# REPORT
# COMPILER CONSTRUCTION

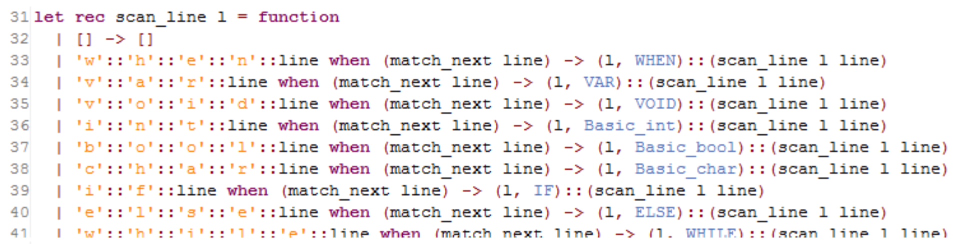
Martin Huyben (s4205332)

Tom Evers (s4205774)

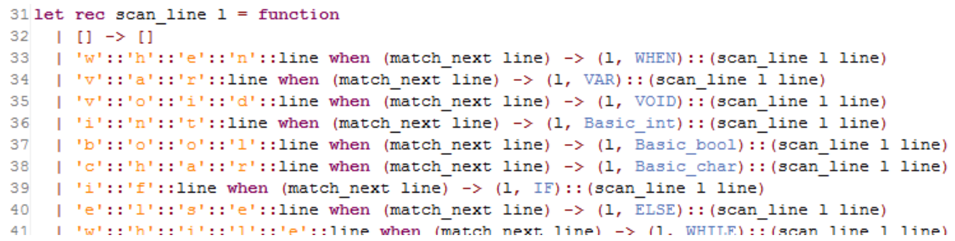
# Lexing and parsing

## tokenizer.ml

This file contains functions for lexing a file. Files are scanned line by line. Every token is accompanied by the number of the line it was found on.



```
let rec scan_line l = function
  | [] -> []
  | 'w'::'h'::'e'::'n'::line when (match_next line) -> (l, WHEN)::(scan_line l line)
  | 'v'::'a'::'r'::line when (match_next line) -> (l, VAR)::(scan_line l line)
  | 'v'::'o'::'i'::'d'::line when (match_next line) -> (l, VOID)::(scan_line l line)
  | 'i'::'n'::'t'::line when (match_next line) -> (l, Basic_int)::(scan_line l line)
  | 'b'::'o'::'o'::'l'::line when (match_next line) -> (l, Basic_bool)::(scan_line l line)
  | 'c'::'h'::'a'::'r'::line when (match_next line) -> (l, Basic_char)::(scan_line l line)
  | 'i'::'f'::line when (match_next line) -> (l, IF)::(scan_line l line)
  | 'e'::'l'::'s'::'e'::line when (match_next line) -> (l, ELSE)::(scan_line l line)
  | 'w'::'h'::'i'::'l'::'e'::line when (match_next line) -> (l, WHILE)::(scan_line l line)
```



```
let rec scan_line l = function
  | [] -> []
  | 'w'::'h'::'e'::'n'::line when (match_next line) -> (l, WHEN)::(scan_line l line)
  | 'v'::'a'::'r'::line when (match_next line) -> (l, VAR)::(scan_line l line)
  | 'v'::'o'::'i'::'d'::line when (match_next line) -> (l, VOID)::(scan_line l line)
  | 'i'::'n'::'t'::line when (match_next line) -> (l, Basic_int)::(scan_line l line)
  | 'b'::'o'::'o'::'l'::line when (match_next line) -> (l, Basic_bool)::(scan_line l line)
  | 'c'::'h'::'a'::'r'::line when (match_next line) -> (l, Basic_char)::(scan_line l line)
  | 'i'::'f'::line when (match_next line) -> (l, IF)::(scan_line l line)
  | 'e'::'l'::'s'::'e'::line when (match_next line) -> (l, ELSE)::(scan_line l line)
  | 'w'::'h'::'i'::'l'::'e'::line when (match_next line) -> (l, WHILE)::(scan_line l line)
```

## parser.ml

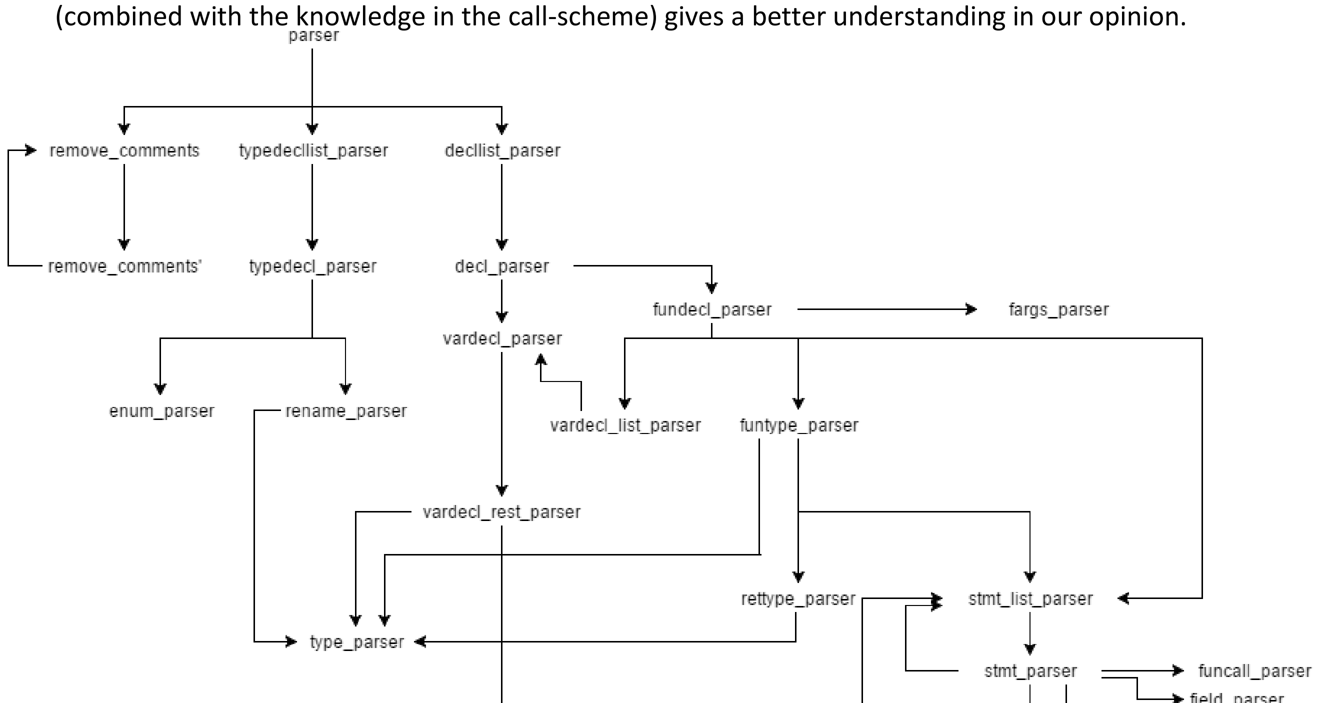
Much more interesting would be the two parser modules, "parser.ml" and "exp_parser.ml". We decided to separate the expression parser from the rest of the parser, since our expression parser is kind of a beast. First, we will cover the normal part of the parser.

Our parser depends heavily on pattern matching, and in most cases works and behaves exactly as one would expect.

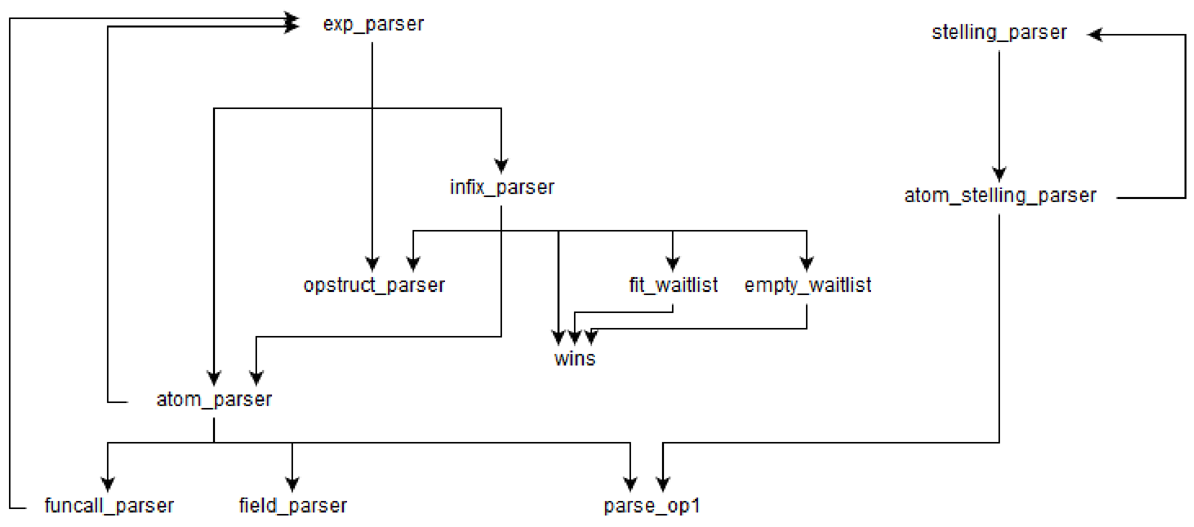
- First, it calls "remove_comments" to clear all comments from the tokenlist
- Then, it calls "typedecllist_parser" to generate a list of self-defined types
- Finally, it calls "decllist_parser" to create the AST

The image below contains a so-called call-scheme. This scheme shows which functions are called by which. This should shed a light on how our parser works, better than a written report could do.

We could spend time trying to describe what each function does in detail, but looking at the code (combined with the knowledge in the call-scheme) gives a better understanding in our opinion.



## exp_parser.ml



This module exists out of two parts: *exp_parser* and *stelling_parser*. A stelling (or clause) is an expression which:

- has no infix operators, except list operator
- every variable can only have one prefix operator
- functions are not allowed
- low bars are allowed

Everything in this module is pretty straightforward, except for the function *infix_parser*. We will examine this function.

### infix_parser

The *infix_parser* makes sure that the order of operators in the AST is correct. For example * is stronger than +, so + should be higher in the AST than *. We call this '+ wins over *'. For this the function uses a waitlist filled with tuples of the form (expression,opstruct). An opstruct is a record containing some information about the operator.
Two invariants hold at all times when *infix_parser* is called:

1. Any opstruct in the waitlist loses from every opstruct in the waitlist with a higher index.
2. Every opstruct in the waitlist wins over the first opstruct argument given to *infix_parser*.

It is best explained how the *infix_parser* works by giving an example.

Given a token list *"a || b / c = d + e"*, where *a* to *e* are atoms, e.g. they can be parsed by the *atom_parser*. We use *"* and *"* to denote that it is a token list and not yet an expression. The *exp_parser* will start parsing this list, find the operator *||*, and call *infix_parser* with the following 5 arguments.
*Infix_parser ([]) (a) (||) (b) ("/ c = d + e").*
The infix operator checks if there is another operator, and finds */*.
It then finds *c*.
*||* wins over */*, so *(a,||)* is put in the waitlist, and the *infix_parser* is called again.
*Infix_parser ([(a,||)]) (b) (/) (c) ("= d + e").*
The *infix_parser* finds = and *d*.
*/* loses from =. So on the right of */* there is an operator from which */* loses, and since every operator in the waitlist wins over */* (as stated by invariant 2), on the left of */* is an operator from which */* loses, *(b/c)* can now be considered atomic; it cannot be split by other operators.

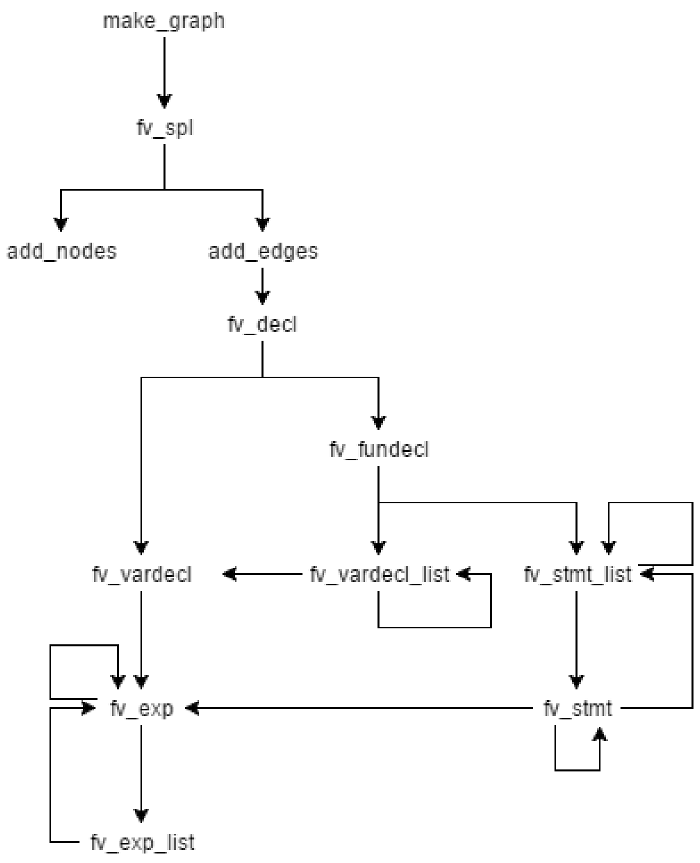
Next the function checks if = wins over the first operator in the waitlist, *||.*
It loses, so our new call is:
*Infix_parser ([a,||]) (b/c) (=) (d) ("+ e").*
The *infix_parser* finds *+* and *e*.
*=* wins over *+*, so *((b/c),=)* is put on the waitlist.
*Infix_parser ([((b/c),=);(a,||)]) (b/c) (+) (e) ("")*.
There are no more operators in the token list, so the waitlist is emptied. *=* wins over *+*, and *||* wins over *=*, as stated by invariant 1.
Our end result is:
*a || ((b/c) = (d+e)).*
And that is how our *infix_parser* works, as it parses from left to right over the token list. The strength of this approach is in its ease to add or change operators, as can be seen in *opstruct_parser*, where for every operator their strength and associativity is given.

# Typechecking

## graph_make.ml

To enable multiple recursion, we had to do some preprocessing to group variable and function declarations in strongly connected components using Tarjan's algorithm. This algorithm itself isn't very interesting, as it is a literal translation from well-known code.

The part of the compiler that creates the nodes and edges is a bit more interesting, though doesn't require much explanation either.



### Make_graph

First, this function explicitly adds "read", "write" and "isEmpty" to the graph. Then, it calls "fv_spl".

### Fv_spl

This function adds all the nodes (which are just all the id's of all the declarations) to the graph. After that, it uses the other "fv"-functions to create edges from a declared function or variable, to all the function or variables it needs.

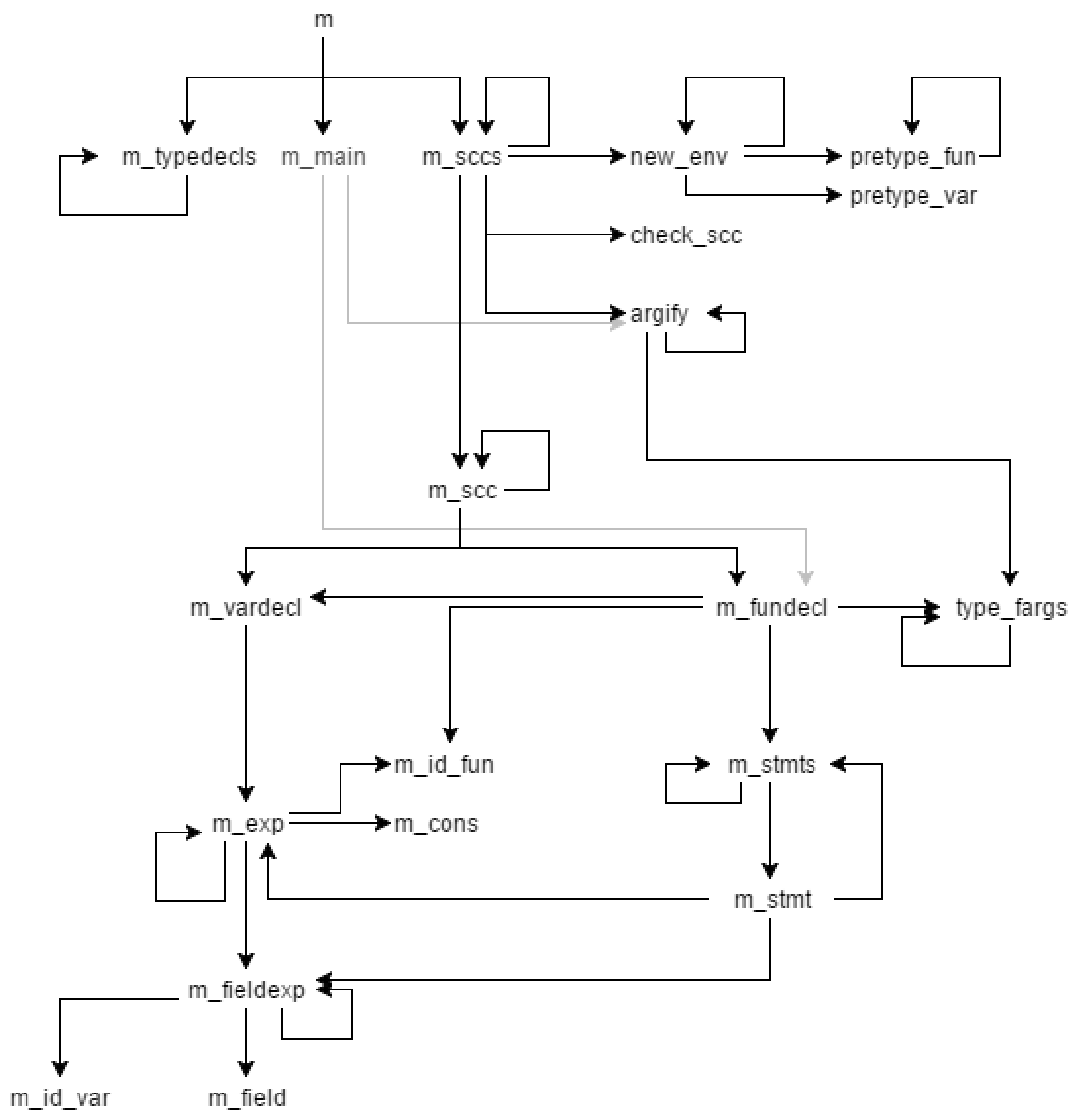
### The other fv-functions

These functions find all the identifiers used by the function or variable, that are not its own name. The exceptions to this:

- Function arguments
- Local variables
- Hyperlocals in match-cases

## typechecker.ml

The call-scheme for our typechecker:



### m

The "m" algorithm first calls "m_typedecls", which reads all the custom type declarations and places them in the environment.

Then it calls the recursive "m_sccs", which starts checking the result of Tarjan's Strongly Connected Component algorithm.

To conclude things, it checks just the main-function with “m_main”.

### m_main

Combination of “m_sccs”, “m_scc” and “m_fundecl”. Checks a few things:

- ‘main’ is a function, not a variable
- ‘main’ has no arguments
- ‘main’ returns an integer

## m_typedecls
See M.

## m_sccs
This function repeats the following actions for each SCC:

- It checks if the component contains interdependent variable declarations with “check_scc”
- It creates a new environment for that specific SCC with “new_env”
- It calls “m_scc” to typecheck the current component in its new environment
- Adds the typed declarations to the old environment, adds arguments to function entries
- It adjusts the function (and function argument) types in the “old” environment according to the result of “m_scc”

## check_scc
Since variables may not be dependent on each other, and a function cannot be dependent on a global variable that is dependent on that function, an SCC that contains a variable cannot contain anything else.

## new_env
Generates a new environment by creating stubs for each function or variable in a Strongly Connected Component. Uses “pretype_fun” and “pretype_var” to take the predefined type into account if necessary.

## pretype_fun/pretype_var
See New_env.

## m_scc
For every declaration within the SCC that it is called with, “m_scc” does the following:

- If it’s a function declaration, call “m_fundecl”
- If it’s a variable declaration, call “m_vardecl” with the type that is found in the environment

## m_fundecl
- Checks for duplicate arguments
- Calls “m_id_fun” to check the environment for the current function
- Gives the arguments the correct types (with “type_fargs”)
- Checks types for all local variables (with “m_vardecl”)
- Calls “m_stmts” with the typed locals added to the environment

## m_vardecl
Just calls “m_exp”, since the variable is already in the environment. This can be caused by the following things:

- If it concerns a global variable, “new_env” did this, way back in “m_sccs”
- If it concerns a local variable, “m_fundecl” did this
- If it concerns a hyperlocal, “m_stmt” did this

The variable’s type merely needs to be unifiable with the one of its expression.

## m_stmts
Recursively checks all statements in a list by calling “m_stmt”. Interesting to note: the only way this can fail, is if one of the statements is something else than a Var. And the only way that can happen, is if it is or contains a return statement.

## m_stmt
Needs “m_fieldexp” for the “Stmt_define” statement.

This function works pretty much as described in the college slides, except for the part that handles the match-statement. This particular statement works like this:

- It checks the type of the match-expression
- Then, for each case:
    - It places the hyperlocal variables from this case in a temporary environment
    - It tries to unify the type of the match-case with the match-expression (“m_exp”)
    - It checks the “when” condition (“m_exp” with as type Bool)
    - Finally, it checks the statementlist with “m_stmts”

## m_exp
Calls “m_id_fun” in the “Exp_function_call” expression.

Needs “m_fieldexp” for the “Exp_field” expression.

The difference with the college slides lies in the expressions that are needed for pattern-matching. These include the “Exp_low_bar” and “Exp_constructor cons” expressions.

- “Exp_low_bar” is treated as a wildcard:
  $\text{m\_exp}\ \Gamma\ (\text{Exp\_low\_bar})\ \sigma = u(\sigma, \alpha)$ with $\alpha$ is fresh
- “Exp_constructor” is handled by “m_cons”:
  $\text{m\_exp}\ \Gamma\ (\text{Exp\_constructor}\ cons)\ \sigma = \text{m\_cons}\ \Gamma\ cons\ \sigma$

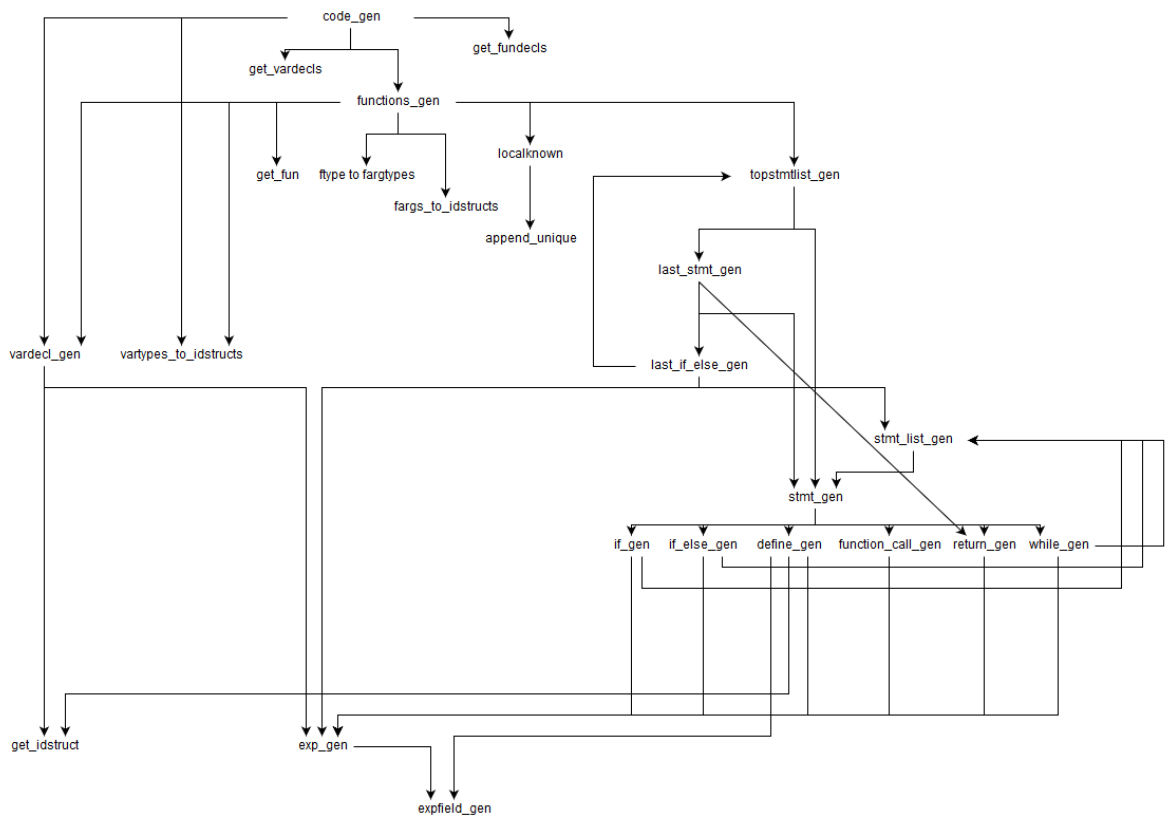
## m_fieldexp
The base-case for this recursive function is “Nofield id”, which represents the case in which there are no field operators (anymore). It is checked with “m_id_var”

When there are field operators left, it follows the function application example from the slide, treating the operator as a function and the rest of the expression as its argument. The operator is checked with “m_field”.

## m_field
Maps “Hd”, “Tl”, “Fst” and “Snd” to functions $[\alpha] \to \alpha$, $[\alpha] \to [\alpha]$, $(\alpha, \beta) \to \alpha$ and $(\alpha, \beta) \to \beta$ respectively, with $\alpha, \beta$ fresh.

## m_cons, m_id_var, m_id_fun
Look in the environment for constructors, variables and functions respectively. Unify the given type with the type found in the environment. Return an error when the identifier isn’t found in the environment.

# Code generation



The most important function *code_generator.ml* is *code_gen*. This function gets an environment filled with vartypes, funtypes and enums made by the *m* function in *typechecker.ml* and it gets an AST made by our parser. Using these it converts the AST to a string of Simple Stack Machine code. This module uses *codefragments.ml*.
*codefragments.ml* is a library-like module filled with small functions which return code strings. If one wants to find out from which module a certain function is, an easy guideline is that (almost) al functions from *code_generator.ml* have a name ending on *_gen,* as to signal that they are large generator functions, and that (almost) all functions from *codefragments.ml* have a name ending on *_code*, as to signal that they are small code returning functions.
Everything in *codefragments.ml* is considered self-explanatory.

## types

This module has its own type to model variables called idstructs. An idstruct is an Ocaml record containing the following:

- **global**: a boolean which is true if the variable is global and false if it is local. Function arguments are seen as local variables.
- **vartype**: this is the type of the variable as defined in *typechecker_types.ml*. This means it can be *Var, Imp, Tup, Lis, Enum, Int, Bool, Char* or *Void.* In practice it can't be an *Imp* or *Void*, since only functions can be those types.
- **id**: the name of the variable. Since this name is unique, this is used to discern idstructs from each other.
- **index**: where this variable can be found in the SSM memory. If the variable is global the index is its location on the heap, if the variable is local the index is its location relative to the mark

pointer. Indexes of function arguments are negative, since they are located before the mark pointer.

Together these four types encapsulate everything we need to know about a variable.

## code_gen

This function does two things:

- first it converts the given datastructures to more usable ones: it creates a list of global idstructs of all the global vars in the env. It also makes a list of the funs in the env. And it makes a list of all enums.
- then it generates code. The code generated does, in order, the following:
    - first space is allocated on the heap for the global vars.
    - then all global vars are instantiated.
    - then the main function is called.
    - after the main function is executed, we have a piece of end code which prints what the main function returned. We decided that every main function should return 1, since if something goes wrong, the SSM machine is most likely to print 0. This is because all negative stack values are zero.
    - beneath this are all the functions the programmer defined as well as the three standard functions read, write and isEmpty.

## functions_gen

This function receives all global vars, called gvars; all funs from the env, called funtypes; all enum names, called types; and a fundecllist. Notice that funtypes is probably not the best name for the env funs, because the env funs contain a lot more than just their types. They also contain the local variables and function arguments.

Just as the *code_gen* function, this function first converts and extracts data and then generates code. The data conversions and extractions are, in order:

- the type of the fundecl we are looking at is searched in the given funtypes.
- the types of the function arguments are extracted from the funtype.
- those function arguments are converted to idstructs.
- the local variables are also extracted from the funtype. Since those extracted variables are not only the locals, but also the function arguments, the function arguments are filtered out. The reason we extract the variables from the funtype, and not from the vardeclllist, is because we need their types.
- the local variables are converted to idstructs.
- all global and local variables and function arguments are put in one idstruct list called localknown. If a global variable has the same name as a local variable or function argument, the global variable is removed.

The code generated does, in order:

- first the label is made referencing this function
- then space is allocated on the stack for the local variables.
- then the local variables are instantiated.
- then all statements are executed

Then this function calls itself recursively so it parses all the fundecls.

## ftype_to_fargtypes

funs from the env do not contain a list of types of the function arguments. This while we do need such a list. Luckily this list can be extracted from the type of the fun. This type is of typechecker_types.types and consists out of nested Imp's. This is because the type of a function is the implication of all its argument types to the result type. This function converts extracts the argument types and result type and returns them as a list. This means we have one argument too many, since we do not need the result type, but this last argument will be thrown away in the *fargs_to_idstructs* function.

## vartypes_to_idstructs

This function, given a boolean global, an int index and an env_var list vartypes, generates a list of idstructs, where every idstruct in the list has an index one higher than the previous idstruct and all have the same scope (global or local).

## localknown

This function, given function arguments, local variables and global variables, all given as idstruct lists, returns an idstruct list where all global variables are removed if their name was already taken by a function argument or local variable.

## vardecl_gen

This function, given vardecls and idstructs, instantiates the vardecls. It first finds the id in the idstructs list which is used in the vardecl, then it generates code that does the following:

- the expression is executed. The result is now on the stack
- the variable which has to be instantiated is set to the value on the stack. Here the idstruct its index and global bool are used to see where the variable is located in the SSM.

## fargs_to_idstructs

This function, given an int i, a types list fargtypes and a id list fargs, generates a list of idstructs, where every idstruct in the list has an index one higher than the previous idstruct, all are local the types and ids correspond with the ones given in order.

## Statements and the label problem

As can be seen in the diagram, this module contains a lot of functions for the generation of statements. This amount is mainly caused by the label problem. Some statements (*if, if-else* and *while*) need labels to do their job. This can cause some code lines to need double labels, something which is not possible. The solution is, whenever a line has double labels, to let one label 'win' and have all references to the other label instead point to the winning label.

For this we need to know when labels collide and how to resolve every case. Let's have a look at the three label using statements. The labels and their references are underlined. When generating all labels are unique, something which is not shown in this example.

**1. if-statement:**
expcode
brf <u>endif</u>
ifbody
<u>endif</u>:

An if-statement looks at a boolean expression. If this expression is true, the ifbody is executed, else one jumps to the end.

**2. if-else-statement:**

```
expcode
brf endif
ifbody
bra endelse
endif: elsebody
endelse:
```

An if-else-statement looks at a boolean expression too. If this expression is true, the ifbody is executed and the elsebody is skipped using a jump. If the expression is false, the ifbody is skipped with a jump and the elsebody is executed.

**3. while-statement:**

```
startwhile: expcode
brf endwhile
whilebody
bra startwhile
endwhile:
```

A while-statement also looks at a boolean expression. If this expression is false, one jumps to the end of the statement, if the expression is true the whilebody is executed and then one jumps back to the beginning of the statement.

In those statements the ifbody, elsebody and whilebody are statement lists. This gives us the following ways labels can collide:

- *If, if-else* and *while* all have labels at the end of their statement. If they are followed by a statement which has a label at the beginning, this gives a clash.
- *While* has a label at the beginning of its statement.
- New functions also have their label at the beginning. Functionlabels should always win.
- The ifbody in *if* is directly followed by a label.
- The elsebody is sandwiched by two labels.

We solve this using the following rules.

- If two statements following each other give a label problem, the first statement wins.
- If a statement has his own statement list (for example the if-statement has an ifbody), and this gives a label problem, then the statement should solve the label problem (it should lose).

After applying these rules only the functionlabel can cause problems at the end of the function.

## The end of the function

Functions should always end with return statements, except for when they end on an if-else-statement, which should then have both ifbody and elsebody end on a return statement (or if-else-statement, which should then have.. etc.). For this we have the three functions:

- *Topstmtlist_gen*, the stmtlist code generator die aangeroepen wordt door *functions_gen*
- *Last_stmt_gen*, the functie die *topstmtlist_gen* aanroept op de laatste stmt. Deze functie dwingt af dat de laatste stmt een if-else-statement of een return statement moet zijn.

Anders geeft het een foutmelding. Dit is de enige foutmelding die *code_generator.ml* kan geven.

- *Last_if_else_gen*, een speciale versie van *if_else_gen* die geen endelse label heeft, aangezien je daar toch niet kan komen, en die afdwingt dat de ifbody en elsebody goed eindigen, door *topstmtlist_gen* aan te roepen in plaats van *stmtlist_gen*.

## choose_label

This function is given a label option and a label. If the label option contains a label then that label is returned, if the label option contains None, then the other label is returned. This function is a help function which indicates that when there is a possibility of two overlapping labels the label first given to this function should win.

## Lists, tuples and constructors

Lists and tuples are saved in the SSM as two values on the heap. The latest value is the tail or second item. The first value is the head or first item.
Constructors are saved as ints. In the types list given, the constructor is searched, and its index is used as int, making sure that every enum has an unique number.

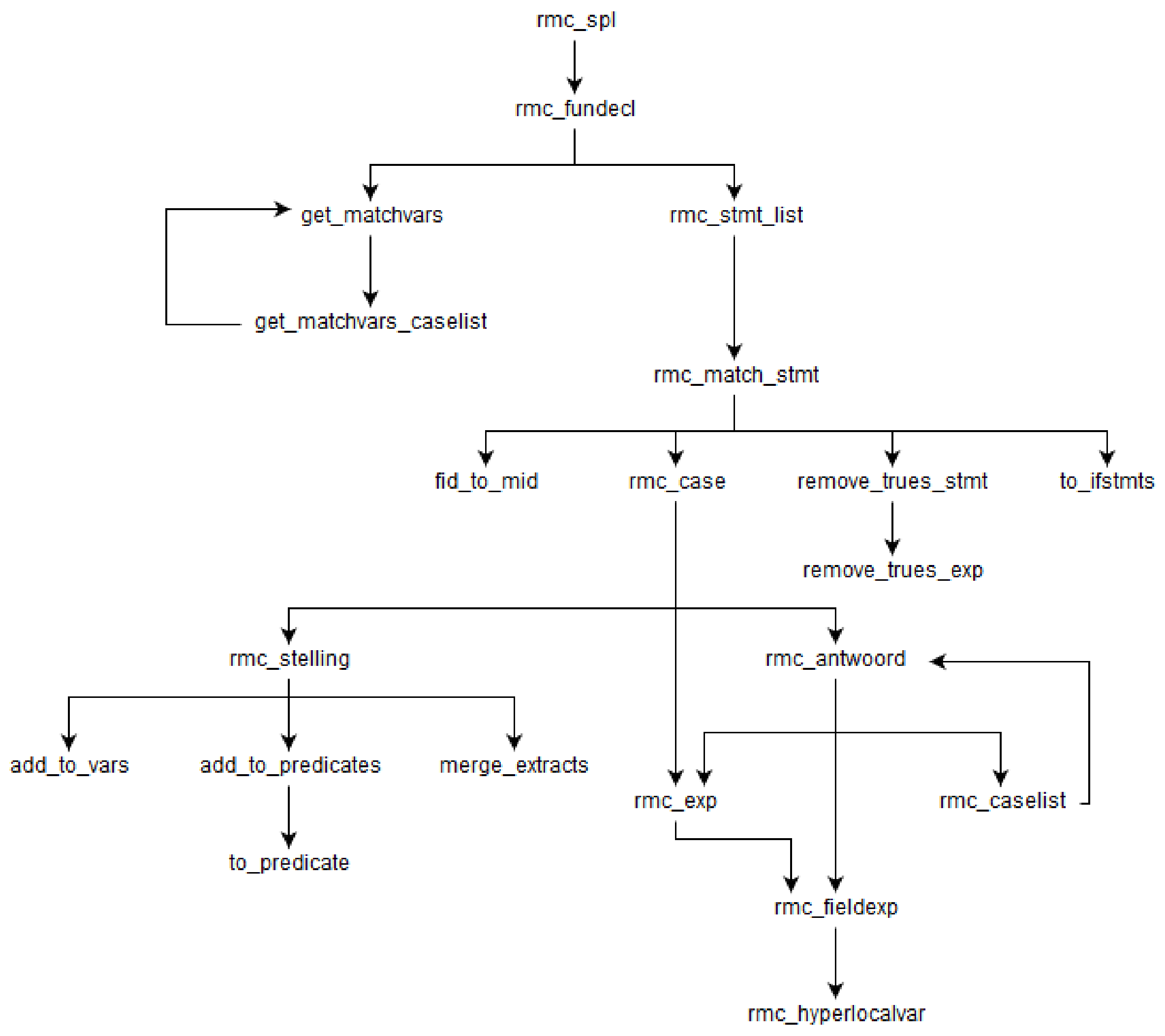
## Constructors with arguments

Constructors with arguments are not added, but could have been added with just a bit more time. The constructor with argument would then be saved in the SSM as a tuple. The latest value is the number corresponding to the constructor. The first value is the argument.

# Extension

Our extension is pattern matching with type renames and enums. Type renames and enums have to be defined at the very beginning of the code and can then be used throughout the rest of the code. A type rename is basically the same type as what it is renamed after. For example if I define name_age as ([char],int), then I can compare this to any other ([char],int) and perform tuple-expecting functions on it like hd. But if I try to do something with it that is not type correct, then the error message from the typechecker will tell us that the type of our element is name_age, and not just simply ([char],int). The same holds for enums. Also, if something goes wrong in a pattern match, then the typechecker will tell that the pattern match failed.

So, for the typechecker type renames, enums and pattern matches are new and completely different cases. For code generation, enums are new, but type renames and pattern matches are not. This is because for one code generation does not care about types, fully assuming that the typechecker took care of this, and secondly, because we have a module which rewrites pattern matches to if- and if-else-statements, which is shown on the next page.



## rewrite_match_casings.ml

In this module we rewrite all match statements in our AST to if- and if-else-statements. A match statement exists out of a match variable and a list of cases. Every case exists out of three parts, called clause (stelling), predicate (predicaat) and answer (antwoord). If the match variable matches the clause and the (optional) predicate holds, then the answer is executed. Otherwise, the next case is checked. All variables in a clause are called hyperlocal. Their scope is the case. This means that global and local variables cannot occur in clauses. If you want to compare to them, you need to use a predicate.

match statements do not need to always match. If no case matches the match variable during run-time, simply no answer is executed. Since a match statement is rewritten to if- and if-else-statements, and a function should not end with an if-statement, functions may only end on a match statement if the last clause and predicate have no restrictions (for example *(a,b)* while matching a tuple) and all answers end with return.

### types

We have two types in this module.

varbeschrijving is a op1 option and a fieldexp. If a hyperlocal var exists in a clause, then this type is used to remember how to get that hyperlocal var out of the match var. Here are some examples:

- (-a,0), a is saved as (Some Neg,matchvar.fst)
- (a:[]):[], a is saved as (None, matchvar.hd.hd)

- !a, a is saved as (Some Not, matchvar)

stelling_extract is a record of vars, predicates and werkruimte. This type keeps for one case all the info that needs to be extracted from the clause and the predicate to rewrite the match statements.

- vars is a list of (id's,varbeschrijving) and contains all hyperlocalvars in the case
- predicates is a list of all the restrictions that can be extracted.
- werkruimte is a temporal space that is used for deriving the varbeschrijving belonging to a hyperlocalvar.

## mid

In a match statement the match variable did not have a name. In the rewritten AST the match variable needs a name that is unique within a function. The name we create is a pipe symbol, an unique number and the id of the function combined to a string. For this we have the function *fid_to_mid*, or function id to match id.

## rmc_fundecl

This function does two things:

- It searches the given stmt_list for match statements and for every one it finds adds a match variable to the given vardecl_list. It uses *get_matchvars* for this.
- It searches the given stmt_list for match statements and rewrites them using *rmc_stmt_list*.

## rmc_match_stmt

This function:

- calls *rmc_case* over every case. That function returns a tuple of (restrictions imposed by the clause and predicate, answer where every hyperlocal var is replaced by its varbeschrijving).
- it rewrites this tuple to if- and if-else-statements using *to_ifstmts*
- The restrictions are conjunctions of boolean expressions. Some of these expressions are 'true'. These need to be filtered out. Also, if(true){answer} can be rewritten to simply {answer}. This is done by the function *remove_trues*.
- all answers have to be checked to see if they contain other match statements. So this function calls *rmc_stmt_list*

## rmc_case

This function:

- calls *rmc_stelling* to make a stelling_extract filled with all hyperlocalvars and their varbeschrijvingen and with all restrictions in the clause.
- calls *rmc_exp* to get the restrictions of the predicate, where every hyperlocalvar is rewritten.
- calls *rmc_antwoord* to rewrite every hyperlocalvar in the answer.

## rmc_stelling

This function gets a stelling_extract and a clause, and returns a stelling_extract. Here are all the things one can encounter in a clause, and what is done with them:

- tuples
  The function calls itself on both expressions in the tuple (made to stelling_extracts). It adds *fst* and *snd* respectively to the varbeschrijving in the werkruimte of the stelling_extracts. It then merges the returned stelling_extracts, throwing away the werkruimte.

- lists (of the form head:tail)
  Pretty much the same as tuples, except that an extra predicate is added stating that the list we are looking at is not empty.
- prefix operators
  The prefix operator is added to the varbeschrijving in the werkruimte of the stelling_extract. The function then calls itself on the rest of the expression.
- constants
  The werkruimte now contains which steps to take to get from the matchvar to this constant. We generate a predicate stating that if those are taken, we should get the constant. We add this predicate to the already know predicates.
- low bars
  No predicate is added. Since we know the match case already typechecked, no predicate is needed.
- hyperlocalvars
  The werkruimte now contains which steps to take to get from the matchvar to this constant. So it is this var his varbeschrijving. We add the var and his varbeschrijving to vars. No predicate is added.

# Programming with our compiler

The following conventions hold when using our compiler:

- The grammar is the same as the original, except for the extensions.
- One can now add type renames, by writing ‘type *typename* = *type*.
- *Type* can be char, int, bool, (*a,a*), [*a*].
- One can now add enums, by writing ‘type *typename* = *enum1* [| *enum*]$^+$
- One cannot define an enum with only one option.
- Enums must start with a capital letter.
- Types are now with lower capital, as to clearly differ from enums.
- Types have to be defined at the start of the code.
- Functions and variables can be defined in any order.
- Pattern matches can now be added at every place one can write a statement.
- One can specify extra conditions in a case using the when keyword.
- Pattern matches do not need to match every case. If no case is matched, the next statement is executed.
- The code should have a main function. This main function should return 1.
- Clauses cannot contain function calls, infix operators other than the list operator or more than one prefix operator per variable.
- All variables in clauses are considered hyperlocal. If one wants to compare to another variable, one will need to use predicates.
- All decision paths within a function must end with a return statement.
- Within a clause the same hyperlocal may not occur multiple times.
- Functions may recursively call each other.