UNIVERSITÄT
BERN

F# parsing expression grammar

Bachelor Thesis

Milan Kubicek
from
Bern BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

15. August 2016

Prof. Dr. Oscar Nierstrasz
Jan Kurš

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

The process of structuring a sequence of characters according to a given grammar is called parsing. Computer program code must be parsed in order to be further processed. Various parsing techniques have been developed since the rise of programming languages mostly relying on bottom-up parsing techniques. Most programming languages are described in deterministic context-free form from which bottom-up parsers can be automatically generated. Unfortunately bottom-up parsers tend to be less human-readable and hard to debug. Recently new parsing methodologies like parsing expression grammars (PEGs) [8] which have compared to traditional parsing techniques similar or better expressive power and parsing performance while promising better grammar composition properties and seamless integration of lexical analysis in parsing. Although PEG implementations for some computer languages exist, the effort of transforming an existing BNF grammar into PEG hasn't been studied yet.

In this thesis we examine and classify the difficulties of developing a PEG parser based on a BNF language specification. For this study we examine the concrete example of F# as it is a mature, open source, cross platform programming language and to our best knowledge, no PEG-based parser for the F# language exists. Our study classifies difficulties when implementing a PEG parser based on a BNF grammar specification. We show that the highest effort to implement a PEG based grammar is due to transformation to a left-recursion free form. Further we demonstrate that AST generation and required post-processing from PEG matching results largely depend on the BNF grammar structure and its differences to the AST structure. We conclude that supplemental grammar documentation not present in BNF, imposes overall the highest implementation effort while developing a PEG based parser.

Acknowledgements

First of all I owe special thanks to Prof. Dr. Oscar Nierstrasz for his calm way of handling things, his valuable inputs and his hospitality at the Software Composition Group. I would like to express my greatest appreciation and deepest gratitude to Dr. Jan Kurš for pushing me and providing valuable guidance throughout this project. Without my beautiful and beloved partner Federica I couldn't take life to the full. Finally I would like to thank the rest of my family, my friends and colleagues for always being there for me.

Contents

1 Introduction

The purpose of a parser is to ensure that a string of symbols conforms to the rules of a computer language and to build a data structure (e.g. abstract syntax trees) which can then be further processed.

Many popular programming languages (Java, C, C++, Python) are formally described in specifications using Backus-Naur Form (BNF) [4], a notation for context-free grammars. Parser creation by hand is a complex and error prone undertaking therefore various tools like parser generators exist (ANTLR [17], Bison [14], YACC [11]) which turn BNF descriptions into ready parsers. Those auto-generated parsers often must be hand-tuned to satisfy context-sensitive language requirements posed by specification details supplemental to the BNF description. Usually this hand-tuning involves the introduction of additional pre- or post-parsing steps, increasing the overall complexity of the often cryptic bottom-up parsing systems themselves.

The formalism of Parsing Expression Grammars (PEGs) introduced in 2004 [8] is getting more and more attention due to its top-down parsing technique which is very close to the mental model of language recognition, its unambiguity as for every valid language string there exists a unique derivation, and its unlimited lookahead capability increasing the range of recognisable languages.

Its modular structure and its ability to parse even a subset of context-sensitive languages without increasing computational complexity compared to traditional parsers makes it an attractive alternative to existing parsing technologies.

The goal of our project is to implement a PEG based parser for the F# language in order to study the difficulties of implementing a PEG parser based on a BNF specification.

The Smalltalk framework PetitParser [12, 18] offers an agile, simple PEG parsing

platform for this purpose.

We take the BNF definition of F# and transform it to a PEG. The transformation of a grammar described in BNF into a PEG brings several problems, namely the inability of PEGs to deal with (in-)direct left-recursion, grammatical ambiguity, as well as associativity and precedence of in-language operators being described outside of the BNF definition. Later we extend the working PEG to simplify the generation of abstract syntax trees (AST) from input. Throughout the implementation section we show how some of the described problems can be tackled by creating simple helper tools and propose strategies to deal with others.

Our study classifies difficulties when implementing a PEG parser based on a BNF grammar specification. We show that the highest effort to implement a PEG based grammar is due to transformation to a left-recursion free form. Furthermore we demonstrate that AST generation and required post-processing from PEG matching results largely depend on the BNF grammar structure and its differences to the AST structure. We conclude that supplemental grammar documentation not present in BNF, imposes overall the highest implementation effort while developing a PEG based parser.

In this thesis we first introduce the concepts of formal grammars and show the relation between computational complexity and the expressive power of a language they recognise before formalising PEGs. Next we introduce the F# programming language and the PetitParser parsing framework. Then we propose an implementation of a PEG based F# parser, starting with the transformation of the F# BNF to PEG and refine the PEG to simplify abstract syntax tree generation. During this process we focus on problems posed by left-recursion, precedence and associativity of language constructs. Finally we classify difficulties of implementing a PEG parser based on a BNF specification, quantify implementation efforts and discuss tool opportunities to support the described development process.

2
Formal grammars

A grammar serves to formally describe a programming language. The concept of a formal grammar was first proposed by Noam Chomsky in 1956[6] as a tuple $(\boldsymbol{N},\ \boldsymbol{\Sigma},\ \boldsymbol{P},\ \boldsymbol{S})$ where

$\boldsymbol{N}$ is a finite set of nonterminal symbols

$\boldsymbol{\Sigma}$ is a finite set of terminal symbols

$\boldsymbol{P}$ is a finite set of productions rule of the form

$$\left(\boldsymbol{\Sigma} \cup \boldsymbol{N}\right)^* \boldsymbol{N} \left(\boldsymbol{\Sigma} \cup \boldsymbol{N}\right)^* \rightarrow \left(\boldsymbol{\Sigma} \cup \boldsymbol{N}\right)^*$$

where * is the Kleene star operator ("zero or more")

$\boldsymbol{S} \in \boldsymbol{N}$ is a starting symbol

Each production rule therefore maps from at least one non terminal to zero or more nonterminal and terminal symbols. A string in $\left(\boldsymbol{\Sigma} \cup \boldsymbol{N}\right)^*$ is called a sentential form if it can be derived from the starting symbol $\boldsymbol{S}$ with a finite number of steps. If a sentential form consists only of nonterminal symbols it is called a sentence. The language of a grammar $\boldsymbol{G}$, is denoted as $\boldsymbol{L}\left(\boldsymbol{G}\right)$ and defined as all those sentences that can be derived in a finite number of steps from the starting symbol $\boldsymbol{S}$. The decision whether a given string $\boldsymbol{\omega}$ is part of $\boldsymbol{L}\left(\boldsymbol{G}\right)$ is the membership problem.

As an example the formal grammar defined as $(\{\boldsymbol{S}\}, \{\boldsymbol{a}, \boldsymbol{b}\}, \{\boldsymbol{S} \rightarrow \boldsymbol{aSb},\ \boldsymbol{S} \rightarrow \boldsymbol{\epsilon}\},\ \boldsymbol{S})$ generates the language $\{\boldsymbol{a^n b^n}\ :\ \boldsymbol{n} \geq \boldsymbol{0}\}$. A derivation for the string aaabbb inside this language consists of the steps $S \rightarrow aSa \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$.

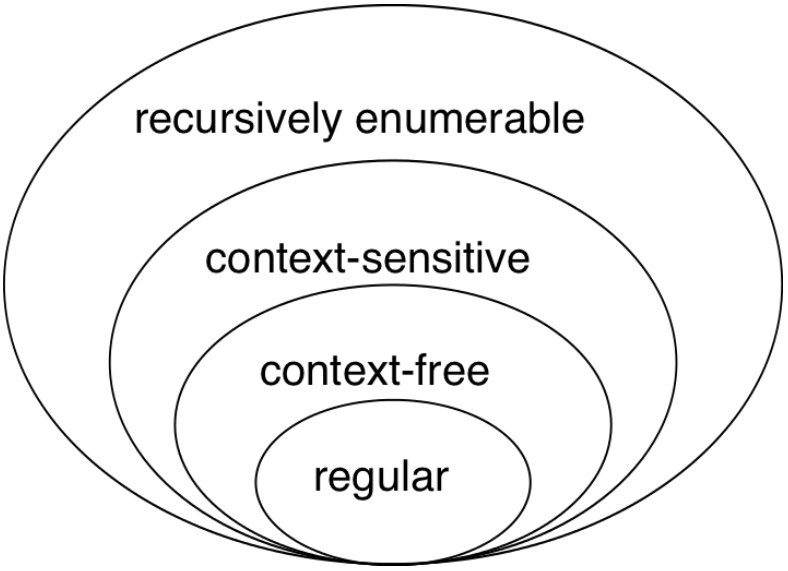

Figure 2.1: The Chomsky hierarchy.

When Noam Chomsky formalized grammars in 1956, he classified them hierarchically into what is now known as the Chomsky hierarchy.[6] The classes in the Chomsky hierarchy differ in the strictness of their production rules and in the expressiveness in the language they generate. All regular languages are also context-free, all context-free languages are context-sensitive and all context-sensitive languages are also recursively enumerable. All inclusions are proper inclusions, so i.e. not all context-free languages can be generated by a regular grammar.

Grammar	Languages	Constraints on production rules
Type-0	Recursively enumerable	$\alpha \to \beta$
Type-1	Context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
Type-2	Context-free	$A \to \gamma$
Type-3	Regular	$A \to \alpha$ and $A \to \alpha B$

Table 2.1: The grammar classes defined by the Chomsky hierarchy.

2.1 Notation techniques

Various notation techniques for context free grammars exist. Most common notations are based on the Backus-Naur Form (BNF) and make use of special operators to deal with

specific applications. Most BNF like notation techniques share a common set of syntax rules and meta-symbols listed in Table 2.2.

Symbol	Description
$\rightarrow$	Assigns a right-hand definition to a left-hand non-terminal token
$\vert$	Or-operator for multiple right-hand definitions
?	Optional-operator for optional right-hand tokens
$+$	One-or-more-operator for repeating right-hand sequences
$*$	Zero-or-more-operator for repeating right-hand sequences

Table 2.2: Meta-symbols and syntax rules of BNF like grammars

2.2 Ambiguity

A context-free grammar which has multiple derivations of the same string is called ambiguous. A common example of ambiguity in programming languages is the dangling else problem. The problem arises when else-statements inside an if-then-else construct are optional. More contextual information is required to determine whether a else-statement inside a nested conditional belongs to the inner or the outer if-then-else construct. Ambiguities in the context-free form are generally resolved in parser implementations by relying on further, sometimes context-sensitive, requirements in pre- or post-parsing steps.

2.3 Complexity

The set of valid programs for almost all programming languages is not context-free due to context-sensitive requirements like “a variable must be declared before it is used”.

Regular grammars can be directly translated into deterministic finite automata, therefore the membership problem can be solved in linear time. Linear complexity can only be guaranteed for this class of languages. In context-free languages the best currently known algorithmic solution for the membership problem is $O(n^{2.81})$[21]. Computing whether a string w is generated by a given context-sensitive grammar requires even more than exponential time, PSPACE and is therefore intractable for practical use.

The Chomsky hierarchy can be further sub-categorised. Especially constructs increasing expressive power of a grammar class without increasing the computational complexity of the membership problem are of great interest for parser developers and linguists.

A variety of techniques exist for parsing context-free languages. Subsets of the CFGs can be parsed using LALR, LR, LL and other parsers in linear time and are therefore interesting for practical parsing.[2, 3] Techniques like GLR[20] and GLL[19] parse all languages generated by CFGs.

For the sake of simplicity and performance, current parsers therefore pass responsibility of context-sensitive language feature requirements to pre- or post-processing routines.

2.4 Parsing expression grammars

Parsing expression grammars (PEG) are a formalism introduced by Bryan Ford in 2004 and are closely related to top-down parsing languages developed by Alexander Birman in the early 1970.[5, 8]

Top-down parsing approaches are very similar to the derivation process in formal grammars. Sentential forms are derived from the starting symbol and then compared to the input string. If the terminal symbols inside a sentential form differ from the input string w, the derivation steps that led to the difference is revoked (*backtracking step*) and if other valid production can be applied the process is continued until either a sentence form is found ($w \in L(G)$) or all possible derivation steps were unsuccessful ($w \notin L(G)$).

The formal definition of a PEG is a tuple $\left(\boldsymbol{N},\ \boldsymbol{\Sigma},\ \boldsymbol{P},\ \boldsymbol{S}\right)$ that looks very similar to the one of formal grammars. Nevertheless especially its parsing rules P, have a different interpretation. Every parsing rule is a relation between a nonterminal symbol A and a parsing expression e of the form $A \leftarrow e$. An atomic parsing expression consists either of a terminal symbol, a nonterminal symbol or the empty string ϵ. A parsing expression is a hierarchical expression consisting of either an atomic parsing expression or a composition of existing parsing expressions, as shown in Table 2.3. If a parsing expression matches input it succeeds and it may consume parts of the input. A failed parsing expression doesn't consume input.

2.4.1 Greedy operator

The operators e^*, $e+$, and $e?$ on a parsing expression e_1 behave greedily. They consume as much input as possible thus letting a following parsing expression fail, if its success depends on one of the already consumed tokens. For example, the sequentially composed parsing expression ${e_1}^*e_2$, where e_1 and e_2 are both parsing expression consuming the same input, will always fail as e_1 does not leave any input for e_2 to consume.

Name	Syntax	Success condition
Sequence	$e_1 e_2$	when both e_1 and e_2 succeed sequantially
Ordered choice	e_1 / e_2	when e_1 succeeds **or** e_1 fails and e_2 succeeds
Zero-or-more	e^*	repeat e zero or more times, always succeed
One-or-more	$e+$	only when e succeeds one or more times
Optional	$e?$	try e, always succed
And-predicate	$\&e$	when e succeeds but doesn't let e consume any input
Not-predicate	$!e$	when e does not succeed but doesn't let e consume any input

Table 2.3: Composed parsing expressions, given any existing parsing expressions e, e_1 and e_2

2.4.2 Ordered choice

A parsing expression grammar can never be ambiguous (see section 2.2) as its ordered choice operator prioritises the first succeeding choice. Therefore a parsing expression has only one unique valid parse tree for a specific input. If a CFG is transliterated directly to a PEG, the choice operator automatically dissolves any ambiguity. By determining a specific order of parsing expressions inside an ordered choice, certain parse trees can be prioritised over others.

2.4.3 Expressive power

It is conjectured that context-free languages exist that cannot be recognised by a PEG but this is yet unproven.

Due to PEGFLs lookahead capability some context-sensitive languages can be parsed. An example is the language $\{a^n b^n c^n : n \geq 1\}$[1] shown in Listing 1.

```
1 S ← &(A'c') ('a')+ B !('a'/'b'/'c')
2 A ← 'a' (A)? 'b'
3 B ← 'b' (B)? 'c'
```

Listing 1: PEG parsing a context-sensitive language

Some CFG rules have to be modified in order to be recognised by PEGs. Due to PEGFLs greedy operator behaviour and ordered choice, CFG rules like $S \leftarrow xSx \mid x$ cannot be implemented directly and have to be rewritten in PEG compatible syntax.

[1] https://en.wikipedia.org/wiki/Parsing_expression_grammar#Examples

Another type of rules requiring modification are rules containing left recursion as discussed in the following subsection.

2.5 Left recursion

Most parsers processing context free grammars in a top-down left-to-right fashion are unable to process left recursion. A grammar is called left recursive, if it contains either direct or indirect left recursion. In this section we first define left recursion before describing an algorithm to transform an arbitrary context-free grammar into a non-left-recursive form.

2.5.1 Direct left recursion

A left-recursive production rule has the form $A \rightarrow A\alpha$, where A is a non-terminal and is recursively defined by reference of itself at the left-most position in its definition. Whenever such a parser tries to parse A at a given position of an input, its first subgoal will be parsing A at the same input position again and again. This results in an infinite loop.

2.5.2 Indirect left recursion

Indirect left recursion is present whenever a grammar production rule can be derived into a sentential form containing direct left recursion. For example substitution on the indirect left recursive grammar production rule set $\{A \rightarrow B|\alpha,\ B \rightarrow C|\beta,\ C \rightarrow A\alpha|\gamma\}$ yields $A \rightarrow A\alpha$, which contains direct left-recursion. Therefore the grammar is left-recursive.

2.5.3 ϵ production rules

In BNF definitions it is common to use production rules yielding the empty string, ϵ. This allows parser developers to write many grammars in a compact way. The most common ϵ *productions* are production rules containing the *zero-or-more* or the *optional* operators. These production rules can cause "hidden" left-recursion in a grammar. If an ϵ production is placed left-most in an production containing direct left-recursion, the direct left-recursion might not be immediately noticeable any more.

Because left-recursion is quite common in programming languages and a lot of parsers can't deal with them, it is necessary to rewrite production rules containing left-recursion beforehand.

2.5.4 Left recursion detection

By traversing a grammar graph, direct and indirect left recursion can be detected by maintaining specialised sets of already traversed vertices. We propose a concrete implementation of a left recursion detection algorithm for PEGs later in subsection 6.4.2.

2.5.5 Left recursion removal

We describe an algorithm to remove left recursion based on Aho et al.[3, p.176-178]. Any context-free grammars containing left recursion can be transformed into a non-left-recursive grammar by performing the following steps:

Let $A, B \in \{N\}$
Let $\alpha_1, ..., \alpha_n, \beta_1, ..., \beta_n \in \{N \cup \Sigma\}$

1. Eliminate ϵ in all production rules:

```
1 ∀A_P ∈ P where A → ϵ | α_1 | α_2 | ... do
2     A_P := A → α_1 | α_2 | ...
3     ∀B_P ∈ P where B → β_1 A β_2 do
4         B_P := B → β_1 A β_2 | β_1 β_2
```

Line 1, 2: From every production rule A containing ϵ remove ϵ.
Line 3, 4: For every B-production rule where A is referenced right hand side, introduce a new rule leaving A out.

2. Remove all direct left recursion:

```
1 ∀A_P ∈ P where A → Aα_1 | ... | Aα_n | β_1 | ... | β_n do
2     A'_P := A' → α_1 A' | ... | α_n A' | ϵ
3     A_P := A → β_1 A' | ... | β_n A'
4     P := { P ∪ A'_P }
5     Remove ϵ rules in P by applying step 1
```

Line 1: Repeat the following steps for all production rules A containing direct left recursion.
Line 2: Introduce a new non-terminal A' and define its production rules as follows: Take the previous direct left recursive rules, remove its left recursive part and add A' as suffix.
Line 3: Replace all production rules of A by the previously non direct left-recursive production rules of A followed by A'.
Line 4: Add the newly generated rules for A' to P.

3. Remove all indirect left recursion:

```
1 Set any particular order N_order, ∀A ∈ N: A_1, A_2, ..., A_n ∈ N_order
2 for i:=1 to n do
3     for j:=1 to i-1
4             Replace each production A_i of the form A_i → A_jβ
5             by A_i → α_1β | ... | α_nβ
6             where A_j → α_1 | ... | α_n are all current A_j-productions
7     Remove any direct left recursion in A_i by applying step 2
```

Line 1: Establish any arbitrary order of all non-terminal symbols in N.
Line 2: Loop through all non-terminal indices i of the ordered set N_{order}.
Line 3: Loop through all indices j smaller than current i.
Line 4-7: Expand all current A_j productions which are left hand side definitions in A_i of the form $A_i \rightarrow A_j\beta$. Each iteration guarantees that the resulting A_i is free of any immediate right hand side self-references. After each iteration, possibly introduced direct left recursion is removed as described earlier. As after all iterations each A_i is free of any immediate right hand side self-references the grammar becomes left recursion free.

By applying the described algorithm it is possible to remove left recursion in context free grammars. Unfortunately the described procedure has some significant drawbacks. In the worst case the number or production rules of the transformed grammar grows exponentially compared to the original grammar. Some parse trees in the new grammar do change. Left-associative constructs for example are right-associative in the new grammar. The above technique for left recursion removal can be optimised and is a subject of research. A careful selection of the topological order of the non-terminals in step 3 for example, can yield more compact, left-recursion free grammars [16].

3
F#

In this chapter we give a brief overview of the F# language. First we describe the languageFLs syntax alternatives, we then introduce its documentation and finally we give the reader a short overview of the official F# parser implementation.

3.1 Introduction

F# was introduced by Microsoft Research and the F# Software Foundation in 2005. It is further developed and maintained by its creators and individual contributors. F# is a strongly typed, functional, imperative and object oriented programming language. The language originates from the ML programming language [15] and was influenced by many programming languages like OCaml, C#, Python, Haskell, Scala and Erlang. Although the F# project was initiated by Microsoft it is open source. A multi-platform compiler and tools are available from the F# Software Foundation.

3.2 Syntax

There are two forms of syntax available for many constructs in the F# language, namely the verbose syntax and the lightweight syntax. The verbose syntax is less sensitive to whitespace characters whereas the lightweight syntax depends on them in order to signal the beginning and end of constructs. The lightweight syntax is shorter, as keywords can be omitted. Verbose constructs may be used in lightweight syntax but not vice versa.

Incorrectly indented constructs in lightweight syntax cause incorrect interpretation of the code or cause the compiler to issue warnings or fail the compilation process.

```
1 let rec fib n =
2   if (n=1||n=2) then begin
3    1 end
4   else begin
5    let result = fib(n-1)
6                 + fib(n-2) in
7    result end in
8
9 for i in 1 .. 10 do
10  printfn "%d:%d" i (fib i)
11 done
```

Listing 2: Verbose syntax

```
1 let rec fib n =
2   if (n=1||n=2) then
3    1
4   else
5    let result = fib(n-1)
6                 + fib(n-2)
7    result
8
9 for i in 1 .. 10 do
10  printfn "%d:%d" i (fib i)
11
```

Listing 3: Lightweight syntax

Some of the differences between verbose and lightweight syntax can be seen in Listing 2 and Listing 3. Whereas the lightweight example uses indentation to indicate block structure, the keywords `begin` and `end` are required for that purpose in verbose syntax, as can be seen in lines 2&3 and lines 4&7 of Listing 2.

The `in` keyword in the verbose syntax example line 6, binds the declared variable to the consequent expression. In this case the consequent expression is just the variable itself, acting as a return value. In lightweight syntax, this keyword can be omitted by aligning the consequent expression with the `let` keyword in line 5.

Listing 2 line 7, contains another `in` keyword which binds the declared function to the consequent for-loop. The lightweight syntax omits the `in` keyword but relies on alignment of the for-loop keyword `for` with the `let` function declaration keyword.

Most code blocks in verbose syntax require a closing keyword, like the `done` keyword in line 11. In lightweight syntax code blocks span over all language constructs that are on-site of the first token of the block.

3.3 Documentation

In this section we give an overview of the F# 3.0[1] language specification[1].

The F# Language Specification explains in-depth technical aspects of the F# language constructs and required compiler behaviour. Lexical analysis, grammar, types including

[1] In January 2016 the F# Foundation finalised version 3.1, introducing just minor changes. A working draft of the version 4.0 is available and can be found along side other releases under `http://fsharp.org/specs/language-spec/`

type inference algorithm, expressions, patterns and other topics are discussed in detail.

3.3.1 BNF specification

The F# language specification [1] contains a description of the F# syntax in BNF which consists of approximately 230 grammar production rules. Informal sections in the language specification resolve ambiguities in the BNF grammar and impose contextually sensitive requirements like the precedence of constructs (discussed in subsection 3.3.2) on the language constructs. Some production rules contain large numbers, mostly further documented sub-production rules. The F# expression rule shown in Listing 4 contains more than 50 terminal and non-terminal sub-rules and relies on further specification of priority and associativity clarifications outside of its BNF definition to allow thorough understanding of the F# expression structure.

```
expr  → const
      | '(' expr ')'
      | 'begin' expr 'end'
      | long-ident-or-op
      | expr'.'long-ident-or-op
      | expr expr
      | expr'('expr')'
      | expr'<'expr'>'
      | expr infix-op expr
      | prefix-op expr
      | expr'.['expr']'
      | expr'.['slice-range']'
      | expr'.['slice-range','slice-range']'
      | expr'←'expr
      | expr(, expr)+
      | 'new' type expr
      | '{' 'new' base-call object-members interface-impls '}'
      | '{'field-initializers '}'
      | '{'expr 'with' field-initializers '}'
      | '['expr(; expr)+']'
      | '[|'expr(; expr)+'|]'
      | expr '{'comp-or-range-expr'}'
      | '['comp-or-range-expr']'
      | '[|'comp-or-range-expr'|]'
      | 'lazy' expr
      | 'null'
      | expr':'type
      | expr':>'type
      | expr':?'type
      | expr':?>'type
      | 'upcast' expr
      | 'downcast' expr
```

```
33        | 'let' function-defn 'in' expr
34        | 'let' value-defn 'in' expr
35        | 'let' 'rec' function-or-value-defns 'in' expr
36        | 'use' ident '=' expr 'in' expr
37        | 'fun' argument-pats '→' expr
38        | 'function' rules
39        | 'match' expr 'with' rules
40        | 'try' expr 'with' rules
41        | 'try' expr 'finally' expr
42        | 'if' expr 'then' expr (elif-branches)? (else-branch)?
43        | 'while' expr 'do' expr 'done'
44        | 'for' ident '=' expr 'to' expr 'to' expr ('done')?
45        | 'for' pat 'in' expr-or-range-expr 'do' expr ('done')?
46        | 'assert' expr
47        | '<@'expr'@>'
48        | '<@@'expr'@@>'
49        | '%' expr
50        | '%%' expr
```

Listing 4: F# expression CFG production rule defined in the F# specification on p.272)

3.3.2 Precedence and associativity of language constructs

Although BNF grammars ensure whether a language construct belongs to a language or not, in most cases further information is required to eliminate ambiguity. The string 1 + 2 is a valid language construct but its parse tree is ambiguous as the grammar in Listing 4 can produce this construct from the rules defined in line 6 and line 9. Moreover, a parser requires additional knowledge about the precedence and associativity of expressions. The F# language specification contains a dedicated section to deal with precedence of symbolic operators, pattern and expression constructs.[1, p.35].

Table 3.1 transcribes the precedence and associativity tables from the specification. It shows the order of precedence from highest to lowest and the associativity of symbolic operators and pattern/expression constructs. The `OP` -suffix represents any token that begins with the given prefix, unless the token appears elsewhere in the table.

3.3.3 Discrepancies between specification and implementation

The authors of the F# language specification note that discrepancies may exist between the F# parser implementation and its specification. In this section we document some of the discrepancies we discovered while working with the F# 3.0 BNF grammar specification.

Operator or expression	Associativity
`f<types>`	left
`f(x)`	left
`.`	left
`prefix-op`	left
`"| rule"`	right
`f x`	left
`lazy expr`	
`assert expr`	
`**OP`	right
`*OP, /OP, \%OP`	left
`+OP, -OP`	left
`:?`	not associative
`::`	right
`^OP`	right
`!=OP, <OP, >OP, =, |OP, &OP`	left
`:>, :?>`	right
`&, &&`	left
`or, ||`	left
`,`	not associative
`:=`	right
`->`	right
`if`	not associative
`function, fun, match, try`	not associative
`let`	not associative
`;`	right
`|`	left
`when`	right
`as`	right

Table 3.1: F# operator and expression priority and associativity, ordered by priority.

3.3.3.1 Missing delimiters

Some BNF non-terminal symbols are referenced with -s suffix, without further specification what the -s suffix implies. Two affected rules are `interface-impls` [1, p.273] and `val-decls` [1, p.279]. The -s suffix is used both as one-or-more repetition operator and zero-or-more repetition operator throughout the grammar definition. Consistent use of the one-or-more notation, where necessary with an optional quantifier,

would resolve this specification ambiguity.

3.3.3.2 Required non-terminals

The `type-defn-elements` non-terminal in `class-type-body` is falsely marked optional [1, p.278] and correctly defined in [1, p.124]. `as-defn` is incorrectly marked as required in `additional-constr-defn` [1, p.279]. The `new`-keyword in `additional-constr-init-expr` is marked as required but is implemented as optional.

3.3.3.3 Missing definitions

`interface-signature` [1, p.278] and `stmt` [1, p.279] both are referenced but not defined.

3.3.3.4 Inconsistent declaration of lightweight syntax keywords

Some keywords are optional in lightweight syntax but the BNF grammar does not follow a consistent definition of these, as some are marked as optional and others as required. `begin` and `end` keywords are marked as optional in `module-defn` but as required in `module-defn-body` [1, p.270].

3.4 AST

The F# language specification[1] does not contain any details about the implemented AST as it serves as an internal structure in the compilation process. More than 70 distinct AST node types are used by the open edition F# 3.0[2] implementation of which 40 describe expression constructs (see Listing 4). We list the major AST nodes in Appendix section A.1.

[2] `http://github.com/fsharp/fsharp/tree/fsharp_30`

4
PetitParser

PetitParser is a parsing-expression grammar based, top-down, parser combinator framework written in Smalltalk [12, 18]. It was developed by Lukas Renggli as part of his work on the Helvetia system, a tool for dynamic language embedding. PetitParser is designed to fit the dynamic nature of Smalltalk. It uses four parser methodologies:

a) *Scannerless Parsers* perform tokenization and parsing in a single step, rather than first breaking individual characters into words and then arranging these words into phrases.[22]

b) *Parser Combinators* represent a parser in a graph like structure enabling high level of modularisation, maintainability and interchangeability.[10]

c) *Parsing Expression Grammars*. PEGFLs ordered choice results in an unambiguous parsing tree for every valid string in the described language. PEGs are discussed in detail in section 2.4.

d) *Packrat Parsers* ensure that each parsing function is invoked at most once at a given position in the input stream in order to enable linear time parsing.[7]

With parsing contexts PetitParser gains the computational power of a Turing machine and allows parsing of some context sensitive languages.[13] A set of ready-made parsers in PetitParser can be composed to consume and transform arbitrary complex languages using common Smalltalk syntax.

4.1 Terminal parsers

The simplest pre-defined parsers are the terminal parsers shown in Table 4.1. With the help of terminal parsers, terminal grammar symbols can be modelled from literal sequences of characters/numbers. An extendable set of symbol ranges can be accessed with the help of factory methods.

Syntax	Matches
`$a asParser`	character 'a'
`'a' asParser`	character 'a'
`'abc' asParser`	string 'abc'
`#any asParser`	any character
`#digit asParser`	any digit 0-9
`#letter asParser`	letters a-z and A-Z
`#word asParser`	letters a-z, A-Z and any digit 0-9
`#lowercase asParser`	letters a-z
`#uppercase asParser`	letters A-Z
`#blank asParser`	horizontal tab and space bar characters
`#newline asParser`	carriage return and line feed characters

Table 4.1: Subset of pre-defined PetitParser terminal parsers.

4.2 Parser combinators

Parser combinators shown in Table 4.2 are used to combine simple parsers into more complex ones. This allows arbitrary grammar reuse and composition.

Name	Syntax	Matches
sequence	p1, p2	p1 followed by p2
ordered choice	p1 / p2	if p1 cannot be matched, match p2
zero-or-more	p star	p zero or more times
one-or-more	p plus	p one or more times
optional	p optional	p if present else nothing
and-predicate	p and	p but doesn't consume it
not-predicate	p not	requires p to fail and doesn't consume

Table 4.2: Subset of pre-defined PetitParser parser combinators.

4.3 Action parsers

In order to get a more convenient representation of the matched input of a parser, it is necessary to process the parsing result. Action parsers from Table 4.3 add ability to transform or perform certain actions on a parser. Using the `==> aBlock` selector a parser is converted into an `PPActionParser` where the block processes the preliminary parsing result turning it into the desired form.

Syntax	Action
`p ==> aBlock`	transforms the result of the parser given aBlock given aBlock.
`p flatten`	creates a string from the result of p.
`p token`	creates a token from the result of p.
`p trim`	trims whitespaces before and after p.

Table 4.3: Subset of pre-defined PetitParser action parsers.

4.4 Layout sensitive parsing support

Layout sensitive parsing support has been recently added to the PetitParser framework by the PetitIdent extension[9]. The extension allows marking specific column positions of the input by pushing indentation tokens which can further be used to check other tokens for either off-side, aligned or on-side positions relative to the pushed indentation token.

4.5 Parsing with PetitParser

Parsers in PetitParser can be implemented using an object oriented approach by deriving from the `PPCompositeParser` base class. Every instance variable corresponds to an element of the grammar and is accessible by accessor-methods returning a dedicated parser for the language construct. Parsers can be referenced directly via instance variables as `PPCompositeParser` uses reflection to look up corresponding methods and store them in instance variables.

The `PPCompositeParser` class sets up all defined parsers as `PPDelegateParser`s to avoid problems caused by mutually recursive definitions. Missing accessor-methods for any production cause the parser to fail on initialisation. A start method forms the entry point for parsing of the specified `PPCompositeParser` subclass.

Test-driven development plays an important role in supporting the evolution of a parser as they can become complex. The `PPCompositeParserTest` class offers a dedicated infrastructure similar to the one for creating parsers.

A PetitParser parser can be divided into multiple layers each with individual responsibilities. First a language grammar can be modelled by defining terminal parsers which subsequently can be composed into productions. A parser then can inherit from this grammar class and process the result of productions using action parsers into an AST. This approach includes benefits like modularity and re-usability of grammar elements.

5 Implementation challenges of PEG-based parsers

Implementing a PEG-based parser poses multiple challenges. In this chapter we first explain the difficulties related to transforming a BNF based grammar rules into PEG based grammar rules, then we describe the generation of AST and finally we show limitations of the PetitParser framework and how they affect a PetitParser based parser. In chapter 7 we classify the described difficulties based on the manual effort and the possibility for tool support and/or automatability.

5.1 BNF to PEG

The F# specification [1] defines the F# language using a BNF ruleset containing 250 grammar rules. Some requirements on the language syntax are not expressed by the specified BNF but are subject to supplemental documentation, such as indentation sensitive lightweight syntax (see section 3.2), expression, operator precedence and associativity (see subsection 3.3.2).

Our goal, transforming the F# BNF grammar into a PEG equivalent, is composed of multiple challenges. PEGs left recursion limitation and ordered choice result in work-intensive transformations increasing the PEG grammars size compared to its BNF form.

Using the simplified expression BNF definition in Listing 5 we will illustrate the described difficulties. This basic expression rule describes the language constructs

expression application print 2 , **prefixed constants** +3 , **arithmetic expressions** 1 + 2 , **tuples** 3, 1, 2 , **expression sequence** print 1; print 2 and compounds of these.

```
expr  → const
      | expr expr
      | pre-op expr
      | expr inf-op expr
      | expr (',' expr)+
      | expr ';' expr

const  → 'print' | '1' | '2' | '3'
pre-op → '+' | '-'
inf-op → '+' | '-' | '*' | '/'
```

Listing 5: Simplified F# expression rule

5.1.1 Left recursion

PEG supports neither direct nor indirect left recursion (see section 2.5) thus all left recursion has to be removed from a PEG grammar. Direct left recursion in the sample expression rule is highlighted in Listing 6. The definition of `expr` refers to itself multiple times on the leftmost side.

```
expr → const
     | expr expr
     | pre-op expr
     | expr inf-op expr
     | expr (',' expr)+
     | expr ';' expr
...
```

Listing 6: Direct left recursion in expression rule

Left recursion removal (see subsection 2.5.5) induces new semantically redundant intermediate rules increasing the grammarFLs size. As the interpretation of these intermediate rules requires some rethinking, they have to be structured and named accordingly to restore grammar readability.

5.1.2 Ordered choice

The fundamental difference between a BNF rule set and a PEG is that the PEG's choice operator is ordered (see subsection 2.4.2). If a parsing function succeeds other parsing functions of the same choice set won't be invoked at the same position in the

same context. With the same rule definition order as in the sample grammar in Listing 5 a PEG wouldn't parse the string `1+2` as `expr infix-op expr` but as `1+2`, `expr expr` because `+2` is recognised as `pre-op expr`. Therefore re-ordering of the BNF based grammar rules is required to ensure the correct PEG parsing behaviour.

5.1.3 Indentation

In real life the F# language is mostly used in its indentation sensitive lightweight syntax. The F# compiler uses a specialised pre-processing scanner phase inserting indentation tokens into the input which allow indentation handling later in the parsing process. As PEGs are scannerless, layout has to be taken into account during the parsing process. Preliminary layout sensitive parsing support has been recently added to PetitParser (see 4.4).

5.2 CST to AST

Given a compatible grammar and an input, PEG generates a parse path if one is available. This parse path, also called concrete syntax tree (CST), is the specific sequence of rules describing how the parser interprets the input. For further processing a more abstract and simplified syntactic representation, the AST is required.

5.2.1 Priorities

While some priority and associativity definitions of F# language constructs are defined in the languageFLs BNF definition others are more informally described in a separate sections of the specification document (see subsection 3.3.2). Some priority and associativity requirements can be handled directly in the syntactical PEG while others, especially arithmetic expression, are better dealt with during the AST generation phase as they would reduce clarity and comprehensibility of the PEG.

5.3 Validation

Pure recognition of a language sentence by a PEG is no evidence for correct parsing. Analysis of parse trees is required for validation. The AST output of our PEG-based parser has to be compared against an independent parser to validate the PEG and AST generation implementations.

5.3.1 F# AST

To enable meaningful validation of our F# PEG implementation it is important to choose an appropriate representation of the parser output. As the F# specification[1] does not contain any details about the F# AST, its structure and documentation has to be extracted from the F# open edition compiler[1] source code.

5.4 PetitParser environment

In this section we show some of the difficulties one can face when implementing complex parsers in PetitParser.

5.4.1 Grammar structure

In PetitParser each parsing function is represented as instance variable internally. The instance variable number per instance is limited to 255 due to design decisions of the hosting environment. Terminal grammar symbols (e.g. keywords) can take up a large fraction of the available parsing function instance variables. As left recursion removal and other required PEG specific grammar refactoring tend to increase grammar size the parsing function number limit can be exceeded quickly. Therefore the PetitParser grammar has to be divided and structured accordingly to reduce the parsing function count.

5.4.2 Parsing performance

A PEG can be correctly implemented and parse a specific language but its parsing performance might suffer from invoking the same parsing function at a given position on the input stream. Packrat Parsers [7] avoid invocation of a parsing function if their result was previously known by memoizing all partial parsing results at a given position. This ensures linear parsing time but results in extensive memory usage. The PetitParser framework implements packrat parsing but the functionality must be activated manually in each parsing function.

The example BNF grammar in Listing 5 has multiple constructs sharing the same left-most non-terminal (`expr expr` and `expr inf-op expr`). Without further optimisation a transformation from BNF to PEG (focusing sole on left recursion removal and ordered choice) results in an inefficient PEG where parsing expressions are evaluated multiple times at the same position in the input stream. A PEG suffers from worse than linear parsing after being transformed from BNF if no precautions are taken.

[1] `https://github.com/fsharp/fsharp`

5.4.3 Left recursion

Like all PEG-based parsing frameworks, PetitParser is neither able to handle direct nor indirect left recursion. Moreover occurrence of left recursion during parsing causes a crash of the PetitParser hosting environment. Therefore every unnoticed or accidentally introduced left-recursion slows down development due to crash recovery and search for left recursion. As long as this fatal restriction is present in the PetitParser development environment, a left recursion detection tool provides the only viable workaround.

6

Implementation of a PEG-based F# parser

The implementation chapter is divided into three parts. First we show how we transformed the F# BNF grammar into a compatible PEG, then we describe how we generated the F# AST in PEG and finally we explain the implementation of the PetitParser F# parser and additional tools. Although it might seem that the development process we describe is sequential, in reality it is more iterative as for example later grammar structuring stages can introduce left recursion or have ordered choice implications.

6.1 BNF to PEG

We present our approach of implementing the F# PEG grammar on the basis of its BNF specification in this section. Our implementation focuses on a partial F# grammar covering the most commonly used expression constructs.

All BNF rules from the F# specification [1] have been transformed manually into a PEG compatible rule set. To illustrate the BNF to PEG transformation a simplified expression rule in BNF is shown in Listing 7.

```
expr  → const
        | expr expr
        | pre-op expr
        | expr inf-op expr
        | expr(','expr)+
        | expr';'expr

const  → 'print' | '1' | '2' | '3'
pre-op → '+' | '-'
inf-op → '+' | '-' | '*' | '/'
```

Listing 7: Simplified F# BNF expression rule

6.1.1 Left recursion

When the simplified expression grammar in Listing 7 is implemented in PetitParser without further modification, it will fail due to the (direct) left recursive definitions. Left recursion can be removed by manually applying the algorithm described in subsection 2.5.5. The resulting left recursion free grammar is shown in Listing 8. As the left recursion removal algorithm introduces new intermediate rules, it obscures the rule-set considerably as logical sequences like `expr inf-op expr` are split into two grammar rules.

```
expr  → (const)expr'
        | (pre-op expr)expr'

expr' → ( expr)expr'
        | ( inf-op expr)expr'
        | ((',expr)+)expr'
        | (';'expr)expr'
        | ϵ

const  → 'print' | '1' | '2' | '3'
pre-op → '+' | '-'
inf-op → '+' | '-' | '*' | '/'
```

Listing 8: Expression BNF rule after left recursion removal

6.1.2 Ordered choice

This left recursion free grammar is PEG compatible and input can be matched using a PEG parser interpreter. Due to PEGFLs ordered choice (see subsection 2.4.2) possible BNF ambiguities are resolved by prioritising the first successful parse tree. Therefore the input $\boxed{1+2}$ is recognised by the grammar in Listing 8, but results in the wrong parse tree `expr (expr)` → `expr (pre-op expr)`. To solve this problem the

grammar rule responsible for infix operator application, line 5 in Listing 8, has to be moved before the application rule, line 4 in Listing 8.

Now that the grammar is left recursion free and its choice sub-rules are prioritised, it can be translated directly into the valid PEG grammar shown in Listing 9.

```
1  expr  ← (const)(expr')?
2          / (pre-op expr)(expr')?
3
4  expr' ← ( inf-op expr)
5          / ( expr)
6          / (','expr)+
7          / (';'expr)
8
9  const  ← 'print' / '1' / '2' / '3'
10 pre-op ← '+' / '-'
11 inf-op ← '+' / '-' / '*' / '/'
```

Listing 9: PEG compatible expression rule

6.2 CST to AST

Although the PEG grammar in Listing 9 matches expression input strings with the correct CST, its structure is lacking in terms of comprehensibility because logically bound grammar rules are split. For example, the input $\boxed{1+2}$ representing an F# infix operator application expression is derived from the CST `expr` → `(const)(expr')` → `const( inf-op expr)` from the PEG grammar in Listing 9. The effort to transform CSTs into ASTs especially depends on the similarity between the two structures. Re-arranging the PEG grammar and logical grouping of rule definitions can simplify this transformation by getting the grammar structure closer to the AST structure. Later post-processing then ensures internal AST node structure details.

6.2.1 Embedding AST structure

By introducing new intermediate rules for each AST node (`exprTup` , `exprInfApp` , `exprApp` and `exprConst`) and grouping sub-rules, our PEG basic expression grammar results in Listing 10. The effort to deduce an AST node type from a CST of this grammar is reduced to the minimum, because each AST node type is part of the CST.

```
expr       ← exprSeq
           / exprTup
           / exprInfApp
           / exprApp
           / exprConst

exprInfApp ← (exprConst)( inf-op expr)

exprApp    ← (exprConst)( expr)

exprTup    ← (exprInfApp / exprApp / exprConst)((','expr)+)

exprSeq    ← (exprTup / exprInfApp / exprApp / exprConst)((';'expr)+)

exprConst  ← const
           / (pre-op expr)

const      ← 'print' / '1' / '2' / '3'
pre-op     ← '+' / '-'
inf-op     ← '+' / '-' / '*' / '/'
```

Listing 10: PEG compatible expression rule restructured for AST generation

6.2.1.1 Redundancies

The grammar in Listing 10 contains redundancies as the sub-rule definitions have overlapping left-hand definition sets, reducing performance and maintainability of the PEG.

Performance suffers as some parsing expressions are applied multiple times at the same position of the input stream. For instance to parse a constant, the grammar tries to match sequentially `exprSeq`, `exprTup`, `exprInfApp`, `exprApp`, `exprApp` and lastly succeeding with `exprSeq` on on the input. As `exprConst` is a valid sub-rule of all these parsing expressions, this parsing expression is applied successfully multiple times and backtracked, resulting in redundant computations.

Grammar maintainability is worsened by repetition of ordered choice sub-sequences in the definitions of `expr`, `exprInfApp` and `exprSeq` and violates the reusability principle. A new grammar rule with higher priority than `exprConst` but lower priority than `exprApp` can't be inserted easily as it has to be incorporated into three grammar rules.

6.2.2 Resolving redundancies

To resolve the redundancies described in subsubsection 6.2.1.1 we can chain the parsing expressions and mark specific right hand definitions optional. Listing 11 shows a grammar where this chaining was applied. This parsing expression chaining has the

drawback of AST node types not being immediately visible in the concrete syntax parse path. Nonetheless the AST node type can be easily deduced recursively during AST node generation by the presence of a specific right hand definition parsing result. For example matching of a constant results in the parse path `expr` → `exprSeq` → `exprTup` → `exprInfApp` → `exprApp` → `exprConst`. All available non-terminal parsing expressions are present in the parse tree, but as all parsing expressions except `exprConst` are missing their suffix, only `exprConst` is returned as the matching result.

This PEG structure now matches all input without backtracking and is easier to maintain as it doesn't contain duplicates.

```
1  expr       ← exprSeq
2  exprSeq    ← (exprTup)(';'expr)*
3  exprTup    ← (exprInfApp)(','expr)*
4  exprInfApp ← (exprApp)( inf-op expr)?
5  exprApp    ← (exprConst)( expr)?
6  exprConst  ← const
7               / (pre-op expr)
8
9  const      ← 'print' / '1' / '2' / '3'
10 pre-op     ← '+' / '-'
11 inf-op     ← '+' / '-' / '*' / '/'
```

Listing 11: PEG compatible expression rule restructured for AST generation with no redundancies

6.2.3 Post-processing

Although most of the AST structure is incorporated into the grammar in Listing 11 some post-processing is required during the final parsing phase, the AST node generation. Some of the AST structure generated during post-processing could be directly generated by PEG grammar extensions, while other parts differ to widely from the given PEG recursive descent given structure.

6.2.3.1 PEG near AST structures

Arithmetic expression AST node compositions change when their arithmetic operators (with different priorities) are interchanged. Figure 6.1 and Figure 6.2 show the different AST node structures of the arithmetic expressions `A * B + C` and `A + B * C`. Arithmetic expression operator prioritisation can be incorporated into a PEG grammar directly. Nevertheless, as operator precedence plays no role in sole input matching and results in additional, hard to maintain grammar rules and more backtracking we suggest

post-processing of the PEG matching result to generate AST nodes with correct internal structure based on the supplemental BNF documentation.

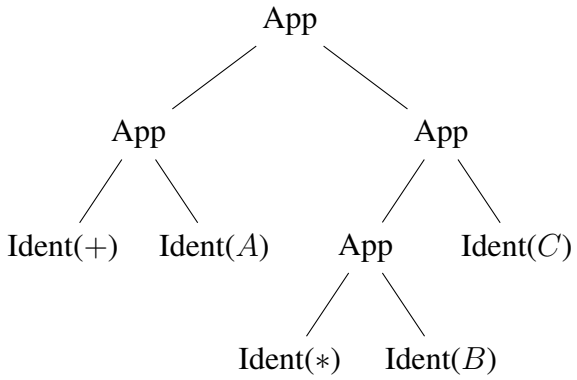

Figure 6.1: F# AST of the expression $A + B * C$

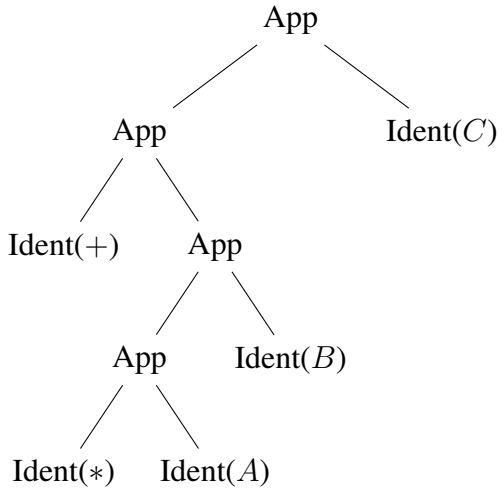

Figure 6.2: F# AST of the expression $A * B + C$

6.2.3.2 PEG far AST structures

AST nesting structure sometimes differs from the PEG parse result dramatically. Grammar adjustments to produce the same AST structure might be hard to realise or even not possible. For example the result of the expression sequence, `exprSeq`, contains a list of consecutive expressions which have to be hierarchically ordered. This hierarchical ordering can be performed relatively easy by assigning each element as child of the

```
exprIf       ← 'if' expr 'then' expr (exprIf-elif)* (else)?

exprIf-elif  ← 'el-if' expr 'then' expr
exprIf-else  ← 'else' expr
```

Listing 12: PEG if-else block

```
if cond1 then expr1
elif cond2 then expr2
elif cond3 then expr3
...
elif condn then exprn
else exprx
```

Listing 13: F# if-elif-else block

preceding element. The construction of other nested AST nodes follow similar principles. AST node structures often have some peculiarities complicating their generation. For instance, some BNF grammar rules define syntactic sugar constructs. The F# if-else block grammar rule shown in Listing 12 defines syntactical sugar syntax by providing else-if branches preventing messy nested if-else blocks. Internally the F# parser converts if-elif-else blocks like the one shown in Listing 13 into a nested if-else AST node hierarchy shown in Listing 12.

6.3 Validation

In this section we show how our parser implementation and our tool for left recursion detection were validated.

6.3.1 PetitParser F# parser

Our PetitParser F# grammar and parser were developed in a test driven way using the PetitParser grammar testing framework. The PetitParser testing framework facilitates the creation of tests for every defined parsing expression in a grammar encouraging modular testing. The parsing expressions in our implementation are backed with test cases extracted from real life F# code using our F# AST extractor described in subsection 6.4.3. Furthermore we validated our implementation by converting several hand-picked F# source files[1] manually into verbose syntax and comparing it against the official implemen-

[1] `https://fsharpsamples.codeplex.com` and `http://fsharp3sample.codeplex.com`

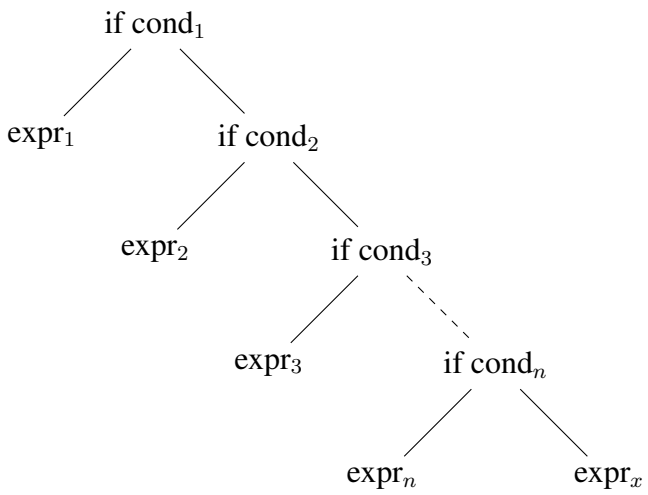

Figure 6.3: AST representation of the F# if-elif-else block from Listing 13

tation output using our F# AST extractor. As our grammar and parser implementation is restricted to the most basic signature, class, type constructs section 6.1 and does not implement the lightweight syntax an empirical validation on real-life F# projects is not possible. A thorough PEG implementation of the F#grammar including support for the indentation sensitive lightweight syntax would go beyond the scope of our study.

6.3.2 Left recursion detection tool

Our left recursion detection tool analyzes PetitParser grammars containing direct and indirect left recursion in all their variants as described in subsubsection 6.4.2.1. Furthermore, our implementation corresponds to the definition of left recursion from section 2.5.

6.4 PetitParser F# parser and tools

In this section we describe the PetitParser F# parser and peripheral support tools implementation.

6.4.1 PetitParser F# parser

We divided our parser implementation into lexical, grammar and parser layers which we discuss at an abstract level in this section.

In summary the lexical layer is responsible for matching terminal tokens like keywords, constants and for trimming unwanted constructs (e.g. comments) between other lexical tokens. The syntactical layer combines tokens provided by the lexical layer into

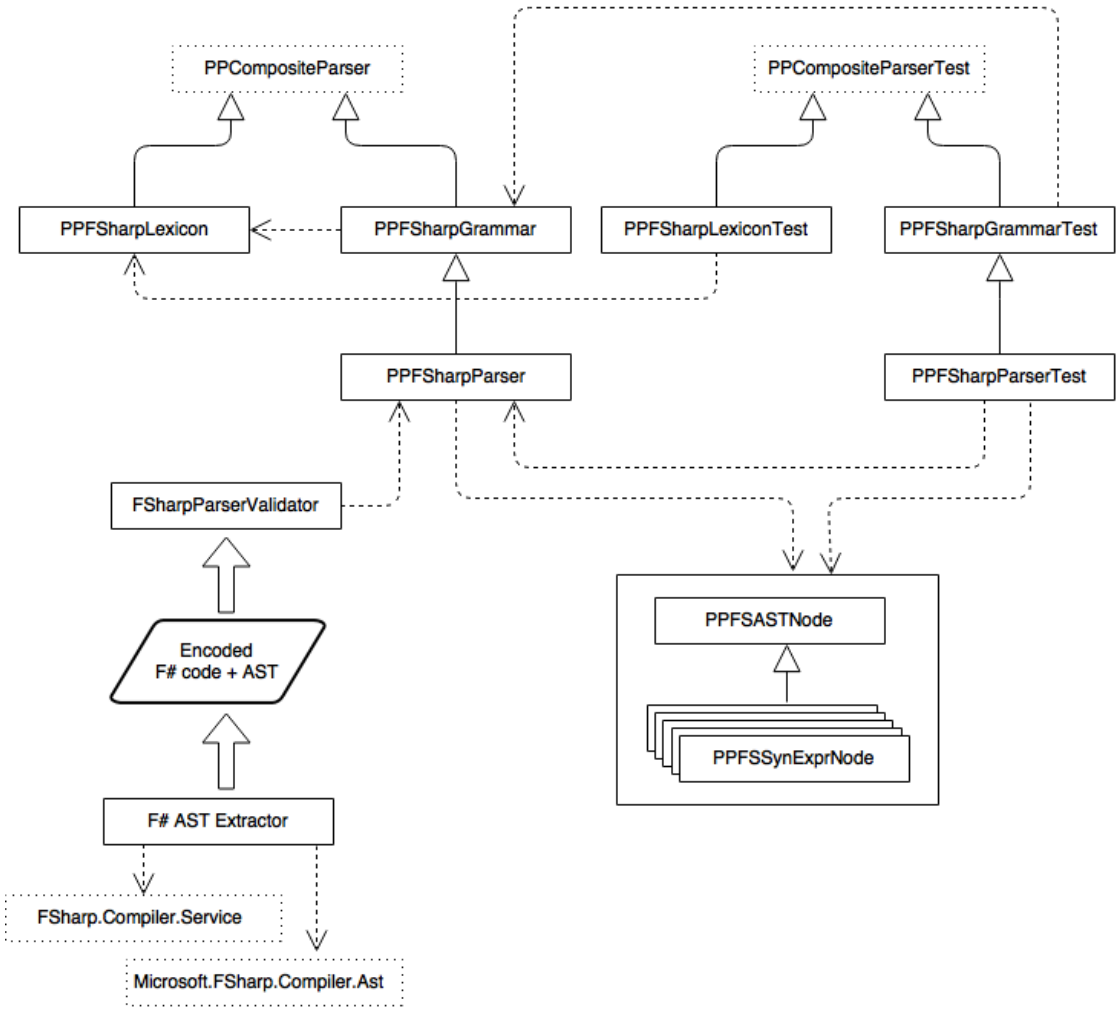

Figure 6.4: PetitParser F# lexicon, grammar and parser implementation overview

valid language sentences and is the core of our parser implementation as it defines the whole language grammar. The parser layer is responsible for processing the result of the syntactical layer and producing the AST of matched inputs. An overview of the implemented parser including test and validation components is shown in Figure 6.4. For an in-depth technical description of our implementation we refer the reader to section B.3.

6.4.1.1 Lexicon

A separate parsing expression definition of terminal language tokens instead of defining the parsing expressions ad-hoc, yields the advantage of providing a central hook-up for trimming unwanted characters (or constructs) in the input stream. Furthermore other

language constructs like custom variable names depend on an aggregation of all available language terminal tokens as they cannot be identical to keywords.

In order to avoid reaching the supported parsing function count limit described in subsection 5.4.1, instead of defining each terminal language token as a separate parsing expression, the lexer stores them all in an internal collection. A public function in the lexer provides access to parsing expressions generated from the string tokens present in this collection.

6.4.1.2 Grammar

The grammar layer implements the F# PEG grammar resulting from the BNF to PEG transformation described in section 6.1. Some grammar rules have been grouped to facilitate later AST generation in the parsing layer like described in section 6.2.

The grammar layer consists of approximately 210 non-terminal parsing expressions and references parsing expressions from the lexer. The grammar layer can in conjunction with the lexer match all valid language constructs. In its matching result the majority of the available information about the parse tree is preserved to allow further post-processing.

6.4.1.3 Parser

By inheriting from the grammar layer, specific parsing expression functions can be overridden and their results can be transformed into a desired form. We use this approach to generate F# AST nodes by transforming the matching results based on the post-processing ideas described in subsection 6.2.3. By overriding 62 parsing expression functions of the overlying grammar layer, we are able to generate the AST nodes of all F# expressions and basic F# script and class definitions.

6.4.2 Left recursion detection

We propose two different approaches for detecting left recursion in PetitParser grammars. The first approach is a tool that detects left recursion before parsing is performed (static algorithm). Second we propose an extension to the PetitParser core that allows on-the-fly left recursion detection while parsing (dynamic algorithm).

The left recursion detection algorithm can be derived from the definition of left recursion: PetitParser grammars are composed into a graph like structure where each vertex is a parsing expression. Left recursion was defined in section 2.5 as applying a specific parsing expression at a given position on an input stream indefinitely. If a parsing expression succeeds, it may consume some input and change the position inside the input stream. To detect left recursion in parsing expressions we are therefore looking for cycles inside the grammar graph where the recurring parsing expression is connected

through a chain of potentially non-consuming parsing expressions. Non-consuming parsing expressions are parsing expressions matching ϵ as well as and- and not- predicate parsing expressions.

6.4.2.1 Static algorithm

The idea described above can be translated into the following left recursion conditions for the different composed parsing expression types described in Table 4.2. Traversing the graph beginning from the starting parsing expression while maintaining a set of visited non-consuming parsing expressions (open set), a parsing expression contains left recursion if it was added to the open set earlier or one of the following parsing expression type specific conditions is satisfied:

Parsing expression type	Left recursion condition
Sequence	First consuming parsing expression in the sequence contains left recursion given the current open set *or* Sequence begins with one or multiple non-consuming parsing expressions and any of them is left recursive given the current open set *or* Any of the remaining parsing expression is left recursive given a new, empty open set
Ordered choice	*Any* of the contained parsing expressions is left recursive given the current open set
One-or-more	Encapsulated parsing expression is left recursive given the current open set
Non-consuming *(Zero-or-more, Optional, And-predicate, Not-predicate)*	Encapsulated parsing expression contains left recursion given the current open set

Table 6.1: Left recursion conditions per parsing expression type

To ensure the check for left recursion terminates after a finite number of steps, a set of processed parsing expressions is kept. We refer to this set as the closed set. When a parser doesn't contain left recursion it is added to the closed set. The closed set is used as a premature recursion exit condition to prevent the check for left recursion more than once.

When a parser is checked for left recursion that is not part of the closed set, the open set (all parsers that didn't consume any symbols of the input stream) represents the left recursion context: If the parser itself or any of its non-consuming left-side descendants are present in the open set, left recursion is present.

The implementation of the described algorithm can be found in appendix subsection B.7.1. A PetitParser grammar can be checked for left recursion occurrence in linear time as our algorithm checks every parsing expression present in the PEG only once.

6.4.2.2 Dynamic algorithm

Ad-hoc left recursion detection during PetitParser parsing is possible by keeping track of the current parsing context and the position in the stream. If the same parsing context occurs in the same position in the input multiple times, left recursion is present. subsection B.7.1 lists the implementation of the described dynamic left recursion check. Although the idea of the dynamic algorithm may be more simplistic and natural than its static counterpart, significant performance and space overhead results from storing the context and input position pair in every parsing step.

6.4.2.3 Static vs. dynamic left recursion detection

While the static implementation does the job it is designed for, it needs to be triggered by the developer after grammar changes prior to parsing, to detect left recursion. The dynamic implementation we propose has the advantage of being relatively simple, well incorporated into the PetitParser core and seamless for developers to use. Unfortunately the performance penalty it introduces and the fact that itFLs left recursion detection depends on the parsing input[2] doesn't make up for its advantages over the static implementation. To combine the advantages of both static and dynamic implementation, the static implementation may be triggered automatically during the first parse with a PetitParser grammar.

6.4.3 F# AST extraction

Our F# AST extractor is based on the official F# Open Edition Compiler implementation. Its output, a textual AST representation of F# code, can be used to validate independent F# parser implementations.

The Open Edition of the F# Compiler[3] contains a compiler services namespace in its core library exposing partial access to the parsing and compiling pipeline. An overview

[2] PEGFLs ordered choice chooses the first matching parse tree. A PEG therefore doesn't necessarily traverses a parse path containing left recursion during parsing if one or more left recursions are present in the grammar.

[3] `https://github.com/fsharp/fsharp/tree/fsharp_30`

of the exposed functionality is listed in subsection B.6.1.

We use the F# Compiler Services API to access the untyped AST from F# code, traverse the AST and produce a textual representation of the AST that can be used to validate our parser implementation.

The AST is encoded into a textual representation. Each node has an identifier and one or multiple values inside brackets. The AST is therefore encoded in nested parenthesised expressions as can be seen in Listing 14, representing the F# expression `A + B * C`.

```
PFSApp(PFSApp(PFSIdent(operator(+)),PFSIdent(A)),PFSApp(
   PFSApp(PFSIdent(operator(*)),PFSIdent(B)),PFSIdent(C)))
```

Listing 14: Textual representation of the AST nodes of the expression A + B * C

Our F# AST extraction implementation is publicly available on GitHub[4]. Appendix section B.5 provides more information about the F# Compiler Services and how to install and use the F# AST extractor.

[4] https://github.com/mkubicek/FSharpAST

7 Implementation effort

Although the difficulties of implementing a PEG-based parser from a BNF grammar described in chapter 5 are to some extent interleaving, in this chapter we classify them based on the manual effort and the possibility for tool support and/or automatability. Table 7.1 gives an overview of the categorized implementation difficulties.

Problem		Tool support	Manual effort
BNF to PEG	Left recursion removal	medium[1]	high
	Ordered choice	low	moderate
CST to AST	Embedding AST structure	-	high
	Redundancy removal	medium[1]	high
	Post-processing	-	low
PetitParser environment	Left recursion handling	high	high
	Parsing performance	medium[1]	high[2]

Table 7.1: Problem classification

[1] Estimated

[2] Based on the effort for grammar redundancy removal

7.1 BNF to PEG

In this section we discuss the implementation effort of a pure PEG grammar based on a BNF grammar definition.

7.1.1 Left recursion removal

Application of the described left recursion removal algorithm is straightforward but requires certain effort. Performed by hand it is easy to lose focus when introducing intermediate rules for the nearly 30 direct left recursion occurrences present alone in the F# expression BNF definition. Accidentally introduced indirect left recursion can be hard to find in a large grammar hierarchy without tool support.

Tool support In subsection 2.5.5 we describe in pseudocode how left recursion can be eliminated. We did not implement the described algorithm to investigate the automatability of the process but assume its practicability as we applied it manually in a large number of cases. Although an automatic approach might be feasible, it obscures the grammar considerably as new intermediate rules are being introduced automatically by the process as shown in subsection 6.1.1. The process of manual left recursion removal benefits from PetitParserFLs functionality to visualise grammars as graphs, providing an overall view on the grammar hierarchy.

Manual effort As we performed the process with foresight of later AST structuring, our left recursion removal effort estimation is obscured. However we estimate the effort rather high, because the structural differences between a grammar and its left recursion free counterpart worsen the comprehensibility and thus complicating the process. Left recursion removal resulted in the introduction of approximately 45 new intermediate rules.

7.1.2 Ordered choice

PEGs ordered choice requires a BNF grammarFLs rules to be shifted based on their relative precedence. We found it difficult to match the formally described BNF grammar rules with the informally described precedence of the F# specification. As grammar restructuring can easily break the delicate order of grammar rules, postponing of ordered choice related grammar changes to the very final PEG tweaking steps can save time during development. Especially the embedding of AST grouping into the grammar is a major intervention performed on a grammar and has the tendency to devastate previous ordered choice related modifications.

Tool support Again PetitParserFLs functionality to visualise grammars as graphs assists the developer in this step. We do not see any need to support the developer further due to the low complexity of this implementation step.

Manual effort Rule shifting to comply with ordered choice isn't a complex transformation but results in moderate effort due to the large number of grammar rule arrangements and the fact that ordered choice related adjustments are often required after other grammar modifications.

7.2 CST to AST

Difficulties originating from the generation of the AST from a CST are discussed in this section.

7.2.1 Embedding AST structure

The specified F# BNF grammar doesn't reflect its AST structure defined by the official F# language implementation. Matching of the grammar rules to AST structure benefits from prior F# coding experience and requires study of the available source code.

Tool support We do not see any possible automation nor tool support option for this step as the relation between grammar rules and AST is not formally described.

Manual effort As grammar rules differ after left recursion removal and ordered choice refactoring from the original BNF definition, embedding an AST structure into the PEG grammar requires high manual effort to match the AST nodes with the corresponding grammar rules and re-group them if necessary. Introduction of described redundancies is very probable and previous ordered choice arrangements are hard to retain due to the invasive modifications. For each AST node this step introduces a new intermediate rule. In our case approximately 40 new intermediate AST rules have been added.

7.2.2 Redundancy removal

A precondition for this step is knowledge about the grammar rules precedence and a solid instinct for the grammar as it is easy to lose track of the large grammar structure. Also, careful test driven development reduces the risk of introducing potentially hard to find bugs in the PEG like subtle but wrong parse path changes.

Tool support PetitParserFLs grammar visualisation is once again a great aid to present parsing expression chains at a glance. Adaption of AST structure in the previous step has resulted in many partially identical sequences throughout the grammar. We believe those sequences might be identified by a tool to assist the developer by proposing the next chain link.

Manual effort As the grammar has to be modified for the most part in this step, the manual effort to perform redundancy removal through chaining is high. The total number of parsing expressions remains untouched after this step.

7.2.3 Post-processing

Ideally the need for post-processing for the generation of AST nodes from a CST is as low as possible after incorporation of the AST structure into the PEG. Whenever the internal AST arrangement deviates in a way that cannot be incorporated well into, or introduces comparatively high impact on the comprehensibility of a PEG, post-processing provides a good alternative.

Tool support We did not observe any automatable processes nor the possibility for tool support in post-processing as post-processing requirements differ widely depending on grammar constructs.

Manual effort Our endeavour to incorporate AST structure into the PEG resulted in lower efforts to generate AST nodes. We couldn't avoid post-processing in some instances but the effort required for implementation rather was low thanks to previous rule grouping based on AST structure.

7.3 PetitParser environment

In this section we discuss the main problems we encountered during implementation of our parser in PetitParser.

7.3.1 Parsing performance

We observed parsing time degeneration during the development of our PEG based parser in the order of several magnitudes due to redundancies in the PEG specification. Redundancies in a PEG cause parsing inefficiencies as described in subsubsection 6.2.1.1.By applying the parsing expression chaining technique described in subsection 6.2.2, we were able to improve the parsing performance without the use of PetitParser memoization.

Tool support Memoization can be useful when optimisation of a grammar is too complex. PetitParserFLs parsing profiling functionality offers assistance in the discovery of performance bottlenecks. PetitParserFLs profiling capability could be further enhanced by monitoring the parsing process and informing the PEG developer about extensive parsing expression applications at a given position of the input.

Manual effort As performance bottlenecks are mostly caused by grammar structure deficiencies, the effort to remove them is identical to the effort of grammar redundancy removal, described in subsection 7.2.2.

7.3.2 Left recursion handling

PetitParser, like other PEG based parsers can not handle left recursion. Even worse left recursion occurring during parsing causes crashes of PetitParserFLs hosting environment we used during our study. The left recursion detection tool we introduced in subsection 6.4.2 therefore proved essential during our parser development in PetitParser.

Tool support Our left recursion detection tool might be automated to warn the developer of left recursion, even before it causes an unsuccessful parse attempt. We consider the benefits of left recursion detection tools in general for PetitParser developers as high due to the described side effects.

Manual effort The effort of dealing with left recursion in a PetitParser grammar is high as not only the input matching fails but in our case also crashed the development environment completely.

8 Conclusion

In this work we have implemented a PEG parser based on a BNF specification with supplemental documentation. We have divided the implementation into several steps and we have classified effort to implement each of these steps.

We have identified left recursion elimination as a strenuous and the only step in the development process of our PEG parser which depends solely on the BNF language definition and not on supplemental documentation. PEGFLs ordered choice implied rule re-ordering (to comply with priority requirements of language constructs) results in moderate effort one is faced with throughout the development process. AST inspired re-grouping of grammar rules results in a high manual effort and may break established ordered choice settings and introduce new left recursion but facilitates later post-processing for AST generation at a large scale.

Based on these observations, the general implementation effort is reduced when grammar rules are grouped according to AST structure first, then left recursion is removed, later ordered choice prioritisation of language constructs is set and finally parsing results are post-processed to generate AST output.

We have seen that BNF supplemental language documentation is responsible for a great extent of the overall parser implementation effort. As traditional parser implementations have to deal with same challenges posed by supplemental language documentation but are in most cases able to deal with left recursion, we consider the transformation to a left recursion free grammar form as the most significant overhead in PEG parser development compared to development of traditional parsers.

A Appendices

A.1 F# AST

In this section we document the AST nodes created by the F# parser. The definition of the F# AST can be found in the `Microsoft.FSharp.Compiler.Ast` namespace in the `ast.fs` file[1].

A.1.1 Expression nodes

Relevant expression CFG rules are listed in the F# specification [1, p.272].

Paren

`(expr)`	A parenthesized expression.

[1] `https://github.com/fsharp/fsharp/tree/fsharp_30/src`

Quote

```
<@ expr @>
<@@ expr @@>
```

Quoted expressions are expressions that are delimited in such a way that they are not compiled as part of the main program but instead are compiled into an object. They are part of the code quotations language feature which enables generation of an abstract syntax tree that represents F# code. This AST later can be traversed and processed according to the needs of an application.

Const

```
true
false
123
```

Represents constants. See subsection A.1.3.

Typed

```
expr : <typename>
```

Binds a type information to an expression. The F# compiler deduces the exact type for all constructs during compilation. If there is not enough information for the compiler to deduce a type, additional type information is required provided by this type annotation.

Tuple

```
( 1, 2.0, "three" )
```

Grouping of unnamed, ordered values, possibly of different types.

ArrayOrList

```
[ e1; ...; en ]
[| e1; ...; en |]
```

Represents arrays and lists of data elements all of the same type.

Record

```
{ driver="Elon"; brand="
   Tesla" }
```

Simple aggregations of named values.

New

```
new Car(...)
```

Constructor to create and initialise class and structure objects.

ObjExpr

```
{ new ... with ... }
```

Object expressions are expressions creating new instances of dynamically created, anonymous object types based on existing base types or sets of interfaces.

While

```
while test-expression do
   expr
```

Iterative execution/looping while the specified test condition is true.

For

```
for i = expr to expr do
   expr
```

For loop.

ForEach

```
for pattern in
   expr_or_range do expr
```

For-each loop.

ArrayOrListOfSeqExpr

```
[ expr ] // list
[| expr |] // array
```

Collection of consecutive data elements of the same type. Lists in F# are immutable whereas arrays are mutable.

CompExpr

```
{ expr }
```

Computation expressions provide a syntax for writing computations that can be sequenced and combined using control flow constructs and bindings.

Lambda

```
fun pat -> expr
```

Lambda expressions define anonymous functions in F#.

MatchLambda

```
function pat1 -> expr
         | ...
         | patN -> exprN
```

Match lambdas provide syntactic sugar for match expressions(see section A.1.1) used on functions.

Match

```
match expr with
   pat1 -> expr
   | ...
   | patN -> exprN
```

Match expressions allow branching control based on comparison of an expression with a set of patterns.

Do

```
do expr
```

Do bindings execute code without the requirement of defining a function or a value. Expressions in a do binding must return unit, see section A.1.3.

Assert

```
assert expr
```

Assert expressions are a debugging feature used to test an expression when in debug mode.

App

```
f x
```

Application of expressions.

TypeApp

```
expr<type1,...,typeN>
```

Generic type application on expressions.

LetOrUse

```
let pat = expr in expr
let f pat1 .. patN = expr
   in expr
let rec f pat1 .. patN =
   expr in expr
use pat = expr in expr
```

Variable and function binding constructs.

TryWith

```
try expr with pat -> expr
```

The try..with expression is used for handling exceptions. The with segment provides pattern matching (see section A.1.1) functionality on thrown exceptions.

TryFinally

```
try expr finally expr
```

Similar to try..with, section A.1.1, but provides a finally block instead of exception matching typically used for clean up of resources allocated in the try block.

Lazy

```
lazy expr
```

Lazy expressions are computations that are evaluated as soon as the result is required (instead of immediate evaluation).

Sequential

```
expr; expr
```

Represents the sequential execution of one expression followed by another.

IfThenElse

```
if expr then expr else expr
if expr then expr
```

Conditional expression to run different branches of code depending on a boolean expression given.

Ident

```
ident
```

Representing identifier patterns.

LongIdent

```
ident.ident...ident
```

Long identifier

LongIdentSet

```
ident.ident...ident <- expr
```

Assignment of values to an identifier.

DotGet

```
expr.ident.ident
```

Value retrieval of a property on an expression.

DotSet

```
expr.ident...ident <- expr
```

Value setting on a property on an expression.

DotIndexedGet

```
expr.[expr,...,expr]
```

Value retrieval on a collection expression.

DotIndexedSet

```
expr.[expr,...,expr] <-
   expr
```

Value setting on an collection expression.

TypeTest

```
expr :? type
```

Checking whether an expression contains a specific type.

Upcast

```
expr :> type
```

Upcast of an object expression.

Downcast

```
expr :?> type
```

Downcast of an object expression.

InferredUpcast

```
upcast expr
```

Upcast of an object expression based on inferring its super class.

InferredDowncast

```
downcast expr
```

Downcast of an object expression to a type that is inferred from program context.

Null

```
null
```

Null value. Normally not used in F# for values or variables but defined for .NET interoperability.

A.1.2 Module declaration nodes

Let

```
module MyModule =
   // top level let-binding
    let topLevelName =
        let nested1 = exp1
        let nested2 = exp2
        finalExpression
```

Top-level let binding, see section A.1.1.

Open

```
open System.IO
```

Opens a namespace.

Hash directive

```
#indent "off"
#load "library.fs"
```

Defines compiler directives.

Nested module

```
module MathStuff =
    let add x y  = x + y
    let subtract x y  = x -
    y
    // nested module
    module FloatLib =
        let add x y :float
   = x + y
        let subtract x y :
   float  = x - y
```

Represents modules inside modules.

Do expression

```
open System
open System.Windows.Forms

let form1 = new Form()

[<STAThread>]
// top level do expression
do
   Application.Run(form1)
```

Top-level *do bindings* are executed when the overlaying module is initialised. The *do*-keyword is required whereas it is optionally in *do expression bindings* A.1.1.

A.1.3 Constant nodes

Relevant Constant CFG rules are listed in the F# specification [1, p.269]. For clarity of presentation the following constants aren't listed as their definitions are identical to other programming languages: *Bool, SByte, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, IntPtr, UIntPtr, Single, Double, Char, Decimal*

Unit

```
()
```

Every F# expression must evaluate to a value. The unit type indicates the absence of a specific value. It acts as a placeholder when no other value exists.

B

Anleitung zu Wissenschaftlichen Arbeiten: Parsing F# in PetitParser

In this chapter we describe how our PetitParser F# parser can be installed, used and what is under the hood. Finally we describe the implementation of the tool for left recursion detection in PetitParser grammars.

B.1 Installation

The Pharo Smalltalk environment[1] virtual machine is required to run the pre-configured PetitParser F# parser Moose[2] image. The following resources are required:

1. Pre-configured PetitParser F# parser Moose image[3]
2. Pharo40.sources file[4]
3. Pharo Virtual Machine
    - For Linux[5]

[1] http://pharo.org/
[2] http://www.moosetechnology.org/
[3] https://github.com/mkubicek/FSharpAST/blob/master/files/moose_suite_5_1_PetitFSharp.image
[4] http://files.pharo.org/get-files/40/sources.zip
[5] http://files.pharo.org/get-files/40/pharo-linux-stable.zip

- For Mac[6]
- ForWindows[7]

B.2 Usage

The Moose PetitParser F# parser image starts up with several open windows whose purpose is to demonstrate the major functionality of PetitParser, our implementation and give a quick start to novices in Pharo Smalltalk and PetitParser.

The windows annotated in Figure B.1 are shown when starting the image:

1. Pharo Playground parsing examples
   (a) Inspecting an AST parsing result
   (b) Output an AST parsing result in encoded text form
   (c) Enabling the PetitParser debugger
2. Class Browser containing the implementations and tests of our PetitParser F# parser and left recursion detection tool

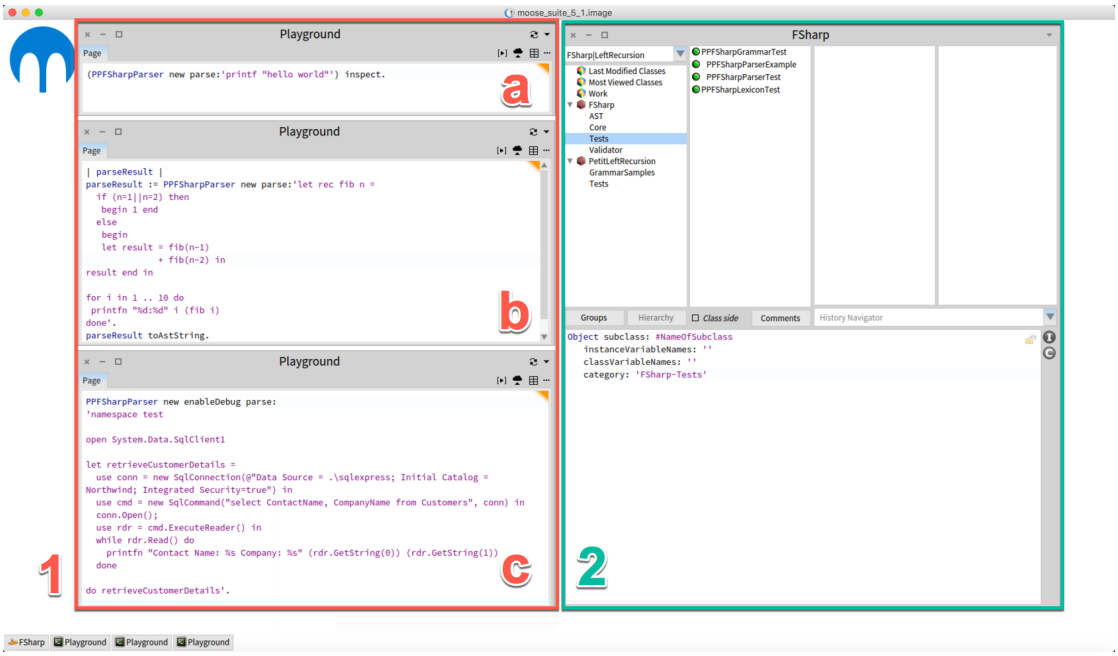

Figure B.1: Starting up the Moose PetitParser F# parser image

[6] http://files.pharo.org/get-files/40/pharo-mac-stable.zip
[7] http://files.pharo.org/get-files/40/pharo-win-stable.zip

The Pharo Playground examples can be run by pressing the button highlighted in Figure B.2.

Figure B.2: Running Pharo Playground code

For a more detailed description of the possibilities Pharo Smalltalk, Moose and PetitParser offer, we suggest the following readings:

- Pharo by example, general introduction to Pharo Smalltalk
  `http://PharoByExample.org/`
- Deep into Pharo, explains advanced Pharo Smalltalk concepts and covers also PetitParser
  `http://www.deepintopharo.com/`
- The Moose Book, explains the Moose platform for software and data analysis. Contains a useful PetitParser chapter
  `http://www.themoosebook.org/`

B.3 Under the hood

It can be beneficial to divide parsers in PetitParser into multiple layers in order to keep the parsing project clear and tidy. In this section we provide some insights into the parser structure we implemented. A dedicated lexical layer ensures that keywords, identifiers, operators, literals and noise tokens (e.g. comments) are well separated and can be accessed while focusing on syntactical grammar rules and semantical parsing. The grammar layerFLs sole purpose is to implement non-terminal grammar rules without having to worry about terminal parsers. The results of the grammar layer then can be transformed into an AST structure by a parser layer.

B.3.1 Lexicon

The responsibility of the lexical parser is to separate terminal symbols from the syntactical parser. Terminal symbols in programming language grammars are keywords, identifiers,

operators and literals. Other unwanted constructs (e.g. comments, compiling statements, attributes,..) can be defined in the lexer and consumed or trimmed alongside with other terminal symbols. Therefore the grammar layer of the parsing project can reference terminal symbols without re-defining them multiple times.

Keywords It is important to ensure that identifiers and keywords are not confused. Therefore it is required to store all keywords in a central place that enables distinction of identifiers and keywords using a predicate parser.

The number of Smalltalk instance variables is limited to 255 due to hosting environment restrictions. PPCompositeParser exposes defined parsers using instance variables. The following explains a practice that ensures keyword terminal parser accessibility and central definition without sacrificing instance variables:

We store all keyword terminal parsers in an instance variable called *keywords* which is a dictionary where the keywordFLs name is the key and the corresponding value is the terminal parser parsing the keyword.

As `PPCompositeParser` instantiates all instance variable as `PPUnresolvedParser` the *keywords* instance variable has to be added to the class side ignore list:

```
PPLanguageLexicon class >> ignoredNames
    ^ #('keywords')
```

To access the keyword terminal parsers we define an accessor method *tokenFor:* which takes a string as an argument and returns a parser from the *keywords* dictionary:

```
PPLanguageLexicon >> tokenFor: aString
    ^ (keywords at: aString)
```

A parsing expression that matches all defined keywords from the *keywords* dictionary can now be defined:

```
PPLanguageLexicon >> keyword
    | keywordParsers |
    keywordParsers := keywords keysSortedSafely
                          collect: [:eachKey | keywords at: eachKey ].
    ^ self asToken: ( (keywordParsers reduce: [ :a :b | a / b ]) )
```

Finally, the keywords dictionary has to be initialised.

```
PPLanguageLexicon >> initialize

    super initialize.
    self initializeKeywords.

PPLanguageLexicon >> initializeKeywords

    | values |
    keywords := Dictionary new.
```

```
10    values := #('abstract' 'assert'
11              'void' 'volatile' 'while').
12
13    values do: [:eachKeyword |
14       keywords at: eachKeyword
15          put: (PPUnresolvedParser named:
16             ('keyword',
17             eachKeyword first asUppercase asString,
18             eachKeyword allButFirst))
19       ].
20
21     keywords keysAndValuesDo:  [:key :value |
22        (keywords at: key) def: (key asParser ,  #word asParser not)]
```

Identifiers Now that all keywords are accessible via `PPLanguageLexicon >> keyword` accessor, it is easy to write a parsing expression that matches just valid identifiers, ignoring keywords.

```
1 PPLanguageLexicon >> identifier
2     ^ ((keyword, #space asParser) / keywords end) not,
3         #letter asParser, #word asParser star
```

Trimming Source code is often annotated by comments which may span over multiple lines. Some computer languages even allow inline comments to be placed in between any language tokens. As handling comments around each language token would violate the *DRY principle*[8] it is recommended to remove them centrally by using the trimming action parser (see section 4.3). For example the surroundings of keywords might be trimmed by adjusting the `TokenFor:` method from section B.3.1.

First a parsing expression has to be defined matching language comments:

```
1 PPLanguageLexicon >> comment
2    ^ '//' asParser, #newline negate star
```

To add more ignorable content in future, we group such constructs in a parsing expression called `ignorable`:

```
1 PPLanguageLexicon >> ignorable
2    ^ comment / #space asParser
```

Now we extend the `tokenFor` method that returns keywords and other terminal symbols to trim the ignorable content around the valid terminal tokens:

```
1 PPLanguageLexicon >> tokenFor: aString
2    ^ (keywords at: aString)    trim: ignorable
```

[8] Acronym for the "Don't repeat yourself" principle which was formulated by Andy Hunt as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

B.3.2 AST

To generate AST nodes from matched input it is necessary to define a data structure to accommodate the AST. It is common practice to define a base class for AST nodes implementing some basic functions. We present a possible implementation that offers easy adding of child nodes with runtime check and a way to represent its values encoded in text.

We first define a base AST node object with a variable called `children` that will later hold the subsidiary AST nodes:

```
Object subclass: #PFSASTNode
   instanceVariableNames: 'children'
   classVariableNames: ''
   category: 'FSharp-AST'
```

The children collection is initialised in the `initialise` method:

```
PFSASTNode >> initialize
   children := OrderedCollection new
```

An accessor method is defined for the children collection:

```
children
   ^ children
```

An method is defined that answers wether the current AST node is a valid FSharpNode:

```
PFSASTNode >> isFSharpNode
^ true
```

Now a method that adds AST nodes to the current node is implemented, which first checks whether the added node really is of the accepted type:

```
PFSASTNode >> addChild: node
   node isNil
   ifTrue: [  ]
   ifFalse:[   self assert: node isFSharpNode.
            self children add: node.]
```

To add several child nodes at once, a new method is created:

```
PFSASTNode >> addChildren: nodes
   nodes do: [ :node | self addChild: node ]
```

In order to access the children nodes of a given node, a dedicated accessor is defined:

```
PFSASTNode >> children
   ^ children
```

Sometimes a node type contains just one child. We provide a dedicated accessor, that ensures the presence of only one child:

```
PFSASTNode >> child
   self assert: children size = 1.
   ^ children first
```

Similarly we define a shortcut accessor for the first and second children:

```
PFSASTNode >> firstChild
   ^ children at: 1
```

```
PFSASTNode >> secondChild
   ^ children at: 2
```

Finally we can define a method, that recursively enumerates all children and collects their textual representation maintaining their nesting levels with the help of parentheses:

```
PFSASTNode >> toAstString
| result |
result := (self class name).
result := result copyFrom:1 to: (self class name size).
children size > 0
   ifTrue: [
      result := result,'('.
      children do: [ :child | result := result,(child toAstString),',' ].
      result := result copyFrom:1 to: (result size -1).
      result := result,')'.
           ].
^  result
```

B.4 Validation

Since even minor changes may break a parser, test-driven development facilitates fast discovery of bugs. The `PetitCompositeTest` class allows testing of each parsing expression separately.

As a PetitParser grammar instance answers a message containing the name of a defined parsing expression with the corresponding parsing function, testing a PetitParser grammar is not any different from testing Smalltalk code.

Testing a PetitParser grammar is done by subclassing `PPCompositeParserTest`:

```
PPCompositeParserTest subclass: #PPFSharpGrammarTest
   instanceVariableNames: ''
   classVariableNames: ''
   category: 'FSharp-Tests'
```

Then, the test case class has to be linked to the PetitParser grammar we want to test, for example `PPFSharpGrammar`:

```
parserClass
    ^ PPFSharpGrammar
```

To test a PetitParser grammar parsing expression, a new method simply has to be implemented which defines the test input and selects the corresponding grammar rule, `PPFSharpGrammar >> exprLet`:

```
testExprLet
self parse:
'let x = 123 in
y' rule:#exprLet.
```

By using the `test`-prefix in test method names, corresponding parsing expression grammar methods not only show the test status but also allow direct test execution. This enables the developer to validate changes quickly.

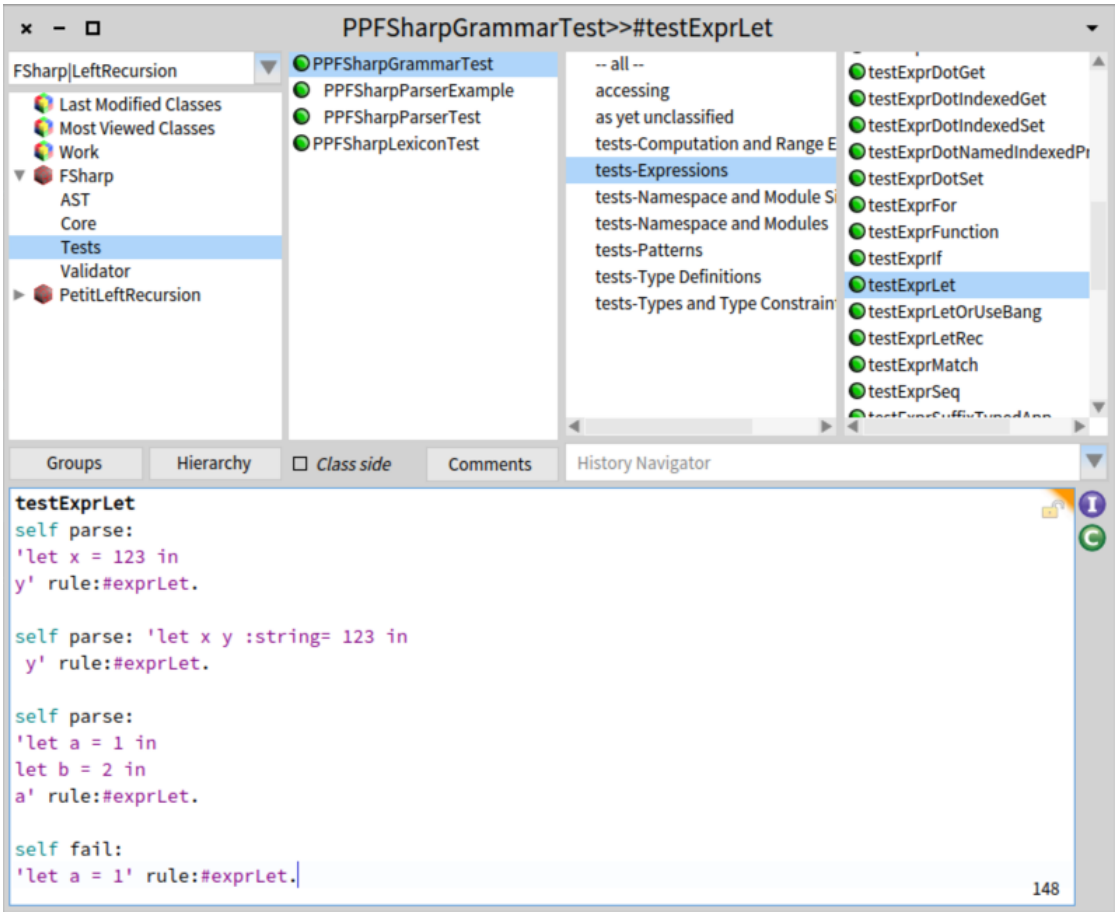

Figure B.3: Testing the ExprLet parsing expression of the PPFSharpGrammar

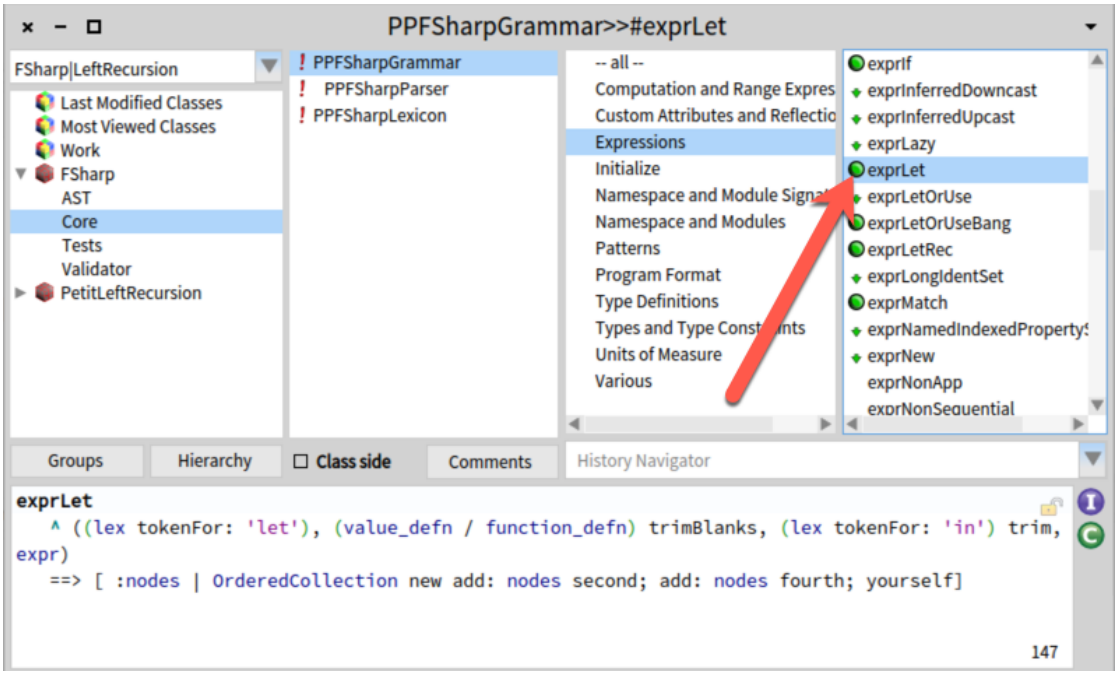

Figure B.4: Representation of the test result from the test defined in Figure B.3 during browsing of the corresponding parsing expression

B.5 F# AST extractor

In this section we explain the F# AST extractor we used to explore the F# AST structure and validate our parser implementation. Our tool is built using the F# Compiler Services Untyped AST processor explained in subsection B.6.2.

B.5.1 Installation

The implementation is available from the following GitHub repository: `https://github.com/mkubicek/FSharpAST`

There are three ways to get the extraction tool up and running:

- Run the pre-compiled binary located in the bin folder in a Microsoft Windows .NET environment or on unix using Mono[9].
- Open and compile the project file in a Xamarin IDE[10]. The necessary F# Compiler Service library will be added into the project automatically from a NuGet package.
- To compile the F# AST extractor project from scratch, the following dependencies have to be installed prior:
  1. F# Compiler, Core Library & Tools[11]
  2. F# Compiler Service[12]

B.5.2 Usage

The AST extraction tool supports the two AST extraction modes: simple extraction and extended let binding extraction. Whereas simple extraction generates the whole textual encoded AST output from any parseable F# input, the extended let binding extraction generates the ASTs of the largest let binding fragments present in the input code.

B.5.2.1 Simple AST extraction

Simple AST extractionFLs purpose is to parse F# input using the official F# compiler and output a textual representation of the generated AST. Simple AST extraction can be performed by passing the `simple` parameter to the extraction tool:

```
FSharpASTExtractor.exe simple <inputFile> <outputFile>
```

[9] `http://www.mono-project.com/docs/getting-started/install/`
[10] `https://www.xamarin.com/platform`
[11] `https://github.com/fsharp/fsharp`
[12] `https://github.com/fsharp/FSharp.Compiler.Service`

For example, `FSharpASTExtractor.exe simple input.fs out.txt` will write the text encoded AST shown in Listing 15 to a file named `out.txt` from the input file containing the F# code listed in Listing 16.

```
PFSSynModuleOrNamespace(PFSLet(PFSIfThenElse(PFSParen(
   PFSConst),PFSParen(PFSLetOrUse(PFSApp(PFSApp(PFSIdent(
   operator),PFSApp(PFSIdent(fib),PFSParen(PFSApp(PFSApp(
   PFSIdent(operator),PFSIdent(n)),PFSConst)))),PFSApp(
   PFSIdent(fib),PFSParen(PFSApp(PFSApp(PFSIdent(operator)
   ,PFSIdent(n)),PFSConst))));PFSIdent(result))))),
   PFSDoExpr(PFSForEach(PFSApp(PFSApp(PFSApp(PFSIdent(
   printfn),PFSConst),PFSIdent(i)),PFSParen(PFSApp(
   PFSIdent(fib),PFSIdent(i)))))))
```

Listing 15: Text encoded AST generated by the simple mode of the AST extraction tool

```
let rec fib n =
  if (n=1||n=2) then
   1
  else
   let result = fib(n-1)
                + fib(n-2)
   result

for i in 1 .. 10 do
 printfn "%d:%d" i (fib i)

```

Listing 16: Content of input.fsharp file, a F# program to print a Fibonacci sequence

B.5.2.2 Extended let binding AST extraction

Extended let binding extraction is useful when the AST of deeply nested F# code is studied as it splits the the output based on the largest let bindings it contains. Complex F# code input can be split into manageable pieces.

```
File: #(./fsharp3sample-25592/SampleProviders/Samples.
   Hadoop/Helpers.fs)#
Code: #(let theProxyProcessAgent = startProxyProcessAgent
   ())#
AST: #(PFSLet(PFSApp(PFSIdent(startProxyProcessAgent),
   PFSConst)))#
ModuleLevel: #(True)#
```

Listing 17: Text encoded AST and corresponding code extracted by the extended let binding" mode of the AST extraction tool

Each extracted code fragment contains the information shown in Listing 17 about its origin (path to F# source file), the code fragment itself, textual representation of the AST it symbolises and additional information whether the code fragment is a module level statement or a nested statement.

Extended let binding AST extraction can be performed by executing the extraction tool with the three parameters
`extendedLet <inputPath> <outputFile>`.
In case the inputPath resolves to a directory, our tool will recursively extract F# code fragments from all available .fs or .fsx files.

B.6 F# ecosystem

F# is being developed by the F# Software Foundation[13], Microsoft[14] and open source contributors. The F# project comes with the Apache Licence 2.0 [15] allowing the use of the project for a wide field of application. Most new contributions to the F# language first go to MicrosoftFLs Visual F# tools repository[16]. This repository acts as the main repository to ensure that MicrosoftFLs main package of F# for Windows includes any contributions that are made and makes sure that the versions do not diverge. Contributions are then merged into the open edition of the F# compiler[17], maintained by the F# Software Foundation. It provides an open source, cross-platform compiler for F#.

Multiple development tools exist for the F# programming language. Compiler, compiler tools, core libraries and integrated development environments (IDEs) are available for multiple platforms. Integrated development environments typically consist of a source code editor, build automation and a debugger.

[13] http://www.fsharp.org
[14] http://www.microsoft.com/
[15] http://www.apache.org/licenses/LICENSE-2.0
[16] https://github.com/Microsoft/visualfsharp
[17] https://github.com/fsharp/fsharp/

The Visual F# tools from Microsoft are fully integrated into the Microsoft Visual Studio[18], Microsofts own IDE for a growing number of programming languages and multi-platform app development.

B.6.1 F# Compiler Services

The F# Compiler Services namespace of the F# core library contains some internal functions used by the F# compiler. It exposes functionality for implementing F# language bindings, additional tools based on the compiler and refactoring tools. The compiler services package[19] contains the following public API services:

Language tokenizer. Turns any F# source code into a stream of tokens. Useful for implementing source code colorization and basic tools. Correctly handle nested comments, strings etc.

Untyped AST processor. Allows accessing the untyped abstract syntax tree (AST). This represents parsed F# syntax without type information and can be used to implement code formatting and various simple processing tasks.

Editor (IDE) services. Exposes functionality for auto-completion, tool-tips, parameter information etc. These functions are useful for implementing F# support for editors and for getting some type information for F# code.

Signatures, types, and resolved symbols services. Many services related to type checking return resolved symbols, representing inferred types, and the signatures of whole assemblies.

Resolved expression services. Services related to working with type-checked expressions and declarations, where names have been resolved to symbols.

Project and project-wide analysis services. The developer can request a check of an entire project, and ask for the results of whole-project analyses such as find-all-references.

F# interactive host environment. Allows calling F# interactive as a .NET library from .NET code. The developer can use this API to embed F# as a scripting language in his projects.

Hosting the F# compiler. Allows the developer to embed calls to the F# compiler.

[18] `https://www.visualstudio.com`
[19] `http://fsharp.github.io/FSharp.Compiler.Service/`

File system API. The FSharp.Compiler.Service component has a global variable representing the file system. By setting this variable the developer can host the compiler in situations where a file system is not available.

B.6.2 F# untyped AST processor

The F# Compiler Services (see subsection B.6.1) offers the possibility to process the untyped AST of arbitrary source code. It can be accessed by an instance of the `FSharpChecker` found in the `FSharp.Compiler.Service` namespace. This type opens a context for type checking and parsing of stand-alone F# script files or project files with multiple source files. The `FSharpChecker` instance can perform an untyped parse. Type-checking is performed at a later stage of the parsing process. Listing 18 shows how the untyped AST can be obtained from some input files. The F# AST is defined in the Microsoft.FSharp.Compiler.Ast namespace and can be traversed using the code in Listing 19.[20]

```
#r "FSharp.Compiler.Service.dll"
open System
open Microsoft.FSharp.Compiler.SourceCodeServices

// Create an interactive checker instance
let checker = FSharpChecker.Create()

// Get untyped tree for a specified input
let getUntypedTree (file, input) =
  // Get compiler options for the 'project' implied by a
   single script file
  let projOptions =
      checker.GetProjectOptionsFromScript(file, input)
      |> Async.RunSynchronously

  // Run the first phase (untyped parsing) of the compiler
  let parseFileResults =
      checker.ParseFileInProject(file, input, projOptions)
      |> Async.RunSynchronously

  match parseFileResults.ParseTree with
  | Some tree -> tree
  | None -> failwith "Something went wrong during parsing!
   "
```

[20] https://fsharp.github.io/FSharp.Compiler.Service/untypedtree.html

Listing 18: F# code to generate an AST using F# compiler services

```
#r "FSharp.Compiler.Service.dll"
open Microsoft.FSharp.Compiler.Ast

let rec traverseTree tree =
     match tree with
     | ParsedInput.ImplFile(implFile) ->
        let (ParsedImplFileInput(_, _, _, _, _, modules,
             _)) = implFile
        traverseModules modules
     | _ -> ()

and traverseModules modules =
  for mod in modules do
    let (SynModuleOrNamespace(_, _, decls, _, _, _, _)) =
   mod
    traverseDeclarations decls

and traverseDeclarations decls =
  for declaration in decls do
    match declaration with
    | SynModuleDecl.Let(isRec, bindings, range) -> ()
    | SynModuleDecl.<anyOtherNode>(..) -> DoSomething
    | _ -> ()

and traverseSynExpr synExpr =
    | SynExpr.Paren(synExpr1,i2,i3,i4) -> ()
   | SynExpr.<anyOtherNode>(..) -> DoSomething
    | _ -> ()
```

Listing 19: F# code to traverse an AST obtained by F# compiler services

B.7 Left recursion detection in PetitParser

In this section we present the concrete implementation of the left recursion algorithms we describe in subsection 6.4.2.

B.7.1 Static PetitParser left recursion detection implementation

PPParser

The superclass PPParser defines default behaviour:

```
PPParser >> hasLeftRecursion
   ^ false
```

```
PPParser >> hasLeftRecursionOpen: set
   ^ false
```

```
PPParser >> hasLeftRecursionOpen: collection closed: set
   ^ false
```

```
PPParser >> isPotentiallyNonConsuming
   ^ false
```

PPDelegateParser

```
PPDelegateParser >> hasLeftRecursion
   ^ parser hasLeftRecursionOpen: (OrderedCollection new)
            closed: IdentitySet new.
```

```
PPDelegateParser >> hasLeftRecursionOpen: set
   ^ parser hasLeftRecursionOpen: (OrderedCollection new)
            closed: set.
```

```
PPDelegateParser >> hasLeftRecursionOpen: openCollection
                        closed: closedSet
   (openCollection includes: self ) ifTrue: [
      openCollection add: self.
      ^ true
   ].

   (closedSet includes: self) ifTrue: [ ^ false].

   openCollection add: self.
   closedSet add: self.

   ^ (parser hasLeftRecursionOpen:
      (openCollection copy) closed: closedSet)
```

PPAndParser

```
PPAndParser >> isPotentiallyNonConsuming
^ true
```

PPNotParser

```
PPNotParser >> isPotentiallyNonConsuming
^ true
```

PPOptionalParser

```
PPOptionalParser >> isPotentiallyNonConsuming
^ true
```

PPRepeatingParser

```
PPRepeatingParser >> isPotentiallyNonConsuming
^ min = 0
```

PPListParser

```
PPListParser >> hasLeftRecursion
   ^ self hasLeftRecursionOpen: OrderedCollection new
           closed: IdentitySet new
```

```
PPListParser >> hasLeftRecursionClosed: set
   ^ self hasLeftRecursionOpen: OrderedCollection new
           closed: set.
```

PPChoiceParser

```
PPChoiceParser >> hasLeftRecursionOpen: openCollection
                          closed: closedSet
   (openCollection includes: self) ifTrue: [
      openCollection add: self.
      ^ openCollection
   ].

   (closedSet includes:self) ifTrue:[ ^ false].

   openCollection add: self.
   closedSet add: self.

   self children do: [:child |
      | leftRecursion |
      leftRecursion := (child hasLeftRecursionOpen:
         (openCollection copy) closed: closedSet).
      (leftRecursion) ifTrue: [ ^ true ] ].

   ^ false
```

PPSequenceParser

```
1  PPSequenceParser >> hhasLeftRecursionOpen: openCollection
2                                   closed: closedSet
3  | consumingParserEncountered   |
4
5     (openCollection includes: self ) ifTrue: [
6        openCollection add: self.
7        ^ true
8     ].
9
10    (closedSet includes: self) ifTrue: [ ^ false].
11
12    openCollection add: self.
13    closedSet add: self.
14
15    consumingParserEncountered:= false.
16    self children do:
17       [ :child |
18       | hasLeftRec |
19        (consumingParserEncountered not)
20             ifTrue: [
21                   hasLeftRec := child hasLeftRecursionOpen:
22                                 (openCollection copy)
23                                    closed: closedSet.
24                    (child isPotentiallyNonConsuming)
25                       ifFalse: [consumingParserEncountered := true].
26                ]
27
28             ifFalse: [
29                   hasLeftRec := child hasLeftRecursionOpen:
30                                 (OrderedCollection new)
31                                  closed: closedSet.
32                ].
33
34          (hasLeftRec) ifTrue: [ ^ true ].
35     ].
36
37 ^ false
```

Line 5-13: Keep track if the current parser occurred in the current traversal or if it was already checked for left recursion

Line 16-36: Until the first true consuming parser appears, check the parser for left recursion with a copy of the current open set. After the first consuming parser is encountered, check with an empty open set as the parsers subgraph still could contain left recursion.

B.7.2 Dynamic PetitParser left recursion detection implementation

PPParser

Note that only lines 14 to 17 and line 21 were added to detect left recursion while performing PPParserFLs debug parsing.

```
1  PPParser >> enableDebug
2     | root newParser |
3     root := PPParserDebuggerResult new.
4     newParser := self transform: [:each |
5        each >=> [:context :continuation |
6           | result child pair |
7           child := PPParserDebuggerResult new
8                 parser: each;
9                 parent: root.
10          root := root children add: child.
11          child start: context position + 1.
12          child showChildren: each debuggerOutput.
13
14          pair := (context position -> each).
15          (context openSet includes: pair)
16                  ifTrue: [ self error: 'left recursion detected' ].
17          context openSet add: pair.
18
19          result := continuation value.
20
21          context openSet remove: pair.
22
23          child end: context position.
24          root result: result.
25          root := root parent.
26          result
27       ]
28    ].
29
30 ^ PPDebugParser on: newParser root: root.
```

Line 15: If context and position pair are known to the current context throw an error as the parsing encountered a left recursion.

List of Figures

List of Tables

Bibliography

[1] F# 3.0 language specification. Technical report, September 2012. http://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf.

[2] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal of Computing*, 1:305–312, 1972.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.

[4] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959.

[5] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *IEEE Conference Record of 11th Annual Symposium on Switching and Automata Theory, 1970*, pages 153–174, October 1970.

[6] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. http://www.chomsky.info/articles/195609--.pdf.

[7] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM.

[8] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[9] Attieh Sadeghi Givi. Layout sensitive parsing in the PetitParser framework. Bachelor's thesis, University of Bern, October 2013.

[10] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[11] S.C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[12] Jan Kurš, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[13] Jan Kurš, Mircea Lungu, and Oscar Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[14] J. Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, 2009.

[15] Robin Milner, M. Tofte, and R. Harper. *The definition of standard ML.* MIT Press, Cambridge, 1990.

[16] Robert C. Moore. Removing left recursion from context-free grammars. *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, 2001.

[17] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.

[18] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

[19] Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, September 2010.

[20] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[21] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308 – 315, 1975.

[22] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.