Multi-Purpose Syntax Definition with **SDF3**

Eelco Visser

TU Delft

SEFM | ‘CWI, Amsterdam’ | September 16, 2020
UvA/CWI 1995 \[\Rightarrow\] TU Delft 2020
A Family of Syntax Definition Formalisms

Eelco Visser
Programming Research Group, University of Amsterdam,
Kruislaan 314, NL-1098 SJ Amsterdam, The Netherlands
email: visser@irv.uva.nl, http://sdf.nwi.uva.nl/~visser/

Abstract. In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, various features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adapting and extending old ones. The formalism is developed with the algebraic specification formalism Aig+SGF. It provides the following features: lexical and context-free syntax, variables, disambiguation by priorities, regular expressions, character classes and module definitions. New are the uniform treatment of lexical syntax, context-free syntax and variables, the treatment of regular expressions by normalization yielding abstract syntax without ambiguity sets, regular expressions as result of productions and modules with hidden imports and renamings.

Key Words & Phrases: syntax definition formalism, language design, context-
free grammar, context-free syntax, lexical syntax, priorities, regular expres-
sions, formal language, parsing, abstract syntax, module, renaming, hidden
imports.

Note: Supported by the Dutch Organization for Scientific Research (NWO) un-
direct grant 612-37-420 incremental parser generation and context-dependent
disambiguation, a multidisciplinary perspective.

1 Introduction

1.1 General

New programming, specification and special purpose languages are being developed continuously [87,94]. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embodied in other languages: regular expressions in editor operations, macro definitions for macro preprocessors, user definable infix or postfix operators in programming languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free gram-
mars, which are widely used in computer science since their introduction by Chomsky in 1956 [Ch56]. A context-free grammar is a set of string rewrite rules of the form $A \rightarrow \alpha$. A string $w$ is member of the language described by a grammar $G$ if it can be rewritten to the start symbol $S$, i.e. if there is a sequence $w = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n = S$ and each step has the form $\alpha_i \rightarrow \alpha_{i+1}$ where $\alpha_i \rightarrow \alpha_{i+1}$ is a production in $G$.

Despite, or maybe due to, the simplicity of this basic structure there has never emerged a standard formalism for syntax definition. The Backus-Naur Form (BNF) [Ba59, N 60], originally developed for the definition of the syntax of Algol, is a commonly used notation for context-free grammars, but it does not have the status of a standard. Several standard notations for syntax definition have been proposed [Wi77, Wi82]. None of these has been convincing, instead a number of similar or overlapping formalisms exist.


Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza and Eelco Visser

1. Australian National University, Australia
2. Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permisive syntax, layout constraints, formatting templates, placeholder syntax, and modularity. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, seamless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · programming language · parsing.

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a grammar formalism in the tradition of Chomsky’s context-free grammars [4] and the Backus-Naur Form [3]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Scoofix [38] language workbenches. The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [50,44]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through

A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen
Pieter Nienhuis
Evelin Mieszko
Andrew Tolmach
Guido Wachsmuth

EbelToh, University of Technology, The Netherlands
UBC, Vancouver, Canada
University of Twente, The Netherlands

Abstract

In previous work, we introduced scope graphs as a formalism for the static semantic analysis and checking of programs with implicit variables. Scope graphs are a generalization of the virtual machine stack used in many languages, including Scheme, C, and Java, that provide a way to represent the static semantics of programs with variables that are declared dynamically. In this paper, we present the formal definition of scope graphs and their properties, and demonstrate how to use them to detect a variety of common programming errors. We also discuss the implementation of a scope graph-based static analyzer for Scheme, and compare it to a traditional static analyzer. Finally, we evaluate the性能 of the scope graph-based analyzer and show that it is competitive with the traditional analyzer.

Keywords: Language Semantics; Static Analysis; Programming Languages; Type Systems; Program Analysis; Software Engineering; Computer Science.

Introduction

Programs with implicit variables present a number of challenges for static analysis, particularly when it comes to determining the scope of variables and ensuring that they are properly bound. In this paper, we present a new approach to static semantic analysis based on scope graphs, which provides a way to represent the static semantics of programs with dynamically declared variables. We show how to use scope graphs to detect a variety of common programming errors, and demonstrate the performance of a scope graph-based static analyzer for Scheme.

Hendrik van Antwerpen (hvanantwerp@utwente.nl)
Pieter Nienhuis (pieter.nienhuis@ubc.ca)
Evelin Mieszko (e.mieszko@utwente.nl)
Andrew Tolmach (Andrew.Tolmach@ubc.ca)
Guido Wachsmuth (Guido.Wachsmuth@twente.nl)

A Family of Syntax Definition Formalisms

Erko Visser
Programming Research Group, University of Amsterdam, National Institute for Research in Information Technology, The Netherlands
email: visser@elc.net, http://elc.net/visser/

Abstract. In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, several features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adding and removing old ones. The formalism is developed with the algebraic specification formalism Antioch. It provides the following features: labelled and unlabelled syntax, signatures, variants, regular expressions, fixed-point and least fixed-point definitions, and contexts for definitions. We use the method treatment of formal syntax, control-free syntax and modularity, the treatment of flexible expressions by automated padding of abstract syntax without auxiliary vars, regular expressions as result of pattern matching with functional backtracking, and module composition. These features support the multi-purpose interpretation of syntax definitions, including definition of type schemes for abstract syntax tree representations, seamless generalised parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesisation and formatting, and syntactic completion. This paper gives a high level overview of SDF by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition - programming language - parsing

1 Introduction

1.1 New programming, specification and spatial language languages are being developed continuously [179]. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embedded in other languages: regular expressions in edit operations, macro definitions for macro preprocessors, user definable rules or diffs operations in programing languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free grammars, which are useful tools for computer science since their introduction in Chomsky in 1956 [159]. A context-free grammar is a set of rewriting rules of the form $A \rightarrow \alpha$. A string $w$ is a number of the language described by a grammar if it can be rewritten to the start symbol $S_i$, i.e., if there is a sequence $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n = w$ and each step has the form $s_i \rightarrow \alpha_i$. This is a common use notation for context-free grammars, but it does not have the status of a standard. Several standard notations for syntax definition have been proposed [61, 178, 182]. None of these is more concise, instead a number of similar or overlapping formalisms exist.


Multi-Purpose Syntax Definition with SDF3

Luis Eduardo Amorim de Souza1 and Erko Visser2
1 Australian National University, Australia
2 Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permits syntax, layout constraints, formatting templates, placeholders, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including definition of type schemes for abstract syntax tree representations, seamless generalised parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesisation and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition - programming language - parsing

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a grammar formulation in the tradition of Chomsky's context-free grammars [1] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax-definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASPf-SDF [3], Syntax/XT [9], and Sphinx [38] language workspaces. The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalised LR (GLR) parsing algorithm [16,44]. Its programming environment, as part of the ASPf-SDF Metadata Interface [46], focused on live development of syntax definitions through

History of SDF
The Syntax Definition Formalism SDF [1989]
- Heering, Hendriks, Klint, Rekers

Lexical Syntax + Context-free Syntax
- Separate scanner, parser
- Syntax definition $\simeq$ algebraic signature

Generalized LR Parsing
- Support full class of context-free grammars
- Lazy, incremental, modular scanner, parser generation

Modular Syntax Definition

ASF+SDF MetaEnvironment
Scannerless Generalized LR (SGLR) Parsing [1997]
- Support character-level grammars
- Lexical disambiguation (follow restrictions, reject productions)

Disambiguation Filters for Associativity and Priority
- Shallow conflicts: Unsafe for prefix/postfix operators with low priority

A Family of Syntax Definition Formalism [1995]
- Transform high-level language to Kernel SDF

Language Composition
- Meta-programming with concrete object syntax [2002]
- Concrete object syntax [2004]

Spoofax Language Workbench [2010]
Multi-Purpose Syntax Definition
- Many tools from single source

Templates
- Formatting instructions from syntax definition

Semantics of Associativity and Priority
- Safe and Complete Disambiguation, Deep conflicts
- Parenthesis insertion

Layout-Sensitive Syntax
- layout constraints, layout declarations

Spoofax 2
Impact

Education
- Compiler Construction
- Language Engineering Project

Research
- Syntax definition in Spoofax Language Workbench
- Meta-Language Design: NaBL, Statix, Stratego, FlowSpec, ...
- DSLs: WebDSL, IceDust, PIE

Industry
- Oracle Labs: Graph Analytics
- Canon: Oil, CSX
- Philips/MasCot: Software Restructuring
<table>
<thead>
<tr>
<th>Category</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Recovery</td>
<td>Kats, De Jonge</td>
</tr>
<tr>
<td>Templates</td>
<td>Vollebregt, Kats</td>
</tr>
<tr>
<td>Layout Constraints</td>
<td>Erdweg</td>
</tr>
<tr>
<td>Layout Declarations</td>
<td>Eduardo Amorim</td>
</tr>
<tr>
<td>Disambiguation</td>
<td>Eduardo Amorim</td>
</tr>
<tr>
<td>Syntactic Completion</td>
<td>Eduardo Amorim</td>
</tr>
<tr>
<td>JSGLR2 (in progress)</td>
<td>Denkers, Sijm</td>
</tr>
<tr>
<td>SDF3 Implementation</td>
<td>Eduardo Amorim</td>
</tr>
</tbody>
</table>
This Talk

Phrase Structure
- constructors

Formatting Templates
- syntactic completion

Declarative Disambiguation
- from unsafe to safe disambiguation

Layout Constraints/Declarations
- for layout-sensitive syntax

Take away: Multi-Purpose Interpretation
- See paper for more
Phrase Structure
What is Syntax?

\((\text{fun } x \rightarrow x + 3) \ y\)
Syntax = Structure of Programs

context-free syntax
Exp = "(" Exp ")"
Exp.Int = INT
Exp.Var = ID
Exp.Add = Exp "+" Exp
Exp.Fun = "fun" ID* "→" Exp
Exp.App = Exp Exp

(fun x → x + 3) y

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010
Constructors ➞ Abstract Syntax Tree

context-free syntax
Exp = "(" Exp ")" {bracket}

Exp.Int = INT

Exp.Var = ID

Exp.Add = Exp "+" Exp

Exp.Fun = "fun" ID* "→" Exp

Exp.App = Exp Exp

(fun x → x + 3) y

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010
context-free syntax
Exp = "(" Exp ")" {bracket}

Exp.Int = INT

Exp.Var = ID

Exp.Add = Exp "+" Exp

Exp.Fun = "fun" ID* "\rightarrow" Exp

Exp.App = Exp Exp

(fn x → x + 3) y

App(
    Fun(["x"], Add(Var("x"), Int("3"))),
    Var("y")
)
Syntax Definition \approx Algebraic Signature

**context-free syntax**

\[
\begin{align*}
\text{Exp} & = "(" \text{Exp} ")" \ \{\text{bracket}\} \\
\text{Exp.Int} & = \text{INT} \\
\text{Exp.Var} & = \text{ID} \\
\text{Exp.Add} & = \text{Exp} \ "+" \ \text{Exp} \\
\text{Exp.Fun} & = "\text{fun}\" \ \text{ID}\* \ "\rightarrow" \ \text{Exp} \\
\text{Exp.App} & = \text{Exp} \ \text{Exp}
\end{align*}
\]

**signature**

sorts INT ID Exp

constructors

\[
\begin{align*}
\text{Int} : \text{INT \rightarrow Exp} \\
\text{Var} : \text{ID \rightarrow Exp} \\
\text{Add} : \text{Exp \* Exp \rightarrow Exp} \\
\text{Fun} : \text{List(ID) \* Exp \rightarrow Exp} \\
\text{App} : \text{Exp \* Exp \rightarrow Exp}
\end{align*}
\]

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010
Parsing Declaratively

\[ \text{parse}(\text{yield}(t)) = t \]

\[ \text{yield} : \text{ParseTree} \rightarrow \text{String} \]
\[ \text{parse} : \text{String} \rightarrow \text{ParseTree} \]
Language Designers focus on Structure of Programs
Formatting Templates
context-free syntax
Exp = <(Exp)> {bracket}
Exp.Int = INT
Exp.Var = ID
Exp.Add = Exp + Exp
Exp.Fun = [fun [ID*] -> [Exp]]
Exp.App = Exp < Exp
Exp.Let = let {Bnd \n\n}* in Exp
Bnd.Bnd = <ID> = Exp

let inc = fun x -> x + 1
in inc 3

let
inc = fun x -> x + 1
in
inc 3

let inc = fun x -> x + 1
in
inc 3

let inc = fun x -> x + 1
in
inc 3

let inc = fun x -> x + 1
in
inc 3

let inc = fun x -> x + 1
in
inc 3

let inc = fun x -> x + 1
in
inc 3
Parsing + Formatting Declaratively

\[
\text{implode}(\text{parse}(\text{format}(t))) = t
\]

\[
\begin{align*}
\text{format} & : \text{AST} \rightarrow \text{String} \\
\text{implode} & : \text{ParseTree} \rightarrow \text{AST} \\
\text{parse} & : \text{String} \rightarrow \text{ParseTree}
\end{align*}
\]
Syntactic Completion = Rewriting Incomplete Programs

Explicit incompleteness: extend language with placeholders
Completion: rewrite placeholders
Templates: Formatting proposals
Soundness: Only syntactically correct proposals
Completeness: Reach all programs

De Souza Amorim, Erdweg, Wachsmuth Visser. Principled syntactic code completion using placeholders. SLE 2016
How does structure map to text?
Declarative Disambiguation
Ambiguous Grammar

context-free syntax
Exp.Int = INT
Exp.Var = ID
Exp.Min = <<Exp> - <Exp>>
Exp.Add = <<Exp> + <Exp>>
Exp.Mul = <<Exp> * <Exp>>
context-free syntax

Exp.Int = INT
Exp.Var = ID
Exp.Min = Exp - Exp
Exp.Add = Exp + Exp
Exp.Mul = Exp * Exp
Ambiguous Sentence has Multiple Parse Trees

context-free syntax

Exp.Int = INT
Exp.Var = ID
Exp.Min = \langle Exp \rangle - \langle Exp \rangle
Exp.Add = \langle Exp \rangle + \langle Exp \rangle
Exp.Mul = \langle Exp \rangle * \langle Exp \rangle
Disambiguation with Associativity and Priority Rules

**context-free syntax**
- \( \text{Exp.Int} = \text{INT} \)
- \( \text{Exp.Var} = \text{ID} \)
- \( \text{Exp.Min} = \text{<Exp> - <Exp>} \)
- \( \text{Exp.Add} = \text{<Exp> + <Exp>} \)
- \( \text{Exp.Mul} = \text{<Exp> * <Exp>} \)

**context-free syntax**
- \( \text{Exp.Min} = \text{<Exp> - <Exp>} \{ \text{left} \} \)
- \( \text{Exp.Add} = \text{<Exp> + <Exp>} \{ \text{left} \} \)
- \( \text{Exp.Mul} = \text{<Exp> * <Exp>} \{ \text{left} \} \)

**context-free priorities**
- \( \text{Exp.Mul} > \{ \text{left: Exp.Min Exp.Add} \} \)
Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

**Rules**

\[
\frac{A.C_1 > A.C_2}{C_1}
\]

\[
C_1
\]

\[
\alpha \quad C_2 \quad \gamma
\]

\[
\beta
\]

\[
A.C_1 \text{ left } A.C_2
\]

\[
C_1
\]

\[
\alpha \quad C_2
\]

\[
\beta
\]

\[
A.C_1 \text{ right } A.C_2
\]

\[
C_1
\]

\[
C_2 \gamma
\]

\[
\beta
\]

**Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)**

**Instances**

\[
E.Mul > E.Add
\]

\[
\frac{E.Mul > E.Add}{Mul}
\]

\[
mul
\]

\[
\frac{Add \ast E}{E + E}
\]

\[
E + E
\]

\[
E.Mul > E.Add
\]

\[
\frac{E.Mul > E.Add}{Mul}
\]

\[
mul
\]

\[
\frac{E*Add}{E + E}
\]

\[
E + E
\]

\[
E.Add \text{ left } E.Add
\]

\[
\frac{E.Add left E.Add}{Add}
\]

\[
add
\]

\[
\frac{E + Add}{E + E}
\]

\[
E + E
\]
context-free syntax

Exp.Min = <<Exp> - <Exp>> {left}
Exp.Add = <<Exp> + <Exp>> {left}
Exp.Mul = <<Exp> * <Exp>> {left}

classic-free priorities

Exp.Mul > {left: Exp.Min Exp.Add}
Safe for High Priority Prefix Operators

Conflict Patterns

Trees
Unsafe for Low Priority Prefix Operators [SDF2]

Conflict Patterns

Trees

Afroozeh, van den Brand, Johnstone, Scott, Vinju: Safe specification of operator precedence rules. SLE 2013
Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

- **Right Recursive in Left Recursive Position**
  - $A.C_1 > A.C_2$
  - $C_1$
  - $C_2 \beta$
  - $\alpha A$

- **Left Recursive in Right Recursive Position**
  - $A.C_1 > A.C_2$
  - $C_1$
  - $\alpha C_2$

- **Associativity**
  - $A.C_1 left A.C_2$
  - $C_1$
  - $A \alpha C_2$
  - $A \beta A$

Conflict Patterns

- **conflict pattern: right recursive**
  - $E.Add > E.Lam$
    - $E.Add$
    - $Add$
    - $Lam + E$
    - $\lambda ID . E$

- **Amorim, Visser: A direct semantics of declarative disambiguation rules. (Under revision)**
  - $E.Add > E.Lam$
    - $Add$
    - $E + Lam$
    - $\lambda ID . E$

- **not a conflict pattern: not left recursive**
  - $E.Add > E.Lam$
    - $E + Lam$
    - $\lambda ID . E$
Shallow Interpretation: Safe for Low Priority Prefix Operators

**Conflict Patterns**

\[
\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}
\]

\[
\text{Lam} + E
\]

\[
\lambda \text{ID} . \ E
\]

**Trees**

\[
\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}
\]

\[
E + \text{Lam}
\]

\[
\lambda \text{ID} . \ E
\]

\[
\begin{array}{c}
\text{Add} \\
\text{Add}
\end{array}
\]

\[
\begin{array}{c}
a + \text{Add} \\
a + \text{Lam}
\end{array}
\]

\[
\begin{array}{c}
\text{Lam} + c \\
\lambda x . \text{Add}
\end{array}
\]

\[
\begin{array}{c}
\lambda x . \text{Add} \\
b + c
\end{array}
\]

\[
\begin{array}{c}
\text{Add} \\
\text{Add}
\end{array}
\]

\[
\begin{array}{c}
a + \text{Lam} \\
\lambda x . \text{Add}
\end{array}
\]

\[
\begin{array}{c}
\text{Add} + c \\
a + \text{Lam}
\end{array}
\]

\[
\lambda x . b
\]
Shallow Interpretation: Incomplete for Low Priority Prefix Operators

**Conflict Patterns**

\[
E.\text{Add} > E.\text{Lam}
\]

\[
\text{Add}
\]

\[
\text{Lam} + E
\]

\[
\lambda \text{ID} . E
\]

\[
E.\text{Add} > E.\text{Lam}
\]

\[
\text{Add}
\]

\[
E + \text{Lam}
\]

\[
\lambda \text{ID} . E
\]

\[
E.\text{Pow} > E.\text{Lam}
\]

\[
\text{Pow}
\]

\[
\text{Lam} ^ E
\]

\[
\lambda \text{ID} . E
\]

\[
E.\text{Pow} > E.\text{Lam}
\]

\[
\text{Pow}
\]

\[
E ^ \text{Lam}
\]

\[
\lambda \text{ID} . E
\]

**Trees**

\[
a * b ^ \lambda x. c + d
\]

\[
\text{Add}
\]

\[
\text{Mul} + d
\]

\[
a * \text{Pow}
\]

\[
b ^ \text{Lam}
\]

\[
\lambda x. \text{Add}
\]

\[
c + d
\]
Deep Priority Conflicts: Match Subpattern in Right-Most Subtree

Add
/ \  
Lam + E
\   /
λ ID . E

Add
/ \  
C₁ + E
\   /
λ ID . E

Add
/ \  
C₁ + E
\   /
α Lam

Add
/ \  
C₁ + E
\   /
β Lam

Add
/ \  
C₂ + E
\   /
α Lam

Add
/ \  
C₂ + E
\   /
α Lam

Add
/ \  
C₁ + E
\   /
λ ID . E

Add
/ \  
C₂ + E
\   /
λ ID . E

... 

Infinite set of conflict patterns

Amorim, Visser: A direct semantics of declarative disambiguation rules. (Under revision)
context-free syntax
Exp.Min = <<Exp> - <Exp>> \{left\}
Exp.Add = <<Exp> + <Exp>> \{left\}
Exp.Mul = <<Exp> * <Exp>> \{left\}

context-free priorities
Exp.Mul > \{left: Exp.Min Exp.Add\}
context-free syntax
Exp.Min = <<Exp> - <Exp>> {left}
Exp.Add = <<Exp> + <Exp>> {left}
Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
Exp.Mul
> {left, right: Exp.Min Exp.Add}
Incomplete: Too Few Disambiguation Rules

context-free syntax
Exp.Min = <<Exp> - <Exp>> \{left\}
Exp.Add = <<Exp> + <Exp>> \{left\}
Exp.Mul = <<Exp> * <Exp>> \{left\}

context-free priorities
\{left: Exp.Min Exp.Add\}
What is the semantics of associativity and priority rules?
- Subtree exclusion: (deep) tree patterns that are forbidden

Is a set of disambiguation rules safe?
- At most one rule for each pair of productions

Is a set of disambiguation rules complete?
- At least one rule for each pair of productions

Correctness guaranteed by language definition
- Manual disambiguation by transformation of grammars is non-trivial
- Proof of safety and completeness is non-trivial
Parenthesize = Disambiguate\(^{-1}\) (Insert Necessary Parentheses)

**context-free syntax**
- \( \text{Exp} = \langle \langle \text{Exp} \rangle \rangle \) \{bracket\}
- \( \text{Exp}.\text{Int} = \text{INT} \)
- \( \text{Exp}.\text{Var} = \text{ID} \)
- \( \text{Exp}.\text{Add} = \langle \langle \text{Exp} \rangle \rangle + \langle \text{Exp} \rangle \rangle \) \{left\}
- \( \text{Exp}.\text{Let} = \langle \text{let} \langle \text{Bnd}^* \rangle \text{ in} \langle \text{Exp} \rangle \rangle \)
- \( \text{Bnd}.\text{Bnd} = \langle \langle \text{ID} \rangle = \langle \text{Exp} \rangle \rangle \)

**context-free priorities**
- \( \text{Exp}.\text{Add} > \text{Exp}.\text{Let} \)

\[
(a + (\text{let } x = b \text{ in } c)) + d
\]

```plaintext
Add(
  Add(
    Var("a")
    , Let([Bnd("x", Var("b"))], Var("c"))
    , Var("d")
  )
)
```

\[
a + (\text{let } x = b \text{ in } c) + d
\]
Layout-Sensitive Syntax
guessValue x = do
putStrLn "Enter your guess:"
guess <- getLine
case compare (read guess) x of
  EQ -> putStrLn "You won!"
_ -> do putStrLn "Keep guessing."
        guessValue x

if x \neq y:
  if x > 0:
    y = x
else:
  y = -x
Token Selectors Identify Two-Dimensional Structure

\[
x = \text{do}\begin{array}{c}
9 + 4 \\
* 3
\end{array}
\]

main = \text{do}\begin{array}{c}
\text{putStrLn } \$
\text{show (x * 2)}$
\end{array}
context-free syntax

\[
\begin{align*}
\text{Exp.Do} &= "do" \text{ ExpList} \\
\text{ExpList.Cns} &= \text{Exp} \\
\text{ExpList.Lst} &= \text{ExpList Exp} \{\text{layout}(1.\text{first.col} \equiv 2.\text{first.col})\} \\
\text{Exp.Id} &= \text{ID}
\end{align*}
\]
Alignment Declaration

context-free syntax

```
Exp.Do  = "do"  ExpList
ExpList.Cns = Exp
ExpList.Lst = exps:ExpList  exp:Exp  {layout(align exps exp)}
Exp.Id   = ID
```

Semantics

```
x.first.col == y.first.col
align x y
```

Amorim, Steindorfer, Erdweg, Visser: Declarative specification of indentation rules. SLE 2018
List Alignment Declaration

context-free syntax

\[\text{Exp.Do} = \text{"do" exps:Exp+ \{layout(align-list exps)\}}\]
\[\text{Exp.Id} = \text{ID}\]
\[\text{Exp+} = \text{Exp+ Exp // normalized}\]
\[\text{Exp+} = \text{Exp // productions}\]

Semantics

\[A+ = A+ A \text{ layout(1.first.col \rightleftharpoons 2.first.col)}\]
\[\text{align-list } x\]
"The offside rule prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure."

Erdweg et. al.. Layout-Sensitive Generalized Parsing. In SLE’12.
“The offside rule prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure.”

Erdweg et. al.. Layout-Sensitive Generalized Parsing. In SLE’12.
context-free syntax

Exp.Do = "do" exp:Exp {layout(offside exp)}
Exp.Add = Exp "+" Exp {left}
Exp.Id = ID

Semantics

\[ \text{x.left.col} > \text{x.first.col} \]
\[ \text{offside x} \]
Relative Offside

class context-free syntax

| Exp.Do = "do" exp:Exp {layout(offside "do" exp)} |
| Exp.Add = Exp "+" Exp {left} |
| Exp.Id = ID |

![Syntax diagrams showing the first and left sets of expressions involving do and + operators, with offside x y indicating semantic condition: y.left.col > x.first.col]
context-free syntax
Exp.Do = "do" exp:Exp {layout(indent "do" exp)}
Exp.Add = Exp "+" Exp {left}
Exp.Id = ID

Semantics
y.first.col > x.first.col
\[\text{indent } x \text{ y}\]
context-free syntax

Exp.Do = "do" exp:Exp {layout(newline-indent "do" exp)}
Exp.Add = Exp "+" Exp {left}
Exp.Id = ID
How does program layout disambiguate structure?
Spoofax Language Workbench
Conclusion
High-Level Declarative Domain-Specific Language

- Context-free grammars extended with
- Constructors
- Template productions
- Disambiguation rules
- Layout constraints
- All syntactic aspects of language in one specification

Multi-Purpose Interpretation

- Parsing, Recovery, Syntax Highlighting, Formatting, Completion, Fuzzing, Testing, Parenthesis Insertion, Signature Generation, ...
- Possible because high-level and declarative

A work in progress
High-Level Declarative Domain-Specific Language
- Declarative semantics
- Abstracts from implementation details
- All aspects of language in one specification

Multi-Purpose Interpretation
- Many tools from one specification
- Execution, Generation, Fuzzing, Analysis, Completion, Reverse Engineering, …
Other Spoofax Meta-Languages

Statix
- static semantics (w/ scope graphs)

Dynamix
- dynamic semantics

FlowSpec
- data-flow analysis

Stratego
- transformation strategies
Other Domains

**WebDSL**
- web programming

**IceDust**
- declarative data modeling
- derivation of incremental computation

**CSX**
- configuration space exploration
Domain-Specific Language
- encodes rules of the domain
- *declarative semantics: formally specified, easy to understand*
- users focus on domain programs

Multiple Interpretations
- operational semantics: sound wrt declarative semantics
- *intrinsically verified (sound by construction)*
- *operational semantics $\Rightarrow$ implementation*

Language Designer’s Workbench
- helps you put this all together with meta-DSLs

*Sooner than another 25 years … ?*