COBOL island grammars in SDF

by Ernst-Jan Verhoeven

Master's thesis August 2000

COBOL island grammars in SDF

by Ernst-Jan Verhoeven

University of Amsterdam
Informatics Institute, programming research group

performed at CWI/SIG, January-August 2000

supervisors: prof. dr. P. Klint, dr. A. van Deursen

0.1 Acknowledgements

Thanks to the people of the SEN1 research group at the CWI for sharing their insights about the subject and helping me get the most out of their MetaEnvironment, and the SIG for providing me with a place to work on this thesis.

Contents

0.1 Acknowledgements 2

1 Introduction **5**

2 An Overview of DocGen **7**

2.1 Global Overview 7

2.2 Perl extractor 8

2.2.1 Regular Expressions in Perl 8

2.3 Perl shortcomings 9

3 The ASF+SDF Meta-Environment **11**

4 Island grammar design **13**

4.1 What is an Island Grammar? 13

4.2 Reuse of an existing grammar 14

4.3 A design approach for island grammars 14

4.3.1 Overview 14

4.3.2 Application to COBOL 15

5 Island grammar specification **19**

5.1 A definition of water 19

5.1.1 Specification 19

5.1.2 Discussion 19

5.2 A simple island grammar 20

5.2.1 Specification 20

5.2.2 Discussion 21

5.3 Islands with nested water 23

5.3.1 Specification 23

5.3.2 Discussion 24

5.4 Water revisited 26

5.4.1 Specification 26

5.4.2 Discussion 27

5.5 Divisions, sections and paragraphs 28

5.5.1 Specification 28

5.5.2 Discussion 29

6 Framework design **31**
6.1 SDF modules . . . 31
6.1.1 Grammar rewriting . . . 32
6.1.2 Name conventions to prevent sort-conflicts . . . 33
6.2 A framework . . . 33

7 Integration with DocGen **37**
7.1 Pre-processing COBOL . . . 37
7.2 ASF rewrite equations . . . 38
7.2.1 Target language . . . 38
7.2.2 Rewrite module . . . 38
7.2.3 The rewrite-framework . . . 39
7.3 Post-processing . . . 40
7.4 Automation of complete rewrite . . . 40

8 Comparison with original extractor **43**
8.1 Speed . . . 43
8.1.1 ASFIX and Aterms . . . 43
8.1.2 Measurements . . . 43
8.2 Extraction results . . . 44
8.2.1 Robustness . . . 44
8.2.2 Correctness . . . 47
8.3 Adaptability . . . 48
8.3.1 System design . . . 48
8.3.2 Syntactic adaptions . . . 48
8.3.3 Lexical adaptions . . . 48

9 Conclusion **51**
9.0.4 Island grammar specification in SDF . . . 51
9.0.5 Island grammar application in DocGen . . . 51
9.1 Future work . . . 53
9.1.1 SGLR . . . 53
9.2 ASF . . . 53
9.3 Alternatives . . . 53

A Sources of DocGen integration components **57**
A.1 Pre-processing . . . 57
A.1.1 strip comments (stripper.pl) . . . 57
A.2 Post-processing . . . 57
A.2.1 add newline-seperators (tidy.pl) . . . 57
A.2.2 close divisions (close.pl) . . . 57
A.3 Makefile for parsing sources with both parsers . . . 58

B ASF+SDF specifications **61**
B.1 SDF-specification of CIG2CPF framework . . . 61
B.2 ASF-equations of CIG2CPF module . . . 69
B.3 CIG-Open island . . . 72

Chapter 1

Introduction

Numerous companies today struggle with the problems caused by legacy code. Their operational systems usually consist of millions of code-lines in some programming language now considered defunct. Little to no knowledge of the semantics of these languages and insufficient documentation about the internal functioning of the system as a whole create an unhealthy foundation for future development.

The analysis and renovation of such systems has grown into an important field of research, both due to the complexity of such a task and the obvious commercial interest for applicable methods. The SEN1 research group at the Centre for Mathematics and Computer Science had participated in several related projects involving renovation of legacy systems. To further exploit the tools and knowledge of SEN1 for commercial use, Paul Klint decided it was a good moment to start a spin-off company. Thus the Software Improvement Group (SIG) was born, a software engineering company with the intention to further develop and apply the technologies of SEN1 for commercial use.

The first technology to be deployed for commercial use is a documentation generation system named DocGen. By analysing the code of the programs that make up a COBOL-system, DocGen provides the user with a detailed review of interesting parts and their location. Furthermore it provides a visual representation of the interactions and dependencies between the separate programs. The information provided by DocGen allows quick reviewing of the functions of separate components within a system. Anomalies like programs with duplicate or redundant functionality can easily be located. Even if manual documentation is required, DocGen provides a foundation for a structured approach of such documentation.

In order to create useful feedback on a legacy system, DocGen needs to examine the code. Although DocGen consists of much more than the reading phase, all other components rely on this phase to provide them with correct information. The reading of code is currently done by a lexical analysis, which was implemented in Perl at the time the work on this thesis was initiated. On basis of several heuristics a decision is made if the code contains what DocGen is looking for, after which it is extracted for further use.

An alternative for this approach is the use of *island grammars* for information extraction. Instead of a full syntax definition, these grammars only contain explicit definitions of syntax parts we're interested in. The questions we are trying to answer are:

- How to write island grammars for COBOL using SDF?

  As a side effect of our research in island grammars, we used SDF and its parser generator SGLR in unorthodox ways. This has resulted in several observations on the behaviour of SGLR.

- How can DocGen integrate with and benefit from island grammars?

Chapter 2

An Overview of DocGen

In this chapter we give a global overview of the DocGen documentation generation system. The main point of focus will be the extraction phase, as we intend to alter it.

2.1 Global Overview

DocGen is a constantly evolving system, but we can represent it as a model of four subsystems that is not likely to change anytime soon.

- The Code Side (front-end)

  The front-end performs the actual extraction from the legacy source. This includes pre-processing the acquired data to a suitable format for the next part. The original version of this phase consisted of a variety of Perl-scripts.

- The Data Side (repository)

  The repository is where the output of the front-end resides for further use. It mainly consists of a MySQL relational database. This is not true for all front-end data however, as some of it still resides in a custom format.

- The User Side (back-end)

  The back-end provides the user with a clear interface to the generated documentation. The interface is provided in HTML, which is generated by queries on the repository database.

- The Integration Side (regulator)

  A Makefile [12] functions as the overall framework regulating the transformation from legacy to documentation. In practice this means feeding the appropriate sources to the front-end, instruct the repository to build the database and the back-end to put it all on display.

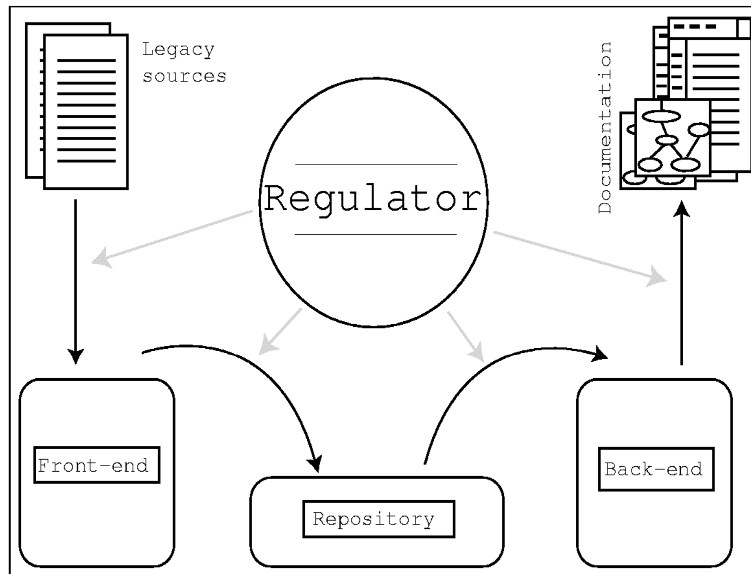

Figure 2.1: the components of DocGen

2.2 Perl extractor

Most of the tools that make up the front-end phase of DocGen were originally written in Perl [18]. Perl as a programming language is hard to characterize, but with some help of its creator, we'll make an attempt anyway.

Perl is an interpreted script language. It started out as a data reduction language; a language for navigating among various files in an arbitrary fashion, scanning large amounts of text efficiently, invoking commands to obtain dynamic data and printing easily formatted reports based on the information gleaned. After its initial release, its number of features has kept increasing. For many tasks Perl seems to provide the right primitives and sufficient efficiency.

Since the extraction of information from COBOL-sources is done using lexical analysis, Perl was an obvious candidate for the job. The most important Perl-feature for this purpose is probably its efficient and flexible implementation of string matching by use of regular expressions.

2.2.1 Regular Expressions in Perl

A regular set as used in formal language theory [13] is defined as being a regular set if it can be generated from the elements of the alphabet using union, concatenation, and the Kleene star operation. Groups of such sets can easily be written down using regular expressions.

Since Perl generally processes files with a little bit more content than just strings over the standard alphabet, its alphabet also contains layout characters (tabs, newlines, etc) and usually some superset of the ASCII table.

The union, concatenation, and Kleene star operations are a bit too general for practical use, thus Perl provides us with some additional operators for generalizing the notion of often-used sequences. They do not extend the expressive

domain of the original operators, thus keeping the overall duration of a match within polynomial time (N*M, with N the length of our regular expression, M the length of the term to match).

However, it is also possible to do something called backreferencing [18, 5] in Perl. Backreferences are just what they imply to be; references to a partial match done in the current expression. This is impossible to express in conventional regular expressions, thus when used they make regular expressions in Perl quite non-regular. In fact, it can be proven that the problem of matching expressions with backreferences is NP-complete [2].

2.3 Perl shortcomings

The extraction phase implemented in Perl is loosely based on insights from [10] and does an admirable job of scanning COBOL for information. However, it is not an ideal solution in all cases.

In [15], the advantages and disadvantages of lexical analysis are discussed. Speed and minimal knowledge of the syntax are tempting reasons for this approach, but both advantages are subject to the precision of information extraction; in order to improve precision we need more matching-criteria, which result in more overhead and knowledge of the syntax.

DocGen uses several heuristics to decide whether the results of its lexical analysis really are what they appear to be. These heuristics of course contain syntactical information about COBOL. As DocGen has to go through greater lengths to ensure a precise result, the line between lexical and syntactical analysis blurs to a point where second thoughts about using Perl arise.

As a result of the attempts to perform complex syntactical analysis from within Perl, the source code of DocGen itself can suffer from the fact that Perl is not designed with large projects in mind. For instance, it does not provide a type-mechanism and support for modularization seems more like an afterthought. To quote Perl's creator Larry Wall, 'Perl is the duct tape of the Internet'. As with real duct tape though, it provides a quick solution where alternatives are better suited for long-term use.

Chapter 3

The ASF+SDF Meta-Environment

The *ASF+SDF Meta-Environment* is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language ([14]). Developed as a replacement of its monolithic predecessor (now referred to as the *old Meta-Environment*), it consists of a set of standalone tools which can be accessed through a unified graphical interface (figure 3.1). In the remainder of this thesis we will simply refer to it as the Meta-Environment.

We will use the *Syntax Definition Formalism* (SDF) to define our island grammars. Although the formalism used in the old Meta-Environment is also called SDF, when we talk about SDF we refer to the formalism as used in the current Meta-Environment (also known as SDF2).

There are a number of advantages in using SDF over alternative parser generators. SDF is supported by *Scannerless Generalized LR Parsing* [16]. SDF allows for concise and natural definition of a syntax. SDF also promotes modularization of a syntax definition, which facilitates reuse.

SDF is complemented by the *Algebraic Specification Formalism* (ASF). ASF allows us to specify rewrite rules over the SDF specification. The ASF-compiler only recognizes a subset of the constructors available in the current SDF definition. Since we will use the ASF-compiler to rewrite our parse-trees to the desired form, SDF specifications will be restricted to this subset.

We will now look at a short example of both SDF and ASF: Suppose we have a language with a normal alphabet and brackets. It accepts any sequence of the alphabet as a string and all strings within a properly enclosing set of brackets. For this we can write

```
lexical syntax
 [a-z] -> Char
context-free syntax
 Char*          -> String
 "(" String ")" -> String
```

This specification will parse terms like

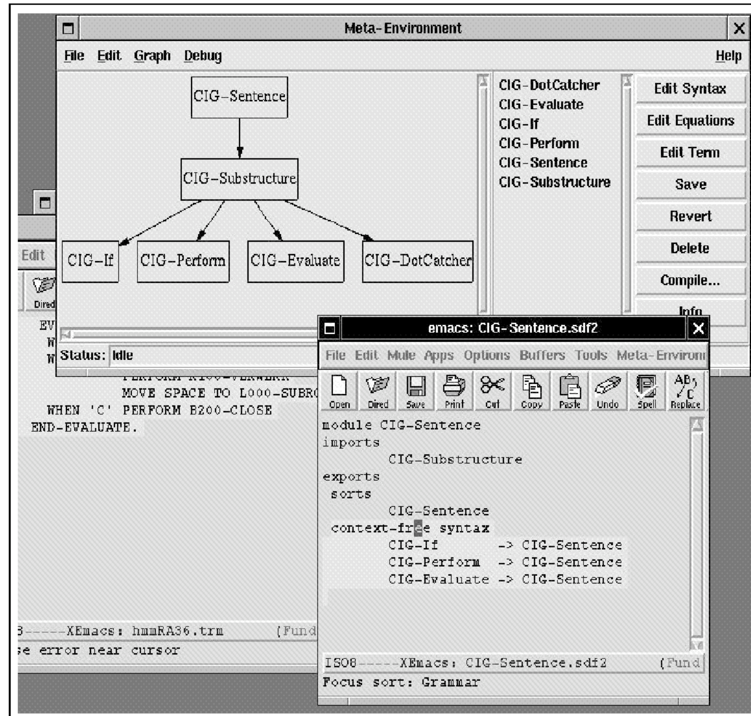

Figure 3.1: The Meta-Environment in action

```
abcdefgh
((abcdefgh))
```

but not

```
abc(defgh)
```

Although a String may be put between as many pairs of brackets as we like, they don't really represent anything meaningful in our current language. We might as well remove them, a nice job for ASF. We add

```
variables
 "stringvar" -> String
```

to the SDF-specification. We can now use 'stringvar' as a variable in our ASF-specification, which looks like this:

```
equations
[1] (stringvar) = stringvar
```

Both sides of the equivalence-operator are of the same sort. Every time a construction of the lefthand-side is encountered, it will be rewritten to the righthand-side. This will effectively eliminate all enclosing brackets, regardless of the level of nesting. Note how the ASF specification already knows what our language looks like from the SDF specification. We only need to add variables to generalize rules.

There is a lot more to both ASF and SDF than this simple example. Without pretending to be a manual for the Meta-Environment (try [14] instead), we provide the reader with explanations of some relevant details along the way.

Chapter 4

Island grammar design

4.1 What is an Island Grammar?

An *island grammar* [15] consists of:

- detailed productions for the language constructs we are specifically interested in (islands);
- liberal productions catching all remaining constructs (water);
- a minimal set of general definitions covering the overall structure of a program (framework).

Simply put, we have a full grammar of parts we're interested in, the islands. These are surrounded by constructs we're not interested in, the water. A simple island grammar could look like this:

```
context-free syntax
 Token*  -> Source
 Island  -> Token
 Water   -> Token
context-free priorities
 Island -> Token >
 Water  -> Token
```

Note that Token is just a sort name, it does not imply a tokenizing process as done by some scanners. The priority states that when in doubt, we prefer to recognize a Token as an Island instead of Water.

The opposite approach, an explicit definition of the parts we're not interested in, is known as a *lake grammar*. This sort of grammar would of course be inappropriate for use in DocGen; since we're not interested in the water-part, it'd be contradictory to take a close look at it. Rewriting the islands would also be messy, since they will all be caught in liberal constructs.

By using an island grammar, we hope to gain some of the power of parsing with a full grammar combined with the minimal knowledge inherent to lexical analysis. A parse with a full grammar provides us with access to detailed

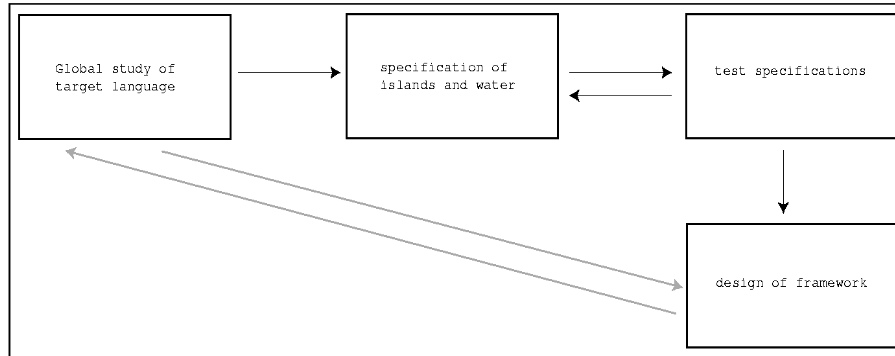

Figure 4.1: An approach for island grammar design

information about the parsed structures. There should be no ambiguities during parsing, which guarantees correctness of extraction beyond what a typical lexical analysis can offer. On the other hand, lexical analysis allows us to get started with collecting information on a basis of minimal knowledge; we only require knowledge about what we're looking for, not about all of its surrounding context. DocGen currently adds syntactic knowledge to a lexical analysis mechanism. We will try to work the other way around and include the ability of working on basis of minimal knowledge to a syntactic parsing mechanism.

4.2 Reuse of an existing grammar

The modular design of most grammars in SDF and the concept of an island grammar suggest we can reuse definitions. Although there are some COBOL-grammars available in SDF, their design makes it harder to reuse them than to rewrite the definitions. This is also a result of the fact that some constructs from a full COBOL-grammar don't qualify for reuse at all. Consider for instance an IF-statement: We're interested in the skeleton of the statement itself, but not necessarily in the statements nested within. Reuse of such a statement from a full grammar would enforce full knowledge of the nested part. This would of course ruin the idea of minimal knowledge about the syntax.

Existing full grammars do provide a good starting point for their island-counterparts. The Browsable VS COBOL II grammar [7, 8] and the definitions in the COBOL-books by Ebbinkhuijsen [3, 4] cover nearly all variations to be expected in the islands.

4.3 A design approach for island grammars

4.3.1 Overview

The design of an island grammar for COBOL turned out to be quite a trial-and-error experience. This is reflected in the design approach we will roughly follow (figure 4.1):

- Global study of the target language

Contrary to when designing a full grammar, we're not interested in every syntactic detail of the target language. Our study will be done with a desire of minimal syntax knowledge in mind. We will concentrate on special symbols for things like comments, nestings, line-separation and divisions, but no full tokenization of the language will be done. During this phase we also examine the overall structure of the language; what syntax can be expected where.

- Specification of islands and water

  We specify what we think our islands and water should look like. Our first specification of water will probably not include much detail, most likely it will be a very liberal constructor with maybe some exceptions for special symbols. We should already have a clear idea about what our islands look like, being the constructs of our main interest.

- Test of specifications

  Using a number variations of target language sources, we test parse behaviour with our current specifications. This phase will interact with the previous phase until our specifications behave as desired.

- Design of a framework

  Our separate island grammars need a unified framework, both for improved clarity and to generalize the specification of future extensions. The modularization of this framework will be done on basis of specification similarities between islands. This approach may seem in conflict with following the overall structure of the target language. However, the two approaches will turn out to be very related. In fact, when in doubt about categorizing a specification, the overall structure of the target language can help us decide.

4.3.2 Application to COBOL

This section gives a global overview of how we intend to apply the approach for designing a COBOL island grammar.

Global study of the target language

Apart from weird comment-conventions, the global syntax of COBOL seems pretty straightforward:

- There are two different comment-conventions

  The first and last few columns of a typical COBOL-sourceline are comment-blocks. Which number of columns contain comments varies between COBOL dialects, but the first 6 and last 8 columns of a 80 column wide line seems to be the standard (figure 4.2). If the first column after the first comment-block contains a '/' or '*' symbol, the whole sourceline is a comment.

  As the comment-columns are defined in a way that makes syntactic analysis very awkward, we simply strip all comments during preprocessing by means of lexical analysis. When we speak about the beginning of a line in

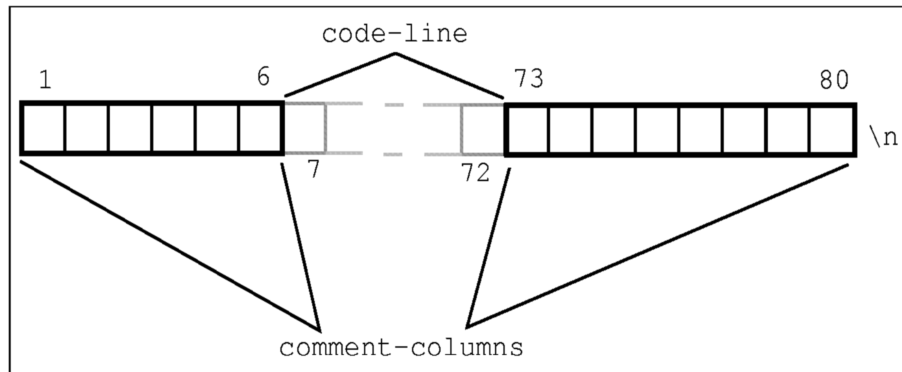

Figure 4.2: comment columns in a typical COBOL-sourceline

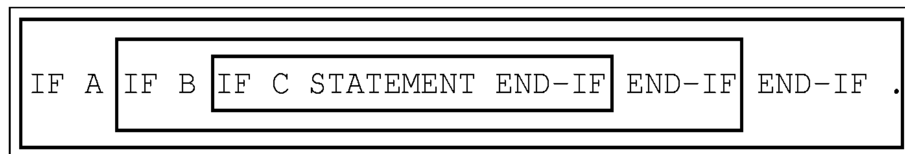

Figure 4.3: an IF-statement nested in an IF-Statement nested in an IF-statement

the remainder of this thesis, we refer to a line without comment-columns unless explicitly stated otherwise.

- The '.' symbol (period) serves as a sentence-terminator.

  Typical COBOL-sentences look like this:

  ```
  STATEMENT .
  ```

- Strings are nested within single or double quotes.

  Strings can also span more than one line using the '-' symbol (line-continuation symbol):

  ```
  STATEMENT "this statement has a string over two lines as
  - "an argument" .
  ```

- COBOL programs are divided in Divisions, Sections and Paragraphs.

  Certain code is restricted to certain divisions. For example, most of the program's functionality is situated in the Procedure Division. The divisions are identified by headers.

- There are no special symbols for statement-nesting.

  Instead, statements that allow nesting of statements use keywords and/or sentence-termination as enclosures, or the fact that the enclosing statement accepts only one nested statement (figure 4.3).

Specification of water and islands

The islands we're interested in consist of simple statements with no water as part of their definition, statements which allow for nestings that might contain water and headers for some of the divisions. Our definition of water will follow the expected design approach, initially it will only contain an exception for the period-symbol. The SDF-specification will be explained in detail starting from section 5.1.

Test of specifications

The SIG already has a number of COBOL-sources in various dialects for evaluation. Testing specifications was done by parsing these sources and tracking down problems. The parser can only tell us whether the complete parse fails or contains ambiguities. If our water-definition accepts more than we like, the parse will simply succeed. This behaviour poses a problem in that we can only spot failures to recognize an island if we already know what the outcome of a parse should look like. This is not an easy task with large COBOL-sources, but luckily we already have a robust extractor in Perl at our disposal. By writing some simple rewrite-rules in ASF to extract our islands from the parse-tree, we can compare the outcome of our tests with the outcome of the Perl-extractor.

Once we spot a problematic island-definition, we can look up the exact statement-definition in the documentation (e.g. [3, 4], or dialect-specific documentation) and create more critical test-cases to test with our parser.

Design of a framework

As mentioned in chapter 3, SDF allows for modularization. We classify our islands in modules. Some of the connecting modules will describe the overall structure of the COBOL-language from our study in section 4.3.2. A more thorough discussion of this framework will be presented in section 6.

Chapter 5

Island grammar specification

5.1 A definition of water

5.1.1 Specification

For our definition of water, the words we are *not* interested in, we use

```
lexical syntax
 ~[\ \t\n\.]+ -> Drop
lexical restrictions
 Drop -/- ~[\ \t\n\.]
```

This means that we accept one or more of the characters that are not a layout-symbol (whitespace, tab and newline) or a period (the sentence-terminator). We suspect that our water-definition might become a bit more complex than this, thus we name our sort Drop instead of the more ambitious Water for the time being.

The restriction demands that a Drop is only accepted as such if all adjacent symbols of the allowed set are part of the sort. This enforces a longest match while parsing.

5.1.2 Discussion

Layout

A sort called LAYOUT is used in every SDF-definition. If not specifically stated otherwise, it can occur between every symbol that is part of a construct. Since our grammar is used for parsing itself, it would be impossible to give it a readable format without declaring this sort in a sensible way. We define our layout as

```
lexical syntax
 [\ \t\n] -> LAYOUT
```

To prevent ambiguity between LAYOUT and Drop, we exclude the symbols that make up the sort LAYOUT from the set of symbols that make up Drop.

Longest match restriction

For the definition of Drop, we could also write

```
lexical syntax
 ~[\ \t\n\.] -> Drop
```

This means that a Drop no longer consists of continuous blocks of characters, but of a single character. We did not choose this option, as it can seriously affect parse performance; the amount of water far exceeds the number of islands during a typical parse. If every character of this water is a Drop on itself, our parse tree will become huge.

SGLR behaviour on longest match restriction

The reason we reject the period as part of Drop is because we suspect it may intervene with correct recognition of complex islands. Now suppose we only work with islands that contain no water as part of their definition, would there still be need to reject the period as part of the Drop-definition? Unfortunately the answer is yes. To see this, consider the term

```
STATEMENT.
```

Notice how there is no layout between the statement and the period. Now suppose the statement is an island. The parser will ignore our preference for recognizing the statement and create an ambiguity of the correct parse and one that accepts the whole block as one Drop. Apparently the greedy lexical constructor does not pay sufficient attention to priorities[1].

5.2 A simple island grammar

5.2.1 Specification

Now that we have a definition of water, all we need is a definition of our islands and define priorities over them. We start with the relatively simple COPY-statement. That is, relatively simple due to our limited interest: we only care for the first argument. Our island looks like

```
COPY text-name
```

or in SDF

```
context-free syntax
 "COPY" Id -> Copy
```

The syntax of text-name is very dependent on the COBOL-dialect being parsed. For now we use the following (quite common) definition in SDF

```
lexical syntax
 [A-Z][A-Za-z0-9\-\_]* -> Id
lexical restrictions
 Id -/- [A-Za-z0-9\-\_]
```

[1] ...which obviously is a bug. The latest Meta-Environment release allows us to enforce this by means of an avoid-property.

We accept anything that starts with a capital and contains zero or more characters from the second constructor.

There still is the unresolved issue of the period. Since the period terminates sentences in COBOL, and COBOL-sources pretty much exist of adjacent sentences, we can write

```
context-free syntax
 Copy        -> Island
 Island      -> Token
 Drop        -> Token
 Token* "." -> Sentence
 Sentence*  -> Program
context-free priorities
 Island -> Token >
 Drop   -> Token
```

Symbols followed by a period are recognized as a Sentence. We prefer recognizing the symbols as Islands, but accept Drops too. At the moment, our only island is Copy. A list of Sentences makes up a Program.

Already we have a framework for recognizing islands that do not contain water as part of their definition. It is easy to see that any such island can be included in the parse by including it as an Island. Consider for example the CALL-statement. We assume we already have a definition for this statement, including it would be a trivial matter of also defining it as an Island:

```
context-free syntax
 "CALL" Id -> Call
 Call      -> Island
```

5.2.2 Discussion

SGLR behaviour on list-element priorities

Looking back at our framework for simple islands, why didn't we define our water-token as

```
context-free syntax
 Drop* -> Token
```

with a priority

```
context-free priorities
 Island -> Token >
 Drop*  -> Token
```

instead? Note the list-constructor, which groups all adjacent Drops together to one big water-Token. Later on this sort of definition will become even more tempting, but it will not work! As far as I can tell, this is due to the internal priorities of the list-constructor completely ignoring any priorities specified at a higher level. For instance,

```
A B C COPY D.
COPY E F G COPY H.
```

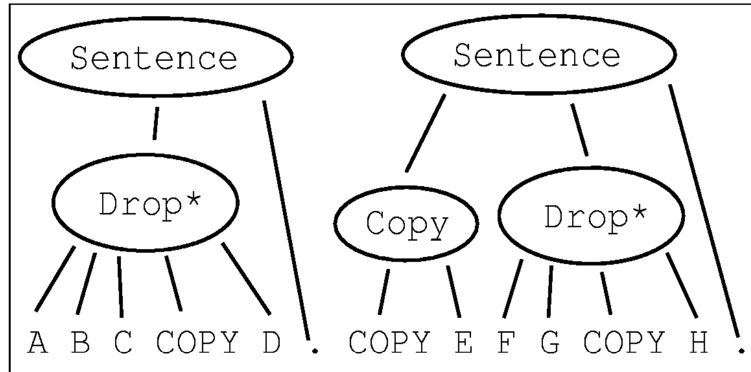

Figure 5.1: Unexpected parse of Copy/Drop* mixture

will parse as (figure 5.1)

```
Drop Drop Drop Drop Drop .
Copy Drop Drop Drop Drop .
```

The Copy in the first sentence was completely ignored in favour of constructing one big Drop-list, in the second sentence the first Copy is recognized but the next Copy is again completely lost once the list-constructor starts eating away. No ambiguities are generated during such a parse either. Whether this is good behaviour or not, it is definitely most un-intuitive. Due to this behaviour, we cannot safely use a list as member of a list if we are to disambiguate it from other members by means of priorities.

Injective priorities

The priorities in our framework may not seem correct at first. However, SDF provides us with so-called *injective priorities.* This means that

```
context-free priorities
 Island -> Token >
 Drop   -> Token
```

has the same result as priorizing the entire chain up to Island before the chain up to Drop. So we automatically gain

```
"COPY" Id  -> Token >
~[\ \t\n]* -> Token
```

If it wasn't for this feature, adding another island would be an error-prone task of specifying priorities on several sorts down the chains. Instead, we have a clean and intuitive mechanism to generalize this.

Properties on sorts

Another way to specify the division between islands and water in SDF is to use properties:

```
context-free syntax
 Island -> Token{prefer}
 Drop   -> Token{avoid}
```

The parser will prefer constructs with the prefer-property above other ambiguous constructs. On the other hand, constructs with the avoid-property will only be used when no other constructs are eligible. The application of these properties are delayed by the parser until all paths are constructed. This delayed application is exactly why we use explicit priorities instead. When the parser encounters paths that are ordered by explicit priorities, the priorities will be applied as soon as possible. This will eliminate any infeasible paths in an early stage. Reducing the amount of paths to be evaluated in an early stage is beneficial for parsing performance.

5.3 Islands with nested water

5.3.1 Specification

So far we only looked at islands that contain no water as part of their definition. When we consider a construct with nested water, things change somewhat. The EVALUATE-statement for example can contain other statements as a nesting, which will lead to nested water due to our liberal grammar definition.

The nestings of EVALUATE can be closed by means of the 'END-EVALUATE' keyword or by the sentence-terminator, the period. If the period ends an EVALUATE, the 'END-EVALUATE' keyword becomes optional. If EVALUATE itself is part of a nesting, the 'END-EVALUATE' keyword is mandatory. Since the period is part of the syntax of the EVALUATE-statement, we partially define the statement on the level of Sentence. Note that we will conveniently ignore the fact that a period may close multiple nested open statements. For example

```
EVALUATE EVALUATE CALL XYZ .
```

will not be recognized by our island-definition.

We also take some precautions to prevent the water from intervening with our boundaries by means of the reject-property. The discussion of this section will give a more thorough explanation of the problems we may encounter if we don't do this.

```
context-free syntax
 Token                                  -> EvalNesting

 "EVALUATE" EvalNesting* "END-EVALUATE" -> Evaluate

 "EVALUATE" EvalNesting* "."            -> Sentence
 Evaluate "."                           -> Sentence

 "EVALUATE"                             -> EvalNesting{reject}
 "END-EVALUATE"                         -> EvalNesting{reject}
```

An example parse with this specification can be found in figure 5.2. The reject-property states that we do not accept the boundary-keywords, 'EVALUATE' and 'END-EVALUATE', as EvalNestings. Since a Drop only occurs

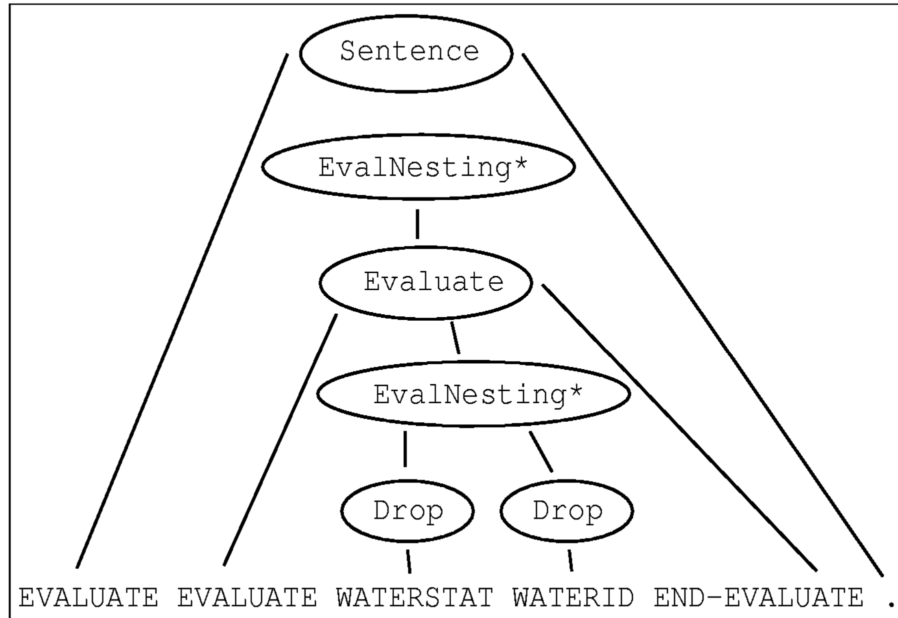

Figure 5.2: An example parse tree of an EVALUATE-statement

as an EvalNesting, they will not be accepted as such. We also introduced a new sort EvalNesting, which allows us to specifically state differences with the Token-sort.

Although internal priorities will already prefer these Sentence-definitions in favour of the earlier definition, it is probably a good idea to explicitly state this in the context-free priorities.

5.3.2 Discussion

The importance of the reject-property

The reject-property might seem like overkill, so let's define Evaluate a bit less restrictive:

```
context-free syntax
 "EVALUATE" Token* "END-EVALUATE" -> Evaluate
 "EVALUATE" Token* "."            -> Sentence
 Evaluate "."                     -> Sentence
```

We removed the rejections and hope our priorities sort things out. We also assume that the Token-sort is specific enough to define the nesting.

And this is where we suddenly introduce a quite unexpected ambiguity. Consider the parse of

```
EVALUATE EVALUATE END-EVALUATE EVALUATE END-EVALUATE END-EVALUATE .
```

This will parse quite happily, but not as you might expect (see figure 5.3):

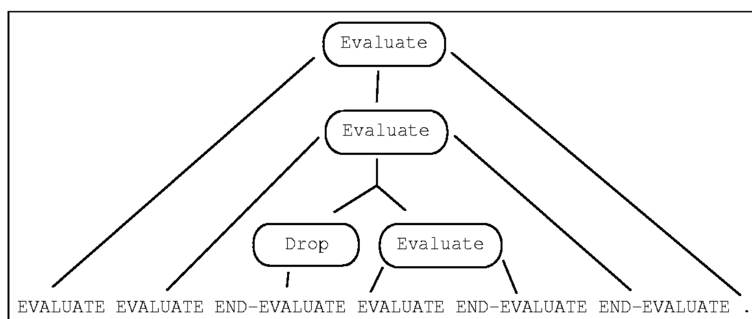

Figure 5.3: unexpected parse of nested evaluates

```
EVALUATE(1) EVALUATE(2) Drop EVALUATE(3) END-EVALUATE(3)
END-EVALUATE(2) (.)1
```

What is going on? Apparently an internal priority decided to go for the shortest way home and replaced the closing of the second Evaluate for a Drop. We still end up with a parse of three Evaluates, but the structure is completely wrong! Maybe we can define the structure of Evaluate in some unambiguous way. It is also tempting to specify a priority ordering over the different Evaluate-constructs. Neither of these solutions seems appropriate, our definitions of Evaluate are already unambiguous if the Drop-construct wasn't so liberal. All we want to do is tell the parser not to accept 'END-EVALUATE', nor 'EVALUATE' for that matter, as a Drop. Luckily, SDF provides us with a powerful reject-mechanism:

```
context-free syntax
 "EVALUATE"     -> Drop{reject}
 "END-EVALUATE" -> Drop{reject}
```

This way, neither boundary-strings of the construct are accepted as sort Drop.

Rejection at different sort-levels

In the previous section we rejected the boundary keywords from the sort Drop to prevent ambiguities. However, we loose some of our flexibility during parsing here; for all we know, those keywords may appear in some context completely unrelated to the current construct. Although highly unlikely in the case of the Evaluate-statement, in general we cannot assume all keywords to be rejected only occur in a context relevant to their statement. To regain some of our flexibility, we define

```
context-free syntax
 Token           -> EvalNesting
 "EVALUATE"      -> EvalNesting{reject}
 "END-EVALUATE"  -> EvalNesting{reject}
```

All occurrences of 'Token*' in the Evaluate-constructs should be replaced by 'EvalNesting*' now. Note the use of 'EvalNesting' instead of a more generic

'Nesting'. Both approaches have something to say for them. In the case of a generic Nesting-sort, we loose flexibility for parsing weird anomalies of rejected keywords of other constructs within a construct. If on the other hand we only reject keywords within the context of their own construct, we take the risk of non-rejected keywords messing up the construct. For example

```
EVALUATE PERFORM END-EVALUATE END-PERFORM END-EVALUATE
```

will not parse if we reject 'END-EVALUATE' as a Nesting. It will however parse if we reject it as an EvalNesting, because PerformNesting will still happily accept it as a Drop. In theory, the PerformNesting could contain an essential part of some alternative Evaluate-construct. However, neither rejection-method would be able to parse this successfully, as a Nesting-reject would trip over the rejection of 'PERFORM' and 'END-PERFORM' and an EvalNesting-reject would cause an ambiguity (which we probably never get to see, again due to internal priorities).

Alternative nesting-definitions

The last detail we will discuss here is the case of a statement that allows for only one nesting. Suppose the EVALUATE-statement allows only one statement within its enclosures, should we still specify the nesting as EvalNesting*, or a single EvalNesting instead?

The problem with the latter specification is the fact that our single nested statement might be an irrelevant statement (water), that could nest an island:

```
EVALUATE WATER EVALUATE END-EVALUATE END-WATER END-EVALUATE.
```

Obviously we will not catch the nested EVALUATE-statement if we simply allow an EvaluateNesting to consist of a list of Drops. An alternative definition of EvalNesting

```
Drop* Token Drop* -> EvalNesting
```

will not behave as we hope for. This probably has to do again with the greedy listconstructor as described in section 5.2.2.

5.4 Water revisited

5.4.1 Specification

With the introduction of islands containing nested water, Drop does not suffice as our only water-construct anymore. Consider for instance

```
EVALUATE WATER 'a period looks like: .' END-EVALUATE.
```

Periods can occur within strings, which can occur as part of the nested water. Currently this would result in a failure to correctly recognize the island. For this specification we assume that strings in the COBOL-sources to be parsed are enclosed within single quotes, although in practice double quotes are just as common. As an extension to nested water, we add a lexical construct for a DotCatcher sort.

```
lexical syntax
 "\'~[\'\n]*\'"                        -> DotCatcher
```

The DotCatcher accepts everything but a single quote or newline enclosed in single quotes. Adding support for double quotes is a simple matter of duplicating the specification and replace the single quote with a double quote.

As we noted in section 4.3.2, strings in COBOL may also span multiple lines by use of the '-' *line-continuation* symbol. We add the following lexical specification for support by our island grammar:

```
lexical syntax
 "\'~[\'\n]*\n\-[\ ]*\'~[\'\n]*\'" -> DotCatcher
```

This construct accepts a string spread over two lines by the line-continuation symbol. Note that the first line does not contain a closing quote-mark, but the second line does contain an opening quote-mark after the line-continuation hyphen.

DotCatcher is an alternative to Drop, we make sure the sorts don't conflict by changing Drop to

```
lexical restrictions
 Drop -/- ~[\ \t\n\.\']
lexical syntax
 ~[\ \t\n\.\']+ -> Drop
```

Drop cannot contain single quotes anymore, whereas DotCatcher requires them.

We introduce the new Water sort, which includes both water sorts.

```
context-free syntax
 Drop       -> Water
 DotCatcher -> Water
```

Furthermore, we replace the occurrence of Drop by Water:

```
context-free syntax
 Island -> Token
 Water  -> Token
context-free priorities
 Island -> Token >
 Water  -> Token
```

5.4.2 Discussion

DotCatcher motivation

Before water could occur as part of an island-nesting, irrelevant strings containing periods would simply be divided in several Sentences. For example

```
WATER 'period: . and again: .'.
```

would parse as

```
"WATER 'period:" "."  -> Sentence
"and again: ." "."    -> Sentence
"'" "."               -> Sentence
```

The same behaviour would clearly ruin the parsing of this irrelevant statement when nested in an Evaluate. Note that this problem is inherent to our design approach; it would probably not occur if we completely tokenize the language.

The existence of DotCatcher may have a more convincing motivation as well; we may be able to parse nestings in a more elegant way, but DotCatcher also affects correctness of island recognition by catching would-be islands within strings. As an example, consider:

```
"COPY A"
```

Without DotCatcher, this string would contain a valid COPY-island. A problem that is not uncommon in lexical analysis either.

Line-continuation pitfalls

Our DotCatcher sort is not sufficient for all cases: line-continuation is not necessarily restricted to only two subsequent lines. The use of line-continuation in COBOL, especially for more than two subsequent lines, is not advisable [4], as there are better alternatives. However, if we want our island grammar to be as flexible as possible, the specification of DotCatcher deserves some more attention.

We would probably like to write something like this:

```
lexical syntax
 "\'~[\'\n]*\'"              -> DotCatcher
 "\n\-[\ ]*\'~[\n\']*"      -> LineContinuation

context-free syntax
 "\'~[\n\']*" LineContinuation* "\n\-[\ ]*\'~[\n\']*\'" -> DotCatcher
```

We now have a definition for several subsequent line-continuations, but the behaviour of our layout conflicts with the newlines in the context-free syntax (remember, LAYOUT can occur between sorts). There are ways around this in SDF, but these are too low-level to be discussed here. Already we deal with more lexical specifics than we wish for in a grammar-definition.

5.5 Divisions, sections and paragraphs

5.5.1 Specification

A typical COBOL-program is divided into four divisions: Identification, Environment, Data and Procedure Division. These are easily identified by simple headers. The header of the Procedure Division for example is

```
PROCEDURE DIVISION .
```

or

```
PROCEDURE DIVISION USING <data-name>+ .
```

The Procedure Division can contain paragraphs and sections, both of which are also identified by a header. For a section this is

```
<section-name> SECTION .
```

and for a paragraph

```
<paragraph-name> .
```

Both the paragraph-name and section-name are user-defined names (with the exception of some reserved keywords). These keywords usually occur in divisions other than the Procedure Division. In SDF we write

```
context-free syntax
 "PROCEDURE" "DIVISION" "."              -> ProcedureDivisionHeader
 "PROCEDURE" "DIVISION" "USING" Id+ "." -> ProcedureDivisionHeader

 Id                                      -> ParagraphId
 Id                                      -> SectionId

 SectionId "SECTION" "."                 -> SectionHeader
 ParagraphId "."                         -> ParagraphHeader
```

We assume the Id sort equals a data-name in the ProcedureDivisionHeader. Since we did not specify an explicit nesting-ordering over the headers (e.g. a SectionHeader occurs within a Procedure Division), we do not simply reuse the Id-sort for paragraphs and sections. We can easily prevent reserved keywords from being accepted as SectionId or ParagraphId with the reject-property:

```
context-free syntax
 "EXIT"         -> SectionId{reject}
 "DECLARATIVES" -> ParagraphId{reject}
```

The headers can now easily be added as islands on the Sentence-level and separated from water.

5.5.2 Discussion

Nesting of headers

We mentioned an explicit nesting-ordering over the headers in section 5.5.1. This ordering becomes apparent when looking at the complete structure of a Procedure Division. Without going into detail, it turns out that (user-defined) sections and paragraphs only occur within the Procedure Division. Paragraphs can be located within the division on their own, but if there are sections in the division, the paragraphs will be nested within the sections (figure 5.4).

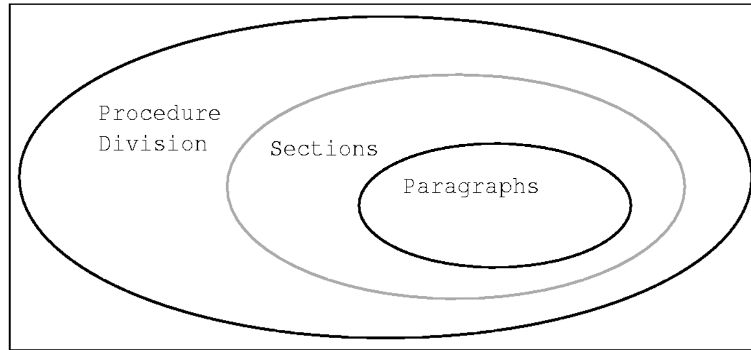

Figure 5.4: Nesting of headers in Procedure Division

source	comment
FIRST SECTION.	section FIRST begins
FIRSTPAR.	paragraph FIRSTPAR begins
..	
..	
	paragraph FIRSTPAR ends(1)
	section FIRST ends(1)
SECOND SECTION.	section SECOND begins(2)

Figure 5.5: An example of header-nesting. Notice how (1) is derived from (2) instead of directly from the source.

Specification motivation

Our ParagraphHeader is very ambiguous if we do not explicitly reject all invalid single-keyword sentences. We already stated that most reserved sections only occur outside the Procedure Division. Why don't we use this knowledge to parse the source for more detailed information about these structures? Currently our grammar has no idea what a complete Section-block looks like, it just spots possible headers.

It turns out that a more detailed specification would result in little gain of information at the expense of performance and minimal syntax knowledge. For instance, if the Procedure Division is the last division of the source then we already know the boundaries of this division (from header to end). A similar situation exists for the sections and paragraphs; If a section starts, the current section and paragraph (if any) are ended. If a new paragraph starts, the current paragraph (if any) is ended. Figure 5.5 demonstrates this concept.

So all information about beginnings and endings of the divisions is already in ordering of the headers. This makes a more detailed specification not only very hard to implement and inefficient to for parsing, it also seems a rather useless exercise. We're better off collecting this information while post-processing the parse-tree.

Chapter 6

Framework design

In this chapter we discuss the creation of a framework for our islands. First we will look at some general issues with modularization of SDF-specifications. We apply our findings to the design of a framework for simple islands (islands that contain no water) and expand this design for use with islands that can contain water and division-headers.

6.1 SDF modules

A typical SDF-module does not merely consist of the specifications we've seen so far. Our specifications are preceded by a module-name header

```
module Main
```

which simply states the name of the module, Main in this case.

The module-header is optionally followed by the imports-header:

```
imports A B
```

This means exactly what it says, import the modules A and B.

Now it is time for our specifications, which are preceded by either

```
exports
 sorts <sortnames>
```

or

```
hiddens
```

Every specification preceded by the hiddens-header will remain hidden for all other modules, even if they explicitly import the module with hiddens. All other specifications are preceded by the exports-header. The names of all sorts declared in the exports-section should be added to the sorts-list. Suppose for example that our Main module defines sorts D and E. Our Main module will look like this (see figure 6.1):

```
module Main

imports A B
```

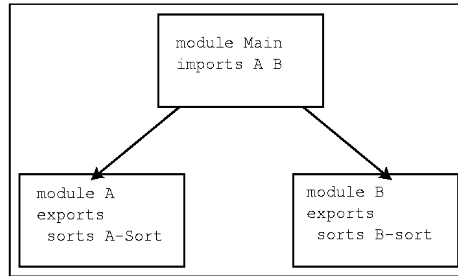

Figure 6.1: SDF import-graph, arrows point at imported modules

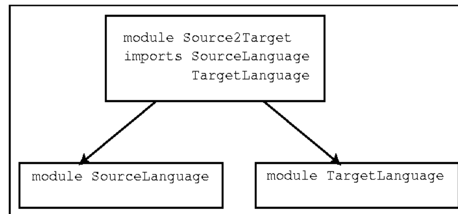

Figure 6.2: import-graph for grammar rewriting

```
exports
 sorts D-sort E-sort

 context-free syntax
   A-sort                 -> D-sort
   B-sort A-sort B-sort -> E-sort
```

We assume here that B-sort and A-sort are exported in either module A or B. If both modules A and B contain constructs for the same sort, Main will import the union of those constructs.

Import-export behaviour is not as self-evident as one might expect, we exploited a 'feature' that exists due to the current absence of type-checking in the MetaEnvironment. It is true that upon evaluation, module B has no knowledge of exports from module A. However upon evaluation of the Main module, exported information from modules A and B are not only available to the Main module but also to each other. In general, upon evaluation of a top module, all lower modules that are connected in the import-graph (regardless of graph-direction) have knowledge about each other and the top module.

6.1.1 Grammar rewriting

When rewriting a grammar, we use the import-structure as presented in figure 6.2. The top-module source2target contains the ASF transformation rules, combined with variable declarations, transformation function signatures and maybe some helper constructs.

The ASF rules will be discussed in more detail in section 7.2, but we already note the use of a function for rewriting

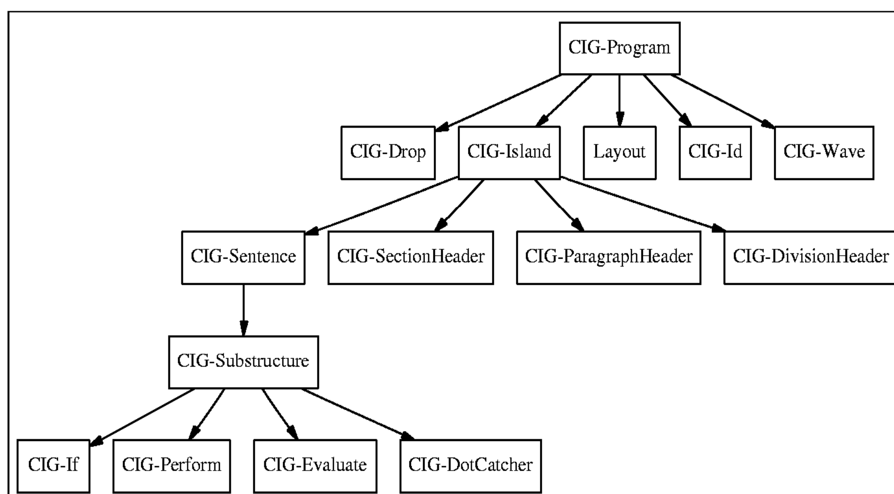

Figure 6.3: import-graph for an SDF island grammar framework

```
f(source) -> target
```

instead of extending the target-language with the source-constructs:

```
source -> target
```

The former method enforces a complete rewrite of the source grammar to target grammar, which have no knowledge of each other. In the latter case, we would 'contaminate' the source grammar with constructs of the target grammar. Thus we can end up with a partial rewrite that contains constructs from both grammars.

6.1.2 Name conventions to prevent sort-conflicts

To keep our grammars as reusable as possible, we change the sort-names to something less general; Our grammar will likely not be the only one to use a sort-name like Program. We simply add the 'CIG-' prefix (Cobol Island Grammar) to all our sorts for this purpose.

6.2 A framework

A possible SDF framework for our islands is presented in figure 6.3. Name conventions may differ somewhat from previous chapters, since the import-graph is directly derived from an experimental grammar. The complete source is presented in appendix B. We will walk through the modules in bottom-up left-to-right fashion:

- CIG-If, CIG-Perform, CIG-Evaluate.

  These are the complex islands of the framework, which more or less all behave like Evaluate in section 5.3.

- CIG-DotCatcher.

  The DotCatcher sort is only used in this framework for catching strings within complex islands.

- CIG-Substructure.

  A more appropriate name would probably be 'CIG-Nesting'. This module contains the definition for every sort that can be nested within a complex island.

- CIG-Sentence.

  Every statement terminated with a period is considered a sentence. In our case these are the complex islands. Don't be fooled by the import of CIG-Substructure; The actual statements are defined in the island-modules (CIG-If, etc) but are imported through CIG-Substructure to prevent import-cycles.

- CIG-SectionHeader, CIG-ParagraphHeader, CIG-DivisionHeader.

  These module contain the definitions of the division-headers. They're not really statements and behave different enough to not include in CIG-Sentence.

- CIG-Island

  This module contains all islands to be separated from water on a sentence-level. In this case the headers and CIG-Islands.

- Layout, CIG-Id, CIG-Drop

  Sorts used by multiple modules. Our only motivation for importing them at top-level is the resulting clean graph.

- CIG-Wave

  A CIG-Wave is actually a Sentence that consists completely of Water terminated by a period. Due to the nature of the islands in this framework we only care about islands that define a Sentence, as opposed to the simple islands in section 5.2. Therefore we define a water-sort at the same level.

- CIG-Program

  At the top level the CIG-Waves (the water at Program-level) and CIG-Islands are separated and some modules for use by multiple lower modules are imported.

The framework represented here tries to mimic the extraction-functionality of one Perl-extractor. As such, it does not contain modules for simple islands (which were handled by a different Perl-extractor). However, depending on our interests it is easy to add support for simple islands to this framework:

- CIG-Substructure level

  We can add our simple islands at CIG-Substructure level to catch occurrences within nestings of complex islands.

- CIG-Island level

  We can add our simple islands at CIG-Island level to catch single-statement sentences containing our simple islands. Note that in this case we cannot simply specify a partial section of a statement anymore, as the island has to match completely until the terminating period.

- CIG-Wave level

  We can add our simple islands at CIG-Wave level to catch simple islands in sentences that otherwise consist of water.

Of course we are not restricted to the framework or its extension for simple islands as described above. Its main function is to show a possible approach and the flexibility and transparency in design with SDF-modularization.

Chapter 7

Integration with DocGen

Of course the island grammar is not capable of accomplishing much on itself within DocGen. In this chapter we discuss the necessary steps to integrate our island grammar in DocGen. For this we need to pre-process the source files, rewrite the island grammar to the desired output and post-process the output to mimic the Perl-results as close as possible. The sources of the integration components that were actually used are shown in appendix A.

7.1 Pre-processing COBOL

There are some things we rather don't attempt in SDF, simply because it is either not possible or much easier to do before the actual parse. We name a few encountered cases for pre-processing:

- Text-format conversion

  Some files may be in a file-format other than the UNIX-style ASCII files we expect (see also section 8.2.1). It is a trivial but necessary step to make sure they are converted to the expected format prior to parsing.

- Strip comments

  COBOL has comment-conventions that do not mix well with syntactic analysis, as we mentioned in section 4.3.2. A simple Perl-script strips comments from our source-files before the parse.

- Connect line-continuations

  We struggled to get a decent line-continuation specification in section 5.4. As the line-continuation symbol has a unique meaning in its column, usually the first column after the preceding comment-column, it seems like a good idea to connect the multiple-line strings before the parse.

All these pre-processes are easily done in Perl, due to their lexical nature. Also, contrary to the actual extraction phase, they're simple, independent tasks that allow for a short and clear implementation.

7.2 ASF rewrite equations

The actual rewriting of the parse tree to a DocGen-compatible format is done in ASF. The grammar of the target-language is attached to the grammar of the source language by means of the rewrite-module, as explained in section 6.1.1.

7.2.1 Target language

The target-format of the complex islands from our framework in section 6.2 consist of a keyword, linenumber and sometimes an argument. They're preceded by the '@'-symbol and separated by newlines. For instance if we assume the EVALUATE-statement from figure 5.2 to span five lines (one for each keyword, starting at line 1), it would result in the lines

```
@EVALUATE 1
@EVALUATE 2
@END-EVALUATE 4
@END-EVALUATE 5
```

We will call the target-grammar CPF, Conditional Perform Format.

7.2.2 Rewrite module

The rewrite module CIG2CPF, as shown in appendix B, contains the functions to be used and the actual ASF-rules on the functions. Without pretending to be an ASF-manual, we will briefly examine a rule for the EVALUATE-example in the previous section.

Assuming the function

```
g(EVALUATE EVALUATE WATERSTAT WATERID END-EVALUATE .)
```

is reached, the rule

```
h-evaluate(cig-evaluatenestings) = cpf-evaluatenestings
=====================================================
g(EVALUATE cig-evaluatenestings .) =
@EVALUATE 42 cpf-evaluatenestings @END-EVALUATE 42
```

is applied. This rule has the variables

```
cig-evaluatenestings
cpf-evaluatenestings
```

and contains the functions

```
g(CIG-Line) = CPF-Line
h-evaluate(CIG-EvaluateNestings) = CPF-EvaluateNestings
```

The equation above the bar is a condition. If it holds true, the equation below the bar also holds true. The condition results in

```
h-evaluate(EVALUATE WATERSTAT WATERID END-EVALUATE) =
@EVALUATE 42 @END-EVALUATE 42
```

Thus we end up with

```
h-evaluate(EVALUATE WATERSTAT WATERID END-EVALUATE) =
@EVALUATE 42 @END-EVALUATE 42
=====================================================
g(EVALUATE EVALUATE WATERSTAT WATERID END-EVALUATE .) =
@EVALUATE 42 @EVALUATE 42 @END-EVALUATE 42 @END-EVALUATE 42
```

We make a few general observations:

- No line-numbers are retrieved

  All line-numbers in our rewriter are currently 42. This is because we actually have no satisfying way of retrieving line-numbers in our island grammar based rewriter yet (also see chapter 8).

- No newlines exist in the CPF-grammar

  Newlines are simply layout, just like in the CIG-grammar. There are several reasons why newlines are not easily used as something other than LAYOUT (sections 5.4.2, 7.2.3).

- Only real productions can be rewritten in ASF

  Although ASF provides us with an abstraction that allows us to manipulate the parse tree on a level that seemingly hides the actual tree-structure for us, it can only rewrite real productions that are made during parsing. For example, we cannot easily rewrite

  ```
  FIRST SECTION.
  SECOND SECTION.
  ```

  to

  ```
  @SECTION 42 FIRST
  @END-SECTION 42 FIRST
  ..
  ```

  as the production for an actual section-block does not exist (section 5.5.2).

7.2.3 The rewrite-framework

The complete framework of the CIG2CPF rewriter is presented in figure 7.1. We make two important observations in the graph:

- Drop is imported *above* CIG2CPF

  This is currently an essential step: If the equations from CIG2CPF have actual knowledge of the definition of Drop, the parsing of the equations themselves cannot be trusted. What happens is that keywords or variables can be substituted within the equations by the sort Drop without warning. Otherwise accepted equations will be mutilated or will be parsed for seemingly endless periods of time. Something might be wrong with the priorities within equations or our island grammar is just too unorthodox for it to ever parse correct or within reasonable time. Currently the only

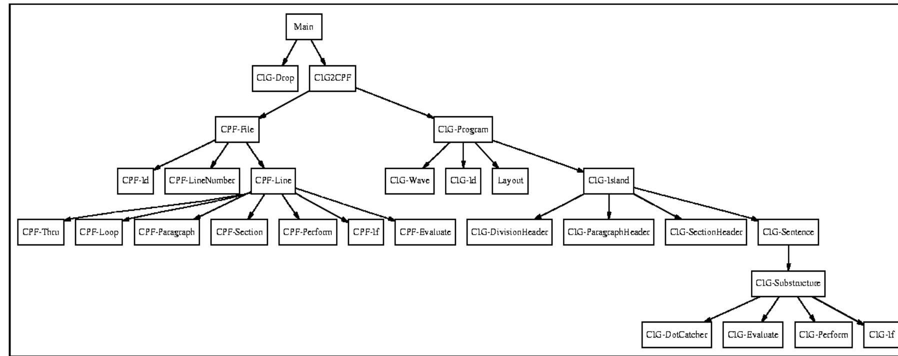

Figure 7.1: SDF import-graph for complete island-rewriter

solution seems to be to hide the definition of Drop from the equations before they are parsed, which can be achieved by importing it above the module with the equations.

- Only one layout can be used

  When several definitions of a sort exist, the union of definitions will be used. This results in the inability to use different LAYOUT-sorts for target and source languages. This prevents us from excluding the newline from the LAYOUT of the CPF-grammar, something which we might want to do because of the newline-separation of lines in the CPF-grammar.

7.3 Post-processing

We apply several small post-processes to our rewrite to mimic the output of the Perl-extractor:

- Newline-separation of the CPF-lines.

  Our rewrite to CPF-grammar did not include a separation of CPF-lines by newlines. We can do this with a small Perl-script by separating on '@'.

- Closing of division-blocks.

  We argued in section 5.5.2 that we can easily close sections and paragraphs by means of a post-process. We did so again by use of a Perl-script, which uses an easy algorithm to close open divisions once a new division-header is detected.

Although not actually done after the rewriting, we also mention the ASF-rule

```
f(cig-line* cig-procdivheader cig-line*2) = f(cig-line*2)
```

which effectively removes all lines before the Procedure Division header from further evaluation.

7.4 Automation of complete rewrite

In order to glue all components of the island-extractor together, we used the GNU Make utility [12]. The resulting Makefile propagates the source through

the pre-processes, island grammar parse, rewrite and post-processes. It allows us to easily change components or study in-between results. As Makefiles can invoke each other and DocGen invokes the separate Perl-extractor components from within its own Makefile, replacing them with island-extractors is an easy task.

Chapter 8

Comparison with original extractor

8.1 Speed

We expect an extractor based on lexical analysis to outperform one based on a syntactic parse. A relatively slow syntactic parse does not matter much for batch-extraction if its speed is still acceptable for sourcefiles of realistic size. The same cannot be said if extraction has to be performed real-time.

8.1.1 ASFIX and Aterms

The native parse tree format for SGLR is ASFIX2 [17]. The trees consist of *annotated terms* or Aterms [1]. ASF only rewrites trees of the older ASFIX format, therefore we need to convert the ASFIX2 trees to ASFIX.

SGLR does not use a tokenizing phase, instead it treats every symbol as a token on itself. This tends to result in very large parse trees. To prevent extreme filesizes, the ASFIX2 trees use term sharing.

If we want to know where terms in the parse tree originated from in the source file, we can turn on position information in SGLR. The originating positions will then be included as annotations in each term. However, as each term will have unique position information, sharing is no longer an option. DocGen uses position information for dynamic links between extraction-results and the original source.

8.1.2 Measurements

To get an idea of the performance of our island extractor, we timed several testcases on a 500mhz AMD-K7 machine (figures 8.1, 8.2, 8.3). Most testcases were generated by concatenating a single COBOL-source serveral times, to make sure no source-anomalies would affect the overal result. Also note that the x-axis of figure 8.1 used a charactercount instead of SGLR's tokencount to determine the amount of tokens, hence the mismatch with the other two graphs. As we are only interested in differences within the graph itself and the points in the

graph were indeed acquired by concatenating the same file, this shouldn't affect results much.

The file from the first points in figures 8.2, 8.1 and fourth point in figure 8.3 consists of 154300 tokens (235946 characters). This file contains 5628 (newline-seperated) lines of COBOL.

Rewrites of the parse tree using the compiled ASF-equations and pre- and post-processes are not included in the measurements. In practice these happen nearly instantaneous.

If we parse sources with default settings, ASFIX2 output with no position information, results are surprisingly favourable for the island grammar. Perl is obviously faster, but the island grammar responded fast enough to even be considered for realtime extraction.

Unfortunately, we're not done yet. The ASFIX2 parse tree needs to be rewritten to the CPF output-format. As the compiled ASF-rewriter only works on ASFIX trees, we need to transform the ASFIX2 tree to ASFIX first. In figure 8.2 we see that this transformation is very inefficient. The conversion uses an algorithm that seems to work in $O(n^2)$ time, and takes too long for the larger filesizes in the relevant domain. We suspect this conversion can be done a lot faster. [1]

If we want to seriously consider using the island extractor with DocGen, we need to know the positions of the islands in the original source. SGLR provides an on-off toggle for position information, which either annotates all or none of the terms with their original position. The inability to maintain term sharing with position information has dramatic effects on performance (figure 8.3). Notice the hook at the end of the otherwise linear graph, an indication of what is causing the problem. At the position of the hook, The size of the parse tree exceeded free real memory at this point, approximately 200mb. This forced the system to turn to virtual memory, which ruins performance even more. The sheer size of the resulting parse trees makes them very awkward to handle.

We aren't interested in position information on *all* of the terms. Only information on the islands will do. Without explanation we mention that the relatively small amount of island-terms and their hierarchical position in the parse tree suggest a large amount of the sharing can be maintained if only terms of the island sort would be annotated.

8.2 Extraction results

8.2.1 Robustness

We want the extraction-phase to be as robust as possible. While failure to scan a file is acceptable as long as it happens during in-house development (and gives us some indication of the problem causing it), it is important that this does not occur with a DocGen system installed at a customer.

[1] ...and so did the MetaEnvironment developers. The algorithm supposedly has been heavily optimised.

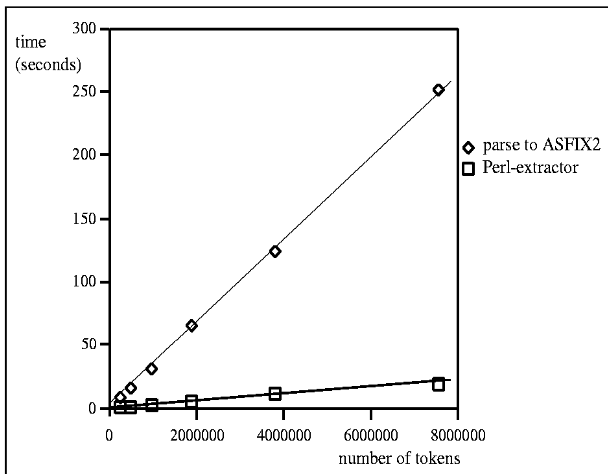

Figure 8.1: Perl-extractor compared to a parse to ASFIX2

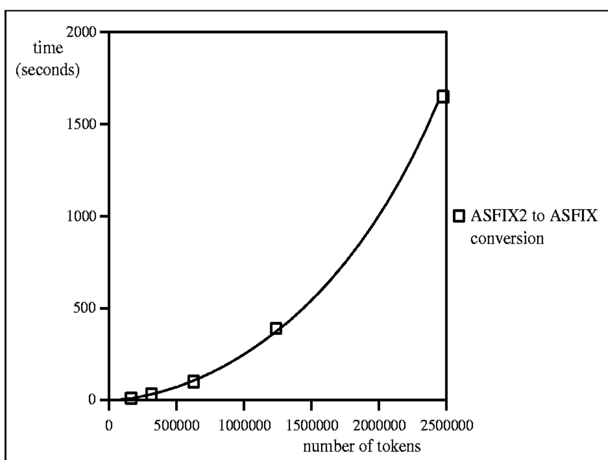

Figure 8.2: ASFIX2 to ASFIX conversion

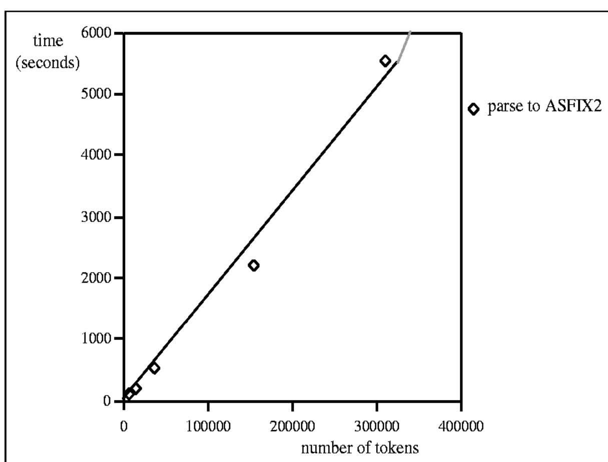

Figure 8.3: Parse to ASFIX2 with position information

The Perl-extractor has the edge here, due to the forgiving nature of its lexical analysis. Sections that do not match with the expected pattern are simply skipped. If the heuristics do not decide on certain doubtful cases (an 'ambiguity'), the regular expression will do so anyway. As a result, most non-robust behaviour will probably be caused by obvious bugs in the Perl-extractor itself. No such bugs showed up during testing.

The island-extractor can suffer from two problems that are inherent to syntax parsing: parse failure and ambiguities. In the case of a parse failure, our parser could not match the source with our grammar. This is almost always due to a failure of our water-definition to catch all irrelevant constructs; in contrast to the lexical analysis the syntax parse will have to accept *all* of the language, not just the patterns we're interested in. The island grammar presented in chapter 5 suffers from at least two occasions that give rise to parse failures:

- The existence of carriage-returns in some source files.

  This is not really a serious problem, but it does demonstrate the robustness of the Perl-extractor. The Perl-extractor may miss out on multi-line patterns, apart from that it is business as usual. The island-extractor however fails on the first carriage-return it encounters. Whether this is sound behaviour (one could argue that Drop should accept carriage-return in its current form) does not really matter: carriage-return conversion is obviously something we want to do before further source analysis.

- Unexpected comment-entries in the Identification Division.

  It turns out that the Identification Division in COBOL can contain comment-style syntax. For example, the Identification Division part

  ```
  AUTHOR. [comment-entry].
  ```

  may look like

  ```
  AUTHOR. D'AGNOSTI.
  ```

  Suddenly we have an apostrophe (') on an awkward position. The Dot-Catcher will try to match it with a closing apostrophe, which may result in a parse failure. If we would add this construct to our Water-definition, we're probably diverting too much from the concept of minimal syntax knowledge; different dialects of COBOL may intervene with our water-definition at unexpected places. The last thing we'd want to do is adapt the water instead of the islands to the dialect.

Parse failures are generally not hard to track down. Typically SGLR will point at the exact location where things went wrong.

Provided our water-land separation is adequate, ambiguities only occur during a parse of an island. No ambiguous parsings occurred during tests with the current island grammar. However, a lot of ambiguities occurred during testing, especially while experimenting with the IF-statement. An ambiguity can easily

sneak undetected in the final grammar, as it may only occur in rare untested forms of a construct. What's worse, we get no feedback whatsoever on the whereabouts of the ambiguity; the native parse tree format of SGLR (ASFIX2) accepts ambiguities. Currently we can browse this parse tree or try to isolate the ambiguity by partitioning the source file. Both methods are awkward to say the least, tools to ease the task are currently in development [9].

8.2.2 Correctness

The island-extractor has an edge in correctness, albeit a questionable one. The island grammar only recognizes patterns that exactly match the specification of an island. As a result, the extracted information is as correct as we specified. In contrast, the Perl-extractor uses heuristics that, while they have some syntactic knowledge, are far more greedy to accept patterns that not completely match what we are looking for. This is especially true for lexical analysis of the statements that can contain nested water in the island grammar.

As an example, consider the term

```
IF IF ELSE ELSE END-IF .
```

In this term, a nested IF-construct is terminated by the ELSE-keyword of the outer IF-construct. Our current island for the IF-statement expects all constructs to be terminated by END-IF or a period instead (appendix B). As a result, this sentence is considered to be water and will not show up on the output of the island-extractor. The Perl-extractor of DocGen did not understand these constructs at first either. Contrary to our island-extractor, it did come up with a resulting output:

```
@IF 1
@IF 3
@ELSE 5
@ELSE 7
@END-IF 10
```

One IF-statement remains unclosed, which may confuse the Perl-extractor later on. However, it *does* give us an indication that something is wrong here [2].

The behaviour of the island-extractor output may be desirable for a system installed at a client, since it only shows correct information. However, if the problem is to be cured it first has to be detected. The parser would have to be extended to report 'doubtful' areas. SGLR already can do this, but not in a way that is practical enough to hunt for would-be islands. Another option is to use a lightweight lexical analysis to double-check the occurences of island-keywords in the source with the extraction results. Of course this would result in a trade-off between being lightweight and reporting irrelevant keyword occurences.

[2]the CPF-grammar from the island-extractor actually provides an excellent tool to test the Perl-output on this sort of anomalies!

8.3 Adaptability

With adaptability we refer to the effort it takes to make changes to the extractor to account for dialect-specific needs.

8.3.1 System design

A good modularization of tasks in the extraction phase eases the implementation of changes; we do not need to guess where a change is to be implemented and it will be less appealing to reimplement certain functionality instead of reusing it.

The island-extractor almost enforces the obvious separation between actual extraction and refining extracted results to the desired results. For example: at one point DocGen examines what device is assigned to a certain variable. Each time something is written to this variable, DocGen reports it to be written to the device belonging to the variable. In the Perl-extractor this mapping of variable onto device is done during extraction. This contributes to the non-uniform way islands are handled in the Perl-extractor. In the island-extractor we could do this in the ASF-section at earliest.

The system design of the Perl-extractor has obviously been affected by shortcomings of the language (section 2.3). The current Java-driven extraction phase has a much better modularization.

8.3.2 Syntactic adaptions

We're usually after a syntactic construct when trying to match a pattern in DocGen. We have shown that the island grammar does little to reduce the expressive power of SDF for syntactic constructs, the islands in this case. This is a far cry from the attempts to express a syntactic construct with the use of heuristic driven regular expressions in Perl. For example, the extractor for the OPEN-statement as shown in appendix B.3 took about half an hour, including testing, to create. If an alternative version of the pattern exists, we do not need to modify the existing pattern match. Instead we simply add another island. We do not need to take special precautions to save specific extracted information during pattern matching either; all information we need is retained in the parse tree.

The expressiveness of SDF comes at a cost, as we can easily break the island extractor if we do not check our specifications for ambiguities. Especially different varieties of a complex island are sensitive to this.

8.3.3 Lexical adaptions

COBOL-syntax depends even more on strange lexical constructs than we have shown so far. For instance, the EXIT SECTION (section 5.5.1) is actually recognized by its column position. If we cannot write a sensible construct in SDF for such a case, pre-processing might grow to a point where the question may arise if we really want to do a syntactic parse at all. None of the preprocesses we mentioned to get our island-extractor up and running are of such complexity to question the sanity of a syntactic parse.

ASF does not yet provide a complete solution for rewriting the island grammar to CPF-format (section 7.2.2). Its inability to properly handle layout and access annotated information from the parse tree force us to resort to additional post-processing with more suitable tools. Most notable is the problem of line-numbers; even when we disregard the impractical trees that are obtained with position information annotations, the simple line-count loop that suffices in Perl is a clear advantage over browsing the parse tree for the right information.

Chapter 9

Conclusion

9.0.4 Island grammar specification in SDF

We succeeded in specifying an island grammar for COBOL in SDF. A combination of injective priorities and the reject-property directs SGLR to parse sources as we intend them to be parsed. The separation between water and islands mostly occurs on a level that is unobtrusive to island specification, which can still be done in a natural way.

The design was troubled by unexpected behaviour of SGLR. Some specification decisions were affected by strange parse results or slow performance. This behaviour is probably the result of both our unorthodox application of SGLR and its still immature status during our design attempts. The latest version may allow for even less obtrusive separation of land and water. In an ideal situation we probably just want to write

```
context-free syntax
Island -> Token{prefer}
Water  -> Token{avoid}
Token* -> Source
```

and not worry about priorities and rejects.

The minimal knowledge approach of our island grammar can result in problems we do not expect in a typical full grammar. We encountered such a problem with periods within strings, as the grammar had no global knowledge of the string-type (section 5.4). In general the ability to specify a grammar for SGLR without first tokenizing the complete source is a big advantage.

9.0.5 Island grammar application in DocGen

Integration

We built a complete extractor that mimics the output of the original Perl-extractors using a rewriter in ASF and some small pre- and post-processes in Perl. A Makefile propagates the rewrite of source-file to extraction-output. Integration with DocGen is a trivial task of replacing the call to the Perl-extractor to a call to the island-extractor.

Island extractor benefits

- Clear and natural specification

  The island-extractor has a clear process-separation. Additional islands can easily be added to its framework using a natural SDF-specification.

- Correctness

  Provided our target-language grammar is correct, the island-extractor can not output wrong constructs. This is much harder to guarantee with the Perl-extractor.

Especially the first benefit is what made us consider island grammars in the first place and still has much appeal to it.

Island extractor weaknesses

- Extraction speed

  Parse speed seems promising if we disregard the inefficient ASFIX2 to ASFIX conversion. However, turning on position information results in unacceptable performance and parse-tree sizes.

- Robustness

  Unexpected ambiguities and parse errors can break a parse at any time. Our island grammar is much more forgiving than a full grammar, but we can not gain robustness equal to that of the Perl-extractor.

- Hard to use for testing purposes

  The same reason that makes the island-extractor beneficial for correctness of output makes it hard to use for testing purposes. There is no mechanism to test for suspect constructs that may qualify as islands, they will simply be seen as water every time they are encountered. The Perl-extractor's word-by-word analysis can easily report suspect constructs. We found such a construct in section 8.2.2.

- ASF-specific weaknesses

  ASF provides a clean, fast and simple way of rewriting a grammar. It does however not completely fulfill its promise; As the same layout-sort is used for source- and target-grammars and the ASF-equations, it is usually impossible to rewrite with layout-specific symbols. If we store position information in the annotated terms of the parse tree, ASF has no means of accessing this information. An alternative post-process would be needed to match the islands with their position information.

In its current incarnation, the island extractor is not yet a viable alternative to DocGen's Perl-counterpart. Especially the position-information bottleneck is too severe to disregard.

9.1 Future work

9.1.1 SGLR

Many problems with the island extractor can be addressed at the level of SGLR:

- Sort-specific position information.

  If we can toggle position information on for just our island sorts, it may be possible to share enough other terms in the parse tree to prevent it from growing to unacceptable size.

- Runtime water removal.

  The position information problem is indirectly caused by the fact that we retain information about the water-sorts in the parse tree. If these could be eliminated from the parse tree once their existence has been confirmed (as opposed to removal after the parse is done), the resulting tree can be much more manageable.

- Report of specific doubtful patterns.

  In its current incarnation, SGLR can show us difficult areas during a parse by means of a counter that tells the amount of tokens parsed. Detection of dubious constructs can be improved if specific troublesome parse-areas involving the island-sort are reported runtime.

9.2 ASF

Our wishlist for ASF is obvious: a solution for the use of layout-specific symbols in constructs and the ability to access annotations. The latest version of ASF allows the use of traversals, it may be interesting to see if the island extractor can benefit from these.

9.3 Alternatives

We tried to add the robustness of a lexical analysis to a syntactic parse. The resulting extractor allows natural specification of the desired patterns with far better robustness than a parse with a full grammar. We can try to combine the strong points of lexical analysis and syntactic parsing in other ways:

In [10] a specification language for a lexical analyser is described that provides a simplified way of writing down regular expressions. The result is a more natural way of writing down otherwise bewildering patterns. Adding the ability of natural construct specification to a lexical analysis may indeed be an interesting alternative to our approach.

A seemingly ideal approach to combine the best of both worlds would be to localize a suspicious section with a lexical analysis and lift this section to parse it. This approach is used in [6]; a scanner marks starting points of sections that qualify for a parse.

An obvious problem would be to determine the dimensions of the suspect section from within the lexical analysis. Instead of determining the dimensions of the suspect section by means of lexical analysis, we can use an *island parser*[11]; a bi-directional parser that can start anywhere within the source.

Bibliography

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[2] M-J. Dominus. *Perl Regular Expression Matching is NP-Complete.* Available at: http://www.plover.com/ mjd/perl/NPC/index.html.

[3] W.B.C. Ebbinkhuijsen. *COBOL, gebaseerd op ANS-COBOL 1974.* Samsom Uitgeverij, 3 edition, 1986.

[4] W.B.C. Ebbinkhuijsen. *COBOL, gebaseerd op ISO-COBOL 1985.* Samsom BedrijfsInformatie bv, 5 edition, 1997.

[5] Jeffrey E.F. Friedl. *Mastering Regular Expressions.* O'Reilly & Associates, Inc., 1997.

[6] Rainer Koppler. A systematic approach to fuzzy parsing. *Software - Practice and Experience*, 27(6):637–649, 6 1997.

[7] R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: http://adam.wins.uva.nl/˜x/grammars/vs-cobol-ii/.

[8] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery, 2000. Available at: http://adam.wins.uva.nl/˜x/ge/ge.html.

[9] Eelco Visser Merijn de Jonge, Joost Visser. Xt - program transformation tools. Available at: http://www.program-transformation.org/xt/.

[10] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3), July 1996.

[11] Wouter Pasman. Incremental parsing. Master's thesis, University of Amsterdam, department of computer science, 8 1991.

[12] Richard M. Stallman and Roland McGrath. *GNU Make - A Program for Directing Recompilation*, 1997.

[13] Thomas A. Sudkamp. *Languages and Machines.* Addison Wesley, 1991.

[14] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*, 2000. Available at: http://www.cwi.nl/projects/MetaEnv/.

[15] Arie van Deursen and Tobias Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[16] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, Programming Research Group, 1997.

[17] Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

[18] L. Wall and R. L. Schwarz. *Programming Perl.* O'Reilly & Associates, Inc., 1991.

Appendix A

Sources of DocGen integration components

A.1 Pre-processing

A.1.1 strip comments (stripper.pl)

```
foreach $line (<STDIN>) {
  $line=~ s/^.{6,6}(.{0,66}).*\n/$1\n/;
  $line=~ s/^[\/*].*//;
  print $line;
}
```

A.2 Post-processing

A.2.1 add newline-seperators (tidy.pl)

```
foreach $line (<STDIN>) {
  $line=~ s/\n//g;
  $line=~ s/@/\n@/g;
  print $line;
}
# close last line
print "\n";
```

A.2.2 close divisions (close.pl)

```
sub close_paragraph {
  if ($paragraph) {
    $paragraph =~ s/\@PARA/\@END-PARA/;
    print $paragraph;
    $paragraph = "";
  }
}
sub close_section {
  if ($section) {
    $section =~ s/\@SECTION/\@END-SECTION/;
```

```
    print $section;
    $section = "";
  }
}
# skip first line, blank due to 'bug' in tidy.pl
$line= <STDIN>;
foreach $line (<STDIN>) {
  if ($line =~ /\@SECTION/) {
    close_paragraph;
    close_section;
    $section = $line;
  } else {
    if ($line=~ /\@PARA/) {
      close_paragraph;
      $paragraph = $line;
    }
  }
  print $line;
}
close_paragraph;
close_section;
```

A.3 Makefile for parsing sources with both parsers

```
STRIPPER = /home/ernst/perl/stripper.pl
GRAMMAR  = /home/ernst/projects/stage/hypos/condperf/Isle.tbl
REWRITER = /home/ernst/projects/stage/hypos/condperf/cdir/Main
TIDY     = /home/ernst/perl/tidy.pl
CLOSE    = /home/ernst/perl/close.pl

LEGACY   := /opt/legacy/hypos-RA/src
TARGETS  := $(wildcard $(LEGACY)/*.cbl) \
$(wildcard $(LEGACY)/*.CBL) \
$(wildcard $(LEGACY)/*.cpy) \
$(wildcard $(LEGACY)/*.CPY)

GRAMMAR2 = /home/ernst/projects/stage/hypos/keyword/Isle.tbl
RELCREATOR = /home/ernst/projects/stage/hypos/keyword/cdir/Main
TIDYREL = /home/ernst/projects/stage/hypos/keyword/tidyrel.pl

#.PRECIOUS: %.function
%.cpf: %.asfix
$(REWRITER) < $< | asource | perl $(TIDY) | perl $(CLOSE) > $@

%.rel: %.asfix2
$(RELCREATOR) < $< | asource | perl $(TIDYREL) $* > $@

%.asfix2: %.function
sglr -p $(GRAMMAR2) -v1 -i $< -o $@

%.asfix: %.function
sglr -p $(GRAMMAR) -vP2 -i $< -o $@
```

```
%.function: %.stripped
echo "f( `cat $< ` )" > $@

%.stripped: $(LEGACY)/%
perl $(STRIPPER) < $< > $@

hypos: $(patsubst %,%.cpf,$(notdir $(TARGETS))) \
$(patsubst %,%.rel,$(notdir $(TARGETS)))

clean:
rm *.cpf
rm *.rel
```

Appendix B

ASF+SDF specifications

B.1 SDF-specification of CIG2CPF framework

```
definition
module CIG-DivisionHeader
exports
 sorts CIG-ProcDivHeader CIG-DivisionHeader

 context-free syntax
  "PROCEDURE" "DIVISION" "."                      -> CIG-ProcDivHeader
  "PROCEDURE" "DIVISION" "USING" Data-name+ "." -> CIG-ProcDivHeader

  ProcDivHeader -> CIG-DivisionHeader

  CIG-Id -> Data-name

module CIG-DotCatcher
exports
 sorts CIG-DotCatcher

 lexical syntax
  "\'" ~[\'\n]* "\'"                                   -> CIG-DotCatcher
  "\'" ~[\'\n]* "\n" "-" [\ ]* ~[\n] ~[\'\n]* "\'" -> CIG-DotCatcher

  "\"" ~[\"\n]* "\""                                   -> CIG-DotCatcher
  "\"" ~[\"\n]* "\n" "-" [\ ]* ~[\n] ~[\"\n]* "\"" -> CIG-DotCatcher

module CIG-Drop
exports
 sorts CIG-Drop

 lexical restrictions
  CIG-Drop -/- ~[\ \t\n\.\'\"]
 lexical syntax
  ~[\ \t\n\.\'\"]+ -> CIG-Drop

module CIG-Evaluate
exports
```

```
 sorts CIG-Evaluate CIG-SubEvaluate CIG-EvaluateNesting
 CIG-EvaluateWhen CIG-EvaluateNestings

 context-free syntax
  "EVALUATE" CIG-EvaluateNestings "END-EVALUATE" -> CIG-SubEvaluate
  "EVALUATE" CIG-EvaluateNestings "."            -> CIG-Evaluate

  "EVALUATE"     -> CIG-Substructure2 {reject}
  "END-EVALUATE" -> CIG-Substructure2 {reject}

  "WHEN" -> CIG-EvaluateWhen

  CIG-SubEvaluate "." -> CIG-Evaluate

  CIG-EvaluateNesting* -> CIG-EvaluateNestings

  CIG-EvaluateWhen -> CIG-EvaluateNesting
  CIG-Substructure -> CIG-EvaluateNesting

 context-free priorities
  EvaluateWhen     -> CIG-EvaluateNesting >
  CIG-Substructure -> CIG-EvaluateNesting

module CIG-Id
exports
 sorts CIG-DataName CIG-Id

 lexical restrictions
  CIG-Id -/- [A-Za-z0-9\-\_]
 lexical syntax
  [A-Za-z0-9\-\_]+   -> CIG-Id

module CIG-If
exports
 sorts
  CIG-If CIG-SubIf CIG-IfNesting CIG-IfNestings

 context-free syntax
  "IF" CIG-IfNestings "ELSE" CIG-IfNestings "END-IF" -> CIG-SubIf
  "IF" CIG-IfNestings "END-IF"                        -> CIG-SubIf

  "IF" CIG-IfNestings "ELSE" CIG-IfNestings "." -> CIG-If
  "IF" CIG-IfNestings "."                       -> CIG-If
  CIG-SubIf "."                                 -> CIG-If

  CIG-Substructure -> CIG-IfNesting

  CIG-IfNesting* -> CIG-IfNestings

  "IF"     -> CIG-Substructure2 {reject}
  "ELSE"   -> CIG-Substructure2 {reject}
  "END-IF" -> CIG-Substructure2 {reject}

module CIG-Island
```

```
imports CIG-Sentence CIG-SectionHeader CIG-ParagraphHeader
  CIG-DivisionHeader
exports
 sorts CIG-Island

 context-free syntax
  CIG-ProcDivHeader -> CIG-Island
  CIG-Sentence -> CIG-Island
  CIG-SectionHeader -> CIG-Island
  CIG-ParagraphHeader -> CIG-Island


module CIG-Line
imports  CIG-ParagraphHeader CIG-SectionHeader CIG-Sentence
exports
 sorts CIG-Line

 context-free syntax
  CIG-ParagraphHeader -> CIG-Line
  CIG-SectionHeader   -> CIG-Line
  CIG-Sentence        -> CIG-Line

module CIG-ParagraphHeader
exports
 sorts CIG-ParagraphHeader CIG-ParagraphId

 context-free syntax
  CIG-ParagraphId "." -> CIG-ParagraphHeader

  "CONTINUE"         -> CIG-ParagraphId {reject}
  "DECLARATIVES"     -> CIG-ParagraphId {reject}
  "END-DECLARATIVES"-> CIG-ParagraphId {reject}
  "DATE-COMPILED"    -> CIG-ParagraphId {reject}
  "GOBACK"           -> CIG-ParagraphId {reject}
  "SPECIAL-NAMES"    -> CIG-ParagraphId {reject}
  "FILE-CONTROL"     -> CIG-ParagraphId {reject}
  "DATE-WRITTEN"     -> CIG-ParagraphId {reject}
  "END-EXEC"         -> CIG-ParagraphId {reject}
  "EXIT"             -> CIG-ParagraphId {reject}
  "SOURCE-COMPUTER" -> CIG-ParagraphId {reject}
  "OBJECT-COMPUTER" -> CIG-ParagraphId {reject}
  "REMARKS"          -> CIG-ParagraphId {reject}
  "AUTHOR"           -> CIG-ParagraphId {reject}
  "PROGRAM-ID"       -> CIG-ParagraphId {reject}

  CIG-Id -> CIG-ParagraphId

module CIG-Perform
exports
 sorts CIG-Perform CIG-SubPerform CIG-PerformNesting
  CIG-PerformNestings CIG-PerformLoopWord

 context-free syntax
  "PERFORM" CIG-Id "THRU" CIG-Id     -> CIG-SubPerform
```

```
  "PERFORM" CIG-PerformLoopWord
   CIG-PerformNestings "END-PERFORM" -> CIG-SubPerform
  "PERFORM" CIG-Id                   -> CIG-SubPerform

  "PERFORM" CIG-PerformLoopWord CIG-PerformNestings "." -> CIG-Perform
  CIG-SubPerform "."                                    -> CIG-Perform

  "VARYING"        -> CIG-PerformLoopWord
  "UNTIL"          -> CIG-PerformLoopWord
  CIG-Drop "TIMES" -> CIG-PerformLoopWord

  CIG-Substructure -> CIG-PerformNesting

  "END-PERFORM" -> CIG-Substructure2 {reject}
  "PERFORM"     -> CIG-Substructure2 {reject}

  CIG-PerformNesting* -> CIG-PerformNestings

module CIG-Program
imports
  CIG-Island Layout CIG-Id CIG-Wave
exports
 sorts CIG-Program CIG-Line

 context-free syntax
  CIG-Island -> CIG-Line
  CIG-Wave   -> CIG-Line

  CIG-Line* -> CIG-Program

 context-free priorities
  CIG-Island -> CIG-Line >
  CIG-Wave   -> CIG-Line

module CIG-SectionHeader
exports
 sorts CIG-SectionHeader CIG-SectionId

 context-free syntax
  CIG-SectionId "SECTION" "."  -> CIG-SectionHeader

  "WORKING-STORAGE" -> CIG-SectionId {reject}
  "CONFIGURATION"   -> CIG-SectionId {reject}
  "INPUT-OUTPUT"    -> CIG-SectionId {reject}
  "FILE"            -> CIG-SectionId {reject}
  "LINKAGE"         -> CIG-SectionId {reject}
  "EXIT"            -> CIG-SectionId {reject}

  CIG-Id -> CIG-SectionId

module CIG-Sentence
imports CIG-Substructure
exports
 sorts CIG-Sentence
```

```
 context-free syntax
  CIG-If       -> CIG-Sentence
  CIG-Perform  -> CIG-Sentence
  CIG-Evaluate -> CIG-Sentence

module CIG-Substructure
imports CIG-If CIG-Perform CIG-Evaluate CIG-DotCatcher
exports
 sorts CIG-Substructure CIG-Substructure1 CIG-Substructure2

 context-free syntax
  CIG-SubEvaluate -> CIG-Substructure1
  CIG-SubIf       -> CIG-Substructure1
  CIG-SubPerform  -> CIG-Substructure1

  CIG-Drop        -> CIG-Substructure2
  CIG-DotCatcher  -> CIG-Substructure2

  CIG-Substructure1 -> CIG-Substructure
  CIG-Substructure2 -> CIG-Substructure

 context-free priorities
  CIG-Substructure1 -> CIG-Substructure >
  CIG-Substructure2 -> CIG-Substructure



module CIG-Wave
exports sorts CIG-Wave CIG-Fluid

context-free syntax
  CIG-Fluid* "." -> CIG-Wave

   CIG-DotCatcher -> CIG-Fluid
  CIG-Drop       -> CIG-Fluid

module CIG2CPF
imports CIG-Program CPF-File
exports
 sorts CPF-IfNestings CPF-IfNesting CPF-Line
   CPF-File CPF-LoopNestings CPF-EvaluateNestings

 context-free syntax
  "f" "("  CIG-Program ")"                 -> CPF-File
  "g" "(" CIG-Line ")"                     -> CPF-Line
  "h-if" "(" CIG-IfNestings ")"            -> CPF-IfNestings
  "h-perform" "(" CIG-PerformNestings ")"  -> CPF-LoopNestings
  "h-evaluate" "(" CIG-EvaluateNestings ")" -> CPF-EvaluateNestings
  "i" "(" CIG-Substructure ")"             -> CPF-Line

  CIG-Id                          -> CPF-Id
  "@PARA" CPF-LineNumber CPF-Id    -> CPF-Line
  "@SECTION" CPF-LineNumber CPF-Id -> CPF-Line
```

```
hiddens
 variables
  "cig-dotcatcher"[1-9]*        -> CIG-DotCatcher
  "cig-drop"[1-9]*              -> CIG-Drop
  "cig-sectionheader"[1-9]*     -> CIG-SectionHeader
  "cig-paragraphheader"[1-9]*   -> CIG-ParagraphHeader
  "cig-procdivheader"[1-9]*     -> CIG-ProcDivHeader
  "cig-sentence*"[1-9]*         -> CIG-Sentence*
  "cpf-section"[1-9]*           -> CPF-Section
  "cig-program"[1-9]*           -> CIG-Program
  "cig-line*"[1-9]*             -> CIG-Line*
  "cig-sentence"[1-9]*          -> CIG-Sentence
  "cpf-file"[1-9]*              -> CPF-File
  "cig-substructure"[1-9]*      -> CIG-Substructure
  "cig-substructureone"         -> CIG-Substructure1
  "cig-substructuretwo"         -> CIG-Substructure2
  "cig-ifnesting*"[1-9]*        -> CIG-IfNesting*
  "cig-performnestings"[1-9]*   -> CIG-PerformNestings
  "cig-performnesting"[1-9]*    -> CIG-PerformNesting
  "cig-performnesting*"[1-9]*   -> CIG-PerformNesting*
  "cpf-loopnestings"[1-9]*      -> CPF-LoopNestings
  "cpf-loopnesting"[1-9]*       -> CPF-LoopNesting
  "cpf-loopnesting*"[1-9]*      -> CPF-LoopNesting*
  "cpf-loop"[1-9]*              -> CPF-Loop
  "cig-evaluatenestings"[1-9]* -> CIG-EvaluateNestings
  "cig-evaluatenesting*"[1-9]* -> CIG-EvaluateNesting*
  "cig-evaluatenesting"[1-9]*  -> CIG-EvaluateNesting
  "cpf-evaluatenestings"[1-9]* -> CPF-EvaluateNestings
  "cpf-evaluatenesting*"[1-9]* -> CPF-EvaluateNesting*
  "cpf-evaluatenesting"[1-9]*  -> CPF-EvaluateNesting
  "cpf-line"[1-9]*              -> CPF-Line
  "cpf-line*"[1-9]*             -> CPF-Line*
  "cig-ifnestings"[1-9]*        -> CIG-IfNestings
  "cig-subif"[1-9]*             -> CIG-SubIf
  "cig-subperform"[1-9]*        -> CIG-SubPerform
  "cig-subevaluate"[1-9]*       -> CIG-SubEvaluate
  "cpf-if"[1-9]*                -> CPF-If
  "cpf-ifnestings"[1-9]*        -> CPF-IfNestings
  "cpf-ifnesting"[1-9]*         -> CPF-IfNesting
  "cpf-ifnesting*"[1-9]*        -> CPF-IfNesting*
  "cig-wave"[1-9]*              -> CIG-Wave
  "cig-id"[1-9]*                -> CIG-Id
  "cpf-paragraphnestings"       -> CPF-ParagraphNestings
  "cpf-paragraphnesting*"       -> CPF-ParagraphNesting*
  "cpf-paragraphnesting"        -> CPF-ParagraphNesting
  "cig-performloopword"         -> CIG-PerformLoopWord

module CPF-Evaluate
exports
 sorts CPF-Evaluate CPF-EvaluateNesting CPF-EvaluateNestings

 context-free syntax
  "@EVALUATE" CPF-LineNumber
```

```
  CPF-EvaluateNestings
  "@END-EVALUATE" CPF-LineNumber -> CPF-Evaluate

  "@WHEN" CPF-LineNumber -> CPF-EvaluateNesting
  CPF-Line               -> CPF-EvaluateNesting

  CPF-EvaluateNesting* -> CPF-EvaluateNestings

module CPF-File
imports CPF-Line CPF-LineNumber CPF-Id
exports
 sorts CPF-File

 context-free syntax
  CPF-Line* -> CPF-File

module CPF-Id
exports
 sorts CPF-Id
 lexical syntax
  [a-zA-Z0-9\_\-]+ -> CPF-Id

module CPF-If
exports
 sorts CPF-If CPF-IfNesting CPF-IfNestings

 context-free syntax
  "@IF" CPF-LineNumber
  CPF-IfNestings
  "@END-IF" CPF-LineNumber -> CPF-If
  "@IF" CPF-LineNumber
  CPF-IfNestings
  "@ELSE" CPF-LineNumber
  CPF-IfNestings
  "@END-IF" CPF-LineNumber -> CPF-If

  CPF-Line -> CPF-IfNesting

  CPF-IfNesting* -> CPF-IfNestings

module CPF-Line
imports CPF-Evaluate CPF-If CPF-Perform CPF-Section
  CPF-Paragraph CPF-Loop CPF-Thru
exports
 sorts CPF-Line

 context-free syntax
  CPF-Evaluate  -> CPF-Line
  CPF-Perform   -> CPF-Line
  CPF-If        -> CPF-Line
  CPF-Section   -> CPF-Line
  CPF-Paragraph -> CPF-Line
  CPF-Loop      -> CPF-Line
  CPF-Thru      -> CPF-Line
```

```
module CPF-LineNumber
exports
 sorts CPF-LineNumber

 lexical syntax
  [1-9][0-9]* -> CPF-LineNumber

module CPF-Loop
exports
 sorts CPF-Loop CPF-LoopNesting CPF-LoopNestings

 context-free syntax
  "@LOOP" CPF-LineNumber
  CPF-LoopNestings
  "@END-LOOP" CPF-LineNumber -> CPF-Loop

  CPF-Line -> CPF-LoopNesting

  CPF-LoopNesting* -> CPF-LoopNestings

module CPF-Paragraph
exports
 sorts CPF-Paragraph CPF-ParagraphNesting CPF-ParagraphId
  CPF-ParagraphNestings

 context-free syntax
  "@PARA" CPF-LineNumber CPF-ParagraphId
  CPF-ParagraphNestings
  "@END-PARA" CPF-LineNumber CPF-ParagraphId -> CPF-Paragraph

  CPF-Line    -> CPF-ParagraphNesting
  CPF-Section -> CPF-ParagraphNesting {reject}

  CPF-ParagraphNesting* -> CPF-ParagraphNestings

  CPF-Id -> CPF-ParagraphId

module CPF-Perform
exports
 sorts CPF-Perform CPF-PerformArg

 context-free syntax
  "@PERFORM" CPF-LineNumber CPF-PerformArg -> CPF-Perform

  CPF-Id -> CPF-PerformArg

module CPF-Section
exports
 sorts CPF-Section CPF-SectionNesting CPF-SectionId
  CPF-SectionNestings

 context-free syntax
  "@SECTION" CPF-LineNumber CPF-SectionId
```

```
  CPF-SectionNestings
  "@END-SECTION" CPF-LineNumber CPF-SectionId -> CPF-Section

  CPF-Line -> CPF-SectionNesting
  CPF-Id   -> CPF-SectionId

  CPF-SectionNesting* -> CPF-SectionNestings

module CPF-Thru
exports
 sorts CPF-Thru CPF-ThruArg

 context-free syntax
  "@THRU" CPF-LineNumber CPF-ThruArg CPF-ThruArg -> CPF-Thru

  CPF-Id -> CPF-ThruArg

module Layout
exports
 restrictions
   <LAYOUT?-CF> -/- [\ \n\t]

 lexical syntax
   [\ \n\t]       -> LAYOUT

module Main
imports
  CIG2CPF CIG-Drop
```

B.2 ASF-equations of CIG2CPF module

```
equations

[f43]
f(cig-line* cig-procdivheader cig-line*2) = f(cig-line*2)
[f0]
f() =

[f1]
f(cig-line*) = cpf-line*
===========================
f(cig-sentence cig-line*) = g(cig-sentence) cpf-line*

[f2]
f(cig-wave cig-line*) = f(cig-line*)

[f3]
f(cig-line*) = cpf-line*
========================
f(cig-id . cig-line*) = @PARA 42 cig-id cpf-line*

[f4]
f(cig-line*) = cpf-line*
```

```
========================
f(cig-id SECTION . cig-line*) = @SECTION 42 cig-id cpf-line*


[g1]
h-if(cig-ifnestings) = cpf-ifnestings
=====================================
g(IF cig-ifnestings .) = @IF 42 cpf-ifnestings @END-IF 42

[g2]
h-if(cig-ifnestings1) = cpf-ifnestings1,
h-if(cig-ifnestings2) = cpf-ifnestings2
====================================
g(IF cig-ifnestings1 ELSE cig-ifnestings2 .) =
@IF 42 cpf-ifnestings1 @ELSE 42 cpf-ifnestings2 @END-IF 42

[g3]
i(cig-subif) = cpf-if
=====================
g(cig-subif .) = cpf-if

[g4]
h-perform(cig-performnestings) = cpf-loopnestings
===========================================
g(PERFORM cig-performloopword cig-performnestings .) =
@LOOP 42 cpf-loopnestings @END-LOOP 42

[g5]
i(cig-subperform) = cpf-line
============================
g(cig-subperform .) = cpf-line

[g6]
h-evaluate(cig-evaluatenestings) = cpf-evaluatenestings
=======================================================
g(EVALUATE cig-evaluatenestings .) =
@EVALUATE 42 cpf-evaluatenestings @END-EVALUATE 42

[g7]
i(cig-subevaluate) = cpf-line
=============================
g(cig-subevaluate .) = cpf-line

[h-if1]
h-if(cig-ifnesting*1) = cpf-ifnesting*1,
h-if(cig-ifnesting*2) = cpf-ifnesting*2
========================================
h-if(cig-ifnesting*1 cig-substructureone cig-ifnesting*2) =
cpf-ifnesting*1 i(cig-substructureone) cpf-ifnesting*2

[h-perform1]
h-perform(cig-performnesting*1) = cpf-loopnesting*1,
h-perform(cig-performnesting*2) = cpf-loopnesting*2
====================================================
```

```
h-perform(cig-performnesting*1 cig-substructureone cig-performnesting*2) =
cpf-loopnesting*1 i(cig-substructureone) cpf-loopnesting*2

[h-evaluate1]
h-evaluate(cig-evaluatenesting*1) = cpf-evaluatenesting*1,
h-evaluate(cig-evaluatenesting*2) = cpf-evaluatenesting*2
==========================================================
h-evaluate(cig-evaluatenesting*1 cig-substructureone cig-evaluatenesting*2) =
cpf-evaluatenesting*1 i(cig-substructureone) cpf-evaluatenesting*2

[i1]
h-if(cig-ifnestings) = cpf-ifnestings
=====================================
i(IF cig-ifnestings END-IF) = @IF 42 cpf-ifnestings @END-IF 42

[i2]
h-if(cig-ifnestings1) = cpf-ifnestings1,
h-if(cig-ifnestings2) = cpf-ifnestings2
========================================
i(IF cig-ifnestings1 ELSE cig-ifnestings2 END-IF) =
@IF 42 cpf-ifnestings1 @ELSE 42 cpf-ifnestings2 @END-IF 42

[i3]
h-perform(cig-performnestings) = cpf-loopnestings
==========================================
i(PERFORM cig-performloopword cig-performnestings END-PERFORM) =
@LOOP 42 cpf-loopnestings @END-LOOP 42

[i35]

i(PERFORM cig-id) = @PERFORM 42 cig-id

[i36]
i(PERFORM cig-id THRU cig-id2) = @THRU 42 cig-id cig-id2

[i4]
h-evaluate(cig-evaluatenestings) = cpf-evaluatenestings
=======================================================
i(EVALUATE cig-evaluatenestings END-EVALUATE) =
@EVALUATE 42 cpf-evaluatenestings @END-EVALUATE 42

[i5]
h-evaluate(cig-evaluatenesting*) = cpf-evaluatenesting*
=======================================================
h-evaluate(WHEN cig-evaluatenesting*) = @WHEN 42 cpf-evaluatenesting*


[h-if42]
h-if (cig-ifnesting*1 cig-substructuretwo cig-ifnesting*2) =
h-if (cig-ifnesting*1 cig-ifnesting*2)

[h-perform42]
h-perform(cig-performnesting*1 cig-substructuretwo cig-performnesting*2) =
h-perform(cig-performnesting*1 cig-performnesting*2)
```

```
[h-evaluate42]
h-evaluate(cig-evaluatenesting*1 cig-substructuretwo cig-evaluatenesting*2) =
h-evaluate(cig-evaluatenesting*1 cig-evaluatenesting*2)

[h-if0]
h-if()=
[h-perform0]
h-perform()=
[h-evaluate0]
h-evaluate()=
```

B.3 CIG-Open island

```
module CIG-Open
exports
 sorts CIG-Open CIG-OpenArg CIG-OpenArgs CIG-Output
  CIG-Input CIG-Extend CIG-IO CIG-InputTail CIG-OutputTail

 context-free syntax

CIG-Id -> CIG-filename

"OPEN" CIG-OpenArgs -> CIG-Open

CIG-Input    -> CIG-OpenArg
CIG-Output   -> CIG-OpenArg
CIG-IO       -> CIG-OpenArg
CIG-Extend   -> CIG-OpenArg
CIG-OpenArg* -> CIG-OpenArgs

"OUTPUT" CIG-filename CIG-OutputTail -> CIG-Output
"INPUT" CIG-filename CIG-InputTail   -> CIG-Input

                     -> CIG-InputTail
"REVERSED"           -> CIG-InputTail
"NO" "REWIND"        -> CIG-InputTail
"WITH" "NO" "REWIND" -> CIG-InputTail
CIG-InputTail        -> CIG-OutputTail
"REVERSED"           -> CIG-OutputTail {reject}

"I-O" CIG-filename+    -> CIG-IO
"EXTEND" CIG-filename+ -> CIG-Extend

"OPEN"     -> CIG-filename {reject}
"OUTPUT"   -> CIG-filename {reject}
"REVERSED" -> CIG-filename {reject}
"EXTEND"   -> CIG-filename {reject}
"REWIND"   -> CIG-filename {reject}
"I-O"      -> CIG-filename {reject}
```