Bachelor thesis
Computing Science

Radboud University

Formalizing the C preprocessor

Author:
Alexander Wolters
s1022959

Daily supervisor/assessor:
Dr. Freek Wiedijk
freek@cs.ru.nl

Second supervisor/assessor:
Dr. Robbert Krebbers
mail@robbertkrebbers.nl

January 14, 2022

Abstract

There have been a number of formalizations which model the C programming language, as defined by its ISO standard documents, but in general, such models simply take the C preprocessor as given.
In this thesis, we present a model of a simplified version of the C preprocessor and investigate how such a model can be used to prove the correctness of statements involving both preprocessing and language-level requirements, using a minimal toy language in place of C.

Contents

Chapter 1

Introduction

Ever since it became a major language (see [1]), there have been several formalizations of the behaviour of the C language (or at least parts of it) as defined in the C standard ([2],[3],[4],[5],[6], among others).
However, those papers generally just assume the C preprocessor and the standard header files as a given, or relegate it to future work.
As such, the questions that we want to consider in this thesis are these:

1. How can we model the behaviour of the C preprocessor, specifically of the macro replacement directives?
    - And how can we do this with as little state as possible to simplify proofs?
2. (How) can we use this model to prove the correctness of program and header files?

In this thesis, we only consider files without condition macros, and instead focus on presenting a mathematical model of a simplified version of the C preprocessor. This simplified version only includes macro replacement directives and omits conditionals, `include` directives and anything else the C preprocessor usually does. While this is only one aspect, it serves as a basis to build upon, especially with regard to verification of C header files.
The model uses as its core idea a transformation of lines to other lines, with as little state as possible being carried between lines. The output of this model is a list of tokens which form a program. In this thesis, the program is not a C program, but rather is written in a toy language, but this is irrelevant, as the preprocessor is largely language-agnostic (except for the definition of what is a token, and the model can cope with adding to that definition just fine).
This is a first step towards building a model of the full preprocessor, which is important as it is required to prove the formal correctness of C programs that include preprocessor directives, which in practice is true of most large-scale programs, but to the best of my knowledge, no such model exists at

this point in time. Such a model would also allow the formal verification of C header files, which form an essential part of the C ecosystem, but are typically merely assumed to be correct.

1.1 Related Work

As already mentioned, there has been little interest in the C preprocessor in academic work so far. Indeed, there is only one work that I am aware of that investigates the difficulties that arise from this additional step in compiling a C program.
Kästner et al. [7] have investigated the challenges of proving statements over code for all values of conditional macros, and while they do acknowledge the problems that arise from the combination of conditionals with macro expansion, their solution builds on a pre-existing compiler to handle the macro expansion itself, and they do not consider this a focus of their work. Consequently, their work, while interesting in itself, is no help if one wishes to have a model hat can be written in a proof assistant, as this thesis aims to achieve.

1.2 Overview

We first go over some preliminary knowledge, which consists of select examples of the behaviour of the C preprocessor in chapter 2, an explanation of the toy language in chapter 3 and some notation used in this thesis in chapter 4. Then we give the formal grammar of our languages before and after the preprocessing step in chapter 5. After that, we give the functions for the preprocessing (chapter 6) and the evaluation (chapter 7) of our language, the former of which is the answer to the first research question. Finally, we consider how to show the correctness of our model in chapter 8 and answer the second research question in chapter 9, before we finish up with our overall conclusions and a discussion on expandability of the model and its applicability to C in chapter 10. A large-scale overview of this thesis in a picture is given in fig. 1.1.

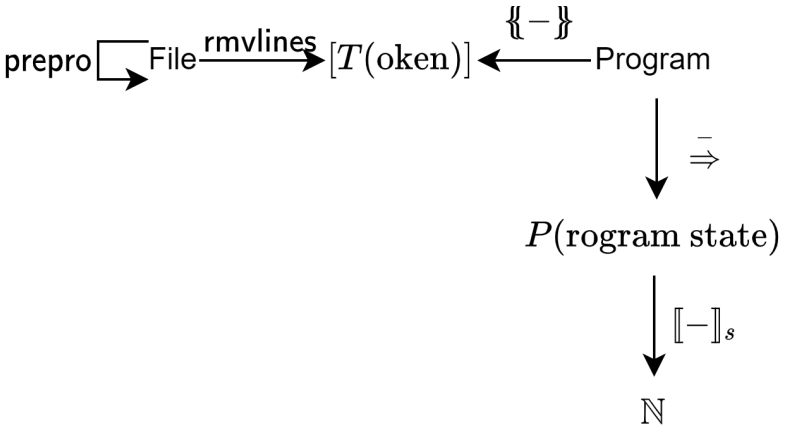

Figure 1.1: Large-scale overview of the thesis

Chapter 2

Behaviour of the C preprocessor

For a full description of the C preprocessor and all its directives, please refer to [8], with particular attention to sections 5.1.1.2 and 6.10.
Here, I merely give some examples to give the reader a general understanding of the directives from the standard that are formalized in this thesis.

2.1 Examples

As the preprocessor is primarily a transformation over lines, the examples are given in a similar style, with the left side being the input line and the right side the output resulting thereof. As some lines do not give any output, the corresponding lines on the right side will be left blank.
Horizontal lines are used to separate examples from each other, consider each block between horizontal lines to be entirely unrelated to any other such block.

`#define A B` `A`	 `B`

`#define A B`
`#define B C`
`A`
`B`
	
	`C`
	`C`

`#define B C`
`#define A B`
`A`
`B`
	
	`C`
	`C`

`#define A(X) X` `A(2)`	 `2`
`#define A(X) X + 2` `A(3)`	 `3 + 2`
`#define A(X) 3` `A(2)`	 `3`

`#define B(X) 2 + X`
`#define A B`
`A(2)`
`B`
	
	`B(2)`
	`B`

`#define A(X) X,Z`
`declare f(A(Z));`
Note that this is not legal in the toy language, but that is irrelevant to the preprocessor.
	
	`declare f(Z,Z);`

`#define ZERO(X) const X = 0;` `ZERO(z)`	 `const z = 0;`
`A` `#define A B`	`A`

`#define A B`
`#define B A`
`A`
`B`
	
	`A`
	`B`

`#define A B`
`#define B C`
`#define C A`
`A`
	`A`

`#define A(X,Y) X + Y`
`A(2,`
`3)`
	`2 + 3`

`#define A(X,Y) X + Y`
`A(2,`
`#define B C`
`3)`
	`2 + 3`

`A(2,`
`#define A(X,Y) X + Y`
`3)`
	`A(2,`
	
	`3)`

As not all macro definitions are legal, I also give some examples of legal and illegal macro definitions, but they can all essentially be summed by this quote from the C standard: "An identifier currently defined as an object-like macro shall not be redefined by another `#define` preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical"([8], 6.10.3.2), and similarly for function-like macros.
Legal:

`#define A B` `#define A B`
`#define A(X) X` `#define A(X) X`
`#define A(X) X` `#define A(Y) Y`
`#define A(X,Y) X + Y` `#define A(Y,X) Y + X`

Note that the last two cases are technically illegal according to the standard as well. This is the one corner case where this thesis does not quite follow the standard and is slightly more permissive instead.

I also give some examples of illegal definitions where macros are redefined differently than they were:

`#define A B`
`#define A C`
`#define A B`
`#define A(X) X`
`#define A(X) X`
`#define A B`
`#define A(X,Y) X + Y`
`#define A(Y,X) X + Y`
`#define A B`
`#define C A`
`#define C B`
`#define A(X,X) X`

Chapter 3

A toy language

The toy language that is used in this thesis as the target of the preprocessor is a very simple language. The full grammar is given in section 5.3, but in short, a program is a list of statements, and there are only three types of statements: Function declarations, definitions, and constant definitions.
The following example illustrates each of these three in turn:

```
declare f(y);
define f(x) = ((5 + x) * 3);
const c = f(2);
```

In this example, the value of the constant `c` is 21. As the language is not very complex, it is expected that the meaning of any expression is easy enough to determine for anyone with experience with any C-like language. However, some things are not permitted, so the next two examples would be invalid programs:

```
define f(x) = ((5 + x) * 3);
define f(x) = ((5 + x) * 3);
```

```
const c = 12;
const c = 13;
```

Note that it does not matter if the second definition is the same as the first or different, no function or constant may be defined twice.
It is allowed, however, to declare a function multiple times, even with different parameter names (But parameter names must be unique, even for declarations):

```
declare f(x);
declare f(y);
```

Of course, the number of parameters must be the same, so this would be forbidden:

```
declare f(x);
declare f(x, y);
```

And a function must be defined if it is to be used, though there is no restriction on the order of the two, so while this is no forbidden, the value of `c` is not defined until a definition for `f` is given (note that this example would be valid even without the function declaration):

```
declare f(x);
const c = f(5);
```

The following example is not forbidden, but it will never be possible to evaluate the constant `c`:

```
const c = (c + 1);
```

Indeed, any constant or function definition that refers back to itself is legal, but in general, the value of such a constant or function cannot be used, unless there is a multiplication with zero involved:

```
const c = (0 * d); define f(x) = (x * f((x - 1)));
```

Note that even though the constant `d` does not exist yet, the evaluation of `c` is possible, as what is written on the right hand side does not matter. Also, since `f(0)` $= 0$, the function `f` always returns 0, and not the factorial of it's input, as one may assume.
This example also demonstrates that it is legal for an expression to contain a constant name that is not yet introduced, as long as the name is defined when it is evaluated.

Chapter 4

Notation

4.1 Sets and Types

Wherever this thesis uses symbolic notation to represent the type for some function or value, the following symbols always have the same meaning:

- Given some type X, $[X]$ stands for a list of elements of type X, which may have repetitions, is well ordered and has variable length (unlike tuples).
  There are also several constructors for and functions on lists, which have the same meaning as the versions in the Haskell standard library if nothing else is explicitly mentioned.

- Given some type X, we use $X^?$ to represent a type that could be called "possibly X", that is, a type which is either really a X or nothing at all, which we denote as $\perp$. (This type is analogous to Haskell's `Maybe` datatype [9] and Java's `Optional` class [10].)
  Given an element $x \in X^?$, we can write $x\downarrow$ to express that $x \in X$, and we can write $x\uparrow$ to express that it is $\perp$.

- $\mathbb{B}$ stands for the set of Boolean values, that is $\mathbb{B} = \{\mathbb{T}, \mathbb{F}\}$, where the symbols $\mathbb{T}$ and $\mathbb{F}$ represent the Boolean values true and false, respectively.

- Σ stands for the set of valid identifiers, which is a letter or underscore followed by any number of letters, digits and underscores. Identifiers are printed in `monospace`.

- T stands for the set of tokens, as defined in the grammar in section 3.1.1.

4.2 Lambda functions

As there are several higher-order functions in this thesis, and it is not useful to give a name to each function, we may wish to pass in as arguments to them, we make use of lambda functions as a shorthand notation. In this document, they are written as such: $\lambda x, y.\ x + y$, which denotes a function that takes two inputs and returns the sum of those inputs.
Note that this notation does not include types, they can be inferred from the context where the lambda function is given.

4.3 Partial functions

In this thesis, partial functions are sometimes used where no reasonable definition would exist for some input to a function, or to represent some kind of error state (For example, trying to redefine a macro constant with a different value). In this case, the type of the function is written as $f : A \rightharpoonup B$, where A and B are types.
In such a case, we write $f(x)\downarrow$ to represent that f is defined for input x, and we write $f(x)\uparrow$ to represent that f is not defined for input x.
We also define the special function σ, which is not defined for any input, and therefore can match any function type. (Much as the empty set can be a set of any type.)

4.4 Updating

We use the notation of $f[a \mapsto b]$ for some (possibly partial) function $f : A \to B$ and some values $a \in A, b \in B$ to represent a new (possibly partial) function, which gives the same result as f for all inputs, except for input a, for which it gives b.
We use the same notation for lists as well, but here, a is a natural number, and the meaning is that the a'th element of the list is set to b, with all the other elements unaltered.

Chapter 5

Formal grammars

We give formal grammars for our languages, both for before and after the macro expansion. Terminals are given in `typewriter font`, whereas non-terminals are given in *italics*. Subsequent indented lines indicate alternative definitions. λ represents the empty string.

5.1 Common non-terminals

ArgumentList:
    Identifier
    Identifier `,` *ArgumentList*

Token:
    Identifier
    IntegerLiteral
    (one of)
        `+ - * / ( ) ; = ,`

Identifier:
    Letter
    Identifier Letter
    Identifier Digit

Letter:
    (one of)
        `a b c d e f g h i j k l m n o p q r s t u v w x y z`
        `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _`

IntegerLiteral:
    IntegerLiteral Digit
    Digit

Digit:
    (one of)
        `0 1 2 3 4 5 6 7 8 9`

Note that *Identifier* is effectively the same as *identifier* is in the C standard, though, for simplicity, we do not use universal characters.

5.2 Before expansion

File:
    λ
    Line `\n` *File*

Line:
    TokenList
    MacroLine

TokenList:
    λ
    Token TokenList

MacroLine:
    `#define` *Identifier TokenList*
    `#define` *Identifier* `(` *ArgumentList* `)` *TokenList*

5.3 After expansion

Program:
    λ
    Statement Program

Statement:
    FuncDecl
    FuncDefn
    ConstDefn

FuncDecl:
    `declare` *Identifier* `(` *ArgumentList* `) ;`

FuncDefn:

`define` *Identifier* `(` *ArgumentList* `)` `=` *Expression* `;`

ConstDefn:
`const` *Identifier* `=` *Expression* `;`

Expression:
`(` *Expression BinOp Expression* `)`
IntegerLiteral
Identifier
Identifier `(` *ExpressionList* `)`

ExpressionList:
Expression
Expression `,` *ExpressionList*

BinOp:
(one of)
`+ - * /`

Chapter 6

Preprocessing function

The preprocessing function turns a *File* with macros into a *File* that only consists of *TokenList*s. To define this function, we create several subfunctions, one for each type of possible line, which are then combined into one overarching function. This function is only a partial function, it is undefined if the *File* would not be legal to write according to the restrictions laid out in the C standard (e.g. redefining a macro with a different value). Symbolically, the "main" function for this step has the type $\mathsf{prepro} : \mathit{File} \rightharpoonup \mathit{File}$.
For brevity, we define two types that we use throughout this section to avoid giving the expanded version each time:

- M stands for the set of macro mappings, which store the macro definitions.
  Such a mapping is a partial function that maps macro names to information about the number of arguments (if any) and their replacements. If the macro takes arguments (i.e. the first element of the tuple is greater than 0), it is a function like macro. In this case, the replacement may contain non-negative integers, which represent the places where the arguments should be substituted. Symbolically:

$$M = \Sigma \rightharpoonup (\mathbb{N} \times [T \cup \mathbb{N}])$$

- S stands for the state related to the expansion of function-like macros. This state must be preserved between lines, as macro arguments may span multiple lines.
  Such a mapping is a 6-tuple of multiple different types, namely

$$S = \{0, 1, 2\} \times \mathbb{N} \times [[T]] \times \mathcal{P}(T) \times T^{?} \times \mathbb{N}$$

  Some explanation is in order as to the meaning of those tuple members: The first member is always 0, 1 or 2, and it represents the state

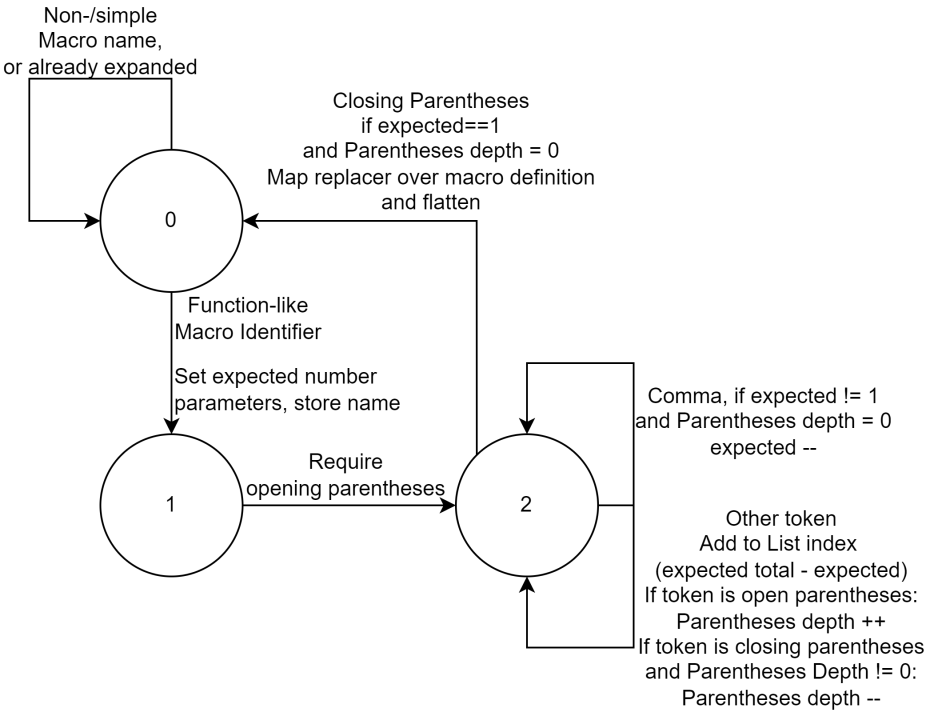

Figure 6.1: State diagram for S

that the preprocessor is in, which is used in gathering the arguments for function-like macros. Refer to fig. 6.1 for a reference of what each state means.
The next integer is the number of arguments still remaining to be read in the current function-like macro, and each sub-list of the double list in the third member represents what is gathered so far for those arguments.
The fourth member, the set of *Token*s, is there to store the macros that were already expanded in recursive expansion to avoid infinite loops.
The fifth member is the name of the function-like macro we are currently working on. The name is required to do the macro replacement, and is not present if we are not currently gathering any macro arguments.
The sixth member keeps track of the parentheses depth, as macro arguments may themselves contain opening and closing parentheses, and we would otherwise be unable to determine whether a closing parentheses is intended to be part of the parameter or to end the parameter list.

Since giving all the definitions in one big expression would be difficult to read, we split the input up into different cases: The two kinds of *MacroLine*s and the case for *TokenList*, and we dispatch based on the type of line in prepro, together with some minor input preprocessing.
A Haskell version of all the functions in this section is also given in annexes A and B.

6.1 Wrapper and main function

The function prepro, with the type as given above, is only a wrapper function that sets up the arguments to the real main function, dispatch. This is necessary to keep the type of the function clean, but the additional information is needed for the actual preprocessing work.
We therefore define:

$$\begin{aligned}
\mathsf{prepro} &: \mathit{File} \rightharpoonup \mathit{File} \\
\mathsf{prepro}(f) &= \mathsf{dispatch}(f, \sigma, (0, 0, [], \emptyset, \bot, 0))
\end{aligned}$$

$$\begin{aligned}
\mathsf{dispatch} &: \mathit{File} \times M \times S \rightharpoonup \mathit{File} \\
\mathsf{dispatch}(\lambda, m, s) &= \lambda
\end{aligned}$$

$$\begin{aligned}
\mathsf{dispatch}(\texttt{\#define}\ i\ t\ \texttt{\textbackslash n}\ f, m, s) &= r'\ \texttt{\textbackslash n}\ \mathsf{dispatch}(f, m', s') \\
&\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r) \\
&\textbf{where}\ (m', s', r) \coloneqq \mathsf{om}(m, s, i, \mathsf{tlisitfy}(t))
\end{aligned}$$

$$\begin{aligned}
\mathsf{dispatch}(\texttt{\#define}\ i\ \texttt{(}\ a\ \texttt{)}\ t\ \texttt{\textbackslash n}\ f, m, s) &= r'\ \texttt{\textbackslash n}\ \mathsf{dispatch}(f, m', s') \\
&\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r) \\
&\textbf{where}\ (m', s', r) \coloneqq \mathsf{fm}(m, s, i, \mathsf{alisitfy}(a), \mathsf{tlisitfy}(t))
\end{aligned}$$

$$\begin{aligned}
\mathsf{dispatch}(t\ \texttt{\textbackslash n}\ f, m, s) &= r'\ \texttt{\textbackslash n}\ \mathsf{dispatch}(f, m', s') \\
&\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r) \\
&\textbf{where}\ (m', s', r) \coloneqq \mathit{tl}(m, s, \mathsf{tlisitfy}(t))
\end{aligned}$$

$$\begin{aligned}
\mathsf{unlistify} &: [T] \to \mathit{TokenList} \\
\mathsf{unlistify}([]) &= \lambda \\
\mathsf{unlistify}(t : ts) &= t\ \mathsf{unlistify}(ts)
\end{aligned}$$

We also give a function which turns a file into list of tokens, which is a necessary step to parsing it as a program. This function is only partial,

because it is impossible to turn a *MacroLine* into a list of *Token*s, and the function is only intended to be called on the output of prepro, anyways.

$$\begin{aligned}\mathsf{rmvlines} &: \mathit{File} \rightharpoonup [T]\\ \mathsf{rmvlines}(\lambda) &= []\end{aligned}$$

$$\mathsf{rmvlines}(t \setminus \mathtt{n}\ f) = \mathsf{tlisitfy}(t) +\!\!+ \mathsf{rmvlines}(f) \qquad \textbf{iff } t \in \mathit{TokenList}$$

6.2 Object Macro

This case is relatively simple, as we only need to transform the *TokenList* into a proper list of tokens and store that in the state, making sure not to overwrite a previous macro.

$$\begin{gathered}\mathsf{om} : M \times S \times \Sigma \times [T] \rightharpoonup M \times S \times [T]\\ \mathsf{om}(m, s, i, r) = (m[i \mapsto (0, r)], s, []) \textbf{ iff } m(i)\uparrow \vee\ m(i) \neq (0, r)\end{gathered}$$

6.3 Function-Like Macro

This case is not quite so simple, so we introduce an intermediate variable to keep the final equation legible:

$$\begin{gathered}\mathsf{fm} : M \times S \times \Sigma \times [\Sigma] \times [T] \rightharpoonup M \times S \times [T]\\ \mathsf{fm}(m, s, i, a, r) = (m[i \mapsto \mathit{res}], s, []) \textbf{ iff } m(i)\uparrow \vee\ m(i) \neq \mathit{res}\\ \textbf{where } \mathit{res} \coloneqq (\mathit{len}(a), \mathsf{map}(\lambda t.\ \mathsf{repind}(a, t), r))\end{gathered}$$

6.4 Text Line

We handle a text line by creating a simple state machine (fig. 6.1) and then transforming that state machine into a series of equations that describe the behaviour of the macro expansion process.
We first remove the requirement to always carry along the mapping function, as it is never changed and would unnecessarily complicate the definition even further.

$$\begin{gathered}\mathsf{tl} : M \times S \times [T] \rightharpoonup M \times S \times [T]\\ \mathsf{tl}(m, s, t) = (m, s', t') \textbf{ where } (s', t') \coloneqq \mathsf{tlr}(m, s, t)\end{gathered}$$

We first define the tlr function, and then give the auxillary functions, followed by explanations for each equation in English afterwards.

$$\mathsf{tlr} : M \times S \times [T] \rightharpoonup S \times [T]$$
$$\mathsf{tlr}(m, (0, 0, [], u, \bot, 0), []) = ((0, 0, [], u, \bot, 0), []) \tag{1}$$

$$\mathsf{tlr}(m, (0, 0, [], u, \bot, 0), t : ts) = (s, t : r) \tag{2}$$
$$\textbf{where } (s, r) \coloneqq \mathsf{tlr}(m, (0, 0, [], u, \bot, 0), ts)$$
$$\textbf{iff } t \in u \vee m(t)\uparrow$$

$$\mathsf{tlr}(m, (0, 0, [], u, \bot), t : ts) = (s, \mathsf{tlr}(m, (0, 0, [], \{t\} \cup u, \bot, 0), e)\mathbin{+\!\!+} r) \tag{3}$$
$$\textbf{where } (s, r) \coloneqq \mathsf{tlr}(m, (0, 0, [], u, \bot, 0), ts)$$
$$\textbf{iff } t \notin u \wedge m(t)\downarrow \wedge m(t) = (0, e)$$

$$\mathsf{tlr}(m, (0, 0, [], u, \bot, 0), t : ts) = \mathsf{tlr}(m, (1, n, \mathsf{repeat}([], n), u, t, 0), ts) \tag{4}$$
$$\textbf{iff } t \notin u \wedge s(t)\downarrow \wedge s(t) = (n, r) \wedge n \neq 0$$

$$\mathsf{tlr}(m, (1, n, ls, u, t, 0), \texttt{(} : ts) = \mathsf{tlr}(m, (2, n, ls, u, t, 0), ts) \tag{5}$$

$$\mathsf{tlr}(m, (2, n, ls, u, t, 0), \texttt{,} : ts) = \mathsf{tlr}(m, (2, n - 1, ls, u, t, 0), ts) \textbf{ iff } n \neq 1 \tag{6}$$

$$\mathsf{tlr}(m, (2, 1, ls, u, t, 0), \texttt{)} : ts) = (s, \mathsf{expand}(s, ls, u, t)\mathbin{+\!\!+} r) \tag{7}$$
$$\textbf{where } (s, r) \coloneqq \mathsf{tlr}(m, (0, 0, [], u, \bot, 0), ts)$$

$$\mathsf{tlr}(m, (2, n, ls, u, t, d), ct : ts) = \mathsf{tlr}(m, (2, n, lsm, u, t, d + dd), ts) \tag{8}$$
$$\textbf{where } lsm \coloneqq ls[nt - n \mapsto \mathsf{append}(ls[nt - n], ct)],$$
$$(nt, r) = m(t), dd = \begin{cases} 1 & \textbf{iff } ct = \texttt{(} \\ 0 & \textbf{otherwise} \end{cases}$$

$$\mathsf{expand} : M \times [[T]] \times \mathcal{P}(T) \times T \rightharpoonup [T]$$
$$\mathsf{expand}(m, ls, u, t) = \mathsf{tlr}(m, (0, 0, [], \{t\} \cup u, \bot, 0), \mathsf{flatten}(\mathsf{map}(\mathsf{repl\_int}(ls), r)))$$
$$\textbf{where } (n, r) \coloneqq s(t)$$

$$\mathsf{flatten} : [[T]] \to [T]$$
$$\mathsf{flatten}([]) = []$$
$$\mathsf{flatten}([] : bs) = \mathsf{flatten}(bs)$$
$$\mathsf{flatten}((a : as) : bs) = a : \mathsf{flatten}(as : bs)$$

$$\mathsf{repl\_int} : [T] \to (T \cup \mathbb{Z}) \to T$$
$$\mathsf{repl\_int}(ls)(i) = ls[i] \qquad \textbf{iff } i \in \mathbb{N}$$
$$\mathsf{repl\_int}(ls)(t) = t \qquad \textbf{iff } t \in T$$

Explanations

(1) This is the easiest case: An empty line remains just that, no expansion can take place.

(2) If we already expanded the current token we leave it alone to avoid infinite replacement loops, and if the token is not defined to have any replacement, then it also stays verbatim.

(3) If the token is a known macro, and the first entry of the tuple is 0, then it refers to a constant macro, we then expand that macro, and possibly any further macro names that result from the expansion, and prepend that expansion to the expansion of the rest of the line.

(4) If we encounter a macro which we know to refer to a function macro, we set up the parameters accordingly, as is mentioned in the state diagram: Store the name of the macro that we are collecting arguments for, keep track of how many arguments we still expect to find and set the state to 1 to await the opening parenthesis that must follow this macro name.

(5) This case is also rather straight-forward, in state 1, we only want to see an opening parenthesis, anything else is not allowed, and we then process the next token(s) in state 2.

(6) If we encounter a comma in state 2 and with no more pending open parentheses pairs, this means that we have reached the end of one argument and are about to encounter the start of another. This is not permitted if we only expected one more argument, as this would mean that we gave to many arguments to a macro.

(7) A closing parentheses is only allowed to close the argument list once we gathered all the necessary arguments, and if there is no pending open parenthesis in the current argument. We then do the expansion, including any recursive macro expansion resulting therefrom, using the aptly named expand function, and continue on with the remainder of the line.

(8) If we encounter any other token than a comma or a closing parentheses, then we consider it part of the macro arguments, and consequently add it to the appropriate sublist of the 5th argument. In addition, if the token is an opening parenthesis, then we keep track of that so we

know when we find it's closing counterpart rather than the end of the argument list.

6.5 Auxiliary functions

Following are the definitions of the functions that were used above, but not defined.

tlistify

The function tlistify turns the syntactical element *TokenList* into the corresponding list of tokens.

$$\begin{aligned} \mathsf{tlistify} &: \mathit{TokenList} \to [T] \\ \mathsf{tlistify}(\lambda) &= [] \\ \mathsf{tlistify}(t\ tl) &= t : \mathsf{tlistify}(tl) \end{aligned}$$

alisitfy

The function alistify turns an *ArgumentList* into a list of *Identifier*s, without allowing any duplicates.

$$\begin{aligned} \mathsf{alistify} &: \mathit{ArgumentList} \rightharpoonup [\Sigma] \\ \mathsf{alistify}(a) &= \mathsf{alistify}'(a, \emptyset) \\ \mathsf{alistify}' &: \mathit{ArgumentList} \times \mathcal{P}(\Sigma) \rightharpoonup [\Sigma] \\ \mathsf{alistify}'(a, s) &= a : [] && \textbf{iff } a \notin s \\ \mathsf{alistify}'((a\,\texttt{,}\,al), s) &= a : \mathsf{alistify}'(al, \{a\} \cup s) && \textbf{iff } a \notin s \end{aligned}$$

repind

This function is mapped over a *Token* list, replacing each element that is in the input list by its index and otherwise leaves it alone.

$$\begin{aligned} \mathsf{repind} &: [\Sigma] \times T \to T \cup \mathbb{Z} \\ \mathsf{repind}(a, b) &= \mathsf{index(a,b)} && \textbf{iff } \mathsf{index}(a, b)\downarrow \\ \mathsf{repind}(a, b) &= b && \textbf{iff } \mathsf{index}(a, b)\uparrow \end{aligned}$$

6.6 Example: Simple macro derivation

To aid the understanding of the process described by the functions above, we go through a simple example involving a function-like macro step-by-step. The example that we use is

```
#define X(A,B) A + B
X(2,3)
```

or `#define X(A,B) A + B\n X(2,3)\n` λ.
We start by applying prepro to this, and then evaluate each function in turn.

$$\begin{aligned}
&\mathsf{prepro}(\texttt{\#define X(A,B) A + B\textbackslash n X(2,3)\textbackslash n } \lambda)\\
=\ &\mathsf{dispatch}(\texttt{\#define X(A,B) A + B\textbackslash n X(2,3)\textbackslash n } \lambda, \sigma, (0,0,[],\emptyset,\bot,0))\\
=\ &\lambda\texttt{\textbackslash n } \mathsf{dispatch}(\texttt{X(2,3)\textbackslash n } \lambda, \sigma[\texttt{X} \mapsto (2,[1,\texttt{+},2])], (0,0,[],\emptyset,\bot,0)) \quad (1)\\
=\ &\lambda\texttt{\textbackslash n 2 + 3\textbackslash n } \mathsf{dispatch}(\lambda, \sigma[\texttt{X} \mapsto (2,[1,\texttt{+},2])], (0,0,[],\emptyset,\bot,0)) \quad (2)\\
=\ &\lambda\texttt{\textbackslash n 2 + 3\textbackslash n } \lambda
\end{aligned}$$

The equal signs in the two numbered equations need some justification to show that they satisfy the equations for dispatch.
Regarding the first one, that is according to the third equation for dispatch, we need to show that

$$\mathsf{fm}(\sigma, (0,0,[],\emptyset,\bot,0), \texttt{X}, [\texttt{A},\texttt{B}], [\texttt{A},\texttt{+},\texttt{B}]) = (\sigma[\texttt{X} \mapsto (2,[1,\texttt{+},2])], (0,0,[],\emptyset,\bot,0), \lambda)$$

To show this, we first calculate the value of res from the equation for fm:

$$\begin{aligned}
res &= (\mathsf{length}([\texttt{A},\texttt{B}]), \mathsf{map}(\lambda t.\ \mathsf{repind}([\texttt{A},\texttt{B}], t), [\texttt{A},\texttt{+B}])\\
&= (2,[1,\texttt{+},2])
\end{aligned}$$

(We collapse the calculation of the length and the index replacement, as that would add a lot of highly trivial steps, without aiding in understanding.)
We then see that $m(res)\uparrow$ holds, so the condition is fulfilled and we can apply the equation for fm, and get the result written above.

We now need to justify the second numbered equals sign, which involves the fourth equation of dispatch. Therefore we need to show that

$$\mathsf{tl}(\sigma[\texttt{X} \mapsto (2,[1,\texttt{+},2])], (0,0,[],\emptyset,\bot,0), [\texttt{X}, \texttt{(}, 2, \texttt{,}, 3, \texttt{)}]) =$$

$$(\sigma[\texttt{X} \mapsto (2,[1,\texttt{+},2])], (0,0,[],\emptyset,\bot,0), [2,\texttt{+},3])$$

We start with $\mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (0, 0, [], \emptyset, \bot, 0), [\mathtt{X}, \mathtt{(}, 2, \mathtt{,}, 3, \mathtt{)}])$, and go through the steps from there.

$$
\begin{aligned}
&\mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (0, 0, [], \emptyset, \bot, 0), [\mathtt{X}, \mathtt{(}, 2, \mathtt{,}, 3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (1, 2, \mathsf{repeat}(\mathtt{[]}, 2), \emptyset, \mathtt{X}, 0), [\mathtt{(}, 2, \mathtt{,}, 3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (1, 2, [[], []], \emptyset, \mathtt{X}, 0), [\mathtt{(}, 2, \mathtt{,}, 3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (2, 2, [[], []], \emptyset, \mathtt{X}, 0), [2, \mathtt{,}, 3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (2, 2, [[2], []], \emptyset, \mathtt{X}, 0), [\mathtt{,}, 3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (2, 1, [[2], []], \emptyset, \mathtt{X}, 0), [3, \mathtt{)}]) \\
&= \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], (2, 1, [[2], [3]], \emptyset, \mathtt{X}, 0), [\mathtt{)}]) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], \mathsf{expand}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], \emptyset, \mathtt{X}) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), \\
&\qquad \mathsf{flatten}(\mathsf{map}(\mathsf{repl\_int}([[2], [3]]), [1, \mathtt{+}, 2]))) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), \\
&\qquad \mathsf{flatten}([[2], [\mathtt{+}], [3]])) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), [2, \mathtt{+}, 3]) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], 2 : \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), [\mathtt{+}, 3]) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], 2 : \mathtt{+} : \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), [3]) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], 2 : \mathtt{+} : 3 : \mathsf{tlr}(\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [[2], [3]], (0, 0, [], \{\mathtt{X}\}, \bot, 0), []) \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], 2 : \mathtt{+} : 3 : [] \mathtt{++} []) \\
&= (\sigma[\mathtt{X} \mapsto (2, [1, \mathtt{+}, 2])], [2, \mathtt{+}, 3])
\end{aligned}
$$

And from the definition of tl, which adds the m back to the tuple, we can see that we got where we needed to get.
However, this derivation also shows that the macro expansion process requires a lot of small step function applications, and so any proof involving it are best only done by automatic proof assistant systems. (You may notice that the above list of equations is not entirely precise, as I do several minor function expansions and tuple member extractions implicitly between steps, but otherwise, the derivation would be even longer, and no easier to read.)

Chapter 7

Interpretation

Once the pre-processing step has concluded, we are left with a list of tokens, which may conform, and be parseable according to, the grammar set out in section 1.3.
The parser for this is not defined explicitly, but rather, we define a function that turns a *Program* into a list of tokens, and define the parsing function as the inverse of that function, and leave it undefined if there is no *Program* that could turn into a given list of tokens, which means that the output of the pre-processing step is not a valid program.
We also give a processing and an evaluation function which together determine the value of any constant or function in the program.
A Haskell version of all the functions in this section is available in Appendix C.

7.1 Unparsing function

We write the unparsing function as $\{\{x\}\}$, where x is some syntactical element that is being turned into a list of tokens. Note that we omit the concatenation and cons operators for clarity, they are present in the Haskell

source files if one wishes to see them.

$$\{\{s\ p\}\} = \{\{s\}\}\{\{p\}\}$$

$$\{\{\texttt{declare}\ i\ \texttt{(}a\texttt{);}\}\} = \texttt{declare}\{\{i\}\}\texttt{(}\{\{a\}\}\texttt{);}$$
$$\{\{\texttt{define}\ i\ \texttt{(}a\texttt{) =}\ e\texttt{;}\}\} = \texttt{define}\{\{i\}\}\texttt{(}\{\{a\}\}\texttt{)=}\{\{e\}\}\texttt{;}$$
$$\{\{\texttt{const}\ i\ \texttt{=}\ e\texttt{;}\}\} = \texttt{const}\{\{i\}\}\texttt{=}\{\{e\}\}\texttt{;}$$

$$\{\{\texttt{(}e_1\ \texttt{+}\ e_2\texttt{)}\}\} = \texttt{(}\{\{e_1\}\} + \{\{e_2\}\}\texttt{)}$$
$$\{\{\texttt{(}e_1\ \texttt{-}\ e_2\texttt{)}\}\} = \texttt{(}\{\{e_1\}\} - \{\{e_2\}\}\texttt{)}$$
$$\{\{\texttt{(}e_1\ \texttt{*}\ e_2\texttt{)}\}\} = \texttt{(}\{\{e_1\}\} * \{\{e_2\}\}\texttt{)}$$
$$\{\{\texttt{(}e_1\ \texttt{/}\ e_2\texttt{)}\}\} = \texttt{(}\{\{e_1\}\}/\{\{e_2\}\}\texttt{)}$$

$$\{\{l\}\} = l$$

$$\{\{i\}\} = i$$

$$\{\{i(l)\}\} = \{\{i\}\}\texttt{(}\{\{l\}\}\texttt{)}$$

$$\{\{e, l\}\} = \{\{e\}\}\texttt{,}\{\{l\}\}$$

$$\{\{i, a\}\} = \{\{i\}\}\texttt{,}\{\{a\}\}$$

where the letters used for the variables indicate the type of syntactical element:

- a is used for *ArgumentList*s
- e is used for *Expression*s
- p is used for *Program*s
- i is used for *Identifier*s
- l is used for *ExpressionList*s
- s is used for *Statement*s

7.2 Processing Function

We write the processing function for a single statement as $s \xrightarrow{t} s'$ to mean that reading the statement t in state s produces a new state s'.
The type of this state is henceforth denoted as P, which is a shorthand notation for

$$P = \Sigma \rightharpoonup ([\Sigma], \mathit{Expression}^?)$$

Expression is defined in section 1.1.3.
The meaning of this function is as follows: If the value of the function for an *Identifier* is not defined, it means that no constant or function declaration by that name is known.
If it is defined, and the list is empty, then the second member of the tuple must be an *Expression*, and this represents a constant definition.
Otherwise, the members of the list represent the names of function parameters, and the second member is either the *Expression* which forms the body of the function in it's definition, or it is $\perp$, in which case the function was only declared, but not defined.
Symbolically, we therefore write the type of this function as:

$$P \xrightarrow{\mathit{Statement}} P$$

$$s \xrightarrow{\texttt{declare } i \texttt{ (} a \texttt{);}} s \qquad \textbf{iff } s(i)\downarrow \wedge \mathsf{len}(\mathsf{alistify}(a)) = \mathsf{len}(s(i)_1)$$

$$s \xrightarrow{\texttt{declare } i \texttt{ (} a \texttt{);}} s[i \mapsto (\mathsf{alistify}(a), 0)] \qquad \textbf{iff } s(i)\uparrow$$

$$s \xrightarrow{\texttt{declare } i \texttt{ (} a \texttt{) = } e \texttt{ ;}} s[i \mapsto (\mathsf{alistify}(a), e)]$$
$$\textbf{iff } s(i)\uparrow \vee (\mathsf{len}(\mathsf{alistify}(a)) = \mathsf{len}(s(i)_0) \wedge s(i)_1 = 0)$$

$$s \xrightarrow{\texttt{const } i \texttt{ = } e \texttt{ ;}} s[i \mapsto ([], e)] \qquad \textbf{iff } s(i)\uparrow$$

Given this definition, we also define the function that takes an entire *Program* and a state to mutate and gives back the state after reading that entire *Program*. Symbolically, we write the type of this function as:

$$P \xrightarrow{\mathit{Program}} P$$

$$s \xRightarrow{\lambda} s$$

$$s \xRightarrow{t\ p} s'' \textbf{ where } s \xrightarrow{t} s' \textbf{ and } s' \xRightarrow{p} s''$$

7.3 Evaluation

We give three distinct functions for evaluation, all of which return a natural number:

1. The first of the three functions evaluates constants, and it is denoted as $[\![c]\!]_s$, for some constant name c and state s.

2. The second function is used to evaluate functions. It is denoted as $[\![f]\!]^l_s$, for some function name f, state s and a list of natural numbers l, which represent the function parameters.

3. The third function evaluates expression, and it is denoted as $[\![e]\!]^p_s$, for some expression e, state s and a function from *Identifier*s to natural numbers to represent the parameter context (when the expression is part of the definition of a function).

These functions are only partial functions, as there is no sensible result to be given in certain instances, such as a constant that is not defined or a function with the wrong name of parameters.

$$
\begin{aligned}
[\![c]\!]_s &= [\![s(c)_1]\!]^\sigma_s && \textbf{iff } s(c)\downarrow \wedge\ s(c)_0 = [] \\
[\![f]\!]^l_s &= [\![e]\!]^{l'} \otimes^l && \textbf{iff } s(f)\downarrow \wedge\ \mathsf{len}(l') = \mathsf{len}(l) \\
& && \textbf{where } (l', e) \coloneqq s(f) \\
[\![(l + r)]\!]^p_s &= [\![l]\!]^p_s + [\![r]\!]^p_s \\
[\![(l - r)]\!]^p_s &= 0 && \textbf{iff } [\![l]\!]^p_s \ \leq\ [\![r]\!]^p_s \\
[\![(l - r)]\!]^p_s &= [\![l]\!]^p_s - [\![r]\!]^p_s && \textbf{iff } [\![l]\!]^p_s \ >\ [\![r]\!]^p_s \\
[\![(l * r)]\!]^p_s &= [\![l]\!]^p_s * [\![r]\!]^p_s \\
[\![(l/r)]\!]^p_s &= \lfloor [\![l]\!]^p_s / [\![r]\!]^p_s \rfloor && \textbf{iff } [\![r]\!]^p_s \neq 0 \\
[\![(l/r)]\!]^p_s\uparrow & && \textbf{iff } [\![r]\!]^p_s = 0 \\
[\![il\ d]\!]^p_s &= 10\ * [\![il]\!]^p_s + [\![d]\!]^p_s \\
[\![0]\!]^p_s &= 0 && \text{and similarly for the other digits} \\
[\![i]\!]^p_s &= p(i) && \textbf{iff } p(i)\downarrow \\
[\![i]\!]^p_s &= [\![i]\!]_s && \textbf{iff } p(i)\uparrow \\
[\![f\,(\ l\ )]\!]^p_s &= [\![f]\!]^{\mathsf{map}(\lambda a.\ [\![a]\!]^p_s, \bar{l})}_s && \textbf{where } \overline{e} = [e] \textbf{ and } \overline{e, l} = e : \bar{l}
\end{aligned}
$$

We also give the definition of an auxiliary function that is used in the definition of the evaluation function. This function takes two lists and gives back another function, which goes through the two lists in lockstep, returning the current element of the second list if the argument matches the current element of the first list, and is undefined when it reaches the end of either list.
As that is a rather abstract description, consider the example

$$f = ([1, 2, 3] \otimes [1, 4, 9])$$

Then, $f(1) = 1, f(2) = 4, f(3) = 9$, and for all other inputs, f is undefined.

$$\otimes : [a] \times [b] \to (a \rightharpoonup b)$$

$$(a : as) \otimes (b : bs) = \lambda c. \begin{cases} b \textbf{ iff } a = c \\ (as \otimes bs)(f) \textbf{ iff } a \neq c \end{cases}$$

$$[] \otimes bs = \sigma$$

$$as \otimes [] = \sigma$$

Chapter 8

Correctness of the expansion process

There are some properties that we would like for the expansion process itself to have: For one, we would like all the examples in section 2.1 to give the same outputs in our model. Of course, the identifiers in those examples were arbitrary, so we would wish to generalise, e.g. the first example as:

$$\forall x, y \in T, \mathsf{prepro}(\texttt{\#define}\ x\ y\ \backslash\texttt{n}\ x) = \lambda\ \backslash\texttt{n}\ y$$

While it would certainly be possible to write this down in some proof checking system, such as Coq, proving that this holds is merely a matter of repeatedly applying function definitions. As such, this kind of proof is not particularly interesting, although it does serve as a good sanity check, especially to diagnose any errors if such are present. Also, doing this for all the examples would add at least 15 pages of dense function derivations, and the chance for error is far too high to do so by hand.
Far more interesting are properties of the functions themselves. For example, we would expect that applying prepro to a file more than once should not change anything, i.e. prepro is idempotent, as there are no macros left in the output. Formally, we would write this as

$$\forall f \in \mathit{File}, \mathsf{prepro}(\mathsf{prepro}(f)) = \mathsf{prepro}(f)$$

Lemma 8.0.1. *Every Line in the output of the prepro function is a TokenList.*

Proof.

1. As the prepro function is defined in terms of dispatch, we prove that every *Line* in the output of the dispatch function is a *TokenList*.

2. We prove this by induction on the first argument of dispatch.

3. Base case: The first argument of the dispatch function is λ.
4. Then the output is also λ. Therefore, there are no lines in the output, and the lemma holds vacuously.
5. Inductive case 1: The first argument of the dispatch function is a *TokenList* l followed by a *File* f.
6. Inductive Hypothesis: Every *Line* in $\mathsf{dispatch}(f, m, s)$ is a *TokenList*, for all m and s.
7. To show: Every *Line* in $\mathsf{dispatch}(l \setminus \mathtt{n}\ f, m, s)$ is a *TokenList*, for all m and s.
8. $\mathsf{dispatch}(l \setminus \mathtt{n}\ f, m, s) = r' \setminus \mathtt{n}\ \mathsf{dispatch}(f, m', s')$
   where $r' \coloneqq \mathsf{unlistify}(r)$
   where $(r, m', s') \coloneqq tl(m, s, \mathsf{tlisitfy}(t))$
9. According to the definition of unlistify, r' is a *TokenList*.
10. According to step 6, every line in $\mathsf{dispatch}(f, m', s')$ is a *TokenList*.
11. From 9 and 10 it follows that 7 holds.
12. Inductive cases 2 and 3: If the first argument is either type of *Macro-Line* followed by a file, steps 6-11 hold by the same construction.
13. From 4 and 11, it follows by induction that the claim in 1 holds.
14. Therefore, as the output of dispatch is precisely the output of prepro, it follows that the lemma holds.

□

Lemma 8.0.2. $\forall t \in [T], tlr(\sigma, (0, 0, [], \emptyset, \mathit{1}, 0), t) = (((0, 0, [], \emptyset, \mathit{1}, 0)), t)$

Proof.

1. Proof by induction on t.
2. Base case: $t = []$.
3. By definition of tlr in equation 5.1, $tlr(\sigma, (0, 0, [], \emptyset, \mathtt{1}, 0), [])$
   $= ((0, 0, [], \emptyset, \mathtt{1}, 0), [])$. Therefore, the lemma holds in the base case.
4. Inductive case: $t = e : l$ for some *Token* e and some *TokenList* l.
5. Inductive hypothesis: $tlr(\sigma, (0, 0, [], \emptyset, \mathtt{1}, 0), l) = ((0, 0, [], \emptyset, \mathtt{1}, 0), l)$
6. Then, $tlr(\sigma, (0, 0, [], \emptyset, \mathtt{1}, 0), e : l) = ((0, 0, [], \emptyset, \mathtt{1}, 0), e : l)$ by using the induction hypothesis in equation 5.2, which applies because $\sigma(e) \uparrow$. Therefore, the inductive case holds.

7. From 3 and 6 it follows that the lemma holds.

□

Lemma 8.0.3. $\forall t \in \mathit{TokenList}, \mathit{unlistify}(\mathit{tlistify}(t)) = t$

Proof.

1. Proof by induction on tl.
2. Base case: $tl = \lambda$.
3. By definition of tlistify and unlistify, $\mathsf{unlistify}(\mathsf{tlistify}(\lambda)) = \lambda$, so the lemma holds in this case.
4. Inductive case: $tl = tr$ for some *Token* t and some *TokenList* l.
5. Induction hypothesis: $\mathsf{unlistify}(\mathsf{tlistify}(r)) = r$
6. Then, $\mathsf{unlistify}(\mathsf{tlistify}(tr)) = \mathsf{unlistify}(t : \mathsf{tlistify}(r))$ by definition of tlistify.
7. We apply the definition of unlistify, and get $\mathsf{unlistify}(t : \mathsf{tlistify}(r)) = t\mathsf{unlistify}(\mathsf{tlistify}(r))$.
8. We apply the induction hypothesis to that and get $t\mathsf{unlistify}(\mathsf{tlistify}(r)) = tr$.
9. Therefore, the lemma holds in the recursive case.
10. From 3 and 9, the lemma holds in all cases by induction.

□

Theorem 8.0.4. $\forall f \in \mathit{File}, \mathit{prepro}(\mathit{prepro}(f)) = \mathit{prepro}(f)$

Proof. For brevity, let g refer to $\mathsf{prepro}(f)$ for the rest of this proof.

1. Proof by induction on the structure of g.
2. Base case: $g = \lambda$. Then, by the definitions of prepro and the first definition of dispatch, $\mathsf{prepro}(g) = \lambda$, and so the theorem holds.
3. Induction hypothesis: Let $g = l$ `\n` h. Then, $\mathsf{prepro}(h) = h$
4. Inductive case 1: g is a *MacroLine* (either type) followed by another *File*.
5. In this case, as g is an output of prepro, we have a contradiction with lemma 8.0.1, therefore, the Theorem holds in this case.
6. Inductive case 2: g is a *TokenList* followed by another *File*.

7. Then, by the definitions of prepro and the last definition of dispatch, and by lemma 8.0.2, we have that $\mathsf{prepro}(g) = \mathsf{dispatch}(l \setminus \mathtt{n}\ h, \sigma, (0, 0, [], \emptyset, \mathtt{1}, 0)) = \mathsf{unlistify}(\mathsf{tlistify}(l)) \setminus \mathtt{n}\ \mathsf{dispatch}(h, \sigma, (0, 0, [], \emptyset, \mathtt{1}, 0))$.

8. By applying the definition of prepro backwards, we rewrite that as $\mathsf{prepro}(g) = \mathsf{unlistify}(\mathsf{tlistify}(l)) \setminus \mathtt{n}\ \mathsf{prepro}(h)$.

9. We then apply lemma 8.0.3 and get $\mathsf{prepro}(g) = l \setminus \mathtt{n}\ h = g$, so the Theorem holds in the inductive case.

10. From 2, 6 and 10, we conclude by induction hat the theorem holds in all cases.

□

Result: Each line stays the same when applying prepro again, by induction on lines. Note: IH must include both the equality of the lines and that the state and mapping are default/empty. Just the equality would not do.

Chapter 9

Predicates over the state(s)

We can now come back to our second research question of proving the correctness of C files with preprocessing macros.
We express the correctness requirements on a file with predicates over that file, such as "The file shall define a constant or constant macro called `CHAR_BIT` with a value of at least 8." (cf. [8], 5.2.4.2.1) This cannot simply be a predicate over any one state, for two reasons: For one, the macro and the program constants are kept in two separate states. And secondly, we wish to make sure that the definition is only added in the file, not previously present.
To be able to define such predicates properly, we need to define a function that gives us access to the preprocessing mappings after the file has been preprocessed. We therefore define this function prepro' as follows. Note that this is exactly the same as prepro in section 6.1, except that it passes the mapping back as well.

$$\mathsf{prepro}' : \mathit{File} \rightharpoonup \mathit{File} \times M$$
$$\mathsf{prepro}'(f) = \mathsf{dispatch}'(f, \sigma, (0, 0, [], \emptyset, \mathtt{1}, 0))$$

$$\mathsf{dispatch}' : \mathit{File} \times M \times S \rightharpoonup \mathit{File} \times M$$
$$\mathsf{dispatch}'(\lambda, m, s) = (\lambda, m)$$

$$\mathsf{dispatch}'(\texttt{\#define}\ i\ t\ \backslash\texttt{n}\ f, m, s) = (r'\ \backslash\texttt{n}\ r'', m'')$$
$$\textbf{where}\ (r'', m'') \coloneqq \mathsf{dispatch}'(f, m', s')$$
$$\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r)$$
$$\textbf{where}\ (m', s', r) \coloneqq \mathsf{om}(m, s, i, \mathsf{tlisitfy}(t))$$

$$\mathsf{dispatch}'(\texttt{\#define}\ i\ \texttt{(}\ a\ \texttt{)}\ t\ \backslash\texttt{n}\ f, m, s) = (r'\ \backslash\texttt{n}\ r'', m'')$$
$$\textbf{where}\ (r'', m'') \coloneqq \mathsf{dispatch}'(f, m', s')$$
$$\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r)$$
$$\textbf{where}\ (m', s', r) \coloneqq \mathsf{fm}(m, s, i, \mathsf{alisitfy}(a), \mathsf{tlisitfy}(t))$$

$$\mathsf{dispatch}'(t\ \backslash\texttt{n}\ f, m, s) = (r'\ \backslash\texttt{n}\ r'', m'')$$
$$\textbf{where}\ (r'', m'') \coloneqq \mathsf{dispatch}'(f, m', s')$$
$$\textbf{where}\ r' \coloneqq \mathsf{unlistify}(r)$$
$$\textbf{where}\ (m', s', r) \coloneqq \mathsf{tl}(m, s, \mathsf{tlisitfy}(t))$$

Then, we need to consider four separate states, the *Program* that would result from it's expansion, and the intermediate list of *Token*s to express the above predicate on some given *File* f:

$$\forall m, m' \in M, \forall s, s' \in P, \forall p \in \mathit{Program}, \forall t \in [T],$$
$$((\mathsf{prepro}'(f, m)_0 = t \wedge \{\{p\}\} = t \wedge \mathsf{prepro}'(f, m)_1 = m' \wedge s \overset{p}{\Rightarrow} s'$$
$$\wedge\ \mathsf{expand}(m, [[\texttt{CHAR\_BIT}]], \emptyset, \texttt{CHAR\_BIT})\uparrow \wedge\ s(\texttt{CHAR\_BIT})\uparrow)\ \Longrightarrow$$
$$([\![\mathsf{expand}(m', [[\texttt{CHAR\_BIT}]], \emptyset, \texttt{CHAR\_BIT})]\!]^{\sigma}_{s'} \geq 8$$
$$\vee [\![\texttt{CHAR\_BIT}]\!]_{s'} \geq 8))$$

As it can be seen that even this relatively simple predicate leads to a rather complex formula, and writing such a formula out fully should be done only as rarely as possible to avoid errors, I therefore introduce various higher-order functions that produce predicates over a *File*, such as the one above.

We then write this higher-order function like this:
(The name is an abbreviation for Macro Constant or Constant Predicate.)

$$\mathsf{MCCP} : (\Sigma \times (\mathbb{N} \to \mathbb{B})) \to (\mathit{File} \to \mathbb{B})$$
$$\mathsf{MCCP}(i, f) = \forall m, m' \in M, \forall s, s' \in P, \forall p \in \mathit{Program}, \forall t \in [T],$$
$$((\mathsf{prepro'}(f, m)_0 = t \wedge \{\{p\}\} = t \wedge \mathsf{prepro'}(f, m)_1 = m' \wedge s \overset{p}{\Rightarrow} s'$$
$$\wedge\ \mathsf{expand}(m, [[\mathtt{i}]], \emptyset, \mathtt{i})\uparrow \wedge\ s(\mathtt{i})\uparrow) \implies$$
$$(f([\![\mathsf{expand}(m', [[\mathtt{i}]], \emptyset, \mathtt{i})]\!]^{\sigma}_{s'}) \vee f([\![\mathtt{i}]\!]_{s'})))$$

And then we can write the above example as

$$\mathsf{MCCP}(\mathtt{CHAR\_BIT}, \lambda v.\ v \geq 8)$$

However, other requirements may set multiple constants in relation to each other. For example, it may be required that $\mathtt{UCHR\_MAX} = 2^{\mathtt{CHAR\_BIT}} - 1$ must hold (cf. [8], 5.2.4.2.1). For this, we define another higher-order function, which also provides a function that maps *Identifier*s to natural numbers:

$$\mathsf{MCCAP} : (\Sigma \times (\mathbb{N} \times (\Sigma \rightharpoonup \mathbb{N}) \to \mathbb{B})) \to (\mathit{File} \to \mathbb{B})$$
$$\mathsf{MCCAP}(i, f) = \forall m, m' \in M, \forall s, s' \in P, \forall p \in \mathit{Program}, \forall t \in [T],$$
$$((\mathsf{prepro'}(f, m)_0 = t \wedge \{\{p\}\} = t \wedge \mathsf{prepro'}(f, m)_1 = m' \wedge s \overset{p}{\Rightarrow} s'$$
$$\wedge\ \mathsf{expand}(m, [[\mathtt{i}]], \emptyset, \mathtt{i})\uparrow \wedge\ s(\mathtt{i})\uparrow) \implies$$
$$(f([\![\mathsf{expand}(m', [[\mathtt{i}]], \emptyset, \mathtt{i})]\!]^{\sigma}_{s'}, v(m', s')) \vee f([\![\mathtt{i}]\!]_{s'}, v(m', s'))))$$

$$v : M \times P \to (\Sigma \rightharpoonup \mathbb{N})$$
$$v(m, s)\lambda i.\ \begin{cases} [\![\mathsf{expand}(m, [[\mathtt{i}]], \emptyset, \mathtt{i})]\!]^{\sigma}_{s} & \textbf{iff } m(i)\downarrow \\ [\![i]\!]_s & \textbf{iff } m(i)\uparrow \end{cases}$$

And we can then write the above requirement as

$$\mathsf{MCCAP}(\mathtt{UCHAR\_MAX}, \lambda v, f.\ v = 2^{f(\mathtt{CHAR\_BIT})-1})$$

Chapter 10

Discussion

10.1 Expansion to other directives

The C standard specifies a number of other preprocessing directives besides macro expansion, such as conditional directives (`#if, #ifdef,...`) and source file inclusion, among others. (For a full list see [8], 6.10.) In addition, there are also the `#` and `##` operators in macro replacement lists, which were also not considered in this thesis. However, to model the full C preprocessor, all of this needs to be included as well. Therefore, I will now briefly discuss how each of the directives and features could be added to the model, without giving any explicit equations for this.

- For the conditional directives, one needs to define the evaluation function for the conditions, which is trivial if one has the syntax tree corresponding to the constant expression. In addition, one must also keep track of how many nested if groups one is in, both those that should be skipped and those that should not.
  This could be achieved by adding two more integers to the state that is carried between lines, one for each kind of nesting (which I shall call s and c), such that if one is currently in a group that should be skipped($s > 0$), all `#if` and `#ifdef` directives just increase s by one, and all `#endif` directives decrease it by one.
  And if the current group should not be skipped ($s = 0$), then any `#if` and `#ifdef` directives increase either s or c y one, depending on whether the condition is true, and all `#endif` directives decrease c by one. This is only a rough outline, but it should serve as a starting point for anyone who wishes to expand this model.

- Regarding file inclusion, the standard only prescribes that a file name might need to undergo macro expansion before it can be included, but leaves everything else implementation-defined. This means that in a

model, we can only fall back to a "magic function" that maps the filename onto a *File*, and act as though the lines of that *File* had been in our original *File* instead of the inclusion directive.

- The `#` operator is easily added to the model by modifying the type of the state M to also allow negative numbers to represent parameters that should be stringified, and adapting the repl_int function to perform this process, but that is not a particularly large effort, though it requires string literals to be a meaningful concept, which is why it was not done in this thesis. The `##` operator is also not difficult to model, the largest difficulty here will be in determining whether something is a valid preprocessing token.

- The `#undef` directive is trivial to implement as a modification of the macro mapping state, it was not done in this thesis solely to keep the model as simple as possible.

- Keeping track of line numbers is possible by adding another integer to the state passed between lines that is incremented each time, and a special case should be added to the definition of tlr to make sure that the macro `__LINE__` always expands to the current line number. Then, the implementation of the line control directive is merely to change this additional integer as indicated by the argument to the directive.

- As regards the error, pragma and null directives, they all likely should simply be treated as no-ops, as we are interested in the behaviour of the result of preprocessing, and these directives do not alter it, except for the floating point directives, which must be somehow preserved into the C program (as they follow it's block structure), presumably using some sort of non-syntactical marker that is left in place of the pragma directive.

10.2 Expansion to C

As mentioned in the introduction, the preprocessor is agnostic with regard to its target language, up to the definition of what constitutes a token. As such, expanding the model of the preprocessor to target C merely requires expanding this part of the grammar and does not require any change to the definitions.
On the other hand, the toy language used in this thesis is rather far removed from C, and so one may expect the predicates to test file correctness are not very useful anymore after the transition. However this is only somewhat correct: While the evaluation function will certainly become far more complex, and will also need to contend with the type system, the general model of those predicates is still sound as long as the outputs of a function depend

only on it's inputs.
This is not the case for all standard functions, such as `strtok`, but it is true of most functions in the standard library, and in any case, the result type of the evaluation function can be modified to also include a new program state if an evaluation does change the state.
In conclusion, while the transition from our toy language to C as a target language does certainly require some changes, I am confident that the basic structure and model presented in this thesis will form a solid foundation to build on for any future work in this direction.

Bibliography

[1] TIOBE Software popularity index. https://www.tiobe.com/tiobe-index/.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[3] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 5–21, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Valery A Nepomniaschy, Igor S Anureev, and AV Promskii. Towards verification of c programs: Axiomatic semantics of the c-kernel language. *Programming and Computer Software*, 29(6):338–350, 2003.

[5] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[6] Robbert Jan Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud Universiteit Nijmegen, 2015.

[7] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. *SIGPLAN Not.*, 46(10):805–824, oct 2011.

[8] ISO/IEC. C standard working draft N2731 (likely to be formalised as ISO/IEC 9899:2023). http://open-std.org/jtc1/sc22/wg14/www/docs/n2731.pdf.

[9] University of Glasgow. *Haskell2010 Reference, Data.Maybe*, 2010. https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Maybe.html.

[10] Oracle Corporation. *Java SE 17 & JDK 17 Reference, java.util.Optional*<T>, 2021. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Optional.html.

Appendix A

Common.hs

```haskell
module Common where

import qualified Data.Set as Set

type Token = [Char]
type Identifier = [Char]
data TokOrInt = Token Token | Int Int deriving (Eq, Show)
data TokenList = Lambda | C Token TokenList
data ArgumentList = Arg Identifier | CA Identifier ArgumentList

instance Show ArgumentList where
  show (Arg a) = a
  show (CA a al) = a ++ ", " ++ show al

alistify :: ArgumentList -> [Token]
alistify a = alistify' a Set.empty
  where
    alistify' (Arg a) s = [a]
    alistify' (CA a al) s =
      if Set.member a s
        then error "Duplicate Argument name"
        else a : alistify' al (Set.insert a s)
```

Appendix B

PreProcess.hs

```
module PreProcess (Line (OM, FM, TL), emptyState) where

import Common
import Data.List (elemIndex)
import Data.Maybe (fromJust, isJust, isNothing)
import qualified Data.Set as Set

data Line = OM Token TokenList | FM Token ArgumentList TokenList |
     TL TokenList

data File = FileEmpty | FileCons Line File

data StateInt = ZERO | ONE | TWO

type Mapping = Token -> Maybe (Int, [TokOrInt])

type State = (StateInt, Int, [[Token]], Set.Set Token, Token, Int)

prepro :: File -> File
prepro f = dispatch f emptyState (ZERO, 0, [], Set.empty, "1", 0)

rmvlines :: File -> [Token]
rmvlines FileEmpty = []
rmvlines (FileCons (TL t) f) = (tlistify t) ++ (rmvlines f)

dispatch :: File -> Mapping -> State -> File
dispatch FileEmpty m s = FileEmpty
dispatch (FileCons (OM i t) f) m s = FileCons (TL (foldr C Lambda
    r)) (dispatch f m' s')
  where
    (m', s', r) = om m s i (tlistify t)
dispatch (FileCons (FM i a t) f) m s = FileCons (TL (foldr C
    Lambda r)) (dispatch f m' s')
  where
```

```
    (m', s', r) = fm m s i (alistify a) (tlistify t)
dispatch (FileCons (TL t) f) m s = FileCons (TL (foldr C Lambda r)
    ) (dispatch f m' s')
  where
    (m', s', r) = tl m s (tlistify t)

om :: Mapping -> State -> Token -> [Token] -> (Mapping, State, [
    Token])
om m s i l = (modMapping m i (0, map Token (l)), s, [])

fm :: Mapping -> State -> Token -> [Identifier] -> [Token] -> (
    Mapping, State, [Token])
fm m s i a l = (modMapping m i res, s, [])
  where
    res = (length a, map (repind a) (l))

tl :: Mapping -> State -> [Token] -> (Mapping, State, [Token])
tl m s t = (m, s', t') where (s', t') = tlr m s t

tlr :: Mapping -> State -> [Token] -> (State, [Token])
tlr m s@(ZERO, 0, [], u, "1", 0) [] = (s, [])
tlr m s@(ZERO, 0, [], u, "1", 0) (t : ts) =
  if Set.member t u || isNothing (m t)
    then (s, t : snd (tlr m s ts))
    else
      if n == 0
        then (fst (tlr m s ts), snd (tlr m (ZERO, 0, [], Set.
    insert t u, "1", 0) r) ++ snd (tlr m s ts))
        else tlr m (ONE, n, replicate n [], u, t, 0) ts
  where
    r = filterTokens r' --if n==0, this will be the same list, but
     this satisfies the types.
    Just (n, r') = m t
tlr m (ONE, n, ls, u, t, 0) ("(" : ts) = tlr m (TWO, n, ls, u, t,
    0) ts
tlr m (ONE, n, ls, u, t, 0) ts = (s', t : ts') where (s', ts') =
    tlr m (ZERO, 0, [], Set.empty, "1", 0) ts
tlr m (TWO, 1, ls, u, t, 0) ("," : ts) = error "Comma to end
    argument list?"
tlr m (TWO, n, ls, u, t, 0) ("," : ts) = tlr m (TWO, n - 1, ls, u,
     t, 0) ts
tlr m (TWO, 1, ls, u, t, 0) (")" : ts) = (s', (expand m ls u t) ++
     ts') where (s', ts') = (tlr m (ZERO, 0, [], u, "1", 0) ts)
tlr m (TWO, n, ls, u, t, d) (ct : ts) = tlr m (TWO, n, lsm, u, t,
    if ct == "(" then d + 1 else d) ts
  where
    lsm = replaceIndex ls (nt - n) ((ls !! (nt - n)) ++ [ct])
    Just (nt, r) = m t
```

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten ([] : bs) = flatten bs
flatten ((a : as) : bs) = a : flatten (as : bs)

replInt :: [[Token]] -> TokOrInt -> [Token]
replInt ls (Int i) = ls !! i
replInt ls (Token t) = [t]

expand :: Mapping -> [[Token]] -> Set.Set Token -> Token -> [Token
    ]
expand m ls u t = snd (tlr m (ZERO, 0, [], Set.insert t u, "1", 0)
     (flatten (map (replInt ls) r)))
  where
    Just (n, r) = m t

replaceIndex :: [a] -> Int -> a -> [a]
replaceIndex (a : as) 0 r = r : as
replaceIndex (a : as) n r = a : replaceIndex as (n -1) r

filterTokens :: [TokOrInt] -> [Token]
filterTokens [] = []
filterTokens ((Token t) : ts) = t : filterTokens ts

tlistify :: TokenList -> [Token]
tlistify Lambda = []
tlistify (C t tl) = t : tlistify tl

repind :: [Token] -> Token -> TokOrInt
repind a b =
  if isJust (elemIndex b a)
    then Int (fromJust (elemIndex b a))
    else Token b

emptyState :: Mapping
emptyState t = Nothing

modMapping :: Mapping -> Token -> (Int, [TokOrInt]) -> Mapping
modMapping m t r t' =
  if isJust (m t) && m t /= Just r
    then error "Redefining existing macro!"
    else if t == t' then Just r else m t'
```

Appendix C

PostProcess.hs

```
module PostProcess (eval, File (..), Statement (..), Expression
    (..), ExpressionList (..), BinOp (..)) where

import Common
import Data.Map ((!))
import qualified Data.Map as Map
import Data.Maybe (isJust)

type PState = Map.Map Identifier ([Identifier], Maybe Expression)

data File = FS Statement | FC Statement File

data Statement = FDecl Identifier ArgumentList | FDefn Identifier
    ArgumentList Expression | CDefn Identifier Expression

data Expression = LIT Integer | IDENT Identifier | FUNC Identifier
     ExpressionList | BI Expression BinOp Expression

data ExpressionList = ELS Expression | ELC Expression
    ExpressionList

data BinOp = PLUS | MINUS | TIMES | OVER

instance Show File where
  show (FS s) = show s
  show (FC s f) = show s ++ "\n" ++ show f

instance Show Statement where
  show (FDecl i al) = "declare " ++ i ++ "(" ++ show al ++ ");"
  show (FDefn i al e) = "define " ++ i ++ "(" ++ show al ++ ") = "
     ++ show e ++ ";"
  show (CDefn i e) = "const " ++ i ++ " = " ++ show e ++ ";"

instance Show Expression where
```

```
  show (LIT i) = show i
  show (IDENT i) = i
  show (FUNC i el) = i ++ "(" ++ show el ++ ")"
  show (BI l PLUS r) = "(" ++ show l ++ "+" ++ show r ++ ")"
  show (BI l MINUS r) = "(" ++ show l ++ "-" ++ show r ++ ")"
  show (BI l TIMES r) = "(" ++ show l ++ "*" ++ show r ++ ")"
  show (BI l OVER r) = "(" ++ show l ++ "/" ++ show r ++ ")"

instance Show ExpressionList where
  show (ELS e) = show e
  show (ELC e el) = show e ++ ", " ++ show el

class Unparseable u where
  unparse :: u -> [Token]

instance Unparseable File where
  unparse (FS s) = unparse s
  unparse (FC s f) = unparse s ++ unparse f

instance Unparseable Statement where
  unparse (FDecl i al) = ["declare", i, "("] ++ unparse al ++ [")"
    ,";"]
  unparse (FDefn i al e) = ["define", i, "("] ++ unparse al ++ [")
    ", "="] ++ unparse e ++ [";"]
  unparse (CDefn i e) = ["const", i, "="] ++ unparse e ++ [";"]

instance Unparseable Expression where
  unparse (LIT i) = [show i]
  unparse (IDENT i) = [i]
  unparse (FUNC i el) = [i, "("] ++ unparse el ++ [")"]
  unparse (BI l o r) = "(" : unparse l ++ unparse o ++ unparse r
    ++ [")"]

instance Unparseable ExpressionList where
  unparse (ELS e) = unparse e
  unparse (ELC e el) = unparse e ++ [","] ++ unparse el

instance Unparseable BinOp where
  unparse PLUS = ["+"]
  unparse MINUS = ["-"]
  unparse TIMES = ["*"]
  unparse OVER = ["/"]

instance Unparseable ArgumentList where
  unparse (Arg a) = [a]
  unparse (CA a al) = [a, ","] ++ unparse al

eval :: File -> Integer
eval f = evalC "main" (processF f Map.empty)
```

```
processF :: File -> PState -> PState
processF (FC s f) os = processF f (processS s os)
processF (FS s) os = processS s os

processS :: Statement -> PState -> PState
processS (CDefn i e) os =
  if Map.member i os
    then error ("Attempted to redefine constant: " ++ i ++ "!")
    else Map.insert i ([], Just e) os
processS (FDecl i a) os =
  if Map.member i os
    then
      if length al == length (fst (os ! i))
        then os
        else error ("Function declaration with wrong number of
    paramenters: " ++ i ++ "!")
    else Map.insert i (al, Nothing) os
  where
    al = alistify a
processS (FDefn i a e) os =
  if Map.member i os
    then
      if length al == length (fst (os ! i))
        then
          if isJust (snd (os ! i))
            then error ("Tried to redefine function" ++ i ++ "")
            else Map.insert i (al, Just e) os
        else error ("Function definition with wrong number of
    paramenters: " ++ i ++ "!")
    else Map.insert i (al, Just e) os
  where
    al = alistify a

evalC :: Identifier -> PState -> Integer
evalC i s =
  if Map.member i s
    then
      if null is
        then evalE e s Map.empty
        else error ("Tried to evaluate constant as function: " ++
    i ++ "!")
    else error ("Tried to evaluate unknown constant: " ++ i ++ "!"
    )
  where
    (is, Just e) = s ! i

evalF :: Identifier -> PState -> [Integer] -> Integer
evalF i s a =
```

```
  if Map.member i s
    then
      if length is == length a
        then evalE e s (Map.fromList (zip is a))
        else error ("Tried to evaluate function with wrong number
    of arguments: " ++ i ++ "!")
    else error ("Tried to evaluate unknown function: " ++ i ++ "!"
    )
  where
    (is, Just e) = s ! i

evalE :: Expression -> PState -> Map.Map Identifier Integer ->
    Integer
evalE (BI l PLUS r) s m = evalE l s m + evalE r s m
evalE (BI l MINUS r) s m = max 0 (evalE l s m - evalE r s m)
evalE (BI l TIMES r) s m = evalE l s m * evalE r s m
evalE (BI l OVER r) s m = evalE l s m `div` evalE r s m
evalE (LIT i) _ _ = i
evalE (IDENT i) s m =
  if Map.member i m
    then m ! i
    else evalC i s
evalE (FUNC i es) s m = evalF i s (map (\e -> evalE e s m) (
    elistify es))

elistify :: ExpressionList -> [Expression]
elistify (ELS e) = [e]
elistify (ELC e es) = e : elistify es
```