Master Thesis

DMCD

A Distributed Multithreading Caching D Compiler

Author:
Robert Schadek

Supervisor:
Prof. Dr.-Ing. Oliver Theel
Dipl.-Inform. Eike Möhlmann

12. September 2012

Abstract

Nearly thirty years have passed since the release of the infamous Dragon Book by Aho, Lam, Sethi, and Ullman. Since then the available memory and CPU power has grown by a factor of at least 1000 and 10000 respectively. The use of Multiprocessors and fast networks has become the rule rather than the exception. Even though compiler and programming language have advanced greatly in this period of time, their basic design has not been adjusted to reflect the enormous hardware improvements. In this Master thesis, I will present a D compiler that is exploits these tremendous hardware advances.

The compiler is split into two parts, a front-end and a daemon. The daemon is started only once and runs until it is terminated manually. The front-end accepts jobs and delivers them to the daemon. These jobs are compiled by the daemon. Parallelisms is realized by organizing lexing, parsing, and semantic analysis in three communicating threads. In order to speed up compilation even more results are cached within the daemon. For further speedup, jobs and results are automatically distributed by the daemons within a network. Results are stored in a cache and network friendly matter.

Besides the conceptual architecture of this new compiler, I will report on the library, that was written to build this compiler.

Contents

Part I.

Introduction

1. Introduction

Compiler have not change much since their invention in 1950. They read input, write executables and exit. CPUs and computer in general have not really changed either. A program is still loaded into memory and executed instruction by instruction. Computers might not have changed that much, but they got much faster, they have incredibly amounts of RAM nowadays and the speed of networks has increased tremendous. Not only did the CPUs got faster, they also have increased in numbers. Multicore CPUs are widely adopted. Current compilers have not adapted to this changes.

This thesis will present a way of adapting a compiler to the performance changes and therefore speed up the compile process. This includes the caching of data, running things in parallel and distributing the work among multiple computers in a network.

Choosing the right programming language is an important factor for this thesis. *D* was selected as it is a modern, feature rich language with a familiar C style syntax. As *D* is a relative new languages not all tools or libraries are present that are required to efficiently create a compiler. To fill this gap a lexer as well as a parser generator have been created. As these tools require container data structures and additional functions that are currently not offered by *Ds* standard library phobos, a library has been created that implements the needed functionality. The library in combination with the two tools have been used to create a prototype compiler that compiles *D*. This compiler will be used to test the effectiveness of architectural changes.

This thesis begins by introducing *D* and presenting the basics of lexing and parsing. The second part describes the implementations of the library, the lexer generator and the parser generator. This is followed by the implementation and evaluation of the compiler. A concluding chapter gives ideas for future projects.

1.1. Introducing *D*

D is a statically typed, compiled, multi-paradigm programming language. The primary author, Walter Bright, started developing *D* as a re-engineered *C++*. In January 2007 the first version of *D* was released after eight years of developing. Version one of *D*, later commonly referred to as *D1*, marked a stabilization of the imperative and object oriented features. Hardly half a year later the first version of *D2* was released. *D2* broke compatibility with *D1* by introducing features like const-correctness, purity and closures. Today, *D* normally refers to *D2*.[1]

D is not only a successor of C++ it also borrows features from other object oriented, functional and generative programming language. Some of these features are described here, to give a short overview of the capabilities of *D* and why it yields an interesting choice for this thesis.

One feature is called compile time function execution (ctfe), this allows functions to be executed at compile time. By itself, this idea is nothing new. *C++* allows this by template meta programming. *D* introduces a new way of doing this. Every normal function, as long as a certain criteria are met, can be executed at compile time. One of those criteria is for example, that the function must be free of side-effects. That means no global or function static variables are to be read and no *IO* must be performed. Another criteria is that no inline assembler can be used. To execute a function at compile time, all values of its parameter must be known and the variable taking the return value must be mark as static or be of type *enum*.

As *D* does not offer defines, another way was devised to allow source code manipulation at compile time. The methods used are string mixin and conditional compilation. Mixins take strings and pass them to the compiler at compile time. This means is that in combination with ctfe, the source code can be generated at compile time to met requirements. Conditional compiling is used to manipulate the compile process through passing values to the compiler. This means that depending on these values different code block are compiled. Mixins in combination with ctfe and conditional compiling allows the programmer to create arbitrary source code and compile it at compile time. Because defines are not present the lexer only has to process the input file only, in contrast to, for example, C where the lexer has to resolve defines and rescan the file afterwards.

Ds object oriented approach follows that of Java, its templating mechanism follow that of C++. The syntax of *D*s template instantiating differs from that of C++ and Java in that it does not use the *greater* and *lesser* tokens for grouping the template parameter. Instead it uses another pair of parentheses to group the template parameter. Such a parentheses block is prefixed by an exclamation mark. This was done to reduce some ambiguity in C++ and Java grammar. Listing 1.1 shows such a ambiguity in C++ and Java [dla12a].

[1]From this point forward we refer to *D2* simply as *D*.

Listing 1.1: C++ parsing error through greater and lesser token.

```
template<int i> class Foo
{
    ...
};
Foo<3 > 4> f; // error: expected unqualified-id before numeric
    constant
```

The `3 > 4` part of listing 1.1 should evaluate to FALSE, which is defined as numerical 0. Therefor the template parameter `int i` should be set to 0. Instead the compiler yields an error. [2] This error arises because the compiler does not know that it has to treat the first *greater* token as comparison operator rather than a template parameter list margin.

Listing 1.2: D equivalent of listing 1.1.

```
class Foo(int i) {
    ...
}
Foo f = new Foo!(3 > 4)();
```

Listing 1.2 shows the same templated class of listing 1.1 in *D* . Notable differences are:

- the template parameter are placed after the class name
- the template parameter instantiation is grouped by `!(` and `)`
- the class is created on the heap [3]

The ambiguity problem in the C++ code is avoided because the greater token is not used to group the template parameter. Another reason for modifying the syntax is that multiple nested template instantiations could lead to unfortunate parse errors. The problematic C++ and Java syntax is shown in listing 1.3.

Listing 1.3: C++ Token semantic error

```
template<class T> class Foo {
    ...
};

template<class T> class Bar {
    ...
};

Foo<Bar<int>> f;
```

The lexer has to recognize that the right shift token shortly before the identifier *f* are actually two greater token. This means that token recognition is no longer a problem of Chomsky Type-3, instead it is now a problem of Chomsky Type-2. From a practical

[2]The version 4.7 of gcc returns with an error. The error reads: "error: expected unqualified-id before numeric constant".

[3]In contrast to C++ all classes in *D* are created on the heap.

perspective this leads to an even bigger problem. The problem is that we can no longer use a DFA to generate the tokens from the source code, because DFA can only parse languages of Chomsky Type-3. [4]

Another notable quality of *D* is its approach to multithreaded programming. The two main concepts to this are:

- immutability
- thread local storage

Thread local storage (tls) allows thread to see their own copy of global and static variables. That means, by default, nothing is shared by threads. This removes the possibility of accidental race conditions. This also means the programmer is forced to use other mechanics for thread communications. One way would be to use two functions defined in the *D* standard library. These functions are called *send* and *receive* and they implement message passing capabilities. Another more crude way of doing things, would be to use the keyword __GSHARED that disables thread local storage for the marked variable. Semaphores needed to be marked as __GSHARED.

Immutability makes a variable assignable only once. A variable is declared immutable by the keyword IMMUTABLE. The benefit of this is that any thread that reads immutable data can be sure that no one can change it. With that knowledge in mind no synchronization primitives are required to provide a race condition free view of the data. The IMMUTABLE keyword can also be used to modify member function behaviour of structs and classes. The effect of this is similar to that of const member functions in C++. A member function that has been marked immutable only sees an immutable this reference. This way the statements of the function and all subsequent functions calls cannot modify members that belong to the *this* reference. Using immutable to modify member functions is important in a way that normal member function or even const member functions could not guarantee immutability correctness.

Listing 1.4: Nessarity of immutability correctness

```
class Foo {
    int a;

    this(int a) immutable { this.a = a; }

    void setA(int newA) { this.a = newA; }

    int getA() immutable { return this.a; }
}

void func(immutable Foo foo) {
    immutable int a = foo.getA();
    assert(a == 33);
    foo.setA(44); // compiler reports an error
}
```

[4]From C++11 forward this problem is addressed. The use of the right shift as a token for template grouping is no longer prohibited.

```
16
17 void func2() {
18     immutable Foo foo = new Foo(33);
19     func(foo);
20 }
```

Listing 1.4 shows why immutable member function are necessary. At line 20 an immutable instance of class *Foo* is created with a value of 33. At line 14 in function *func2* the member function *setA* has write access the this reference of the class. It can therefore change the value of *this.a*. The assertion on line 13 holds, but when in line 14 the *setA* function is called an mutable this reference would be needed. As this is not the case the compiler reports an error.

Immutability is used for another purpose. It is used to mark the elements of an array immutable. By doing so the array itself stays mutable, but its elements are immutable. The most prominent usage of this is the definition of string types. They are defined as IMMUTABLE(CHAR)[]. This allows assign a new string to a variable, but assign certain characters in the string is forbidden.

1.2. *D* Crash Course

As much of this thesis evolves around the programming language *D* , understanding some *D* characteristics is important. The following sections will give an introduction into the language.

1.2.1. Parts of C

As *D* is in some parts based on C *D* behaves in many ways as C. The differences are discussed here. The first major difference is that *D* does not allow textual includes, instead declarations are imported symbolically. Functions can be overloaded. The overloading is based on the different function parameters. *D* removes the need for forward references. Defines are removed as well. The procedural parts of C are directly copied to *D*. A, now deprecated, feature of *D* that is present in C are typedefs. The ALIAS expression is used to replace typedef expression.

1.2.2. Classes

Classes in *D* follow the classes of *Java* in most ways. *D* classes follow a single inheritance model similar to *Java*[Sun12a]. Methods can be synchronized and a reference to the instance of the class is provided through the *this* keyword[Sun12b]. The constructor methods are named *this* and have no return values. The listing 1.5 shows some of the basic constructors.

Listing 1.5: An exemplary *D* class

```
 1 class Foo {
 2     public int bar;
 3     protected float tzz;
 4 
 5     this(int bar) {
 6         this.bar = bar;
 7     }
 8 
 9     this(int bar, float tzz) {
10         this(bar);
11         this.tzz = tzz;
12     }
13 
14     invariant() {
15         assert(bar < 100 && bar >= 0);
16     }
17 
18     synchronized void increment() {
19         this.bar++;
20     }
21 
22     synchronized int getBar() const {
23         return bar;
24     }
25 
26     private:
27         string name;
28 
29         string getString() {
30             return this.name;
31         }
32 }
```

Lines five to seven show a basic constructor that assigns the value of *bar* to the member *bar*. The second constructor on lines nine to twelve delegates the bar value to the first constructor. *D* allows constructors to call other constructors or member function, much like *Java*. A feature nearly unique to *D* is show on lines 14 to 16. The *invariant* statements runs before and after every member function call. This allows to check whether or not the given invariants hold true. This way it can be enforced that a given class always honours certain specifications. Two methods are marked as *synchronized*, this serializes possible parallel calls to the class. One method is marked as *const*, methods marked this way can not modify the members of the class. As the member declaration at the beginning and end of the class show, protection attribute can be set in the *Java* as well as the C++ way.

1.2.2.1. Inheritance

As mentioned earlier, classes in *D* follow a single inheritance model similar to *Java*. The syntax follows that of C++. Unless marked final every member function is to be considered *virtual*. To actually override a member function the `override` keyword needs to be used. The example in listing 1.6 shows a simple inheritance use case.

Listing 1.6: Simple inheritance

```
class Triangle {
    float x1, x2, x3;
    float y1, y2, y3;

    this(float x1, float x2, float x3,
                float y1, float y2, float y3) {
        this.x1 = x1; this.x2 = x2; this.x3 = x3;
        this.y1 = y1; this.y2 = y2; this.y3 = y3;
    }

    void draw() {
        ...
    }
}

class Box : Triangle {
    float x4;
    float y4;

    this(float x1, float x2, float x3, x4,
                float y1, float y2, float y3, float y4) {
        super(x1, x2, x3, y1, y2, y3);
        this.x4 = x4;
        this.y4 = y4;
    }

    override void draw() {
        super.draw();
        ...
    }
}
```

Important to note in this example is that the member function of the parent class can still be accessed through the `super` keyword.

1.2.2.2. Interfaces

As the inheritance model does not allow a class to inherit from multiple classes, interfaces where introduced as in *Java*. Interfaces only define methods but do not supply implementations. What they however can do is to provide contracts. An example interface is given in listing 1.7.

Listing 1.7: Interface example

```d
interface FancyMathFunctions {
    int calculatePi();
    float calculateE();
    float calculateEpsilon(float input)
        in { assert(input > 0.0); }
        out(result) { result > 0.0; }
}
```

Only classes can implement interfaces in *D* , structs are considered to be pure value types. To make a class implement an interface the interface identifier needs to be placed behind the class identifier in the same way an inherited class identifier is placed. Should a class or interface already be inherited or implemented the interface is appended to the list comma separated. Listing 1.8 shows the process of implementing an interface.

Listing 1.8: Interface implementation

```d
class SimpleMathClass {
    int addInteger(int a, int b) { return a + b; }
}

class AdvancedMathClass : SimpleMathClass, FancyMathFunctions {
    int calculatePI() { return 3; }
    float calculateE() { return 2.7; }
    float calculateEpsilon(float input) {
        return doCalculation(input);
    }
}
```

The *AdvancedMathClass* does not only inherited the member function of the *SimpleMathClass* it only implements the interface *FancyMathFunctions* defined in listing 1.8. As discussed, the inherited class and the interface are listed comma separated [dla12c].

1.2.3. Type Casts

Type casts are a way of forcefully changing the type of values. Explicit type casts are usually considered bad practice, even though they are needed in most modern programming languages. *D* features a complex implicit type propagation scheme, particularly for integer types.

Listing 1.9: Cast operator

```d
real veryBigValue = -999999999.234234234234;
ubyte verySmallStore = cast(ubyte)veryBigValue;
```

The above listing 1.9 shows how the cast expression is used.

1.2.4. Templates

Template programming plays a big role in *D*. Templates can be used to create generic functions as well as generic aggregations. This allows the behavior of code to be changed by their instantiation, without any code duplication. Templates are defined by placing the function or aggregation in a template block. Such a block is shown in listing 1.10.

Listing 1.10: Template block

```
template Foo(T) {
    ...
}
```

The *T* enclosed by the parentheses defines a type that is not yet defined. Another way to define templates is the shorthand notation that will be introduced later. We will now first present template functions, followed by template aggregations.

1.2.4.1. Template Functions

A template function is a function where one or more variable types are not defined at the declaration stage. The template example of listing 1.10 could hold a function that would return the default initialize value of the given type. Listing 1.11 shows such a function and its instantiation.

Listing 1.11: Get default init value with templates

```
 1 template Foo(T) {
 2     T getInitValue() {
 3         return T.init;
 4     }
 5 }
 6
 7 int init = Foo!(int).getInitValue();
 8 assert(init == 0);
 9
10 float init2 = Foo!(float).getInitValue();
11 assert(init2 == NaN);
```

The property *init* returns the value that is assigned as default value to a given type. This value is, for example, zero for all integer value types. The template function *getInitValue* returns the given init value for the type of *T*. On line seven the function is instantiated with type *int*. If we were to do the instantiation manually, we would replace every occurrence of *T* with *int*. This is exactly what the compiler does. The second example makes the same replacements only with the *float* type. The init value for all float types is the *NaN* value. *NaN* abbreviates for not a number. It may not always be desired to write so much code for a single template function. A shorthand notation for the given getInitValue function is given in listing 1.12

Listing 1.12: Get default init value with templates, shorthand

```d
T getInitValue(T)() {
    return T.init;
}

int init = getInitValue!(int)();
assert(init == 0);

float init2 = getInitValue!(float)();
assert(init2 == NaN);
```

In this example the template block is omitted and the variable type declaration is moved between the function name and the parameter declaration. A function that has two parameter blocks is always a template function. The first grouping consists of values that are known at compile time. To distinguish the two blocks at instantiation the template parameter grouping is proceeded by an exclamation mark. The process of instantiation is the same as for the more verbose version. If there is only a single template parameter the grouping parenthesis can even be omitted. Template parameter can also be used to represent function parameter. The listing 1.13 shows a minimum function that works for all types that work with the $<$ operator.

Listing 1.13: Generic minimum function

```d
T minimum(T)(T l, T r) {
    return l < r ? l : r;
}
```

Again, doing the instantiation with the *int* type, we will see a function that is equal to what is shown in listing 1.14.

Listing 1.14: Generic minimum function instantiated with the int type

```d
int minimum(int l, int r) {
    return l < r ? l : r;
}
```

Template functions are not limited to simply placing types, values can be placed also. This allows to create function that do highly specific tasks, but stay generic till instantiated. For instance, the template function shown in listing 1.15 adds a value to a given integer and returns the sum. The clue is that the compiler creates a separate function for every different template parameter and can therefore apply specific optimization.

Listing 1.15: Add value to integer

```d
int addN(int staticValue)(int nonStaticValue) {
    return staticValue + nonStaticValue;
}

assert(10 == addN!(5)(5));
assert(1337 == addN!(1330)(7));
```

[dla12f]

1.2.4.2. Template Parameter Type Deduction

In special cases *D* even allows to omit the template type parameter. This is called template parameter type deduction. Considering we use the template function minimum defined in listing 1.13, we could omit the first parameter grouping, because the compiler can deduce the type of *T*. As the rules for template parameter type deduction are rather complicated a more complex explanation is left to the official *D* reference [dla12f].

1.2.4.3. Template Aggregations

Template aggregations enclose everything that creates types depended on one or more template parameter. This means templated interfaces, structs and classes. The idea behind templated aggregations is the same as behind templated functions, the type or value of a specific variable is defined when the template is instantiated.

1.2.4.3.1. Template Structs

Listing 1.16: Templated struct

```
template Bar(T) {
    struct PairOfT {
        T first;
        T second;
    }
}

Bar!(int).PairOfT instance1;
```

The above listing 1.16 shows a templated struct where the type of the two members is fist defined at the stage of instantiation. A shorthand notation exists as well, it is shown in listing 1.17.

Listing 1.17: Templated struct shorthand

```
struct PairOfT(T) {
    T first;
    T second;
}

PairOfT!(int) instance1;
```

Aggregations do not have a parameter list the first grouping. The exclamation mark is still required to make the syntax similar.

1.2.4.3.2. Template Interfaces

Interfaces can be templated. The template parameter allows the same interface to represent different functional characterics depending on the type.

Listing 1.18: Template interface example

```
interface FancyMathFunctions(T) {
    T calculatePi();
    T calculateE();
    T calculateEpsilon(T input)
        in { assert(input > 0.0); }
        out(result) { result > 0.0; }
}
```

The interface from listing 1.7 is changed into a template interface in listing 1.18.

1.2.4.3.3. Template Classes

As the existence of template interface suggests template classes exists as well. Through the use of template type parameter the member function can be defined generically. This way they can apply for a bigger number of problems. All container implementation by libhurt, the library created for this thesis, make excessive use of template classes.[5] Similar to template structs, the template type parameters are passed as a grouping after the class identifier. This is the shorthand notation, the more verbose version is omitted.

Listing 1.19: Simple fixed size buffer

```
class Buffer(T) {
    T[128] buffer;
    size_t idx;

    this() { this.idx = 0; }

    void append(T value) {
        this.buffer[idx++] = value;
    }

    T[] getContents() {
        return this.buffer[0 .. idx];
    }
}

Buffer!(int) intBuf = new Buffer!(int)();
Buffer!(float) floatBuf = new Buffer!(float)();
Buffer!(string) stringBuf = new Buffer!(string)();
```

[5]Compare to chapter 5 on page 65 and following.

The example of a template class in listing 1.19 shows how simple it is to create a buffer for different size, without duplication parts of the implementation. The lines 16 to 18 show the usage of the *Buffer* class. Creating different kinds of buffers simply requires as changing a type.

Template classes can implement template interfaces. The listing 1.18 defines a template interface. To implement that interface we not only have to implement the methods defined in it, we also have to propagate template parameter to it. Listing 1.20 shows the process. The interface *FancyMathFunctions* was defined in listing 1.18 on page 15. The type *T* will be passed to the interface on instantiation.

Listing 1.20: Template interface implementation

```d
class AdvancedMathClass(T) : FancyMathFunctions!(T) {
    T calculatePI() { return cast(T)3.14; }
    T calculateE() { return cast(T)2.7; }
    T calculateEpsilon(T input) {
        return doCalculation!(T)(input);
    }
}

AdvancedMathClass!(int) amc = new AdvancedMathClass!(int)();
int pi = amc.calculatePI();
```

It is also possible to pass values at compile time, this allows to create even more specific classes at compile time. The next listing 1.21 shows how this can be used to create a more sophisticated buffer.

Listing 1.21: Variable length buffer

```d
 1 class Buffer(T,size_t bufsize) {
 2     T[bufsize] buffer;
 3     size_t idx;
 4 
 5     this() { this.idx = 0; }
 6 
 7     void append(T value) {
 8         this.buffer[idx++] = value;
 9     }
10 
11     T[] getContents() {
12         return this.buffer[0 .. idx];
13     }
14 }
15 
16 auto intBuf = new Buffer!(int,32)();
```

The buffer size is set at compile time. To save some typing work the `auto` keyword is used in line 16 to automatically infer the type of the intBuf variable.

1.2.4.4. Template Restrictions

Sometimes it makes sense to have template functions or aggregations that do not work for all kinds of template parameters. As the template parameters are known at compile time it would make sense to check them at compile time. One way to achieve this is to write the code in a way that it only compiles for the allowed types and values. The problem with that is that it might not always be clear why the given template should not work for the given values. To generate clear error messages and to give the programmer good information about the allowed types and values, *D* allows templates to be restricted. The restriction is done through the well understood if statement. The only new thing about it is the place where the if condition is located. To restrict a template the if condition is placed after the name of the template or respectively after the last closing parenthesis and before the opening curly brace. This can also be used to overload templates as listing 1.22 shows.

Listing 1.22: Template restriction

```
T fac(T,int value)() if(value == 1) {
    return 1;
}

T fac(T,int value)() if(value > 1) {
    return value * fac!(T,value-1)();
}

int fac16 = fac!(int,16)();
assert(fac16 == 2004189184);
```

The listing calculates the faculty of a given value and returns the value in the defined type. The compiler takes the first matching template and instantiates the call to the templates till the recursion stops after instantiating the first template. This shows how an if statement can be used to control template instantiation. The if condition needs to be computable at compile time. Now that we can calculate the faculty of a given number at compile through template restrictions, we might want limit the type of the returned value to *ulong* or *real*. The next listing 1.23 shows how this is done.

Listing 1.23: More complex template restriction

```
T fac(T,int value)()
        if((value == 1) && (is(T == ulong) || is(T == real))) {
    return 1;
}

T fac(T,int value)()
        if((value > 1) && (is(T == ulong) || is(T == real))) {
    return value * fac!(T,value-1)();
}
int r = fac!(int,16); // does not compile
real r = fac!(real,16); // compiles
```

As the comments on the last two lines of the listing indicate, the first instantiation fails and the second works. The first one fails, because the given type of *int* does not match the second part of the restrictions, which states that the type must be either *ulong* or *real*. The conditional statement can be any valid *D* code as long as they are computable at compile time. The *is* keyword yields a boolean with the result of the type comparison.

1.2.5. Compile Time Control Flow and Execution

Restricting templates is not the only way to the control flow of templates. Three additional ways exists in *D* to manipulate the control flow or the compile flow respectively.

1.2.5.1. Static If Statements

Static if statements are pretty close to what the if statement does when restricting templates. Instead of writing them in the template declaration they are preempt with the `static` keyword and placed within any block statement. Again the condition of the if statement needs to be computable at compile time. Static if statements are equal to normal if statements so much that they can even hold alternative branches and conditional alternative branches. Listing 1.24 shows the faculty calculation implemented by using a static if statement.

Listing 1.24: Static if faculty computation

```
T fac2(T,int value)() if(is(T == ulong) || is(T == real)) {
    static if(value == 1) {
        return 1;
    } else static if(value > 1) {
        return value * fac!(T,value-1)();
    } else {
        assert(false);
    }
}

ulong fac16 = fac2!(ulong,16)();
assert(fac16 == 2004189184);
```

What happens here is that the compile looks at the static if statements and generates the code for the branch whose condition evaluates to true.

1.2.5.2. Version Statements

The *version* statement can be seen as a specific `#ifdef` preprocessor macros in C. This is, because the condition can be passed to the compiler as an argument. Similar to static *if* statements, code is emitted only if the condition is matched. Version statements can have an else branch. In contrast to static if statements the conditions are special values that can be considered true or false. Listing 1.25 shows a simple example.

Listing 1.25: Version statement

```
version(fancyversion) {
    do_fance_stuff();
}
```

If the compiler receives a `-version=fancyversion` option the given code block will be emitted.

1.2.5.3. Debug Statements

The debug statement is very similar to the version statement but has a debug specific property. The condition can be integer value. This integer value can be understood as a debug level. The higher the level the more debugging needs to be done. If the compiler finds a debug statement whose debug level is lower or equal the code or the debug statement is emitted, listing 1.26 gives an example.

Listing 1.26: Debug level

```
debug(128) {
    print_not_so_important_stuff();
}
debug(64) {
    print_not_more_important_stuff();
}
debug(1) {
    print_most_important_stuff();
}
```

Depending on what the debug value is the compiler will generate code for one, two or three of the debug statements.

1.2.6. Pure Functions

Pure functions are functions that only depend on their input parameter. They do not read mutable global values and do not perform *IO*. They can only call other pure functions. Pure function can ease debugging expenses as their results are only dependent on the input and are therefore side effect free. Functions that are marked *pure* indicated by the `pure` keyword can also easily be used for ctfe.

Listing 1.27: Pure static if faculty computation

```
pure T fac2(T,int value)() if(is(T == ulong) || is(T == real)) {
    static if(value == 1) {
        return 1;
    } else static if(value > 1) {
        return value * fac!(T,value-1)();
    } else {
        assert(false);
    }
}

ulong fac16 = fac2!(ulong,16)();
assert(fac16 == 2004189184);
```

The above listing 1.27 shows how a function or, in this case template function, is marked as pure.

1.2.7. Function Pointer and Delegates

Function pointers exist already in C, but *D* introduces a more obvious syntax for them, as well as a new kind of function pointer called delegates. The difference between function pointer and delegates is that delegates not only point to an executable function, they also carry a scope pointer. This scope pointer allows the function to access member of the scope pointed to. Listing 1.28 shows a function pointer to a function called bar that returns an integer and takes a float as parameter.

Listing 1.28: Function pointer

```
int bar(float toAdd) {
    return cast(int)(1337 + toAdd);
}

int foo(int function(float) bar, float barv) {
    return cast(int)(1337 + bar(barv));
}

void main() {
    int function(float) bPtr = &bar;

    int value = foo(bPtr, 3);
    assert(value = 2677);
}
```

The function foo takes a function pointer and a float as input. Then it calls the function pointer with the given float, adds another value and returns the result.

As mentioned earlier delegates are function pointer combined with a scope. This scope can either be a struct, a class or a function. It is even possible to return a delegate from a function and still refer to variables that were created on the stack of that function. This particular feature is called closures. The following listing 1.29 shows an exemplary usage of delegates.

Listing 1.29: Delegate example

```
struct Foo {
    int a = 7;
    int bar() { return a; }
}
```

```

int foo(int delegate() dg) { return dg() + 1; }

void test() {
    int x = 27;
    int abc() { return x; }
    Foo f; int i;
    i = foo(&abc); // i is set to 28
    assert(i == 28);
    i = foo(&f.bar); // i is set to 8
    assert(i == 8);
}
```

The listing also shows a nested function. The name of the function is abc. [dla12b]

1.2.8. Advanced Loop Statements

The *foreach* statement allows to simply iterate over an array and self defined structs or classes. A simple foreach statement is presented in listing 1.30.

Listing 1.30: Basic foreach

```
int[] a = [1, 4, 5, 7, 11];
foreach(it; a) {
    printfln("%d", it);
}
```

The foreach statement iterates over the array *a*. The variable *it* holds one value of the array after another. The type of these variable can be omitted, because the compiler can easily infer them.

If a struct or class should be iterable it must implement the opApply member function. This functions has one parameter of type `int delegate`. The parameter of the delegate depends on the number of iterator variables of the foreach statement. Considering we want to have a struct that stores an array and returns the index and the value on every step of the iteration we need to accept a delegate that has two parameters. Such a struct is shown in listing 1.31.

Listing 1.31: Struct opApply

```
struct Foo {
    int[] arr;

    this(int[] arr) {
        this.arr = arr;
    }

    int opApply(int delegate(ref size_t, ref int) dg) {
        int result = 0;
        for(size_t i = 0; i < arr.length; i++) {
            result = dg(i, arr[i]);
            if(result)
                break;
        }
        return result;
    }
}

Foo f = Foo([10, 9, 8, 7]);
foreach(idx, it; f) {
    printfln("%u:%d", idx, it); // prints 0:10 1:9 2:8 3:7
}
```

The block statement of the foreach is converted to a delegate. It is than checked if the Foo struct implements any opApply member function that takes a delegate that matches the created. The parameter of the delegate correspond to the iterator variables of the foreach statement.

1.2.9. Modules and Imports

A module is what is called a package in *Java*. Every source file can have only one module declaration. The main purpose for modules are encapsulating and avoiding name collision. The identifier of the module declaration does not need to equal the name of the file. The import statement is used to import the symbols defined in a module. The import statement expects the module name not the file name. The following listings 1.32 shows the import of two modules and the resolution of a name collision.

Listing 1.32: Module import

```
// file: fileA.d
module A;

int foo() {
    return 1337;
}

// file: fileB.d
module B;

int foo() {
    return 7331;
}


// file: importer.d
import A;
import B;

void main() {
    assert(A.foo() == 1337);
    assert(B.foo() == 7331);
    assert(foo() == 7331);  // compile error
}
```

Part II.

Theory

2. The Lexer

2.1. Introduction

A lexer is typically used to generate tokens from a character stream. Usually, languages accepted by a lexer are of Chomsky Type-3. Lexer implementation can be divided into two main categories, generated and hand-written. Hand-written lexer are conceptionally more powerful than generated, because they are not bound to Type-3 grammars. In contrast generated lexer are easier to create and maintain.

To facilitate, the compiler written for this thesis, a lexer generator was created that uses a table based finate state machine approach. Before the implementation of the lexer generator, who is named dex, the general theory of lexing and lexer generation are discussed.

For a theoretical standpoint DFAs are defined by a tuple of five elements (Q, Σ, δ, q_0, F).

Q is a finite set of states.

Σ is a finite set of input characters. This set is sometimes called the alphabet of the machine.

δ defines a transition function from one state of Q to the next on input of a character of Σ ($\delta : Q \times \Sigma \rightarrow Q$).

q_0 is the start state.

F is a finite set of accepting states ($F \subseteq Q$).

NFAs are equal to DFA with the exception that they allow epsilon transition.

2.2. From Regular Expression to Finite State Machines

Finite state machines are able to accept all regular languages. Regex are another form of defining regular languages. Within the Chomsky hierarchy regular languages are of Type-3. Regular expressions are used from now on to describe the potential of regular languages. This is done because regex are closer to how dex sees the languages.

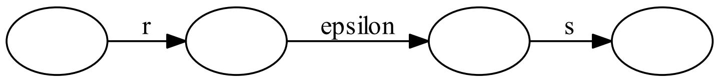


Figure 2.1.: nfa for regex **rs**

2.2.1. How Finite State Machines are Created

Even though the finate state machine could be created in a single regex expression, it is basically split in several small regular expressions. Construction of the state machine works in the following matter.

1. Create a NFA for every regular expression.
2. Create a start state.
3. Join the beginnings of all NFAs to the created start state by an epsilon transition.
4. Convert the obtained non-deterministic finite state machine into a DFA.
5. Minimize the DFA.

After the DFA has been minimized it can be stored as an array. Usually, a reduction phase occurs after the table has been created. This is due to the fact, that in most cases several rows and columns of the table are equal.

2.2.2. Basic Operations of Regular Expression

Regex consists of three basic operations. The basic operations are *concatenation*, *Kleene closure* or *star operation* and *union operation*.

Concatenations are simply one character after another. For instance, the NFA for the regex `rs` would lead to a NFA displayed in figure 2.1. For consistence reason the epsilon is placed between the r and s state, even though it is strictly not necessary. That NFA would only accept the word *rs*, nothing else.

The **Kleene closure**, which is also known as the *star operator*, creates a NFA that accept 0 to n occurrences of the leading character. A regex like **r*** would lead to NFA that is displayed in figure 2.2. A finate state machine modeled to match the NFA of figure 2.2 would for example accept the words *r*, *rrrr*, *rrrrrrrr* or the empty word.

The last operation is the **union operation**. The purpose of the union operation is to introduce a choice on which branch to choose. Figure 2.3 shows the NFA for the regular expression of **(r|s)**. The NFA displayed in figure 2.3 would either accept the words *r* or *s*.

These basic operations can be mixed and concatenated in any possible way to allow the accepting of far more complex regular expression. In order to achieve this regular

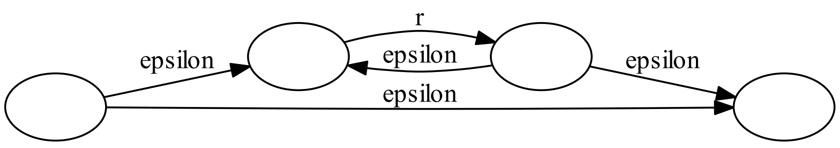


Figure 2.2.: nfa for regex **r***

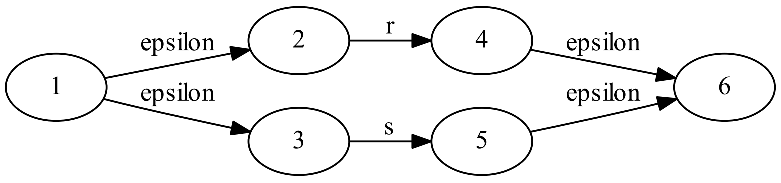


Figure 2.3.: nfa for regex **(r|s)**

expression is processed not unlike an arithmetic expression. Section 2.2.3 explains this procedure in detail.

Some arithmetic operations have precedence before other operations, in regular expression this is also the case. Operator precedence are:

1. Star operator
2. Concatenation operator
3. Union operator

The star operator has the highest precedence, the union operator the lowest. The primary reason for operator precedence is to remove parenthesis from the expression to make them more human readable [ALSU06, p. 121].

2.2.3. Regular Expression to Non Deterministic Finite State Machines

As mentioned earlier in section 2.2.2, a regular expression is evaluated in the similar manner as an arithmetic expression. The procedure works as follows:

1. Convert regular expressions from infix notation into postfix notation.
2. Use basic operation to evaluate single postfix expressions and connect them through epsilon transitions.

token	action	output	operator stack
a	to output	a	
|	push	a	|
b	to output	a b	|
*	pop	a b |	
*	to output	a b | *	
.	push	a b | *	.
c	to output	a b | * c	.
.	pop	a b | * c .	
.	push	a b | * c .	.
d	to output	a b | * c . d	
$	pop	a b | * c . d .	

Table 2.1.: Shunting yard algorithm on **(a|b)*cd**.

The postfix notation is reached by applying the shunting yard algorithm on the regular expression. Table 2.1 demonstrates this on the regular expression **(a|b)*cd** [Dij61]. The dot represents the concatenation operator, it can be inserted while processing the input or beforehand. A concatenation operator, represented as a dot, is placed between every two input symbols that would result in an concatenation.The result of the shunting yard algorithm is **(a b | * c . d .)**. It is important to note at this point that the star operator is an unary operator, concatenation and union on the other hand are binary operators and therefore need two operands. The resulting postfix notation is a linear representation of parse tree of the regular expression. In order to present the nfa construction in a simpler manner figure 2.4 shows the parse tree. To get the non-deterministic finite state machine for this tree it is now just a matter of taking the templates, displayed in figure 2.1, 2.2 and 2.3 and instantiating them at the operator nodes. This is shown in figure 2.5, 2.6, 2.7 and 2.8. The instantiation is done bottom up. First step is replacing the union subtree with the template shown in figure 2.3. The tree nodes *union, a, b* are replaced by nodes *1-6*. Figure 2.5 shows the result of that replacement. The only difference to the template of figure 2.3 is the input character for the transition between the nodes 2, 4 and 3, 5.

The next step is to apply the star operation. The result is shown in figure 2.6. Again the result is pretty close to what the star operation template of figure 2.2 suggests. The last two steps is to apply the two concatenation.

The results of these two steps are shown in 2.7 and 2.8. Considering the postfix notation of and the parse tree of figure 2.4 no surprising result appear.

Now that we have created the NFA for the regular expression, we need to convert it into a DFA.

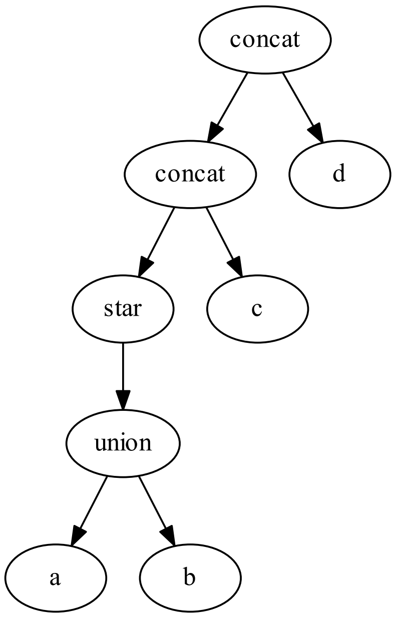


Figure 2.4.: Tree representation of postfix expression

2.2.4. NFA to DFA

Converting the NFA into the DFA serves several purposes. The first reason is that DFAs are easiert to execute, because they are not ambiguous. Another good reason converting NFAs into DFAs is that the number of states in a DFA is considerably lower than in a NFA. As every language that can be accepted by a NFA can be accepted by a DFA, it is a matter of using an algorithm to convert the non-deterministic finite state machine to a deterministic finite state machine. An algorithm for doing so is defined in [ASU86, p. 118]. Before we discuss the algorithm two helper functions are introduced. The epsilon-closure computation in listing 2.1 returns a set of all states reached when following all epsilon edges recursively. By doing so we can ignore the epsilon edges at a later state.

The *move* function returns a set of NFA states that is reached by the same input symbol from any of the NFA states in T. Listing 2.2 shows the implementation of the function. The implementation in listing 2.2 on page 36 is rather straight forward. The function iterates over all *nfa_nodes* in T. For every, of these elements, we get all transition on the passed input character. All elements of these sets get inserted into a set that is return from the function. The idea behind the listing 2.3 on page 36 is to simulate the NFA in every state with every possible input in parallel [ASU86, p. 159]. Figure 2.9 shows the resulting deterministic finite state machine, converted from the NFA of figure 2.8 on page 35. The resulting graph is five nodes in contrast to 18 of the final NFA. This is a reduction by a factor of at least three.

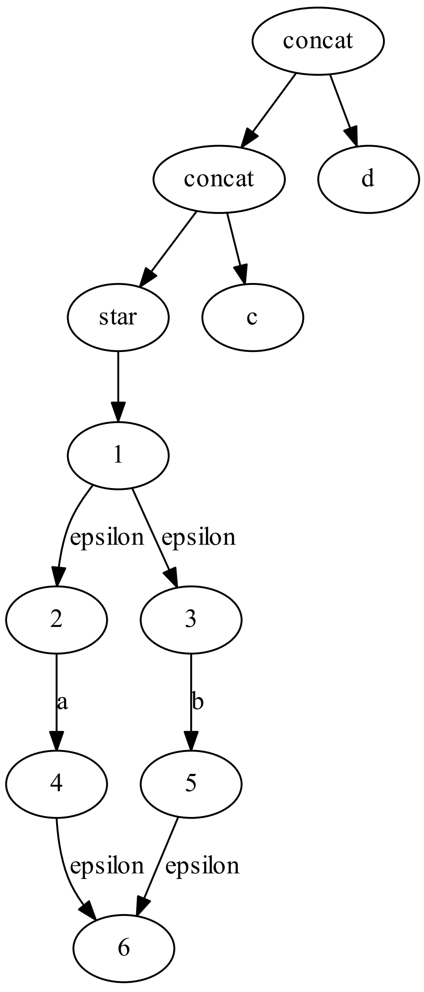


Figure 2.5.: Tree representation after insertion on union template

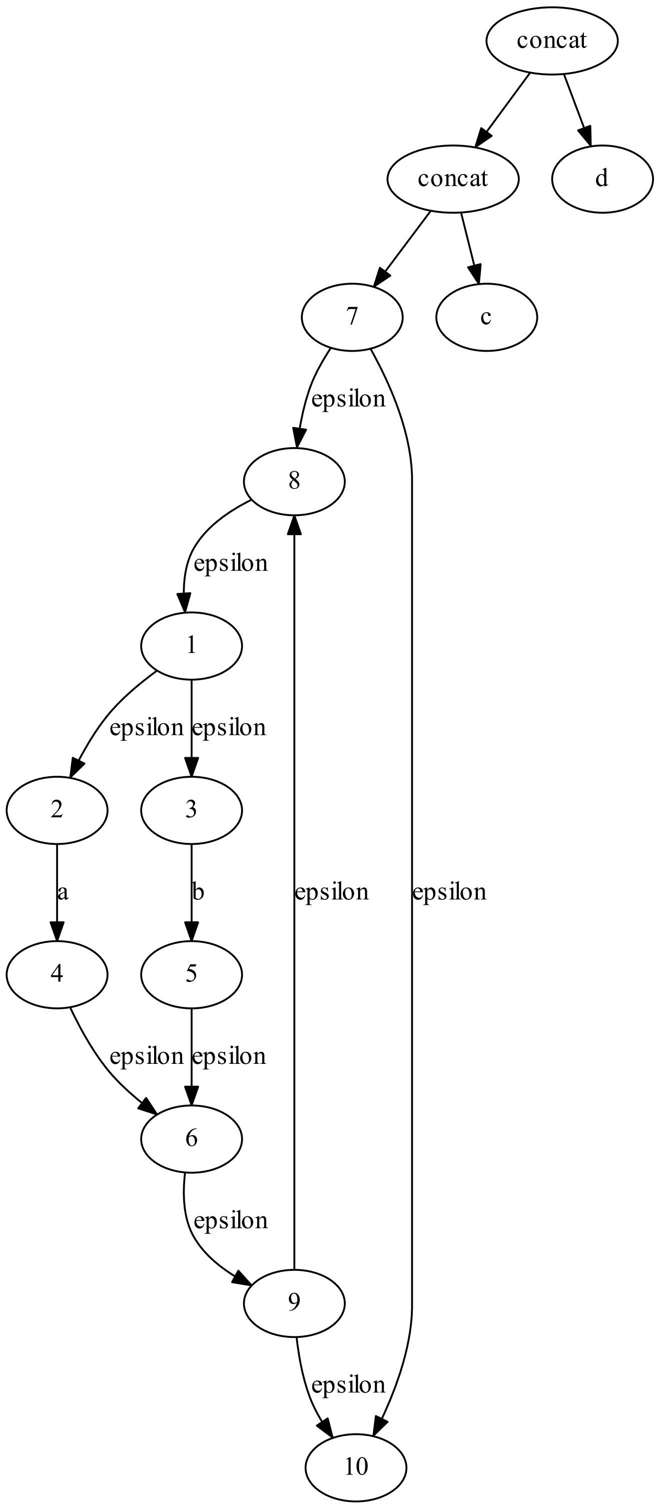


Figure 2.6.: Tree representation after insertion of star template

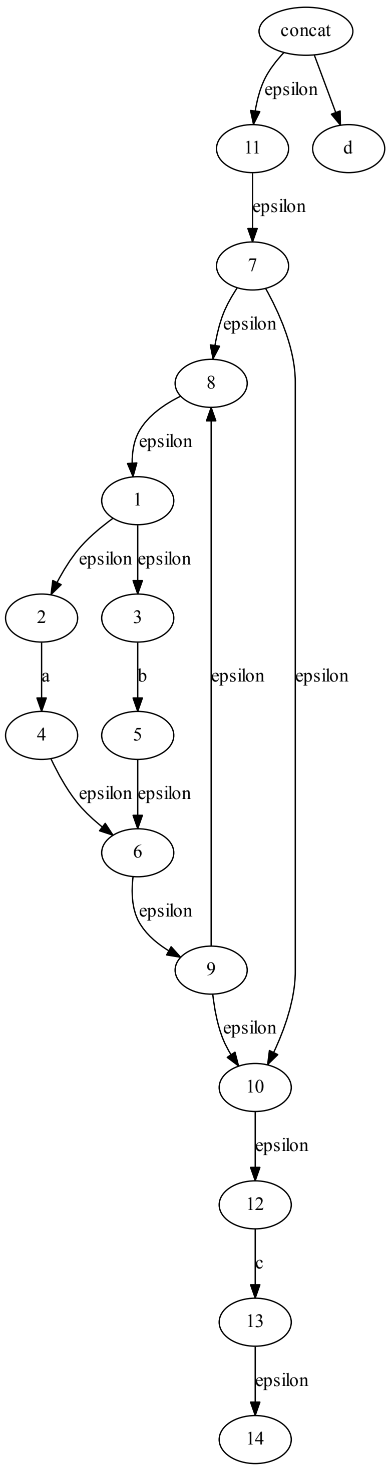


Figure 2.7.: Tree representation after first concatenation

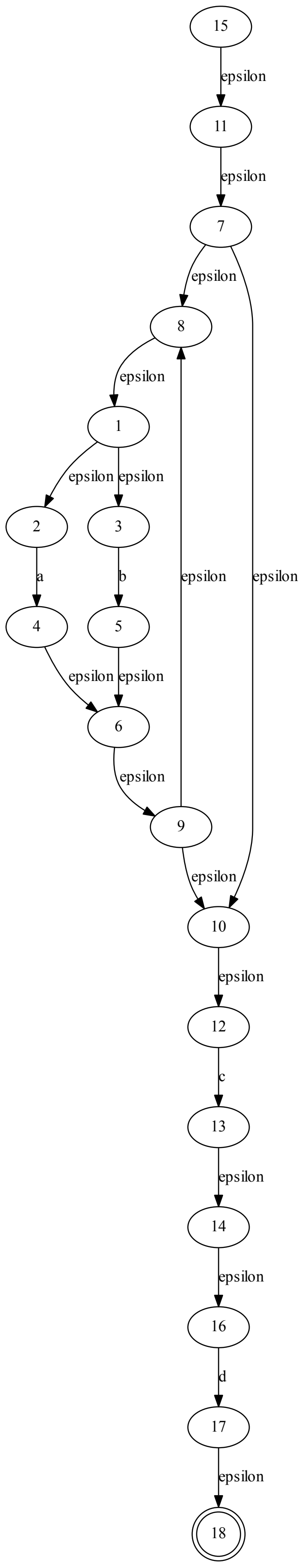


Figure 2.8.: Tree representation after second concatenation

Listing 2.1: Computation of epsilon-closure

```
Set!(nfaNodes) epsilon_closure(set_of_nfa_nodes T) {
    foreach(it; T) { stack.push(it); }
    initialize epsCol(T) to T;
    while(!stack.isEmpty() {
        t = stack.pop();
        foreach(state u ; Set!(edge) fromTtoUepsilon) {
            if(!epsCol(T).contains(u) {
                add u to epsCol(T);
                push u onto stack;
            }
        }
    }
    return epsCol;
}
```

Listing 2.2: move function

```
Set!(nfaNodes) move(char a, Set!(nfa_nodes) T) {
    foreach(nfa_node it; T) {
        Set!(nfaNodes) s = next_states(it, a);
        foreach(jt s) {
            append(ret, jt);
        }
    }
    return ret;
}
```

Listing 2.3: Subset construction

```
//initially, epsilon-closure(stateNum_0) is the only
//state in Dstates and it is unmarked;
while((T = Dstates.pop()) !is null) {
    T.mark();
    foreach(input symbol a) {
        U = epsilon_closure(move(T,a));
        if(!Dstates.contains(U)) {
            U.unmark();
            Dstates.push(U)
        }
        Dtran[T, a] = U;
    }
}
```

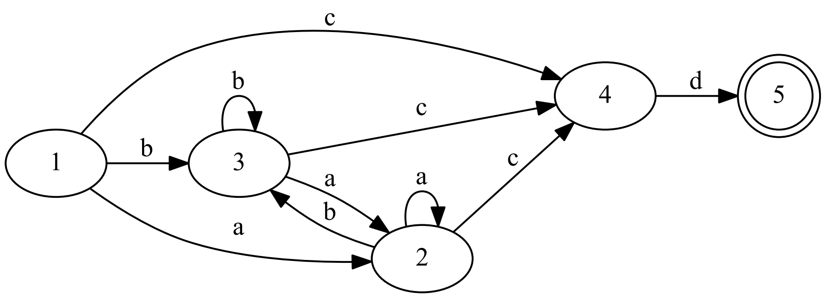


Figure 2.9.: DFA graph of regular expression **(a|b)*cd**.

2.2.5. DFA Minimization

Now that we have converted the non-deterministic finite state machine into a deterministic finite state machine we need to minimize the DFA to make it feasible for storing it into an array that is used to run the lexing algorithm. The algorithm used was presented by Hopcroft in 1971 [Hop71]. It runs in $O(n\,log\,n)$. A good pseudocode implementation is given in [Hol90, p. 143].

Listing 2.4: Hopcroft algorithm for DFA minimization

```
//Repeat until no new groups are added to groups

foreach(group; groups) {
    auto new = Set!(State)();
    auto first = the_first_state_in_the_current(group);
    auto next = the_next_state_of_the_current_(group);
    while(next) {
        foreach(character c) {
            goto_first = state reached by making a transition on c
               from first;
            goto_next = state reached by making a transition on c
               from next;

            if(goto_first is not in the same group as goto_next) {
                move next from the current group into new;
            }
        }
        if(new is not empty) {
            add it to groups;
        }
    }
}
```

step	group	state	a	b	c	d
1	1	1	2	3	4	-1
1	1	2	2	3	4	-1
1	1	3	2	3	4	-1
1	1	4	-1	-1	-1	5
1	2	5	-1	-1	-1	-1
2	1	1	2	3	4	-1
2	1	2	2	3	4	-1
2	1	3	2	3	4	-1
2	2	4	-1	-1	-1	5
2	3	5	-1	-1	-1	-1

Table 2.2.: DFA minimization steps

state	a	b	c	d
1	1	1	4	-1
4	-1	-1	-1	5
5	-1	-1	-1	-1

Table 2.3.: Minimized deterministic finite state machine

The idea behind the algorithm of John Hopcroft, that is shown in listing 2.4, is to start all states of the DFA in either of two groups. The first group contains all finishing states, the second groups contains every other state. Figure 2.8 and figure 2.9 both contain a double circular node, which means it is a final state. Then it is checked for every group, if all its members lead to the same group using the same input symbol. If a state does not lead to the same group, it is placed in the group called *new*. After a group has been processed and the new group is not empty it is placed in the set of groups. This is repeated till no new groups are created.

Running this algorithm on the DFA of figure 2.9 we get the steps shown in table 2.2

A transition labeled -1 means that the state has no transition on the given input, with other words an error would occur on that specific input.

As shown in table 2.2 the first step is to partition the states in two groups, accepting and normal states. The next step is to put state four into its own group, though its transitions are different to that of the other states in group one.

The final step of the DFA minimization is to merge all states of one group into one state. There are two steps to the finalization of a group.

1. Choose a state id.
2. Change the endpoint of all transition with any state in the finalized group to that of the chosen state id.

Applying this steps to the groups in the last step of table 2.2 we get three groups that are shown in table 2.3. A graphical representation is shown in figure 2.10. The number

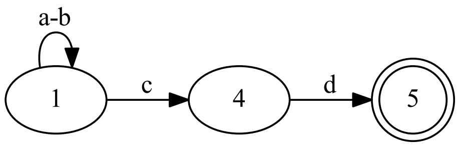


Figure 2.10.: Minimized DFA

state	a,b	c	d
1	1	4	-1
4	-1	-1	5
5	-1	-1	-1

Table 2.4.: Minimized table representation

of states has been reduced by 15 states which is a factor of five.

2.2.6. Minimizing the Transition Table of the Minimized DFA

Taking a look at table 2.3 we see that the columns for input a and b are equal therefore these can be merged. This can also be done for equal rows in table, but as there are no equal rows nothing can be done in this example. This table minimization does not change the DFA, it merely minimizes its representation, by the cost of other redirection. The redirection in this case is to map the input of a and b to the same column of the reduced array. Table 2.4 shows the reduced table. This technique is particularly useful when storing lexer tables that have groups of input character that are used for special purposes. For instance a lexer, that allows utf characters to be part of a string token, will properly have many rows that could be merged into one.

2.2.7. From Multiple Regular Expression to a Lexer

Above, the way of creating a deterministic finite state machine from a regular expression has been shown. Lexer accept multiple regex to accommodate multiple words in a language. The way of combining multiple regular expression is rather straight forward.

1. Create NFAs for every regex
2. Create new start state
3. Make an epsilon transition from the new start state to the beginning of every new regex

After the NFAs are merged into a single NFA the process continued as described in section 2.2.4 and earlier.

Another, rather practical, problem is to allow user supplied code to be run after a word has been matched to a defined regex. This is done by marking the end state of every created NFA with an unique identifier that points to the user code. This unique identifier has to be accommodated to the created DFA as well as the minimized DFA.

3. The Parser

3.1. Introduction

A parser checks if a token stream is matched by a grammar [ASU86]. This is done, because if a token stream is not well formed [1] it will be impossible to create a correct running program of it. Parsing algorithms can be divided into two major categories. The first category are top-down algorithms. The second category are bottom-up parsing algorithms. Both algorithm categories work on context free grammars or an useful subset of it.

3.2. Parser Types

3.2.1. Top-Down Parsing

Top down parser accept a word on the language from the top to the bottom.

3.2.1.1. Recursive Decent Parser

Recursive decent parsers usually define multiple functions that call each other recursively. Every function accepts a different rule of the grammar. Recursive decent parser are used by compilers like clang [com10] or gcc [fsf05].

Listing 3.1: Recursive decent parser example

```
void parseParameterList() {
    parseTyp();
    Token token = nextToken();
    if(token.typ == TokenTyp.RParen) {
        return;
    } else {
        assert(token == TokenTyp.identifier);
    }

    token = nextToken();
    if(token == TokenTyp.comma) {
        parseParameterList();
    }
```

[1]Well formed means, that the token is a word of the language.

```
}

void parseTyp() {
    Token token = nextToken();
    assert(token.typ == TokenTyp.void ||
        token.typ == TokenTyp.int ||
        token.typ == TokenTyp.float);
    return;
}

void parseSimpleFunctionPrototyp() {
    parseTyp();
    Token token = currentToken();
    assert(token.typ == TokenTyp.Identifier);
    token = nextToken();
    assert(token.typ == TokenTyp.Lparent);
    parseParameterList();
    assert(token.typ == TokenTyp.Rparent);
    token = nextToken();
    assert(token.typ == TokenTyp.Semicolon);
}
```

The listing 3.1 accepts all words described by the grammar shown in figure 3.1.

$$
\begin{array}{lll}
\langle \textit{Proto} \rangle & ::= & \langle \textit{Typ} \rangle\ \langle \textit{Identifier} \rangle\ \text{'('}\ \langle \textit{ParameterList} \rangle\ \text{')'}\ \text{';'} \\
 & | & \langle \textit{Typ} \rangle\ \langle \textit{Identifier} \rangle\ \text{'('}\ \text{')'}\ \text{';'} \\
\\
\langle \textit{Typ} \rangle & ::= & \text{'void'} \\
 & | & \text{'int'} \\
 & | & \text{'float'} \\
\\
\langle \textit{ParameterList} \rangle & ::= & \langle \textit{Typ} \rangle\ \langle \textit{Identifier} \rangle \\
 & | & \langle \textit{Typ} \rangle\ \langle \textit{Identifier} \rangle\ \text{','}\ \langle \textit{ParamterList} \rangle
\end{array}
$$

Figure 3.1.: EBNF for function prototyp

As context free grammars (cfgs) usually require a push-down automata to be accepted, recursive decent parser implement the stack implicitly by the use of the function call stack. The transition table is stored in the function itself.

3.2.1.2. LL Parser

Left to right, left derivation parser (LL) parser parse a grammar from left to right. In contrast to recursive descent parser, LL parsers keep an explicit stack and the grammar is stored in a transition table. There are two common types of LL parser, these are LL(1) and LL(k) parser. LL(1) parser use one token of lookahead to choose the production they follow. LL(k) use an arbitrary number of lookahead tokens to choose which production to follow. Both are described below.

3.2.1.2.1. LL(1) parser

The number in the parenthesis accounts for the number of lookahead token, in this case one. The listing 3.2 shows the LL parse algorithm.

Listing 3.2: LL parser algorithm

```
Stack!(Token) stack([eof, startSymbol]);
Token lookahead = empty;
do {
    if(lookahead == empty) {
        lookahead = nextToken();
    }

    Token currentSymbol = pop(stack);

    if(currentSymbol == lookahead) {
        lookahead = empty;
    } else if(currentSymbol == terminal && currentSymbol !=
        lookahead) {
        exit(ERROR);
    } else if(currentSymbol == nonterminal) {
        Token[] prod = Table[currentSymbol][lookahead];
        foreach_reverse(it; prod) {
            push(stack, it);
        }
    }
} while(!empty(stack));
```

Most programming languages can not be parsed by $LL(1)$ grammars [oP05].

3.2.1.2.2. LL(k) parser

LL(k) parsers use as many lookahead symbols as needed to distinguish between productions. It was long thought, that because of the number of lookahead tokens is variable, LL(k) parsers are impractical [?].

3.2.2. Bottom-Up Parsing

Button up parsers (LRs) works the opposite way to LL parsers. Each token is read first and the productions are matched from the right to the left. This method of parsing is also referred to as shift reduce parsing. Shift reduce parsers where invented by Donald Knuth in 1965 [Knu65]. Even though LR parsers are more powerful than LL parsers they can not accept the complete class of context free grammars. LR parsers usually work with a lookahead of one. There are three types of LR parsers that use one token in practice. The most powerful of the three are LR(1) parsers. On the other side of the spectrum are Simple button up parser (SLR)(1) parsers, they typically generate smaller parse table than LR(1) parsers. The last and most widely used are Lookahead bottom-up parser (LALR)(1) parsers. The parse table are equalent to that of SLR parser but they are nearly as powerful as LR parser. As the created parsers generator uses LALR(1) grammars and these combine parts of LR and glsaslr parser generator these are not discussed in any more detail.

3.2.3. Lalr(1) Parsing

LALR parsers are commonly used, for instance in popular tools like yacc and bison. While LALR(1) grammars can match languages like C they are unable to parse languages like C++. Even though they are widely used since the 1970, the year yacc first released in reference with Unix [JJ79]. The most important part of LALR(1) parsers is the parse table that drives the parsing algorithm. The construction of this table is presented in section 3.3.

The parser algorithm is shown as in listing 3.3.

Listing 3.3: Lalr(1) parsing algorithm

```
while(true) {
    action = trantable[top_of_stack,input];
    if(action == Accept) {
        break;
    } else if(action == Error) {
        reportError();
    } else if(action == Shift) {
        stack.pushBack(action);
        input = nextToken;
    } else if(action == Reduce) {
        stack.pop(length(action));
        stack.pushBack(goto[top_of_stack, left_side_of_reduction]);
    }
}
```

The algorithm is table driven. Every round an action is selected depending on the current top of stack symbol as well as the current lookahead character. Four types of action exists:

Accept when reaching an accept action the parser accepts the parse.

Shift on a shift action a value is pushed on the stack and the next token is read from the input.

Reduce when reaching a reduce action the number of items of the right hand side of the action are popped from the stack.

Error should none of the first three action take, an error is produced.

3.2.4. Parsing all of Chomsky Type-2

Unfortunately not all context free grammars or grammar of Chomsky Type-2, can be represented as an LALR(1) parse table. These grammars are ambiguous, as they have two or more possible actions, for some states. The problem with ambiguity is that in a traditional Bottom-up parser there must be exactly one action for every state lookahead combination. Even though most grammars used are nearly free of ambiguities a single ambiguity can pose serious problems and can lead to a considerable amount of work making the grammar fit into LALR(1). Ambiguities do arise usually in bigger grammars, nevertheless they can appear in very small grammars as well. Figure 3.2 shows a grammar

$$
\begin{array}{lcl}
\langle S \rangle & ::= & \langle \mathit{IfElse} \rangle \\
\\
\langle \mathit{IfElse} \rangle & ::= & \text{'if' 'bool'} \ \langle \mathit{IfElse} \rangle \\
 & | & \text{'if' 'bool'} \ \langle \mathit{IfElse} \rangle \ \text{'else'} \ \langle \mathit{IfElse} \rangle \\
 & | & \text{'statement'}
\end{array}
$$

Figure 3.2.: Dangling else

that is known as the dangling else grammar. The problem with this grammar is that an LALR(1) can not decide whether to shift the else or to reduce the if branch. The parse table which shows the problem is table 3.1. In itemset 6 on input *else* the parser can

	Action					**Goto**	
ItemSet	$	if	bool	else	statement	S	IfElse
0		s2			s3	g1	
1	$						
2			s4				
3	r3			r3			
4		s2			s3	g5	
5	r1			s6,r1			
6		s2			s3	g7	
7	r2			r2			

Table 3.1.: Dangling else parse table

either shift using rule six, which means as much as shift an else or it can reduce with rule one, which is in this case $IfElse \rightarrow \mathit{bool}\ IfElse$ rule. There are two types of conflicts that can occur, we saw the first, which is called shift-reduce conflict. Shift-reduce conflicts are usually solved by choosing the shift action over the reduce action [2]. The other kind of conflicts are reduce-reduce conflicts. The problem with those is that it is not clear which rule is the right one to choose. Choosing the right rule can be a complex task.

[2]This is basic operation for yacc and bison[Joh12].

Finding a sufficient solution for this is a critical if large and complex grammar are to be used with parser generators. The following a selection of parsing algorithm, that are able to parse the whole class of Chomsky Type-2 languages.

3.2.4.1. Glr Parsing

Generalized LR parsers are LR type parsers that are able to follow more than one rule at a state. They were introduced by Masaru Tomita in 1984 [Tom84]. Whenever an Generalized LR (GLR) parser sees more than one action it copies all information of the current parse. This includes for example the parse stack and intermediary results like the partially constructed abstract syntax tree. After enough copies are created to process all action independently, one action is mapped to each copy. It is desirable that at the end of parsing only one parse is left. If there are more than one parse left the resulting program would be ambiguous and this is generally not correct behavior for a programming language. To prevent this from happening there are multiple ways parses can be removed. The first is a parse error occurs. Should an error occur in an parse and there parses are present the parse gets removed silently. The other possibility is that two or more parses reach the same state. The relation same state is defined as a state which two or more parses have the same lookahead character and their parse stack is equal. At that point a function needs to be called, this function needs to choose one parse, making it the correct one. This merge function also needs to be called when more than one parse accepts. The worst case time complexity is $\mathcal{O}(n^3)$, but the less ambiguous the grammar is the closer the complexity gets to $\mathcal{O}(n)$ [Tom84, Tom87].

3.2.4.1.1. Elkhound

Elkhound is a parser and parser generator that was developed by Scott McPeak in 2002. Elkhound is basically a Glr parser. The difference to Tonita is that Elkhound does not copy the whole parse tree or any other data. Whenever two or more new states are created due to ambiguous rules, all new parse stack items will have the same parent node. All user defined actions that are usually run when a production is reduced are stored until a single parse is selected as the correct one. Another optimization of Elkhound is to count the number of stack elements that do not contain a fork. This number is used to calculate who long a simple LR parser algorithm can be used. This way the speed of Elkhound is close to that of LR parser even on grammars that are not LR [McP02, McP03].

3.2.4.2. Earley Parsing

The Earley parsers were invented by Jay Earley in 1970. They are part of the chart parser class that apply dynamic programming.[3] Earley parser have a runtime complexity of $\mathcal{O}(n^3)$, where n is the number of tokens. Like all parsing algorithms presented in this

[3]Chart parser store their temporary results in a chart, hence the name.

section Earley parsers accept all languages of Chomsky Type-Two. As Earley parsers parse the right hand side of the productions from left to right in a top down manner, they can conceptional be compared to LL parsers [Ear70, JA].

3.2.4.3. CYK

CYK parsers have a runtime complexity of $\mathcal{O}(n^3)$ sometimes even described as a complexity of $\mathcal{O}(n^3 * |G|)$, where G is the length of the parsed string. The grammar has to be presented in Chomsky normal form. They apply the production rules bottom up. Instead of a stack, CYK parser require a triangle like data structure that has $\frac{n(n+1)}{2}$ elements storage capacity, where n is the length of the input string. CYK parsers are named after there inventors John Cocke, Tadao Kasami and Daniel H. Younger. They were invented in the late sixties [Mar12].

3.2.5. Comparison

Elkhound is the fastest way to parse GLR languages, but the speed comes with complex data structures and user interaction. Earley parsers and CYK parsers have the worst average runtime complexity of the four presented algorithm. The GLR parsers presented by Tomita needs more memory than Elkhound as it does not share subtrees of the parse trees. One the other hand this allows for simple user interaction, as all user defined actions can be run directly after accepting a grammar rule. As the Tomita parser algorithm gives good performance with relatively easy usage and are closely linked to simple LALR(1) parsers they will be used for the parser generator in this thesis. Because of this the following section will discuss how such a parse table is constructed.

3.3. Parse Table Construction

In this section we will discuss how we construct an LALR parse table. There are multiple ways to construct the parse table. Here we present a method of constructing an LALR parse table through an SLR parse table. The way of constructing the parse table works through the following steps.

1. construct itemsets
2. transition table construction
3. creating an extended grammar
4. first set computation of an extended grammar
5. follow set computation of an extended grammar
6. action and goto table construction
7. extended follow set reduction
8. placing reduced follow set rules

These steps are explained in the following sections. The grammar to explain the steps is shown in figure 3.1 on page 42.

3.3.1. Itemset Construction

Itemsets are constructed by moving a dot through the productions. Whenever there is a dot in front of a non-terminal all productions with that non-terminal for a start symbol are added to the itemset. A dot is placed in front of the first token of every newly added production. If that symbol is a non-terminal this process is repeated. The next step is to create the follow itemsets. To achieve this, we group the productions the token following the dot. For every group we create a new itemset. After we have created the new itemset we move the dot to the next token. Then we complete the itemset as described earlier. For easier insertion of the end of input symbol, we need to insert a new start production into the grammar. Figure 3.3 shows this production. Completing the first itemset we get

$\langle S \rangle ::= \langle \mathit{Proto} \rangle$

Figure 3.3.: Start production

the itemset that is displayed in figure 3.4. Considering we start with the dot in front of the first token of the first production, we get line (0) in figure 3.4. Doing the expansion described earlier we get the rules one and two. The dot is in front of the first token of these added rules. Now that we have a rule with a dot in front of a *Typ* token we need to expand that group of rules as well, doing so leads to rules three to five shown in figure 3.4. As VOID, INT, FLOAT are terminal, no more rules need to expand.

As we have now constructed the first itemset we can create the follow itemsets. To do so, we first need to group the rules. Two or more productions are grouped together if the dot is in front of the same token. In figure 3.4 this is the case for production one and two. For every group in an itemset we create a new itemset. This is done by taking the productions of the group and moving the dot one token to the right. These productions are sometimes referred to as the kernel of an itemset. Nothing more needs to be done for this newly created itemset because the dot is in both cases in front of *Identifier* token and this token is a terminal. Should the dot be in front of a non-terminal we expand the itemset as described in the beginning of this section. In figure 3.5 we see that the kernel of the itemset is equal to the whole itemset. This is done for every production till there is an itemset where the dot is at the of end of every production.

These itemsets basically span a graph, where the edges are labeled by the token that is consumed by the movement of the dot from one itemset to the next. The whole graph is shown in figure 3.6 on page 50. Even though the number of productions is rather small, in comparison to a programming language like C, the number of itemsets growths rapidly.

So far the construction of the parse table does not differ from that of an SLR parse table.

```
State #0
(0) S -> •Proto
(1) Proto -> •Typ Identifier ( ParameterList ) ;
(2) Proto -> •Typ Identifier ( ) ;
(3) Typ -> •void
(4) Typ -> •int
(5) Typ -> •float
```

Figure 3.4.: First completed itemset

```
State #2
(1) Proto -> Typ •Identifier ( ParameterList ) ;
(2) Proto -> Typ •Identifier ( ) ;
```

Figure 3.5.: Second itemset

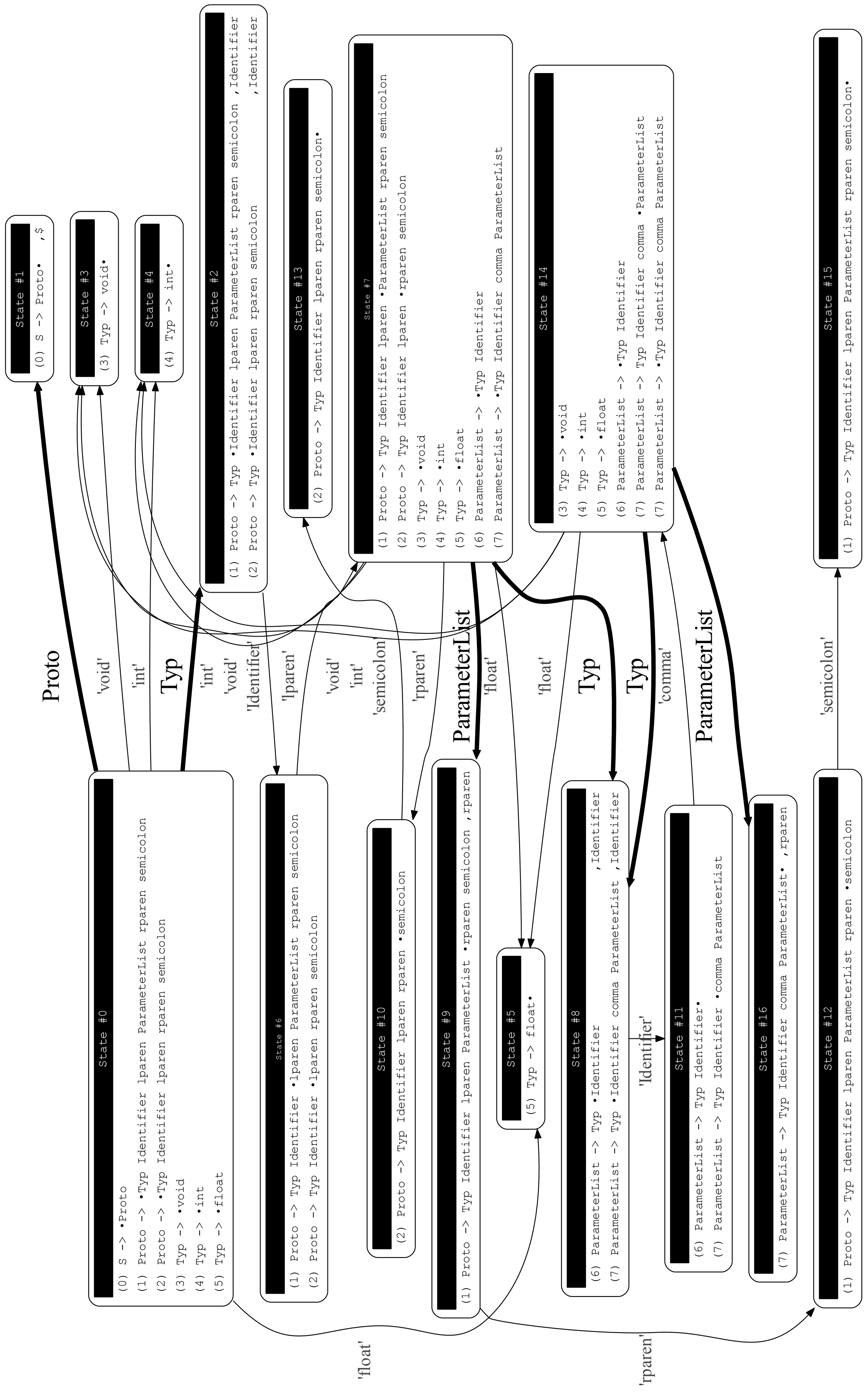

State #0
(0) S -> •Proto
(1) Proto -> •Typ Identifier lparen ParameterList rparen semicolon
(2) Proto -> •Typ Identifier lparen rparen semicolon
(3) Typ -> •void
(4) Typ -> •int
(5) Typ -> •float
State #1
(0) S -> Proto• ,$
State #2
(1) Proto -> Typ •Identifier lparen ParameterList rparen semicolon ,Identifier
(2) Proto -> Typ •Identifier lparen rparen semicolon ,Identifier
State #3
(3) Typ -> void•
State #4
(4) Typ -> int•
State #5
(5) Typ -> float•
State #6
(1) Proto -> Typ Identifier •lparen ParameterList rparen semicolon
(2) Proto -> Typ Identifier •lparen rparen semicolon
State #7
(1) Proto -> Typ Identifier lparen •ParameterList rparen semicolon
(2) Proto -> Typ Identifier lparen •rparen semicolon
(3) Typ -> •void
(4) Typ -> •int
(5) Typ -> •float
(6) ParameterList -> •Typ Identifier
(7) ParameterList -> •Typ Identifier comma ParameterList
State #8
(6) ParameterList -> Typ •Identifier ,Identifier
(7) ParameterList -> Typ •Identifier comma ParameterList ,Identifier
State #9
(1) Proto -> Typ Identifier lparen ParameterList •rparen semicolon ,rparen
State #10
(2) Proto -> Typ Identifier lparen rparen •semicolon
State #11
(6) ParameterList -> Typ Identifier•
(7) ParameterList -> Typ Identifier •comma ParameterList
State #12
(1) Proto -> Typ Identifier lparen ParameterList rparen •semicolon
State #13
(2) Proto -> Typ Identifier lparen rparen semicolon•
State #14
(3) Typ -> •void
(4) Typ -> •int
(5) Typ -> •float
(6) ParameterList -> •Typ Identifier
(7) ParameterList -> Typ Identifier comma •ParameterList
(7) ParameterList -> •Typ Identifier comma ParameterList
State #15
(1) Proto -> Typ Identifier lparen ParameterList rparen semicolon•
State #16
(7) ParameterList -> Typ Identifier comma ParameterList• ,rparen
Proto
'void'
'int'
Typ
'int'
'void'
'Identifier'
'lparen'
'void'
'int'
'semicolon'
'rparen'
ParameterList
'float'
'float'
Typ
Typ
'comma'
ParameterList
'semicolon'
'Identifier'
'float'
'rparen'

Figure 3.6.: Complete itemset graph

3.3.2. Transition Table Construction

The construction of the transition table is straight forward. As we have constructed the complete itemset graph in section 3.3.1, show in figure 3.6, we now transform the graph into a table. The algorithm for this procedure is shown in listing 3.4.

Listing 3.4: Transition table construction

```
int table[numberOfState][numberOfInputStrings];
foreach(itemset it; itemsets) {
    foreach(string jt; inputStrings) {
        table[it][inputString] = it.getFollowItemset(jt);
    }
}
```

What is does is creating a two dimensional array and fills it with the id of the following itemset[4]. The collection with the name *inputString* contains all terminals and non-terminal. If an itemset has no follow itemset on a given input, the entry in the table is an error.

ItemSet	identifier	(	)	;	void	int	float	,	S	Proto	Typ	ParameterList
0					3	4	5			1	2	
1												
2	6											
3												
4												
5												
6		7										
7			10		3	4	5				8	9
8	11											
9			12									
10				13								
11								14				
12				15								
13												
14					3	4	5				8	16
15												
16												

Table 3.2.: Transition table for graph 3.6

This table will be used in a later stage to indicate the shift and goto symbols.

[4]Using the inputString as array index will likely not yield an index aligned to zero. The procedure of aligning is omitted for simplicity reasons.

3.3.3. Creation of Extended Grammars

The result of this step is used as a basis for the next two steps. The extended grammar can be understood as the combination of the itemset graph and the given grammar. For every production in every itemset that has a dot in front of the first right hand token we create a new extended grammar production. Such an extended production consist of so called extended items. An extended item consists of a token connecting two itemsets and the ids of the connected itemsets. As an example we are going to do this for production two of itemset zero. At first the dot is in front of the token labeled *Typ*. Following that token we get into itemset two, so the first item of the extended grammar rule is ${}_0Typ_2$. Now the dot is in front the *identifier* token, following that token we get to itemset six. The corresponding extended grammar item is ${}_2identifier_6$, as we go from itemset two to itemset six on token *identifier*. The next item we get is ${}_6({}_7$, because we follow the left parenthesis token from itemset seven to itemset nine. The last two items we get of that new production are ${}_9)_{12}$ and ${}_{12};_{15}$. When we combine all created item we get the right hand side of the first production of the extended grammar. This right hand sides reads ${}_0Typ_2\ {}_2identifier_6\ {}_6({}_7\ {}_7)_{10}\ {}_{10};_{12}$. The only thing missing at this point is the left hand side of this production. The left hand side is created by treading the left hand side of the original production as input to itemset zero. Doing so leads us to itemset one. So the left hand side reads ${}_0Proto_1$. Combining left and right hand side we get the first complete extended grammar rule: ${}_0Proto_1 \rightarrow_0 Typ_2\ {}_2identifier_6\ {}_6({}_7\ {}_7)_{10}\ {}_{10};_{13}$

One special case exists for the dedicated start rule, as it has no transition out from itemset zero on input S, because of this the left hand side always reads ${}_0S_\$$. This means that this production is the accepting production. Doing this for all itemsets and all productions we get the extended grammar shown in table 3.3 on page 53.

3.3.4. First Set Computation

At this stage we need to compute the first set for every non-terminal. The first set holds the symbols that can be found as the first terminal on the right hand side of a production. The first set can be computed considering the following three rules.

1. If x is a terminal First(x) = x
2. If $V \rightarrow x$; First(V) = x, also if $V \rightarrow \epsilon$; First(V) = ϵ
3. If $V \rightarrow A\,B\,C$ where A,B,C are non-terminals we add First(A) minus ϵ to First(V). If First(A) contains ϵ we do the same with B. Should we reach C and First(C) contains ϵ we add ϵ to First(V).

These rules are applied as long as the first sets of the non-terminals change. After applying the rules we get the following result shown in table 3.4 on page 54. All first sets look the same, that is due to the fact that an extended grammar item of kind *Typ* is, aside from *Proto*, the only non-terminal that starts a right hand side of an extended

$_0S_\$ \rightarrow_0 Proto_1$
$_0Proto_1 \rightarrow_0 Typ_2\ {}_2identifier_6\ {}_6(_7\ {}_7ParameterList_9\ {}_9)_{12}\ {}_{12};_{15}$
$_0Proto_1 \rightarrow_0 Typ_2\ {}_2identifier_6\ {}_6(_7\ {}_7)_{10}\ {}_{10};_{13}$
$_0Typ_2 \rightarrow_0 void_3$
$_0Typ_2 \rightarrow_0 int_4$
$_0Typ_2 \rightarrow_0 float_5$
$_7Typ_8 \rightarrow_7 void_3$
$_7Typ_8 \rightarrow_7 int_4$
$_7Typ_8 \rightarrow_7 float_5$
$_7ParameterList_9 \rightarrow_7 Typ_8\ {}_8identifier_{11}$
$_7ParameterList_9 \rightarrow_7 Typ_8\ {}_8identifier_{11}\ {}_{11},_{14}\ {}_{14}ParameterList_{16}$
$_{14}Typ_8 \rightarrow_{14} void_3$
$_{14}Typ_8 \rightarrow_{14} int_4$
$_{14}Typ_8 \rightarrow_{14} float_5$
$_{14}ParameterList_{16} \rightarrow_{14} Typ_8\ {}_8identifier_{11}$
$_{14}ParameterList_{16} \rightarrow_{14} Typ_8\ {}_8identifier_{11}\ {}_{11},_{14}\ {}_{14}ParameterList_{16}$

Table 3.3.: Extended grammar rules

rule. The first set of every *Typ* item is always followed by VOID, INT, FLOAT, so all first sets devised from that contain these three as well.

The first sets will be used in the follow set computation, which is described in section 3.3.5.

3.3.5. Follow Set Computation

Now that we have constructed the first sets for the extended grammar we can construct the follow sets. Follow sets are used to define the reduce operation in the final parse table. In layman's terms a follow set is a set of terminals that follow a non-terminal in the right hand side of a production. This needs to be done on the extended grammar, because these sets can differ between extended grammar items even though they have the same symbols as the normal grammar. Hence, $_7Typ_8$, $_0Typ_2$ are different non-terminals for the parser.

To compute the follow sets we need four rules. For these rules to make sense some definitions need to be made. The definitions read: [5]

1. a,b and c represent terminal or non-terminal symbols
2. a* represents zero or more terminals or non-terminals

[5]At this point it is not present which symbols of the extended grammar are terminals or non-terminals. These information need to be computed before follow sets can be created. To compute these information's we need to iterate over all productions of the extended grammar. Initially all symbols are placed in the terminal set. While we iterate the productions we remove the left hand side of the production from the terminal set and place it into the non-terminal set.This finally gives use two non-intersecting sets of terminals and non-terminals.

$$\begin{aligned}
\text{First}({}_0S_\$) &= \text{void, int, float}\\
\text{First}({}_0Proto_1) &= \text{void, int, float}\\
\text{First}({}_{14}Typ_8) &= \text{void, int, float}\\
\text{First}({}_7yp_8) &= \text{void, int, float}\\
\text{First}({}_0yp_2) &= \text{void, int, float}\\
\text{First}({}_8identifier_{11}) &= \text{identifier}\\
\text{First}({}_2identifier_6) &= \text{identifier}\\
\text{First}({}_6(_7) &= (\\
\text{First}({}_7ParameterList_9) &= \text{void, int, float}\\
\text{First}({}_{14}ParameterList_{16}) &= \text{void, int, float}\\
\text{First}({}_9)_{12}) &=)\\
\text{First}({}_7)_{10}) &=)\\
\text{First}({}_{12};_{15}) &= ;\\
\text{First}({}_{10};_{13}) &= ;\\
\text{First}({}_{14}void_3) &= \text{void}\\
\text{First}({}_7void_3) &= \text{void}\\
\text{First}({}_0void_3) &= \text{void}\\
\text{First}({}_{14}int_4) &= \text{int}\\
\text{First}({}_7int_4) &= \text{int}\\
\text{First}({}_0int_4) &= \text{int}\\
\text{First}({}_{14}float_5) &= \text{float}\\
\text{First}({}_7float_5) &= \text{float}\\
\text{First}({}_0float_5) &= \text{float}\\
\text{First}({}_{11},_{14}) &= ,
\end{aligned}$$

Table 3.4.: First sets of the extended grammar

3. a+ represents one or more terminals or non-terminals
4. D is a nonterminal

Now for the rules:

1. An end of input symbol (\$) is placed into follow set of the starting rule
2. Considering a rule $R \rightarrow a^*Db$. Every element of the first set of b, with exception of ϵ is placed in Follow(D). In short $Follow(D) = First(b) - \epsilon$. If First(b) contains ϵ then everything in Follow(R) is put in Follow(D), too.
3. If a rule reads $R \rightarrow a^*D$ then everything in Follow(R) is placed into Follow(D).
4. The follow set of all terminals are empty.

Similar to the first set computation these rules, expect rule one, are applied as long as the follow sets change. Rule number one has the purpose of making the first rule been able to accept input.

The end of input symbol (\$) is returned by the lexer after no more characters can be read from the input. The second rule which is does what the layman's term description implied.

The third rule is introduced for clarity. The last sentence of rule two reads: If First(b) contains ϵ then everything in Follow(R) is put in Follow(D). The ϵ has the meaning of a not present token. If we remove the last b from the production in rule two we get the production of rule number three. The idea by the back-referencing, through assigning the follow set of R to D, is that after we are done with parsing D we continue with whatever comes after R

Rule four simply means that the follow set of a terminal is of no interest, since it gets merged into the follow sets of non-terminals.

Table 3.5 shows the follow set for the extended grammar of figure 3.3. The follow sets

Follow($_0S_\$$)	= \$
Follow($_0Proto_1$)	= \$
Follow($_{14}Typ_8$)	= identifier
Follow($_7Typ_8$)	= identifier
Follow($_0Typ_2$)	= identifier
Follow($_{14}ParameterList_{16}$)	= (
Follow($_7ParameterList_9$)	= (

Table 3.5.: Follow set of extended grammar

for $_0S_\$$ and $_0Proto_1$ depend only on rule one and three. The rest of the follow sets are created through the use of rule two.

It is important to understand here, that the token in the follow sets are token of the normal grammar, that means they don not have two numbers linking them to a specific itemset. This is because these token need to match a token coming from the lexer.

The follow sets will be used, in a later step, to create the reduce operations in the final parse table.

3.3.6. Extended Follow Set Reduction

Now having the follow sets and the extended grammar, we need to convert them into a form that enables us to use them as reductions in the final parse table. The first step is to merge the extended rules based on their base rule and the last itemset number of the last extended item of the right hand side. [6] When merging one or more rules their follow sets get merged as well and assigned to a newly merged rule. Finally we assign a final set number to every rule. This final set number is again the itemset number of the last token of the right hand side.

Before we merge the rules we combine the follow set of the extended grammar and their matching production. The follow set gets selected based on the right hand side of the production. Doing this gets us the combination shown in table 3.6. This table is used to carry out the merge operation as described above.

[6]Base rule means in this case, the rule without the itemset numbers to the left and right of the actual token.

Number	**Rule**	**Follow set**
1	${}_0S_\$ \rightarrow_0 Proto_1$	\$
2	${}_0Proto_1 \rightarrow_0 Typ_2\ {}_2identifier_6\ {}_6(_7\ {}_7ParameterList_9\ {}_9)_{12}\ {}_{12};_{15}$	\$
3	${}_0Proto_1 \rightarrow_0 Typ_2\ {}_2identifier_6\ {}_6(_7\ {}_7)_{10}\ {}_{10};_{13}$	\$
4	${}_0Typ_2 \rightarrow_0 void_3$	identifier
5	${}_0Typ_2 \rightarrow_0 int_4$	identifier
6	${}_0Typ_2 \rightarrow_0 float_5$	identifier
7	${}_7Typ_8 \rightarrow_7 void_3$	identifier
8	${}_7Typ_8 \rightarrow_7 int_4$	identifier
9	${}_7Typ_8 \rightarrow_7 float_5$	identifier
10	${}_7ParameterList_9 \rightarrow_7 Typ_8\ {}_8identifier_{11}$	(
11	${}_7ParameterList_9 \rightarrow_7 Typ_8\ {}_8identifier_{11}\ {}_{11},_{14}\ {}_{14}ParameterList_{16}$	(
12	${}_{14}Typ_8 \rightarrow_{14} void_3$	identifier
13	${}_{14}Typ_8 \rightarrow_{14} int_4$	identifier
14	${}_{14}Typ_8 \rightarrow_{14} float_5$	identifier
15	${}_{14}ParameterList_{16} \rightarrow_{14} Typ_8\ {}_8identifier_{11}$	(
16	${}_{14}ParameterList_{16} \rightarrow_{14} Typ_8\ {}_8identifier_{11}\ {}_{11},_{14}\ {}_{14}ParameterList_{16}$	(

Table 3.6.: Combination of follow set and extended grammar productions.

Rules that can be merged are $4, 7, 12$, $5, 8, 13$, $6, 9, 14$, $10, 15$ and $11, 16$ as they decent from the same basic rules and share a common last itemset number. Running the merge operation exemplary on the first tuple we get a new rule follow set combination that is shown in table 3.7. As expected the new final number is three, due to being three the

Final number	**Pre merge rules**	**Rule**	**Follow set**
3	$4, 7, 12$	$Typ \rightarrow void$	identifier

Table 3.7.: Exemplary result of merge operation on rules $4, 7, 12$

last itemset number of the right hand side. The column *pre merge rules* is inserted for readability. The third column, labeled rule, shows the base rule the three productions accent from. The last column finally shows the follow set, as all three rules have the same elements in their respective follow set, the merged follow set only contains the *identifier* token. Doing the merge operation for the rest of merge candidates as well as the single rules we get table 3.8. The only thing that is interesting in the result is, that almost all not merged rule got a final number that is unlike their pre merged number. This is because their final number is selected by evaluating the right most itemset number of the last token of the right hand side of each production. We will use this table in section 3.3.7 to obtain the reduction actions.

Final number	**Pre merge rules**	**Rule**	**Follow set**
1	1	$S \rightarrow Proto$	\$
3	$4, 7, 12$	$Typ \rightarrow void$	identifier
4	$5, 8, 13$	$Typ \rightarrow int$	identifier
5	$6, 9, 14$	$Typ \rightarrow float$	identifier
11	$10, 15$	$ParameterList \rightarrow Typ\ identifier$	(
13	3	$Proto \rightarrow Typ\ identifier\ (\ )\ ;$	\$
15	2	$Proto \rightarrow Typ\ identifier\ (\ ParameterList\ )\ ;$	\$
16	$11, 16$	$ParameterList \rightarrow Typ\ identifier\ ParameterList$	(

Table 3.8.: Merged extended rules and follow sets

3.3.7. Action and Goto Table Construction

This step constructs the final parse table. As its input we use the transition table shown on page 51.

The construction of the action and goto table is done in four steps. These steps are:

1. Place accept actions
2. Create goto part
3. Place shift actions
4. Place reduction actions

The first thing that needs to be done is to create a column for the end of input token \$. After that we need to place an accept action in every row where the corresponding itemset contains the start production with the dot at its end. As we can see in figure 3.6 that is only the case for itemset one. That means we have to place the accept action in row one.

The next step is to create the goto part of the table. This step is as simple as copying the non-terminal entries of table 3.2 into the new final parse table.

Placing the shift action is not much harder than creating the goto part of the table. To create the shift entries we copy the terminal entries of the transition table and place an s in front of them to indicate that they are shift operations. The numbers are not changed.

Finally we place the reduction action according to the merged extended rules of table 3.8. The final number indicates the row and the follow symbol is used as the column index. At that position in the table a reduce action is placed, that corresponds to the rule of the rule column of the table 3.8. To make the final parse table better readable the productions are enumerated here:

1. $Proto \rightarrow Typ\, identifier$ ($ParameterList$) ;
2. $Proto \rightarrow Typ\, identifier$ ( ) ;
3. $Typ \rightarrow void$
4. $Typ \rightarrow int$
5. $Typ \rightarrow float$
6. $ParameterList \rightarrow Typ\, identifier$
7. $ParameterList \rightarrow Typ\, identifier$, $ParameterList$

A reduction action like a for example $r4$ means reduce with rule $Typ \rightarrow int$. The final parse table is show in table 3.9. The only part different from the construction of an SLR

	Action									**Goto**			
ItemSet	$	identifier	(	)	;	void	int	float	,	S	Proto	Typ	ParameterList
0						s3	s4	s5			1	2	
1	accept												
2		s6											
3		r3											
4		r4											
5		r5											
6			s7										
7				s10		s3	s4	s5				8	9
8		s11											
9				s12									
10					s13								
11			r6						s14				
12					s15								
13	r1												
14						s3	s4	s5				8	16
15	r2												
16			r7										

Table 3.9.: Final parse table

parse table is the way the reduction actions are created and placed in the table. This method of parse table construction is therefore sometimes called LALR by SLR.

4. The Compiler

4.1. Introduction

A compiler is an program that takes source code and creates an executable program of it. To reduce complexity a compiler is usually split into two parts. These two parts consist of three sub-parts each. The first part is called the frontend. The frontend analyses the input for correctness.[1] The second group is called the backend. The backend creates the actual executable.

4.2. The Frontend

The frontend has to check the input for errors and to pass the backend all needed information to create an executable.

4.2.1. Parts of the Frontend

The aim of the frontend is check if the source code form a word of the programming language and to check if all semantic rules are abided. The parsing of the languages plays a big rule in accepting the input. To facilitate this, the first two parts of the frontend are tiddly coupled to achieve this.

4.2.1.1. Lexical Analysis

The lexical analysis reads the input character by character and forms tokens of it. This process is discussed in section 2 on page 27 and following.

4.2.1.2. Syntactical Analysis

The syntactical analysis consists of a parser that reads one token after another from the lexer. Different kinds of parsers are discussed in detail in section 3 on page 41 and following. It is important to understand that a lexer only generates token when parser requests them. While matching the input token to constructs of the languages the parser usually generates a tree like representation of the input.

[1]Correctness in the sense that it matches the language's definition.

4.2.1.2.1. Parse Trees

A parse tree are an one to one representation of the parsed token stream. Considering a rule for a *if* condition in figure 4.1 we get the parse tree shown in figure 4.2. A parse

$\langle \mathit{If} \rangle$ ::= 'if' '(' $\langle \mathit{Condition} \rangle$ ')' '{' $\langle \mathit{BlockStatement} \rangle$ '}'

Figure 4.1.: Grammar rule for if condition

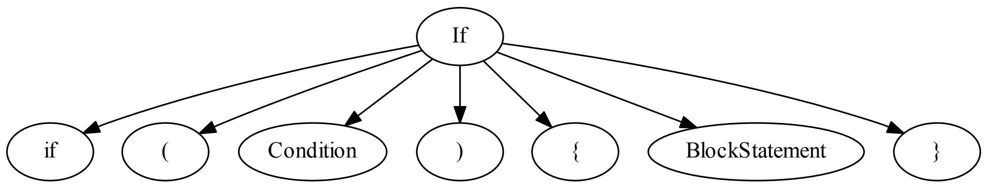


Figure 4.2.: Parse tree for the if grammar rule

tree contains all tokens received the lexer. Parse trees can be generated very easily from a grammar. The left hand side is the parent of all items on the right hand side of a production. Unfortunately not all token are required for the following steps. For instance the opening and closing parenthesis as well as the curly braces are not important as they do not carry information that is not reconstructible from the other nodes.

4.2.1.2.2. Abstract Syntax Trees

To overcome the shortcomings of parse trees, abstract syntax trees are created. As the name suggests the syntax is not converted one to one, instead all redundant information are left. Figure 4.3 shows a possible abstract parse tree for the *if* grammar. Building an

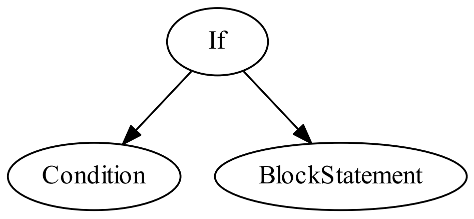


Figure 4.3.: The abstract parse tree for the if grammar rule

abstract syntax tree is a bit more complex than building a parse tree. This is because depending on the rule different tokens can be omitted from the tree. Building an abstract syntax tree one can remove a lot of nodes from the tree to represent the input. This is important because the input can be arbitrary long and as parse trees are usually generated on the heap fewer calls to the allocator procedures allows the tree to be build faster. Another good side effect, of the reduced number of nodes in the tree, is that traversing it in semantic analysis is faster, because less nodes have to be checked if they match certain types of nodes.

4.2.1.2.3. Symbol Table

A symbol table is a data structure that stores symbols like functions or variables and their visibility.[2] This structure is usually built while parsing.

4.2.1.3. Semantic Analysis

A The semantic analysis verifies if rules, that could not be expressed by a grammar of the language, are obeyed. These rules are for instance whether all called function are visible or if all assignments are in agreement with the type system. The complexity as well as the runtime of the semantic analysis can vary significantly depending on the programming language. Depending on the semantic analysis the parse tree might be modified in certain ways.

4.3. The Backend

After the semantic analysis is done, the abstract parse tree is passed to the backend. The backend has the task to create an executable file. This is usually done in three steps.

1. Intermediary code generation
2. Optimization of the intermediary code
3. Executable generation

The intermediary code is usually represented in a programming language that could run on an abstract central processing unit (CPU). These languages are normally designed to be easily transferable into platform specific machine code. The reason for converting the parse tree into a temporary representation is to run the same optimization strategies for different programming languages. This way the complexity of the backend can be reduced and or the same backend can be used for different compilers.

As already mentioned the optimization runs on the intermediary code. The code is usually optimized for speed and space requirements.

The last step is to create the actual executable. Two ways are possible. The first is to directly emit the machine code from the optimized intermediary code. The second is to create assembler code for the CPU and call an assembler to create the machine code.

[2]Visibility in this context means whether or not a piece of program has a given symbol in its scope.

Part III.

Implementation

5. The library Libhurt

The default *D* compiler comes with a standard library. Due to historical reason the library is split into two parts, *druntime* and *phobos*. For *D1* two standard libraries exists. Some members of the D community where unhappy with the scale of *phobos* for *D1*, so they developed *Tango*. They did not stop at adding high level functionality. Over time a complete new runtime was created, from garbage collection to string parameter mapping. This has finally lead to incompatibility between the two libraries. At some point programmers had to choose. To avoid this in *D2* all low level functionality was placed in *druntime*. High level functionality like math functions for stream- and file-abstraction where placed in *phobos*.

Even though a library is present some vital functionality for compiler construction is missing. Most of it are container classes. As mentioned earlier the idea developed from studying different kind of *std::vector* implementations in *D*. Data structures take a big role in compiler construction, they even marked the start of this project. Other components of the library are logging, string formatting, string utilities, random number generators, stream- and file-abstraction, algorithms and time handling. Random number generators and stream-abstraction where ported from *Tango* respectively *phobos*. [1]

The following chapters will explore the design and the implementation of the container classes because they played a key role in the development of this project. These classes include different kind of lists, array like types, maps, sets and multimaps.

5.1. Insert Search Remove interface

The idea behind the insert search remove interface is that containers like maps, multimaps or even sets need underlying data structures that offer the three basic operations insert, search and remove to work. Through the use of polymorphism the interface of these containers can stay the same but the runtime properties can be changed by the passing of an argument. This is good, because this allows the program to exchange the data structures it uses at runtime. It is important to note here that the data structures of container classes cannot change at runtime, instead a copy of the container can be created using a different underlying data structure. To further facilitate this idea the insert search remove (ISR) specification also introduces an interface for iterators and storable types.

[1] *Tango* is licensed under a GPL compatibly *BSD* license and *phobos* is licensed under the *Boost* license, which is also GPL compatible [Ige04] [com12a].

5.1.1. The insert search remove specification

Listing 5.1: ISR specification

```
enum ISRType {  // This enumerates all available underlaying
    BinVec,     // data structures
    BinarySearchTree,
    HashTable,
    RBTree
}

abstract class ISRNode(T) {
    T getData();
}

interface ISR(T) {
    public bool insert(T data);
    public bool remove(T data);
    public bool remove(ISRIterator!(T) data, bool dir = true);
    public ISRIterator!(T) begin();
    public ISRIterator!(T) end();
    public ISRNode!(T) search(T data);
    public ISRIterator!(T) searchIt(T data);
    public bool isEmpty() const;
    public size_t getSize() const;
}

abstract class ISRIterator(T) {
    public void opUnary(string s)() if(s == "++") { increment(); }
    public void opUnary(string s)() if(s == "--") { decrement(); }
    public T opUnary(string s)() if(s == "*") { return getData(); }
    public T getData();
    public bool isValid() const;
    public ISRIterator!(T) dup();
    public void increment();
    public void decrement();
}
```

The ISR specification consists of four parts. The first, the *ISRType* enum listing all available ISR implementation. At the moment libhurt provides four implementations which are presented later. In practice this ENUM gets used to control the control flow of the data structure construction.

The second part of the specification presents an abstract template class called *ISRNode*. Derivations of this class are what is actually stored by the ISR implementations. The template parameter *T* indicates the type of data the data structure stored. The only thing this class enforces is that there is a *getData* function. This function returns the data stored by the node. Everything else is up to the specific data structures. For instance the tree implementations would also store the child and parent pointers in this class.

The interface ISR defines all functions a ISR data structure needs implement. On top of the *insert*, *search* and *remove* function some more functions are defined. These functions don not change the basic idea, but are merely convince functions. The only function that are new and interesting are *begin* and *end*, both return an *ISRIterator*, which is an iterator type class. The iterator, returned by *begin*, points to the first element of the ISR data structure instance. As the name suggest, the iterator returned by*end* points to the last element of the ISR data structure. The function *getSize* returns the number of stored *ISRNodes*. The BOOL returned from*isEmpty* indicates if no *ISRNodes* are stored. The three main methods are defined as well. Remove and search are defined even twice. Insert acts as expected, data is passed and a boolean indicating whether or not the insertion where successful. An insertion returns FALSE if the value was already present, this is particularly interesting for the set container. *Search* and *searchIt* are basically the same function the reason for two different names is that functions can not be overloaded on return type in *D* , neither C. Both return a link to the searched data or null if the data is not present. *Remove* is the exact opposite to *insert*, passing the data or an *ISRIterator* to it removes the data from the data structure. The returned boolean indicates whether the data was present before removal.

The last type of the ISR is the iterator type *ISRIterator*. As being an abstract class *ISRIterator* already implements three methods. This is because all *opUnary* function are overloaded on their template parameter value. These kinds of function cannot be overloaded in a derivating class. The three methods call abstract functions of an *ISRIterator* class. *ISR* data structures can be navigated with *ISRIterator* in the same way iterator work in the standard template library of C++, with the exception that an iterator knows by them self if it is no longer valid. The method *isValid* returns a boolean that indicates whether an iterator is pointing to a valid entry. [2]

5.1.2. Implementation

In the following sections the different *ISR* implementations are presented. The two tree based types share big parts of the implementation. The only thing unique to both are the insert and remove functionality. A good example for shared code in this case is the tree traversal.

5.1.2.1. Tree based container

5.1.2.1.1. Binary Search Tree

A binary search tree is a data structure that builds a tree by placing new elements as so called leafs. The first element inserted is the so called root node. Every other element is inserted at a position defined by its value. The insertion procedure walks the tree from the root to each leaf. The value of each new element is compared to every element on the insertion path. Should the value be greater the right child of the current node is visited next. If the value is smaller the left child is traversed instead. If any of these children

[2]The standard template library (STL) iterator are used in a manner that they are valid as long as the compare unequal to a special end iterator that is defined by every container type.

are not yet defined the new element becomes that child. On average the complexity for insert, search and remove operation is $\mathcal{O}(n\,log\,n)$, where n is the number of objects in the tree structure. A drawback of binary search trees (bsts) is that their worst case time complexity is $\mathcal{O}(n)$ [Knu98, Wal08a, Wal08b].

5.1.2.1.2. Red-Black trees

Every operation on a red-black tree takes $\mathcal{O}(n\,log\,n)$ time. To achieve this, some constraints are added to the tree. Every node of the tree is colored either red or black. This red-black coloring builds another structures in conjugation with the tree. This structure requires that every black node only has red children and that every red node only has black children, every leaf is colored black and every path to a descended node has the same length[CLRS09, Wal08d].

5.1.2.2. Hashtables

A hash-table allows insertion, removal and searching to run on an average time complexity of $\mathcal{O}(1)$. The idea behind this data structure is to compute a unique key from the given data and use this key as an index to an array.[3] The array does have a fixed size, but the range of the key values is not fixed. This presents a problem, even when considering a perfect hash function, the problem is that the integer value has to be trimmed to the length of array by the euclidean division [Cic80]. The euclidean division will make every hash function imperfect again, this means that key collision can occur. The resolution algorithm implemented in libhurt is to place all colliding keys in a linked list for that particular key. Even though this could lead to a $\mathcal{O}(n)$ runtime complexity tests have shown that this possibility has virtually no influence on the performance of this hash-table. Another way of counteracting the number of key collision is to only fill the array to a certain level. Typically the fill level is between a value of 0.5 to 0.7[CLRS09, p. 253-280]. Libhurt uses a value of 0.7. As shown in table 5.2 the speed of the hash-table is good in comparison to the other ISR containers and the implementation is considerable simpler. Listing 5.2 shows the insertion implementation.

Listing 5.2: Hash-table insert methode

```
static void insert(Node!(T)[] t, size_t hash, Node!(T) node) {
    Node!(T) old = t[hash];
    t[hash] = node;
    t[hash].next = old;
    t[hash].prev = null;
    if(old !is null) {
        old.prev = t[hash];
    }
}

bool insert(T data) {
```

[3]The unique key must be convertible into an integer type to be used as an array index.

```
12      Node!(T) check = this.search(data);
13      if(check !is null) {
14          return false;
15      }
16
17      size_t filllevel = cast(size_t)(this.table.length*0.7);
18      if(this.size + 1 > filllevel) {
19          this.grow();
20      }
21
22      size_t hash = this.hashFunc(data) % table.length;
23      insert(this.table, hash, new Node!(T)(data));
24      this.size++;
25
26      return true;
27  }
```

Lines 12 to 15 check if the data to be inserted already exists. Should that be the case FALSE is returned to indicate the failure of the operation. The next two statements test whether the array needs to be grown. As mentioned earlier the threshold is 0.7. The last three statements in combination with the static insert function implements the actual insertion. In line 20 the index or hash, is computed from the data. The index and a newly constructed node, are passed to the static function.[4] To keep track of the number of saved values the size is incremented and finally a boolean is returned to indicate a successful insertion. The static insert function is, at its core, an insert function for a double linked list, with the exception, that all nodes are inserted at the beginning. New nodes are inserted at the beginning of the list, as shown in line three. Lines four to eight build the link structure of the double linked list. The conditional statement of line six is needed to prevent segmentation faults due to null pointer dereferencing. Inserting takes a total of 45 lines[5], in comparison to 72 lines in a red-black tree. The total number of lines in combination with the complexity of the control flow makes the insertion function of the hash-table easier compared to the tree ISR implementations. [6] [7] The *grow* function is trivial, as it simply iterates over all elements in the hash-table and inserts them into a new array via the static insert function. The other functions defined by the ISR interface are implemented in a similar, simple manner[Wal08c].

[4]The current array is also passed. The name of that array is table. This is done because the static insert function is also used when the hash-table is grown. To prevent code duplication, in these two functions, the insertion logic is implemented statically and all used data is passed explicitly to it.

[5]Both insertion methods take 27 lines plus 18 for the grow function.

[6]The insertion method for the red-black tree is implemented in file `libhurt/hurt/container/rbtree.d`.

[7]For completeness the insertion function of the bst has a total of 31 lines and the insertion function of the binary vector has 16 lines.

5.1.2.3. Binary vector

The binary vector is the most unusual container of the ISR implementations. It uses a vector, a sort function and binary search to implement the ISR interface.

The idea is to have a sorted array and use binary search for finding data. When data is inserted it is simply appended to the back, as a vector does implement this on average in $\mathcal{O}(1)$. After appending the new data to the vector it gets sorted. As *libhurt* has an implementation of *quicksort*, this sorting function is used. Because the array is sorted before every insertion, the time for sorting is not high. [8] To get a unique key for sorting the data items, the same hash function used in the hash-table implementation is used.

Searching data is done, as discussed earlier, by means of the binary search algorithm.

Removing objects is the simplest operation. The index of the value in the vector is obtained trough the search method. This index is passed to the remove method of the vector.

5.1.3. Comparison

The availability of four ISR interface implementations offers the ability to choose the best suited for a specific purpose. To make this choice, one must now the theoretical and practically properties of the implementations. Table 5.1 shows the theoretical runtime complexity. Even so the binary vector works in general without any heap allocation it

Name	**Insertion**		**Searching**		**Removal**	
	average	**worst**	**average**	**worst**	**average**	**worst**
BST	$\mathcal{O}(log\, n)$	$\mathcal{O}(n)$ [a]	$\mathcal{O}(log\, n)$	$\mathcal{O}(n)$ [a]	$\mathcal{O}(log\, n)$	$\mathcal{O}(n)$ [a]
RB tree	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$
Hash-table	$\mathcal{O}(1)$	$\mathcal{O}(n)$ [b]	$\mathcal{O}(1)$	$\mathcal{O}(n)$ [b]	$\mathcal{O}(1)$	$\mathcal{O}(n)$ [b]
Binary vector	$\mathcal{O}(n\, log\, n)$	$\mathcal{O}(n^2)$ [c]	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$	$\mathcal{O}(log\, n)$

[a] In the worst case a binary search tree has the complexity of a linked list. Viewing a graphical representation of such a tree would show a linked list as well.
[b] In the worst case a hash-table has the complexity of a linked list. In this case all entries yield the same hash.
[c] The worst time complexity is $\mathcal{O}(n^2)$, this is due to the worst time complexity of *quicksort* being $\mathcal{O}(n^2)$.

Table 5.1.: Runtime complexity four ISR implementations.

is very slow in comparison to the other ISR implementations as table 5.2 shows. As table 5.2 shows, the hash-table is in general the fastest ISR implementation. The binary vector does fail to compete against the other ISR implementations. Inserting 8192 elements in the binary vector takes over half a minute, which is nearly 1000 times slower than the fastest. The table 5.2 on page 71 shows, that the properties of the different

[8] This is empirically gained data. Theoretical the sorting time can be $\mathcal{O}(n^2)$ as this is the worsted case complexity of quick sort [CLRS09].

Insertion [b,c]				
#	**BST**	**RB Tree**	**Hash**	**Bin-Vec** [a]
64:	0	0	0	1
128:	0	0	0	5
256:	0	0	0	22
512:	0	0	0	101
1024:	0	0	0	444
2048:	1	1	***1***	1946
4096:	2	4	***3***	8499
8192:	5	7	***4***	36510
16384:	11	16	***12***	–
32768:	***27***	39	29	–
65536:	***69***	91	71	–
131072:	***170***	196	238	–
262144:	368	515	***330***	–
524288:	***822***	975	998	–
1048576:	***1883***	2563	2942	–

Searching [b,c]				
#	**BST**	**RB Tree**	**Hash**	**Bin-Vec** [a]
64:	0	0	0	0
128:	0	0	0	0
256:	0	0	0	0
512:	0	0	0	0
1024:	0	0	0	0
2048:	0	0	0	1
4096:	1	1	***0***	2
8192:	3	2	***1***	12
16384:	7	6	***1***	–
32768:	20	13	***3***	–
65536:	46	33	***9***	–
131072:	118	83	***22***	–
262144:	277	223	***48***	–
524288:	665	505	***103***	–
1048576:	1586	1232	***213***	–

Removal [b,c]				
#	**BST**	**RB Tree**	**Hash**	**Bin-Vec** [a]
64:	0	0	0	0
128:	0	0	0	0
256:	0	0	0	0
512:	0	0	0	1
1024:	0	0	0	1
2048:	0	0	0	4
4096:	1	2	0	13
8192:	2	4	***1***	53
16384:	5	9	***1***	–
32768:	13	23	***3***	–
65536:	35	58	***8***	–
131072:	87	147	***19***	–
262144:	207	347	***41***	–
524288:	518	759	***92***	–
1048576:	1288	1784	***194***	–

[a] Because of very long the runtimes the binary vector was only testes up to 8192 elements.
[b] The highlighted number mark the minimum runtime.
[c] Values are in microseconds. The test machine is a Intel Core 2 Duo with 1.3GHz and 4GiB RAM. BST stands for binary search tree, RB Tree for Red-black trees, Hash for Hash-table and Bin-Vec for binary vector.

Table 5.2.: Performance tests of the different *ISR* implementations.

implementation result in no speed difference up to 2048 elements. With the exception of the insertion operation the hash-table is always the fastest container. Insertion is on average fastest with the binary search tree, followed by the hash-table.

5.1.4. Using *ISR* types

Now that we have seen the different incarnations of *ISR* type implementations, we will discuss containers that use these data structures as a basis for their implementation. The presented containers take the standard template library container of C++ as a model.

5.1.4.1. Maps

The map container implements an associative container that maps a value to a key. The map allows the programmer to obtain values by searching for the corresponding keys. Values or keys can not exist on their own inside the map. As the ISR interface only allows single values to be stored a proxy type was introduced to be stored. This so called *MapItem*, takes two values, the first representing the key, the second representing the value. Listing 5.3 shows a shortened implementation of the *MapItem*. [9]

Listing 5.3: MapItem implementation

```
class MapItem(T,S) {
    T key;
    S data;

    override bool opEquals(MapItem!(T,S) f) const {
        return this.key == f.key;
    }

    override int opCmp(MapItem!(T,S) f) const {
        if(this.key > f.key)
            return 1;
        else if(this.key < f.key)
            return -1;
        else
            return 0;
    }

    override hash_t toHash() const {
        return this.key.toHash();
    }
}
```

The important part of this class are the three const methods *opEquals*, *opCmp*, *toHash*. The methods *opEquals* allows for two instances to be compared through the == operator.

[9]The implementation shown omits some details, like necessary casts, the constructors and getter and setter functions. This code was omitted to shorten the listing and make it easier to argument about. The omissions will not change the general usage of the class.

This operator is used in all ISR implementation to check whether a found object is the searched object. Both tree implementations use the *opCmp* function through the <,>, <=, >= operators to figure out where to store a node [dla12e]. All presented functions in the implementation depend on the value of the member *key* for their result. Concluding it can be said that the *MapItem* class acts like a tuple of two values whose position within a data structure is depends only on the first tuple value, the key.

The map implementation is no surprises like the partial implementation in listing 5.4 shows.

Listing 5.4: Map container

```
 1 class Map(T,S) {
 2     private ISR!(MapItem!(T,S)) map;
 3     ISRType type;
 4
 5     this(ISRType type=ISRType.RBTree) {
 6         this.type = type;
 7         this.makeMap();
 8     }
 9
10     void makeMap() {
11         if(this.type == ISRType.RBTree) {
12             this.map = new RBTree!(MapItem!(T,S))();
13         } else if(this.type == ISRType.BinarySearchTree) {
14             this.map = new BinarySearchTree!(MapItem!(T,S))();
15         } else if(...)
16     }
17
18     size_t getSize();
19     size_t isEmpty();
20     void clear();
21     bool contains(T key);
22     MapItem!(T,S) find(T key);
23     bool insert(T key, S data);
24     void remove(T key);
25     void remove(ISRIterator!(MapItem!(T,S)) it, bool dir = true);
26     S[] values();
27     T[] keys();
28     ISRIterator!(MapItem!(T,S)) begin();
29     ISRIterator!(MapItem!(T,S)) end();
30 }
```

The constructor on lines four shows the typical use of the *ISR* implementations. The type of the underlying data structure is passed to the constructor. The value is stored for later use. Then the *makeMap* member is called to construct the underlying data structure. Depending on the *ISRType* a *RBTree*, a *BinarySearchTree* or any other*ISR* implementation is created.

The following three member functions *getSize*, *isEmpty* and *clear* act quite literally. The first returns the number of mappings stored in the container. The second checks whether or not at least one mapping is present. The member *clear* is implemented by calling the *makeMap* member. This can be done, because *D* is garbage collected and therefore no memory leakage can occur.

The member function *contains*, *remove* and *find* share a common problem, the passed parameter is of type *T*, the *key* type. This parameter can therefore not be passed to the *search* or *searchIt* member function of the insert search remove data structures, because they only accept values of type *MapItem!(T,S)* at this point.[10] To overcome this problem a *MapItem* is created whose key is the passed parameter. As described earlier the key is the only important part when searching for a *MapItem*.

Inserting a new value is done by passing a key and value to the *insert* member function. The function constructs a *MapItem* of these two parameters and passes it to the insert function of the ISR implementation accessible through the *map* member. The returned boolean indicates whether a mapping on that key was already present. If a mapping was present FALSE is return and TRUE otherwise.

The next two functions, called *values* and *keys*, return arrays of either all keys or all values. These arrays are created by iterating over the underlying ISR data structure.

The member functions *begin* and *end* return iterators. These iterators point either to the beginning or the end of the maps. It is important to note here, that the ordering of the *MapItems* depend on the used ISR data structure.[11] Tree based *ISRTypes* are sorted the other are not.

5.1.4.2. Sets

Set containers follow a mathematical set in the sense that every element can be present exactly once. Listing 5.5 shows the class and some of its member functions.

Listing 5.5: Set implementation

```
class Set(T) {
    public size_t getSize() const;
    public size_t isEmpty() const;
    public bool contains(T data);
    public bool insert(T data);
    public bool remove(ISRIterator!(T) it);
    public bool remove(T data);
    ISRIterator!(T) begin();
    ISRIterator!(T) end();
    public void clear();
}
```

The constructor and the *makeMap* member function are omitted because they are equal to those of the map implementation. The boolean returned by *contains*, *insert* and *remove* indicates if the operation was successful or the data was contained. The member

[10]Compare to line two of listing 5.4 and listing 5.1.
[11]Compare this to the map and unsorted_map implementations of C++.

function *clear* works the same way it does in the map container.[12] The member function *begin* and *end* return iterators to the beginning and the end of the set, respectively.[12] Again, *getSize*, and *isEmpty* return information about the number of elements stored in the container.

5.1.4.3. MultiMaps

A *multimap* is an associative container allowing a one to many mapping. The implementation follows that of *map* very closely. Again a proxy class with the needed member function is created to place it into the ISR data structures. This class also holds two member variables. The first is the key variable. The difference to the *MapItem*, described in 5.1.4.1 is that it, instead of a simply placing the value variable, has a double linked list to take all the possible values. Whenever a new mapping is created and the key has been placed previous, the new value is inserted into the already existing double linked list. The listing 5.6 shows the available member function of a multimap.

Listing 5.6: Multimap implemenatation

```
class MultiMap(T,S) {
    Map!(T, DLinkedList!(S)) mapping;

    Iterator!(T,S) insert(T key, S value);
    Iterator!(T,S) begin();
    Iterator!(T,S) end();
    Iterator!(T,S) invalidIterator();
    Iterator!(T,S) lower(T key);
    Iterator!(T,S) upper(T key);
    Iterator!(T,S) range(T key);
    bool contains(T key);
    size_t getCountKeys() const ;
    size_t getSize() const ;
    bool isEmpty() const ;
    bool remove(Iterator!(T,S) it);
    DLinkedList!(S) removeRange(T key);
    void clear() ;
}
```

On the first look it can be seen that iterators play a much bigger role than in any other container present this far. The reason for this is to navigate the double linked list efficiently and iterators are the only option. This leads to the question, why a double linked lists is used instead of single linked list or vectors? Single linked list are dismissed because they only allow iteration in one direction and removing elements from them makes the iterator more complex. Vectors are not chosen, because they waste memory.

If our goal where to iterate over the mapped values of a certain key we have two choices. The first choice, closely resembling the C++ approach, would be to get iterators. One through the member functions *lower* and one through the function *upper*. Now we can

[12]Compare to map implementation in section 5.1.4.1.

simply use the iterator obtained through *lower* to iterate the values. When the iterator compares equal to the iterator obtained through *upper* we have traveled all the values. A less powerful, but easier, approach is to use the *range* member function. It returns an iterator that can only travel a specific mapping, starting at the beginning.

The last two member function worth mentioning are the remove functions. The first remove function takes an iterator as parameter. This allows to specify exactly which mapping to remove. If we were to only pass the key as parameter we could not infer which of the possible multiple mappings we want to remove. Earlier it was said that the use of single linked list would make removal operations more difficult. This is because the passed iterator actually must point to one element before the current element to remove it. Doing so would make handling iterators more complex, because they actually don not point to the expected location. The function *removeRange*, as the name suggests, removes all mappings of the given key.

5.1.4.4. MapSets

A *mapset* is a container original to *libhurt*. Similar to a multimap it allows to represent a one to many relations. The unique part about it is that the many part of the relation follows the set restriction: Any element must appear only once. The actual implementation builds on already implemented containers.

Listing 5.7: MapSet implemenatation

```
1 class MapSet(T,S) {
2     // data structure to store the mappings
3     private Map!(T, Set!(S)) map;
4
5     this(ISRType mapType, ISRType setType);
6     public bool insert(T t, S s);
7     public bool remove(T t, S s);
8     public bool contains(T t, S s);
9 }
```

Line 3 of listing 5.7 shows the actual data structure used to store the mapping. As mentioned it uses already present data structures. The line quiet literally says the variable map is a mapping from key type T to a set that stores elements of type S. The operations follow a common pattern. First, the parameter t is used to get the correct set out of the map. Then the parameter s is used to perform the action on the set. The set property of the stored value is particularly useful with the lexer- and parser-generators.[13] A good property of the *mapset*, in comparison to the *multimap*, is that look ups can be

[13]The origin of this data structure is the parser-generator *dalr*. After creating several instances of map set combinations that looked similar to line 3 of listing 5.7, The *mapset* class was created to replace those and to provide a uniform interface.

done in $\mathcal{O}(1)$, compared to $\mathcal{O}(n)$.[14] This runtime behavior can be achieved because the constructor of the *mapsets*, the prototype is shown on line five, allows to set the ISR types used by the map and the set. This allows the programs to create *mapsets* that are, best for its task, at runtime.

5.1.4.5. MultiSets

A *multiset* is, in a mathematical sense, not really a set, as it stores multiple instances of a unique element. Even though it behaves like a set, apart from its insert member function. Insertion also works when the passed element is already present. Should this be the case, a counter is incremented that stores the number of insertions of the element. Removing elements decrements this counter till no more elements are present. It shares the same properties as every other container that is using the ISR data structures.

5.2. Random Access Containers

Arrays are nearly universal data structures that, offer a number of advantages like fast access time, easy iteration and good caching properties. Unfortunately though, they are fixed in size. This means whenever an element is inserted or removed it usually needs to be reallocated, one element bigger or smaller and all elements need to be copied to the new location.[15] The usual way to counteract this is to create an array, that is bigger than needed and carry an integer value that holds the number of placed elements.[16] This path was taken by the vector implementation shown in the next section. The following section discusses the implementation of two random access containers in libhurt.

5.2.1. Vectors

The vector implementation in libhurt follows the above mentioned idea. The constructor creates an array with an defined number of elements. Whenever an element is appended it is stored at the next free position accessible through a stored index. Should the index indicate that there are no more free elements, the array is reallocated with twice its original size, the old entries copied and the new element appended.[17] Listing 5.8 shows the available member function of the vector container.

[14]The access time of $\mathcal{O}(1)$ results in the use of a hashtable, as underlying data structure, for the map as well as the set. Accessing the multimap has the runtime complexity of $\mathcal{O}(n)$ because searching the double linked list inside takes $\mathcal{O}(n)$ time.

[15]Copying the values is not always necessary when we consider the behavior of *realloc* of the C standard library. As with runtime complexities, the worst case is considered.

[16]Compare this to the vector implementation of C++.

[17]The growth rate of two is empirically obtained value.

Listing 5.8: Vector implementation

```
class Vector(T) : Iterable!(T), RandomAccess!(T) {
    Vector!(T) pushFront(T toAdd);
    Vector!(T) pushBack(T toAdd);
    T popBack();
    T popFront();
    T peekBack();
    T peekFront();
    Vector!(T) insert(in size_t idx, T toAdd);
    T remove(in size_t idx);
    T opIndex(size_t idx);
}
```

As the listing shows, the vector implements two interfaces, called *Iterable* and *RandomAccess*. The *Iterable* interfaces defines two function prototypes with the name *opApply*. The *opApply* functions are used by the *D* compiler to allow custom classes to use the *foreach* statement. This interface was designed to easily exchange containers that only need linear traversal. The *RandomAccess* interface, as the name suggests, supplies function definitions that when implemented allow random access through the bracket operator. If a class implements the function *opIndex*, the compiler converts the square bracket operator into a call to *opIndex*. The *opIndex* function is used for implementing random access to the vector and the deque as well. Removing the element with value 4 from

array:	3	1	4	1	5	9	2			
index:							↑			

Table 5.3.: Representation of vector data structure

the vector is shown in 5.3. To remove an element we need to copy the following four elements one position to the left. As the number of copy operations is dependent of the number of elements the runtime complexity is linear. As the operation *popFront* can be understood as a remove of the first element, the same runtime complexity assumed. The member function *pushFront* and insert are the opposite of remove and *popFront*, because up to n elements need to be shifted to the right.

5.2.2. Deques

The goal for a deque implementation was to improve on the vector, so that more operations run in average in $\mathcal{O}(1)$. To achieve this the deque implementation of libhurt is basically a circular buffer implemented on an array. The implementation uses two indices, one marking the beginning and one marking the end. Introducing a second index makes for a more difficult implementation. Table 5.4 shows three possible states of the index. State number one shows the empty state. The low and high index could point to any of the array elements at this state. The second state shows something familiar to the vector. The high index points to an array element with a higher index than the element pointed

State 1

array:										
index:	↓↑									

State 2

array:	3	1	4	1	5	9	2			
index:	↓						↑			

State 3

array:	1	5	9	2				3	1	4
index:				↑				↓		

Table 5.4.: Example states of the deque

to by the low index. Even though the low index points to index zero, this does not mean the low index must be zero. The last state shows the case, when the low index points to an array element located at a higher position than the element pointed to by the high index.[18] By simply decrementing or incrementing the low index we can now *pushFront* or *popFront* elements of the deque in an average runtime of $\mathcal{O}(1)$. If the low index, through *pushFront* operation, reaches the logical position of -1 it is assigned the length of the array -1. It continues at the back so to speak. Unfortunately, inserting and removing from anywhere but the beginning or the end, still has a runtime complexity of $\mathcal{O}(n)$. A disadvantage to the vector is, that the condition on which to grow the array is not as trivial. Before every insertion, *pushFront* or *pushBack* it has to be checked whether or not both indices point to the same element of the array. The calculation for this is more complex, than simply checking if the high index $+1$ equals the array length.[19]

5.2.3. Random access container performance comparison

Table 5.5 shows the runtime complexity of the implemented random access containers in libhurt: vector and deque. As expected the *opIndex* member function of *vector* always runs in $\mathcal{O}(1)$ because the function simply delegates the index to its array and the array access time is constant. So does *pushBack* on average. On average, because in the worst case, a new array needs to allocated and all previously placed elements need to be copied into it. The member function *popBack* always runs in constant time because the implementation is as simple as decrementing the index. The runtime complexity of *popFront*, *pushFront*, insert and remove is $\mathcal{O}(n)$, because all these operations change the number of elements not even the end of the array but somewhere before. That means as much as n elements might need copying one position down.

[18]The up arrow represents the high index and the down arrow the low index.

[19]As the deque implements the same member functions as the vector container, the listing of the class prototype is omitted.

Container	**vector**		**deque**	
Operation, Runtime	**worst**	**average**	**worst**	**average**
pushBack	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
pushFront	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
popBack	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
popFront	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
remove	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
opIndex	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 5.5.: Operation complexity for random access container vector and deque

Table 5.6 on page 5.6 shows empirically gained runtime benchmarks comparing the vector and the deque. Operations that have equal theoretical performance are usually faster in the vector. This is, because the deque has to calculate the actual position, of the index passed to the internal array. The gain of a vector can be seen in the operations *pushBack*, *opIndex* and *popBack*. Operations on the front of the deque are faster than on a vector, this is due to the second index. Inserting and removing is faster in the deque. This is, because of the two indices at maximal half of the nodes need to be copied, the vector on the other hand can only move the index.

5.3. List based containers

List based containers offer the same operations as random access container. The difference is the way they store the data. They store their elements in separate nodes that are connected by references or pointers.

5.3.1. Double linked list

A double linked list is a heap based list implementation where every node has two references: One to the next and one to the previous node. A double linked list container holds references to the first and the last node. As table 5.7 on page 82 shows the runtime complexity of a double linked list is always the fastest.[20]

5.3.2. Single linked list

A single linked list is a heap based list implementation where every node has exactly one references to another node. The referenced node is the following node. The list container holds only one reference pointing to the first element of the list. Table 5.7 shows that a

[20]The fast double linked list is always equally fast, because it is also a double linked list.

pushBack[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	0	0
1024:	0	0
2048:	0	0
4096:	0	0
8192:	**0**	1
16384:	**0**	3
32768:	**2**	4

pushFront[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	2	**0**
1024:	6	**0**
2048:	26	**0**
4096:	104	**0**
8192:	411	**0**
16384:	1556	**0**
32768:	6138	**1**

opIndex[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	0	0
1024:	0	0
2048:	0	0
4096:	0	0
8192:	0	0
16384:	**0**	1
32768:	**1**	3

popBack[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	0	0
1024:	0	0
2048:	0	0
4096:	0	0
8192:	0	0
16384:	0	0
32768:	0	0

popFront[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	2	**0**
1024:	12	**0**
2048:	50	**0**
4096:	200	**0**
8192:	795	**0**
16384:	3097	**1**
32768:	12299	**2**

insert[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	1	**0**
1024:	3	**2**
2048:	13	**12**
4096:	52	**45**
8192:	208	**177**
16384:	803	**673**
32768:	3118	**2573**

remove[a,b]		
#	**vector**	**deque**
64:	0	0
128:	0	0
256:	0	0
512:	0	0
1024:	3	**2**
2048:	12	**10**
4096:	48	**41**
8192:	195	**157**
16384:	771	**633**
32768:	3083	**2533**

[a] The highlighted number mark the minimum runtime.
[b] Values are in microseconds. The test machine is an Intel Core 2 Duo with 1.3GHz and 4GiB RAM.

Table 5.6.: Runtime complexity of random access implementation

Container	**double**		**single**		**fast**	
Operation, Runtime	**worst**	**average**	**worst**	**average**	**worst**	**average**
pushBack	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
pushFront	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
popBack	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
popFront	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
remove	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
get	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 5.7.: Runtime complexity for list based container

single linked has performance problems accessing elements anywhere but the front. This is, because the list has to walked every time at specific position is to accessed.

5.3.2.1. Fast double linked list

The so called fast double linked list tries to combine the speed of array with the flexibility of linked lists. The speed bottleneck of linked lists is that for every insertion a new node needs to be created on the heap and heap allocations are generally slow. The fast linked lists avoids this by creating an array of structs. Instead of references the index of the previous and following node is stored. These indices point to previous and following node. In addition to an index to the first and an index to the last node, the fast double linked list needs another index variable and a stack of indices. The additional index and the stack is used to take track of the free nodes in the array.

5.3.3. List performance comparison

The tables 5.8 and 5.9 show empirically gained data comparing the speed of the different list implementations. If new elements are added at the front or the back the fast double linked list is the fastest. Adding elements to the end is particularly slow with the single linked list, as it has to be traveled completely for every insertion. When elements are added at the front a fast double linked list is also the fastest. The speed advantage comes from the saved node allocation. Inserting anywhere, but the front or back, is fastest with double linked list. Even though the algorithm for iterating to the position is the same and creating a node is faster, the fast double linked list is still slower.[21] The results of opIndex in table 5.9 shows iterating to a given position is faster with the normal double linked lists. A single linked list falls behind again, because in average more elements have to be traversed to reach a given position. This is, because there is no way to traverse backwards in a single linked list. As both double linked lists allow for backwards traversal, the fastest route to a given position can be computed, might it be from the front or the back. This way the maximum longest way to traverse is always

[21]The reason for this could not be ascertained.

pushBack[a,b]			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	1
1024:	0	0	5
2048:	1	**0**	20
4096:	2	**0**	84
8192:	4	**1**	342
16384:	7	**1**	1424
32768:	17	**8**	6182

pushFront			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	0
1024:	0	0	0
2048:	1	**0**	**0**
4096:	2	**0**	1
8192:	4	**0**	2
16384:	8	**1**	4
32768:	22	**2**	11

popBack			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	0
1024:	0	0	4
2048:	0	0	17
4096:	0	0	73
8192:	0	0	297
16384:	**1**	**1**	1247
32768:	**2**	**2**	5435

popFront			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	0
1024:	0	0	0
2048:	1	0	1
4096:	6	0	1
8192:	4	0	2
16384:	9	**2**	5
32768:	18	**4**	21

insert			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	0
1024:	**1**	**1**	2
2048:	6	**5**	12
4096:	**21**	27	61
8192:	**92**	131	269
16384:	**391**	633	1326
32768:	**1754**	2882	8084

remove			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	1	**0**	2
1024:	4	**1**	11
2048:	19	**5**	47
4096:	83	**30**	205
8192:	342	**149**	906
16384:	1515	**653**	4340
32768:	7040	**3239**	25879

[a] The highlighted number marks the minimum runtime.
[b] Values are in microseconds. The test machine is an Intel Core 2 Duo with 1.3GHz and 4GiB RAM.

Table 5.8.: Empirical runtime of list implementation 1/2

opIndex[a,b]			
#	**double**	**fast**	**single**
64:	0	0	0
128:	0	0	0
256:	0	0	0
512:	0	0	0
1024:	**2**	**2**	4
2048:	**7**	9	19
4096:	**33**	37	81
8192:	**131**	150	332
16384:	**528**	603	1368
32768:	**2134**	2419	5981

[a] The highlighted number marks the minimum runtime.
[b] Values are in microseconds. The test machine is an Intel Core 2 Duo with 1.3GHz and 4GiB RAM.

Table 5.9.: Empirical runtime of list implementation 2/2

half of the length of the list. Removing elements from the back is again slowest with a single linked list. Removing elements from the front is fast with all lists. A fast double linked list is again fastest. As removing from anywhere but the front or the back is again fastest with the fast double linked lists, it seems that the garbage collector has a negative performance impact.

5.4. Miscellanies functionality

Even though the biggest use of libhurt are its container, some additional functionality is needed. Some of these are presented in the following sections.

5.4.1. Logger

The implemented logger allows easy standard output message logging. The listing 5.9 shows the available functions. The minimal output of the logger should be the file and the line it was called from, everything after that is subject to normal *printf* string formatting.

Listing 5.9: Logger prototyp

```
void log(string File = __FILE__, int Line = __LINE__)(bool need);
void log(string File = __FILE__, int Line = __LINE__)
    (bool need, string format, lazy ...);
void log(string File = __FILE__, int Line = __LINE__)();
void log(string File = __FILE__, int Line = __LINE_)
    (string format, ...);
```

The logger function make excessive use of *D* features. All functions are template functions. The first two template parameter describe the file and the line the log function is called from. This way the logger function can have a variable amount of parameter for string formatting and still print the file and line without the help of the programmer. To allow conditional logging, all possible log function prototypes are duplicated and the function parameter are preempted with a boolean. If the boolean is true the log message is printed, if it is false the message is not printed. The boolean version in combination with the lazy variable arguments allows for interesting possibilities. The lazy argument property means that all arguments are evaluated as late as possible.

Listing 5.10: Lazy example

```
void lazyFunction(bool cond, lazy float exp) {
    if(cond) {
        arr[99] = cast(int)exp();
    }
}

bool nextFunctionIsValid = checkIfNextFunctionIsValid();
lazyFunction(nextFunctionValid, 10 / 0);
```

The listing 5.10 shows an example on how this becomes useful. The boolean returned by the checkIfNextFunctionValid function indicates if the, for demonstration purposes static, function passed to *lazyFunction* will compute a valid result or not. As the equation will yield a divided by zero exception, it is not valid. If the argument to the *lazyFunction* would not be marked LAZY the exception will occur at calling the function. As the function checks the passed *cond* boolean which indicates if the exp expression is valid or not, the exception can be avoided.

5.4.2. Main argument parser

Listing 5.11 shows a common main function, common here means that important information for the program is passed through a string array called *args*.

Listing 5.11: Main function

```
int main(string[] args) {
    parse_args_variable_for_important_options(args);
    ...
```

In theory parsing these arguments can be arbitrarily complex, in practice it comes down to mostly setting values to variables and setting switches. Another important part about these runtime arguments is documenting their meaning and presenting it.

*D*s standard library phobos offers a parser for parameters, but this module lacks the capability of creating and presenting documentation[com12b].

Over the course of this thesis a number of programs where created that require at lot of parameters to function properly. To easily access their documentation, a new main function argument parser where created and added to libhurt. This module is located in `hurt/util/getopt.d` and is from now on referred to as *getopt*.

Listing 5.12: Getopt function prototype

```
Args setOption(T)(string opShort, string opLong, string desc,
    ref T value, bool last = false, string conflicts = null);
```

The function prototype in listing 5.12 shows the structure of the *getOption* function. Before for we discuss the *getOption* function we need to introduce the basic idea behind the work flow of that function.

The *getopt* module processes arguments in order of their definition not in their occurrence. That means if the *print help* option is first in a long string of options, but it is defined last in the source code, it is evaluated last. This is, because the option are not mapped against the input string, but against a multimap that is created from them. This multimap is constructed by the *getopt* struct constructor received the string array as its single input. As an option is inserted the short and long version is looked up in the multimap. The multimap stores each string of the string array and an index to its position in the array. If an index is found either for the long or the short version of the option, the found index +1 is checked if it can be converted to the type of the parameter value. An exception to this is the type boolean. If the following string does resolve neither to true nor false, true is assigned.

A special case is the options called `-h` and `--help` they are omitted from free use as they force the *getopt* module to print the descriptions of all options after the last option has been passed. To make this work the last parsed option is to be marked. This is done by passing true as last parameter to the *setOption* member function.

An example on how to use the *getopt* parser is given in listing 5.13.

Listing 5.13: Getopt example

```
1  string[] args = ["fileone.d", "filetwo.d", "--bar", "1337", "-z"];
2  Args arguments = Args(args);
3  arguments.setHelpText("a good example to the getopt module");
4  int bar = 0;
5  bool tar = false;
6  arguments.setOption("-b", "--bar", "bar option", bar);
7  arguments.setOption("-z", null, "tar option", tar, true);
8
9  assert(bar == 1337);
10 assert(tar);
```

The arguments normally passed from the operating system are simulated through the string array created on the first line. The string array created by the operating system is of the same structure.[22] Evaluating the string array *args*, by the *getopt* struct *Args*, will make the assertions on line nine and ten valid. Line number three shows how to provide additional information about the program, on top of the option description.

[22]The contents may vary depending on the command issued to start the program.

6. The D lexer generator Dex

The first usage of libhurt was the lexer generator Dex. Dex allows the user to create lexer in a fast and flexibel manner. Using Dex follows a simple work flow.

1. Create the Dex input file
2. Run Dex
3. Compile Dex output into lexer
4. Use the lexer

The described work flow allows the programmer to create lexer without writing hand-written token recognition. The only source code the user has to create, is the actions that should be run if a token has been recognized.

The transition table is created in the way that was presented in chapter 2 on page 27 and following.

6.1. Flow of Execution

The following sections present the work flow of Dex.

6.1.1. Parsing the Input File

Parsing the input file presents a chicken egg problem, as Dex needs to parse an input file that should lead to a parser. To break the cycle the parser for the input file is a simple handwritten parser with integrated token recognition. The parser is implemented as a simple state machine with four states. The grammar of the input file easily fits into type three of the Chomsky hierarchy as the regexs in table 6.1 proofs. For every of those regex

Type	**regex**
Token description	`"[^"]+"`
User action	`{:[.]*:}`
Input error action	`{%[.]*%}`

Table 6.1.: Regex definition of Dex input files

expression a dedicated state exists that adds characters to a buffer as long as the end symbol combination has not been read. The fourth state represent the case when none of the other states are set currently. This state is called *None*. Scanning for the start and end marker of specific action is implement with the help of the `hurt.string.stringutil` module. All possible transitions of these states and the corresponding tokens are shown

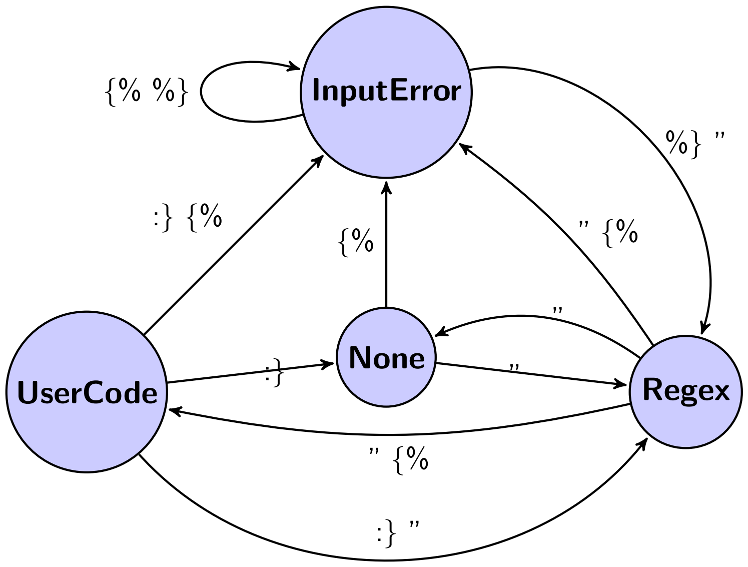


Figure 6.1.: Possible transition of dex input file parser

in figure 6.1. The figure shows that user code can only follow a regex. This allows binding user code to a specific token. Input error code can follow all other three states. The input error code will be placed in a user defined error recovery function. This function is called whenever the lexer algorithm runs into an undefined state. The error code is not specific to a token. The state None acts like a start state. A single regex and an user defined action if present, is stored together in a class called *RegexCode*. All objects of this type are stored in a vector.[1]

6.1.2. Preprocessing the Regex

Now that the regexs are parsed from the file they get fed into the *createNfa* member function. The first thing done here is do prepare the regex for processing.

This requires converting whitespace representations into single char representations. Single character, because the algorithm who is creating the NFA later should not bother with variable length encoding. Therefor the characters of an regex are treated as utf-32 characters. The algorithm could not be simpler, it iterates over the input regex and searches for any occurrences of a backslash. Whenever we find a backslash we look at the next character to determine what to do. The backslash is widely used to escape special character or to mark other actions. Four backslash character are treated as a single backslash. The second case is the `\t` marking a tabular. The tabular character is

[1]Compare with vector description in section 5.2.1 on page 77.

passed as an int value 9 to the output string.[2] The `\n` marks newline character. The int value 10 is used to encode the character in the output value. The backslash `\r` character is converted into the value of 13. The last specially treated character is the " character. This character is problematic as it marks the beginning or the end of regex. But unfortunately this character is also often use to mark the start and the end of string token. To make the character also usable as a input character it has to be prefixed by a backslash. The algorithm eats the backslash and passes the tick to the output string.

The next step is to prepare the unions of the input regex. This is done by replacing the opening and closing square bracket with `\v` and `\f` respectively. Between every member of the union the character the value 6 is placed. Also the plus operator is converted to a regex matching the same string by use of the star operator. When a plus operator is found the output string is searched backwards and the characters are evaluated. Should the character at the position -1 is a simple character it is appended another time to the output string followed by a star operator. Should the character found be a closing square bracket representation, the whole union is appended again, also followed by the star operator.

The last step is to place a special character that marks the concatenation operation.

The returning result is a fully prepared easy to process regular expression. The source code of these procedures is located in the file `dex/dex/strutil.d`.

6.1.3. Building a NFA

After a regular expression has been translated into an algorithm friendly version, we can create a non-deterministic finite state machine from it. As presented in section 2.2.3 on page 29 a regular expression needs to be converted to postfix notation. This is done by the Shunting Yard algorithm exemplary shown in table 2.1 on page 2.1. This in done in the *eval* member function of the regex module located in file `dex/dex/regex.d`. After the shunting yard algorithm returns it calls the *eval* member function.

The eval member function builds the NFA tree similar to the example presented in section 2.2.3. After the NFA tree is created, the last state must be marked with an unique identifier. This identifier is later used to assign the correct user defined action to an accepting state.

All regex are processed and joined to an unique start state by an epsilon transition.

6.1.4. Transforming the NFA to a DFA

Listing 6.1 shows the algorithm part of the member function converting a NFA to a DFA.

[2]Some of the lower numbers are not used neither in ascii nor utf encoding, so they can be misused to allow an easier NFA creation algorithm later on.

Listing 6.1: NFA to DFA algorithm used by dex

```
Set!(State) passAround = new
Set!(State)(theType); while(!unmarkedStates.empty()) { State
processingDFAState = unmarkedStates.popBack();

    foreach(it;this.inputSet) { passAround.clear();
        Set!(State) moveRes = this.move(it,
            processingDFAState.getNFAStates(), passAround);
        Set!(State) epsilonClosureRes = this.epsilonClosure(moveRes)
            ;

        bool found = false;
        State s;
        foreach(jt; this.dfaTable) {
            s = jt;
            if(s.getNFAStates() == epsilonClosureRes) {
                found = true;
                break;
            }
        }

        if(!found) {
            State u = new State(++this.nextStateId,
                epsilonClosureRes);
            unmarkedStates.append(u);
            this.dfaTable.pushBack(u);

            processingDFAState.addTransition(it, u);
        } else {
            processingDFAState.addTransition(it, s);
        }
    }
}
```

As described in listing 2.3 on page 36 we continue as long as at least one state is not marked. We take one of these states and traverse it for every character of the input set. A simple optimization is made to not allocate all data structures new for every pass. The passAround set is this optimisation. The last lines are the subset construction algorithm. The result is a DFA representation of a NFA.

6.1.5. Minimizing the DFA

Now that we have a DFA we need to minimize it. The algorithm dex uses is the algorithm of Hopcroft that is shown in listing 2.4 on page 37. The implementation is so close to the pseudocode that the real source is omitted.

6.1.6. Minimizing a Transition Table

The more interesting minimization algorithm is the algorithm that minimize the transition table. Interesting is that while iterating over a two dimensional vector we keep track of the indices of duplications.

The actual implementation is divided in three parts.

1. Transition table construction
2. Column minimization
3. Row minimization

The result of the DFA minimization is a vector of states.

To make the algorithm for the row and column minimization easier we first need to create a two dimensional vector of it. And two mappings from the state id to the row and the input character to the column. The vector is of type `Vector!(Vector!(int))`. The state id will donate the index for the row, and the sorted input character will provide the index for the column. The mappings are of type `Map!(int,Row)` for the row mapping and of type `Map!(dchar,Column)` for the input character mapping. The types `Row`, and `Column` store the index of the row or column as well as the row and column itself. This redundancy is used to update the state and input mappings in a later step. The construction is straight forward for every state. We create a vector of integer that is added to vector of integer vectors. Than we iterate over all the input characters and place the id of each follow state at a specific position. Should no follow state be defined we place the integer -1 to mark an error. Table 6.2 shows the two mappings and the transition table before the reduction.

The minimization of either a row or a column requires three steps. The first step is to identify equal rows or columns, the second step is to remove all but one duplication and the third is to remap the column and row index. The row reduction algorithm works as

state mapping	
state	**row**
0	0
4	1
5	2
7	3

input mapping	
input	**column**
a	0
b	1
c	2
d	3

transition table				
	0	**1**	**2**	**3**
0	0	0	4	-1
1	-1	-1	-1	5
2	-1	-1	-1	5
3	-1	-1	7	-1

Table 6.2.: Transition table and mappings before reduction

follow. A global index is incremented every round, initially set to zero. In the next step a temporary index is create that is initialized with the value of the global index plus one. This temporary index iterates over all following rows. Every row is compared to the row pointed to by the global index. Should two rows be equal the one pointed to by the temporary index is removed. Two rows are equal if all stored values are equal. Whether or not two rows are equal, the temporary index points to the next row afterwards. If the temporary index reaches the end of the rows the global index is incremented. This process repeats till there are no more rows that need to be comparison.

Now that only unique rows are present, the state row mapping needs to be updated. To achieve this the mappings are iterated and the rows stored in the iterator are compared to the rows in the transition table. Table 6.3 shows the result of the row reductions. As

state mapping	
state	**row**
0	0
4	1
5	1
7	2

input mapping	
input	**column**
a	0
b	1
c	2
d	3

transition table				
	0	**1**	**2**	**3**
0	0	0	4	-1
1	-1	-1	-1	5
2	-1	-1	7	-1

Table 6.3.: Transition table and mappings after row reduction

the tables show a single row gets removed and two state mapping indices changed. The changed row indices are that of state five and state seven. The removed row had the index two.

Reducing the columns works similar to the rows reduction. But before the columns can be reduced copies of them have to be created and stored in the input mappings, to allow remapping of the indices at a later state, equal to that of the remapping of the rows indices. The resulting transition table and mappings are shown in table 6.4. This

state mapping	
state	**row**
0	0
4	1
5	1
7	2

input mapping	
input	**column**
a	0
b	0
c	1
d	2

transition table			
	0	**1**	**2**
0	0	4	-1
1	-1	-1	5
2	-1	7	-1

Table 6.4.: Final transition table and mappings

transition table is stored to run the lexer algorithm later on.

6.1.7. Input Mapping Optimization

One last optimization is done at this stage. The input characters can not be used as indices to the transition table, because they don not align to zero and more importantly, are not necessarily continues. Continues in this case means that, when the input characters are considered to be integer and for every but the last input character there would be a character that is reachable through a $valueOfChar + 1$ calculation. Another reason for the storing the input character mapping in the following data structure is presented later. The first step is to store the character column mappings in a range. The ranges are constructed by iterating over the input characters in a sorted manner. Internally the input characters are stored as `dchar` the *D* equivalent to utf-32 characters. These types take four bytes memory, the size of a integer.[3] These `dchar`s can be sorted simply by passing them to a sort function that evaluates them as integers. The next step is to either append a character to a range or to create a new range with it. A character is appended to a range if they point to the same table column. As ranges are build by iterating over sorted input characters, the ranges will be sorted as well. Being sorted manner allows to search for entries using binary search. The other reason for storing the character column mapping in the semantic value of the characters. In a normal programming language, in this case *D* , not all characters have equal value with view on the tokens. Only characters that were already present in the ascii encoding are used to define keywords. Identifier and string literals on they other hand can be made of all utf characters [dla12d]. Because of this, it can be deduced that these characters will share the same column in the transition table in most cases and therefore can be stored as a range. A range stores the highest and lowest character to mark the beginning and end of a range. A simple calculation will show the usefulness of ranges. Currently 110181 character are defined in utf-8 [Con12]. Considering we ignore the printable ascii character we still have 110087 mappings left. As simple mapping consist of one `dchar` with a size of four bytes and one `size_t` with a size of four bytes as well. With a simple mapping we have to store $110087 * 8 = 880696$ bytes or 0.83 MiB. Storing this in a range requires 1140 bytes.[4] A range consist of two utf-32 characters, one `size_t` index and a bool. This optimization allows us to create lexer for Unicode complete languages that need little more memory than a pure ascii language lexer.

In order to get the correct column the lexer now uses binary search to get the character range mapping.

6.1.8. Writing the Transition Table and Other non-static Parts

Writing the actual transition table to a file is an easy task. The string formatter function and the stringbuffer of libhurt are used to create a string of comma separated integer values. The array are enclosed by an opening and closing square bracket. The same goes for the state mapping.

Listing 6.2: Exemplary transition table and state mapping

```
public static immutable(byte[][]) table = [
[  1, 13, 13, 13,  2, -1,  2,  2, 10,  2],
[  4,  4,  4,  4,  2,  5, 11,  2,  2, 12],
...
];

immutable byte[] stateMapping = [
  0,  1,  2,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  3];
```

[3]Normal in this context means the 32 bit int type.
[4]The size_t type is four byte on 32 bit architectures and eight bytes on 64 bit architectures.

Listing 6.2 shows an exemplary transition table and a state mapping.

More complicated is the mapping for the ranges to an array. Instead of simply writing the values to an array, the ranges need to be build through their constructor.

Listing 6.3: Range array

```
immutable(Range!(dchar,size_t)[2]) ra = [Range!(dchar,size_t)('1',1)
    , Range!(dchar,size_t)('2','7',2)];
```

Listing 6.3 shows such an array. Because a range can a have a length of one or more, two different constructors are needed. The listings shows the use of both. The first range has a length of one, with character '1'. The second range spans from character '2' to '7'.

The two last things to store in the output file are the actions the lexer has to run on accepting a token and the switch case statement which holds the information whether or not a state is an accepting state. The switch case also returns the number of the action to run. As the actions are stored as strings in the RegexCode class instances, the strings are copied into a switch statement. Indexed by a so called action number. The action number is obtained by looking at the accepting switch case that returns a positive integer or −1. The id of the current state is used to get the correct accepting action number. The −1 return value indicates that the current state is not an accepting state. The listing 6.4 gives an idea how these switch case statements look in practice.

Listing 6.4: Action ouput

```
public static byte isAcceptingState(stateType state) {
    switch(state) {
        case -1:
            return -1;
        case 0:
            return -1;
        case 1:
            return 4;
        case 2:
            return 4;

        ...

        default:
            assert(false, format("an invalid state with id %d was
                passed",
                 state));
    }
}

public static immutable(string) acceptAction =
`   case 1: {
 printfln("int %s", this.getCurrentLex());
        }
        break;
    case 2: {
```

```
printfln("hex %s", this.getCurrentLex());
         }
         break;
     case 3: {
printfln("bin %s", this.getCurrentLex());
         }
         break;
     case 4: {
printfln("printable %s", this.getCurrentLex());
         }

     ...
‘;
```

The action evaluation works as follows. Considering the lexer finds that the next character would lead to an error, it will pass the current state to the isAcceptingState function. If the returned value is not -1 it will pass that value to a function where the acceptAction string was mixed in. The accpetAction string needs to be mixed in rather than be a sandal function, because this way the actions defined by the user has access to all data structures managed by the lexer.

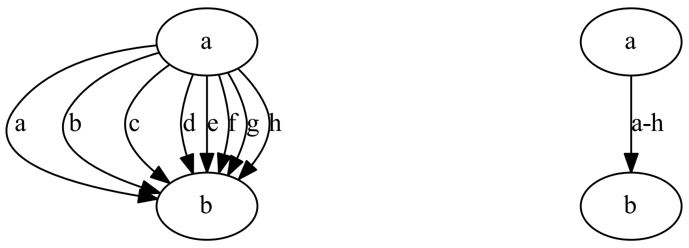
For example the lexer accepts in state number one. The isAcceptingState function returns the integer four. The corresponding action is the print the string that lead to this state.

6.2. Additional Functionality

6.2.1. Creating Graphs

Another important feature of dex is its ability to create graphs of the different stages of lexer construction. Dex does not created these graphs directly, instead it creates textual representations of them and calls a program called dot to create a images of the graph. These graphs helps debugging.

The graph printing functionality of dex changed during development. First the in-memory representation of the different graphs where simply written to files and dot was called to create the image. When the graphs started to get bigger, the time dot took to create these graphics increased quickly. Through testing it turned out the slowdown is caused by the huge number of edges. To counteract that development ranges introduced. These ranges grouped together edges together that lead from one node to another differing only on the edge labeling. For this to work the values of the edge labels need to be continuous. The figures 6.2a and 6.2b show two graphs once without ranges and once with. The introduction of ranges allowed graphs to be visualized again.



(a) Lexer graph without ranges (b) Lexer graph with ranges

Figure 6.2.: Lexer graph with and without use of ranges

6.2.2. Writing a Lexer Template

On top of writing the transition table, dex can write a generic lexer template that can be used to implemented objective specific lexer.

7. The D glr parser generator Dalr

7.1. Introduction

Dalr is a parse table as well as parser generator for LALR(1) and GLR(1) grammars. Dalr was developed to meat the requirements of the compiler dmcd developed in this thesis. At the beginning the implementation followed the LALR(1) parse table construction described in chapter 3 starting on page 41. If LALR(1) would have enough for parsing *D* no additional work would have been required. When GLR was required these capabilities were simply added to the LALR parts. This was possible, because the GLR parse table generation algorithm only differs in the last step from the LALR parse table construction algorithm.

7.2. Parse Table Construction

Dalr constructs parse tables for LALR(1) as well as GLR grammars. Allowing both makes it possible for the programmer to use the appropriate parser for the given task. A LALR parser for more restrictive languages and a GLR for languages that require more task.

Lalr(1) Parse Table Dalr generates LALR(1) parse tables that follow yacc closely. Shift reduce conflicts are solved in favor of the shift operation. Precedence can be defined for symbols as well as production. User defined actions can be inserted after any part of the right hand side of the production. Dalr differs from yacc in that the stack items are defined to be tokens that come from dex.[1]

Glr(1) Parse Table The GLR parse table generator only differs in the construction of the final transition table. GLR parse tables can store more than one action for every given state.

7.3. Parser Templates

Dalr can not only create parse tables it can also create parser templates. In parallel as in the parse table construction, Dalr can construct parser templates for LALR(1)- as well as GLR(1)-parser.

[1]Every lexer using the same token struct can be used with Dalr.

Lalr(1) Parser The LALR(1) parser template only implements the basic LALR parser algorithm. The error recovery functionally is reduced to printing the current state and the lookahead token. No functionality the for abstract syntax tree (ast) construction is implemented not even for the simple parse tree construction.

Glr(1) Parser The template for the GLR(1) parser follows the LALR(1) template in not defining any functionality other than simply parsing. The merge function is defined to always select the first valid parse.

7.4. Implementation of the Parse Table Generator

7.4.1. Parse Table Construction

The parse table construction. The process of creating a parse table will be discussed in the following sections. The construction which follows the LALR parse table construction described in section 3.3 starting on page 47 is as follows:

1. Parsing the input file
2. Construction the itemsets
3. Construction of the extended grammar
4. First set construction
5. Follow set construction
6. Final table construction

Each step is implemented in a specific function or module. All steps are presented in the following sections.

7.4.1.1. Input File Language

The parser generator will output a parse table for a context free grammar. The input language must therefore be able to represent any context free grammar. Tools like yacc or bison use languages for their input files are that close to the backus-naur form. The grammar of the input language can be represented as a Backus-naur form (bnf). Figure 7.1 on page 108 shows the bnf for input language. A rule consists of the left hand side, a separator and a right hand side.[2][3] The right hand side consists of one or more ruleparts separated by whitespace. A rulepart again expands to a string followed by an optional action. The action is a *D* code block enclosed by two special tokens.

[2]In the bnf rule the left hand side is simply a string token.

[3]RHS stands for right hand side.

7.4.1.2. Input File Parsing

The parser for the input file is implemented as a recursive decent parser. Similar to Dex the input file reader of Dalr returns class instances that contain the production combined with the action. The precedence symbol are directly stored in a MapSet.

7.4.1.3. Input File Validation

More interesting than the parsing of the input file are the automatic checks that are run afterwards. These tests were introduced because large input files are error prone to typing errors. These typing errors can lead to parse errors that are difficult to track down.

Typing errors are same checked the way spell checkers work. Every token string is permuted in different ways and checked against a dictionary containing all token strings. Should a permutation match a dictionary entry, it is reported.

Another kind of errors are that subtrees of productions are not reached. This can happen either through typing errors or through simply forgetting productions that needs these productions.

To check for unreached subtrees, a grammar tree is build. Before the tree is created all tokens that appear on the left hand side of a production are placed in a set. Whenever a subtree is created and joined to the grammar tree, the left hand side symbol of the production is removed from the set. Every symbol present in this set, after the tree is completely constructed, is never reached. The remaining token names are printed to the user.

The last type of error detected by *Dalr*, are errors within the user defined action. This error checking breaks the boundary between the parser generator and the abstract syntax tree build methods of the compiler. Even though this can be considered bad practice it has shown its value in the creation of the parser for dmcd. The methods for the ast generation use negative indices to get tokens from the token stack to construct the ast nodes. This works, because instead of a regular stack the *deque* of libhurt is used and the *deque* allows negative indexing. The problem arises if the indices are bigger than the number of tokens of the right hand side of the production the ast build method is run for. Listing 7.1 shows a single production and user defined action taken directly from the used grammar in dmcd.

Listing 7.1: Erronous grammar production

```
LabeledStatement := identifier colon NoScopeStatement;
        {: ret = buildTreeWithLoc(termLabeledStatement, [-4,-1],
            actionNum, this.tokenStack[-2].getLoc()); :}
```

These productions shows the instantiation of a *LabeledStatement*. The array of two negative number, that is passed as the second argument to the buildTreeWithLoc function in the user action, contains an error. The first index −4 points one element to far, it should say −3. The −3 would get the identifier of the *LabeledStatement*. The −4 gets an undefined token. To avoid these kinds of bugs Dalr checks indices to be equal or less than the number of tokens on the right hand side of a production. Dalr would warn the user that in this production an used index is out of bound. It also checks that the index that is used to obtain the token that builds the root of the created subtree fulfills the same requirements. The index -2 in this user action is good, as it is smaller than the number of tokens on the right hand side.

The last check is that the first argument to the*buildTreeWithLoc* function is the name of a left hand side token prepended with the word term.

7.4.1.4. Grammar Rule Preprocessing

After the grammar has been read from the input file, the parse table is constructed. Before we begin constructing the parse table the grammar rules are converted in a more usable format. To this point they are strings separated by blanks and a special assign symbol.

For this purpose every unique token gets assigned an integer value. A rule is stored as a deque of integer. Additionally the type of the token is stored. Type in this case means whether they are terminals or non-terminals.

7.4.1.5. Itemset Construction

The first step is, as described in section 3 on page 41, to construct the itemsets. Dalr implements the itemset construction as follows. The first itemset is constructed from dedicated start production. To store itemsets in an efficient way an abstraction for a production with a dot is introduced. This abstraction is a class called *Item*. The name originates from the idea that many items form an itemset. The *Item* class has two integer members. The first is the index of the productions in the deque of production that form up the language. The second integer indicates the position of the dot within the rule. The method constructing the itemsets has a stack that holds the unprocessed itemsets. For every item it is first checked if an itemset was already constructed. If that is the case this itemsets is referenced. If that is not the case a new itemset is constructed and pushed on the stack. To construct a new itemset the items, that have the dot in front of the specific token, are copied and the dots are advanced to the next position.

The itemsets have maps that link them together. The keys of these maps are the tokens that lead to the construction of the following itemset.[4]

7.4.1.6. Extended Grammar Construction

As discussed in section 3.3.3 on 52 the next step is to construct the extended grammar. The construction is split in two parts. The basic algorithm, located in `dalr/productionmanager.d`, does housekeeping and starts the recursive calls to the itemsets. The second part is

[4]The mappings use the map container implemented in libhurt.

the recursive construction of the extended grammar rules by the itemsets, located in `dalr/itemset.d`.

Listing 7.2: Extended Grammar construction part of productionmanager

```
Deque!(Deque!(int)) extendedGrammer = new Deque!(Deque!(int))(
    this.itemSets.getSize()*2);
Iterator!(ItemSet) iSetIt = this.itemSets.begin();
for(size_t jdx = 0; iSetIt.isValid(); iSetIt++, jdx++) {
    foreach(size_t idx, Item it; (*iSetIt).getItems()) {
        if(it.getDotPosition() != 1) {
            continue;
        } else {
            Deque!(int) p = this.getProduction(it.getProd());
            Deque!(int) extProd = new Deque!(int)();
            size_t s = (*iSetIt).makeExtendedProduction
                (1, p, extProd);
            (*iSetIt).makeFrontOfExtended(p[0], extProd, false);
            extendedGrammer.pushBack(extProd);
        }
    }
}
```

Listing 7.2 shows the first part. It simply iterates over all itemsets and all their items. Whenever the dot is in front of the first token of the right hand side a recursive call is started to the current itemset. This call has the current dot position, the normal grammar rule and the deque that will store the result as arguments. The member function of the called itemset has the name *makeExtendedProduction*. The recursive part is shown in listing 7.3.

Listing 7.3: Extended Grammar construction recursive call

```
size_t makeExtendedProduction(size_t pos, const Deque!(int) prod,
        Deque!(int) extProd) {

    extProd.pushBack(conv!(long,int)(this.id));
    if(pos < prod.getSize()) {
        if(!this.followSets.contains(prod.opIndexConst(pos))) {
            throw new Exception(
                format("no followSet present for input %d",
                    prod.opIndexConst(pos)));
        } else {
            MapItem!(int, ItemSet) next = this.followSets.find(
                prod.opIndexConst(pos));
            extProd.pushBack(prod.opIndexConst(pos));
            next.getData().makeExtendedProduction
                (pos+1, prod, extProd);
        }
    }
    return extProd.getSize();
}
```

The first thing done by the recursive member function of the itemset is to push its id to the resulting deque. Then it checks whether the current token pointed to by the dot has a follow itemset defined in the map of follow itemsets of the current itemset. If that is not the case this is the error case an exception is thrown. If a mapping is present the token is pushed back in the results deque and the following itemset is called. This call is the recursive part, as the same member function is called for the following itemset. This continues until the pos variable is no longer smaller than the length of the production the extended grammar is constructed for. The pos variable stands for the position of the dot.

7.4.1.7. First Set Generation

The first set construction by Dalr differs from the previously presented first set construction algorithm as it does not handle epsilon productions. This is no problem as every grammar containing epsilon transitions can be rewritten as an epsilon transition free grammar. This done simplifies the algorithm and therefore makes the algorithm faster. The simplification allows the algorithm to not care for the second part of the second and third rule.[5] In order to fulfill the set property of the first set a MapSet is used to store the mappings. This way the implementation does not have to check if the token tto been inserted is already present.

7.4.1.8. Follow Set Generation

Not allowing epsilon token in productions has implications for the follow set generation as well. The second part of second rule of the follow set rules can be ignored.[6] Again a MapSet is used to store the set. The implementation itself is not shown as it is a direct implementation of the rules.[7]

Merging the rules follow sets is fast, because the MapSet allows for fast searching of the extended rule items.

7.4.1.9. Final Parse Table Construction

Dalr generates parse table for LALR(1) and GLR(1) parser. The final step of the parse table construction is the only step where the two parse table require different handling.

7.4.1.9.1. Transition Table

The first step is equal for both kinds of parse tables. The transition table is constructed as discussed in section 3.3.2 on page 51. In the implementation the transition table consists of a deque of deques of deques of *FinalItems*. In layman's words, the transition

[5] Compare to first set construction rules in section 3.3.4 on page 52.
[6] Compare to rules in section 3.3.5 on page 53.
[7] The implementation can be found in file `dalr/productionmanager.d`.

table is a three dimensional deque storing *FinalItems*. A *FinalItem* is a class that stores an enum and an integer. The enum indicates the type of action the FinalItem describes and the integer stores the value of the action. The third dimension is introduced to store possible ambiguities as multiple *FinalItems*. Doing this already at the transition table stage allows to share the same data structures and parts of the member functions a longer way for both processes. The algorithm for constructing the transition table simply traverses all itemsets and all terminals and non-terminals. It than checks if the current itemset has a transition for the current token. If it has a transition a shift or goto *FinalItem* is placed depending on the token type. A shift *FinalItem* is placed in case that the token is a terminal. A goto *FinalItem* is placed if the token is a non-terminal.

Placing the reduction actions reveals the difference between Lookahead bottom-up parser(1) and Generalized LR(1) parse tables. If the Lookahead bottom-up parser(1) code path is confronted with an ambiguity it warns and checks if a precedence rule is defined. If no rule is defined and the conflict is a reduce reduce conflict, an error is printed and the program exists unsuccessfully. If the conflict involves a shift operation that operation is chosen.

The Generalized LR path checks if there is a precedence rule defined. If one is, it uses it to resolve the conflict. If no rule is present is simply executes both actions. Only in the later case a warning is printed.

7.4.2. Additional Functionality

Dalr can not only generate parse tables, additional functionality like printing the itemset graph is shipped with it. The three most important features are presented in the following section.

7.4.2.1. Parser Template Printing

Generating a parse table is a very important task when writing a parser. Also important is, to correctly implementing the actual parser algorithm. Dalr therefore allows to print a template for a LALR(1), as well as GLR(1) parser. This templates can be integrated into a compiler or to understand how to use the parse table. To print a template provide the `-d` option followed by the wanted file name to Dalr. To change from a LALR(1) parser to a GLR(1) template pass the `--glr` option to the parser generator. A detailed description about the parser algorithm implemented can be found in section 7.5.1 and 7.5.2.

7.4.2.2. Itemset Printing

An itemset graph allows the programmer to spot errors in the grammar and get a feeling for the tree that is described by the language. Dalr can be told to create such a graph by simply passing the option `-g`, followed by a string. This string will be the name of the file the graph is written to. The dot program, of the graphviz program suite, is again used to layout the graph. Very large languages can not be printed in an acceptable time, due to the performance of dot. To counter act this problem, Dalr can generate dot files for every itemset individually. To print a specific itemset the option `-a` or `--printaround` followed by the id of the itemset can be used. Figure 3.6 on page 50 was created with Dalr, except the layout. The layout was manipulated to fit the page.

7.4.2.3. Log File Printing

Properly the most important feature is the log file writer. It not only prints all warnings and errors, it also prints the itemsets in a textually as well as the follow sets on given input. In addition, the first and follow sets are printed as well as the extended grammar.

7.5. Implementation of the Parser

As actual parsers are a big part of any compiler, the available algorithm are important to present. Like most parser generators Dalr does provide parser templates. As Dalr can generate two different kinds of parse tables, two different parser templates are needed. These will be presented below, with focus on the GLR parser as it is used in dmcd.

7.5.1. Lalr(1) Parser

The following listing 7.4 shows the LALR parser algorithm emitted by Dalr.

Listing 7.4: Dalr lalr(1) parser algorithm implementation

```
public void parse() {
    // we start at state (zero null none 0)
    this.parseStack.pushBack(0);

    TableItem action;
    Token input = this.getToken();

    while(true) {
        action = this.getAction(input);
        if(action.getTyp() == TableType.Accept) {
            this.parseStack.popBack
                (rules[action.getNumber()].length-1);
            this.runAction(action.getNumber());
            break;
        } else if(action.getTyp() == TableType.Error) {
            this.reportError(input);
            assert(false, "ERROR");
        } else if(action.getTyp() == TableType.Shift) {
            this.parseStack.pushBack(action.getNumber());
            this.tokenStack.pushBack(input);
            input = this.getToken();
        } else if(action.getTyp() == TableType.Reduce) {
            // do action
            // pop RHS of Production
            this.parseStack.popBack
                (rules[action.getNumber()].length-1);
            this.parseStack.pushBack(
                this.getGoto(rules[action.getNumber()][0]));

            // tmp token stack stuff
            this.runAction(action.getNumber());
        }
    }
}
```

The parse stack, first used in line three, is a deque of `ints`. The *tokenStack* is again a deque, a deque of tokens. The token type is defined by the lexer generator dex. The TableItem type assigned in line five is the actual type that populates the parse table. It is an struct of an enum, a short integer and an additional single byte for aligning the structure to 32bit.

7.5.2. Glr(1) Parser

The GLR parser is more complex. This is because different parses have to be kept in sync. To make this feasible the parsing is split in two parts. A class that represents a single parse. And the algorithm stepping all instances of this class. The two phases are explained below.

7.5.2.1. A Single Parse

The class that represents a single parse is called *Parse*. The member of the *Parse* class are almost completely the same as those of the LALR parser. This is because every parse by itself is a simple LALR parser and only the combination of multiple parses lead to a GLR capable parser. Instead of retrieving the action as shown on line nine of listing 7.4 on page 104 the action is passed to the parse algorithm. And as it makes a single step with no while loop surrounding the parse logic. The main difference is that the *Parse* class has an member function that returns an array of possible actions on specific input. This is important in the stepper algorithm.

7.5.2.2. Running in Parallel

Running the single parses step by step is the main work done by the GLR parser algorithm.

Listing 7.5: GLR stepper algorithm

```
bool parse() {
    while(!this.parses.isEmpty()) {
        log();
        // for every parse
        for(size_t i = 0; i < this.parses.getSize(); i++) {
            // get all actions
            immutable(TableItem[]) actions = this.parses[i].
                getAction();
            // if there is more than one action we found a conflict
            if(actions.length > 1) {
                for(size_t j = 1; j < actions.length; j++) {
                    Parse tmp = new Parse(this, this.parses[i],
                        this.nextId++);
                    auto rslt = tmp.step(actions, j);
                    if(rslt.first == 1) {
                        this.acceptingParses.pushBack
                            (this.parses[i]);
                        this.toRemove.pushBack
                            (this.parses[i].getId);
                    } else if(rslt.first == -1) {
                        this.toRemove.pushBack
                            (this.parses[i].getId);
                    } else {
                        this.newParses.pushBack(tmp);
                    }
                }
            }

            // after all one action is left
            auto rslt = this.parses[i].step(actions, 0);
            if(rslt.first == 1) {
                this.acceptingParses.pushBack(this.parses[i]);
                this.toRemove.pushBack(this.parses[i].getId);
            } else if(rslt.first == -1) {
                this.toRemove.pushBack(this.parses[i].getId);
                if(this.newParses.isEmpty() &&
                        this.acceptingParses.isEmpty()) {
                    println("%s", rslt.second);
                }
            }
        }
        // copy all new parses
        while(!this.newParses.isEmpty()) {
            this.parses.pushBack(this.newParses.popBack());
        }

```

```
            this.mergeRun(this.parses);

            this.parses.removeFalse(delegate(Parse a) {
                return this.toRemove.containsNot(a.getId()); });

            this.toRemove.clean();
        }

        // this is necessary because their might be more than
        // one accepting parse
        this.mergeRun(this.acceptingParses);
        return !this.acceptingParses.isEmpty();
}
```

The above listing 7.5 shows the parser stepping method. The deque called parses stores all currently active parses. It is iterated in the first for loop. Whenever a parse returns more than one possible action it is copied as often as needed to run all actions.[8] All parses are than stepped. The step member function implemented in the *Parse* class returns a value indicating the new properties of the parse. There are three possibilities.

1. The parse accepts. That means this particularly parse accepts the input. The parse is than stored in a deque of accepting parses, because one parse has to be selected as the final parse.
2. The parse yields an error. Unless this parse is the last parse no error message will be printed. This is because no error occurs till the last parse failed.
3. The third possible outcomes, is a successful action.

Parses that return an error are not stored in the list of parses but get removed if they were stored before.

Another important part is the merge function, that checks if there are parses that are equal. This function is run after all parses are stepped. Two parses can be merged if their parse stacks and lookahead tokens are equal. The two parses are passed to a user defined, selection function.

After all remaining parses have accepted the selection function is run again to selects the final result of the overall parsing process.

[8]The parse is actually copied one time less, as the original is still present.

```
⟨Dlr⟩          ::= ⟨Rules⟩
                 |  ⟨Precs⟩ ⟨Rules⟩

⟨Precs⟩        ::= ⟨Precs⟩ ⟨Prec⟩
                 |  ⟨Prec⟩

⟨Prec⟩         ::= '%left' ⟨TokenNames⟩
                 |  '%right' ⟨TokenNames⟩
                 |  '%nonassoc' ⟨TokenNames⟩

⟨TokenNames⟩   ::= ⟨TokenNames⟩ ⟨string⟩
                 |  ⟨string⟩

⟨Rules⟩        ::= ⟨Rules⟩ ⟨Rule⟩
                 |  ⟨Rule⟩

⟨Rule⟩         ::= ⟨string⟩ ':=' ⟨RHS⟩
                 |  ⟨string⟩ ':=' ⟨RHS⟩ '|' ⟨RHS⟩

⟨RHS⟩          ::= ⟨RuleParts⟩ ';'

⟨RuleParts⟩    ::= ⟨RuleParts⟩ 'whitespace ' ⟨RulePart⟩
                 |  ⟨RulePart⟩

⟨RulePart⟩     ::= ⟨RulePart⟩ ⟨Action⟩
                 |  ⟨string⟩

⟨Action⟩       ::= '{:' ⟨Any_valid_D_code⟩ ':}'
```

Figure 7.1.: BNF for Dalr input language

8. The D compiler DMCD

8.1. Introduction

DMCD is the name of the compiler created for this thesis. It uses the lexer generator dex and the parser generator Dalr for lexer respectively parser generation. The way they are combined is unique to dmcd. This will be discussed later in detail. To facilitate the new concepts dmcd is split into two parts: A frontend and a daemon. The frontend takes compile directives from a shell and passes them to the daemon. The daemon runs in the background till it is advised to quit. The reason for the background daemon is to provide a consisted memory for the cache and to aid in the distribution of the work among a network.[1]

8.1.1. The Lexer

Dex is used to generate the lexer table for dmcd. The lexer template generated by dex is also used by dmcd. The input file for dex contains 192 regular expression. Only 15 of them use for anything else than concatenations. The resulting minimized DFA has 569 nodes.

8.1.2. The Parser

The grammar for *D* is very complex, the input file is over 3000 lines long and has 813 rules. Dalr generates a parse table that is written to the file `dmcd/src/parsetable.d`. The parser that resits in the file `dmcd/src/parser.d` was devised from a GLR parser template that Dalr generated. The actions assigned to the productions build the ast and the symbol table.

8.2. Intermediate Representation

The result of different compiler stages are an important step in deriving an executable. The symbol table and the abstract syntax tree are the most prominent structures.

[1]The implementation used for testing, that is available on the dvd coming with this, is not split in two parts, because the caching and the distributing is not implemented, but discussed theoretically.

8.2.1. Symbol Table

A symbol table is created as a map. This map uses the binary vector as an underlying data structure. This way symbol table entries get stored in a cache friendly manner. The symbol table entries contain the position of the symbol as well as access rights, derived from the access modifier keywords and the symbol itself.

8.2.2. Abstract-Syntax-Tree

As the results of the different compiler stages are supposed to be cached they need to be laid out in a cache friendly matter. Unfortunately, trees are usually stored on the heap as structures linked by pointers. As a cache is in most cases just a continuous fixed size block of memory, trees are not best to store it.

Two ways of storing a tree come to mind. The first is to convert the tree into an encoding like json or XML. The second possibility is to build the tree of STRUCT store them within an array and link them by their array indices. This way the tree is flattened into a cache friendly form.

The first solution has a significant drawback. Whenever a value is to be read from the cache, it needs to be parsed and converted into a tree on the heap. Storing a tree requires the tree to be walked and encoded in the used format. These conversations will take time at least linear to the size of the tree as they require a parser.

The second method will be more suitable for caching, because converting the returned data from the cache to something usable is as easy as casting to the specific array type. Storing the array, building the tree, into the cache is easy as well. The array only needs to be written byte by byte to the allocated memory in the cache.

8.2.2.1. Flattening the Tree

Mapping a tree to an array is easy as long as the number of children for every node is known and constant.[2] Unfortunately an ast does not provide this property. Considering the exemplary tree in figure 8.1, that represents the listing 8.1, we see that every node has as little as zero or as much as three children.

Listing 8.1: Simple main function

```
int main() {
    return 1;
}
```

To store a tree with a variable number of children another array has to be created that stores the indices of the children. The nodes of the tree only stores the index of index of the first child and the total number of children.[3] This way all members of the struct

[2]Compare to binary trees or heaps.

[3]This can be compared the pointer to pointer idiom of C.

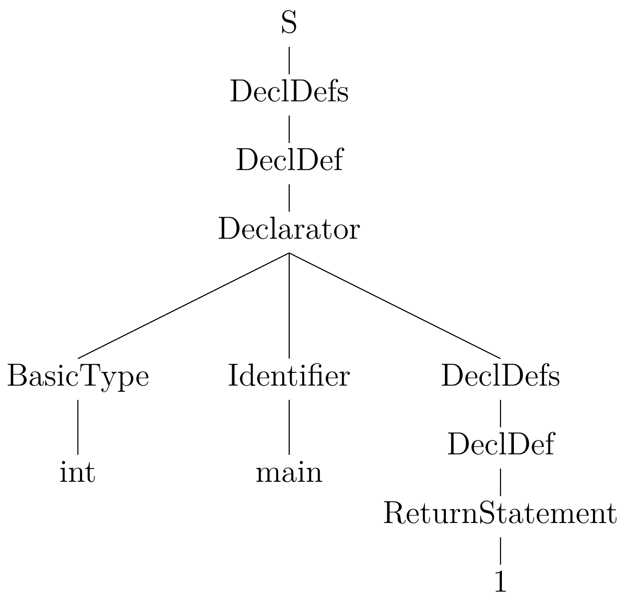


Figure 8.1.: Ast of simple main function

can be simple values. And because the nodes are build of structs they can be placed continuously within a chunk of memory. Tables 8.1 and 8.2 show the two arrays building the tree.[4]

The index of the first child in table 8.1 is only valid if the number of children is at least one. Otherwise the value is invalid and can be ignored. The structs of tree nodes as well as the children indices are actually stored in a deque. The reason for this is that in a deque new nodes to be appended fast and easily.[5]

Listing 8.2 shows the *ASTNode* struct without it member function.

Listing 8.2: ASTNode from dmcd

```
struct ASTNode {
    private Token token;
    private int typ;
    private Pair!(size_t,ubyte) childs;

    ...
}
```

[4]The array is filled, as it where, by a top down parser. Dmcd uses a bottom up parser and the resulting arrays would show a significant different layout of the single nodes. This is however not important, because the task of the example is to show how the tree can be mapped on two arrays.

[5]An array would require to be grown on insertion or a separate last index to be used. A deque abstracts this logic.

Array of tree nodes												
Name of node	S	DeclDefs	DeclDef	Declarator	BasicType	int	Identifier	main	DeclDefs	DeclDef	ReturnStatement	1
Index in array	0	1	2	3	4	5	6	7	8	9	10	11
Number of children	1	1	1	3	1	0	1	0	1	1	1	0
Index of first child	0	1	2	3	6	0	7	0	8	9	10	0

Table 8.1.: Tree as array representation

Children index array											
Index	0	1	2	3	4	5	6	7	8	9	10
Child index	1	2	3	4	6	8	5	7	9	10	11

Table 8.2.: Children index array

It can be seen that the struct holds except for the token no more values than used in the table 8.1.[6] The two arrays are simply two deques one of type *ASTNode* and the other of type *size_t*.

8.2.2.2. Restructuring the Tree

One problem is left with the flattened tree. Removing or inserting nodes other than behind the last leaf is difficult. This is because for every but the last added node every child index, located in the children array, is not at the end. Forcefully inserting a new index anywhere but the end, in either array, will render the indices to the right of that index invalid. They are invalid stored indices rely on constant positions in the both arrays.

A solution to this problem is to basically implement heap allocation. This is because the solution to this problem, even if not implemented yet, is very close to what heap allocation algorithms do. Whenever a new child is to be inserted, anywhere but the end, the whole sub array containing the original children indices are copied to the end and the new child index is appended. And the children index pair of the nodes father is updated. By doing this a number of indices become unused somewhere in the middle of the children index array. This can be understood as fragmentation. When new ast nodes are created these fragments need to be considered first for possibly taking the children indices. Selecting the fragmented areas can be done by well understood selection algorithms like first- or best-fit [Bre89, Kno65].

[6]The complete struct can be found in `dmcd/src/ast.d`.

Number of nodes	Struct based	Class based
7	0.748	**0.427**
15	**0.167**	0.869
31	0.263	**0.225**
63	0.819	**0.154**
127	**0.167**	0.297
255	**0.335**	0.590
511	0.697	**0.117**
1023	**0.147**	0.239
2047	**0.164**	0.184
4095	0.415	**0.115**
8191	0.824	**0.238**
16383	1.559	**0.481**
32767	2.639	**1.569**
65536	5.219	**2.575**
131072	10.354	**6.124**
262144	20.693	**11.14**
524288	41.497	**20.86**

Table 8.3.: Tree building speed comparison.

8.2.2.3. Speed comparison

Profiling the performance is an important part. Building the ast structure in an array needs to be compared to the typical, heap structured, approach. Creating the ast building facilities for both approaches is not practical because of the amount of required work. To get an idea of the performance a syntactic test was created. Table 8.3 on page 113 shows the result of this syntactic test. The boldfaced entries mark the fastest entries. The test builds a fully populated binary tree. The times are measured in seconds. The size of the nodes in the created tree have the size of the ast nodes used by dmcd. As the table shows the values dependent on the speed of the heap allocation algorithm. The deque, which is used to store the ast nodes, uses an array internally and the allocation algorithms seams to have problems finding large continues blocks. The struct based heap does not appear much slower up to 2^{15} elements.[7]

8.3. Distribution

Making dmcd a distributed compiler means to have compile jobs distributed in a network. The idea behind this is that under a normal workload, development machines speed most of their time in an Integrated development environment (IDE) idling. The rest of the performance, of many core architectures, could be used to compile source files from

[7]The test can be found in the file `libhurt/tests/structtreetest.d`.

clients coming from somewhere in the network. This way the individual compile jobs would take less time for every developer in the network.

This feature was not implemented for time reason, but the implementation would have looked as follows. Every developer would start a daemon and join the compiler network in the local area network.[8] From this point onward he would accept compile jobs as well as give them.

Compile jobs would have been distributed on a per file bases. This means that a daemon would send a file to one or more other daemons in the network.[9] At first the asked daemon would probe into their caches to see whether there is already a result. The caching is discussed in section 8.5 on page 120 in detail. If no result are present the file would be compiled. After the file has been parsed but before the semantic analysis, the import statements need to be resolved. To do this the compiling daemon would ask the client for the hashes of the import files. After he received them, he would again probe his cache. Whenever a result for a file is not present he would ask the client for it. The client will than send the file. This is done recursively till all dependencies are fulfilled. After the file is compiled the resulting intermediate code is returned to the client.

To understand at what point distributing work in the network becomes reasonable, several things have to be considered. The first thing is the question how fast files can be read from disk and send to other machines. To answer this question we need to consider the speed of the harddisk or ssd and the speed of the ethernet. The current sata 3.0 standard aims at a transfer rate of $6Gbis/s$ [saio12]. Considering the local area network (lan) is using GBit ethernet. The network is the limiting factor. The biggest file in the project is `dmcd/src/parsetable.d` with a size of $3.5MiB$, which is generated by Dalr, transferring the file would take about 0.03 seconds. The official *D* compiler takes a little over 7.6 seconds to compile the file.[10] Assuming returning the result would again take 0.03 seconds and compile jobs are run in sequence on the client distributing the work would result in a speedup from the first file on. Another advantage is that multiple machines result in multiple caches to draw from.

8.4. Multithreading

Multicore CPUs are present in almost any current desktop pc. Even though, compilers are, to this day, single threaded. Dmcd can use multiply CPU cores and divides the work among them. How dmcd does this is explained in the following sections.

[8]The compiler network is meant to be restricted to a local area network to keep the management overhead limited.

[9]The number of assigned daemons depends on a strategy that tries to give good availability. The complexity of this strategy can be arbitrary as many design goals are to brought in for consideration. These goals include availability, answer time, network traffic and number of daemons.

[10]The official *D* compiler was used because it implements the full semantic analysis as well as a backend to create assembler code.

8.4.1. Lexer Parser Communication

Normally a lexer is driven by a parser. The parser asks the lexer for a new token and the lexer reads as many characters from the file as needed to construct one.

Dmcd works completely different. The parser and the lexer life in different threads. The lexer reads the file from top to bottom into buffer. The parser takes the token out of the buffer. The only time the lexer and the parser interact is when synchronizing the push and pop operations on the buffer. Below both ways through the synchronization are described. The synchronization equals the producer consumer problem. The lexer, the producer, generates one token after the other and places them in a buffer. The consumer, the parser, copies the tokens out of the buffer. The parser does not copy the tokens one at a time. It copies a specific amount of tokens, if less are present the parser waits on a semaphore.

The idea behind this is that the lexer has to do IO and IO is generally slow. Especially, reading from a harddisk can be very slow. But when data is present the lexer should process as many as he can. If the lexer is to read token by token, it could happen that the lexing algorithm or transition table is no longer present in the caches of the CPU or even worse was swapped to harddisk. To avoid this the lexer is placed in a separate thread and reads, greedily, as much of the input as the harddisk offers him. Another consideration is that the parser will take longer to process a token than it takes the create token from the input. If the lexer creates the token in parallel the parser does not has to wait for a new token instead. It simply copies them. And as copying a token, into an already allocated heap memory, is faster than lexing a new token from a file, another speedup occurs.

8.4.1.1. Push back token

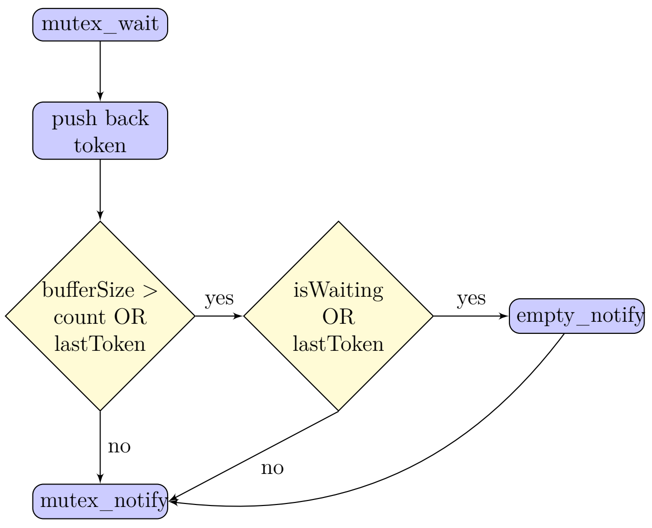


Figure 8.2.: Multithreaded token pushBack

Tokens are pushed to the buffer by a lexer. Figure 8.2 shows the control flow. The first step is to acquire the *mutex* that protects the buffer from race-conditions. The next step is to push the token. If the token marks the end of the file or the buffer is filled to a specific limit another test is run. This test checks if a thread is waiting for tokens. That thread runs the parser. Should it be the case the semaphore is incremented to *deblock* the thread. The last thing to do is to release the *mutex*.

8.4.1.2. Get token

Coping tokens from the buffer is a bit more complex. Figure 8.3 shows the process. Again, the first thing to do is to create mutual exclusion. This is achieved by locking the

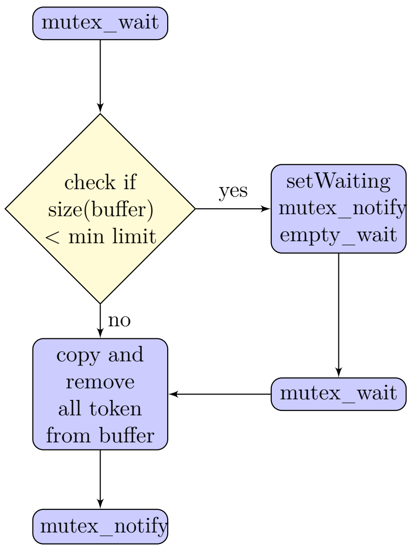
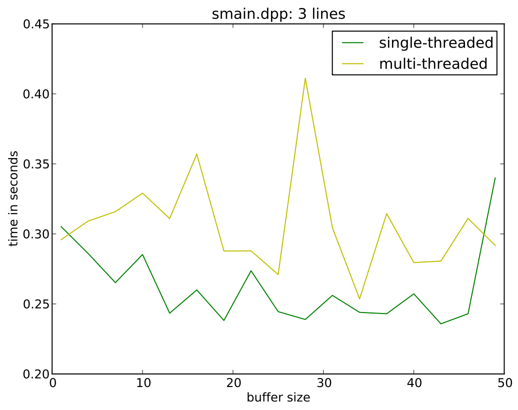

mutex_wait
check if size(buffer) < min limit
yes
setWaiting mutex_notify empty_wait
no
copy and remove all token from buffer
mutex_wait
mutex_notify

Figure 8.3.: Multithreaded token get

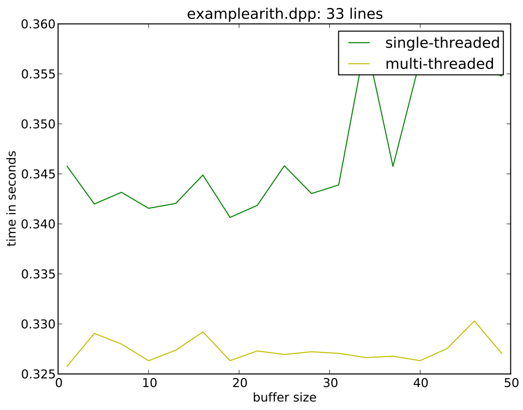
mutex. Then it is checked if there are more tokens present in the buffer than a defined limit. If that is the case these token are copied and removed and the mutex is released. If the buffer holds too few, the parser is marked as waiting, the mutex is released and the parser begins to wait on a semaphore called empty wait. After it is awakened by the lexer, the mutex is acquired again, the token are copied and removed and the mutex is released. After it is awaken by the lexer no further checks on the buffers are needed because of the condition for the lexer leading to the awakening.

8.4.1.3. Benchmark

The figures 8.4, 8.5 and 8.6 on pages 117 and 118 show a benchmark on files of different sizes. The yellow line represent the multi threaded lexer parser combination. The green line represents the single threaded lexer parser combination. The values are arithmetic middle of 100 runs.[11] Expect for the smallest and biggest file, the multi threaded

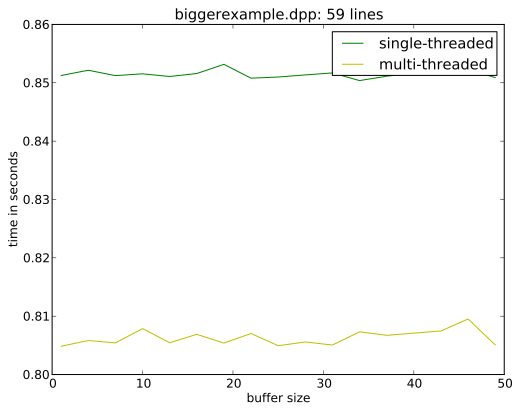


(a) Benchmark for 3 line file

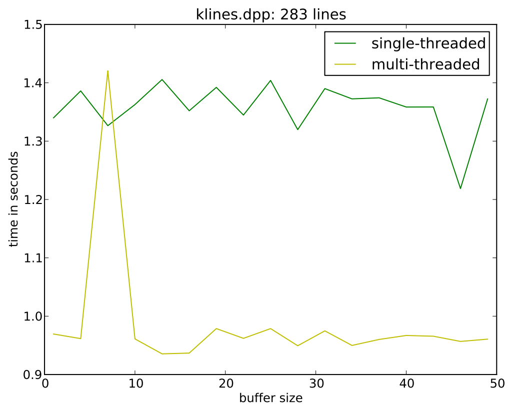


(b) Benchmark for 33 line file

Figure 8.4.: Benchmark multi threading lexing parser 1/3



(a) Benchmark for 59 line file



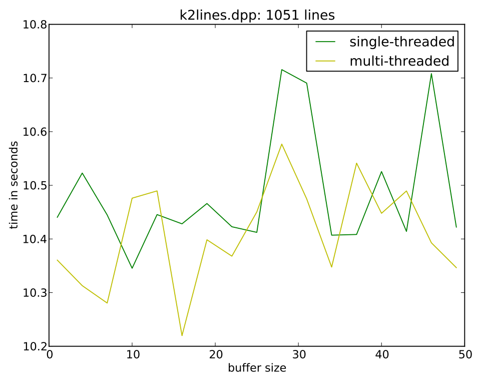
(b) Benchmark for 283 line file

Figure 8.5.: Benchmark multi threading lexing parser 2/3

combination is the fastest. The exception with the smallest file can be explained by the overhead it takes to create the separate threads. The results for the biggest file, displayed in figure 8.6a on 118, are based on the relative low number of samples taken.[12] For the rest of the files the multi threaded lexer parser is faster, usually a few hundreds up to a few tens of seconds.

[11]The buffer size was iterated in steps of three. This was done to reduce the time the test suite takes to complete.

[12]Only 100 samples per test where taken because the test suite took more than 8 hours to complete.



(a) Benchmark for 1051 line file

Figure 8.6.: Benchmark multi threading lexing parser 3/3

8.4.2. Semantic-Analyzer

The output of the parser is a ast and a symbol table. The next step is to check semantic restrictions of the language. These restrictions are for example:

Type checking verifies that all assignment expressions are done with correct types. That means for example that a string is not assigned to a variable of type INT.

Const correctness checks that values or objects marked as CONST are not modified. For example that a const variable is not assigned.

Return statement reachability means that the control flow of a function reaches a return statement. For instance, a function consists of an if statement that holds a return statement, nothing else. In this case it depends on the if condition if the return statement is executed. If the statement is false no return statement is executed leading to undefined behavior.

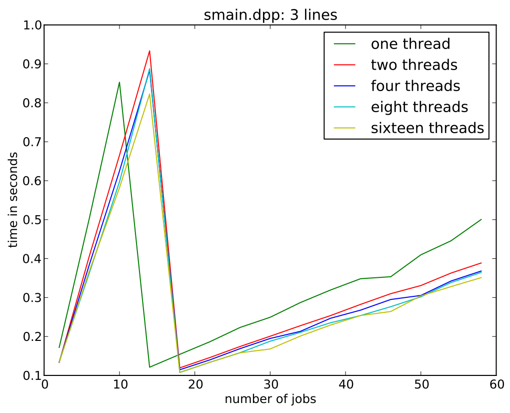
All these verifications are independent of each other.[13] Another observation is that they do not modify the ast nor the symbol table. That means they do not even interact in any way.

8.4.2.1. Benchmark

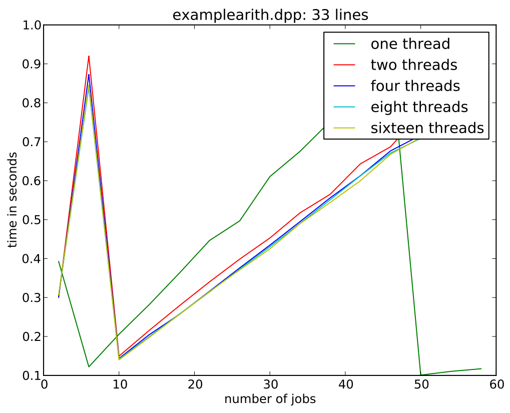
In traditional compilers these checks are run one after another or are integrated in parsing process. The idea for dmcd is to run them in parallel on an abstract syntax tree and a symbol table that is constant.[14] Figure 8.7, 8.8 and 8.9 on page 119 and 120 show the results of an benchmark comparing the speed of the semantic analysis. Several test where run, the number of used threads varied from 1 to 16 and the number of semantic analysis

[13]Considering the mixin statement of *D* this is not completely true, as the statement changes the ast. As the mixin statement was not implemented, no problems occur. To implement the mixin statement, an additional single threaded traversal of the ast, need to be done.

[14]Compare to const-correctness.

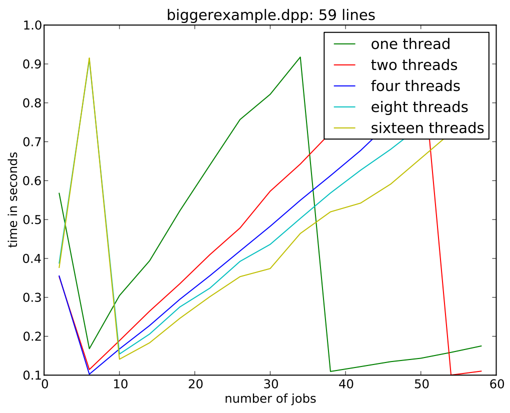


(a) Benchmark for 3 line file

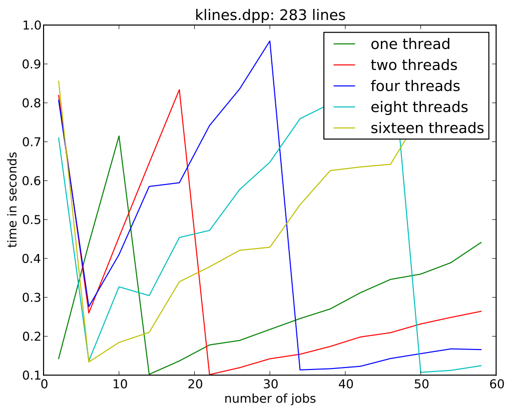


(b) Benchmark for 33 line file

Figure 8.7.: Benchmark multithreading semantic analysis 1/3



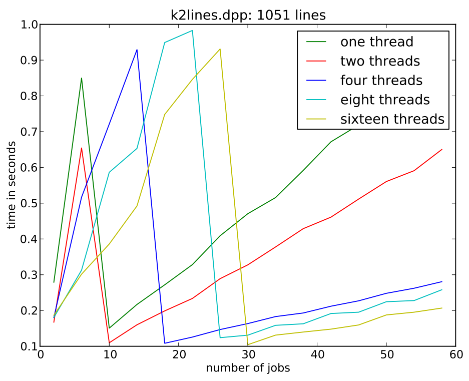
(a) Benchmark for 59 line file



(b) Benchmark for 283 line file

Figure 8.8.: Benchmark multithreading semantic analysis 2/3

from 1 to 60. The tests where run on a quad core CPU. Every of those combinations where run 250 times on all five test files. As the figures show the single threaded variant of the semantic analysis has a so-so performance. Figure 8.9a on page 120 shows that with growing number of jobs, more threads perform better. With eight threads nearly four times as fast. Considering the CPU can run four threads in parallel that is the possible expect speedup. On small files the speedup is smaller. This is due to the bad ratio between the overhead of the thread creation and the runtime of the thread itself.



(a) Benchmark for 1051 line file

Figure 8.9.: Benchmark multithreading semantic analysis 3/3

8.5. Caching

Caching results of the compile process builds on the idea that many compile units are code dependent or are used in many places. This assumption can be made because most build management tool like for example *make*, *SConstruct* or *ant* have special features to build and resolve dependency graphs.

For example libhurt consists of 98 files, on average every of these files depend on ≈ 57 files. That means that after building only one files the first cache hits likely will occur. Figure 8.10 shows the dependencies of the files of the parser generator Dalr. The gray nodes have $0 - 1$ files depending on them. The blue files have $2 - 3$, the green nodes with $6 - 7$ files depending on them. The yellow are imported by $8 - 9$ files and the red by $10 - 11$. This, real life, example shows that even for small programs caching could reduce the number of files to read from the harddisk dramatically.

8.5.1. Cached Data

The question occurs what to cache? It makes sense to cache everything from the source code files to intermediate code. The reason for that is simple. The source could can be required to be send offside. The tokens might be needed if the file is to be parsed again. The abstract syntax tree might be needed if a whole file is mixed in. The symbol table is needed if a module (file) is imported and the intermediate code is needed when the executable is rebuilt.

8.5.2. Performance

To understand the value of a cache, the way the cache works needs to be understood first. Libhurt brings a least recently used cache implementation. The properties are that searching an entry is as fast as the map structure used. In general a hash-table is used,

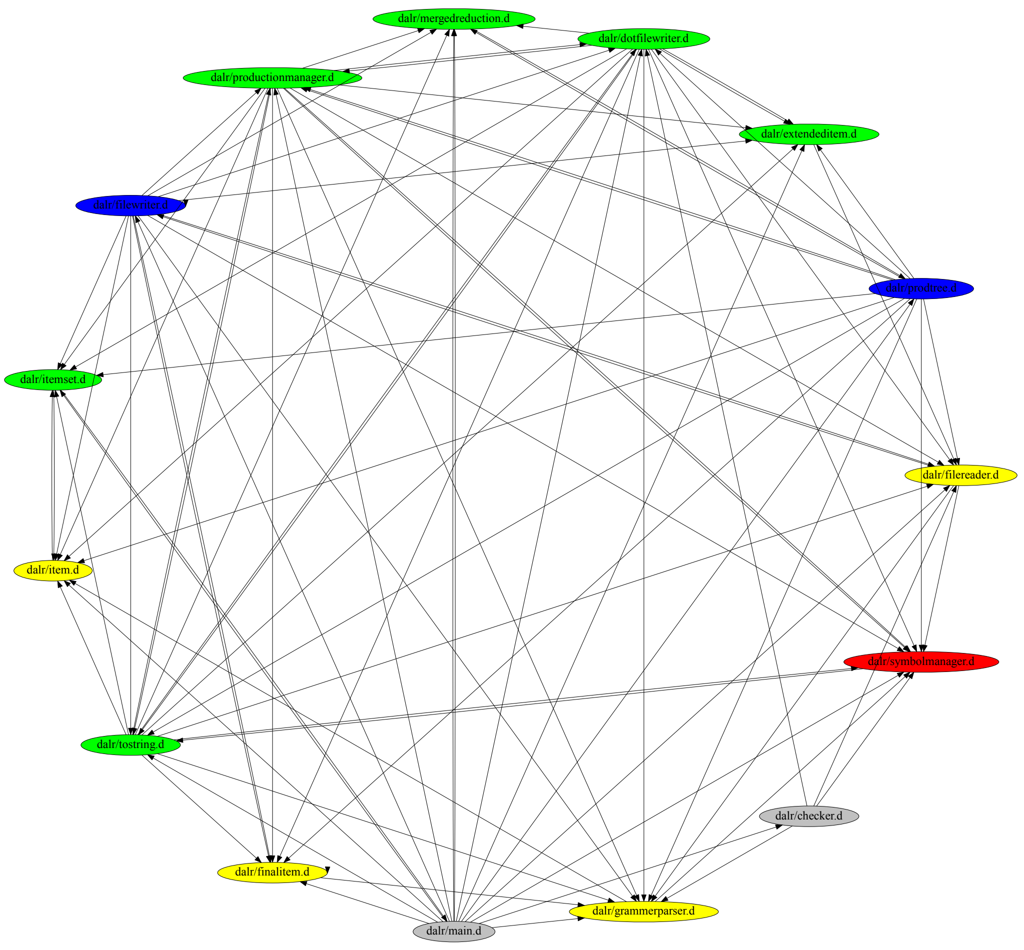


Figure 8.10.: The import relations for Dalr

with a runtime complexity of $\mathcal{O}(1)$. A single search operation takes virtually no time at all.[15]

[15]Compare to table 5.2 on 71.

8.6. Overview

Figure 8.11 gives a high level view of dmcd. The dotted red boxes represent processes.

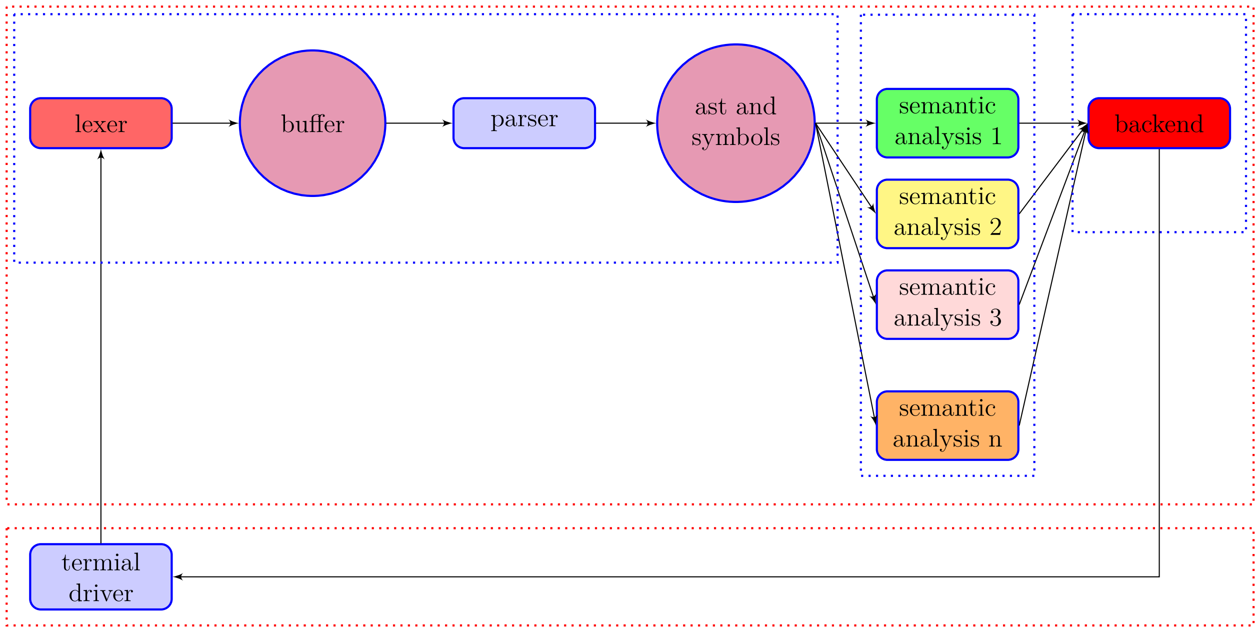


Figure 8.11.: High level view of dmcd

The rectangle in the dotted blue boxes represent groups of threads. Different colored rectangles represents different threads. The circle represent shared data structures. The thick black line represents the control flow. Not shown is the cache. A cache would be connected to all thread groups.

Part IV.

Conclusion

9. Future work

9.1. Difference Compiler

Programming languages are structured in blocks. A class is a block of functions. Functions are blocks by itself. A function block can have blocks, like *if* statements for example.

The idea is to track changes on a block to block bases and only recompile these blocks. For that to work, the compiler has to store results on a block bases.

9.2. Recursive parser

Building the parser with the parser generator Dalr takes a couple of minutes. Running dmcd for every little change, for instance while debugging, takes a lot of time. Building a recursive decent parser for an 800 rule grammar is a big task.

The benefits would be a faster build compiler. Another benefit is the language is no longer restricted to Chomsky Type-2.

9.3. Binding to llvm

Initial dmcd was to bind to the Low Level Virtual Machine backend. This was deferred, because actual emitting Low Level Virtual Machine (LLVM) intermediate code would give no new information about the usefulness of multi threading and caching in a compiler. This is because the intermediate code needs to be created single threaded after the semantic analysis finishes.

9.4. Integrating Libhurt Containers into phobos

As mentioned earlier the *D* standard library, phobos, is still missing containers. Presenting the container to the *D* community will properly lead to interesting input and might lead to a second life of the containers implementation.

9.5. Thread Pool for Semantic Analysis

As the overhead for the thread creation in the semantic analysis is considerable, implementing the dispatching as a thread pool can possible reduce this overhead. Implementing the dispatching as a thread pool and comparing the result against the present method could yield interesting results.

9.6. Combining Parts of the Semantic Analysis with the Parser

Many semantic analysis can be run before the complete syntax tree has been build. The parser could dispatch these task as threads while he is still parsing the input. This way more of the available CPUs could be used in parallel.

9.7. Make Replacement

Having a daemon running in the background might be sometimes to be change. Replacing make[1] with a program that evaluates the dependencies and issues a call to the compiler with all files that need to be rebuild could be a good compromise. Passing all the files, that need compiling, at once would allow the compiler to cache and reuse all results.

[1]Make is used as a synonym for all automatic build tools.

10. Conclusion

The goal of this thesis was to build tools and library functions that are needed to create a compiler for D in D and to create a compiler that ought to adapt to modern hardware, this includes multicore CPUs as well as big amounts of RAM.

Libhurt fills the gaps left by phobos and druntime. On over 31000 lines all during this thesis required functionality is implemented. Especially the container implementations where used over and over again. The created interfaces, like for example the ISR interface, gave the containers a new perspective on already well understood designs. Compare to chapter 5 starting on page 65.

Dex generates efficient lexers that are even able to parse utf-8. In addition to the productivity gain, through this lexer generator, excellent debugging facilities, which allow a user to print transition graphs and tables were implemented. Compared to libhurt, dex is a rather lightweight program with just over 4000 lines. This is because dex makes excessive use of libhurt and uses many functions provided. See chapter 6.

The parser generator dalr, as presented in chapter 7, with its more than 7500 lines of source, generates parsers from LALR(1) as well as GLR(1) grammars. Dalr showed its capabilities by providing the parser for the D grammar used in the compiler implementation. Again libhurt was used exhaustinvly.

Considering only the hand-written parts of the compiler dmcd, it appears to be small with only 3500 lines of code, but this is not the case, as the created lexer and parser created big portions of the compiler. Counting these lines as well, the compiler exceeds the 100000 lines. Again libhurt was used extensively here as well, as describted in chapter 8. The compiler presents four new ways of building a compiler. The separation of lexer and parser into different threads shows a performance gain of up to 25%. The multithreaded semantic analysis in combination with the array based tree presents an efficient way of using a big number of CPU cores with nearly linear speedup. The distributing of work and the caching of result gives another performance increase.

Appendix

List of Figures

Listings

List of Tables

Acronyms

ast abstract syntax tree. 46, 98, 99, 109, 110, 112, 113, 118, 120, 137

bnf Backus-naur form. 98, 137

bst binary search tree. 67–69, 137

cfg context free grammar. 41, 42, 137

CPU central processing unit. 61, 114, 115, 119

ctfe compile time function execution. 5, 19

DFA deterministic finite state machine. vii, 7, 27, 28, 30, 31, 37–40, 89–91, 109, 132, 133, 137

GLR Generalized LR. 46, 47, 97, 98, 102–106, 109, 137

ia inline assembler. 5, 137

IDE Integrated development environment. 113

ISR insert search remove. vii, 65–70, 72–75, 77, 132, 137

LALR Lookahead bottom-up parser. 44, 45, 47, 58, 97, 98, 102–105, 137

lan local area network. 114, 137

LL Left to right, left derivation parser. 42, 43, 47

LLVM Low Level Virtual Machine. 125, 137

LR Button up parser. 43–46, 137

lru least recently used. 120, 137

NFA non-deterministic finite state machine. vii, 27, 28, 30, 31, 37, 39, 40, 88–90, 137

red-black tree rbtree. 68, 69, 137

regex regular expression. vii, 27–29, 39, 87–89, 109, 138

SLR Simple button up parser. 44, 47, 48, 58

STL standard template library. 67, 72, 138

tls thread local storage. 7, 137

Glossary

inline assembler Inline assembler describes the way of implementing a function or part of functions in an assembly language. The assembler statements are written directly in the source file, instead of been written in a separate file.. 5

Bottom-up parser So called shift reduce parser.. 45

thread local storage Variable that are defined global as well as static but have an instance for every thread are called thread local. This means if one thread changes the variable, that change is not visible in any other thread.. 7

abstract syntax tree The abstract syntax tree represents parts of the syntax tree that are relevant for further processing.. 46, 99, 109, 118, 120

backus-naur form The backus-naur form is a meta-language that allows an efficient representation of context free languages.. 98

binary search tree Binary search trees are search structures that allow insertion,search, and removal in an average complexity of $\mathcal{O}(n\,log\,n)$.. 67

context free grammars Context free grammar are called languages of type-2 in the Chomsky hierarchy.. 41

deterministic finite state machine Finite state machine where for every input the transition must be strictly deterministic.. 31, 37–39, 133

Generalized LR Genearl are LR parser that follow all possible action should there be more than one present at a given state.. 46, 103

insert search remove The interface is the base for all data structures that can be used in the construction of any mapping container types.. vii, 65, 66, 74

Lookahead bottom-up parser Popular incarnation of bottom up parser with a typical lookahead of one.. 103

local area network A is a spacial restricted network, usually connect by ethernet.. 114

Low Level Virtual Machine The Low Level Virtual Machine is a compiler backend.. 125

least recently used Least recently used is a replacement strategy for caches. The data that hasn't been read the longest is removed.. 120

non-deterministic finite state machine Finite state machine where for every input the transition must not be strictly deterministic.. 28, 30, 31, 37, 89

red-black tree Red-black trees are search structures that allow insertion,search, and removal with a time complexity of $\mathcal{O}(n\,log\,n)$.. 68, 69

regular expression Regular expression allow it to construct all languages that fit into chomsky type-3.. 29, 39, 89, 109

standard template library A library of template function and class that are used for common store and algorithmic tasks in .. 67, 72

abstract The abstract keyword prevents a class of being instantiated.. 66, 67

C C is in imperativ programming language developed by Dennis Ritchie.. v, 3, 5, 8, 18, 20, 44, 48, 67, 77, 110, 138

C++ C++ is an object oriented programming language develop initially by Bjarne Stroustrup as a successor to .. 5–7, 9, 10, 44, 67, 72, 74, 75, 77, 138

C++11 C++ 11 is the latest revision off C++. Released in 2011.. 7

class Classes are an integral part of object oriented programming. In lamens' terms they are structs with method's.. 66

compile time The time when the compiler is run is called compile time.. 5, 139

conditional compiling Conditional compiling allows, through passing values to the compiler, to manipulate the parsing process of the compiler.. 5

Dalr Dalr is a parser generator created from this thesis. 97, 100, 102–104, 109, 114, 120, 121, 125, 130, 132

define Object to textural replacement another object. Used mainly in *C* and *C++*. . 5

dex Dex is a lexer generator created for this master thesis.. 95–97, 105, 109

dmcd Dmcd is the compiler developed in this thesis.. 97, 99, 104, 109, 111, 113–115, 118, 122, 125

dot Dot is a graph layout program that comes with the graphviz program collection.. 95

finate state machine Finate state machine are able to accept all word defined by Chmosky type 3 languages.. 27, 28

hash-table Datastructure that allows insertion, removal and searching with an average time complexity of $\mathcal{O}(1)$.. 68–70, 120

heap Heap memory is dynamically allocated. Usually allocating memory on the heap is a slow procedure and should therefore be avoided. Anyway, it is required to create objects in memory that should exists longer than the scope they are created in.. 61

immutability A variable marked immutable can only be assigned once.. 7

Java The Java programming language is an object oriented programming language. It was developed by James Gosling, and is currently owned by the Oracle Corporation.. 5, 6

json Json is a human readable data exchange format. The name json is an acronym for java script object notation.. 110

lexer A lexer is usually defined as a program that reads a string and splits it into words. These words are known as token.. 27, 39, 54, 55, 109, 115–117

libhurt Libhurt is the standard library created for the DMCD compiler.. 15, 66, 68–70, 76–79, 84, 86, 87, 93

member function Function of classes and structs are called member function.. 7–11, 15, 21, 22, 73–79, 86, 88, 89, 101–103, 105, 107, 111, 139

memory leak Memory leakage describes the case in which heap space is allocated by no pointer or reference exists that can be used to access the memory.. 74

mixin *D* keyword that allows to insert a string at compile to manipulate the source code.. 5

null A special pointer address that indicates that the pointer is invalid.. 67

parser A parser verifies whether or not a stream of tokens forms a word of a defined language.. 41–44, 46, 47, 109–111, 115–118, 120

phobos Phobos is the *D* standard library.. 3, 85

polymorphism Polymorphism allows the different class to be used as the same typ. For instance two class could implement a vehicle specification, but in a very different way.. 65

Semaphore A semaphore is an abstract datatype that can be used to synchronise processes.. 7

template A template allows a function or class to have members or parameter that are defined at time of instantiation.. 66

template meta programming Template function are instanced at to compute the result of function at .. 5

this A *this* pointer is a reference that can be used inside a to get access to members of the class instance.. 7

Unix Unix is an operating system first released by the company AT&T.. 44

utf Utf defines a way of encoding the same character in different ways. There are two variable length encoding called utf8 and utf16. The thrid style of encoding is called utf32, all three encodings are capable of storing the same characters. At the point of writing the utf standard defines more than one million symbols.. 39, 93

virtual The virtual keywork marks functions in C++ that can be overwritten by a derivating class.. 10

XML XML is extensible markup language that is human as well as machine readable.. 110

yacc Yacc is a LALR(1) parser, and parse table generator originally created by employees of AT&T.. 97

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[ASU86] Alfred V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[Bre89] R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11:388–403, 1989.

[Cic80] Richard J. Cichelli. Minimal perfect hash functions made simple. *Commun. ACM*, 23(1):17–19, January 1980.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[com10] Clang community. Clang recursive decent parser. `http://clang.llvm.org/features.html`, 2010.

[com12a] D community. Faq. `http://dlang.org/faq.html#q5_2`, 2012.

[com12b] D community. Getopt. `http://dlang.org/phobos/std_getopt.html`, 2012.

[Con12] The Unicode Consortium. Unicode 6.1.0. `http://www.unicode.org/versions/Unicode6.1.0/`, 2012.

[Dij61] Edsger W. Dijkstra. Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical Report 35, Mathematisch Centrum, Amsterdam, 1961.

[dla12a] dlang.org. Cpp template instance ambiguity. `http://dlang.org/template-comparison.html`, 2012.

[dla12b] dlang.org. Functions. `http://dlang.org/function.html`, 2012.

[dla12c] dlang.org. Interfaces. `http://dlang.org/interface.html`, 2012.

[dla12d] dlang.org. Lexical. `http://dlang.org/lex.html`, 2012.

[dla12e] dlang.org. Operator overloading. `http://dlang.org/operatoroverloading.html`, 2012.

[dla12f] dlang.org. Templates. `http://dlang.org/template.html`, 2012.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[fsf05] fsf. New c parser. `http://gcc.gnu.org/wiki/New_C_Parser`, 2005.

[Hol90] Allen I. Holub. *Compiler Design in C.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990. Prentice-Hall Software Series, Editor: Brian W. Kernighan.

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[Ige04] Lars Ivar Igesund. License agreement. `http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst2.aspx`, 2004.

[JA] R. Nigel Horspool John Aycock. Practical earley parsing. *The Computer Journal*, pages 620–630.

[JJ79] Stephen Johnson and Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[Joh12] Stephen C. Johnson. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/yacc/`, 2012.

[Kno65] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Mar12] Torsten Marek. Cyk javascript. `http://www.diotavelli.net/people/void/demos/cky.html`, 2012.

[McP02] Scott G. McPeak. Elkhound: A fast, practical glr parser generator. Technical Report UCB/CSD-02-1214, EECS Department, University of California, Berkeley, Dec 2002.

[McP03] Scott G. McPeak. Elkhound: A fast, practical glr parser generator. Technical report, Berkeley, CA, USA, 2003.

[oP05] University of Pittsburgh. Cs 1622 lecture 10 parsing(5). lecture10.pdf, 2005.

[saio12] The serial ata international organization. Sata 3.0 specification. `http://www.sata-io.org/technology/6Gbdetails.asp`, 2012.

[Sun12a] Sun. Object oriented programming concepts. `http://docs.oracle.com/javase/tutorial/java/concepts/index.html`, 2012.

[Sun12b] Sun. Synchronized methods. `http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`, 2012.

[Tom84] Masaru Tomita. Lr parsers for natural languages. In *Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics*, ACL '84, pages 354–357, Stroudsburg, PA, USA, 1984. Association for Computational Linguistics.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, January 1987.

[Wal08a] Julienne Walker. Binary search tree tutorial 1. `http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_bst1.aspx`, 2008.

[Wal08b] Julienne Walker. Binary search tree tutorial 2. `http://www.dsource.org/projects/tango/wiki/LibraryLicense`, 2008.

[Wal08c] Julienne Walker. Hash table tutorial 1. `http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_hashtable.aspx`, 2008.

[Wal08d] Julienne Walker. Red black tree tutorial. `http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx`, 2008.

Erklärung

Hiermit erkläre ich, Robert Schadek, dass ich diese Masterarbeit eigentständig verfaßt, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Oldenburg, den 12. September 2012