand enums! *)
    | TEnumName fullname -> T.I (Hashtbl.find env.enums fullname)
    (* expand typedefs! *)
    | TTypeName fullname -> Hashtbl.find env.typedefs fullname
```

8.6.4 Variables

Local vars

⟨Typecheck.stmt() *match* st0.s *cases* 86b⟩+≡ (82d) ◁84c

```
  | Var { v_name = fullname; v_loc = loc; v_type = typ;
          v_storage = stoopt; v_init = eopt} ->

    let t = type_ env typ in
    let ini = expropt env eopt in
    (match t with
    (* stricter: forbid nested prototypes *)
    | T.Func _ ->
       raise (Error(E.Misc("prototypes inside functions are forbidden",loc)));
    | _ -> ()
    );
    let sto =
      match stoopt with
      | None -> S.Local
      (* stricter? forbid? confusing anyway to shadow locals *)
```

```
      | Some S.Extern ->
        raise(Error(E.Misc
                    ("extern declaration inside functions are forbidden",loc)))
      | Some S.Static ->
        raise Todo
      | Some S.Local ->
        (* stricter: I warn at least *)
        Error.warn "useless auto keyword" loc;
        S.Local
      | Some (S.Global | S.Param) ->
        raise (Impossible "global/param are not keywords")
    in

    (match ini with
    | None -> ()
    | Some e ->
      (* less: no const checking for this assign *)
      check_compatible_assign Eq_ t e.e_type loc
      (* todo: add cast if not same type *)
    );
    Hashtbl.add env.ids_ fullname { typ = t; sto; ini; loc };

    Var { v_name = fullname; v_loc = loc; v_type = typ; v_storage = stoopt;
          v_init = ini }
```

Global vars

⟨Typecheck.check_and_annotate_program.toplevel() *cases* 87⟩+≡ (84d) ◁85c 88▷

```
  (* remember that VarDecl covers also prototypes *)
  | VarDecl { v_name = fullname; v_loc = loc; v_type = typ;
              v_storage = stoopt; v_init = eopt} ->
    let t = type_ env typ in
    let ini = expropt env eopt in

    (* step 0: typechecking initializer *)
    (match ini with
    | None -> ()
    | Some e ->
      (* less: no const checking for this assign *)
      check_compatible_assign Eq_ t e.e_type loc
    );

    (* step1: check for weird declarations *)
    (match t, ini, stoopt with
    | T.Func _, Some _, _ ->
      raise (Error(E.Misc
                   ("illegal initializer (only var can be initialized)",loc)))
    (* stricter: 5c says nothing, clang just warns *)
    | _, Some _, Some S.Extern ->
      raise (Error (E.Misc ("'extern' variable has an initializer", loc)))
    | _ -> ()
    );

    (try
       (* step2: check for weird redeclarations *)
       let old = Hashtbl.find env.ids_ fullname in

       (* check type compatibility *)
       if not (same_types t old.typ)
```

```
      then raise (Error (E.Inconsistent (
           (* less: could dump both type using vof_type *)
           spf "redefinition of '%s' with a different type"
             (unwrap fullname), loc,
           "previous definition is here", old.loc)))
      else
        let finalt =
          merge_types t old.typ in
        let finalini =
          match ini, old.ini with
          | Some x, None  -> Some x
          | None, Some x -> Some x
          | None, None -> None
          | Some _x, Some _y ->
            raise (Error (E.Inconsistent (
            spf "redefinition of '%s'" (unwrap fullname), loc,
            "previous definition is here", old.loc)))
        in
        (* check storage compatibility and compute final storage *)
        let finalsto =
          merge_storage_toplevel (unwrap fullname) loc stoopt ini old in

        Hashtbl.replace env.ids_ fullname
          {typ = finalt; sto = finalsto; loc = loc; ini = finalini }
    with Not_found ->
      let finalsto =
        match stoopt with
        | None -> S.Global
        | Some S.Extern -> S.Extern
        | Some S.Static -> S.Static
        | Some S.Local ->
          raise (Error(E.Misc
                       ("illegal storage class for file-scoped entity",loc)))
        | Some (S.Global | S.Param) ->
          raise (Impossible "global or param are not keywords")
      in
      Hashtbl.add env.ids_ fullname
        {typ = t; sto = finalsto; loc = loc; ini = ini }
  )
```

8.6.5 Functions

⟨Typecheck.check_and_annotate_program.toplevel() *cases* 88⟩+≡ (84d) ◁87

```
  | FuncDef ({f_name=name; f_loc=loc; f_type=ftyp;
              f_storage=stoopt; f_body=st;} as def) ->

    (* less: lots of code in common with Var_decl; we could factorize
     * but a few things are different still.
     *)
    let t = type_ env ({t = TFunction ftyp; t_loc = loc}) in
    let fullname = (name, 0) in
    (* we use a fake initializer for function definitions to
     * be able to store those definitions in env.ids. That way
     * we can detect function redefinitions, useless redeclarations, etc.
     *)
    let ini = Some { e = Id fullname; e_loc = loc; e_type = T.Void } in

    (try
       (* check for weird redeclarations *)
```

```
   let old = Hashtbl.find env.ids_ fullname in

   (* check type compatibility *)
   if not (same_types t old.typ)
   then raise (Error (E.Inconsistent (
        (* less: could dump both type using vof_type *)
         spf "redefinition of '%s' with a different type"
           (unwrap fullname), loc,
         "previous definition is here", old.loc)))
   else
     let finalt =
       merge_types t old.typ in
     let finalini =
       match ini, old.ini with
       | Some x, None -> Some x
       | None, Some x -> Some x
       | None, None -> None
       | Some _x, Some _y ->
         raise (Error (E.Inconsistent (
         spf "redefinition of '%s'" (unwrap fullname), loc,
         "previous definition is here", old.loc)))
     in
     (* check storage compatibility and compute final storage *)
     let finalsto =
       merge_storage_toplevel (unwrap fullname) loc stoopt ini old in

     Hashtbl.replace env.ids_ fullname
       {typ = finalt; sto = finalsto; loc = loc; ini = finalini }
 with Not_found ->
   let finalsto =
     match stoopt with
     | None -> S.Global
     | Some S.Static -> S.Static
     (* different than for VarDecl here *)
     | Some S.Extern ->
       raise (Error (E.Misc ("'extern' function with initializer", loc)))
     | Some S.Local ->
       raise (Error(E.Misc
                    ("illegal storage class for file-scoped entity",loc)))
     | Some (S.Global | S.Param) ->
       raise (Impossible "global or param are not keywords")
   in
   Hashtbl.add env.ids_ fullname
     {typ = t; sto = finalsto; loc = loc; ini = ini }
);

(* add params in environment before process st *)
let (tret, (tparams, _dots)) = ftyp in
tparams |> List.iter (fun p ->
  p.p_name |> Option.iter (fun fullname ->
    let t = type_ env p.p_type in
    (match t with
    (* stricter: 5c and clang says nothing, could convert in pointer *)
    | T.Array _ | T.Func _ -> type_error t p.p_loc
    (* todo: convert small types to int? see paramconv? *)
    | _ -> ()
    );
    Hashtbl.add env.ids_ fullname
      {typ = t; sto = S.Param; loc = loc; ini = None }
  )
```

```
    );
    (* the expressions inside the statements are now annontated with types *)
    let st = stmt { env with return_type = type_ env tret } st in

    funcs := { def with f_body = st }::!funcs;
```

⟨*function* Typecheck.merge_storage_toplevel 90⟩≡ (163b)

```
  (* If you declare multiple times the same global, we need to make sure
   * the storage declarations are compatible and we need to compute the
   * final (resolved) storage.
   * This function works for toplevel entities (globals but also functions).
   *)
  let merge_storage_toplevel name loc stoopt ini old =
    match stoopt, old.sto with
      (* The None cases first *)

      (* this is ok, a header file can declare many externs and a C file
       * can then selectively "implements" some of those declarations.
       *)
      | None, S.Extern -> S.Global
      | None, S.Global ->
          (* stricter: even clang does not say anything here *)
          if ini =*= None
          then raise (Error (E.Inconsistent (
            spf "useless redeclaration of '%s'" name, loc,
            "previous definition is here", old.loc)))
          else S.Global

      (* stricter: 5c just warns for this *)
      | (None | Some S.Extern), S.Static ->
        raise (Error (E.Inconsistent (
         spf "non-static declaration of '%s' follows static declaration" name,loc,
         "previous definition is here", old.loc)))
      | _, (S.Local | S.Param) ->
        raise (Impossible "globals can't be auto or param")

      (* The Some cases *)

      (* stricter: useless extern *)
      | Some S.Extern, (S.Global | S.Extern) ->
        raise (Error (E.Inconsistent (
          spf "useless extern declaration of '%s'" name, loc,
          "previous definition is here", old.loc)))

      | Some S.Local, _ ->
        raise (Error(E.Misc("illegal storage class for file-scoped entity", loc)))
      | Some S.Static, (S.Extern | S.Global) ->
        raise (Error (E.Inconsistent (
         spf "static declaration of '%s' follows non-static declaration" name,loc,
         "previous definition is here", old.loc)))

      | Some S.Static, S.Static ->
          if ini =*= None
          then raise (Error (E.Inconsistent (
            spf "useless redeclaration of '%s'" name, loc,
            "previous definition is here", old.loc)))
          else S.Static

      | Some (S.Global | S.Param), _ ->
        raise (Impossible "param or global are not keywords")
```

8.7 Const checking and evaluation

⟨*signature* Eval_const.eval 91a⟩≡ (158e)
```
(* may raise NotAConstant or Error *)
val eval: env -> Ast.expr -> integer
```

⟨*type* Eval_const.env 91b⟩≡ (159a 158e)
```
(* less: could do that in rewrite.ml so no need to pass is to eval *)
type env = (Ast.fullname, integer * Type.integer_type) Hashtbl.t
```

⟨*type* Eval_const.integer 91c⟩≡ (159a 158e)
```
(* less: return also float at some point?
 * TODO: use Int64.t (vlong) like in 5c
 *)
type integer = int
```

⟨*exception* Eval_const.NotAConstant 91d⟩≡ (159a 158e)
```
exception NotAConstant
```

⟨*exception* Eval_const.Error 91e⟩≡ (159a 158e)
```
exception Error of error
```

⟨*type* Eval_const.error 91f⟩≡ (159a 158e)
```
(* less: could factorize things in error.ml? *)
type error = Check.error
```

⟨*function* Eval_const.eval 91g⟩≡ (159a)
```
(* stricter: I do not handle float constants for enums *)
let rec eval (env : env) (e0 : expr) : integer =
  match e0.e with
  (* todo: enough for big integers?
   * todo: we should also return an inttype in addition to the integer value.
   *)
  | Int (s, _inttype) -> int_of_string s

  | Id fullname ->
     if Hashtbl.mem env fullname
     then
       let (i, _inttype) = Hashtbl.find env fullname in
       i
     else raise NotAConstant
  | Binary (e1, op, e2) ->
    let i1 = eval env e1 in
    let i2 = eval env e2 in
    (match op with
    | Arith op ->
      (match op with
      | Plus -> i1 + i2
      | Minus -> i1 - i2
      | Mul -> i1 * i2
      | Div ->
        (* stricter: error, not warning *)
        if i2 = 0
        then raise (Error (E.Misc ("divide by zero", e0.e_loc)))
        else i1 / i2
      | Mod ->
        if i2 = 0
        then raise (Error (E.Misc ("modulo by zero", e0.e_loc)))
        else i1 mod i2
      | And -> i1 land i2
```

```
    | Or -> i1 lor i2
    | Xor -> i1 lxor i2
    | ShiftLeft -> i1 lsl i2
    (* less: could be asr! need type information! *)
    | ShiftRight -> i1 lsr i2
    )
  | Logical op ->
    (match op with
    (* ugly: bools are considered ints in C *)
    | Eq    -> if i1 =  i2 then 1 else 0
    | NotEq -> if i1 <> i2 then 1 else 0
    | Inf   -> if i1 <  i2 then 1 else 0
    | Sup   -> if i1 >  i2 then 1 else 0
    | InfEq -> if i1 <= i2 then 1 else 0
    | SupEq -> if i1 >= i2 then 1 else 0
    | AndLog -> raise Todo
    | OrLog -> raise Todo
    )
  )
| Unary (op, e) ->
  let i = eval env e in
  (match op with
  | UnPlus -> i
  | UnMinus -> - i
  | Tilde -> lnot i (* sure? *)
  | _ -> raise Todo
  )

| _ ->
  raise NotAConstant (* todo: more opporunities? *)
```

Chapter 9

Assembly Generation

9.1 Overview

9.2 Arch-specific settings

⟨*type* Arch_compiler.t 93a⟩≡ (147a)
```
type 'instr t = {
  width_of_type: env -> Type.t -> int;
  (* really a (Ast_asm.register, bool) Hashtbl.t with the bool set for
   * reserved registers (e.g., rPC, rLINK, rSP (but not rRET which is ok))
   *)
  regs_initial: int array;
  rSP: Ast_asm.register;
  rRET: Ast_asm.register;

  arith_instr_of_op:
    Ast.binaryOp -> Ast_asm.register -> Ast_asm.register -> Ast_asm.register ->
    'instr;
  move_instr_of_opds:
    (Ast.fullname -> Ast_asm.offset -> Ast_asm.entity) (* entity_of_id *) ->
    Ast_asm.move_size -> opd -> opd -> 'instr;
}
```

⟨*type* Arch_compiler.env 93b⟩≡ (147a)
```
type env = {
  (* same than Typecheck.typed_program.structs? *)
  structs: (Ast.fullname, Type.struct_kind * Type.structdef) Hashtbl.t;
}
```

⟨*function* Arch5.width_of_type 93c⟩≡ (147c)
```
let rec width_of_type (env : Arch_compiler.env) (t : Type.t) : int =
  match t with
  | T.Void -> 0
  | T.I (inttype, _sign) ->
    (match inttype with
    | T.Char -> 1
    | T.Short -> 2
    | T.Int -> 4
    | T.Long -> 4
    | T.VLong -> 8
    )
  | T.F T.Float -> 4
  | T.F T.Double -> 8
  | T.Pointer _ -> 4
```

```
    | T.Func _ -> raise (Impossible "width of Func")
    | T.Array (iopt, t) ->
      (match iopt with
      | None -> raise (Impossible "width of incomplete array")
      | Some i -> i * width_of_type env t
      )
    (* TODO: if union then width is not a sum but a max! *)
    | T.StructName (_su, fullname) ->
        let (_su, flds) = Hashtbl.find env.structs fullname in
        (* todo: align so extra size *)
        flds
        |> List.map (fun (_fld, t) -> width_of_type env t)
        |> List.fold_left (+) 0
```

⟨*signature* Arch5.arch 94a⟩≡ (147b)

```
  val arch: Ast_asm5.instr_with_cond Arch_compiler.t
```

⟨*constant* Arch5.arch 94b⟩≡ (147c)

```
  let arch = {
      width_of_type;
      regs_initial;
      rSP = A5.rSP;
      rRET = A5.rRET;
      arith_instr_of_op;
      move_instr_of_opds;
    }
```

9.3 Code generation environment

⟨*type* Codegen.env 94c⟩≡ (154c)

```
  (* Environment for code generation *)
  type 'i env = {

    (* computed by previous typechecking phase *)
    ids_:     (Ast.fullname, TC.idinfo) Hashtbl.t;
    structs_: (Ast.fullname, Type.struct_kind * Type.structdef) Hashtbl.t;

    (* less: compute offset for each field?
     * fields: (Ast.fullname * string, A.offset) Hashtbl.t
     *)
    arch: Arch.t;
    a: 'i Arch_compiler.t;

    (* the output *)

    pc: Ast_asm.virt_pc ref;

    (* growing array *)
    code: ('i Ast_asm.line * A.loc) array ref;
    (* should contain only DATA or GLOBL (TODO? restrict to pseudo_instr?) *)
    data: ('i Ast_asm.line * A.loc) list ref;

    (* reinitialized for each function *)

    (* reference counting the used registers (size = 16),
     * really a (A.register, int) Hashtbl.t;
     *)
```

```
  regs: int array;
  ⟨Codegen.env other function fields 97c⟩
}
```

9.4 codegen() skeleton

⟨function Codegen.codegen 95a⟩≡ (154c)

```
let codegen (arch : Arch.t) (tp : Typecheck.typed_program) : 'i Ast_asm.program =
  let env = env_of_tp arch tp in
  tp.funcs |> List.iter (codegen_func env);

  (* todo: generate code for ids after, for CGLOBAL *)
  let instrs =
    (Array.sub !(env.code) 0 !(env.pc) |> Array.to_list) @
     List.rev !(env.data)
  in
  (* TODO *)
  let locs = [] in
  instrs, locs
```

⟨function Codegen.env_of_tp 95b⟩≡ (154c)

```
let env_of_tp (arch: Arch.t) (tp : Typecheck.typed_program) : 'i env =
  let arch_compiler: 'i Arch_compiler.t =
    match arch with
    | Arch.Arm -> Obj.magic Arch5.arch
    | Arch.Mips -> Obj.magic Archv.arch
    | _ -> failwith (spf "unsupported arch: %s" (Arch.to_string arch))

  in

  {
    ids_ = tp.ids;
    structs_ = tp.Typecheck.structs;
    arch;
    a = arch_compiler;

    pc = ref 0;
    code = ref [||];
    data = ref [];

    (* TODO: move in nested struct so then can reuse default_xxx () *)
    size_locals = 0;
    size_maxargs = 0;
    offset_locals = ref 0;
    offsets       = Hashtbl_.create ();
    labels        = Hashtbl_.create ();
    forward_gotos = Hashtbl_.create ();
    break_pc = None;
    continue_pc = None;
    regs          = [||];
  }
```

⟨type Codegen.error 95c⟩≡ (154)

```
type error = Check.error
```

⟨exception Codegen.Error 95d⟩≡ (154)

```
exception Error of error
```

9.5 Functions

⟨*function* Codegen.codegen_func 96a⟩≡ (154c)
```
let codegen_func (env : 'i env) (func : func_def) : unit =
  let { f_name=name; f_loc; f_body=st; f_type=typ; f_storage=_ } = func in

  let fullname = (name, 0) in
  let idinfo = Hashtbl.find env.ids_ fullname in
  (* todo: if Flag.profile (can be disabled by #pragma) *)
  let attrs = A.default_attr in

  let spc = add_fake_instr env "TEXT" in

  ⟨Codegen.codegen_func() set offsets for parameters 97e⟩
  ⟨Codegen.codegen_func() adjust env 97f⟩

  stmt env st;

  set_instr env spc
    (A.Pseudo (A.TEXT (global_of_id env fullname, attrs,
                       env.size_locals + env.size_maxargs))) f_loc;
  add_instr env (A.Virtual A.RET) f_loc;

  ⟨Codegen.codegen_func() sanity check env.regs 98a⟩
  ()
```

⟨*function* Codegen.add_fake_instr 96b⟩≡ (154c)
```
let add_fake_instr env str =
  let spc = !(env.pc) in
  add_instr env (A.LabelDef (str ^ "(fake)")) fake_loc;
  spc
```

⟨*function* Codegen.set_instr 96c⟩≡ (154c)
```
let set_instr env pc instr loc =
  if pc >= !(env.pc)
  then failwith (spf "set_instr: pc > env.pc (%d >= %d)" pc !(env.pc));

  !(env.code).(pc) <- (instr, loc)
```

⟨*function* Codegen.add_instr 96d⟩≡ (154c)
```
let add_instr env instr loc =
  (* grow array if necessary *)
  if !(env.pc) >= Array.length !(env.code)
  then begin
    let increment = 100 in
    let newcode =
      Array.make (Array.length !(env.code) + increment)  (fake_instr, fake_loc)
    in
    Array.blit !(env.code) (Array.length !(env.code)) newcode 0 0;
    env.code := newcode
  end;

  !(env.code).(!(env.pc)) <- (instr, loc);
  incr env.pc;
  ()
```

⟨*constant* Codegen.fake_instr 96e⟩≡ (154c)
```
let fake_instr = A.Virtual A.NOP
```

⟨*constant* Codegen.fake_loc 96f⟩≡ (154c)
```
let fake_loc = -1
```

⟨*constant* `Codegen.fake_pc` 97a⟩≡ (154c)
```
let fake_pc = -1
```

⟨*function* `Codegen.global_of_id` 97b⟩≡ (154c)
```
let global_of_id env fullname =
  let idinfo = Hashtbl.find env.ids_ fullname in
  { name = Ast.unwrap fullname;
    priv =
      (match idinfo.TC.sto with
      | S.Static -> Some (-1)
      | S.Global | S.Extern -> None
      | S.Local | S.Param ->
          raise (Impossible "global can be only Static/Global/Extern")
      );
    (* less: analyse idinfo.typ *)
    signature = None;
  }
```

⟨`Codegen.env` *other function fields* 97c⟩≡ (94c) 97d▷
```
mutable size_locals: int;
mutable size_maxargs: int; (* 5c: maxargssafe *)
```

⟨`Codegen.env` *other function fields* 97d⟩+≡ (94c) ◁97c 99f▷
```
(* for parameters and locals *)
offsets: (Ast.fullname, int) Hashtbl.t;
```

⟨`Codegen.codegen_func()` *set* `offsets` *for parameters* 97e⟩≡ (96a)
```
(* TODO: introduce helper *)
(* set offsets for parameters *)
let offsets = Hashtbl_.create () in

let (_typret, (typparams, _varargs)) = typ in

let t = idinfo.TC.typ in
let tparams =
  match t with
  | T.Func (_tret, tparams, _varargs) -> tparams
  | _ -> raise (Impossible "not a FUNC")
in
assert (List.length tparams =|= List.length typparams);

let xs = List_.zip typparams tparams in

let offset = ref 0 in
xs |> List.iter (fun (p, t) ->
  let sizet = env.a.width_of_type {Arch_compiler.structs = env.structs_} t in
  p.p_name |> Option.iter (fun fullname ->
    Hashtbl.add offsets fullname !offset
  );
  (* todo: align *)
  offset := !offset + sizet;
);
```

⟨`Codegen.codegen_func()` *adjust* `env` 97f⟩≡ (96a)
```
(* todo: align offset_locals with return type *)
let env = { env with
  size_locals = 0;
  size_maxargs = 0;
  offset_locals = ref 0;
  offsets       = offsets;
```

```
    labels        = Hashtbl_.create ();
    forward_gotos = Hashtbl_.create ();
    regs          = Array.copy env.a.regs_initial;
  }
  in
```

⟨Codegen.codegen_func() *sanity check* env.regs 98a⟩≡ (96a)
```
  (* sanity check register allocation *)
  env.regs |> Array.iteri (fun i v ->
    if env.a.regs_initial.(i) <> v
    then raise (Error (E.Misc (spf "reg %d left allocated" i, f_loc)));
  );
```

9.6 Register allocation

⟨*constant* Codegen.regs_initial 98b⟩≡ (154c)

⟨*constant* Codegen.rEXT1 98c⟩≡ (154c)

⟨*constant* Codegen.rEXT2 98d⟩≡ (154c)

⟨*function* Codegen.reguse 98e⟩≡ (154c)
```
  let reguse env (A.R x) =
    env.regs.(x) <- env.regs.(x) + 1
```

⟨*function* Codegen.regfree 98f⟩≡ (154c)
```
  let regfree env (A.R x) =
    env.regs.(x) <- env.regs.(x) - 1;
    if env.regs.(x) < 0
    then raise (Error (E.Misc ("error in regfree", fake_loc)))
```

⟨*function* Codegen.with_reg 98g⟩≡ (154c)
```
  let with_reg env (r : A.register) f =
    (* less: care about exn? meh, if exn then no recovery anyway *)
    reguse env r;
    let res = f () in
    regfree env r;
    res
```

⟨*function* Codegen.regalloc 98h⟩≡ (154c)
```
  let regalloc (env : 'i env) loc : int =
    (* less: lasti trick? *)
    let rec aux (i : int) (n : int) : int =
      (* This happens in extreme case when the expression tree has a huge
       * depth everywhere. In that case, we should allocate a new temporary
       * on the stack but this complexifies the algorithm.
       *)
      if i >= n
      then raise (Error(E.Misc("out of fixed registers; rewrite your code",loc)));

      (* TODO: lasti opti *)
      if env.regs.(i) =|= 0
      then begin
        env.regs.(i) <- 1;
        i
      end
      else aux (i+1) n
    in
    aux 0 (Array.length env.regs)
```

9.7 Operand part 1

⟨*type* Codegen.operand_able 99a⟩≡ (154c)
```
type opd = Arch_compiler.opd
```

⟨*type* Codegen.operand_able_kind 99b⟩≡ (154c)

⟨*function* Codegen.opd_regalloc 99c⟩≡ (154c)
```
(* We can reuse a previous register if 'tgtopt' is a register.
 * See for example return.c where we can reuse R0 instead of a new R1.
 *)
let opd_regalloc (env : 'i env) (typ : Type.t) loc (tgtopt : opd option) : opd =
  match typ with
  | T.I _ | T.Pointer _ ->
    let i =
      match tgtopt with
      | Some { opd = Register (A.R x); typ=_; loc=_ } ->
        reguse env (A.R x);
        x
      | _ -> regalloc env loc
    in
    { opd = Register (A.R i); typ; loc }
  | _ -> raise Todo
```

⟨*function* Codegen.opd_regfree 99d⟩≡ (154c)
```
(*
let opd_regalloc_opd env opd tgtopt =
  opd_regalloc env opd.typ opd.loc tgtopt
let opd_regalloc_e env e tgtopt =
  opd_regalloc env e.e_type e.e_loc tgtopt
*)
let opd_regfree env opd =
  match opd.opd with
  | Register r -> regfree env r;
  | _ -> raise (Impossible "opd_regfree on non-register operand")
```

9.8 Statements

⟨*function* Codegen.stmt 99e⟩≡ (154c)
```
let rec stmt (env : 'i env) (st0 : stmt) : unit =
  match st0.s with
  ⟨Codegen.stmt match st0.s cases 99g⟩
```

9.8.1 Local variables

⟨Codegen.env *other function fields* 99f⟩+≡ (94c) ◁97d 101b▷
```
offset_locals: int ref;
```

⟨Codegen.stmt *match* st0.s *cases* 99g⟩≡ (99e) 100a▷
```
| Var { v_name = fullname; v_loc=_;v_storage=_;v_type=_;v_init=_iniTODO} ->
    let idinfo = Hashtbl.find env.ids_ fullname in
    (* todo: generate code for idinfo.ini, handle static locals, etc. *)

    (* update env.offsets *)
    let t = idinfo.TC.typ in
    let sizet = env.a.width_of_type {Arch_compiler.structs = env.structs_} t
    in
```

```
    (* todo: align *)
    env.offset_locals := !(env.offset_locals) + sizet;
    env.size_locals <- env.size_locals + sizet;
    Hashtbl.add env.offsets fullname !(env.offset_locals);
```

9.8.2 Blocks, sequences

⟨Codegen.stmt *match* st0.s *cases* 100a⟩+≡ (99e) ◁99g 100b▷

```
| ExprSt e -> expr env e None
| Block xs -> xs |> List.iter (stmt env)
```

9.8.3 Conditionals

⟨Codegen.stmt *match* st0.s *cases* 100b⟩+≡ (99e) ◁100a 101a▷

```
| If (e, st1, st2) ->
  let goto_else_or_end = ref (expr_cond env e) in
  if st1.s <> Block []
  then stmt env st1;
  if st2.s <> Block []
  then begin
    let goto_end = add_fake_goto env st2.s_loc in
    patch_fake_goto env !goto_else_or_end !(env.pc);
    stmt env st2;
    goto_else_or_end := goto_end;
  end;
  patch_fake_goto env !goto_else_or_end !(env.pc)
```

⟨*function* Codegen.add_fake_goto 100c⟩≡ (154c)

```
let add_fake_goto (env : 'i env) loc =
  let spc = !(env.pc) in
  add_instr env (A.Virtual (A.Jmp (ref (Absolute fake_pc)))) loc;
  spc
```

⟨*function* Codegen.patch_fake_goto 100d⟩≡ (154c)

```
let patch_fake_goto (env : 'i env) (pcgoto : A.virt_pc) (pcdest : A.virt_pc) =
  match fst !(env.code).(pcgoto) with
  (* TODO? what about BL? time to factorize B | BL | Bxx ? *)
  (* ocaml-light: | Instr (B aref) | A.Instr (A.Bxx aref) -> ... *)
  | A.Virtual (A.Jmp aref) ->
    if !aref =*= (Absolute fake_pc)
    then aref := Absolute pcdest
    else raise (Impossible "patching already resolved branch")
  | A.Virtual (A.JEq aref) ->
    if !aref =*= (Absolute fake_pc)
    then aref := Absolute pcdest
    else raise (Impossible "patching already resolved branch")
  | _ -> raise (Impossible "patching non jump instruction")
```

⟨*function* Codegen.expr_cond 100e⟩≡ (154c)

```
(* 5c: bcomplex? () *)
let expr_cond (env : 'i env) (e0 : expr) : virt_pc =
  (* todo: *)
  with_reg env env.a.rRET (fun () ->
    let dst = { opd = Register env.a.rRET; typ = e0.e_type; loc = e0.e_loc } in
    expr env e0 (Some dst);
    (* less: actually should be last loc of e0 *)
    let loc = e0.e_loc in
```

```
    add_instr env (A.Virtual (A.Cmp (0, env.a.rRET))) loc;
    let pc = !(env.pc) in
    add_instr env (A.Virtual (A.JEq (ref (A.Absolute fake_pc)))) loc;
    pc
  )
```

9.8.4 Switch

⟨Codegen.stmt *match* st0.s *cases* 101a⟩+≡ (99e) ◁100b 101c▷

```
  | Switch _
  | Case _ | Default _ ->
    raise Todo
```

9.8.5 Labels and goto

⟨Codegen.env *other function fields* 101b⟩+≡ (94c) ◁99f 102c▷

```
  (* for goto/labels *)
  labels: (string, Ast_asm.virt_pc) Hashtbl.t;

  (* if the label is defined after the goto, when we process the label,
   * we need to update previous goto instructions.
   *)
  forward_gotos: (string, Ast_asm.virt_pc list) Hashtbl.t;
```

⟨Codegen.stmt *match* st0.s *cases* 101c⟩+≡ (99e) ◁101a 101d▷

```
  | Label (name, st) ->

    let here = !(env.pc) in
    Hashtbl.add env.labels name here;

    if Hashtbl.mem env.forward_gotos name
    then begin
      let xs = Hashtbl.find env.forward_gotos name in
      xs |> List.iter (fun xpc -> patch_fake_goto env xpc here);
      (* not really necessary because can not define the same label twice *)
      Hashtbl.remove env.forward_gotos name
    end;
    (* todo? generate dummy Goto +1? *)
    stmt env st
```

⟨Codegen.stmt *match* st0.s *cases* 101d⟩+≡ (99e) ◁101c 102a▷

```
  | Goto name ->
    let here = !(env.pc) in
    let dstpc =
      if Hashtbl.mem env.labels name
      then Hashtbl.find env.labels name
      else begin
        Hashtbl.replace env.forward_gotos name
          (here:: (if Hashtbl.mem env.forward_gotos name
                   then Hashtbl.find env.forward_gotos name
                   else []));
        fake_pc
      end
    in
    add_instr env (A.Virtual (A.Jmp (ref (A.Absolute dstpc)))) st0.s_loc;
```

9.8.6 Loops

⟨Codegen.stmt *match* st0.s *cases* 102a⟩+≡ (99e) ◁101d 102b▷
```
  | While (e, st) ->
    let goto_entry        = add_fake_goto env e.e_loc in
    let goto_for_continue = add_fake_goto env e.e_loc in
    let goto_for_break    = add_fake_goto env e.e_loc in
    patch_fake_goto env goto_for_continue !(env.pc);
    patch_fake_goto env goto_entry !(env.pc);

    let goto_else = expr_cond env e in
    patch_fake_goto env goto_else goto_for_break;

    let env = { env with
      break_pc = Some goto_for_break;
      continue_pc = Some goto_for_continue;
    }
    in
    stmt env st;

    (* less: should be last loc of st? *)
    let loc = e.e_loc in
    add_instr env (A.Virtual (A.Jmp (ref(A.Absolute goto_for_continue)))) loc;
    patch_fake_goto env goto_for_break !(env.pc)
```

⟨Codegen.stmt *match* st0.s *cases* 102b⟩+≡ (99e) ◁102a 102d▷
```
  | DoWhile (st, e) ->

    let goto_entry        = add_fake_goto env e.e_loc in
    let goto_for_continue = add_fake_goto env e.e_loc in
    let goto_for_break    = add_fake_goto env e.e_loc in
    patch_fake_goto env goto_for_continue !(env.pc);
    (* for a while: patch_fake_goto env goto_entry env.pc; *)

    let goto_else = expr_cond env e in
    patch_fake_goto env goto_else goto_for_break;
    (* for a dowhile! *)
    patch_fake_goto env goto_entry !(env.pc);

    let env = { env with
      break_pc = Some goto_for_break;
      continue_pc = Some goto_for_continue;
    }
    in
    stmt env st;

    (* less: should be last loc of st? *)
    let loc = e.e_loc in
    add_instr env (A.Virtual (A.Jmp (ref(A.Absolute goto_for_continue)))) loc;
    patch_fake_goto env goto_for_break !(env.pc)
```

⟨Codegen.env *other function fields* 102c⟩+≡ (94c) ◁101b
```
  (* for loops (and switch) *)
  (* reinitialized for each block scope *)
  break_pc: Ast_asm.virt_pc option;
  continue_pc: Ast_asm.virt_pc option;
```

⟨Codegen.stmt *match* st0.s *cases* 102d⟩+≡ (99e) ◁102b 103a▷
```
  | Break ->
    (match env.break_pc with
```

```
    | Some dst ->
      add_instr env (A.Virtual (A.Jmp (ref(A.Absolute dst)))) st0.s_loc;
    | None -> raise (Impossible "should be detected in check.ml")
    )
  | Continue ->
    (match env.continue_pc with
    | Some dst ->
      add_instr env (A.Virtual (A.Jmp (ref(A.Absolute dst)))) st0.s_loc;
    | None -> raise (Impossible "should be detected in check.ml")
    )
```

⟨Codegen.stmt *match* st0.s *cases* 103a⟩+≡ (99e) ◁102d 103c▷

```
  | For (e1either, e2opt, e3opt, st) ->
    (match e1either with
    | Left e1opt -> expropt env e1opt
    (* todo: scope, should reset autoffset once processed loop *)
    | Right decls ->
      decls |> List.iter (fun var -> stmt env { s = Var var; s_loc=var.v_loc});
    );
    let goto_entry        = add_fake_goto env st0.s_loc in
    let goto_for_continue = add_fake_goto env st0.s_loc in
    let goto_for_break    = add_fake_goto env st0.s_loc in

    patch_fake_goto env goto_for_continue !(env.pc);
    expropt env e3opt;
    patch_fake_goto env goto_entry !(env.pc);
    (match e2opt with
    | None -> ()
    | Some e2 ->
      let goto_else = expr_cond env e2 in
      patch_fake_goto env goto_else goto_for_break;
    );

    let env = { env with
      break_pc = Some goto_for_break;
      continue_pc = Some goto_for_continue;
    }
    in
    stmt env st;

    let loc = st0.s_loc in
    add_instr env (A.Virtual (A.Jmp (ref(A.Absolute goto_for_continue)))) loc;
    patch_fake_goto env goto_for_break !(env.pc)
```

9.8.7 Control flow jumps

⟨*constant* Codegen.rRET 103b⟩≡ (154c)

⟨Codegen.stmt *match* st0.s *cases* 103c⟩+≡ (99e) ◁103a

```
  | Return eopt ->
    (match eopt with
    | None ->
      add_instr env (A.Virtual A.RET) st0.s_loc

    | Some e ->
      (* todo: if type compatible with R0 *)
      with_reg env env.a.rRET (fun () ->
        let dst = { opd = Register env.a.rRET; typ = e.e_type; loc = e.e_loc } in
        expr env e (Some dst);
```

```
      add_instr env (A.Virtual A.RET) st0.s_loc
    )
  )
```

9.9 Basic expressions

⟨*function* Codegen.expr 104a⟩≡ (154c)
```
(* todo: inrel ?
 * todo: if complex type node
 * 5c: called cgen/cgenrel()
 *)
let rec expr (env : 'i env) (e0 : expr) (dst_opd_opt : opd option) : unit=
  match operand_able e0 with
  | Some opd1 -> gmove_opt env opd1 dst_opd_opt
  | None ->
    (match e0.e with
    | Int _ | Float _ | Id _ ->
        raise (Impossible "handled in operand_able()")
    | String _ | ArrayAccess _ | RecordPtAccess _ | SizeOf _ ->
        raise (Impossible "should have been converted before")
    ⟨Codegen.expr() when not operand able, match e0.e cases 107d⟩
    | RecordAccess _
    | Cast _
    | Postfix _ | Prefix _
    | CondExpr _
    | ArrayInit _ | RecordInit _ | GccConstructor _
      ->
      Logs.err (fun m -> m "%s" (Dumper_.s_of_any (Expr e0)));
      raise Todo
    )
```

⟨*function* Codegen.expropt 104b⟩≡ (154c)
```
let expropt (env : 'i env) (eopt : expr option) : unit =
  match eopt with
  | None -> ()
  | Some e -> expr env e None
```

9.9.1 Operand part 2

⟨*function* Codegen.operand_able 104c⟩≡ (154c)
```
let operand_able (e0 : expr) : opd option =
  let kind_opt =
    match e0.e with
    | String _ | ArrayAccess _ | RecordPtAccess _ | SizeOf _ ->
      raise (Impossible "should have been converted")
    ⟨operand_able() match e0.e cases 104d⟩
  in
  match kind_opt with
  | None -> None
  | Some opd -> Some { opd; typ = e0.e_type; loc = e0.e_loc }
```

⟨operand_able() *match* e0.e *cases* 104d⟩≡ (104c) 105a▷
```
(* less: could be operand_able if we do constant_evaluation later *)
| Binary (_e1, _op, _e2) -> None
| Call _ | Assign _ | Postfix _ | Prefix _ | CondExpr _ | Sequence _
  -> None
```

```
  | Cast _ -> raise Todo
  | RecordAccess _ -> raise Todo

  | ArrayInit _ | RecordInit _ | GccConstructor _ ->
    None
```

9.9.2 Numeric constants

⟨operand_able() *match* e0.e *cases* 105a⟩+≡ (104c) ◁104d 105b▷

```
  | Int (s, _inttype) -> Some (ConstI (int_of_string s))
```

⟨operand_able() *match* e0.e *cases* 105b⟩+≡ (104c) ◁105a 105c▷

```
  (* todo: float handling *)
  | Float _ -> None
```

9.9.3 Entity uses

⟨operand_able() *match* e0.e *cases* 105c⟩+≡ (104c) ◁105b 105d▷

```
  | Id fullname -> Some (Name (fullname, 0))
```

9.9.4 Pointers part1

⟨operand_able() *match* e0.e *cases* 105d⟩+≡ (104c) ◁105c

```
  | Unary (op, e) ->

    (match op with

    (* special case to handle *(&arr + <cst>) *)
    | DeRef  ->

      (match e.e with
      (* less: this should be handled in rewrite.ml *(&x) ==> x *)
      | (Unary (GetRef, { e = Id fullname; e_loc=_;e_type=_ })) -> Some (Name (fullname, 0))
      (* less: should normalize constant to left or right in rewrite.ml *)
      | Binary ({ e = Int (s1, _); e_loc=_;e_type=_ },
                Arith Plus,
                {e = (Unary (GetRef, { e = Id fullname; e_loc=_;e_type=_ })); e_loc=_;e_type=_ })
        -> Some (Name (fullname, int_of_string s1))
      | Binary ({e = (Unary (GetRef, { e = Id fullname; e_loc=_;e_type=_ })); e_loc=_;e_type=_ },
                Arith Plus,
                { e = Int (s1, _); e_loc=_;e_type=_a })
        -> Some (Name (fullname, int_of_string s1))
      | _ -> None
      )

    | GetRef ->

      (match e.e with
      (* why 5c does not make OADDR (ONAME) an addressable node? *)
      | Id fullname -> Some (Addr fullname)
      | _ -> None
      )

    | (UnPlus | UnMinus | Tilde) ->
      raise (Impossible "should have been converted")
    | Not -> None
    )
```

9.9.5 gmove()

⟨*function* Codegen.gmove_opt 106a⟩≡ (154c)
```
let gmove_opt (env : 'i env) (opd1 : opd) (opd2opt : opd option) :
    unit =
  match opd2opt with
  | Some opd2 -> gmove env opd1 opd2
  | None ->
    (* SURE? should we still registerize and all? *)
    (* less: should have warned about unused opd in check.ml *)
    ()
```

⟨*function* Codegen.gmove 106b⟩≡ (154c)
```
(* Even though two arguments are operand_able, it does not mean
 * we can move one into the other with one instruction.
 * In theory, 5a supports general MOVW, but 5l restricts those
 * MOVW to only store and load (not both at the same time).
 * This is why we must decompose below the move in 2 instructions
 * sometimes.
 *)
let rec gmove (env : 'i env) (opd1 : opd) (opd2 : opd) : unit =
  match opd1.opd with

  (* a load *)
  | Name _ | Indirect _ ->
    let move_size =
      match opd1.typ with
      | T.I (T.Int, _) | T.Pointer _ -> A.Word
      | _ -> raise Todo
    in
    (* less: opti which does opd_regfree env opd2 (Some opd2)? worth it? *)
    let opd1reg = opd_regalloc env opd1.typ opd1.loc (Some opd2) in
    gmove_aux env move_size opd1 opd1reg;
    gmove env opd1reg opd2;
    opd_regfree env opd1reg

  | ConstI _ | Register _ | Addr _  ->

    (match opd2.opd with

    (* a store *)
    | Name _ | Indirect _ ->
      let move_size =
        match opd2.typ with
        | T.I (T.Int, _) | T.Pointer _ -> A.Word
        | _ -> raise Todo
      in
      (* less: opti which does opd_regfree env opd2 (Some opd1)?? *)
      let opd2reg = opd_regalloc env opd2.typ opd2.loc None in
      gmove env opd1 opd2reg;
      gmove_aux env move_size opd2reg opd2;
      opd_regfree env opd2reg

    | ConstI _ | Register _ | Addr _ ->

      (* the simple cases *)
      let move_size =
        match opd1.typ, opd2.typ with
        | T.I (T.Int, _), T.I (T.Int, _) -> A.Word
        | T.Pointer _, T.Pointer _ -> A.Word
```

```
          (* todo: lots of opti related to float *)
          | _ -> raise Todo
        in
        gmove_aux env move_size opd1 opd2
      )

  (* At this point, either opd1 or opd2 references memory (but not both),
   * so we can do the move in one instruction.
   * 5c: called gins()
   *)
  and gmove_aux env move_size (opd1 : opd) (opd2 : opd) : unit =
    (* less: should happen only for register? *)
    if opd1.opd =*= opd2.opd
    then ()
    else
    add_instr env
      (A.Instr (env.a.move_instr_of_opds (entity_of_id env) move_size opd1 opd2))
      opd1.loc
```

⟨*function* `Codegen.mov_operand_of_opd` 107a⟩≡ (154c)

⟨*function* `Codegen.entity_of_id` 107b⟩≡ (154c)

```
  let entity_of_id (env : 'i env) (fullname : fullname) (offset_extra : int) :
       A.entity =
    let idinfo = Hashtbl.find env.ids_ fullname in
    match idinfo.TC.sto with
    | S.Param ->
      let offset = Hashtbl.find env.offsets fullname + offset_extra in
      Param (Some (symbol fullname), offset)
    | S.Local ->
      let offset = Hashtbl.find env.offsets fullname + offset_extra in
      (* - offset for locals *)
      A.Local (Some (symbol fullname), - offset)
    | S.Static | S.Global | S.Extern ->
      let offset = offset_extra in
      A.Global (global_of_id env fullname, offset)
```

⟨*function* `Codegen.symbol` 107c⟩≡ (154c)

```
  let symbol fullname = Ast.unwrap fullname
```

9.10 Complex expressions

⟨`Codegen.expr()` *when not operand able, match* `e0.e` *cases* 107d⟩≡ (104a) 108b ▷

```
  | Sequence (e1, e2) ->
    expr env e1 None;
    expr env e2 dst_opd_opt
```

9.10.1 Complexity

⟨*constant* `Codegen.fn_complexity` 107e⟩≡ (154c)

```
  let fn_complexity = 100
```

⟨*function* `Codegen.complexity` 108a⟩≡ (154c)

```
(* less: could optimize by caching result in node, so no need
 * call again complexity on subtree later
 *)
let rec complexity (e : expr) : int =
  if operand_able e <> None
  then 0
  else
    match e.e with
    | Int _ | Float _ | String _ | Id _ -> 0
    | Call _ -> fn_complexity
    | Assign _ | ArrayAccess _ | Binary _ | Sequence _ ->
      let (e1, e2) =
        match e.e with
        | Assign (_, e1, e2) -> e1, e2
        | ArrayAccess (e1, e2) -> e1, e2
        | Binary (e1, _, e2) -> e1, e2
        | Sequence (e1, e2) -> e1, e2
        | _ -> raise (Impossible "see pattern match above")
      in
      let n1 = complexity e1 in
      let n2 = complexity e2 in
      if n1 =|= n2
      then 1 + n1
      else max n1 n2

    | CondExpr (e1, e2, e3) ->
      complexity {e with e = Sequence (e1, { e with e = Sequence (e2, e3) } ) }

    | RecordAccess _ | RecordPtAccess _ | Cast _ | Postfix _ | Prefix _ | Unary _
      ->
       let e =
         match e.e with
         | RecordAccess (e, _) -> e
         | RecordPtAccess (e, _) -> e
         | Cast (_, e) -> e
         | Postfix (e, _) -> e
         | Prefix (_, e) -> e
         | Unary (_, e) -> e
         | _ -> raise (Impossible "see pattern match above")
      in
      let n = complexity e in
      if n =|= 0 then 1 else n
    (* should be converted in Int anyway *)
    | SizeOf _ -> 0

    | ArrayInit _ | RecordInit _ | GccConstructor _ -> raise Todo
```

9.10.2 String constants

9.10.3 Arithmetic expressions

⟨`Codegen.expr()` *when not operand able, match* `e0.e` *cases* 108b⟩+≡ (104a) ◁107d 109b▷

```
(* less: lots of possible opti *)
| Binary (e1, op, e2) ->

  (match op with
  | Arith (Plus | Minus
          | And | Or | Xor
```

```
          | ShiftLeft | ShiftRight
          | Mul | Div | Mod) ->

      let n1 = complexity e1 in
      let n2 = complexity e2 in

      let opdres, opdother =
        if n1 >= n2
        then begin
          let opd1reg = opd_regalloc env e1.e_type e1.e_loc dst_opd_opt in
          expr env e1 (Some opd1reg);
          let opd2reg = opd_regalloc env e2.e_type e2.e_loc None in
          expr env e2 (Some opd2reg);
          (match opd1reg.opd, opd2reg.opd with
          | Register r1, Register r2 ->
            (* again reverse order SUB r2 r1 ... means r1 - r2 *)
            add_instr env (A.Instr (env.a.arith_instr_of_op op r2 r1 r1))
              e0.e_loc;
          | _ -> raise (Impossible "both operands comes from opd_regalloc")
          );
          opd1reg, opd2reg
        end
        else begin
          let opd2reg = opd_regalloc env e2.e_type e2.e_loc dst_opd_opt in
          expr env e2 (Some opd2reg);
          let opd1reg = opd_regalloc env e1.e_type e1.e_loc None in
          expr env e1 (Some opd1reg);
          (match opd1reg.opd, opd2reg.opd with
          | Register r1, Register r2 ->
            (* This time we store result in r2! important and subtle.
             * This avoids some extra MOVW; see plus_chain.c
             *)
            add_instr env (A.Instr (env.a.arith_instr_of_op op r2 r1 r2))
              e0.e_loc;
          | _ -> raise (Impossible "both operands comes from opd_regalloc")
          );
          opd2reg, opd1reg
        end
      in
      (* This is why it is better for opdres to be the register
       * allocated from dst_opd_opt so the MOVW below can become a NOP
       * and be removed.
       *)
      gmove_opt env opdres dst_opd_opt;

      opd_regfree env opdres;
      opd_regfree env opdother;

    | Logical _ ->
      raise Todo
    )
```

⟨*function* `Codegen.arith_instr_of_op` 109a⟩≡ (154c)

9.10.4 Boolean expressions

9.10.5 Assignments part 1

⟨`Codegen.expr()` *when not operand able, match* `e0.e` *cases* 109b⟩+≡ (104a) ◁108b 110▷

```
  | Assign (op, e1, e2) ->
    (match op with
    | Eq_ ->
      (match operand_able e1, operand_able e2, dst_opd_opt with
      (* ex: x = 1; *)
      | Some opd1, Some opd2, None ->
        (* note that e1=e2 -->  MOVW opd2,opd1, (right->left -> left->right)*)
        gmove env opd2 opd1

      (* ex: return x = 1;, x = y = z, ... *)
      | Some _opd1, Some _opd2, Some _dst ->
        raise Todo


      (* ex: y = &x;, y = x + y, ... *)
      | Some opd1, None, None ->
        let opd2reg = opd_regalloc env e2.e_type e2.e_loc None in
        expr env e2 (Some opd2reg);
        gmove env opd2reg opd1;
        opd_regfree env opd2reg;

      (* ex: return x = x+y;, x = y = z, ... *)
      | Some opd1, None, Some dst ->
        let opd2reg = opd_regalloc env e2.e_type e2.e_loc None in
        expr env e2 (Some opd2reg);
        gmove env opd2reg opd1;
        gmove env opd2reg dst; (* only diff with case above *)
        opd_regfree env opd2reg;


      (* ex: *x = 1; *)
      | None, _, _ ->
        raise Todo
      )
    | OpAssign _op ->
      raise Todo
    )
```

9.10.6 Pointers part2

⟨Codegen.expr() *when not operand able, match* e0.e *cases* 110⟩+≡ (104a) ◁109b

```
  | Unary (op, e) ->

    (match op with
    | GetRef ->
      (match e.e  with
      | Id _fullname ->
        raise (Impossible "handled in operand_able()")
      | Unary (DeRef, _) ->
        raise (Impossible "should be simplified in rewrite.ml")
      | _ ->
        raise (Impossible "not an lvalue?")
      )

    | DeRef ->
      (* less: opti of Deref of Add with constant? *)
      let opd1reg = opd_regalloc env e.e_type e.e_loc dst_opd_opt in
      expr env e (Some opd1reg);
      gmove_opt env
```

```
      (match opd1reg.opd with
      | Register r -> { opd = Indirect (r, 0); loc = e.e_loc;
                        typ = e0.e_type }
      | _ -> raise (Impossible "opd_regalloc_e returns always Register")
      ) dst_opd_opt;
    opd_regfree env opd1reg;


  | (UnPlus | UnMinus | Tilde) ->
    raise (Impossible "should have been converted")
  | Not -> raise Todo
  )
| Call (e, es) ->
    if complexity e >= fn_complexity
    then
      (* (foo(...))(...), so function call in e itself *)
      raise Todo
    else begin
      arguments env es;
      (match operand_able e with
      (* complex call *)
      | None -> raise Todo
      | Some opd ->
         add_instr env (A.Virtual (A.JmpAndLink
                        (ref (branch_operand_of_opd env opd)))) e0.e_loc;
         dst_opd_opt |> Option.iter (fun dst_opd ->
            with_reg env env.a.rRET (fun () ->
                (* TODO? need Cast? 5c does gopcode(OAS, ...) *)
                let src_opd = { opd = Register env.a.rRET; typ = e0.e_type;
                                loc = e0.e_loc;} in
                gmove env src_opd dst_opd
            );
         );
      );
      ()
    end
```

9.10.7 Assignments part 2

9.10.8 Assignments and arithmetic

9.10.9 Function calls

9.10.10 Array accesses

9.10.11 Field accesses

9.10.12 Cast

9.10.13 Ternary expressions

9.10.14 Prefix/postfix

9.10.15 `sizeof()`

9.11 Boolean expressions

9.12 Other topics

9.12.1 Sizes

9.12.2 First argument

9.12.3 Alignment

9.12.4 Toplevel globals

9.12.5 Initializers

9.13 Advanced assembly generation

9.13.1 Function calls

9.13.2 Switch

Chapter 10

Object File Generation

10.1 Object format

10.2 Instruction output

Chapter 11

AST-level Optimizations

11.1 AST simplifications

11.1.1 General rewrites

11.1.2 Constant evaluation

11.2 Comma hoisting

11.3 Bitshifting opportunities

11.4 And opportunities

11.5 Immediate operands

11.5.1 Substraction

11.5.2 Addition, or, etc.

11.5.3 Multiplication

11.6 Associative-commutative arithmetic optimisations

Chapter 12

Assembly-level Optimisations

12.1 Nop detection

12.2 ARM special instructions

12.3 Register passing argument: `REGARG`

12.4 Register allocation optimisations

12.5 Peephole optimizer

12.6 Dominators

Chapter 13

Linking Support

13.1 `#pragma lib` and automagic linking

13.2 Safe linking with type signatures: `5c -T`

13.2.1 `sign()`

13.2.2 `#pragma incomplete`

Chapter 14

Debugging Support

14.1 General debugging metadata

14.2 Acid debugger metadata: `5c -a`

14.2.1 Entities

14.2.2 Structure/union definitions

14.3 Pickle: `5c -Z`

Chapter 15

Profiling Support

15.1 Text attributes

15.2 `#pragma profile`

Chapter 16

Preprocessing

16.1 Overview

⟨*signature* `Parse_cpp.parse` 119a⟩≡ (167a)
```
(* Wrapper function around a parser/lexer.
 *
 * Regarding Cap.open_in, [parse] will [open_in] the passed file parameter but
 * may also [open_in] other files as [parse] will recursively process
 * [#include] directives.
 *)
val parse:
  < Cap.open_in; .. > ->
  ('token, 'ast) hook -> Preprocessor.conf -> Fpath.t ->
  'ast * Location_cpp.location_history list
```

16.2 Core data structures

16.2.1 Preprocessor configuration

⟨*type* `Preprocessor.conf` 119b⟩≡ (168)
```
type conf = {
  (* -D *)
  defs: (string * string) list;
  (* -I + system paths (e.g., /usr/include) *)
  paths: Fpath.t list;
  (* the directory of the C file so it is looked for "" but not for <> *)
  dir_source_file: Fpath.t;
}
```

⟨`CLI.main()` *main code path, define macropreprocessor* `conf` 119c⟩≡ (26b)
```
let system_paths : Fpath.t list =
  ⟨CLI.main() main code path, define system_paths 120a⟩
in
let conf = Preprocessor.{
  defs = !macro_defs;
  (* this order? *)
  paths = system_paths @ List.rev !include_paths;
  dir_source_file = Fpath.v (Filename.dirname cfile);
}
in
```

⟨CLI.main() *main code path, define* system_paths 120a⟩≡ (119c)
```
    (try CapSys.getenv caps "INCLUDE" |> Str.split (Str.regexp "[ \t]+")
    with Not_found ->
      [spf "/%s/include" thestring;
       "/sys/include";
      ] |> (fun xs -> if !ape then "/sys/include/ape"::xs else xs)
    ) |> Fpath_.of_strings
```

⟨CLI.main() *other locals* 120b⟩≡ (25d) 140f▷
```
  (* Ansi Posix Environment for plan9 *)
  let ape = ref false in
```

⟨CLI.main() options *elements* 120c⟩+≡ (25d) ◁59b 134b▷
```
  "-ape", Arg.Set ape,
  " ";
```

16.2.2 Directives AST

⟨*type* Ast_cpp.directive 120d⟩≡ (165a)
```
  type directive =
    | Include of Fpath.t * bool (* true if <>, false if "" *)

    | Define of macro
    | Undef of string

    | Ifdef of string
    | Ifndef of string
    | Else
    | Endif

    | Line of int * Fpath.t
    (* ex: #pragma lib "libc.a" -> Pragma("lib", ["libc.a"]) *)
    | Pragma of string * string list
```

⟨*type* Ast_cpp.macro 120e⟩≡ (165a)
```
    and macro = {
      name: string;
      (* Note that you can have None or Some [] for parameters.
       * The first is a macro without parameter, the second a macro with
       * 0 parameters (but that still needs to be called by FOO())
       *)
      params: string list option;
      varargs: bool;
      (* The body below has been processed and every parameter occurence
       * is replaced by a #xxx where xxx is the number corresponding to the
       * ith parameter.
       *)
      body: string option
    }
```

16.2.3 Location and line history

⟨*signature* Location_cpp.history 120f⟩≡ (166a)
```
  (* both should be reseted each time you parse a new file *)
  val history: location_history list ref
```

⟨*signature* Location_cpp.line 120g⟩≡ (166a)
```
  val line: loc ref
```

⟨*type* Location_cpp.loc 121a⟩≡ (166)
```
(* global line number, after pre-processing *)
type loc = int
```

⟨*type* Location_cpp.final_loc 121b⟩≡ (166)
```
(* final readable location *)
type final_loc = Fpath.t * int
```

⟨*type* Location_cpp.location_history 121c⟩≡ (166)
```
type location_history = {
  location_event: location_event;
  global_line: loc;
}
```

⟨*type* Location_cpp.location_event 121d⟩≡ (166)
```
  and location_event =
    (* #include "foo.h" *)
    | Include of Fpath.t
    (* #line 1 "foo.c" *)
    | Line of int * Fpath.t
    (* end of #include, back to includer *)
    | Eof
```

⟨*constant* Location_cpp.history 121e⟩≡ (166b)
```
let history = ref []
```

⟨*constant* Location_cpp.line 121f⟩≡ (166b)
```
(* Global line number (after pre-processing).
 * Note that you need another data structure to map a global line number
 * to a (file, line) pair (see history and final_loc_of_loc below).
 *)
let line = ref 1
```

16.2.4 Parser hook

⟨*type* Parse_cpp.token_category 121g⟩≡ (167)
```
type token_category =
  | Eof
  | Sharp
  | Ident of string
  | Other
```

⟨*type* Parse_cpp.hook 121h⟩≡ (167)
```
type ('token, 'ast) hook = {
  lexer: Lexing.lexbuf -> 'token;
  parser: (Lexing.lexbuf -> 'token) -> Lexing.lexbuf -> 'ast;
  category: 'token -> token_category;
  (* TODO: delete, redundant with Eof token_category above *)
  eof: 'token;
}
```

⟨*function* Parse.parse 122a⟩≡ (160e)
```
let parse (caps : < Cap.open_in; .. >) (conf : Preprocessor.conf) (file : Fpath.t) : Ast.program =
  let hooks = Parse_cpp.{
     lexer = Lexer.token;
     category = (fun t ->
       match t with
       | T.EOF    -> Parse_cpp.Eof
       | T.TSharp -> Parse_cpp.Sharp

       (* ocaml-light: | T.TName (_, s) | T.TTypeName (_, s) *)
       | T.TName (_, s)  -> Parse_cpp.Ident s
       | T.TTypeName (_, s) -> Parse_cpp.Ident s
        (* stricter: I forbid to have macros overwrite keywords *)
        (*
        | T.Tvoid | T.Tchar | T.Tshort | T.Tint | T.Tlong
        | T.Tdouble | T.Tfloat | T.Tsigned | T.Tunsigned
        | T.Tstruct | T.Tunion | T.Tenum | T.Ttypedef
        | T.Tconst | T.Tvolatile | T.Trestrict | T.Tinline
        | T.Tauto | T.Tstatic | T.Textern | T.Tregister
        | T.Tif | T.Telse | T.Twhile | T.Tdo | T.Tfor
        | T.Tbreak | T.Tcontinue | T.Treturn | T.Tgoto
        | T.Tswitch | T.Tcase | T.Tdefault | T.Tsizeof
        *)
       | _ -> Parse_cpp.Other
     );
    parser = Parser.prog;
    eof = T.EOF;
  }
  in
  Parse_cpp.parse caps hooks conf file
```

16.3 Lexing

⟨macroprocessor/Lexer_cpp.mll 122b⟩≡
```
{
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Regexp_.Operators

open Ast_cpp
module L = Location_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* A port of plan 9 builtin preprocessing in OCaml.
 *
 * The main functions of the text preprocessors are:
 *  - including other files
 *  - expanding macros
 *  - skipping text inside ifdefs
 *
 *
 * Limitations compared to cpp:
 *  - no support for unicode
 *
 * stricter:
 *  - no space allowed between '#' and the directive
```

```
 *    (even though some people like to do  '#  ifdef', when have nested ifdefs,
 *     but nested ifdefs are bad practice anyway)
 *  - single '#' are not converted in #endif
 *    (who does use that?)
 *  - only space or comment between the directive and the newline;
 *    we do not skip any character
 *    (who does put garbage there?)
 *)

let error s =
  raise  (L.Error (spf "Lexical error in cpp: %s" s, !L.line))

(* needed only because of ocamllex limitations in ocaml-light
 * which does not support the 'as' feature.
 *)
let symbol_after_space lexbuf =
  let s = Lexing.lexeme lexbuf in
  s =~ ".*[ \t]\\([^ \t]+\\)" |> ignore;
  Regexp_.matched1 s
let char lexbuf =
  let s = Lexing.lexeme lexbuf in
  String.get s 0

}

(*****************************************************************************)
(* Regexps aliases *)
(*****************************************************************************)
let space  = [' ''\t']
let letter = ['a'-'z''A'-'Z']
let digit  = ['0'-'9']

let symbol = (letter | '_') (letter | digit | '_')*

(*****************************************************************************)
(* Main rule *)
(*****************************************************************************)

(* pre: 'token' below assumes the '#' has already been consumed *)
rule token = parse

  (* note that filenames containing double quotes are not supported *)
  | "include" space* '"' ([^ '"' '\n']+ (*as file*)) '"'
      { let file =
          let s = Lexing.lexeme lexbuf in s =~ ".*\"\\(.*\\)\"" |> ignore;
          Regexp_.matched1 s in
        space_or_comment_and_newline lexbuf;
        Include (Fpath.v file, false)
      }
  | "include" space* '<' ([^ '>' '\n']+ (*as file*)) '>'
      { let file =
          let s = Lexing.lexeme lexbuf in s =~ ".*<\\(.*\\)>" |> ignore;
          Regexp_.matched1 s in
        space_or_comment_and_newline lexbuf;
        Include(Fpath.v file, true)
      }
  | "include" { error "syntax in #include" }

  (* Macro definition part 1 *)
  | "define" space+ (symbol (*as s1*)) '(' ([^')']* (*as s2*)) ')'
```

```
    { let (s1, s2) =
        let s = Lexing.lexeme lexbuf in s =~ "define[ \t]+\\(.*\\)(\\(.*\\))" |> ignore;
        Regexp_.matched2 s in
      let xs = Str.split (Str.regexp "[ \t]*,[ \t]*") s2 in
      (* check if identifier or "..." for last one *)
      let params, varargs =
        let rec aux xs =
          match xs with
          | [] -> [], false
          (* todo: __VA_ARGS__ *)
          | ["..."] -> [], true
          | "..."::_ -> error "... should be the last parameter of a macro"
          | x::xs ->
             if x =~ "[A-Za-z_][A-Za-z_0-9]*"
             then let (params, bool) = aux xs in x::params, bool
             else error (spf "wrong syntax for macro parameter: %s" x)
        in
        aux xs
      in
      let body = define_body s1 (List_.index_list_1 params) lexbuf in
      Define { name = s1; params = Some (params); varargs = varargs;
               body = Some body }
    }
(* a space after the symbol means the macro has no argument, even if
 * this space is followed by a '('.
 *)
| "define" space+ (symbol (*as s1*)) space+
    { let s1 = symbol_after_space lexbuf in
      let body = define_body s1 [] lexbuf in
      Define { name = s1; params = None; varargs = false; body = Some body}
    }
(* if do '#define FOO+' then? we should return a syntax error because
 * of the space_or_comment_and_newline below.
 *)
| "define" space+ (symbol (*as s1*))
    { let s1 = symbol_after_space lexbuf in
      space_or_comment_and_newline lexbuf;
      Define { name = s1; params = None; varargs = false; body =  None }
    }
| "define" { error "syntax in #define" }

(* stricter: require space_or_comment-only after sym *)
| "undef" space+ (symbol (*as name*))
    { let name = symbol_after_space lexbuf in
      space_or_comment_and_newline lexbuf;
      Undef name
    }
| "undef" { error "syntax in #undef" }


| "ifdef" space+ (symbol (*as name*))
    { let name = symbol_after_space lexbuf in
      space_or_comment_and_newline lexbuf;
      Ifdef name
    }
| "ifndef" space+ (symbol (*as name*))
    { let name = symbol_after_space lexbuf in
      space_or_comment_and_newline lexbuf;
      Ifndef name
    }
```

```
| "ifdef" { error "syntax in #ifdef" }
| "ifndef" { error "syntax in #ifndef" }

| "else"
    { space_or_comment_and_newline lexbuf;
      Else
    }
| "endif"
    { space_or_comment_and_newline lexbuf;
      Endif
    }

(* stricter: I impose a filename (with no quote in name, like original?) *)
| "line" space+ (digit+ (*as s1*)) space* ('"' ([^'"']* (*as s2*)) '"')
    {
      let (s1, s2) =
         let s = Lexing.lexeme lexbuf in s =~ ".*[ \t]+\\([0-9]+\\).*\"(\\(.*\\)\"" |> ignore;
         Regexp_.matched2 s in
      space_or_comment_and_newline lexbuf;
      Line (int_of_string s1, Fpath.v s2) }
| "line" { error "syntax in #line" }


| "pragma" space+ "lib" space* '"' ([^'"''\n']+ (*as file*)) '"'
    { let file =
        let s = Lexing.lexeme lexbuf in s =~ ".*\"\\(.*\\)\"" |> ignore;
        Regexp_.matched1 s in
      space_or_comment_and_newline lexbuf;
      Pragma("lib", [file])
    }
| "pragma" space+ "src" space* '"' ([^'"''\n']+ (*as file*)) '"'
    { let file =
        let s = Lexing.lexeme lexbuf in s =~ ".*\"\\(.*\\)\"" |> ignore;
        Regexp_.matched1 s in
      space_or_comment_and_newline lexbuf;
      Pragma("src", [file])
    }

| "pragma" space+ "varargck" space* "argpos" [^'\n']+
    { space_or_comment_and_newline lexbuf;
      Pragma("varargck", ["TODO"])
    }
| "pragma" space+ "varargck" space* "type" [^'\n']+
    { space_or_comment_and_newline lexbuf;
      Pragma("varargck", ["TODO"])
    }
| "pragma" space+ "varargck" space* "flag" [^'\n']+
    { space_or_comment_and_newline lexbuf;
      Pragma("varargck", ["TODO"])
    }

| "pragma" space+ "incomplete" space+ (symbol (*as name*))
    { let name =
        let s = Lexing.lexeme lexbuf in s =~ ".*[ \t]+\\([^ \t]+\\)" |> ignore;
        Regexp_.matched1 s in
      space_or_comment_and_newline lexbuf;
      Pragma("incomplete", [name])
    }
(* stricter: we do not silently skip unknown pragmas *)
| "pragma"  { error "syntax in #pragma" }
```

```
  | symbol { error (spf "unknown #: %s" (Lexing.lexeme lexbuf)) }

(*****************************************************************************)
(* Macro definition part 2, the body *)
(*****************************************************************************)
and define_body name params = parse
  | symbol (*as s *)
     { let s = Lexing.lexeme lexbuf in
       try
        let i = List.assoc s params in
        (* safe to use # for a special mark since C code can not
         * use this symbol since it is reserved by cpp
         *)
        spf "#%d" i ^ define_body name params lexbuf
      with Not_found ->
        s ^ define_body name params lexbuf
     }

  | [^ '\n' '_''a'-'z''A'-'Z' '\'' '"' '\\' '/' '#']+ (*as s *)
      { let s = Lexing.lexeme lexbuf in
        s ^ define_body name params lexbuf }

  (* end of macro *)
  | '\n' { incr Location_cpp.line; "" }

  (* special cases *)
  | ['\'' '"'] (*as c *)
      { let c = char lexbuf in
        let s = define_body_strchar c name params lexbuf in
        String.make 1 c ^ s ^ define_body name params lexbuf
      }

  | "//" [^'\n']* { define_body name params lexbuf }
  | "/*"
      { comment_star_no_newline lexbuf;
        define_body name params lexbuf
      }

  | '/' { "/" ^ define_body name params lexbuf }
  (* we escape single '#' by doubling it (classic) *)
  | '#' { "##" ^ define_body name params lexbuf }

  (* could do " " ^ but that is not what 5c does *)
  | '\\' '\n' { incr Location_cpp.line; define_body name params lexbuf }
  | '\\' { "\\" ^ define_body name params lexbuf }

  | eof  { error (spf "eof in macro %s" name) }

(*****************************************************************************)
(* Strings and characters (part1) *)
(*****************************************************************************)

(* Diff with string/character handling in lexer.mll for C?
 *  - need to recognize macro parameter (yes 5c does that! no need stringify)
 *  - need to escape '#' to ##
 *  - do not care about precise parsing of \\, octal, escaped char,
 *    just return the string
*)
```

```
and define_body_strchar endchar name params = parse
  (* no need for stringify! substitute also in strings *)
  | symbol (*as s*)
     { let s = Lexing.lexeme lexbuf in
       try let i = List.assoc s params in
        spf "#%d" i ^ define_body_strchar endchar name params lexbuf
       with Not_found -> s ^ define_body_strchar endchar name params lexbuf
     }
  | [^ '\n' '_' 'a'-'z''A'-'Z' '\'' '"' '\\' '#']+ (*as s *)
      { let s = Lexing.lexeme lexbuf in
        s ^ define_body_strchar endchar name params lexbuf }

  | '\\' '\n'
      { incr Location_cpp.line; define_body_strchar endchar name params lexbuf }
  | '\\' _ (*as s*)
       { let s = Lexing.lexeme lexbuf in
         s^define_body_strchar endchar name params lexbuf }

  (* escape # to disambiguate with use of # to reference a parameter *)
  | '#' { "##" ^ define_body_strchar endchar name params lexbuf }

  | ['\'' '"'] (*as c*)
      { let c = char lexbuf in
        if c = endchar
        then String.make 1 c
        else String.make 1 c ^ define_body_strchar endchar name params lexbuf
      }

  | '\n' { error (spf "newline in character or string in macro %s" name) }
  | eof  { error (spf "eof in macro %s" name) }

(*****************************************************************************)
(* Macro use *)
(*****************************************************************************)

(* Extracting the arguments.
 * Note that as opposed to the other rules in this file, here
 * we do not parse a directive; we parse C code used as arguments to
 * a macro. We still use ocamllex for that because it is convenient.
 *)
and macro_arguments = parse
 | space* "("
     { let xs = macro_args 0 "" [] lexbuf in
       let xs = List.rev xs in
       if xs = [""] then [] else xs
     }
 (* stricter: better error message *)
 | _ (*as c*) { let c = char lexbuf in
                error (spf "was expecting a '(' not %c for macro arguments" c) }
 | eof    { error "was expecting a '(', not an eof for macro arguments" }

and macro_args depth str args = parse
 | ")"
     { if depth = 0
       then str::args
       else macro_args (depth - 1) (str ^ ")") args lexbuf
     }

 | [^ '\'' '"' '/' ',' '\n' '(' ')']+
     { macro_args depth (str ^ Lexing.lexeme lexbuf) args lexbuf }
```

```
 | ","
     { if depth = 0
       (* todo: if reached varargs! *)
       then macro_args 0 "" (str::args) lexbuf
       else macro_args depth (str ^ ",") args lexbuf
     }

 | "("  { macro_args (depth+1) (str^"(") args lexbuf }

  | ['\'' '"'] (*as c*)
      { let c = char lexbuf in
        let s = macro_args_strchar c lexbuf in
        macro_args depth (str ^ String.make 1 c ^ s) args lexbuf
      }

 | "//" [^ '\n']* { macro_args depth str args lexbuf }
 | "/*"
     { comment_star_newline_ok lexbuf;
       macro_args depth (str^" ") args lexbuf
     }
 | "/"  { macro_args depth     (str^"/") args lexbuf }

 | '\n' { incr Location_cpp.line; macro_args depth     (str^" ") args lexbuf }

 | _ (*as c*) { let c = char lexbuf in error (spf "unrecognized character: '%c'" c) }
 (* stricter: better error message *)
 | eof { error "eof in macro arguments" }

(* Substituting the arguments.
 * Note that the lexbuf here will be different. It will correspond
 * to the string of the body of a macro.
 *)
and subst_args_in_macro_body name args = parse
 | [^ '\n' '#']+ (*as s*) {
       let s = Lexing.lexeme lexbuf in
       s ^ subst_args_in_macro_body name args lexbuf }
 | "##" { "#" ^ subst_args_in_macro_body name args lexbuf }
 | "#" (digit+ (*as s*))
     { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
       let i = int_of_string s in
       try
         let arg = List.nth args (i - 1) in
         arg ^ subst_args_in_macro_body name args lexbuf
       with Failure _ ->
         (* stricter: better error message *)
         error (spf "could not find argument %d in macro of %s" i name)
     }
 (* the escaped newlines should have been removed *)
 | '\n' { failwith "impossible: newline in macro body" }
 | eof { "" }


(*****************************************************************************)
(* Strings and characters (part1) *)
(*****************************************************************************)
and macro_args_strchar endchar = parse
  | [^ '\n' '\'' '"' '\\' ]+ (*as s*)
      { let s = Lexing.lexeme lexbuf in
        s ^ macro_args_strchar endchar lexbuf }
```

```
  | '\\' '\n' { incr Location_cpp.line; macro_args_strchar endchar lexbuf }
  | '\\' _ (*as s*)
        { let s = Lexing.lexeme lexbuf in
          s^macro_args_strchar endchar lexbuf }

  | ['\'' '"'] (*as c*)
      { let c = char lexbuf in
        if c = endchar
        then String.make 1 c
        else String.make 1 c ^ macro_args_strchar endchar lexbuf
      }
  | '\n' { error (spf "newline in character or string") }
  | eof  { error (spf "eof in character or string in macro argument") }


(*****************************************************************************)
(* Comments *)
(*****************************************************************************)

and space_or_comment_and_newline = parse
  | space+        { space_or_comment_and_newline lexbuf }
  | '\n'          { incr Location_cpp.line }

  | "//" [^'\n']* { space_or_comment_and_newline lexbuf }
  | "/*"          { comment_star_no_newline lexbuf;
                    space_or_comment_and_newline lexbuf }

  (* stricter: new error message *)
  | _ (*as c*)    { let c = char lexbuf in
                    error (spf "unexpected character %c after directive" c) }
  | eof           { error "expected newline, not EOF" }

and comment_star_no_newline = parse
  | "*/"          { }
  | [^ '*' '\n']+ { comment_star_no_newline lexbuf }
  | '*'           { comment_star_no_newline lexbuf }

  | '\n'          { error "comment across newline" }
  | eof           { error "eof in comment" }

and comment_star_newline_ok = parse
  | "*/"          { }
  | [^ '*' '\n']+ { comment_star_newline_ok lexbuf }
  | '*'           { comment_star_newline_ok lexbuf }
  | '\n'          { incr Location_cpp.line; comment_star_newline_ok lexbuf }
  | eof           { error "eof in comment" }


(*****************************************************************************)
(* Skipping, for ifdefs *)
(*****************************************************************************)

and skip_for_ifdef depth bol = parse
  | '\n'         { incr Location_cpp.line; skip_for_ifdef depth true lexbuf }
  | [' ' '\t']+  { skip_for_ifdef depth bol lexbuf }
  | [^'#' '\n' ' ' '\t']+ { skip_for_ifdef depth false lexbuf }

  | "#endif"
      { if bol
```

```
        then
          if depth = 0
          (* done! just eat until newline *)
          then space_or_comment_and_newline lexbuf
          else skip_for_ifdef (depth - 1) false lexbuf
        else skip_for_ifdef depth false lexbuf
      }
  | "#else"
      { if bol && depth = 0
        then space_or_comment_and_newline lexbuf
        else skip_for_ifdef depth false lexbuf
      }
  | "#ifdef" | "#ifndef"
      { if bol
        then skip_for_ifdef (depth + 1) false lexbuf
        else skip_for_ifdef depth false lexbuf
      }

  | "#" { skip_for_ifdef depth false lexbuf }

  (* stricter: *)
  | eof { error "eof in #ifdef or #ifndef" }
```

16.4 Parsing

⟨*function* Parse_cpp.parse 130⟩≡ (167b)

```
let parse (caps : < Cap.open_in; ..>) hooks (conf : Preprocessor.conf) (file : Fpath.t) =
  L.history := [];
  L.line := 1;
  Hashtbl.clear hmacros;
  conf.defs |> List.iter define_cmdline_def;

  let chan = CapStdlib.open_in caps !!file in
  Logs.info (fun m -> m "opening %s" !!file);

  L.add_event (L.Include file);
  let lexbuf = Lexing.from_channel chan in
  let stack = ref [(Some chan, lexbuf)] in
  (* less: let push x = check if too deep? *)

  (* for more precise error reporting *)
  let last_ident = ref "" in

  let rec lexfunc () =
    match !stack with
    | (chanopt, lexbuf)::xs ->
        let t = hooks.lexer lexbuf in
        let categ = hooks.category t in

        (match categ with
        | Eof ->
            stack := xs;
            chanopt |> Option.iter (fun chan ->
              close_in chan;
              L.add_event L.Eof;
            );
            lexfunc ()
```

```
| Sharp ->

    (* treating cpp directives *)
    let t = Lexer_cpp.token lexbuf in
    (match t with
    | D.Include (f, system_hdr) ->
        let path = find_include conf f system_hdr in
        (try
          let chan = CapStdlib.open_in caps !!path in
          Logs.info (fun m -> m "opening included file %s" !!path);
          L.add_event (L.Include path);
          let lexbuf = Lexing.from_channel chan in
          (* less:
             if List.length stack > 1000
             then error "macro/io expansion too deep"
          *)
          stack := (Some chan, lexbuf)::!stack;
        with Failure s ->
          error s
        )
    | D.Define macro_ast ->
       (* todo: stricter: forbid s to conflict with C keyboard *)
        define macro_ast
    | D.Undef s ->
        (* stricter: check that was defined *)
        if not (Hashtbl.mem hmacros s)
        then error (spf "macro %s was not defined" s);
        Hashtbl.remove hmacros s
    | D.Line (line, file) ->
        L.add_event (L.Line (line, file));
    (* less: for "lib" should add a L.PragmaLib event? *)
    | D.Pragma _ -> ()
    | D.Ifdef s ->
        if Hashtbl.mem hmacros s
        then ()
        else Lexer_cpp.skip_for_ifdef 0 true lexbuf
    | D.Ifndef s ->
        if not (Hashtbl.mem hmacros s)
        then ()
        else Lexer_cpp.skip_for_ifdef 0 true lexbuf
    | D.Else ->
        Lexer_cpp.skip_for_ifdef 0 true lexbuf
    | D.Endif -> ()
    );
    lexfunc ()

| Ident s ->
    last_ident := s;
    if Hashtbl.mem hmacros s
    then
      let macro = Hashtbl.find hmacros s in
      match macro.m_nbargs with
      | None ->
          let body = macro.m_body in
          if !Flags_cpp.debug_macros
          then Logs.app (fun m -> m "#expand %s %s" s body);
          let lexbuf = Lexing.from_string body in
          stack := (None, lexbuf)::!stack;
          lexfunc ()
```

```
              | Some n ->
                  let args = Lexer_cpp.macro_arguments lexbuf in
                  if List.length args <> n
                  then error (spf "argument mismatch expanding: %s" s)
                  else begin
                    let body = macro.m_body in
                    let lexbuf = Lexing.from_string body in
                    let body =
                      Lexer_cpp.subst_args_in_macro_body s args lexbuf in
                    if !Flags_cpp.debug_macros
                    then Logs.app (fun m -> m "#expand %s %s" s body);
                    let lexbuf = Lexing.from_string body in
                    stack := (None, lexbuf)::!stack;
                    lexfunc ()
                  end
            else t
        | _ -> t
        )
      (* no more stack, could raise Impossible instead? *)
      | [] -> hooks.eof
    in

    (try
      let ast = hooks.parser (fun _lexbuf -> lexfunc ()) lexbuf in
      ast, !L.history
    with Parsing.Parse_error ->
      error ("Syntax error" ^
                (if !last_ident = ""
                 then ""
                 else spf ", last name: %s" !last_ident))
    )
```

⟨*function* Parse_cpp.error 132a⟩≡ (167b)
```
  let error s =
    raise (L.Error (s, !L.line))
```

⟨*function* Parse_cpp.define_cmdline_def 132b⟩≡ (167b)
```
  let define_cmdline_def (k, v) =
    Hashtbl.add hmacros k
      { m_name = k; m_nbargs = None; m_varargs = false; m_body = v; }
```

⟨*signature* Location_cpp.add_event 132c⟩≡ (166a)
```
  (* add to history *)
  val add_event:
    location_event -> unit
```

⟨*signature* Location_cpp.final_loc_of_loc 132d⟩≡ (166a)
```
  (* !uses history! you should avoid this function in a multifile processing
   * context (e.g., in the linker) *)
  val final_loc_of_loc:
      loc -> final_loc
```

⟨*signature* Location_cpp.dump_event 132e⟩≡ (166a)
```
  val dump_event:
    location_event -> unit
```

⟨*exception* Location_cpp.Error 132f⟩≡ (166)
```
  exception Error of string * loc
```

⟨*function* Location_cpp.dump_event 133a⟩≡ (166b)
```
(* for 5c -f
 * alt: just rely on deriving show but kept this for compatibility with kencc
 *)
let dump_event (event : location_event) : unit =
  match event with
  | Include file ->
      Logs.app (fun m -> m "%4d: %s" !line !!file)
  | Line (local_line, file) ->
      Logs.app (fun m -> m "%4d: %s (#line %d)" !line !!file local_line)
  | Eof ->
      Logs.app (fun m -> m "%4d: <pop>" !line)
```

⟨*function* Location_cpp.add_event 133b⟩≡ (166b)
```
let add_event (event : location_event) : unit =
  (* alt: use Logs.debug instead of a flag *)
  if !Flags_cpp.debug_line
  then dump_event event;
  history := {location_event = event; global_line = !line }::!history
```

⟨*function* Location_cpp.final_loc_of_loc 133c⟩≡ (166b)
```
(* 'history' contains the list of location_events in reverse order
 * since we always add an event to the end, for instance:
 * [200; 150; 130; 60; 1]. The first step is to reverse this list:
 * [1; 60; 130; 150; 200]. Then, if we look for information about line 135,
 * we want to stop when we encounter 150,
 * so when lineno < x.global_line below succeed for the first time.
 *)
let final_loc_of_loc (lineno : loc) : final_loc =
  let rec aux (lastfile, lastlineno, lastdelta) stack xs =
    match xs with
    | [] -> lastfile, lineno - lastlineno + lastdelta
    | x::xs ->
      if lineno < x.global_line
      then lastfile, lineno - lastlineno + lastdelta
      else
        (match x.location_event, stack with
        | Eof, (lastfile, _lastlineno, lastdelta)::ys ->
            (* bugfix: wrong!! TODO *)
            aux (lastfile, x.global_line, lastdelta) ys xs
        | Eof, [] ->
            failwith ("impossible: wrong location history, unpaired Eof")
        | Include file, ys ->
            aux (file, x.global_line, 1)
              ((lastfile, lastlineno, lastdelta)::ys) xs
        | Line (line, file), _y::ys ->
            aux (file, x.global_line, line) ys xs
        | Line (line, file), [] ->
            (* todo: wrong *)
            aux (file, x.global_line, line) [] xs
        )
  in
  aux (Fpath.v "<nofile>", 0, 0) [] (List.rev !history)
```

⟨*function* Location_cpp._final_loc_and_includers_of_loc 133d⟩≡ (166b)
```
let _final_loc_and_includers_of_loc _lineno =
  raise Todo
```

16.5 #include

16.5.1 Include paths, -I

⟨CLI.main() *macroprocessor locals* 134a⟩≡ (25d) 134e▷
```
(* for cpp *)
let include_paths : Fpath.t list ref = ref [] in
```

⟨CLI.main() options *elements* 134b⟩+≡ (25d) ◁120c 135a▷
```
"-I", Arg.String (fun s ->
  include_paths := Fpath.v s::!include_paths
), " <dir> add dir as a path to look for '#include <file>' files";
```

16.5.2 Tracing origin

16.5.3 #include

⟨*constant* Parse_cpp._cwd 134c⟩≡ (167b)
```
(* cwd is used to manage #include "...". It is altered when you
 * include a file. cwd becomes the dirname of the included file???
 * TODO: dead? used?
 *)
let _cwd = ref (Sys.getcwd ())
```

⟨*function* Parse_cpp.find_include 134d⟩≡ (167b)
```
(* less: Could use Set instead of list for the set of include paths
 * todo: if absolute path, need to find it.
*)
let rec find_include (conf : Preprocessor.conf) (f : Fpath.t) (system : bool) =
  if system
  then find_include_bis conf.paths f
  else find_include_bis (conf.dir_source_file::conf.paths) f
and find_include_bis (paths : Fpath.t list) (f : Fpath.t) : Fpath.t =
  match paths with
  (* stricter: better error message *)
  | [] -> failwith (spf "could not find %s in include paths" !!f)
  | x::xs ->
      let path : Fpath.t =
        if !!x = "."
        (* this will handle also absolute path *)
        then f
        else x // f
      in
      if Sys.file_exists !!path
      then begin
        if !Flags.debug_include
        then Logs.app (fun m -> m "%d: %s" !L.line !!path);
        path
      end
      else find_include_bis xs f
```

16.6 #define

16.6.1 -D

⟨CLI.main() *macroprocessor locals* 134e⟩+≡ (25d) ◁134a
```
let macro_defs = ref [] in
```

⟨CLI.main() options *elements* 135a⟩+≡ (25d) ◁134b 140a▷
```
  "-D", Arg.String (fun s ->
    let (var, val_) =
      if s =~ "\\(.*\\)=\\(.*\\)"
      then Regexp_.matched2 s
      else (s, "1")
    in
    macro_defs := (var, val_)::!macro_defs
  ), " <name=def> (or just <name>) define name for preprocessor";
```

16.6.2 #define

⟨*function* Parse_cpp.define 135b⟩≡ (167b)
```
  let define (macro : D.macro) =
    let {D.name = s; params; varargs; body} = macro in
    (* We could forbid here 's' to conflict with C keyboard, but this
     * should be done in the caller, as cpp can be used with different
     * languages, which may use different keywords.
     *)
    let sbody = match body with Some x -> x | None -> "1" in
    if Hashtbl.mem hmacros s
    then error (spf "macro redefined: %s" s)
    else begin
      let macro =
        match params with
        | None ->
            { m_name = s; m_nbargs = None; m_varargs = false; m_body = sbody }
        | Some params ->
            { m_name = s; m_nbargs = Some (List.length params);
              m_varargs = varargs; m_body = sbody }
      in
      if !Flags_cpp.debug_macros
      then Logs.app (fun m -> m "#define %s %s" s macro.m_body);

      Hashtbl.add hmacros s macro
    end
```

16.6.3 Macro Expansion

⟨*type* Parse_cpp.macro 135c⟩≡ (167b)
```
  (* similar to Ast_cpp.macro *)
  type macro = {
    m_name: string;
    m_nbargs: int option;
    m_varargs: bool; (* use "..." *)

    (* body contains #xxx substrings corresponding to the parameter of the macro.
     * For instance, #define foo(a,b) a+b --> {name="foo";nbargs=2;body="#1+#2"}.
     *
     * Is there a risk of having numbers squashed with the macro parameter?
     * No, because if you have 'a1+b' then 'a1' is a separate identifier
     * so you can not generate #11+#2 .
     *)
    m_body: string;
  }
```

⟨*constant* Parse_cpp.hmacros 135d⟩≡ (167b)
```
  let hmacros = Hashtbl.create 101
```

16.7 #undef

16.8 #ifdef

16.9 #pragma

16.10 #line

Chapter 17

Advanced Topics TODO

17.1 Advanced C features

17.1.1 Function and function pointer automatic conversions

17.1.2 Array and pointer automatic conversions

17.1.3 Local `static`

17.1.4 Local `volatile`

17.1.5 Bitfields

17.1.6 Old style prototypes

17.1.7 `_Noreturn()`

17.2 C extensions

17.2.1 Typing extensions

17.2.2 Unnamed structure elements

17.2.3 Struct constructors

17.2.4 Per-processor storage: `extern register`

17.2.5 Type reflection: `typestr()`

17.2.6 Signature reflection: `signof()`

17.2.7 Format's arguments checking

17.3 Floats

17.4 Big values

17.4.1 Complex types

17.4.2 Complex return value

17.4.3 Complex argument

Chapter 18

Conclusion

Appendix A

Debugging

A.1 Dumpers

⟨CLI.main() options *elements* 140a⟩+≡ (25d) ◁135a 140b▷
```
  (* pad: I added that *)
  "-dump_tokens", Arg.Set Flags.dump_tokens,
  " dump the tokens as they are generated";
```

⟨CLI.main() options *elements* 140b⟩+≡ (25d) ◁140a 140c▷
```
  "-dump_ast", Arg.Set Flags.dump_ast,
  " dump the parsed AST";
```

⟨CLI.main() options *elements* 140c⟩+≡ (25d) ◁140b 140d▷
```
  "-dump_typed_ast", Arg.Set Flags.dump_typed_ast,
  " dump the typed AST";
```

⟨CLI.main() options *elements* 140d⟩+≡ (25d) ◁140c 140g▷
```
  "-dump_asm", Arg.Set Flags.dump_asm,
  " dump the generated assembly";
```

⟨CLI.frontend() *if* dump_ast 140e⟩≡ (28c)
```
  (* debug *)
  if !Flags.dump_ast
  then Logs.app (fun m -> m "%s" (Dumper_.s_of_any (Ast.Program ast)));
```

⟨CLI.main() *other locals* 140f⟩+≡ (25d) ◁120b 141f▷
```
  (* for debugging *)
  let action = ref "" in
```

⟨CLI.main() options *elements* 140g⟩+≡ (25d) ◁140d 142a▷
```
  (* pad: I added that *)
  "-test_parser", Arg.Unit (fun () -> action := "-test_parser"), " ";
```

⟨*constant* Flags.dump_tokens 140h⟩≡ (159c)
```
  let dump_tokens = ref false
```

A.1.1 AST dumper: `5c -x`

⟨*signature* Dumper_.s_of_any 140i⟩≡ (158a)
```
  val s_of_any: Ast.any -> string
```

⟨*function* `Dumper_.s_of_any` 141a⟩≡ (158b)
```
(* less: could have s_of_any_with_pos *)
let s_of_any x =
  Meta_ast.show_types := false;
  Meta_ast.show_all_pos := false;
  let v = Meta_ast.vof_any x in
  OCaml.string_of_v v
```

⟨*type* `Ast.any` 141b⟩≡ (149)
```
(* for visitor and dumper *)
type any =
  | Expr of expr
  | Stmt of stmt
  | Type of typ
  | Toplevel of toplevel
  | Program of program
  | FinalType of Type.t
[@@deriving show { with_path = false } ]
```

⟨*constant* `Flags.dump_ast` 141c⟩≡ (159c)
```
let dump_ast = ref false
```

⟨`CLI.main()` action *management* 141d⟩≡ (25d)
```
(* to test and debug components of mk *)
if !action <> "" then begin
  do_action caps thestring !action (List.rev !args);
  raise (Exit.ExitCode 0 )
end;
```

⟨*function* `CLI.do_action` 141e⟩≡ (153b)
```
let do_action (caps: < caps; .. >) thestring s xs =
  match s with
  | "-test_parser" ->
      xs |> List.iter (fun file ->
        Logs.info (fun m -> m "processing %s" file);
        let conf = Preprocessor.{
          defs = [];
          paths = [spf "/%s/include" thestring; "/sys/include";] |> Fpath_.of_strings;
          dir_source_file = Fpath.v ".";
        }
        in
        try
          let _ = Parse.parse caps conf (Fpath.v file) in
          ()
        with Location_cpp.Error (s, loc) ->
          let (file, line) = Location_cpp.final_loc_of_loc loc in
          failwith (spf "%s:%d %s" !!file line s)
      )

  | _ -> failwith ("action not supported: " ^ s)
```

A.2 `5c -v`, verbose mode

⟨`CLI.main()` *other locals* 141f⟩+≡ (25d) ◁140f 146d▷
```
let level = ref (Some Logs.Warning) in
```

⟨CLI.main() options *elements* 142a⟩+≡ (25d) ◁140g 142c▷
```
"-v", Arg.Unit (fun () -> level := Some Logs.Info),
 " verbose mode";
"-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
" verbose mode";
"-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
" guess what";
"-quiet", Arg.Unit (fun () -> level := None),
" ";
```

⟨CLI.main() *logging setup* 142b⟩≡ (25d)
```
Logs_.setup !level ();
Logs.info (fun m -> m "assembler ran from %s" (Sys.getcwd()));
```

A.3 Preprocessing debugging

⟨CLI.main() options *elements* 142c⟩+≡ (25d) ◁142a 142d▷
```
"-e", Arg.Set Flags_cpp.debug_include, " ";
"-f", Arg.Set Flags_cpp.debug_line, " ";
"-m", Arg.Set Flags_cpp.debug_macros, " ";
```

⟨CLI.main() options *elements* 142d⟩+≡ (25d) ◁142c 144▷
```
(* pad: I added long names for those options *)
"-debug_include", Arg.Set Flags_cpp.debug_include, " ";
"-debug_line",      Arg.Set Flags_cpp.debug_line, " ";
"-debug_macros",    Arg.Set Flags_cpp.debug_macros, " ";
```

A.3.1 Macro debugging: `5c -m`

⟨*constant* Flags_cpp.debug_macros 142e⟩≡ (165b)
```
(* -m *)
let debug_macros = ref false
```

A.3.2 File inclusion debugging: `5c -e`

⟨*constant* Flags_cpp.debug_include 142f⟩≡ (165b)
```
(* -e *)
let debug_include = ref false
```

A.3.3 Line information debugging: `5c -f`

⟨*constant* Flags_cpp.debug_line 142g⟩≡ (165b)
```
(* -f *)
let debug_line = ref false
```

A.4 Parsing debugging

A.4.1 Printing names: `5c -L`

A.4.2 Printing declarations: `5c -d`

A.5 Typing debugging

⟨*constant* Flags.dump_typed_ast 142h⟩≡ (159c)

```
  let dump_typed_ast = ref false
```

⟨*signature* Dumper_.s_of_any_with_types 143a⟩≡ (158a)
```
  val s_of_any_with_types: Ast.any -> string
```

⟨*function* Dumper_.s_of_any_with_types 143b⟩≡ (158b)
```
  let s_of_any_with_types x =
    Meta_ast.show_types := true;
    Meta_ast.show_all_pos := false;
    let v = Meta_ast.vof_any x in
    OCaml.string_of_v v
```

⟨CLI.frontend() *if* dump_typed_ast 143c⟩≡ (28c)
```
  (* debug *)
  if !Flags.dump_typed_ast
  then begin
    (* alt: Logs.app (fun m -> m "%s" (Typecheck.show_typed_program tp));*)
    tp.ids |> Hashtbl.iter (fun (k : Ast.fullname) (v : Typecheck.idinfo) ->
      match v.sto with
      | Storage.Global | Storage.Static | Storage.Extern ->
        Logs.app (fun m -> m "%s: %s" (Ast.unwrap k)
                              (Dumper_.s_of_any (Ast.FinalType v.typ)));
      | Storage.Param | Storage.Local -> ()
    );
    tp.funcs |> List.iter (fun func ->
      Logs.app (fun m -> m "%s"
                (Dumper_.s_of_any_with_types (Ast.Toplevel (Ast.FuncDef func))))
    );
  end;
```

A.5.1 Printing expression type trees: `5c -t`

A.6 Code generation debugging

⟨*constant* Flags.dump_asm 143d⟩≡ (159c)
```
  let dump_asm = ref false
```

⟨CLI.backend5() *if* dump_asm 143e⟩≡ (28d)
```
  (* debug *)
  if !Flags.dump_asm
  then begin
    let pc = ref 0 in
    asm |> List.iter (fun (line, _loc) ->
      (match arch with
      | Arch.Arm ->
        let instr = line in
        (* less: use a assembly pretty printer? easier to debug? 5c -S *)
        let v = Meta_ast_asm5.vof_line instr in
        Logs.app (fun m -> m  "%2d: %s" !pc (OCaml.string_of_v v));
      | Arch.Mips ->
        let instr = Obj.magic line in
        Logs.app (fun m -> m  "%2d: %s" !pc (Ast_asmv.show_line instr));
      | _ ->
       failwith (spf "TODO: arch not supported yet: %s" (Arch.thestring arch))
      );
      incr pc;
    );
  end;
  (* nosemgrep: do-not-use-obj-magic *)
  Obj.magic asm, !Location_cpp.history
```

A.6.1 Printing processed opcodes: `5c -G`

A.6.2 Printing code generation information: `5c -g`

A.6.3 Printing generated assembly: `5c -S`

⟨`CLI.main()` options *elements* 144⟩+≡ (25d) ◁142d 146e▷

```
"-S", Arg.Set Flags.dump_asm,
" dump the generated assembly";
```

A.7 Optimization debugging

A.7.1 Printing arithmetic trees: `5c -m`

A.7.2 Printing registerization: `5c -r`

A.8 Printing initializations: `5c -i`

Appendix B

Error Management

B.1 errorexit()

⟨*signature* Error.errorexit 145a⟩≡ (158c)
```
(* this raises Exit.ExitCode 1 *)
val errorexit: string -> 'a
```

⟨*function* Error.errorexit 145b⟩≡ (158d)
```
let errorexit s =
  Logs.err (fun m -> m "%s" s);
  raise (Exit.ExitCode 1)
```

B.2 Lexing error

B.3 Parsing error

B.4 Typechecking error

⟨*signature* Typecheck.string_of_error 145c⟩≡ (162c)
```
val string_of_error: error -> string
```

⟨*function* Typecheck.string_of_error 145d⟩≡ (163b)
```
let string_of_error err =
  Check.string_of_error err
```

B.5 Checking error

⟨*signature* Check.string_of_error 145e⟩≡ (151a)
```
val string_of_error: error -> string
```

⟨*function* Check.string_of_error 145f⟩≡ (151b)
```
let string_of_error err =
  match err with
  | Inconsistent (s1, loc1, s2, loc2) ->
    let (file1, line1) = Location_cpp.final_loc_of_loc loc1 in
    let (file2, line2) = Location_cpp.final_loc_of_loc loc2 in
    spf "%s:%d error: %s\n%s:%d note: %s" !!file1 line1 s1 !!file2 line2 s2
  | Misc (s, loc) ->
    let (file, line) = Location_cpp.final_loc_of_loc loc in
    spf "%s:%d error: %s" !!file line s
```

B.6 Code generation error

⟨*signature* `Codegen.string_of_error` 146a⟩≡ (154a)
```
val string_of_error: error -> string
```

⟨*function* `Codegen.string_of_error` 146b⟩≡ (154c)
```
let string_of_error err =
  Check.string_of_error err
```

B.7 Backtraces

⟨*constant* `Flags.backtrace` 146c⟩≡ (159c)
```
let backtrace = ref false
```

⟨`CLI.main()` *other locals* 146d⟩+≡ (25d) ◁141f
```
let backtrace = ref false in
```

⟨`CLI.main()` `options` *elements* 146e⟩+≡ (25d) ◁144
```
(* pad: I added that *)
"-backtrace", Arg.Set backtrace,
" dump the backtrace after an error";
```

⟨`CLI.main()` *when* `exn` *if* `backtrace` 146f⟩≡ (27c)
```
if !backtrace
then raise exn
```

B.8 Error location

Appendix C

Extra Code

C.1 Arch_compiler.ml

⟨Arch_compiler.ml 147a⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

(* operand *)
type opd =
 { opd: operand_kind;
   typ: Type.t;
   loc: Ast.loc;
 }
and operand_kind =
 | ConstI of Ast_asm.integer
 | Register of Ast_asm.register

 (* indirect *)
 | Name of Ast.fullname * Ast_asm.offset
 | Indirect of Ast_asm.register * Ast_asm.offset

 (* was not "addressable" in original 5c, but I think it should *)
 | Addr of Ast.fullname

⟨type Arch_compiler.env 93b⟩
⟨type Arch_compiler.t 93a⟩
```

C.2 Arch5.mli

⟨Arch5.mli 147b⟩≡
```
⟨signature Arch5.arch 94a⟩
```

C.3 Arch5.ml

⟨Arch5.ml 147c⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Arch_compiler
module C = Ast
module A = Ast_asm
module T = Type
```

```
module A5 = Ast_asm5

⟨function Arch5.width_of_type 93c⟩

(* opti: let rARGopt = Some (A.R 0) *)

(* for 'extern register xx;', used in ARM kernel *)
let rEXT1 = A.R 10
let rEXT2 = A.R 9

let regs_initial =
  let arr = Array.make A5.nb_registers 0 in
  (* note that rRET is not in the list; it can be used! *)
  [A5.rLINK; A5.rPC;       (* hardware reseved *)
   A5.rTMP; A5.rSB; A5.rSP; (* linker reserved *)
   rEXT1; rEXT2;         (* compiler reserved *)
  ] |> List.iter (fun (A.R x) ->
    arr.(x) <- 1
  );
  arr

(* alt: add binaryOp to Ast_asm.ml so generalize to a few archs *)
let arith_instr_of_op (op : C.binaryOp) r1 r2 r3 =
  A5.Arith (
    (match op with
    | C.Arith op ->
      (match op with
      | C.Plus -> A5.ADD | C.Minus -> A5.SUB
      | C.And -> A5.AND | C.Or -> A5.ORR | C.Xor -> A5.EOR
      (* todo: need type info for A.SLR *)
      | C.ShiftLeft -> A5.SLL | C.ShiftRight -> A5.SRA
      (* todo: need type info for A.MULU, etc *)
      | C.Mul -> A5.MUL | C.Div -> A5.DIV | C.Mod -> A5.MOD
      )
    | C.Logical _ -> raise Todo
    ),
    None,
    A5.Reg r1, Some r2, r3
  ), A5.AL

(* 5c: part of naddr()
 * less: opportunity for bitshifted registers? *)
let mov_operand_of_opd entity_of_id (opd : opd) : A5.mov_operand =
  match opd.opd with
  | ConstI i   -> A5.Imsr (A5.Imm i)
  | Register r -> A5.Imsr (A5.Reg r)
  | Name (fullname, offset) -> A5.Entity (entity_of_id fullname offset)
  | Indirect (r, offset) -> A5.Indirect (r, offset)
  | Addr fullname -> A5.Ximm (A.Address (entity_of_id fullname 0))

let move_instr_of_opds entity_of_id (move_size : A.move_size)
     (opd1: opd) (opd2: opd) : A5.instr_with_cond =
  A5.MOVE (move_size, None,
                      mov_operand_of_opd entity_of_id opd1,
                      mov_operand_of_opd entity_of_id opd2), A5.AL

⟨constant Arch5.arch 94b⟩
```

C.4 Ast.ml

⟨Ast.ml 149⟩≡

```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An Abstract Syntax Tree (AST) for C.
 *
 * The AST below does not match exactly the source code; I do a few
 * simplications at parsing time:
 *  - no nested struct definitions; they are lifted to the toplevel and
 *    a blockid is associated with the tag name to avoid name conflicts
 *  - no anonymous structure; an artificial name is gensym'ed.
 *  - no anonymous structure element; an artificial field name is gensym'ed
 *  - no mix of typedefs with variable declarations;
 *    again typedefs are lifted to the top
 *  - enums are also lifted to the top (and its constants are tagged with
 *    a blockid)
 *
 * This AST is actually more a named AST (but not a typed AST).
 * Indeed, in C, you can not separate completely the naming phase from parsing.
 * The grammar of C has an ambiguity with typedefs, so we need to keep track of
 * typedefs and identifiers and their scope during parsing. It would be
 * redundant to do this work again in a separate naming phase, so I
 * name and resolve the scope of identifiers at parsing time
 * (however I check for inconsistencies or redefinitions after parsing).
 * Moreover, because I lift up struct definitions, I also keep track
 * and resolve the scope of tags.
 *
 * See also pfff/lang_c/parsing/ast_c.ml and pfff/lang_cpp/parsing/ast_cpp.ml
 *
 * todo:
 *  - have a (large) integer and (large) double for Int and Float?
 *    Int64.t? if unsigned long long constant? enough? overflow in int64_of_str?
 *)

(*****************************************************************************)
(* The AST related types *)
(*****************************************************************************)

⟨type Ast.loc 16b⟩
[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Name *)
(* ------------------------------------------------------------------------ *)

⟨type Ast.name 16c⟩
[@@deriving show]

⟨type Ast.blockid 16d⟩
[@@deriving show]

⟨type Ast.fullname 16e⟩
[@@deriving show]

⟨type Ast.idkind 16f⟩
```

⟨*type* Ast.tagkind 16h⟩

```
(* ------------------------------------------------------------------------ *)
(* Types *)
(* ------------------------------------------------------------------------ *)
```
⟨*type* Ast.typ 17a⟩
⟨*type* Ast.type_bis 17b⟩

⟨*type* Ast.function_type 21a⟩

⟨*type* Ast.parameter 21b⟩

```
(* ------------------------------------------------------------------------ *)
(* Expression *)
(* ------------------------------------------------------------------------ *)
```
⟨*type* Ast.expr 17e⟩
⟨*type* Ast.expr_bis 17f⟩

⟨*type* Ast.argument 18a⟩

⟨*type* Ast.const_expr 18b⟩

⟨*type* Ast.unaryOp 18c⟩
⟨*type* Ast.assignOp 18d⟩
⟨*type* Ast.fixOp 19a⟩

⟨*type* Ast.binaryOp 19b⟩
⟨*type* Ast.arithOp 19c⟩
⟨*type* Ast.logicalOp 19d⟩
```
[@@deriving show { with_path = false }]
```

```
(* ------------------------------------------------------------------------ *)
(* Statement *)
(* ------------------------------------------------------------------------ *)
```
⟨*type* Ast.stmt 19e⟩
⟨*type* Ast.stmt_bis 19f⟩

⟨*type* Ast.case_list 20a⟩

```
(* ------------------------------------------------------------------------ *)
(* Variables *)
(* ------------------------------------------------------------------------ *)
```

⟨*type* Ast.var_decl 20e⟩
⟨*type* Ast.initialiser 20f⟩
```
[@@deriving show { with_path = false }]
```

```
(* ------------------------------------------------------------------------ *)
(* Definitions *)
(* ------------------------------------------------------------------------ *)
```
⟨*type* Ast.func_def 20i⟩
```
[@@deriving show { with_path = false } ]
```

⟨*type* Ast.struct_def 21c⟩
⟨*type* Ast.field_def 21d⟩
```
[@@deriving show { with_path = false } ]
```

⟨*type* Ast.enum_def 21e⟩
⟨*type* Ast.enum_constant 21f⟩

```
  [@@deriving show { with_path = false }]

  ⟨type Ast.type_def 22a⟩

  (* ------------------------------------------------------------------------ *)
  (* Program *)
  (* ------------------------------------------------------------------------ *)
  ⟨type Ast.toplevel 20b⟩
  [@@deriving show { with_path = false }]

  ⟨type Ast.toplevels 20c⟩
  [@@deriving show]

  ⟨type Ast.program 20d⟩
  [@@deriving show]

  (* ------------------------------------------------------------------------ *)
  (* Any *)
  (* ------------------------------------------------------------------------ *)
  ⟨type Ast.any 141b⟩

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)
  ⟨function Ast.tagkind_of_su 17d⟩

  ⟨function Ast.unwrap 64a⟩

  open Common
  open Regexp_.Operators
  ⟨function Ast.is_gensymed 79d⟩
```

C.5 Check.mli

⟨Check.mli 151a⟩≡

```
  ⟨type Check.error 58c⟩

  ⟨signature Check.string_of_error 145e⟩

  ⟨exception Check.Error 58b⟩
  ⟨signature Check.failhard 58e⟩

  ⟨signature Check.check_program 58a⟩
```

C.6 Check.ml

⟨Check.ml 151b⟩≡

```
  (* Copyright 2016 Yoann Padioleau, see copyright.txt *)
  open Common
  open Fpath_.Operators
  open Either

  open Ast

  (*****************************************************************************)
```

```
(* Prelude *)
(*****************************************************************************)
(* This module makes sure every entity used is defined. In some cases
 * it also checks if an entity is unused, or if it is incorrectly redeclared.
 *
 * For typechecking see Typecheck.ml.
 * For naming and scope resolving see Parser.mly.
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Check.usedef 59d⟩

⟨*type* Check.env 59e⟩

⟨*type* Check.error 58c⟩

⟨*function* Check.string_of_error 145f⟩

⟨*exception* Check.Error 58b⟩

⟨*constant* Check.failhard 58f⟩

⟨*function* Check.error 58d⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```
⟨*function* Check.inconsistent_tag 65d⟩

⟨*function* Check.check_inconsistent_or_redefined_tag 65c⟩

⟨*function* Check.inconsistent_id 66b⟩

⟨*function* Check.check_inconsistent_or_redefined_id 66a⟩

⟨*function* Check.check_unused_locals 66c⟩

⟨*function* Check.check_labels 67c⟩

```
(*****************************************************************************)
(* Use/Def *)
(*****************************************************************************)
```
⟨*function* Check.check_usedef 59g⟩

```
(*****************************************************************************)
(* Unreachable code *)
(*****************************************************************************)
(* TODO *)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Check.check_program 58g⟩

C.7 CLI.mli

⟨CLI.mli 153a⟩≡
⟨*type* CLI.caps 25c⟩

⟨*signature* CLI.main 25b⟩

⟨*signature* CLI.compile 26c⟩

C.8 CLI.ml

⟨CLI.ml 153b⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators
open Regexp_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of 5c/vc, the Plan 9 C compiler for ARM and MIPS.
 *
 * Main limitations compared to 5c/vc/...:
 *  - no unicode support
 *  - can not compile multiple files at the same time
 *    (but you should use mk anyway)
 *  - can not compile from stdin
 *    (but who uses that?)
 *  - no -. to remove auto search for header in current directory
 *    (but who uses that?)
 *  - no error recovery, we stop at the first error (except in check.ml)
 *    (but compiler now fast enough and errors have a domino effect anyway)
 *  - no support for certain kencc extensions:
 *     * unnamed structure element
 *       (confusing anyway, and annoying when porting code to gcc/clang)
 *     * typestr
 *       (seems dead)
 *  - no support for certain C features:
 *     * enum float
 *       (who uses that?)
 *
 * stricter:
 *  - stricter for grammar (see parser.mly), for instance force a specific
 *    order between the sign, qualifier, and type.
 *  - disallow implicit declarations of functions
 *  - stricter for typechecking (see typecheck.ml), for instance
 *    we do not support void* conversions (5c -V), and we use name
 *    equality for typechecking structs, not field equality.
 *    we also do not automatically transform 0 in nil; I force to write
 *    nil
 *
 * improvements:
 *  - we forbid more constructs:
 *     * typedef and initializers,
 *     * typedef function definitions,
 *     * three dots parameter in the middle,
 *     * far more (see tests/)
 *  - better error location (no use of vague nearln) and
```

```
 *    better error messages (a la clang)
 *
 * todo:
 *  - finish enough to at least handle helloc.c (basic function calls, basic
 *    types)
 *  - safe-linking support
 *  - debugger support
 *  - profiler support
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)
⟨type CLI.caps 25c⟩

(*****************************************************************************)
(* Testing *)
(*****************************************************************************)

⟨function CLI.do_action 141e⟩

(*****************************************************************************)
(* Main algorithms *)
(*****************************************************************************)

⟨function CLI.frontend 28c⟩

⟨function CLI.backend5 28d⟩
⟨function CLI.compile5 28b⟩

⟨function CLI.compile 28a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function CLI.main 25d⟩
```

C.9 Codegen.mli

⟨Codegen.mli 154a⟩≡

```
⟨type Codegen.error 95c⟩
⟨signature Codegen.string_of_error 146a⟩
⟨exception Codegen.Error 95d⟩

⟨signature Codegen.codegen 28e⟩
```

C.10 Codegen.ml

⟨type Codegen.integer 154b⟩≡ (154c)

⟨Codegen.ml 154c⟩≡

```
(* Copyright 2016, 2017, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Eq.Operators
open Either
```

```
open Arch_compiler
open Ast_asm
open Ast
module C = Ast
module A = Ast_asm
(* can't depend on Ast_Asm5 or Ast_asmv here! the code needs to
 * work for all archs!
 *)

module T = Type
module S = Storage
module TC = Typecheck
module E = Check

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * TODO:
 *   - fields, structures
 * LATER:
 *   - firstarg opti
 *   - alignment
 *     * fields (sualign)
 *     * parameters
 *   - other integer types and cast
 *   - float
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```

⟨*type* Codegen.env 94c⟩

⟨*constant* Codegen.rRET 103b⟩
⟨*constant* Codegen.rEXT1 98c⟩
⟨*constant* Codegen.rEXT2 98d⟩

⟨*constant* Codegen.regs_initial 98b⟩

⟨*function* Codegen.env_of_tp 95b⟩

⟨*type* Codegen.integer 154b⟩

⟨*type* Codegen.operand_able 99a⟩
⟨*type* Codegen.operand_able_kind 99b⟩

⟨*type* Codegen.error 95c⟩
⟨*function* Codegen.string_of_error 146b⟩
⟨*exception* Codegen.Error 95d⟩

```
(*****************************************************************************)
(* Instructions  *)
(*****************************************************************************)
```
⟨*constant* Codegen.fake_instr 96e⟩
⟨*constant* Codegen.fake_loc 96f⟩
⟨*constant* Codegen.fake_pc 97a⟩

⟨*function* Codegen.add_instr 96d⟩

⟨*function* Codegen.set_instr 96c⟩

⟨*function* Codegen.add_fake_instr 96b⟩

⟨*function* Codegen.add_fake_goto 100c⟩

⟨*function* Codegen.patch_fake_goto 100d⟩

```
(*****************************************************************************)
(* C to Asm helpers  *)
(*****************************************************************************)
```
⟨*function* Codegen.global_of_id 97b⟩

⟨*function* Codegen.symbol 107c⟩

⟨*function* Codegen.entity_of_id 107b⟩

⟨*function* Codegen.mov_operand_of_opd 107a⟩

```
(* 5c: part of naddr() called from gopcode *)
let branch_operand_of_opd (env : 'i env) (opd : opd) : A.branch_operand2 =
  match opd.opd with
  | Name (fullname, offset) ->
      assert (offset =|= 0);
      A.SymbolJump (global_of_id env fullname)
  | ConstI _ -> raise (Impossible "calling an int can't typecheck")
  | Register _ -> raise (Impossible "calling a register can't typecheck")
  | Indirect (r, offset) ->
      assert (offset =|= 0);
      A.IndirectJump r
  | Addr _ -> raise Todo
```

⟨*function* Codegen.arith_instr_of_op 109a⟩

```
(* 5c: regaalloc but actually didn't allocate reg so was a bad name *)
let argument_operand (env : 'i env) (arg: argument) (curarg : int) : opd =
  (* add 4 for space for REGLINK in callee *)
  { opd = Indirect (env.a.rSP, curarg + 4 (* TODO: SZ_LONG *)) ;
    typ = arg.e_type;
    loc = arg.e_loc;
  }

(*****************************************************************************)
(* Operand able, instruction selection  *)
(*****************************************************************************)
```
⟨*function* Codegen.operand_able 104c⟩

⟨*constant* Codegen.fn_complexity 107e⟩

⟨*function* Codegen.complexity 108a⟩

```
(*****************************************************************************)
(* Register allocation helpers *)
(*****************************************************************************)
```
⟨*function* Codegen.reguse 98e⟩

⟨*function* Codegen.regfree 98f⟩

⟨*function* Codegen.with_reg 98g⟩

⟨*function* Codegen.regalloc 98h⟩

⟨*function* Codegen.opd_regalloc 99c⟩

⟨*function* Codegen.opd_regfree 99d⟩

```
(*****************************************************************************)
(* Code generation helpers *)
(*****************************************************************************)
```
⟨*function* Codegen.gmove 106b⟩
⟨*function* Codegen.gmove_opt 106a⟩

```
(*****************************************************************************)
(* Expression *)
(*****************************************************************************)
```
⟨*function* Codegen.expr 104a⟩
```
and arguments (env : 'i env) (xs : argument list) : unit =
  let curarg = ref 0 in
  xs |> List.iter (fun (arg : argument) ->
      (* TODO: if complex argument type or complex arg *)
      (* TODO: if use REGARG and curarg is 0 *)
      let opd = opd_regalloc env arg.e_type arg.e_loc None in
      expr env arg (Some opd);
      let opd2 = argument_operand env arg !curarg in
      let sizet =
        env.a.width_of_type {Arch_compiler.structs = env.structs_} arg.e_type
      in
      (* TODO: cursafe, curarg align before and after *)
      curarg := !curarg + sizet;
      (* TODO: 5c does gopcode(OAS, tn1, Z, tn2) *)
      gmove env opd opd2;
      opd_regfree env opd
  );
  (* TODO: 4 -> SZ_LONG *)
  env.size_maxargs <- Int_.maxround env.size_maxargs !curarg 4;
  ()
```

⟨*function* Codegen.expr_cond 100e⟩

⟨*function* Codegen.expropt 104b⟩

```
(*****************************************************************************)
(* Statement *)
(*****************************************************************************)
```
⟨*function* Codegen.stmt 99e⟩

```
(*****************************************************************************)
(* Global *)
(*****************************************************************************)
(* TODO, in env.ids ? iter over it? *)

(*****************************************************************************)
(* Function *)
(*****************************************************************************)
```
⟨*function* Codegen.codegen_func 96a⟩

```
(*****************************************************************************)
(* Main entry point *)
(*****************************************************************************)
```
⟨*function* Codegen.codegen 95a⟩

C.11 Dumper_.mli

⟨Dumper_.mli 158a⟩≡

⟨*signature* Dumper_.s_of_any 140i⟩

⟨*signature* Dumper_.s_of_any_with_types 143a⟩

C.12 Dumper_.ml

⟨Dumper_.ml 158b⟩≡

⟨*function* Dumper_.s_of_any 141a⟩

⟨*function* Dumper_.s_of_any_with_types 143b⟩

C.13 Error.mli

⟨Error.mli 158c⟩≡

⟨*signature* Error.warn 58j⟩

⟨*signature* Error.errorexit 145a⟩

C.14 Error.ml

⟨Error.ml 158d⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators
```

⟨*function* Error.warn 59c⟩

⟨*function* Error.errorexit 145b⟩

C.15 Eval_const.mli

⟨Eval_const.mli 158e⟩≡

⟨*exception* Eval_const.NotAConstant 91d⟩

⟨*type* Eval_const.error 91f⟩
⟨*exception* Eval_const.Error 91e⟩

⟨*type* Eval_const.integer 91c⟩

⟨*type* `Eval_const.env` 91b⟩

⟨*signature* `Eval_const.eval` 91a⟩

C.16 `Eval_const.ml`

⟨`Eval_const.ml` 159a⟩≡
```
(* Copyright 2016, 2017 Yoann Padioleau, see copyright.txt *)
open Common

open Ast
module E = Check

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```
⟨*type* `Eval_const.integer` 91c⟩

⟨*type* `Eval_const.env` 91b⟩

⟨*exception* `Eval_const.NotAConstant` 91d⟩

⟨*type* `Eval_const.error` 91f⟩
⟨*exception* `Eval_const.Error` 91e⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* `Eval_const.eval` 91g⟩

C.17 `Flags.ml`

⟨*constant* `Flags.debugger` 159b⟩≡ (159c)
```
let debugger = ref false
```

⟨`Flags.ml` 159c⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
```
⟨*global* `Flags.warn` 58h⟩
⟨*global* `Flags.warnerror` 59a⟩

```
(* see also macroprocessor/flags_cpp.ml *)
```
⟨*constant* `Flags.dump_tokens` 140h⟩
⟨*constant* `Flags.dump_ast` 141c⟩
⟨*constant* `Flags.dump_typed_ast` 142h⟩
⟨*constant* `Flags.dump_asm` 143d⟩

⟨*constant* `Flags.debugger` 159b⟩
⟨*constant* `Flags.backtrace` 146c⟩

C.18 Globals.ml

⟨Globals.ml 160a⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)

(* See also globals in ../macroprocessor/location_cpp.ml  *)

⟨global Globals.hids 16g⟩
```

C.19 Lexer.mll

⟨Lexer helpers 160b⟩≡ (29)
```
⟨function Lexer.error 30b⟩
⟨function Lexer.loc 31a⟩
⟨function Lexer.inttype_of_suffix 34e⟩
⟨function Lexer.floattype_of_suffix 35a⟩
(* dup: lexer_asm5.mll *)
⟨function Lexer.code_of_escape_char 35d⟩
(* dup: lexer_asm5.mll *)
⟨function Lexer.string_of_ascii 36c⟩
⟨function Lexer.char_ 31b⟩
```

C.20 Main.ml

⟨Main.ml 160c⟩≡
```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
open Xix_compiler

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨toplevel Main._1 25a⟩
```

C.21 Parse.mli

⟨Parse.mli 160d⟩≡
```
⟨signature Parse.parse 37a⟩

⟨signature Parse.parse_no_cpp 37b⟩
```

C.22 Parse.ml

⟨Parse.ml 160e⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

module L = Location_cpp
module T = Parser  (* T for Tokens *)

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
```

```
  (*****************************************************************************)
  (* Entry points *)
  (*****************************************************************************)
```

⟨*function* `Parse.parse` 122a⟩

⟨*function* `Parse.parse_no_cpp` 37c⟩

C.23 Parser.mly

⟨*Parser helpers* 161a⟩≡ (37d)
⟨*function* `Parser.error` 38a⟩
⟨*function* `Parser.mk_e` 48g⟩
⟨*function* `Parser.mk_t` 52a⟩
⟨*function* `Parser.mk_st` 45e⟩

⟨*global* `Parser.defs` 40a⟩
⟨*function* `Parser.get_and_reset` 40b⟩

⟨*type* `Parser.env` 38c⟩

⟨*function* `Parser.add_id` 39b⟩
⟨*function* `Parser.add_tag` 39c⟩

⟨*global* `Parser.env` 39a⟩

⟨*global* `Parser.block_counter` 38b⟩

⟨*function* `Parser.gensym` 40c⟩

⟨*function* `Parser.new_scope` 39d⟩
⟨*function* `Parser.pop_scope` 39e⟩

⟨*other stmt related rules* 161b⟩≡ (40d)
⟨*rule* `Parser.tag` 44b⟩
⟨*rule* `Parser.tfor` 46f⟩
⟨*rule* `Parser.forexpr` 46g⟩
⟨*rule* `Parser.labels` 47d⟩
⟨*rule* `Parser.label` 47e⟩

C.24 Rewrite.mli

⟨`Rewrite.mli` 161c⟩≡

```
val rewrite: Typecheck.typed_program -> Typecheck.typed_program
```

C.25 Rewrite.ml

⟨`Rewrite.ml` 161d⟩≡

```
(* todo mandatory:
 *  - pointer arithmetic
 *  - automatic casts
 * todo for opti?:
 *  - OADDR/OIND simplifications
```

```
   *  - put constants on the right for commutative operations
   *)

  let rewrite (tast: Typecheck.typed_program) : Typecheck.typed_program =
    tast
```

C.26 Storage.ml

⟨Storage.ml 162a⟩≡
```
  (* Copyright 2016 Yoann Padioleau, see copyright.txt *)

  ⟨type Storage.t 23g⟩
  [@@deriving show]
```

C.27 Type.ml

⟨Type.ml 162b⟩≡
```
  (* Copyright 2016 Yoann Padioleau, see copyright.txt *)

  (* TODO? rename to Type_.ml because conflict with OCaml5 module name *)

  ⟨type Type.blockid 23e⟩
  [@@deriving show]
  ⟨type Type.fullname 23f⟩
  [@@deriving show]

  ⟨type Type.t 22b⟩

  ⟨type Type.integer_type 22d⟩
  ⟨type Type.integer_kind 22e⟩
  ⟨type Type.sign 22f⟩

  ⟨type Type.float_type 23a⟩

  ⟨type Type.struct_kind 17c⟩
  [@@deriving show { with_path = false }]

  ⟨type Type.qualifier 23b⟩
  [@@deriving show]


  ⟨type Type.structdef 24b⟩
  [@@deriving show]

  ⟨constant Type.int 23c⟩
  ⟨constant Type.long 23d⟩
```

C.28 Typecheck.mli

⟨Typecheck.mli 162c⟩≡
```
  open Common

  ⟨type Typecheck.idinfo 24a⟩
```

```
⟨type Typecheck.typed_program 23h⟩
val show_typed_program: typed_program -> string

⟨type Typecheck.error 69c⟩
⟨signature Typecheck.string_of_error 145c⟩
⟨exception Typecheck.Error 69b⟩

⟨signature Typecheck.check_and_annotate_program 69a⟩
```

C.29 Typecheck.ml

```
⟨type Typecheck.integer 163a⟩≡                                    (163b)
  type integer = int

⟨Typecheck.ml 163b⟩≡
  (* Copyright 2016, 2017 Yoann Padioleau, see copyright.txt *)
  open Common
  open Eq.Operators
  open Either

  open Ast
  module T = Type
  module S = Storage
  module E = Check

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* This module does many things:
   *  - it expands typedefs
   *  - it assigns a final (resolved) type to every identifiers
   *  - it assigns a final storage to every identifiers
   *  - it assigns a type to every expressions
   *  - it returns a typed AST and also transforms this AST
   *    (e.g., enum constants are replaced by their value)
   *
   * Thanks to the naming done in parser.mly and the unambiguous Ast.fullname,
   * we do not have to handle scope here.
   * Thanks to check.ml we do not have to check for inconsistencies or
   * redefinition of tags. We can assume everything is fine.
   *
   * limitations compared to 5c:
   *  - no 'void*' conversions
   *    (clang does not either? but we do accept so void* vs xxx* )
   *  - no struct equality by field equality. I use name equality.
   *    (who uses that anyway?)
   *  - no float enum
   *    (who uses that anyway?)
   *  - no 0 to nil automatic cast inserted
   *    (I prefer stricter typechecking)
   *
   * todo:
   *  - const checking (in check_const.ml?) need types!
   *  - format checking (in check_format.ml?) need types!
   *  - misc checks
   *    * stupid shift bits (need constant evaluation) outside width
   *)
```

```
(*****************************************************************************)
(* Types *)
(*****************************************************************************)

(* less: vlong? *)
```
⟨*type* Typecheck.integer 163a⟩
```

```
⟨*type* Typecheck.idinfo 24a⟩
```
[@@deriving show]

(* for ocaml-light to work without deriving *)
let show_typed_program _ = "NO DERIVING"
[@@warning "-32"]

(* alt: Frontend.result, Frontend.entities, Frontend.t, Typecheck.result *)
```
⟨*type* Typecheck.typed_program 23h⟩
```
[@@deriving show]

```
⟨*type* Typecheck.env 70a⟩
⟨*type* Typecheck.expr_context 70c⟩

(* less: could factorize things in error.ml? *)
⟨*type* Typecheck.error 69c⟩

⟨*function* Typecheck.string_of_error 145d⟩

⟨*exception* Typecheck.Error 69b⟩

⟨*function* Typecheck.type_error 69d⟩

⟨*function* Typecheck.type_error2 69e⟩

(***)
(* Types helpers *)
(***)

⟨*function* Typecheck.same_types 71a⟩

⟨*function* Typecheck.merge_types 71b⟩

(* when processing enumeration constants, we want to keep the biggest type *)
(*
let max_types t1 t2 = ...
*)

⟨*function* Typecheck.check_compatible_binary 74⟩

⟨*function* Typecheck.result_type_binary 75⟩

⟨*function* Typecheck.check_compatible_assign 77a⟩

⟨*function* Typecheck.check_args_vs_params 80b⟩

(***)
(* Storage helpers *)
(***)
⟨*function* Typecheck.merge_storage_toplevel 90⟩

(***)
(* Other helpers *)
```
```

```
  (*****************************************************************************)
  ⟨function Typecheck.lvalue 76b⟩

  ⟨function Typecheck.array_to_pointer 71c⟩

  ⟨function Typecheck.unsugar_anon_structure_element 79c⟩

  (*****************************************************************************)
  (* AST Types to Types.t *)
  (*****************************************************************************)
  ⟨function Typecheck.type_ 86a⟩

  (*****************************************************************************)
  (* Expression typechecking *)
  (*****************************************************************************)
  ⟨function Typecheck.expr 72a⟩
  ⟨function Typecheck.expropt 72b⟩

  (*****************************************************************************)
  (* Statement *)
  (*****************************************************************************)
  ⟨function Typecheck.stmt 82d⟩

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)
  ⟨function Typecheck.check_and_annotate_program 69f⟩
```

C.30 macroprocessor/Ast_cpp.ml

```
⟨macroprocessor/Ast_cpp.ml 165a⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* An Abstract Syntax Tree (AST) for the C Pre-Processor (CPP) directives. *)

  (*****************************************************************************)
  (* The AST *)
  (*****************************************************************************)

  ⟨type Ast_cpp.directive 120d⟩

  ⟨type Ast_cpp.macro 120e⟩
  [@@deriving show]
```

C.31 macroprocessor/Flags_cpp.ml

```
⟨macroprocessor/Flags_cpp.ml 165b⟩≡

  ⟨constant Flags_cpp.debug_line 142g⟩
  ⟨constant Flags_cpp.debug_include 142f⟩
  ⟨constant Flags_cpp.debug_macros 142e⟩
```

C.32 macroprocessor/Lexer_cpp.mll

C.33 macroprocessor/Location_cpp.mli

⟨macroprocessor/Location_cpp.mli 166a⟩≡

```
⟨type Location_cpp.loc 121a⟩
[@@deriving show]
⟨type Location_cpp.final_loc 121b⟩

⟨type Location_cpp.location_history 121c⟩
⟨type Location_cpp.location_event 121d⟩
[@@deriving show]

⟨signature Location_cpp.history 120f⟩
⟨signature Location_cpp.line 120g⟩

⟨exception Location_cpp.Error 132f⟩


⟨signature Location_cpp.add_event 132c⟩

⟨signature Location_cpp.final_loc_of_loc 132d⟩


⟨signature Location_cpp.dump_event 132e⟩
```

C.34 macroprocessor/Location_cpp.ml

⟨macroprocessor/Location_cpp.ml 166b⟩≡

```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(*
 * less: normalize filenames? use realpath?
 *)

(*****************************************************************************)
(* Types and globals *)
(*****************************************************************************)

⟨type Location_cpp.loc 121a⟩
[@@deriving show]

⟨type Location_cpp.final_loc 121b⟩
[@@deriving show]

⟨type Location_cpp.location_history 121c⟩
⟨type Location_cpp.location_event 121d⟩
[@@deriving show {with_path = false}]

(* TODO? move to local in Parse_cpp.ml? *)
⟨constant Location_cpp.history 121e⟩
```

⟨*constant* `Location_cpp.line` 121f⟩

⟨*exception* `Location_cpp.Error` 132f⟩

```
(*****************************************************************************)
(* Debugging *)
(*****************************************************************************)
```

⟨*function* `Location_cpp.dump_event` 133a⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```
⟨*function* `Location_cpp.add_event` 133b⟩

⟨*function* `Location_cpp.final_loc_of_loc` 133c⟩

⟨*function* `Location_cpp._final_loc_and_includers_of_loc` 133d⟩

C.35 macroprocessor/Parse_cpp.mli

⟨`macroprocessor/Parse_cpp.mli` 167a⟩≡

⟨*type* `Parse_cpp.token_category` 121g⟩
`[@@deriving show]`

`(* wrapper around a parser/lexer (e.g., 5c) to preprocess first the file *)`
⟨*type* `Parse_cpp.hook` 121h⟩

⟨*signature* `Parse_cpp.parse` 119a⟩

C.36 macroprocessor/Parse_cpp.ml

⟨`macroprocessor/Parse_cpp.ml` 167b⟩≡

```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

module D = Ast_cpp (* D for Directives *)
module L = Location_cpp
module Flags = Flags_cpp

(* for record matching for ocaml-light *)
open Ast_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Cpp as a library. This is used by 5c but also 5a.
 *
 * Main limitations compared to the cpp embedded in 5c of Plan 9:
 *  - no support for unicode
 *  - see Lexer_cpp.mll
 * Main limitations compared to ANSI cpp:
```

```
 *  - no complex boolean expressions for #ifdefs
 *
 * stricter:
 *  - see Lexer_cpp.mll
 * more general:
 *  - allow any number of arguments for macros
 *    (not limited to 25 because of the use of #a to #z)
 *  - allow any body size
 *    (no 8196 buffer limit)
 *  - allow any filename length in #include
 *    (no 200 limit)
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
⟨type Parse_cpp.token_category 121g⟩
[@@deriving show]

⟨type Parse_cpp.hook 121h⟩

⟨type Parse_cpp.macro 135c⟩

(*****************************************************************************)
(* Globals *)
(*****************************************************************************)
⟨constant Parse_cpp.hmacros 135d⟩

⟨constant Parse_cpp._cwd 134c⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Parse_cpp.error 132a⟩


⟨function Parse_cpp.define_cmdline_def 132b⟩

⟨function Parse_cpp.define 135b⟩

⟨function Parse_cpp.find_include 134d⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Parse_cpp.parse 130⟩
```

C.37 macroprocessor/Preprocessor.ml

⟨macroprocessor/Preprocessor.ml 168⟩≡

```
  (* Copyright 2016 Yoann Padioleau, see copyright.txt *)

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Data structure containing the command-line data that one can pass to cpp
   * (e.g, -D and -I CLI flags).
   * This is passed then to Parse_cpp.parse() in addition to hooks.
```

```
 *)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```
⟨*type* `Preprocessor.conf` 119b⟩
```
[@@deriving show]
```

Glossary

SUE = Structure/Union/Enum
SU = Structure/Union

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 12

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 12