FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Pattern matching in C11
Student:	Jan Jindráček
Supervisor:	Ing. Filip Křikava, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

Pattern matching is a general mechanism found in many programming languages for checking a value against a certain pattern.
A matching pattern can also be used to deconstruct a value into its constituent parts.
This work will study the possibility of adding such a feature to the C programming language.
Analyze the pattern matching in different programming languages and propose syntax and semantic suitable for the C programming language.
Implement a prototype including suitable test suite and proper documentation.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 30, 2019

Acknowledgements

I wish to thank Dr. Filip Křikava for his patience in guiding me to this point.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 16, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Jindráček, Jan. *Pattern matching in C11.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Pattern matching je mechanismus který se využívá ve velkém množství programovacích jazyků, kde je využit jako způsob jak prověřit, jestli daný výraz, objekt, a nebo proměnná má vlastnosti nebo obsah specifikovaný daným vzorem. Tato práce se zaměří na to, zda-li a jak se dá taková feature přidat do jazyku C. Budu tedy implementovat prototyp pattern matchingu do jazyku C, včetně testů a dokumentace.

Klíčová slova MPS, mbeddr, C, jazyk, rozšíření, AST projekce, gramatika, unifikace, vzory

Abstract

Pattern matching is a general mechanism found in many programming languages for checking a value against a certain pattern. A matching pattern can also be used to deconstruct a value into its constituent parts. This work will study the possibility of adding such a feature to the C programming language. Analyze the pattern matching in different programming languages and propose syntax and semantic, suitable for the C programming language. Implement a prototype including suitable test suite and proper documentation.

Keywords MPS, mbeddr, C, language, extension, AST projection, grammar, unification, patterns

Contents

Introduction

There is a growing interest in Functional Programming (FP). As a result, we can see mainstream languages adopting many of the FP concepts. For example the introduction of functional interface in Java [5] or lambda expression in C++ [6]. So far these adoptions were mainly in high-level, object-oriented programming languages. In this thesis, I decided to implement an FP concept in a lower-level language. Concretely, to add pattern matching into the C programming language. The goal is to create a new switch-like statement, match, allowing one to pattern match on C data types in a similar fashion to Racket [7], OCaml [8] or Scala [3].

The reason why I chose C instead of, Java or C++, is that C is a smaller language and has a simpler type system which will reduce the design and implementation effort. Also, the match statement will be particularly useful for pattern matching composite data types such are C structs or unions. It will allow programmers to decompose these structures bringing their fields into the current block scope and thus simplifying their access and making the code easier to reason about.

There are multiple ways of adding such a feature into C. In this thesis I experimented with a couple and finally chose to use JetBrains MPS [9], a meta-programming system to design DSL and language extensions. The result is C language with the match statement, implemented in a complete language workbench, that comes with an editor, compiler and debugger.

Chapter 1

Pattern matching

Pattern matching is a mechanism which compares a data sequence against a pattern. The pattern usually consists of literals, deconstructed arrays and structs, tuples, variables and wildcards (A good example of variable and wildcard patterns is here [2]). In order for a data sequence to be validated against a pattern, the pattern must be an exact match.

In order to showcase pattern matching, I have implemented a simple coin sorting machine in Rust and Scala. The coin sorting machine will simply return a value of the coin inserted and, if the coin is a quarter, will print out the name of the state in which it was minted.

Figure 1.1 shows the implementation in the Rust programming language. In it, one can use `enum` keyword to define an enumeration containing different elements. Notice that the last value of the enumeration has a String type appended to it. Then, all that is left is to define a function which will match the enumeration value of the `coin` variable to one of the four options. The function will return a number based on the enumeration type. What is interesting is that one can define additional behaviour based on the string appended to the `Quarter` enumeration value - the state in which the quarter was minted.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(String)
}

fn value_in_cents(coin: Coin) -> u32 {
  match coin {
    Coin::Penny => 1,
    Coin::Nickel => 5,
    Coin::Dime => 10,
    Coin::Quarter(state) => {
      println!("State quarter from \{:?\}!", state);
      25
    }
  }
}
```

Figure 1.1: Coin sorting machine in Rust.

Figure 1.2 shows the second example of the implementation of coin sorting machine. Here, I have used an object called `Demo` - inside it, there is one main function and several `Coin` classes defined. These classes serve as an enumeration, similar to previous example. Then, three variables are defined with different subclasses extending the `Coin` class. The match expression then determines which numeric value will be printed by `println` function. Similarly to the Rust example, the last case, `Quarter` will print out a name of the state in which it was minted.

```
object Demo {
   def main(args: Array[String]) {
      val penny = new Penny()
      val nickel = new Nickel()
      val quarter = new Quarter("Washington")

      for (coin <- List(penny, nickel, quarter)) {
         val amount = coin match {
            case Penny() => 1
            case Nickel() => 5
            case Dime() => 10
            case Quarter(name) => {
              println(name)
              25
            }
         }
         println(amount)
      }
   }
   sealed trait Coin
   case class Penny() extends Coin
   case class Nickel() extends Coin
   case class Dime() extends Coin
   case class Quarter(name: String) extends Coin
}
```

Figure 1.2: Coin sorting machine in Scala

1.1 Guard conditions and pattern matching

A guard conditions is a condition which is appended to a pattern inside a pattern matching mechanism. This condition is evaluated only after the main pattern passes - if it fails, the case will not be evaluated as valid and another case might be run instead. Figure 1.3 shows an example of how a guard condition can be added to a pattern matching case in Rust. The function in the example will only print a `"SQUARE!"` when it's width and height are equal.

```
enum Geometry {
    Rectangle(u32, u32), //width and height
    Triangle(u32, u32, u32) //sides
}

fn typeofshape( g : Geometry ) {
  match g {
    Geometry::Rectangle(x, y) if x == y => println!( "SQUARE!\n" )
    Geometry::Rectangle(x, y) if x != y => println!( "RECTANGLE!\n" )
    Geometry::Triangle(a, b, c ) => println!("TRIANGLE!\n")
}
```

Figure 1.3: Determining the shape of an object in Rust

1.2 Pattern matching and Algebraic Data Types

An Algebraic Data Type (ADT) is a composite data structure containing two or more different types within itself. There are two main types of Algebraic Data Type (ADTs) - products and sums. An ADT which combines a recursive sum type of product types is called a general ADT.

The key difference is that with an ADT sum, one can perform something called "exhaustiveness checking". What this means is that a compiler or an interpreter can alert a programmer that there is an option possible within the data structure which they have not accounted for. A good example of this would be an enumeration, or a boolean type - since there is a limited number of options for both.

Product type, conversely, has no limit of possible options. An example of a product type would be a structure containing a string. Since a string has no finite length, there is an infinite number of different possible product structures.

An example of a general ADT is a class structure with inheritance, with the parent class having a string attribute. The class system contains a limited number of options - a sum type. While the string attribute does not - therefore it would be a product type. Figure 1.4 shows an example of pattern matching being done on a general ADT. The integer values inside the `Rgb` enumeraiton value are a product type, while the enumerations themselves are a sum type. This example is written in Rust and it simply prints out a text into the console dependent on the sum part of the hybrid type in the match expression.

```
enum Color {
   Rgb(i32, i32, i32),
   Hsv(i32, i32, i32)
}
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}
```

```
fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));
    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        },
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!(
                "Change the color to hue {}, saturation {}, and value {}",
                h,
                s,
                v
            )
        }
        _ => ()
    }
}
```

Figure 1.4: Breaking down ADTs using match in Rust. Source: [4]

Other examples of a general ADT are list or tree structures - one can define an enumeration of a list item and nil. The list item sum type would contain another list and a value (product type). A tree structure could be a leaf, tree and nil, with a tree and a list containing a product. One can also define a recursive ADT, where the sums can be numeric operations and product being numbers - in this way, the general ADT would be a data type representing a numeric expression. An example is shown in Figure 2.5.

Chapter 2

Design

In this chapter I will discuss a design of a pattern matching functionality in C. Furthermore, I will describe the semantics and syntax of the match statement, as they relate to its translation to plain C. Most of its syntax is either rooted directly in plain C, or was inspired by languages like Rust and Scala. This was done for familiarity reasons. Programmers are more likely to be able to work with concepts which they already know either from C, or from other languages.

2.1 Pattern matching in C - overview with examples

The match statement used for pattern matching in my solution has two key parts: an expression to evaluate and a list of cases. The cases can contain three concepts overall, a case pattern to match, an guard condition (Figure 1.3 shows an example of the guard condition) similar to an `if` statement, and a code block at the end.

If the matched expression gets validated by a case, the cases code block will be executed. The precise details of the execution are in 2.2.3.

The statements will be executed if the matched expression matches a pattern inside the case. Cases in a match are disjunct, meaning one expression cannot be matched by multiple cases. In case an expression could be matched by several cases, it will only be matched by the first one. There are six distinct concepts which can be used in a case pattern: wildcards, variables, literals and strings, enumerations, structs and unions. In the following subsections, I will present examples of the different pattern cases including how the match statement is translated into regular C code.

2.1.1 Wildcards and variables

Wildcard (the underscore character inside the Figure 2.1) means that any value put inside the match statement will be regarded as valid. Variable (example shown in figure 2.2) also validates any value inside the match, but that value can also be used inside the case statement list.

```
int compare = 5;
match ( compare ) {
  case _ : {
    printf("Number!\n");
  }
}
```

```
int compare = 5;
{
  int matchMe = compare;
  int onePassedMatch = 0;
  if (true && !onePassedMatch) {
    onePassedMatch = 1;
    {
      printf("Number!\n");
    }
  }
}
```

Figure 2.1: Translation of wildcard into C code

```
int compare = 5;
match ( compare ) {
  case x : {
    printf("%d\n", x);
  }
}
```

```
int compare = 5;
{
  int matchMe = compare;
  int onePassedMatch = 0;
  if (true && !onePassedMatch) {
    int x = matchMe;
    onePassedMatch = 1;
    {
      printf("%d\n", x);
    }
  }
}
```

Figure 2.2: Translation of variable case pattern into C code

2.1.2 Case pattern literal matching

One can match literals and strings in C to the expression inside the match statement. These literals have to be of the same type as the matched expression (Example of literal matching in figure 2.3).

```
int compare = 1;
match ( compare ) {
  case 1 : {
    return 1;
  }
}
```

```
int compare = 1;
{
  int matchMe = compare;
  int onePassedMatch = 0;
  if (1 == matchMe && !
      onePassedMatch) {
    onePassedMatch = 1;
    {
      return 1;
    }
  }
}
```

Figure 2.3: Literal comparison inside a match statement

2.1.3 Matching of composite structures

Composite structures, structs and unions in C, have been added in order to facilitate pattern matching of complex data structures - Example is provided in figure 2.5. Pointers to structs and unions are also considered valid, however, it is not possible to compare multiple pointers, such as `int ** ptr` inside case pattern. Multiple pointer were not added into pattern matching since there was no discernible use for them.

```
typedef struct list list_t;
struct list_t {
  int value;
  list_t *list;
};
```

Figure 2.4: List used in examples of pattern matching

```
void matchNext(list_t* xs) {
  match ( xs ) {
    case list_t * { 5,  next } : {
      printf("Found five!\n");
      matchNext(next);
    }
    case list_t * { _,  next } : {
      matchNext(next);
    }
    case list_t * { _,  NULL } : {
      printf("Last!\n");
    }
  }
}
```

```
void matchNext(list_t *xs)
{
  {
    int onePassedMatch = 0;
    list_t * matchMe = xs;
    if (matchMe != NULL
     && matchMe->value == 5
     && matchMe != NULL
     && true
     && !onePassedMatch) {
      list_t *next = matchMe->list;
      onePassedMatch = 1;
      {
        printf("Found five!\n");
        matchNext(next);
      }
    }
    if (true
     && matchMe != NULL
     && true
     && !onePassedMatch) {
      list_t *next = matchMe->list;
      onePassedMatch = 1;
      {
        matchNext(next);
      }
    }
    if (true
     && matchMe != NULL
     && matchMe->list == 0
     && true
     && !onePassedMatch) {
      onePassedMatch = 1;
      {
        printf("Last!\n");
      }
    }
  }
}
```

Figure 2.5: Match statement, pattern matching inside a list (For the list definition, see figure 2.4)

2.1.4 Implicit null checks

Since the match statement case patterns can work with pointers to structures and unions, there is always a possibility that the matched expression will resolve into a structure with some of its members being a null pointer. The case patterns have been designed in such a way that prevents an accidental runtime error by creating implicit null checks.

These null checks will be created automatically, based on whether the match expression is a pointer. Also, null checks will be created in case of struct comparison, when the struct itself contains a pointer. This however, does not mean that a null pointer comparison is not possible - the null checks are created only when a comparison would result in segmentation fault if not checked properly.

An example of an implicit null check can be found above - `matchMe != NULL` has been generated to that the access to `matchMe->value` will not cause an error in case `matchMe` happens to be null.

2.1.5 Guard conditions

The guard condition (example provided in figure 2.6) appended to a case pattern adds an additional check before a case pattern gets matched. If it fails, the matched expression will not be validated by its case. The matched expression will possibly be matched by another case after. It can contain the variables defined in the case pattern. The guard condition will only be run if the case pattern is already matched as true. The code inside the condition can contain function calls, literals and everything else one can put into a regular `if` statement. The implicit null checks are not placed here.

```
void divs(list_t* xs, list_t**
    result, int mod) {
  match ( xs ) {
    case list_t * { number,  next }
        if number % mod == 0 : {
      list* another = (((list*) (
          malloc(sizeof[list])))));
      another->list = NULL;
      another->value = number;
      (*result)->list = another;
      *result = another;
      divs(next, result, mod);
    }
    case list_t * { _,  next } : {
      divs(next, result, mod);
    }
  }
}
```

```
void divs(list_t *xs, list_t **
    result, int mod)
{
  {
    int onePassedMatch = 0;
    list_t * matchMe = xs;
    if (matchMe != NULL
     && matchMe != NULL
     && true
     && !onePassedMatch) {
      int number = matchMe->value;
      list_t *next = matchMe->list;
      if (number % mod == 0) {
        onePassedMatch = 1;
        {
          list_t *another = (((
              list_t *)((malloc(
              sizeof(list_t ))))));
          another->list = NULL;
          another->value = number;
          (*result)->list = another;
          *result = another;
          divs(next, result, mod);
        }
      }
    }
    if (true
     && matchMe != NULL
     && true
     && !onePassedMatch) {
      list_t *next = matchMe->list;
      onePassedMatch = true;
      {
        divs(next, result, mod);
      }
    }
  }
}
```

Figure 2.6: Struct pattern matching with a guard condition - list structure in the example can be found in figure 2.4

2.1.6 Examples of enumeration and ADT

Algebraic data types (Example in figure 2.8) can be expressed by using an enumeration to determine a type of a union containing the ADT types. Both the union and the enumeration are wrapped inside a struct. Here is an example of how pattern matching works inside the match statement.

```
typedef enum type type_t;
enum type {
  Add,
  Multiply,
  Value
};

typedef struct value value_t;
struct value {
  int val;
};

typedef union expr_data expr_data_t;
union expr_data {
  multiply_t *multiply;
  add_t *addition;
  value_t value;
};

typedef struct expr expr_t;
struct expr {
  type_t type;
  expr_data_t operation;
};

typedef struct multiply multiply_t;
struct multiply {
  expr_t left;
  expr_t right;
};

typedef struct add add_t;
struct add {
  expr_t left;
  expr_t right;
};
```

Figure 2.7: ADT defined by unions and structs

```
int eval(expr e) {
  match ( e ) {
    case expr { Add,  expr_data { _,
          add * { x,  y } } } : {
      return eval(x) + eval(y);
    }
    case expr { Multiply,  expr_data
         { multiply * { x,  y } } }
        : {
      return eval(x) * eval(y);
    }
    case expr { Value,  expr_data {
        _,  _,  value { x } } } : {
      return x;
    }
  }
  return 0;
}
```

```
int eval(expr_t e)
{
  {
    int onePassedMatch = 0;
    expr_t matchMe = e;
    if (matchMe.type == 0
     && true
     && matchMe.operation.addition
          != NULL
     && matchMe.operation.addition
          != NULL
     && true
     && !onePassedMatch) {
      expr_t x = matchMe.operation.
          addition->left;
      expr_t y = matchMe.operation.
          addition->right;
      onePassedMatch = 1;
      {
        return eval(x) + eval(y);
      }
    }
    if (matchMe.type == 1
     && matchMe.operation.multiply
          != NULL
     && matchMe.operation.multiply
          != NULL
     && true
     && !onePassedMatch) {
      expr_t x = matchMe.operation.
          multiply->left;
      expr_t y = matchMe.operation.
          multiply->right;
      onePassedMatch = 1;
      {
        return eval(x) * eval(y);
      }
    }
    if (matchMe.type == 2
     && true
     && true
     && true
     && !onePassedMatch) {
      int x = matchMe.operation.
          value.val;
      onePassedMatch = 1;
      {
        return x;
      }
    }
  }
  return 0;
}
```

Figure 2.8: Pattern matching an expression in the form of ADT - for the structures used in this example, see figure 2.7

2.1.7 Limitations

One cannot use any dynamic expressions, such as function calls, numeric operations (`1 + 2`) or array access cannot be used inside a case pattern. This is not allowed in order to ensure that the case pattern stays pure - always having the same result to the same set of data.

Another limitation of my work, is that a C array cannot be used inside a pattern. This feature was not added because of time constraints and for the lack of an implicit length of an array. This could be amended by creating another case pattern in order to match the first several elements of any array.

There is also a limitation on the scope of any variable defined with the variable case pattern - such variable can only be used inside a guard condition, or inside the code block appended to the case. This limitation in scope was done in order to create a code which is overall cleaner and easier to reason about.

Last limitation of my project is, that due time constraints, control of match case content similarities was not implemented. What this means is that one can define two of the same cases without it raising an error. This could have been fixed by expanding unreachable code detection - Section 2.5.

2.2 Semantics of the match statement

In this section, the semantics of the match statement will be explained in terms of the C statements, code blocks, variables and expressions to which the match statement is being translated to.

The match statement is translated to C as a code block, containing a definition of two variables - `onePassedMatch` and `matchMe`. The type of `matchMe` is decided based on the type of the expression being matched and its value is always the same as the matched expression. The type of `onePassedMatch` is an integer and its value is zero.

Next, in the code block, there is a series of `if` conditions. Each of those `if` conditions represents one case. The expressions inside the `if`s are connected by `&&`. The expressions are derived from a case pattern inside the case. Here is how the derivation looks:

Pattern name	Pattern	Translated expression
Wildcard	_	true
Variable	Valid C variable name	true
Literal	Valid C literal	matchMe == `literal`
Enumeration	C enumeration	matchMe == `enum`
String	C string	strcmp(matchMe, `str`) == false

One can also insert two more case patterns inside any case (examples - Fig: 2.5) - struct case pattern and union case pattern. The struct and union case patterns have a more complex translation pattern. Inside each of the struct or union case pattern is a type of the struct/union, or a pointer type of struct or a union. Then there is a list of other case patterns.

My solution during the translation process creates an access path from the `matchMe` variable through the struct/union case patterns in order to access the literal/enumeration/string case pattern inside the embedded union/struct patterns. The necessary null checks for the access path are created as well. Then, those access paths are used to create boolean expressions similarly to those mentioned in the table above.

2.2.1 Variable definitions inside cases

Inside the `if` statements, created by the case translation, there can be a series of variables defined thanks to the variable case pattern (example of multiple variable case patterns in figure 2.9). The type of these variables is dependent upon the variable case pattern. In case the variable case pattern is used as such: `case X {...}`, where X stands for the variable name, the type of the variable will be defined by the type of the matched expression. If, however, the variable is used inside the struct pattern, it will be defined thusly:

```
typedef struct e e_t;
typedef struct e2 e2_t;

struct e {
        int a;
        char* b;
        char c;
        e2_t e2;
}

struct e2 {
        int ab;
        int cd;
}
```

```
match(var) {
  //type of var is e_t
  case e_t {x, y, _, e2_t {a, b}}: {
    //Type of x is int,
    //Type of y is char*,
    //The wildcard is typeless,
    //Type of a is int,
    //Type of b is int
  }
}
```

Figure 2.9: An example of how a variable case pattern can be used.

2.2.2 Semantics of the guard condition

Since my solution supports guard conditions for case patterns (Explanation provided in subsection 2.1.5), this subsection will offer a semantic description of this concept - it will be translated into an `if` condition after the variables from the variable case pattern have been defined. Therefore, this condition can contain references to the variables mentioned previously.

2.2.3 Case code block

In order to facilitate pattern matching, the match cases needs a code block to run in case of a pattern match being successful. This code block will be executed only, and only after:

1. every null check inside the case pattern condition passes,
2. every case pattern expression inside the above condition is resolved as true,
3. no other case code block has been run (variable `onePassedMatch` was 0),
4. all variables from variable case pattern has been defined,
5. if added, the optional condition must be resolved as true,
6. variable `onePassedMatch` has been set to 1

If any of the conditions (1-3) were not met, the remaining steps (3-6) will not be executed and the next match case will be evaluated against the matched expression.

2.3 Syntax for the match statement

This section will cover the breakdown of the new syntax which I have added to plain C, in order for the reader to have a easier way to reason about the match statement. The complete match statement syntax can be found in figure 2.10.

```
match: 'match' '(' expression ')' '{'
('case' pattern ('if' condition)? ':' '{'
(statement)*
'}')+
'}'
pattern : struct_pattern
| literal_pattern
| variable_pattern
| wildcard_pattern
| union_pattern
struct_pattern : type pointer? '{' (pattern ',')* pattern '}'
union_pattern : type pointer? '{' (pattern ',')* pattern '}'
literal_pattern : character | string | integer | float | enumeration
variable_pattern : Valid C variable name
wildcard_pattern : '_'
type : Valid C type
pointer : '*'
character : Valid C character
string: Valid C string
integer: Valid C integer
float: Valid C float
statement: Valid C statement
expression: Valid C expression
condition: Valid C expression
```

Figure 2.10: Pattern matching grammar in Extended Backus-Naur Form (EBNF)

2.4 Typing rules

Since C is a typed language, my solution needs to be able to perform type checks of the programmers code in order to ensure that the match statement can be correctly translated into a C code.

2.4.1 Type inference for pattern variables

Variables in the case pattern match have their types derived from the expression in the match statement. This was done so that whenever one uses such variable inside the code block or guard condition appended to the case, the typing rules will ensure the validity of its use. An example of this is show in figure 2.11.

```
typedef struct list list_t;
struct list {
  int value;
  list_t* list;
};

void matchNext(list_t* xs) {
  match ( xs ) {
    case list_t * { 5,  next } : {
      printf("found five!\n");
      // The type of next is list_t*
      matchNext(next);
    }
    case list_t * { _,  NULL } : {
      printf("Last element!\n");
    }
  }
}
```

Figure 2.11: Case pattern variable typing rules

2.4.2 Explicit struct types

When a case pattern match contains a struct, the type has to be explicitly written during the struct pattern definition, with the possibility of using a * to signify that the expected match expression is a pointer - for an example, see figure 2.12.

The same logic applies when a struct pattern match contains multiple structs within it. Also, one does not need to define all attributes of the struct/union in a case pattern - an incomplete definition is just as valid.

```
typedef struct list list_t;
struct list {
  int value;
  list_t* list;
};

typedef struct addition addition_t;
struct addition {
  int a;
  int b;
};

void matchNext(list_t* xs) {
  match ( xs ) {
    case list_t * { 5,  next } : {
      printf("found five!\n");
      matchNext(next);
    }
    case list_t * { _,  NULL } : {
      printf("Last element!\n");
    }
    case addition_t { _ } : {
      // This will cause an error since
      // the type addition_t is not list_t*
    }
  }
}
```

Figure 2.12: Necessary explicit typing for structs

Chapter 3

Implementation

This chapter is dedicated to the details of how the match statement was implemented, which tools were used in order to achieve the goal of this thesis and their advantages and disadvantages, compared to different approaches to similar tasks.

3.1 Choosing implementation strategy

In order to start extending the C language, I needed to choose a specific way of extending a programming language. There were several options available:

- Directly changing the pre-processor (in the sense of Objective-C [12] or early C++ [13])
- Custom compiler (writing an extension in GCC or LLVM compilation pipeline)
- Using a language workbench (such as Xtext [18], MPS, Silver [16], Spoofax [14] or Metaedit+ [15])

Extending a language at the compiler level has its advantages - mainly not restricting the user to a certain Integrated Development Environment (IDE). The main disadvantage of this route is that it does not provide the user with a compatible debugger, nor with a compatible editor. Both of those would need to be developed separately. What this means is creating a custom highlighting syntax plugin for an IDE which supports C syntax in order for a user to be able to work with the extension, creating a compatible debugger and a plugin to fix semantic navigation for such IDE to support the language extension.

3.1.1 Solutions similar to mine

Another solution, created in AbleC [17], is extending the C syntax in order to be able to resolve ADTs [10] with pattern matching. This solution was made specifically to be used in conjunction with `union`-like constructs - datatypes.

A datatype contains information about which of its subtypes was defined. A match statement, defined in the AbleC solution, has an expression and a number of cases. The match statement then uses information about which type was defined in order to determine which case will be run.

3.1.2 Language workbench

Language workbench is a software tool, which is used to design Domain Specific Language (DSL) or to extend an existing language. It then produces an IDE, with which one can write, run and debug said language. This IDE will have semantic highlighting and syntax control. It will also be able to step-by-step debug the created language. This means that if one were to write a Java extension, it would be possible to debug the created language as if one was debugging regular Java code in another IDE. Also, in case of a compilation error, one will be able to see the error directly in the created language.

There are, however, two noticeable disadvantages of implementing a language extension in a workbench. The first disadvantage is that the programmer who will be using the extension will not be able to choose their own IDE. The second disadvantage is that language workbenches consume a noticeably higher portion of computation power than a compiler extension.

Since language workbench noticeably eases the implementation effort compared to the other ways of extending languages, I have chosen to implement my solution in it. There were two different language workbenches that were tried.

3.1.2.1 Xtext

The first language workbench that was tried for this solution was Eclipse Xtext. In Xtext, one designs the syntax for a DSL or a language extension in a similar way to EBNF, with which one creates parsing rules. The translation into a runnable code from a language extension of DSL is achieved by Xtext consuming the programmers code by using the parsing rules. Then, one can run another embedded Eclipse editor, in which one can use custom highlighting and run/debug the application created. This solution was abandoned in favor of MPS. There were several advantages of using MPS instead of Xtext:

- Xtext does not have support for header files or linker on the level of the newly created language
- Type and scope control
- Xtext is a code parser - meaning one needs to reason about ambiguous references

3.1.2.2 MPS

The language workbench chosen in order to implement the language extension was Jetbrains MPS. MPS is a projectional editor. This means that one does not work with a textual version of code. Rather, MPS uses it's autocomplete feature to help one to create an Abstract Syntax Tree (AST). This tree syntax is used both to design a language and to run it. The main advantage of this approach is that one does not need to worry about ambiguous concepts and text parsing. Each node in the AST tree has several ways with which it can be manipulated:

- Concepts - the syntax item inside a tree (More on concepts: Fig: 3.2)
- Editors - how will the syntax look
- Behavior - Utility methods which can be used by others
- Constraints - in what scope can a concept appear
- Typesystem - defining the types and type checks
- TextGen - a lambda-like language used for text generation
- DataFlow - a lambda-like language used to determine unreachable nodes
- Generator - main plan of how to translate the root nodes

In order to show in-depth the MPS language workbench there is a screenshot of the IDE provided below. The screenshot shown in figure 3.1 showcases the code example of the match statement, the Mbeddr modules and the node inspector. The match statement shown on the picture below is not actually text - as it was mentioned above, this is merely a projection of an AST created by autocomplete.

This approach however, has a couple of disadvantages - mainly that it somewhat restricts the programmers ability to freely rewrite code, as one needs to rely upon the autocomplete feature in order to be able to create the AST.

The node inspector serves as a way for a programmer to look up which type of node is currently being rewritten. The node inspector is mainly useful for debugging added concepts.

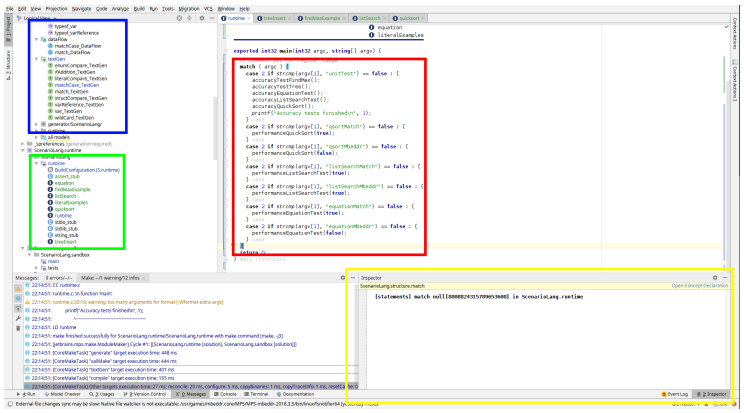

Figure 3.1: Screenshot of MPS Language workbench

Legend:

- Mbeddr C AST tree with match statement (Red)
- Language concepts (Blue)
- List of Mbeddr C internal/external modules (more on modules in 3.3.4) (Green)
- Inspector of AST nodes (Yellow)

Another screenshot shown in figure 3.2 shows how the node definition looks like. The concept in question is the match statement. Every node in MPS has three important categories: properties, children and references. The properties are values with primitive data types (including a string) which can be filled by a programmer. The children nodes are those that will be initialized after the creation of the parent node. Finally, the reference nodes - MPS allows for a node to be referenced by another - this is used when one needs to create concepts like variable and function references. Also, each node can inherit from another node and implement more than one interface (inheritance is especially important for Behaviour - see subsection: 3.1.2.2).

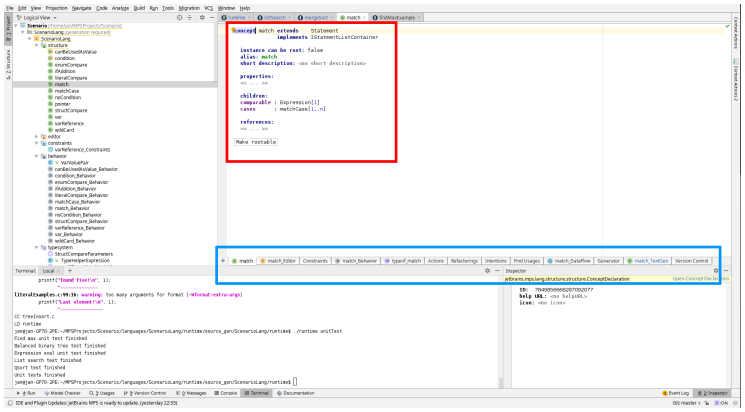

Figure 3.2: Screenshot of MPS Concept node editor

Legend:

- Definition of MPS concept, with children (other concepts) (Red)
- Other aspects of the MPS concept, such as UI Editor and typing rules (Blue)

The last screenshot in figure 3.3 is of the MPS debugger. Currently, there is a breakpoint set on the match statement itself. Here we can see a great advantage of the MPS language workbench - it has a support for integrating the new language concepts into its debugging UI. The MPS debugging menu is otherwise standard - breakpoints and watches are very similar to other comparable IDE.

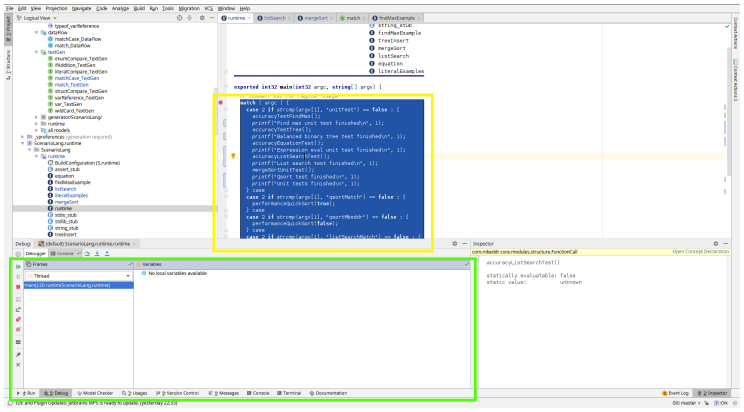

Figure 3.3: Screenshot of MPS debugger

Legend:

- Debugger of a custom language, hitting a breakpoint (Yellow)
- Debugger menu with standard options (Run, Continue, defined variables) (Green)

3.2 Mbeddr

Since the language primarily supported by MPS is Java, I needed a way to add C syntax into it without having to implement it myself in order to reduce implementation effort and because of time constraints. I have chosen to import the Mbeddr Development language package (devkit) [11].

It contains an improved version of C syntax (see section 3.3 for more details), preprocessor and build tools, custom reporting and error messaging. The devkit is read-only, so my solution is extending (inheriting from) and using (as a child concept) Mbeddr concepts. Mbeddr also automatically creates a makefile and supports it's own version of header files.

Another advantage of using Mbeddr is that it already has support for infering scopes of variables. My solution does not need to have scope constraints implemented within itself (except for the implicitly defined variables). This is especially useful for dealing with variables defined/used inside the statement lists appended to match cases.

My solution also does not need to create a generator, since it does not add any root nodes to the project. The generator which my solution uses has been fully provided by mbeddr extension (more on generators in the subsubsection 3.1.2.2).

Since MPS offers standard run and debug modes, mbeddr concepts contain a custom support for this as well. Both the match statement and its cases contain support for mbeddr debugging systems. One can go through the statement lists inside a case in a similar way one would go through `while` statement list.

3.2.1 Graph of usage

In order to showcase how the different elements mentioned in the implementation chapter work together, the graph of usage in figure 3.4 was created. In this graph, the yellow color is for the user, while the green is for a file or program which directly assists the user in creating and manipulating the AST and in generating the C code. Red color has been chosen to represent the three debugger levels which together with the MPS IDE work together to allow the user to debug their newly created language. Blue squares are short descriptions of important steps.

This graph also shows the the sections of the translation process, where a custom defined concept can enter and change the behaviour and results. "My Solution" + "Mbeddr C" are these changes - they change the MPS on-screen editor UI, they are a part of the build process and finally, MPS uses them in order to allow one to correctly debug the code.
For a reference on the internal and external modules shown in the graph, see subsection 3.3.4.

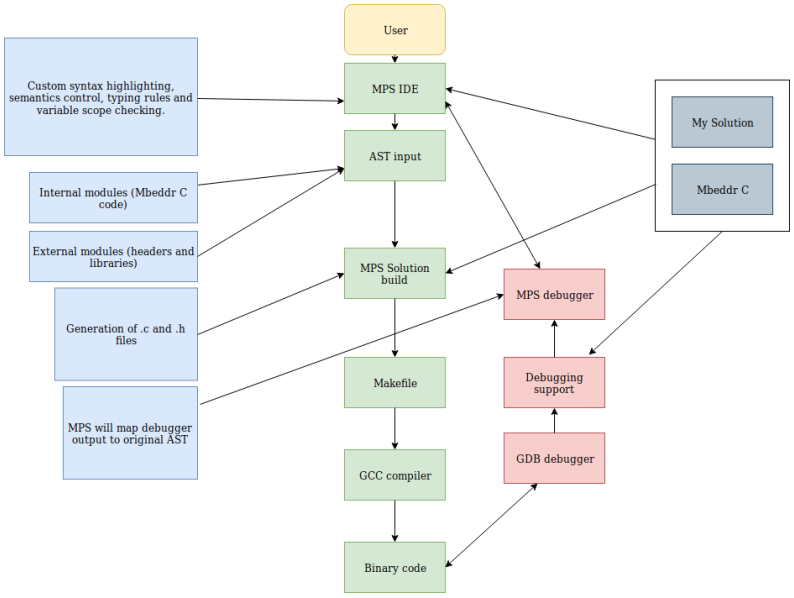

Figure 3.4: Workflow of MPS Language workbench

3.3 Differences between Mbeddr and C11 standard

Mbeddr extension has several important differences from the C11 standard. Since my solution heavily relies on Mbeddr C syntax, these changes have affected my solution as well. This means that some constructs, while valid in C11, will not be compatible with the match statement. This is however not entirely a disadvantage, as Mbeddr offers several improvement over the C11 standard. In the subsections below, there will be a brief overview of those differences and improvements.

3.3.1 Stronger primitive typing rules

Mbeddr has its own Boolean type - what this means is that any expression used inside an `if` statement must resolve to a Boolean type. Also, any integer inside Mbeddr C has to have byte size specification. This means that `int` type does not exist. Instead, one has to use `int32` or other sized types. Here is an example of how these typing rules affect the match statement:

```
boolean isZero(int number) {
  match ( number ) {
    case false : {
      // This will cause an error
      return true;
    }
    case 0 : {
      // This will not
      return true;
    }
  }
  return false;
}
```

Figure 3.5: Example of bad typing in Mbeddr

3.3.2 String type

In Mbeddr, one can use the `string` type instead of `char *` pointer type. This means that my solution considers any string to be a literal. This does not, however, mean that one can use, as it is common in higher-level languages, the "+" sign for concatenation.

3.3.3 Automatic typedef for structs

When one creates a struct in Mbeddr, they do not need to add a typedef in order to shorten the type definition syntax. Mbeddr does this by itself - this affects how one needs to define a struct type in the match statement as well.

3.3.4 Internal and External modules

In Mbeddr, there are modules, as opposed to standard C .c files. These modules can either be internal (code supplied by a programmer) or external (header-like modules in order to import .h and .c files from standard C). Any module can be imported into an internal module - this will allow one to run methods and use structs from the other module inside.

Mbeddr C contains a keyword for the functions, enumeration, structs and unions which can be used in another module on import. The keyword is `exported`. This keyword does not appear in external modules, as Mbeddr assumes that everything defined inside the external module will be used in other modules.

3.4 Unreachable code detection

Inside any list of statements inside C code, there are situations where one can write code in such a way, where there is no chance of reaching certain statements. These situations can be bothersome, since they increase code complexity and a chance of error. Here are some examples of how these situations can arise:

- `if` statement with `false` boolean inside the condition
- `else` statement, where the connected if contains `true` boolean
- `while` statement with `false` boolean
- `while` statement with `true` boolean, where there are statements after the `while`
- `for` statement, similar to `while`
- `do { } while()` statements and similar
- Two or more `return` statements in a row

Therefore, MPS offers a way of detecting these situations by letting a language designer create a dataflow "language" (mentioned in 3.1.2.2). It allows one to check whether a particular statement can be run.

Mbeddr uses dataflows in order to determine whether the Mbeddr C code created by the programmer is valid from this point of view as well. My solution then cooperates with Mbeddr and MPS in order to integrate itself with the dataflow of the surrounding code.

What this means is that match statement is from the dataflow MPS point of view merely a series of `if`s in a row. There is however, one exception to this principle - wildcard case pattern. If one were to define a wildcard case pattern, it would get treated as `if (true) { }`. For the limitations in my implementation of unreachable code detection for the match statement, please see the section 2.1.7.

3.5 Testing

In order to reduce the probability of bugs occurring in the match statement and its cases, I have added a series of tests which one can run from the MPS language workbench.

3.5.1 Translation tests

Translation tests contain a set of functions inside of which is the match statement. The tests are passed, if the functions and match statements are translated without an error by MPS with the Mbeddr devkit(translation to C finishes without any errors).

3.5.2 Performance tests

In order to measure performance of an algorithm written with the use of a match statement and an algorithm which was written using ordinary C, four performance tests were created. These tests are designed to be used to perform a performance analysis.

The first pair of tests is focused on searching through a 10000000 items long linked list in order to find a certain pattern and return how many times this pattern has been found. The second test focuses on how long it would take to parse through an ADT. This ADT has a form of binary tree, 22 layers deep. Each of those pairs contain one test with the match statement and one test without.

3.5.3 Unit tests

There are currently 4 unit tests appended to my solution. These tests are in place to reduce the probability of runtime errors and unexpected behaviour. These tests check for algorithm results and whether or not are all the necessary null checks in place.

Chapter 4

Evaluation

My solution will be evaluated on qualitative basis by comparing the readability of code with and without the match statement on several chosen examples. The comparison is there to show how much easier can one reason about a particular problem, when one can add a higher level of abstraction. Performance evaluation was not done. The reason for this are time constraints.

4.1 Balanced Binary Tree Insertion

This example (see figure 4.2) is about creating a balanced binary tree, where each left node is empty or less than the parent nodes value and every right node is either empty or has a value greater than it's parent.

```
enum listType {
  nil;
  tree;
}

// This value will always be 0 - it is a placeholder
struct nil {
  int32 value;
};

struct node {
  tree* left;
  tree* right;
  int32 value;
  char* print;
};

union typeofTree {
  node tree;
  nil nil;
};

struct tree {
  listType type;
  typeofTree treeTypes;
};
```

Figure 4.1: ADT for balanced binary tree used in examples of pattern matching

```
void insert(int32 value, root* tree) {
  match ( *tree ) {
    case root { nil } : {
      root newLeaf = createTree(value);
      *tree = newLeaf;
    }
    case root { tree, treeTypes { tree { left, right, val } } }
    if value < val : {
      insert(value, left);
    }
    case root { tree, treeTypes { tree { left, right, val } } }
    if value < val : {
      insert(value, right);
    }
  }
}


void insertOrd(int32 value, root* tree) {
  root treeCont = *tree;
  if (treeCont.type == nil) {
    root newLeaf = createTree(value);
    *tree = newLeaf;
  } else if (treeCont.type == tree) {
    if (value < treeCont.treeTypes.tree.value) {
      insertOrd(value, treeCont.treeTypes.tree.left);
    } else if (value > treeCont.treeTypes.tree.value) {
      insertOrd(value, treeCont.treeTypes.tree.right);
    }
  }
}
```

Figure 4.2: Comparison between balanced binary tree insertion in Mbeddr C with and without the match statement. For a look on what the structures used here are, see figure 4.1

4.2 Binary Tree Comparison

In this example, I wish to showcase the ability of pattern matching with the match statement to simplify comparing values in a binary tree. In figure 4.3 is an example of a binary tree comparison.

```
boolean compareTrees(tree* expected, tree* result) {
  toCompare comp = {
    expectedTree = expected,
    resultTree = result
  };
  match ( comp ) {
    case toCompare { tree * { nil }, tree * { nil } } : {
      return true;
    }
    case toCompare { tree * { tree, typeofTree { node { left1, right1,
        value1 } } }, tree * { tree, typeofTree { node { left2, right2,
        value2 } } } } : {
      return value1 == value2 && compareTrees(left1, left2) && compareTrees(
          right1, right2);
    }
  }

  return false;
}



boolean compareTreesOrd(tree* expected, tree* result) {
  if (expected->type == nil && result->type == nil) {
    return true;
  }
  if (expected->treeTypes.tree.value == result->treeTypes.tree.value) {
    return compareTreesOrd(expected->treeTypes.tree.left, result->treeTypes.
        tree.left) && compareTreesOrd(expected->treeTypes.tree.right, result
        ->treeTypes.tree.right);
  }
  return false;
}
```

Figure 4.3: Binary tree comparison in Mbeddr with and without match - the structures used are listed in figure 4.1

4.3 Finding maximum in a list

The example (figure 4.4) in this section is a simple function which finds out the largest integer in a linked list.

```
int32 findMaxOrd(list* numbers, int32 carry) {
  if (numbers == NULL) {
    return carry;
  }
  if (carry > numbers->value) {
    return findMaxOrd(numbers->list, carry);
  }
  return findMaxOrd(numbers->list, numbers->value);
}


int32 findMax(list* numbers, int32 carry) {
  match (numbers) {
    case list *{num, next} if carry < num : {
      return findMax(next, num);
    }
    case list *{_, next} : {
      return findMax(next, carry);
    }
  }
  return carry;
}
```

Figure 4.4: List search comparison between a solution with the match statement and without - linked list definition is in figure 2.4

4.4 Pattern matching used in UNIX networking structures

The purpose of this set of examples is to showcase how a pattern matching structure in C might simplify how one might be able to use the C structures used for networking in UNIX [19].

In the example provided in figure 4.6, we get a list of servents (server descriptors), from which we have to choose the ones that both operate with the POP protocol and have an UDP type of connection and then return the port number of the first one found.

```
struct serventList {
  servent servent;
  serventList* next;
};

struct servent {
  char* s_name; //Server name
  char* s_aliases; //Server aliases
  int32 s_port; //server port
  char* s_proto; //Type of connection
};
```

Figure 4.5: Servent data structure

```
int32 findPOPWithUDPPort(serventList* xs) {
  match ( xs ) {
    case serventList * { servent { "POP", _, port, "UDP" } } : {
      return port;
    }
    case serventList * { _, next } : {
      return findPOPWithUDPPort(next);
    }
  }
  return -1;
}


int32 findPOPWithUDPPordOrd(serventList* xs) {
  while (xs != NULL) {
    if (xs->servent.s_name != NULL
     && strcmp(xs->servent.s_name, "POP") == false
     && xs->servent.s_proto != NULL
     && strcmp(xs->servent.s_proto, "UDP") == false) {
      return xs->servent.s_port;
    }
    xs = xs->next;
  }
  return -1;
}
```

Figure 4.6: Servent searching algorithms - for the servent struct, see figure 4.5

In the second example, shown in figure 4.8, the match statement is being used in order to deconstruct socket information and select all 32-bit IP addresses, where the socket is an internet socket and the port number is between 0 and 1023 (system ports).

```
struct socketList {
  sockaddr_in socket;
  socketList* next;
};

struct in_addr {
  uint64 s_addr;
};

struct sockaddr_in {
  int8 sin_family; //Type of connection
  uint8 sin_port; //Port number
  in_addr sin_addr; //IP address struct
  char sin_zero; //Not used
};

struct ipList {
  uint64 addr; //IP address
  ipList* next;
};
```

Figure 4.7: Socket data structure

```
void getSysInetSockets(socketList* xs, ipList** head, ipList** tail) {
  match ( xs ) {
    case socketList * { sockaddr_in { fam,  port,  in_addr { ip } },  next }
     if fam == AF_INET && port < 1024 : {
      cons(head, tail, ip);
      getSysInetSockets(next, head, tail);
    }
    case socketList * { _,  next } : {
      getSysInetSockets(next, head, tail);
    }
  }
}


void getSysInetSocketsOrd(socketList* xs, ipList** head, ipList** tail) {
  if (xs == NULL) {
    return;
  }
  if (xs->socket.sin_family == AF_INET
   && xs->socket.sin_port < 1024) {
    cons(head, tail, xs->socket.sin_port);
  }
  getSysInetSocketsOrd(xs->next, head, tail);
}
```

Figure 4.8: Socket searching algorithms. For the structures in this example, see figure 4.7

4.5 Coin sorting machine

The last example (see figure: 4.10) will be a coin sorting machine (for an example of implementation in Rust and Scala see: chap:matching). The machine will simply take in coins and return their numeric value. In case of the coin being a quarter, the machine will print out the name of the state which has minted that quarter.

```
enum CoinType {
  Penny;
  Nickel;
  Dime;
  Quarter;
}

struct Quarter {
  string state;
};

struct Coin {
  CoinType type;
  Quarter* quarter;
};
```

Figure 4.9: Structures used for coin sorting

```
int32 coinSorter(Coin coin) {
  match ( coin ) {
    case Coin { Penny } : {
      return 1;
    }
    case Coin { Nickel } : {
      return 5;
    }
    case Coin { Dime } : {
      return 10;
    }
    case Coin { Quarter,  Quarter *
        { state } } : {
      printf("Quarter was made in %s
          \n", state);
      return 25;
    }
    case _ : {
      return -1;
    }
  }
}
```

```
int32 coinSorterOrd(Coin coin) {
  if (coin.type == Penny) {
    return 1;
  }
  if (coin.type == Nickel) {
    return 5;
  }
  if (coin.type == Dime) {
    return 10;
  }
  if (coin.type == Quarter) {
    printf("Quarter was made in %s \
        n", coin.quarter->state);
    return 25;
  }
  return -1;
}
```

Figure 4.10: A coin sorting machine, written with the match statement - to see the structures used, see figure: 4.9

Conclusion

In this thesis I extended the C programming language with a pattern matching concept known from functional programming. It allows for a cleaner, more manageable code, fewer errors and a higher degree of abstraction.

The extension was implemented using MPS and mbeddr which together provide a framework for adding extension to the C programming language. The advantage of using MPS is that MPS gives you custom highlighting, custom semantic navigation and debugging support.

Overall, this thesis has shown a useful direction for further research and development. Extensions of mainstream languages to include concepts from functional or logical programming are often well rewarded in increases of effectivity of the programmer and cleaner code.

Bibliography

[1] Matsakis, N. D.; Klock II, F. S. The rust language. In *ACM SIGAda Ada Letters*, volume 34, ACM, 2014, pp. 103–104.

[2] Klabnik, S.; Nichols, C. *The Rust Programming Language*. No Starch Press, 2018.

[3] Odersky, M.; Altherr, P.; et al. The Scala language specification. 2004.

[4] Rust online documentation. `https://doc.rust-lang.org/book/ch18-03-pattern-syntax.html`, accessed: 2019-05-14.

[5] Subramaniam, V. *Functional programming in Java: harnessing the power of Java 8 Lambda expressions*. Pragmatic Bookshelf, 2014.

[6] Järvi, J.; Freeman, J. *C++ lambda expressions and closures*, volume 75. Elsevier, 2010, 762–772 pp.

[7] Felleisen, M.; Findler, R. B.; et al. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[8] Yallop, J. Practical generic programming in OCaml. In *Proceedings of the 2007 workshop on Workshop on ML*, ACM, 2007, pp. 83–94.

[9] Voelter, M.; Pech, V. *Language modularity with the MPS language workbench*. 2012, 1449–1450 pp.

[10] Hartel, P. H.; Muller, H. L. Simple algebraic data types for C. *Software: practice and experience*, volume 42, no. 2, 2012: pp. 191–210.

[11] Voelter, M.; Ratiu, D.; et al. *mbeddr: an extensible C-based programming language and IDE for embedded systems*. 2012, 121–140 pp.

[12] Stallman, R. M.; Weinberg, Z. The C preprocessor. *Free Software Foundation*, 1987.

[13] Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994.

[14] Kats, L. C.; Visser, E. The spoofax language workbench: rules for declarative specification of languages and IDEs. *ACM sigplan notices*, volume 45, no. 10, 2010: pp. 444–463.

[15] Kelly, S.; Lyytinen, K.; et al. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *International Conference on Advanced Information Systems Engineering*, Springer, 1996, pp. 1–21.

[16] Van Wyk, E.; Bodin, D.; et al. Silver: An extensible attribute grammar system. *Science of Computer Programming*, volume 75, no. 1-2, 2010: pp. 39–54.

[17] Kaminski, T.; Kramer, L.; et al. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, volume 1, no. OOPSLA, 2017: p. 98.

[18] Eysholdt, M.; Behrens, H. *Xtext: implement your language faster than the quick and dirty way*. 2010, 307–309 pp.

[19] Coffield, D.; Shepherd, D. Tutorial guide to Unix sockets for network communications. *Computer Communications*, volume 10, no. 1, 1987: pp. 21–29.

Appendix A

Acronyms

ADT Algebraic Data Type.

ADTs Algebraic Data Type.

AST Abstract Syntax Tree.

devkit Development language package.

DSL Domain Specific Language.

EBNF Extended Backus-Naur Form.

FP Functional Programming.

IDE Integrated Development Environment.

Appendix B

Contents of enclosed USB drive

- `readme.txt` ..... USB drive file contents description
- `MPS-mbeddr-2018.3.5` ..... Directory with MPS editor with Mbeddr
  - `bin` ..... MPS executables
    - `mps.sh` ..... Script to run MPS
- `Scenario` ..... Directory containing my solution
  - `ScenarioLang` .. Directory containing language concepts for the match statement
  - `Documentation` ..... Directory with LaTeX thesis source codes
- `text` ..... Directory with thesis text
  - `thesis.pdf` ..... the thesis text in PDF format