VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ JAZYKA S UŽIVATELSKY DEFINOVANÝMI SYNTAKTICKÝMI KONSTRUKCEMI

COMPILER OF A LANGUAGE WITH USER-DEFINED SYNTAX FOR NEW CONSTRUCTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. LUKÁŠ KUKLÍNEK
AUTHOR

VEDOUCÍ PRÁCE Doc. Dr. Ing. DUŠAN KOLÁŘ
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce si klade za cíl navrhnout a implementovat experimentální programovací jazyk s podporou uživatelsky definovaných syntaktických konstrukcí. Nový jazyk je kompilován do nativní binární podoby a vyžaduje statickou typovou disciplínu v době překladu. Jazyk se skládá ze dvou hlavních komponent. První z nich je minimalistické jádro založené na principech zásobníkově orientovaných jazyků. Druhou částí je mechanismus pro definici nových syntaktických konstrukcí uživatelem. Poté jsou shrnuty poznatky nabyté při návrhu a experimentování s prototypem překladače tohoto jazyka.

Abstract

This project aims to design and implement an experimental programming language. The main feature of the language shall be the ability of the user to define new syntactic constructs. The language shall be statically typed and compiled to a native binary form. The language consists of two parts. The first part is a minimalistic core based on the principles of stack-oriented languages. The second part is a mechanism that lets users define new syntactic constructs. Then we elaborate on findings that have risen from design and experiments performed with the prototype implementation of the language.

Klíčová slova

programovací jazyky, překladače, syntaktická analýza, rozšiřitelná syntax, metaprogramování

Keywords

programming languages, compilers, syntax analysis, extensible syntax, metaprogramming

Citace

Lukáš Kuklínek: Compiler of a Language with User-Defined Syntax for New Constructs, diplomová práce, Brno, FIT VUT v Brně, 2013

Compiler of a Language with User-Defined Syntax for New Constructs

Prohlášení

Prohlašuji, že jsem tuto magisterskou diplomovou práci vypracoval samostatně pod vedením pana Doc. Dr. Ing. Dušana Koláře.

....
Lukáš Kuklínek
22. května 2013

Contents

1 Introduction **6**
1.1 Syntactic Sugar 6
1.2 Domain-Specific Languages 7

2 Prior Art **9**
2.1 Source Preprocessors 9
2.2 C preprocessor 10
2.3 C++11 user-defined literals 11
2.4 camlp4 12
2.5 Haskell user-defined operators 13
2.6 Template Haskell and Quasi-Quotes 13
2.7 Agda mixfix operators 15
2.8 Factor 15
2.9 SugarJ 16

3 Background theory **18**
3.1 Context-free Grammars 18
3.1.1 Ambiguity 19
3.1.2 Attribute Grammars 20
3.2 Haskell Categorical Classes for Parsing 21
3.2.1 Functor 21
3.2.2 Applicative Functor 21
3.2.3 Monad 22
3.3 Type systems 23
3.3.1 Simply Typed Lambda Calculus 23
3.3.2 Polymorphism and Unification 24
3.3.3 Row Polymorphism 24
3.4 Concatenative Programming Languages 24
3.4.1 Examples 25
3.4.2 Typing Concatenative Programs 26

4 Technologies for Compiler Design **27**
4.1 Parser generators 27
4.2 Parsec 28
4.3 Attribute Grammar Compiler 29
4.4 Code Pretty Printers 30
4.5 libgc 31
4.6 LLVM Compiler Infrastructure 31

5 Language Design **33**
5.1 Design Goals . . . 33
5.2 The Core . . . 35
5.2.1 Type System . . . 36
5.3 Extension Mechanism . . . 37
5.3.1 Primitive Parsers . . . 38
5.4 Built-in Functions and Standard Library . . . 38

6 Implementation **39**
6.1 Modules . . . 39
6.2 Runtime Organization . . . 40
6.3 Command Line Compiler Driver . . . 41

7 Examples **42**
7.1 Bootstrapping EEL . . . 42
7.2 Structured EEL . . . 43
7.3 Meta Extensibility . . . 44
7.4 Encoding Brainfuck . . . 45
7.5 DSL for Cellular Automata . . . 45

8 Conclusion **47**
8.1 Achievements . . . 47
8.2 Limitations and Shortcomings . . . 48
8.3 Future Directions . . . 48

Bibliography **49**

A EEL Built-in Functions **52**

B EEL Help **54**

C CD-ROM Contents **55**

List of Figures

3.1 Context-free grammar ambiguity 19
3.2 Typing rules for simply-typed λ-calculus 23
3.3 Unification algorithm 24

5.1 EEL Core abstract syntax 35
5.2 EEL Core data types summary 36
5.3 EEL typing rules 36
5.4 Partial ordering on phase types 37
5.5 EEL parsing algorithm overview 38

6.1 Overview of the EEL compiler driver pipeline 41

7.1 EEL syntactic extensions metalanguage 44

Listings

2.1 C with regex extension 10
2.2 C with regex extension after preprocessor expansion 10
2.3 C++ preprocessor-based syntactic extension 11
2.4 Syntactic extension in the form of variadic C++ templates and user-defined literals 11
2.5 camlp4 extension definition 12
2.6 camlp4 extension use 12
2.7 A user-defined implication operator in Haskell 13
2.8 Routing DSL in Yesod example 14
2.9 If-then-else as Agda mixfix operator 15
2.10 Infix expressions extension in Factor 16
2.11 Idiom brackets in SugarHaskell 16
4.1 Parenthesis language in applicative style in Parsec 28
4.2 Parenthesis language in monadic style in Parsec 28
4.3 Metaparsing with Parsec 29
4.4 UUAGC example 29
4.5 Sample Pretty-Printer definition 30
4.6 Sample Pretty-Printer output 30
4.7 LLVM IR example 32
7.1 Alternative function definition syntax 43
7.2 Alternative function definition use 43
7.3 List constants extension 44
7.4 Rule 110 definition in the CA DSL 46
7.5 Rule 110 in operation 46
B.1 EEL command-line options 54
B.2 EEL interactive interpreter commands 54

List of Abbreviations

AG	Attribute Grammar
AST	Abstract Syntax Tree
CA	Cellular Automaton
CFG	Context-Free Grammar
DSL	Domain-Specific Language
EDSL	Embedded Domain-Specific Language
EEL	Experimental Extensible Language
GHC	Glasgow Haskell Compiler
I/O	Input / Output
IDE	Integrated Development Environment
IR	Intermediate Representation
QQ	Quasi Quote
RPN	Reverse Polish Notation
SSA	Static Single Assignment
TH	Template Haskell
UUAGC	Utrecht University Attribute Grammar Compiler

Chapter 1

Introduction

In most widely used programming languages today, the extensibility is limited to the possibility to define user-specified functions and data structures. The textual representation of programming language constructs is usually fixed by the grammar definition of the language. The ability of the programmer to enrich the programming language with new constructs might help to close the semantic gap between ideas of a programmer or a domain expert and the source code that needs to be written to encode the ideas.

Possible language extensions can be roughly divided into two main groups. The first group contains various general-purpose syntactic sugar to ease the development process, for example a new loop statement. The other group consists of Domain-Specific Languages (or DSLs) that are designed to perform a highly specialized task using a set of constructs specifically tailored for the task. The following sections introduce a few motivating examples of both types of syntax extensions.

1.1 Syntactic Sugar

Many languages offer constructs that do not add anything to their overall expressive power. Such constructs are good for convenience and code readability. For example, the `for` loop in C/C++ can be expressed in terms of `while` loop (which can itself be in turn expressed in terms of `goto` statements). There are many examples of such fundamentally redundant but useful constructs in programming languages.

Taking Haskell [15, 25] as an example:

1. Any function (`f a b = e`) can be transformed into its definition using λ-abstraction (`f = \a b -> e`). Abstractions with multiple arguments can be further turned into λ-abstractions with a single argument through currying (`f = \a -> \b -> e`).
2. The infix function call (``a `f` b``) is equivalent to (`f a b`).
3. The list syntax (`[a,b,c]`) can be de-sugared into a cons-based expression (`a:b:c:[]`).
4. Sequences (`[1..9]`) are replaced by function calls (`enumFromTo 1 9`).
5. Any operator (`a+b`) can be expressed as function application (`(+) a b`).
6. List comprehensions are converted into the corresponding monadic form.

7. `do` notation for monads is replaced with corresponding calls to functions defined by the `Monad` type class (`return`, `(>>=)`, `(>>)`, `fail`).

8. Special syntax for Applicative Functors has been proposed [26].

9. Special syntax for Arrows can be turned on via GHC option.

10. String literals are just a neater way to express lists of Unicode characters.

The list above is not complete. In addition to that, Haskell constantly evolves and new features are being added all the time.

The Ruby[1] programming language is known to provide handful of convenience syntactic constructs as well. Many of them deal with easy construction of various data structures such as strings, sets, lists, hash tables, or regular expressions. A particularly elegant one is string the interpolation that allows to substitute the results of expression evaluation into string literals using the `#{}` construct, for example:

```
"The square of #{x} is #{x*x}."
```

1.2 Domain-Specific Languages

Another kind of syntax extensions are so-called Domain-Specific Languages (DSLs) [13]. These languages are designed to solve a specific task. They exhibit a great expressive power within their domain while leaving out the unnecessary clutter.

DSLs come in three main flavours that are relevant for this project:

1. *External DSLs* come with their own interpreter or compiler as stand-alone tools.

2. *Generative DSLs* also come with a tool to process the DSL, but their output is a source code of another (usually general-purpose) programming language. The resultant code is in turn compiled by the target language compiler and linked with the rest of the project.

3. *Embedded DSLs* employ the expressive power of a general-purpose language to define a highly-specialized set of routines to perform a specific task. Embedded DSLs (EDSLs) usually come as libraries for the host language. The more syntax extensibility features the host language offers, the closer can the DSL match the problem domain.

One of the possible outcomes of this project might be to make the former two (especially the second one) unnecessary as they could be defined in terms of modifiable syntax of the host language.

The area of domain-specific languages is huge. The following list provides just a tiny sample of their capabilities. Standalone DSLs are particularly widespread in the UNIX world [37] (tools typeset in `verbatim` in the following list).

- Regular expressions specified as a string and compiled at run time to match against other strings can be viewed as a kind of a degenerate EDSL.

- Resource compilers are used to embed arbitrary data into executable binaries. They take a list of resources, such as strings, icons, images and other files as input, and generate a C file or an object file as output.

[1] `http://www.ruby-lang.org/en/`

- Faust [30] is a declarative language specialized in digital signal processing.
- Qt's QML [1] is a declarative language used primarily for the design of fluid interactive user interfaces. QML adds several syntactic extensions on top of the JavaScript engine and is tightly integrated with Qt written in C++. Qt itself uses a custom preprocessor to add features to C++, such as reflection and a signal-slot mechanism.
- `awk` and `sed` are tools specialized to text file processing utilizing pattern matching and substitution.
- `m4` is another text processing language, this time based on macro expansion. It is heavily used as a pre-processor in GNU autotools.
- GNU `make` is a rule-based system for automated build management.
- Database query languages, such as SQL, can be used both as embedded or standalone languages specialized in describing, querying and manipulating structured data.
- Many languages provide a command-line argument processing EDSL as a library. Option specification for C function `getopt_long` is a rather poor man's version of such EDSL.
- `yacc` is a so-called parser generator. Given a context-free grammar in EBNF form, it generates a C implementation of a parser for the grammar.

Tighter integration of the techniques and tools mentioned above with the host language compiler brings several advantages. Syntax errors in regular expressions can be reported at compile time rather than run time. Mismatches between the identifiers in the generated parser and the host code can be localized more precisely – that is in the grammar specification code rather than in the generated C code. Last but not least, once the tools (such as syntax highlighters and IDEs) contain the support for user-defined syntax, the user gets the support of the tool for his DSL virtually for free.

Chapter 2

Prior Art

This chapter reviews existing approaches to provide syntactic extensions in various programming languages. Each system is briefly described, an example of its usage is demonstrated and its properties, advantages and disadvantages are evaluated.

The approaches vary in several different ways:

- The degree of integration with the target programming language.
- The complexity of the constructs, by which the target language can be extended.
- Guarantees about the correctness of the code generated by the extension to the syntax.

2.1 Source Preprocessors

Source preprocessors take input source in a target language with some additional constructs, and produce the source with the constructs being replaced with equivalent constructs in the plain target language.

New constructs can be to some degree added to any language using a general purpose macro preprocessor (such as `m4`) or a template engine. These tools usually perform textual search-and-replace based on a user-defined set of rewriting rules, often called *macros*.

Example

In this example, we consider adding regular expressions to the C language. Regular expressions are enclosed between two slash characters (Perl-inspired syntax). They are transformed into raw C strings and passed to a regex compilation function. The `sed` tool [37] is used to perform the transformation, and the program reads as follows:

```
sed 's:/\([^*][^/]*\)/\([miu]*\):re_compile("\1", "\2"):g' $CFILE
```

A sample input can be seen in listing 2.1 and the corresponding output is in listing 2.2. Several problems with this approach can be spotted straight away. First of all, the replacement takes place even in comments and string literals. That is probably not the desired behaviour. If the regex contains a double quotation mark (`"`), the generated source will probably not even be a lexically valid C source. Also, the regex itself is not checked for syntactic correctness, for example parentheses need to be balanced. Some of these shortcomings can be at least partially solved by a more elaborate regular expression passed to the `sed` command, but many cannot.

```
void main(int argc, char** argv)
{
    if (re_match(/[a-z0-9.]+@[a-z0-9.]+\.[a-z]+/i, argv[1]))
        printf("OK\n");
    else printf("Bad e-mail\n");
}
```

Listing 2.1: C with regex extension

```
void main(int argc, char** argv)
{
    if (re_match(re_compile("[a-z0-9.]+@[a-z0-9.]+\.[a-z]+", "i"), argv[1]))
        printf("OK\n");
    else printf("Bad e-mail\n");
}
```

Listing 2.2: C with regex extension after preprocessor expansion

Evaluation

Most of the time, a general preprocessor is a particularly inelegant solution with numerous disadvantages. It adds complexity to the build process. As the macro expansion step can produce an invalid input for the host compiler, the error elimination process involves inspection of the generated code, which is usually rather unreadable for the programmer. Also, the new constructs usually do not play well with IDEs used for source code editing. The use of a general-purpose preprocessor is also inappropriate for extending syntax of a layout-sensitive programming language, such as Python. Most of the disadvantages are caused by lack of any integration with the target language what so ever.

Specialized Preprocessors Some of the limitations of text-based preprocessors might be overcome using a specialized preprocessor. By a specialized preprocessor, we mean any preprocessor that is to some degree integrated with the target language. Examples of such systems include the C preprocessor (section 2.2), OCaml's camlp4 (section 2.4, page 12). Some preprocessors extend the target language with a fixed set of features. For example UUAGC (briefly discussed in section 4.3 on page 29) adds support for attribute grammars to the Haskell programming language.

2.2 C preprocessor

The C preprocessor can be used for somewhat limited syntactic extensions as well. It is based on a simple search-and-replace over C/C++ tokens, unlike character-based matching of a general-purpose preprocessor. The preprocessor is distributed in one package with the C compiler and forms an inseparable part of the C tool-chain.

Example

Take the new `FOREACH` loop as an example of a loop iterating over a C++ container. The loop expands to a regular C++ for loop. Sadly, the implementation in listing 2.3 depends on the new C++11 `auto` keyword semantics. Without it, the macro definition would have to be much more convoluted.

```
#define FOREACH(i, c) for (auto i = (c).begin(); i != (c).end(); ++i)

void func(std::vector<int>& array)
{
    FOREACH(it, array)
        std::cout << "The element is " << *it << std::endl;
}
```

Listing 2.3: C++ preprocessor-based syntactic extension

Evaluation

The C preprocessor has a couple of advantages over general-purpose preprocessors. First, it takes the lexical structure of the C language into account avoiding accidentally fused tokens and other possible lexical errors. Second, IDEs and tools are aware of the preprocessor, so they are able to guide the programmer (usually in a somewhat limited way) through the code even in the presence of macros. On the other hand, the C preprocessor ignores C/C++ syntax and can yield invalid constructs resulting in relatively incomprehensible error messages. Also names introduced by macros can clash with other identifiers in the program. Expressions have to be parenthesized to avoid unexpected behaviour due to operator precedence. Another disadvantage is that the expansion of multiple macro parameters can lead to duplication of side effects and thus unexpected behaviour of the program .

2.3 C++11 user-defined literals

The new version of the C++ standard called C++11 features several additions to the language that open new possibilities for implementing EDSLs [42]. Extensible user-defined literals can be processed character-by-character by variadic templates and constant expressions at compile time.

Example

Listing 2.4 shows an example of an extension of the C++ language with binary literals processed at compile time.

```
template<char...> struct Bin;
template<> struct Bin<> { enum { val = 0 }; };
template<char... tail> struct Bin<'0', tail...>
        { enum { val = Bin<tail...>::val }; };
template<char... tail> struct Bin<'1', tail...>
        { enum { val = (1 << sizeof...(tail)) + Bin<tail...>::val }; };

template<char... str> int operator"" _b() {
    return Bin<str...>::val;
}

int main(void) {
    std::cout << 10110_b << std::endl;  // prints 22
}
```

Listing 2.4: Syntactic extension in the form of variadic C++ templates and user-defined literals

Although not explicitly supported by the standard, there are ways to turn user-defined string literals into template arguments and process the string at compile time. It can be used to embed DSLs, such as regular expressions, into C++ as string literals with compile-time syntax checking.

Evaluation

The error messages reported on a syntax error in the DSL can be very convoluted. This can be partially remedied by providing custom error messages via `static_assert`. The DSL defined this way must be fully self-contained, it cannot reference outer lexical environment. Therefore, features like string interpolation cannot be implemented by this techniques. Other disadvantages are very cluttered syntax of templates, and the complexity of their implementation – with many unexpected corner cases to be taken into account causing hard very debugging.

2.4 camlp4

Camlp4 [8] stands for Pre-Processor Pretty Printer (hence the p4) for OCaml. It is tightly integrated with the OCaml compiler. It can be used to introduce many language features, such as local operator overloading, syntax for exception handling, syntax sugar for monads, convenient building of data structures, regular expressions, loading of data from external files to OCaml data structures, DSL implementation and so on and so forth.

Example

An example of a syntax extension for an if-then-else operator similar to the C ternary operator is defined in listing 2.5 and used in listing 2.6.

It is a simple extension to a grammar rule. Productions are bound to variables c, t, e and the resultant piece of the abstract syntax tree is constructed by means of the `expr` quasi quotation – with the condition, the if-part and the else-part expression variables interpolated.

```
EXTEND
  GLOBAL: Pcaml.expr;

  Pcaml.expr: LEVEL "simple" [
    [ "{"; c=Pcaml.expr; "?"; t=Pcaml.expr; ":"; e=Pcaml.expr; "}" ->
        <:expr< if $c$ then $t$ else $e$ >>
    ]
  ]
END;;
```

Listing 2.5: camlp4 extension definition

```
print_endline { 3 > 5 ? "foo" : "bar" };;
```

Listing 2.6: camlp4 extension use

The `EXTEND` clause introduces a grammar extension. We are extending expression syntax as indicated by the `Pcaml.expr` label. The `LEVEL` indicates the priority of the operator (priorities are named). Terminals in the new grammar rule are indicated by double quotes. Results of non-terminals (in this case subexpressions) are bound to local variables, which are in turn used on the right-hand side of the rule indicating what OCaml construct is the new syntax mapped to.

Evaluation

Camlp4 is tightly integrated with the OCaml compiler. It is able to report the location of errors inside a source code with an extended syntax. The compiler can be instructed to load syntax extension files, avoiding direct invocation of the preprocessor. The syntax extension definition file has to be processed and compiled. As a consequence, the build process is considerably more complicated when compared to the usage of plain OCaml.

2.5 Haskell user-defined operators

Haskell [25] is a pure functional language. The prominent role of functions in a pure functional language implies that many more constructs can be expressed by function application instead of brand new language constructs. Higher-order functions, such as *fmap* and *fold*, provide useful loop abstractions that are hard to express in an imperative language in any other way than by a built-in statement.

Haskell provides a limited user-defined syntax capability. Users can introduce new infix operators via fixity declaration where they specify priority, associativity (left, right, or none), and the operator symbol itself.

Example

For example a logical implication operator can be seen in listing 2.7.

```
infix 1 -->
(-->) :: Bool -> Bool -> Bool
a --> b = not a || b

test = [(a, b, a --> b) | a <- ft, b <- ft ]
    where ft = [False, True]
```

Listing 2.7: A user-defined implication operator in Haskell

Evaluation

User-defined operators with higher-order functions together with type classes form a powerful mechanism for EDSL creation. Although this approach has some limitations, it is often possible to match the domain quite closely. Because no “real” syntax extensions are actually added to the host language, generated error messages are as readable as with the usual function applications. Only binary operators can be introduced, therefore only the expression syntax can be altered this way.

2.6 Template Haskell and Quasi-Quotes

Quasi-Quotes (QQ) and Template Haskell (TH) are extensions of the Haskell programming language implemented in the Glasgow Haskell Compiler[1]. This powerful combination of

[1]`http://www.haskell.org/ghc/`

extensions can be used to extend Haskell's syntax far beyond capabilities of user-defined operators. But they come with their own set of shortcomings.

Template Haskell is Haskell's take on compile-time metaprogramming [40]. It allows the user to write functions that generate Haskell AST, execute them at compile time, and splice their output into the AST being built by the compiler. The function can be called at the top level to generate new declarations or definitions, such as new data types or functions. Alternatively, the resulting AST subtree is attached to the global tree at the position of the `$(thFunc a b)` construct, for example inside expressions.

Quasi-Quotes [23] are pieces of code enclosed in square-bar brackets `[qq|...|]`, where `qq` is the name of the Quasi-Quoter used to parse the content between the vertical bars. Arbitrary Haskell code with required type signature may be used to parse the quasi quote contents, including Parsec (see page 28). Template Haskell can be used to generate the resultant AST that will be placed in the position of the original quote. The splice syntax `$(...)` can be made available inside the quasi quoter too.

Example

Yesod web framework [41] makes a heavy use of both TH and QQs to implement DSLs on a number of different places:

- Specifying mapping from URLs to page handler functions and vice versa (routing). When using this system, it is guaranteed that no invalid URL to the pages of the website being developed are ever generated by the framework. Listing 2.8.
- Specifying database layout in a simple DSL is used to access the database in a type-safe manner, to automate database schema creation and migrations, and to abstract the user away from a particular database implementation.
- Number of DSLs for templating the HTML/CSS/JavaScript ensure that all variables used in the templates are present and type correct at compile time.

```
mkYesod "TestApplication" [parseRoutes|
    /              HomeR  GET
    /about         AboutR GET
    /blog/#PostId  PostR  GET POST
|]
```

Listing 2.8: An example of the Yesod's routing DSL. Entries come in the following order: URL template (variables are prefixed with a hash mark), resource name, and a list of available HTTP methods for the resource.

Evaluation

The combination of Quasi-Quotes and Template Haskell is a very expressive way to extend the language syntax. Any string can be put into a quasi-quote. TH makes it possible to load external files with arbitrary syntax, for which a parser can be written, and incorporate them into the project. Possible use cases include extending Haskell with new constructs as well as parsing embedded and external DSLs.

Although the ability to use arbitrary code to parse the input is really powerful, it causes several problems. The possibilities of how the language grammar can be analyzed suffer significantly. As a consequence, it is possible to generate an AST that contains semantic errors. The compiler cannot infer the exact location of the error, so it reports the whole quasi-quote as containing an error. Creating extensions this way is not on-line: The Quasi-Quoter and Template Haskell definitions have to be compiled in a separate module from the one they are used in.

2.7 Agda mixfix operators

Agda [29] is a research programming language with dependent types. From the syntactic extensibility point of view, it introduces an interesting concept called mixfix operators, as opposed to Haskell's ability to define only binary operators. The underscore (_) character acts as a place-holder for operands, everything else is a part of the operator. The parser builds a directed acyclic precedence graph, which is used to efficiently parse the code with mixfix operators [7].

Example

As an example, we take the `if-then-else` construct. It is not a built-in operator in Agda. Instead, it is defined as a mixfix operator. See listing 2.9.

```
if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true then x else y = x
if false then x else y = y
```

Listing 2.9: If-then-else as Agda mixfix operator

Evaluation

The concept of mixfix operators seems to be quite powerful, especially in a declarative language where almost everything is an expression, so it is enough to have just a single non-terminal, expressed as the underscore. As with Haskell's user-defined operators, top-level declaration cannot be extended this way, only expressions can. It is not possible to change the lexical syntax either.

2.8 Factor

Factor is a dynamic concatenative programming language designed by Sviatoslav Pestov [33]. By default, it uses *Reverse Polish Notation* (or RPN) to describe expressions and algorithms. The parser uses a fixed lexical analysis stage which breaks the source code into a series of white-space separated tokens. Almost everything else can be changed. Special parsing words can be used to bypass the built-in parsing mechanism and replace it locally with a custom one.

Example

For example, the `infix`[2] vocabulary (library) uses this mechanism to introduce the ability to write expressions with infix operators and traditional syntax for function application with parentheses $f(x_1, x_2, \dots)$. Such expressions are enclosed in special brackets:

[2]`http://docs.factorcode.org/content/article-infix.html`

[infix ... infix]. An example of the usage is shown in listing 2.10, the sample evaluation of myHypot(3.0,4.0) yields 5.0. The :: symbol introduces a new word (function).

```
USING: infix locals math.functions ;
:: myHypot ( x y -- z ) [infix sqrt(x * x + y * y) infix] ;
[infix myHypot(3.0, 4.0) infix] .
```

Listing 2.10: Infix expressions extension in Factor

Evaluation

The extension mechanism in Factor is very powerful with a broad class of possible syntactic extensions. Strong metaprogramming capabilities make complex compile-time code transformations possible [32]. Changing lexical syntax is somewhat limited but it does not seem to be a problem as lexical analysis can be bypassed to some degree, and it is already very liberal in what it accepts. Being a dynamic language, Factor does not do many sanity checks during the compile-time and possible semantic problems may be harder to spot through the enriched syntax.

2.9 SugarJ

The SugarJ project[3] aims to provide library-based syntactic extensibility. Originally, the project was Java-centric [11] but it has recently been extended to support multiple host languages, including Haskell [12]. It is essentially a preprocessor that interprets language extensions written in a context-free and/or layout sensitive syntactic rules and expands them into a de-sugared form. A specialized DSL called Stratego [45] is used to perform term transformations between source and target trees.

Example

Listing 2.11 demonstrates a use of a syntax extension that introduces so called idiom brackets [27] for applicative functors (see section 3.2.2, page 21) in SugarHaskell. With this extension, (|f a b|) is desugared to much longer combinatoric version pure f <*> a <*> b. The example implements a parser for the balanced brackets language that returns the maximum bracket nesting depth in an input string (provided the string is in the language). De-sugared version would look similar to the Parsec example in listing 4.1.

```
import Text.Parsec
import Control.Applicative
import Control.Applicative.IdiomBrackets

parens = (| max (| inc (char '[') parens (char ']') |) parens |)
     <|> (| 0 |)
  where inc _p1 depth _p2 = depth + 1
```

Listing 2.11: Idiom brackets in SugarHaskell

Evaluation

Using SugarHaskell feels quite similar to using Template Haskell with a Quasi Quoter but offers more free-form syntax (no actual quasi quoter brackets are needed in the source file).

[3]http://www.student.informatik.tu-darmstadt.de/~xx00seba/projects/sugarj/index.html

It makes it possible to define a broad range of composable syntactic extensions that fit well the general Haskell syntax and can be freely mix-and-matched. It also offers some integration with the Eclipse IDE[4].

However, its preprocessor nature prevents the compiler from locating semantic errors in the source code, the location is given in terms of the expanded de-sugared code. Also, newly introduced syntax cannot be used in the file where it is defined.

[4]`http://www.eclipse.org/`

Chapter 3

Background theory

This chapter contains the 10,000 feet overview of the necessary theoretical background to facilitate further discussion. The text on following pages assumes that the reader is familiar with basic concepts and terminology of these areas:

- *Discrete mathematics:* sets, relations, functions, trees, morphisms, monoids, logic
- *Formal language theory:* alphabet, strings, languages
- *Haskell:* higher-order functions, type definitions, type classes (or equivalent concepts in a similar programming language) + lambda calculus

3.1 Context-free Grammars

Context-free grammars [28] are one of the fundamental foundations for defining programming language syntax. The concept of CFG is easily understandable and forms a good base for user-defined syntactic extensions.

Definition 1 (Context-free grammar). *Context-free grammar is a quadruple $G = (N, \Sigma, P, S)$, where*

- *N is a finite set of non-terminals*
- *Σ is a finite set of terminals, $N \cap \Sigma = \emptyset$*
- *P is a finite binary relation $P \subseteq N \times (N \cup \Sigma)^*$, members of the set are also called production rules*
- *S is the start symbol, $S \in N$*

An ordered pair $(A, \beta) \in P$ is usually denoted as $A \to \beta$. $A \to \beta_1 \,|\, \beta_2$ is an abbreviation for two rules $A \to \beta_1, A \to \beta_2$.

Definition 2 (Rewriting relation). *Rewriting rule application is a binary relation $\Rightarrow$ over strings $u, v \in (N \cup \Sigma)^*$, $u \Rightarrow v$ iff $\exists A \to \beta \in P, u_1, u_2 \in (N \cup \Sigma)^* : u = u_1 A u_2 \wedge v = u_1 \beta u_2$. Transitive and reflexive closure of $\Rightarrow$ is denoted as $\Rightarrow^*$.*

Definition 3 (Context-free language). *$L(G)$ is a context-free language iff exists a context-free grammar G such that $L(G) = \{w \,|\, w \in \Sigma^*, S \Rightarrow^* w\}$. $L(G)$ is said to be a language generated by grammar G.*

3.1.1 Ambiguity

A *parse tree* of a string w captures the syntactic structure of the string according to a context-free grammar G. It is an n-ary tree with ordered branches. Intermediate nodes are labelled with non-terminals and leafs are labelled with terminals. For any intermediate node labelled A with children $\alpha \in (N \cup \Sigma)^*$, there must be a production rule $A \to \alpha \in P$. Labels of leaf nodes in the post-order traversal order make up the string w.

Definition 4 (Ambiguous CFG)**.** *A context-free grammar G is said to be ambiguous iff there exists a string $w \in L(G)$ and at least two distinct G-based parse trees for w.*

Ambiguity Example Let $G_1 = \{\{E\}, \{a, b, c, +, *\}, \{E \to E + E \,|\, E * E \,|\, a \,|\, b \,|\, c\}, E\}$ be a context-free grammar. String $a + b * c$ can be generated by two parse trees as shown in figure 3.1. Both parse trees correctly indicate that $a + b * c \in L(G_1)$. However if the E non-terminal is interpreted as a (sub)expression of a language of arithmetic expressions with addition and multiplication, the semantics implied by the two parse trees is different, as $(a + b) * c \neq a + (b * c)$.

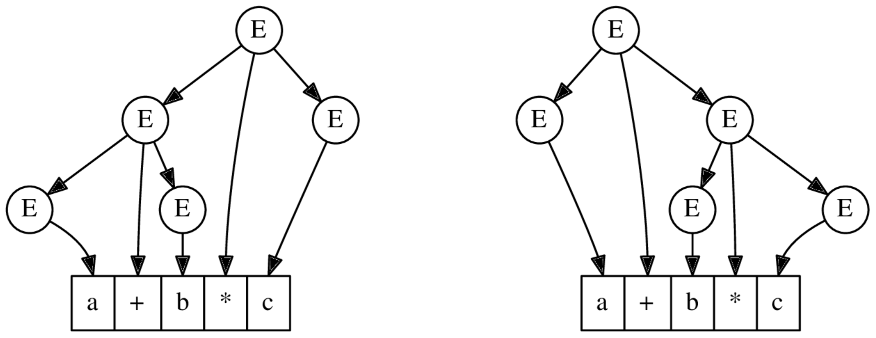

Figure 3.1: Context-free grammar ambiguity

Implications for Extensible Syntax In the presence of user-defined syntactic constructs, ambiguity causes a severe problem. It is easy to define an ambiguous grammar by accident, as grammar rules may be scattered among several source files.

Furthermore, it is not generally possible to detect such ambiguity at the time of the grammar definition, because the problem of determining whether a CFG is ambiguous or not is known to be undecidable [28]. The decision has to be made partially, and/or postponed until the parser is presented with a string. Alternatively, the grammar can be disambiguated in some other way, such as by introducing an ordered choice operator or by fixing the order in which rewriting rules are applied (e.g. always expand the leftmost non-terminal as in the LL parsing). One way or another, any such restriction changes the class of languages that can expressed with the grammar, sacrificing the conceptual clarity and simplicity of CFGs.

3.1.2 Attribute Grammars

Attribute grammars were introduced by Donald E. Knuth as an extension to context-free grammars [19] for formalizing semantics of context-free languages. The abstract syntax tree of a context-free language is annotated with attributes [43, 44]. Production rules of the grammar are extended to provide attribute definitions in terms of functions of other attributes, called *semantic functions*.

Let $G = (N, \Sigma, P, S)$ be a CFG. For any production rule $X \to X_1, X_2, \ldots, X_n \in P$ let $S(X_i)$ and $I(X_i)$ denote the set of synthesised and inherited attributes respectively. $A(X_i) = S(X_i) \cup I(X_i)$ is the set of all attributes of X_i for $1 \leq i \leq n$. Attribute a of node X is denoted $X.a$.

Synthesized attributes For synthesized attributes, data flow bottom-up the syntax tree. Semantic functions for $S(X)$ depend on $A(X_i)$, $1 \leq i \leq n$. On the bottommost level, the values of synthesised attributes of terminal nodes are determined as functions of type $\Sigma \to T_a$, where T_a is the type of the attribute.

Synthesised attributes can be used for the tracking of positions of grammatical constructs in the source code, for the performation of the type checking and/or type inference where the type of a compound expression depends on types of its sub-expressions, or for partial evaluation.

Inherited attributes Inherited attributes flow from the root node towards the leafs in a top-down fashion. Inherited attribute value for $I(X_i)$ depends on values of $A(X)$. Initial values for synthesised attributes for the root node have to be provided explicitly.

Inherited attributes are useful for passing the environment, such as the symbol tables and data type definitions. They can also be used to pass the aggregate values computed by the synthesised attributes back down the tree.

Example Take the normalization by average of leafs L in AST generated by grammar $G_2 = \{\{L, R, N\}, \{x \,|\, 0 \leq x \leq 10\}, \{L \to 0 \,|\, 1 \,|\, \ldots \,|\, 10, N \to NN \,|\, L, R \to N\}, R\}$:

1. Count and partial sums of leafs are passed upwards as synthesised attributes
   (a) For $L \to x, 0 \leq x \leq 10$: $L.sum = x$, $L.count = 1$
   (b) For $N \to L$: $N.sum = L.sum$, $N.count = L.count$
   (c) For $N \to N_1 N_2$: $N.sum = N_1.sum + N_2.sum$, $N.count = N_1.count + N_2.count$
2. The average is computed at the root and passed as an inherited attribute through the tree all the way down to the leafs.
   (a) For $R \to N$: $N.avg = N.sum/N.count$
   (b) For $N \to N_1 N_2$: $N_1.avg = N_2.avg = N.avg$
3. Leafs subtract the average from their initial value to compute the result attribute.
   (a) For $N \to L$: $L.result = L.sum - N.avg$

3.2 Haskell Categorical Classes for Parsing

In this section, we examine three of the most ubiquitous categorical structures in Haskell with a special emphasis on the parsing use case. They are represented in terms of Haskell type classes. In addition to that, they shall obey certain laws in order to preserve the expected semantics. The three structures are *functors*, *applicative functors* (augmented with *alternatives*) and *monads*. Rather than diving deeply into theory and full generality of the structures, they are presented from a practical parser-programming standpoint.

3.2.1 Functor

Definition 5. *Functor is a type constructor F of kind $* \to *$ together with the mapping operation $fmap : (a \to b) \to (F\ a \to F\ b)$, subject to these laws:*

1. *Identity: $fmap\ id = id$*
2. *Composition: $fmap\ f \circ fmap\ g = fmap\ (f \circ g)$*

Functors generalize the *map* function over lists. Intuitively, using $fmapf$ applies the function f independently to all elements of some greater structure of type $F\ a$, transforming it into a structure of type $F\ b$. Categorically, Haskell functors are considered to be endofunctors [4].

In the context of parsing, $F\ a \equiv Parser\ a$, where $Parser\ a$ is a parser which after possibly consuming some input yields a result of type a, e.g. an Abstract Syntax Tree. The parser combines several monadic effects, namely the possibility of failure, state for tracking the input stream, and sometimes non-determinism. Applying $fmap\ f$ to a parser just transforms the result of a parser without causing any further effects. This is enforced by the functor laws. $Parser$ is a functor.

3.2.2 Applicative Functor

Definition 6. *Applicative functor F is a functor with these operations:*

- $pure : a \to F\ a$
- $\circledast : F\ (a \to b) \to (F\ a \to F\ b)$

Subject to these laws:

1. *Identity: $pure\ id \circledast x = x$*
2. *Composition: $pure\ (\circ) \circledast f \circledast g \circledast x = f \circledast (g \circledast x)$*
3. *Homomorphism: $pure\ f \circledast pure\ x = pure\ (f\ x)$*
4. *Interchange: $u \circledast pure\ v = pure\ (\lambda f \mapsto f\ v) \circledast u$*

Applicative functors [26] are more powerful than plain functors. While the list functor represents mapping an unary function over a list, its applicative counterpart can be seen as combinatorially mapping an arbitrary number of n-ary functions of the same type over n lists. Functor $fmap$ operation can be reconstructed from applicative methods as $fmap\ f\ x = pure\ f \circledast x$. Applicative functor laws make sure that functor laws hold.

In the context of parsing, it is possible to convert rules in the $X \to Y_1 Y_2 \dots Y_n$ form to an applicative expression $p_X = \mathit{pure}\ f \circledast p_{Y_1} \circledast p_{Y_2} \circledast \dots \circledast p_{Y_n}$. The p_N term stands for a parser for a non-terminal N, assuming we have a family of parsers char_a for each $a \in \Sigma$ that consume terminal symbols and return them. The results of partial parsers are combined using the function f. Thus, building an AST can be done by representing the AST as an algebraic data type and using a constructor as f in the production rule parsers. The $\mathit{pure}\ x$ function represents a parser that does not consume any input and immediately yields value x.

Yet, the applicative interface itself does not allow to model alternatives. That is when there is more than one production rule for a single non-terminal on the left hand side of the rule. Extending an applicative functor with a monoid gives an *Alternative functor:*

Definition 7. *Applicative functor with alternatives F is an applicative functor with an additional constant and an additional operation:*

- $\mathit{empty} : F\ a$
- $\oslash : F\ a \to F\ a \to F\ a$

Subject to these laws:

1. *Identity:* $\mathit{empty} \oslash p = p = p \oslash \mathit{empty}$
2. *Associativity:* $(a \oslash b) \oslash c = a \oslash (b \oslash c)$

Parsing-related semantics of the *empty* constant is a parser that always fails. The $\oslash$ implements the choice operator. Grammar rule $X \to \beta_1 \,|\, \beta_2 \,|\, \dots \,|\, \beta_n$ can be translated as $p_X = p_{\beta_1} \oslash p_{\beta_2} \oslash \dots \oslash p_{\beta_n}$. Sometimes, parsers implement the ordered choice operator returning just the result of the first parser that matches the input violating the associativity law.

3.2.3 Monad

Definition 8. *Monad M is a functor together with these two operations:*

- $\mathit{return} : a \to M\ a$
- $\mathit{bind} : M\ a \to (a \to M\ b) \to M\ b$

Let $f \circledcirc g = \lambda x \mapsto (\mathit{bind}\ (\mathit{bind}\ x\ g)\ f)$ be the monadic composition operator (analogous to function composition) of type $(b \to M\ c) \to (a \to M\ b) \to (a \to M\ c)$. Monad M is subject to these laws:

1. *Identity:* $\mathit{return} \circledcirc f = f = f \circledcirc \mathit{return}$
2. *Associativity:* $(f \circledcirc g) \circledcirc h = f \circledcirc (g \circledcirc h)$

In Haskell, the *bind* operation is denoted as an infix operator `>>=`. Monad is the most powerful structure discussed so far. Monads can be used to model many effects, including non-determinism, mutable state, or communication with the outside world [34].

Looking at the *bind* operation from the parsing point of view, it can be used to replace the $\circledast$ operation for sequencing of parsers. $X \to Y_1 Y_2 \dots Y_n$ becomes:
$(\mathit{bind}\ p_{Y_1}\ (\lambda a_1 \mapsto \mathit{bind}\ p_{Y_2}\ (\lambda a_2 \mapsto \dots (\mathit{bind}\ p_{Y_n}\ (\lambda a_n \mapsto \mathit{return}\ (f\ a_1\ a_2 \dots a_n))))))$

As can be seen from the above statement, subsequent parsers can be determined by the results of preceding parse. That is exactly what is needed for extensible syntax implementation. Take the expression ($\mathit{bind}\ p_A\ (\lambda a \mapsto \text{if } \mathit{pred}\ a \text{ then } p_X \text{ else } p_Y)$) as an example. Here, the way to parse the rest of the output depends on whether the result of parser p_A matches given predicate *pred*. The predicate can be an arbitrarily complex computation.

3.3 Type systems

This chapter briefly introduces type systems on the language of lambda calculus [35]. Task of the type system is to prevent certain class of programming errors from occurrence by restricting sets of possible values that can be computed by any phrase of the programming language in question. Type system for our new programming language will be discussed later on.

Let us introduce λ-terms first.

Definition 9 (λ-term)**.** *Any λ-term t can be generated by repeatedly applying the following rules one at a time a finite number of times:*

1. $t = v$ *(t is a variable)*
2. $t = \lambda x.t_1$*, where x is a variable and t_1 is a λ-term (abstraction)*
3. $t = t_1 t_2$*, where t_1, t_2 are λ-terms (application)*

Semantics of evaluation of a λ-term t is briefly and rather informally described as follows:

1. If $t = (\lambda x.t_1)t_2$, the resultant term is t_1 with all occurrences of variable x in t_1 bound by the header of the λ-abstraction being replaced by t_2 in such a way that no free variable in t_2 becomes bound (variable renaming might be needed).
2. Otherwise, the term t is not reducible.

3.3.1 Simply Typed Lambda Calculus

Simple types extend the untyped λ-calculus in a number of ways. Most notably, types τ are introduced. The first type to deal with is the function type denoted $\tau_1 \to \tau_2$, where τ_1, τ_2 are types. The set of all types with function types only is empty, so the simple type system is usually further enriched by either the boolean type or the naturals. To simplify the type-checking, each variable bound by an λ-abstraction is explicitly annotated with its type: $\lambda x : \tau.t$ specifies the type of variable x to be τ.

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \cup \{(x,\tau_1)\} \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2}\ \text{ABS} \qquad \frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau_1}\ \text{APP}$$

Figure 3.2: Typing rules for simply-typed λ-calculus

Typing relation $\vdash$: is a ternary relation between an environment Γ, a term t, and a type τ. Relation $\Gamma \vdash t : \tau$ reads as: given the environment Γ, term t has type τ. The environment maps free variables to their types. Any well-typed term is provable from typing rules in figure 3.2 plus used extensions. The listed rules deal only with function types, rules for typing any extensions (such as booleans or naturals) are not included.

3.3.2 Polymorphism and Unification

Polymorphism is introduced by adding type variables to the mix. In this section, we restrict ourselves to universally quantified type variables. For all valid types, the substitution of the type for any type variable has to be well-typed. With type polymorphism, a λ term really represents a family of functions.

Type reconstruction also called type inference, relaxes the need for type annotations in λ-abstractions. Take an application of function $f : \tau_2 \to \tau_1$ to $x : \tau_3$ as an example: fx generates a constraint $\tau_2 = \tau_3$. The constraint follows from the typing rule APP from figure 3.2. Then, unification takes place to make τ_2 and τ_3 equal. The result of unification is a mapping from type variables to type terms which can be applied to τ_1 to get the final type of the term $fx : unify(\tau_2, \tau_3)(\tau_1)$.

Figure 3.3 shows the basic skeleton of the unification function. The $\circ$ symbol is used as a substitution composition symbol capturing the notion of substitution transitivity: if c is substituted for b and b is substituted for a then c is substituted for a. Type constructors other than function are unified analogously to them. Note that the algorithm may fail which means the generated constrain is not satisfied meaning that the term is not well typed.

$$unify(\tau_1, \tau_2) = \begin{cases} \emptyset & \text{if } \tau_1 = \tau_2 \\ \{v \to \tau_2\} & \text{if } v = \tau_1 \text{ is a type variable} \\ unify(\tau_2, \tau_1) & \text{if } \tau_2 \text{ is a type variable} \\ unify(\sigma(\tau_{1b}), \sigma(\tau_{2b})) \circ \sigma & \text{if } \tau_1 = \tau_{1a} \to \tau_{1b} \\ & \text{and } \tau_2 = \tau_{2a} \to \tau_{2b} \\ & \text{and } \sigma = unify(\tau_{1a}, \tau_{2a}) \\ \text{fail} & \text{otherwise} \end{cases}$$

Figure 3.3: Unification algorithm

3.3.3 Row Polymorphism

Row polymorphism was originally introduced by Didier Rémy for typing record types [38]. A row type variable does not represent just a single type, it can be unified with a whole row of types. Consider a record r with labels l_1, l_2, where $r : \{l_1 : \tau_1, l_2 : \tau_2 \,|\, v\}$. Here, v is the row variable. Record of this type can be unified with any other row polymorphic record if all labels are distinct. In the process of unification, the row variable v unifies with all the fields of the record r it is being unified with.

The concept of row polymorphism has been later extended to accommodate sum types (tagged unions) [22] and ordered products (tuples) [10].

3.4 Concatenative Programming Languages

Concatenative programming, described by some as "An Overlooked Paradigm in Functional Programming" [14], is a way to write computer programs using composition of functions [46]. Most concatenative languages are stack-based: all the functions map a stack to a stack.

The programs (= functions) are constructed as follows:

1. Any pre-defined built-in program is a program.

2. If p_1 and p_2 are programs, then p_1 p_2 is a program. This is the program composition and has the semantics of $p_2 \circ p_1$. In a concatenative language, data flow from left to right, whereas in the mathematical notation it is the other way around.

3. If p is a program then $[p]$ is a program that pushes p onto the stack as an anonymous function. This construct is called *quotation*. Higher-order functions can be modelled this way.

Functions with n arguments take n topmost values from the stack, compute the results and push them onto the stack. Constants are just nullary functions that push the value of the constant onto the stack.

Concatenative programming has several interesting properties:

- Programs do not need parentheses to specify operation priority, the programs are written in the Reverse Polish Notation (also called Postfix Notation).
- Programs are completely point-free, no variables are used anywhere.
- Function can naturally return multiple values just by pushing them all onto the stack.
- Expressive power of the language can be partially regulated by choice of built-in combinators e.g. by allowing only linear recursion combinator, preventing other kinds of recursion [18].
- The syntax is very minimalistic and the semantics is reasonably simple.
- Both syntax and semantics form a monoid. The syntactic monoid is the set of strings with concatenation (associative) and empty string as the neutral element. The semantic monoid is the set of functions from a stack to a stack with function composition (associative) and the identity function as the neutral element. Compilation can be seen as homomorphism from the syntactic monoid to the semantic monoid [46].

  In presence of static typing, the semantic monoid turns into a semantic category as the totality requirement is not satisfied. That is due to the fact that some syntactically correct terms are being ruled out by the type system.

3.4.1 Examples

Following examples demonstrate some of the features of concatenative programs. The traditional functional equivalent is given on the right hand side of the $\equiv$ sign.

- $5\ 8\ \mathit{add} \equiv 13$ — Adding two constants.
- $[1\ \mathit{add}]\ \mathit{map} \equiv \mathit{map}\ (\lambda x \mapsto x + 1)$ — List-based map function.
- $[5\ \mathit{mul}]\ \mathit{dip}$ — Combinator *dip* applies the quotation one level under the top of the stack. If applied to the stack $[7, 1, 3, 2\rangle$, the program will transform it to $[7, 1, 15, 2\rangle$ (top of the stack is on the right, *mul* is multiplication).
- $\mathit{dup}\ \mathit{mul} \equiv \lambda x \mapsto x^2$ — The item on the stack is duplicated and then multiplied with itself, implementing the square function.

- *qot dip* — Swaps two topmost items on the stack. The *qot* combinator adds one level of quotation to the topmost item x, turning it into $[x]$. It is then unquoted underneath the top by the *dip* combinator.

- [*dup mul*] *map sum sqrt* — Computes the length of a vector. Notice that there is a subprogram equivalent to the previous example. The absence of variables makes subprograms somewhat independent on the surroundings turning some simple refactoring tasks into just search-and-replace.

3.4.2 Typing Concatenative Programs

There are ways to introduce types to concatenative programs [10] as has been done for the Cat programming language [9]. Each function reads just a few top-most elements of the stack. The rest of the stack can contain an arbitrary number of elements. Hence, functions need to be polymorphic in the rest of the stack. This is modelled using row polymorphism (section 3.3.3). Otherwise, the type system is based on similar concepts as the simply typed lambda calculus.

Chapter 4

Technologies for Compiler Design

This chapter reviews existing techniques, technologies and tools that support design process and (semi-)automatic implementation of compilers. Compilation process can be divided into several phases. Various tools aim to ease development of different phases of the compilation.

Traditional compiler design can be roughly divided into two main parts [2]:

1. Front-end
   (a) Lexical analysis
   (b) Syntax analysis
   (c) Semantic analysis
2. Back-end
   (a) Conversion to intermediate representation
   (b) Optimizations
   (c) Target code emission

The task of the front-end is to turn the textual representation of the language into compiler internal representation. As syntax analysis takes place in the front-end, it is the central subcomponent of focus for this project. The back-end converts the intermediate representation into native machine code. The intermediate representation undergoes several transformations such as optimization passes, before register allocation and target code emission itself is performed.

When picking appropriate tools for for language design and implementation, unusual special features of the language have to be kept in mind. Syntax extensibility definitely is such a feature.

4.1 Parser generators

The idea of parser generators is to generate source code implementing syntax analysis from a grammar definition usually encoded in EBNF form. The output is a static parser with hard-coded constructs.

Yacc is a parser generator for C/C++. The abbreviation stands for *Yet Another Compiler Compiler* [37]. The output is a LALR parser implemented in C. Yacc requires an external lexical analysis routine such as hand-coded state machine or generated by a similar tool for lexical analysis, e.g. `lex`. As C/C++ is not the implementation language of choice, `yacc` is mentioned here just as one of the most prominent examples of parser generators.

Happy is a parser generator for Haskell [24] and as such is more relevant for this project than `yacc`. It features integrated lexical analysis, integration with arbitrary monad, and support for attribute grammars [19].

Evaluation

One way to use a parser generator for a language with user-defined syntax would be to parse just a skeleton of the language, leaving some parts tokenized but not parsed. The unparsed parts would be processed during the second pass utilising user-provided grammar rules. This approach possesses several limitations such as the inability to extend the syntax arbitrarily. Overall, it seems to be better to employ a more dynamic tool than a parser generator.

4.2 Parsec

Parsec [21] is a parser combinator library for Haskell. It is a library providing a set of simple parsers and a set of combinators to compose simpler parsers into more complex ones. It is designed as a monad transformer which allows Parsec to trigger other effects, such as interaction with outer world via IO or logging of compilation process, while parsing. Parsec is predictive by default causing the order of operands of the choice operator to be significant. Parsec is primarily designed as a scanner-less parser: it does not distinguish between lexical and syntactic analysis.

Example

Parsers can be expressed in monadic or applicative form (section 3.2, page 21). Listings 4.1 and 4.2 show the language of nested parentheses in Parsec expressed in applicative and monadic style respectively. Both parsers return maximal depth of parentheses. Applicative style is more compact and arguably more readable while it has less expressive power.

```
parens = max <$> (inc <$> char '(' <*> parens <*> char ')') <*> parens
     <|> pure 0
  where inc _p1 depth _p2 = depth + 1
```

Listing 4.1: Parenthesis language in applicative style in Parsec

```
parens = par <|> return 0
  where par = do
    char '('
    a <- parens
    char ')'
    b <- parens
    return $ max (a + 1) b
```

Listing 4.2: Parenthesis language in monadic style in Parsec

Parsec also features user-defined state, which can be used to keep track of a symbol table. It could be also employed to store user-defined syntax rules and extract them on-demand via monadic binding.

Evaluation

Monadic parsing offers enough expressive power to build parsers dynamically as the parser to be used for consuming subsequent input can depend on results from parsing preceding input. Listing 4.3 shows a parser that uses the `parseGrammarDefinition` parser to dynamically generate a parser that is used to parse a subsequent chunk of code. The example is given both in the `do`-notation and in combinatoric style.

```
import Text.Parsec

parseGrammarDefinition :: Stream s m t => ParsecT s u m (ParsecT s u m a)
parseGrammarDefinition = undefined  -- we leave this unspecified for now

loadGrammarAndRun = do
    dynamicParser <- parseGrammarDefinition   -- load a grammar
    dynamicParser                             -- and run its parser

loadGrammarAndRun' = parseGrammarDefinition >>= id
```

Listing 4.3: Metaparsing with Parsec

4.3 Attribute Grammar Compiler

The *Utrecht University Attribute Grammar Compiler*[1] (UUAGC) [43] is a preprocessor that augments bare Haskell with support for Attribute Grammars [31] which were described in section 3.1.2. UUAGC has been used to implement a brand new Haskell compiler UHC[2], a compiler of the Tiger language and it has been also used to bootstrap itself.

UUAGC introduces a few new keywords, most notably `attr` which declares an attribute annotated with the its type (the most common types are `syn` for synthesised attributes and `inh` for inherited ones), and `sem` which defines the semantics of an attribute.

Example

```
data Tree
    | Node  left, right :: Tree
    | Tip   value       :: Int

attr Tree
    syn sum :: Int

sem Tree
    | Node lhs.sum = @left.sum + @right.sum
    | Tip  lhs.sum = @value
```

Listing 4.4: UUAGC example

[1] `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`

[2] `http://www.cs.uu.nl/wiki/bin/view/Ehc/WebHome`

The example in listings 4.4 shows how to use UUAGC to annotate nodes of a tree with a synthesised attribute representing the sum of all its descendants. The definition is very straightforward. To mix attribute grammars with plain Haskell, the Haskell code has to be enclosed in curly braces and special generated functions have to be used to extract the attributes bound to particular data structures

Evaluation

UUAGC is a very useful tool for language and compiler implementation. The notion of attribute grammars matches the problem domain very well. It is particularly well suited for doing semantic analyses over the abstract syntax tree of a program. On the other hand, the preprocessing stage complicates the build process a little bit.

While it is certainly possible to express syntax extensions with attribute grammars due to their ability to encode any Turing-complete computation [31], it does not seem to be an overwhelmingly elegant solution.

4.4 Code Pretty Printers

Many compilers produce assembly language or other textual code rather than target machine binary. In order for it to be easier to inspect by humans, it is desirable that it is nicely formatted. Several software tools have been developed to perform the task.

Standard Haskell distribution, the Haskell Platform[3], comes with a pretty-printer library designed by John Hughes and improved by Simon Peyton Jones [17]. The library exports many combinators to compose textual layouts and render them into a character string in a number of different ways. Limiting the output width, indentation, enclosing in braces and parentheses are all well supported.

Example

Although the example pretty-printer definition in listing 4.5 is much larger than corresponding output in listing 4.6, it has defined several re-usable functions and demonstrates document composability the library provides.

```
fhead ret fname args = text ret <+> text fname <> fargs args
fargs args = parens . sep $ punctuate comma args'
    where args' = [text ty <+> text name | (ty, name) <- args]
fcall fname args = text fname <> parens args' <> semi
    where args' = sep $ punctuate comma args
fbody body = lbrace $+$ nest 4 body $+$ rbrace

main = putStrLn . show $
    fhead "int" "main" [("int", "argc"), ("char**", "argv")]
    $+$ fbody (fcall "puts" [text "argv" <> brackets (int 0)])
```

Listing 4.5: Sample Pretty-Printer definition

```
int main(int argc, char** argv)
{
    puts(argv[0]);
}
```

Listing 4.6: Sample Pretty-Printer output

[3]http://www.haskell.org/platform/

Evaluation

If a compiler has a textual output, using a pretty printer can save many lines of code dealing with low-level string manipulation while maintaining a nice and readable output formatting. Pretty-printer coming with the Haskell Platform is a well-designed piece of software that fits our use case of generating LLVM IR well.

For alternative pretty-printing library designs, see [3], [47] and [43, chap. 4].

4.5 libgc

The *gc* library[4] provides a conservative garbage collector. The conservativeness property implies it can be safely used with languages that have not been designed with garbage collecting in mind such as C and C++ [5]. In most cases, switching from manual deallocation to a `libgc`-based one is just a matter replacing all `malloc` calls with calls to `GC_malloc` and linking `libgc` to the binary.

It can also be used for automatic deallocation of objects in runtime environment of other languages when linked as a library.

Evaluation

Automatic garbage collection is one of the most common memory management schemes in programming languages with high degree of abstraction. Plug-and-play technology for automatic memory deallocation is an extremely useful tool when implementing such a high-level language.

4.6 LLVM Compiler Infrastructure

LLVM compiler infrastructure [20] is a language agnostic set of libraries and tools suitable for use as a back-end of a programming language. It includes many pre-made optimization passes, register allocator and binary generation for many target platforms.

The Intermediate Representation (IR) has several features that make it simple for it to be generated automatically from an abstract syntax tree. Main characteristics of the IR include:

- Pre-defined instruction set. Instruction invocations are grouped into basic blocks, each block is introduced by a label and terminated by a terminating instruction (such as jump or function return).
- Unlike most assembly languages, LLVM IR has an explicit notion of a function.
- Based on register machine with an infinite supply of read-only registers and unlimited read/write memory. Registers are statically typed though unsafe casts are possible.
- Each register is assigned only once adhering to the Static Single Assignment (SSA) form [6] simplifying the code analysis. When a register shall be assigned a value from either of several different variables coming from different basic blocks, the special `phi` instruction has to be used.

[4]`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

Example

The example in listing 4.7 shows hand-coded LLVM IR for a program computing and printing $1 + n^2$ for $0 \leq n < 10$. It shows many features of LLVM IR including declarations of external functions, function definitions, constructing loops with branching and `phi` instructions, and loading global data from memory.

```
declare double @llvm.sqrt.f64(double %Val)
declare i32 @printf(i8*, ...);
@.fmt = private unnamed_addr constant [4 x i8] c"%f\0A\00"

define double @len(double %x, double %y) {
        %x2 = fmul double %x, %x
        %y2 = fmul double %y, %y
        %len2 = fadd double %x2, %y2
        %len = call double @llvm.sqrt.f64(double %len2)
        ret double %len
}

define i32 @main(i32 %argc, i32** %argv) {
    entry:
        br label %loop
    loop:
        %n = phi i32 [ 0, %entry ], [ %nnext, %body ]
        %le10 = icmp slt i32 %n, 10
        br i1 %le10, label %body, label %exit
    body:
        %nd = sitofp i32 %n to double
        %z = call double @len(double %nd, double 1.0)
        %fmt = getelementptr [4 x i8]* @.fmt, i64 0, i64 0
        call i32 (i8*, ...)* @printf(i8* %fmt, double %z)
        %nnext = add i32 %n, 1
        br label %loop
    exit:
        ret i32 0
}
```

Listing 4.7: LLVM IR example

Evaluation

LLVM is an advanced and stable backend used in several production-ready compilers. It offers many optimization passes and code transformations as well as some sanity checks of its input. It abstracts the designer of a natively-compiled programming language from many details of the target platform such as concrete instruction set or register file layout. It seems to be a suitable part of the backend of our toy language.

Chapter 5

Language Design

In this chapter, we propose the Experimental Extensible Language, or **EEL**. The design consists of two main parts:

1. Minimalistic declarative core language
2. Way to extend and modify the language syntax

Syntax extensions are defined by means of mapping the newly introduced constructs to the core language. The core representation is in turn type-checked and ultimately transformed to a native binary executable via series of intermediate steps.

5.1 Design Goals

First several criteria will be set up. These have to be kept in mind while designing and implementing the language itself. The main focus of the language being proposed is practical research on syntax extensibility. Following criteria either immediately follow from that main idea or narrow down the specification. Some of them refer to concrete implementation rather than design of the language itself.

Syntax Extensibility

The central focus of EEL is the ability to let the user define new syntax. The syntax extension mechanism shall be:

1. *Easy to understand*: The mechanism of defining new syntax shall be based on (a modification of) Context-Free Grammars with semantic annotations attached to the rules. Context-Free Grammars are a part of common knowledge of most programmers so the syntax extension system should not be too hard to grasp.
2. *User-friendly*: Semantic errors, such as type errors, encountered during core representation analysis shall be propagated back and point to a specific position in the original source code.
3. *Meta-extensible*: The syntax for definition of syntax extensions shall itself be extensible.

Core Minimalism

The core language acts as a target language for syntax extension translation mechanism. It is a declarative, functional in nature, and statically-typed programming language.

The core specification shall be stripped down to the bare minimum:

1. *In terms of syntax*: Any additional useful syntactic constructs can be introduced by the syntax extension mechanism. Concatenative languages seem to be a good match as they have only three syntactic constructs: functions, function composition, and quotations.

2. *In terms of semantics*: A simple but demonstrative type system inspired by simply-typed lambda calculus with a few extensions. Several built-in data types.

3. *In terms of number of built-in functions*: A minimal set of combinators that form a base for Turing-complete computations together with a minimal set of functions manipulating built-in data types.

Static Type Safety

EEL shall feature static strong type discipline. The same discipline must apply for all the user-defined syntax extensions as well. Newly introduced constructs and dialects must exhibit at least the same degree of safety guarantees as the core language.

Transparency

Being an experimental language, it is desirable to be able to examine the internals of its compiler. It should be possible to print out the outputs of various passes of the compilation process as a human-readable text file. The philosophy is similar to LLVM, where the intermediate representation can be turned into a human-readable format so the effects of various optimization passes and other transformations can be easily assessed by a human investigator. This is design objective of the compiler rather than the language itself.

Other Features

Other nice features that are not a big concern with respect to the primary research goal on syntax extensibility include:

1. *Native Compilation*: The result of the compilation process is a native binary that can be executed on real hardware.

2. *Library Design*: Make various translation stages separable so that they can be reused in external utilities or IDEs.

These features are also a matter of concrete compiler design and implementation rather than being intrinsic to the language.

Experimentation

Finally, EEL is a research prototype language and its primary goal is experimentation. This implies several non-goals:

1. *Simple type system*: The type system of EEL does not aim to be all-expressive. It may be complex just enough to demonstrate the typing preservation of the syntax extensions on several not overly complicated examples but not more. Advanced features such as type classes or subtyping will be left out, user-defined types might be disallowed.

2. *Limited interaction with outer environment*: To demonstrate its capabilities, the language does not need to be full-featured in terms of interaction with the operating system. Basic *stdin/stdout* console IO should be just enough.

3. *Other common features*: Some features that are common in practical programming languages may be very limited or not present at all. These include module system, explicit symbol name spaces etc.

4. *Limited paradigm support*: Although a support for some language paradigms could be added by a well thought-out mix of higher-order functions and syntax extensions, the type system can be a limiting factor here.

5. *Performance*: Performance of either the compiler itself or the generated executable is not considered very important.

The points mentioned above just limit the scope of this project and they can be revised in the future. If the language was intended to become a practically useful one, addressing these issues would be more than appropriate.

5.2 The Core

EEL Core is a stack-based concatenative programming language. There are good reasons to use this paradigm minimalism being the most important one. Stack code is also one of common intermediate code representations used in compilers and interpreters. Using a similar language as a target platform for translation of high-level user-defined syntactic constructs seems to be a good match.

The core uses a strict evaluation strategy making the runtime simpler and eventual possible debugging easier. It also makes side effects hurt less when mixed with the pure code. Subprograms are executed in order. EEL is not a pure functional language. One can freely mix pure and side-effecting, the type system does not enforce any kind of isolation between the two.

$$\begin{array}{lcll} F & ::= & \mathit{name} & \text{invocation of function } \mathit{name} \\ & | & F\ F & \text{function composition} \\ & | & [\ F\ ] & \text{anonymous function (quotation)} \\ & | & \varepsilon & \text{empty function (identity)} \end{array}$$

Figure 5.1: EEL Core abstract syntax

Syntax of the Core matches the common syntax of other concatenative languages as described in section 3.4 on page 24 and summarized in figure 5.1.

EEL Core pushes the idea of concatenative programming to the limit. Even function definitions use postfix notation. This concept makes it possible to treat runtime expressions and function definitions uniformly from the point of view of user-defined syntactic extensions. The square function would be defined as follows: `[ dup mul ] "square" def`. Here, the `def` is just a function that defines a new named function from a quotation. The `def` function is only available at the compile time.

5.2.1 Type System

EEL uses polymorphic type system with fake row variables heavily inspired by [9, 10]. All type variables are treated as if they were universally quantified, existentials are missing. Available types are summarized in figure 5.2.

$$\begin{array}{rcll}
\tau & ::= & \mathrm{int} \mid \mathrm{char} \mid \mathrm{float} & \text{built-in atomic types} \\
 & \mid & \mathrm{U} & \text{unit type} \\
 & \mid & \tau + \tau & \text{sum type (tagged union)} \\
 & \mid & \tau \times \tau & \text{product type (ordered pair)} \\
 & \mid & \tau \to \tau & \text{exponential type (function)} \\
 & \mid & [\tau] & \text{list type} \\
 & \mid & var & \text{type variable}
\end{array}$$

Figure 5.2: EEL Core data types summary

EEL does not support any form of user-defined types or type annotations, types of all terms are inferred by the compiler. Recursion patterns are supported only via the fixed-point built-in combinator `fix`, list is the only type exhibiting structural recursion. Any other term that would require recursive types to be typeable, such as an attempt to perform a self-application `dup apply`, will be rejected by the type checker.

The stack is modelled as a series of nested pairs. The left component of the most deeply nested pair is the fake row variable. For example function `add` takes two `int`s from the stack and returns one integer result. Its type is $((r \times \mathrm{int}) \times \mathrm{int}) \to (r \times \mathrm{int})$. Not supporting user type annotations simplifies matters a lot as otherwise the fake row variable r would have to be of a different kind from ordinary type variables.

The lack of recursive types forces us to use dynamic typing at the compile time. All safety guarantees for generated target code remain intact as all functions to be emitted by the backend are type-checked statically. Summary of the typing rules for basic syntactic categories is shown in figure 5.3.

$$\frac{}{\Gamma \vdash \varepsilon : \tau \xrightarrow{\pi} \tau}\ \text{EPS} \qquad \frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{FUNC} \qquad \frac{\Gamma \vdash t : \tau_1 \quad \mathrm{fresh}(\tau_2) \quad \mathrm{fresh}(\pi)}{\Gamma \vdash [t] : \tau_2 \xrightarrow{\pi} \tau_2 \times \tau_1}\ \text{QUOT}$$

$$\frac{\Gamma \vdash f : \tau_1 \xrightarrow{\pi_1} \tau_2 \quad \Gamma \vdash g : \tau_2 \xrightarrow{\pi_2} \tau_3 \quad \pi = \pi_1 \wedge \pi_2}{\Gamma \vdash f\ g : \tau_1 \xrightarrow{\pi} \tau_3}\ \text{COMP}$$

Figure 5.3: EEL typing rules

Phase Distinction As it has been already mentioned, some functions can be performed during the compile time only. Such functions are marked with a plus sign (+). On the other hand, there are functions that can be performed only in the run time, namely functions executing I/O operations, marked with a minus sign (−). Furthermore, to provide syntactic extensions, there are a few primitive parsers marked with an asterisk ($*$).

The phase any function operates in is captured by the type. Type of function from τ_1 to τ_2 operating in phase π is denoted $\tau_1 \xrightarrow{\pi} \tau_2$. Functions can be polymorphic in phase. Unification of phases is resolved according to partial ordering among phase annotations which is shown in figure 5.4. The top represents phase-polymorphic functions, the bottom is a type error. The phase annotations can be viewed as a simple effect system [36, chap. 3].

$$\begin{tikzcd}
 & \top \arrow[dl, dash] \arrow[dr, dash] & \\
+ \arrow[d, dash] & & - \arrow[ddl, dash] \\
* \arrow[dr, dash] & & \\
 & \bot &
\end{tikzcd}$$

Figure 5.4: Partial ordering on phase types

5.3 Extension Mechanism

The mechanism of user-defined syntactic extensions in EEL is inspired by monadic parser combinator libraries. Separation of parsers from the rest of the functions is enforced by phase types, not an explicit notion of a monad. When it comes to the expressive power, the two are equivalent.

The Core offers a function that introduces a new rule: `defrule`. It takes a string name of the non-terminal to extend, an integer priority, and the right-hand-side of the rule. The right-hand-side is a function composed of primitive or user-defined parsing combinators or invocations of parsers for other non-terminals.

The priority together with ordered choice is our way to deal with ambiguity. The priority levels are tried in descending order. If one priority level fails, the parser backtracks and tries the next lower priority level. If all priority levels fail, the parse of the non-terminal definitely fails. Rules with the same priority do not backtrack: if one consumes any output, the parser is 'committed' to the alternative. If there are several matching rules of the same priority, the one that has been defined last applies. This makes it possible to override parts of the syntax defined by previously introduced rules. The parsing algorithm itself is captured in figure 5.5

This two-stage disambiguation mechanism of a set of parsers with backtracking each composed from a set of parsers without backtracking has been chosen because it fits well the implementation with Parsec. In order to provide more freedom in user-defined grammar extensions, the EEL parser is a scanner-less one. Tokens are individual characters of the input sequence. Parsec intended to be used as a scanner-less parser too, so with this respect it fits our needs well.

EEL supports special 'dotted' non-terminals. Non terminals starting with a dot are used to parse files with the matching extension. For example, if the EEL compiler is about

$parse(N)$:

1. Find all parsers for non-terminal N and group them by priority i. Let R_N be the set of parsers (rules) for non-terminal N. $P_i = \{p \in R_N \,|\, priority(p) = i\}, i \in \mathbb{Z}$.
2. For each i in descending order:
    - For each $p \in P_i$ in the reverse order to the order of the definition of the parser (most recently defined comes first):
        - If p matches any input (single character is enough):
            (a) If p fails, continue with the next priority group P_{i-1}.
            (b) If p succeeds, return its result.
3. If none of the parsers succeeds, parsing of N fails

Figure 5.5: EEL parsing algorithm overview

to process a file named `hello.xml`, it will use the non-terminal with name `.xml` to parse the file.

5.3.1 Primitive Parsers

In order to be able to build production rules, the core has to offer primitive parsers which can be used to build more complicated ones. The minimal set of such parsers is defined by two built-ins:

- `ppchar` Takes a predicate which is a function from char to bool. If the next character in sequence matches the predicate, it is consumed and parsing proceeds. Otherwise, the primitive parser fails.
- `ppfail` Parser that always fails with given error message. It can be used to check advanced properties of the input such as checking whether opening and closing tag names in a XML fragment match.

5.4 Built-in Functions and Standard Library

The list of all built-in functions is available in the appendix A. The built-ins are relatively low-level, so they are used to build up a standard library of other often used functions called *prelude.eel*. The standard library is automatically available in all translation units unless the user specifies otherwise.

Chapter 6

Implementation

Prototype compiler of EEL has been implemented in Haskell with Parsec used to build the parser on-the-fly and subsequently consume input code. In this chapter, several interesting implementation details are discussed.

6.1 Modules

The source tree is broken down into several modules, each having a few sub-modules. Here, the top level modules of the hierarchy are described.

Parser The parser module uses Parsec as its underlying library. User-supplied state is defined here. It consists of a symbol table, a grammar rule definitions table, and current evaluation stack for the compile-time interpreter. Several convenience functions for querying and manipulating the state are provided.

Definition of the Core syntax lies in this module too. In addition to that, representation of user-defined syntax rules and an algorithm building a parser from the rules is present.

The top-level *Parser* module takes care of bootstrapping the initial state of the parser and calls semantic checks after it is done. It is a convenient interface for building different EEL-based tools.

Sema This module defines Abstract Syntax Tree representation of EEL functions. Representation of EEL data types and type inference algorithms live in this module.

Builtins The *Builtins* module enumerates all the built-in functions along with their types. The *Eval* sub-module implements the compile-time interpreter for the EEL built-ins. Much of the heavy lifting is done here.

Backend The backend consists of three parts. The first one is a somewhat generic pretty-printer for the text representation of LLVM IR based on the `Text.PrettyPrint` module that comes with standard Haskell distribution.

The second one generates the preamble: a piece of LLVM code that is the same for all EEL programs. The preamble contains external function definitions, a function to convert a C string to an EEL string, and trampoline functions for quotations and closures. There is also C-based `main` function that converts command-line arguments to the list of EEL strings, initializes the stack, runs the EEL `main` function, extracts the result value and returns it as an exit code of the program.

Finally, the third part walks the AST and generates LLVM IR code for encountered syntactic categories and built-in functions.

Main The main module contains a compiler driver, a read-evaluate-print interactive interpreter, and a command-line option parser. This module is described in more detail in section 6.3.

6.2 Runtime Organization

Runtime environment is kept as simple as possible because code generation is not our research objective. EEL stack is represented as a simple linked list with a pointer-sized data field and a next pointer. Functions pass over a pointer to the top of the stack.

Atomic types (floats, integers, chars, and units) are stored unboxed. Compound types (sums, products, lists, functions) are represented as pointers to actual values (boxed [39]). In case of sums, it is the tag-data pair, lists are represented in the same way as stacks. Boxed representation greatly simplifies code generation for polymorphic functions.

Anonymous functions (quotations) in the code are emitted as proper LLVM functions with an auto-generated name. Functions (quotations) on the stack are represented using triples with two data fields and a function pointer to be executed when the quotation is executed (unquoted). When the quotation is about to be executed, the two data fields together with current stack are passed to the function pointer stored with the quotation. The function pointer can point to three different functions:

- `@eelrun.func` — This trampoline treats the first data field as a pointer to function and applies it to the current stack. This type of quotation is generated if it appears in the code as an anonymous function.

- `@eelrun.data` — This trampoline treats the first data field as an data item to be pushed onto the stack. This type of quotation is introduced by the *qot* built-in that adds a layer of quotation to the topmost item on the stack.

- `@eelrun.comp` — This trampoline implements function composition as introduced by the *cat* built-in function. It treats the two data fields as another quotations and executes them in sequence.

All boxed values and EEL stack frames are allocated on a garbage-collected heap managed by `libgc`[1].

Generated LLVM code is not very nice mainly due to frequent typecasts. That is because values of all data types are represented uniformly. Furthermore, LLVM in its recent versions lacks unions (untagged sum types). The names of local variables (registers) generated by the code generator contain a hint of the semantics of the variables. For example variables with a name starting with `%stk` are all pointers to the top of the stack.

Generated code is not very efficient and it is quite a nice garbage-collector stress test because of the constant stack shuffling. All data in EEL are immutable, so the SSA form is generated directly without using stack allocation instructions.

[1]`http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

6.3 Command Line Compiler Driver

EEL compiler driver is operated from the command line. First, it loads the input files specified on the command line and passes them to the EEL compiler implementation. The resultant LLVM code is saved into a temporary file which is in turn presented to the LLVM compiler binary `llc`. The result is a temporary assembly file, which is passed to `gcc`. We use `gcc` as both, assembly language compiler and linker. Three additional libraries are linked to the resulting binary:

- `libc` — C standard library provides I/O operations
- `libm` — C mathematical library implements some built-ins ($sin, pow, \ldots$)
- `libgc` — conservative garbage collector

Overview of the whole process can be seen in figure 6.1. The compiler driver can be instructed to keep any of the temporary files generated during the compilation process via command line options. For the full list of available options, see appendix B.

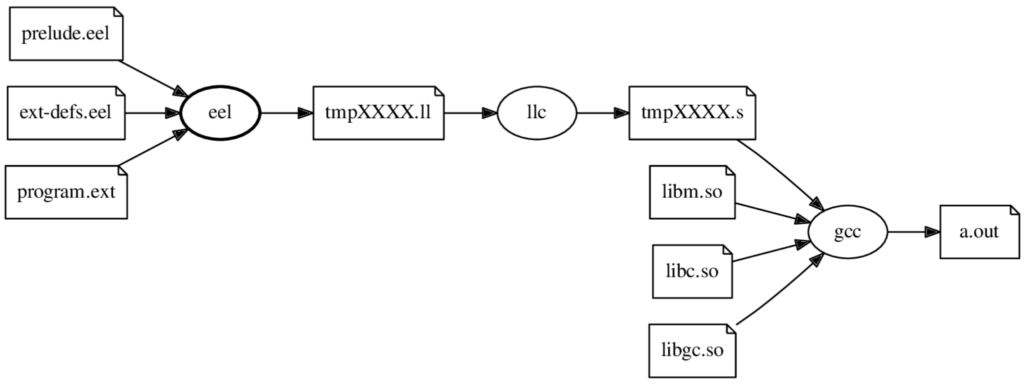

Figure 6.1: Overview of the EEL compiler driver pipeline

Interactive interpreter EEL also provides an interactive interpreter. It is invoked by specifying the `-i` option on the command line. The list of commands implemented by the interactive interpreter is in appendix B as well.

Chapter 7

Examples

In this chapter, syntactic extensibility of EEL is demonstrated on several simple examples. Methods for adjusting the language syntax for specific needs are presented. Towards the end of the chapter, it is shown how to use the extensibility capabilities to process external domain specific languages.

7.1 Bootstrapping EEL

First, we have to deal with the fact that by default, the core EEL language is not extensible. To provide this feature to the core, its syntax has to be bootstrapped. In other words, grammar definition of EEL core itself will be provided in terms of a syntactic extension. Files ending with `.eeel` (notice the extra '`e`') will be parsed using the bootstrapped syntactic analyser. Bootstrapped syntax allows new syntactic extensions to be defined on-the-fly, they take effect right after the definition.

Grammar Definition

The grammar definition is mostly equivalent to the built-in one. One difference is that the bootstrapped EEL syntax supports negative integer constants out of the box. When it comes to extending the syntax even further, it is necessary to know the names and meanings of the non-terminals. The most important ones defined by the bootstrap module follow:

- `skip1` — Used for skipping white space and comments. Extend this non-terminal to add a new kind of comment.
- `atom` — Used to parse atomic functions. Can be used to introduce new syntax for any kind of function ranging from nicer constants or specifying parts of a computation in a different format (e.g. a regular expression) to a completely different approach to a part of a computation (e.g. using a monadic composition instead of the plain function composition). The result of this parser shall be a quotation with the resulting function.
- `prog` — Function composed from atomic functions.
- `escseq` — This non-terminal can be used to specify new escape sequences that appear after a backslash inside a string literal.

The bootstrap grammar definition lives in `lib/boot.eel`.

Demonstration

As EEL compiler does not have any clever module system, files with all the required definitions have to be specified on the command line explicitly:

```
./eel -o output_binary lib/boot.eel samples/test.eeel
```

The `samples/test.eeel` file is a simple EEL core source with a few functions. Due to the file extension, it is recognised as a bootstrapped EEL core file and as such it is parsed using a dynamically built parser. It also contains a simple grammar extension so it already demonstrates something that cannot be done with just plain core.

7.2 Structured EEL

Although the fact that function definitions are using the same syntax as everything else has a nice feel of regularity, it is not very natural nor common in programming languages. Especially having function name specified after its body might look quite unnatural. We present a grammar extension that changes the syntax of top-level structure of the source code file. We call it "Structural EEL" and the sources using this extension can be recognised by the file extension `.seel`.

The function definition syntax changes as follows:

`define` *function_name* { *function_body* }

Grammar Definition

The grammar builds on top of the bootstrap. It introduces a new top-level non-terminal that rewrites into series of definitions represented by the `definition` non-terminal. Introduction of the `define` construct itself is in listing 7.1. The `_eelkey` function is the parser for EEL keywords: any pre-determined sequence of characters followed by white space or a non-alphanumeric character. The `tok` does the same for other kinds of tokens. Function `_prog` is a shortcut for `"prog" invoke` which just invokes regular EEL core function composition parser from the bootstrap.

```
# 'define' construct: function definition
[
    "define" _eelkey  # 'define' keyword
    _symbol _skip     # function name
    "{" _tok          # opening brace
        _prog         # function body
    "}" _tok          # closing brace
    swap def          # define the function
] "definition" 0 defrule
```

Listing 7.1: Alternative function definition syntax

Demonstration

The demonstration in listing 7.2 shows that even top-level constructs can be replaced and/or extended in EEL.

```
define vector_length { [ square ] lmap sum }
# core way to do this: [ [ square ] lmap sum ] "vector_length" def
```

Listing 7.2: Alternative function definition use

7.3 Meta Extensibility

A function definition is not the only top-level construct in EEL. The other one are extensions to the syntax of the language. It would be nice to have a convenient syntax for them too. Here, we introduce a syntactic extension for writing syntactic extensions.

The new top-level clause `grammar` contains a list of production rules. The left hand side of the rules specify a non-terminal to rewrite. The right hand side of the rules is a sequence of elements specifying what is the non-terminal rewritten to. Each element is of one of three types:

1. Invocation of another non-terminal simply by spelling out its name.
2. Literate keyword specified in double quotes. Keywords are parsed as tokens, the trailing white space is skipped automatically. This behaviour can be suppressed using the caret symbol.
3. Semantic action enclosed in square brackets as a piece of EEL core program.

Grammar Definition

In abstract terms, the syntactic extension for extending grammar is specified in figure 7.1. The actual implementation in EEL is quite a bit longer.

D	$::=$	grammar $\{\ G\ \}$	syntactic extension
G	$::=$	$R\ G \mid \varepsilon$	list of rules
R	$::=$	$name \to T$	non-terminal for *name*
T	$::=$	$name\ T$	non-terminal invocation
	$\mid$	"*string*" T	literate keyword *string*
	$\mid$	$[\ P\ ]\ T$	semantic action, P is the `prog` non-terminal from the bootstrap
	$\mid$	;	end of rule

Figure 7.1: EEL syntactic extensions metalanguage

Demonstration

This extension is a part of the Structured EEL module `seel.eel`. EEL core lacks a convenient way to describe list constants. Listing 7.3 shows a way to introduce them. `nil` and `cons` are the list constructors. Non-terminal `atom` expects a function, not a literal list. Hence the call to `qot`.

```
grammar {
    # list constants
    atom      --> "$[" listcont "]" [ qot ] ;
    listcont  --> [ nil ] ;                            # empty list
    listcont  --> listentry listcont1 [ cons ] ;       # non-empty
    listcont1 --> [ nil ] ;                            # end of list
    listcont1 --> "," listentry listcont1 [ cons ] ;   # list continued
    listentry --> func [ i ] ;                         # list element
}
# the test below results in $[6, -9, 0]
define test { $[ $[1, 5], $[-8, 1, -2], $[] ] [ sum ] lmap }
```

Listing 7.3: List constants extension

7.4 Encoding Brainfuck

Brainfuck[1] is a minimalistic esoteric programming language designed for amusement rather than serious computing. It has been chosen as an example because it is a simple but self-contained Turing-complete programming language. Quite different from EEL yet its grammar definition and representation in terms of EEL core is straightforward.

Brainfuck interprets only eight characters from the whole ASCII range. Its runtime environment is an array of bytes together with a pointer pointing to the 'current' cell in the array. The instructions are '`+`' and '`-`' for incrementing/decrementing the current cell, '`<`' and '`>`' for shifting the pointer to the left/right, '`.`' writes a character to the output, '`,`' reads a character from the input, and `[]` is a simple looping construct. Any other character is treated as a comment.

In EEL we represent the Brainfuck memory array as a zipper [16]. The array is split into two parts represented by two lists. One is the region before the pointer and the other is the region past the pointer. The lists are grown on demand.

Grammar Definition Remarks

To define the translation from Brainfuck to EEL core, we define Brainfuck instructions as EEL functions first. Defining the rules is straightforward afterwards. The top-level rule for parsing Brainfuck source code files `.bf` generates the main function that first initialises the zipper, then generates the actual Brainfuck code, and finally extracts the value of the cell where the execution has suspended and uses it as an exit code of the program.

An extension to the bootstrapped EEL core has been defined as well. When loaded, pieces of Brainfuck code can be used for some sub-computations in a larger EEL program.

7.5 DSL for Cellular Automata

In this section, we present a simple Domain Specific Language for specifying cellular automata [48]. We restrict ourselves to one-dimensional Cellular Automata. Cellular Automaton is a regular grid, or in our 1D case a vector, of cells. The next state of each cell depends on the previous state of itself and its neighbourhood. All the cells operate in locksteps, they are updated synchronously.

Specification of the DSL

We choose to represent states of the cells as a subset of ASCII character set. CA specification language is based on a sequence of pattern rules. The first matching rule applies. The special '`?`' symbol can be used as a wild-card pattern.

The rules have the following basic format: `pattern : next_state`. The pattern contains expected previous states of the current cell and its neighbourhood with the current cell being enclosed in square brackets. The format is as follows: `left[X]right`. The `X` is the previous state of the current cell, `left` and `right` is the previous state of its left-/right neighbourhood. The neighbourhood patterns can be arbitrarily long, although most "classical" cellular automata use the neighbourhood range of one.

[1]`http://esolangs.org/wiki/Brainfuck`

Grammar Definition Remarks

The grammar definition itself is quite simple, the runtime support and pattern matching the DSL translates to is much more complicated. It would be easy to extend the next-state part with an arbitrary EEL computation with previous states of some cells being passed to it. Such an extension might be convenient but it does not add any anything to the expressive power of the DSL.

Source code files of the CA DSL are recognised by the file extension of `.ca`.

Demonstration

The source distribution comes with several demo CA definitions. One of the canonical examples is the Rule 110. The important part of its definition is in listing 7.4.

To compile it, run:

```
./eel -o rule110 lib/boot.eel samples/cellular.eel samples/rule110.ca
```

The resulting binary expects two command line arguments: number of iterations of the CA to perform, and the initial state of the CA. The sample invocation can be seen in listing 7.5.

```
# rule 110 - the 0 case
?[_]_  : _
@[@]@  : _

# otherwise @
 [?]   : @
```

Listing 7.4: Rule 110 definition in the CA DSL

```
|_______________@______@|
|______________@@_____@@|
|_____________@@@____@@@|
|____________@@_@___@@_@|
|___________@@@@@__@@@@@|
|__________@@___@_@@___@|
|_________@@@__@@@@@__@@|
|________@@_@_@@___@_@@@|
|_______@@@@@@@@__@@@@_@|
|______@@______@_@@__@@@|
|_____@@@_____@@@@@_@@_@|
|____@@_@____@@___@@@@@@|
|___@@@@@___@@@__@@____@|
|__@@___@__@@_@_@@@___@@|
|_@@@__@@_@@@@@@@_@__@@@|
```

Listing 7.5: Rule 110 in operation

Chapter 8

Conclusion

Provision of syntax extensibility is not a common feature in mainstream programming languages. Some can get closer than others via greater expressiveness, ability to define new operators, specialized preprocessors or compile-time string processing. Our language EEL takes a fundamentally different approach. It is designed to support syntactic extensibility from ground up.

The EEL compiler is still an experimental prototype rather than a production-ready tool. Its primary goal is to explore the design space in the area of languages with user-extensible syntax. Therefore, the scope of this project is narrowed and the design of the core language is kept as simple as possible while keeping the possibility of extensions to the grammar of the language in mind.

8.1 Achievements

The prototype EEL compiler has proven its value for experimentation with extensible programming languages. Output of most compilation stages, ranging from AST and LLVM IR to generated assembly, is relatively easy to inspect. The interactive interpreter makes testing of various features and prototyping new functions very fast. Ability to select a different parser based on file extension makes it convenient to process external file formats using EEL.

EEL Core is already quite a nice language with a distinctive set of features even without any syntactic extensibility. The combination of the concatenative paradigm with static typing, native compilation, and uniform representation of compile-time and run-time functions with phase types is to our best knowledge very unique.

Syntactic extensions mostly work, including meta-extensions, i.e. introducing a new syntax for defining new syntactic constructs. Extensions have been tested both by adding new features to the core language and by defining simple self-contained domain-specific programming languages.

Stack-based core has been shown to be a viable option to translate higher level language constructs into. Mostly due to its simple semantics and context-independence. Unifying the representation of compile-time and run-time operations makes the core more regular. As a consequence, defining a new construct is easier and the result tends to be more reusable.

8.2 Limitations and Shortcomings

By no means do we pretend that EEL is a perfectly designed and implemented programming language, quite the contrary. Proper identification of problems with current design is crucial for success of future development.

The types of the compile-time functions are not sound. To check for errors, we rely on dynamic typing at the compile time postponing the error detection to the function execution time rather than keeping it in the function definition time. This can be partially solved by introducing proper recursive types. But not entirely because some of the functions do non-trivial things such as symbol table lookups. To assign a type to such a function, the type would have to encode the symbol table in some way. This applies only to compile-time (and parse-time) functions. All the functions that make their way to the resultant binary are type-checked properly.

Introducing the possibility to declare local variables would make the language with syntactic extensions a great deal more expressive and useful. However, it would require significant changes to the current code base.

In some cases, parsing with syntactic extensions is currently quite slow. Although some optimizations have been attempted, many layers of interpretation and excessive backtracking make the time complexity of the current parsing algorithm rather unattractive.

Also meta-data tracking such as propagating exact location of semantic errors is quite complicated with current design and the results could be better.

The last two problems can be largely attributed to the presence of full-blown monadic parsing used in production rule definitions. A bit more restricted parsing model would increase our ability to analyse grammar rules and together with a smarter parsing algorithm could yield a much better performance. Monadic parsing is still useful (if not required) for implementation of a language with extensible syntax itself.

8.3 Future Directions

In order to make EEL usable for real-world use, the obvious direction for future improvement is removal of the shortcomings mentioned in the previous section. Next step would be dropping the intentional limitations defined on page 35.

We estimate that properly typing the compile-time functions and user-defined grammar rules would require a significant amount of theoretical research. Also, formalization of the very concept of extensible (self-modifying) grammar would be nice. However in the end, it might be worth the effort.

A more practical direction would be implementing support for extensible syntax in an IDE, including syntax highlighting and code completion. This would probably require an incremental extensible parser with much more elaborate analysis of user-defined grammar rules. Design of such a parser poses a significant technical challenge.

Bibliography

[1] Introduction to Qt Quick. online, 2012. Accessed jan 2013, modified may 2012. URL: `http://qt-project.org/wiki/Introduction_to_Qt_Quick`.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1. Pearson/Addison Wesley, 2007.

[3] Pablo Azero and Doaitse Swierstra. Optimal pretty-printing combinators, 1998.

[4] Michael Barr and Charles Wells. *Category theory for computing science*, volume 10. Prentice Hall New York, 1990.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[6] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[7] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, pages 80–99. Springer, 2011.

[8] Daniel de Rauglaudre. Camlp4 – reference manual. online, 2003. Accessed jan 2013, modified sep 2003.
URL: `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`.

[9] Christopher Diggins. Cat: A typed functional stack-based language. 2007.

[10] Christopher Diggins. Simple type inference for higher-order stack-oriented languages. Technical report, 2008.

[11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, 2011.

[12] Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 symposium on Haskell symposium*, pages 149–160. ACM, 2012.

[13] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[14] Dominikus Herzberg and Tim Reichert. Concatenative programming – an overlooked paradigm in functional programming. *Proceedings of ICSOFT*, 2009, 2009.

[15] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[16] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[17] John Hughes. The design of a pretty-printing library. In *First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, UK, 1995. Springer-Verlag.

[18] Brent Kerby. The theory of concatenative combinators, 2002.

[19] Donald E. Knuth. The genesis of attribute grammars. In *Attribute Grammars and their Applications*, pages 1–12. Springer, 1990.

[20] C. Lattner. Introduction to the llvm compiler system. In *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*, 2008.

[21] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, 2001.

[22] DJP Leijen. First-class labels for extensible rows. *UU-CS*, (2004-051), 2004.

[23] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[24] Simon Marlow. Happy: The parser generator for Haskell. online, 2010. Accessed jan 2013, modified jun 2010. URL: `http://www.haskell.org/happy/`.

[25] Simon Marlow. Haskell 2010 language report, 2010. URL: `http://www.haskell.org/onlinereport/haskell2010/`.

[26] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[27] Conor McBride and Ross Paterson. Functional pearl: Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[28] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer Verlag, 2005.

[29] Ulf Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[30] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632, September 2004.

[31] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995.

[32] Sviatoslav Pestov. A survey of domain-specific languages in Factor. online, sep 2009. URL: http://factor-language.blogspot.com/2009/09/survey-of-domain-specific-languages-in.html.

[33] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: A dynamic stack-based programming language. In *Acm Sigplan Notices*, volume 45, pages 43–58. ACM, 2010.

[34] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[35] Benjamin C Pierce. *Types and programming languages.* The MIT Press, 2002.

[36] Benjamin C Pierce. *Advanced topics in types and programming languages.* The MIT Press, 2005.

[37] Eric Steven Raymond. *The Art of Unix Programming.* Addison-Wesley, 2003. URL: http://www.catb.org/esr/writings/taoup/html/.

[38] Didier Rémy. Typing record concatenation for free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–176. ACM, 1992.

[39] Zhong Shao and Andrew W Appel. *A type-based compiler for Standard ML*, volume 30. ACM, 1995.

[40] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[41] Michael Snoyman. *Developing Web Applications with Haskell and Yesod.* O'Reilly Media, Inc., 2012.

[42] Bjarne Stroustrup. C++11 – the new iso c++ standard. online, 2012. Accessed jan 2013, modified oct 2012. URL: http://www.stroustrup.com/C++11FAQ.html.

[43] S Doaitse Swierstra, Pablo R Azero Alcocer, and Joao Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206. Springer, 1999.

[44] Wouter Swierstra. Why attribute grammars matter. *The Monad.Reader*, September 2005.

[45] Eelco Visser. Program transformation with Stratego/XT. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

[46] Manfred von Thun. Mathematical foundations of Joy. online, 1994. URL: http://www.kevinalbrecht.com/code/joy-mirror/j02maf.html.

[47] Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[48] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, 2002.

Appendix A

EEL Built-in Functions

Here, the built-in functions are listed and briefly described. The type signatures contain several shortcuts. The 'maybe' type $\tau?$ stands for $(\tau + \text{unit})$, bool stands for $(\text{unit} + \text{unit})$, string stands for [char]. The product type operator $\times$ is assumed to be left-associative, the top of the stack is on the right hand side. For better readability, ρ is used to denote the fake row variables and π for phase variables.

EEL built-ins are:

1. **Fundamental combinators and stack manipulation**

   (a) $id : \rho \xrightarrow{\pi} \rho$ / The identity function.

   (b) $id2 : \rho \times \tau \times \tau \xrightarrow{\pi} \rho \times \tau \times \tau$ / Also identity function, but forces the two topmost stack items to have equal types. Can be used as a poor man's type coercion.

   (c) $dip : \rho_1 \times \tau \times (\rho_1 \xrightarrow{\pi} \rho_2) \xrightarrow{\pi} \rho_2 \times \tau$ / Run a function one level deep into the stack.

   (d) $zap : \rho \times \tau \xrightarrow{\pi} \times \tau$ / Pop the top item from the stack.

   (e) $dup : \rho \times \tau \xrightarrow{\pi} \rho \times \tau \times \tau$ / Duplicate the top item.

   (f) $qot : \rho \times \tau \xrightarrow{\pi} \rho \times (\rho_1 \xrightarrow{\pi_1} \rho_1 \times \tau)$ / Quote the topmost item.

   (g) $cat : \rho \times (\rho_1 \xrightarrow{\pi_1} \rho_2) \times (\rho_2 \xrightarrow{\pi_1} \rho_3) \xrightarrow{\pi} \rho \times (\rho_1 \xrightarrow{\pi_1} \rho_3)$ / Quotation composition.

   (h) $fix : \rho_1 \times (\rho_1 \times (\rho_1 \xrightarrow{\pi} \rho_2) \xrightarrow{\pi} \rho_2) \xrightarrow{\pi} \rho_2$ / Fixed point combinator.

   (i) $unit : \rho \xrightarrow{\pi} \rho \times \text{unit}$ / Push a unit constant onto the stack.

2. **Compound type (de-)constructors**

   (a) $pair : \rho \times \tau_1 \times \tau_2 \xrightarrow{\pi} \rho \times (\tau_1 \times \tau_2)$ / Construct a pair.

   (b) $unpair : \rho \times (\tau_1 \times \tau_2) \xrightarrow{\pi} \rho \times \tau_1 \times \tau_2$ / De-construct a pair.

   (c) $ina : \rho \times \tau_1 \xrightarrow{\pi} \rho \times (\tau_1 + \tau_2)$ / Sum type case A injection.

   (d) $inb : \rho \times \tau_2 \xrightarrow{\pi} \rho \times (\tau_1 + \tau_2)$ / Sum type case B injection.

   (e) $sel : \rho_1 \times (\tau_1 + \tau_2) \times (\rho_1 \times \tau_1 \xrightarrow{\pi} \rho_2) \times (\rho_1 \times \tau_2 \xrightarrow{\pi} \rho_2) \xrightarrow{\pi} \rho_2$ / Sum type deconstructor applies one of the two functions depending on the tag of the sum. This is the only built-in that encodes branching.

   (f) $listu : \rho \times [\tau] \xrightarrow{\pi} \rho \times (\tau \times [\tau])?$ / Unwrap a list: returns either a pair of list head and tail or nothing if the list is empty.

   (g) $listw : \rho \times (\tau \times [\tau])? \xrightarrow{\pi} \rho \times [\tau]$ / Wrap a list: take either a pair of head and tail to make a constructed list or nothing to create an empty list. Inverse of *listu*.

3. **Arithmetics**

   (a) $add, sub, mul, div : \rho \times \text{int} \times \text{int} \xrightarrow{\pi} \rho \times \text{int}$ / Binary integer functions.

   (b) $fadd, fsub, fmul, fdiv, fpow : \rho \times \text{float} \times \text{float} \xrightarrow{\pi} \rho \times \text{float}$ / Binary floating-point functions.

   (c) $fsin, flog : \rho \times \text{float} \xrightarrow{\pi} \rho \times \text{float}$ / Unary floating-point functions.

   (d) $cmp : \rho \times \text{int} \times \text{int} \xrightarrow{\pi} \rho \times \text{bool?}$, $fcmp : \rho \times \text{float} \times \text{float} \xrightarrow{\pi} \rho \times \text{bool?}$ / Comparison operations return nothing if the operands are equal, just true if the first operand is less than the second operand, and just false otherwise.

   (e) $float : \rho \times \text{int} \xrightarrow{\pi} \rho \times \text{float}$, $floor : \rho \times \text{float} \xrightarrow{\pi} \rho \times \text{int}$ / int $\leftrightarrow$ float conversion.

   (f) $char : \rho \times \text{int} \xrightarrow{\pi} \rho \times \text{char}$, $ord : \rho \times \text{char} \xrightarrow{\pi} \rho \times \text{int}$ / int $\leftrightarrow$ char conversion.

4. **Input / Output**

   (a) $putchar : \rho \times \text{char} \xrightarrow{-} \rho$ / Write a character to the standard output.

   (b) $getchar : \rho \xrightarrow{-} \rho \times \text{char?}$ / Read a character from the standard input. Returns either the character that has been read or nothing on the end-of-file or if an error occurred.

5. **Compilation-related functions**

   (a) $def : \rho \times (\rho_1 \xrightarrow{\pi} \rho_2) \times \text{string} \xrightarrow{+} \rho$ / Function definition.

   (b) $promote : \rho \times \text{string} \xrightarrow{+} \rho \times (\rho_1 \xrightarrow{\pi} \rho_2)$ / Lookup a function by a string. If the function does not exist, it is a compilation error.

   (c) $defrule : \rho \times (\rho_1 \xrightarrow{*} \rho_2) \times \text{string} \times \text{int} \xrightarrow{+} \rho$ / Define a production rule for given non-terminal with given priority.

6. **Built-in parsers**

   (a) $invoke : \rho_1 \times \text{string} \xrightarrow{*} \rho_2$ / Invoke the parser for given non-terminal.

   (b) $ppchar : \rho_1 \times (\rho_1 \times \text{char} \xrightarrow{+} \rho_2 \times \text{bool}) \xrightarrow{*} \rho_2$ / Parse a character matching given predicate.

   (c) $ppfail : \rho_1 \times \text{string} \xrightarrow{*} \rho_2$ / Parser that always fails with given error message.

   (d) $pponeof : \rho \times [\text{char}] \xrightarrow{*} \rho$ / Parse one of the characters specified or fail.

   (e) $ppnotof : \rho \times [\text{char}] \xrightarrow{*} \rho$ / Parse a character not specified in the list or fail.

Remark: Types of the compile-time and parse-time functions are mostly unsound, we rely on dynamic type checking at the compile time to catch the remaining errors.

Appendix B

EEL Help

```
Usage:
    eel [OPTIONS] [SOURCES...]

Options:
    -o FILE, --output FILE
        generate output binary file named FILE
    -L FILE, --llvm FILE
        generate LLVM IR text file named FILE
    -S FILE, --asm FILE
        generate assembly source file named FILE
    -v, --verbose
        increase output verbosity
    -i, --interactive
        launch interactive read-eval-print interpreter
    -P, --no-prelude
        do not load the prelude library automatically
    -e EXPR, --eval EXPR
        evaluate given EXPRession in EEL core
    -M NAME, --main NAME
        specify the name of the main function
    -h, --help
        show this help message
```

Listing B.1: EEL command-line options

```
  Available commands:
    EXPR        evaluate the EXPR
    :?          show this help message
    :q          quit
    :t EXPR     print type of an expression
    :d EXPR     dump AST of an expression
    :g EXPR     generate LLVM code for EXPR
    :i NAME     dump AST of an user-defined function
    :x          clear the stack
    :s          show current stack
    :l          list defined functions
```

Listing B.2: EEL interactive interpreter commands

Appendix C

CD-ROM Contents

The CD-ROM accompanying this technical report contains the following principal files and folders:

- **eel/** — EEL source tree
  - **doc/** — Haddock-generated source code documentation.
  - **lib/** — Directory containing the standard library *prelude.eel* and EEL language syntax definition *boot.eel*.
  - **samples/** — Sample source files in EEL language and dialects.
  - **src/** — Source codes of the EEL compiler.
  - **README** — Brief instructions on how to build and run the EEL compiler and examples.
- **report/** — LaTeXsources of this report
- **report.pdf** — Electronic version of this report