

CS4200-B: Compiler Construction Back-End

Eelco Visser



CS4200 | Compiler Construction | November 12, 2020

CS4200-A: Front-End (5 ECTS)

- Syntax and Type Checking
- Project: Build front-end of compiler for ChocoPy in Spooftax
- Exam in October

CS4200-B: Back-End (5 ECTS)

- Analysis and Code Generation
- Project: Build back-end of compiler for ChocoPy in Spooftax
- Exam in January

Lecture Topics CS4200-B (Q2) (Tentative)

- Transformation
- Virtual Machines
- Code Generation
- Data-Flow Analysis
- Monotone Frameworks
- Register Allocation
- Memory Management

ChocoPy: Project Language

ChocoPy: A Typed Restricted Subset of Python 3

```
# Binary-search trees
class TreeNode(object):
    value:int = 0
    left:"TreeNode" = None
    right:"TreeNode" = None

    def insert(self:"TreeNode", x:int) → bool:
        if x < self.value:
            if self.left is None:
                self.left = makeNode(x)
            return True
        else:
            return self.left.insert(x)
    elif x > self.value:
        if self.right is None:
            self.right = makeNode(x)
            return True
        else:
            return self.right.insert(x)
    return False

    def contains(self:"TreeNode", x:int) → bool:
        if x < self.value:
            if self.left is None:
                return False
            else:
                return self.left.contains(x)
        elif x > self.value:
            if self.right is None:
                return False
            else:
                return self.right.contains(x)
        else:
            return True
```

ChocoPy is a programming language designed for classroom use in undergraduate compilers courses. ChocoPy is a restricted subset of [Python 3](#), which can easily be compiled to a target such as [RISC-V](#). The language is [fully specified using formal grammar, typing rules, and operational semantics](#). ChocoPy is used to teach [CS 164](#) at UC Berkeley. ChocoPy has been designed by Rohan Padhye and Koushik Sen, with substantial contributions from Paul Hilfinger.

At a glance, ChocoPy is:

- **Familiar:** ChocoPy programs can be executed directly in a Python (3.6+) interpreter. ChocoPy programs can also be edited using standard Python syntax highlighting.
- **Safe:** ChocoPy uses Python 3.6 [type annotations](#) to enforce static type checking. The [type system](#) supports nominal subtyping.
- **Concise:** A full compiler for ChocoPy be implemented in about 12 weeks by undergraduate students of computer science. This can be a hugely rewarding exercise for students.
- **Expressive:** One can write non-trivial ChocoPy programs using lists, classes, and nested functions. Such language features also lead to interesting implications for compiler design.

Bonus: Due to static type safety and ahead-of-time compilation, most student implementations outperform the reference Python implementation on non-trivial benchmarks.

A Compiler and IDE for ChocoPy

```
binary_tree.pyx
50             return False
51     else:
52         return self.root.contains(x)
53
54 def makeNode(x: int) -> TreeNode:
55     b:TreeNode = None
56     b = TreeNode()
57     b.value = x
58     return b
59
60
```

ChocoPy IDE with syntax checking, syntax coloring, type checking (CS4200-A)

```
65 # Data
66 t:Tree = None
67 i:int = 0
68 k:int = 37813
69
70 # Crunch
71 t = Tree()
72 while i < n:
73     t.insert(k)
74     k = (k * 37813) % 37831
75     if i % c != 0:
76         t.insert(i)
77     i = i + 1
78
79 print(t.size)
80
81 for i in [4, 8, 15, 16, 23, 42]:
82     if t.contains(i):
83         print(i)
84
```

```
binary_tree.rv32im
1 .equiv @sbrk, 9
2 .equiv @print_string, 4
3 .equiv @print_char, 11
4 .equiv @print_int, 1
5 .equiv @exit2, 17
6 .equiv @read_string, 8
7 .equiv @fill_line_buffer, 18
8 .equiv @_obj_size_, 4
9 .equiv @_len_, 12
10 .equiv @_int_, 12
11 .equiv @_len_
12 .equiv @_int_
13 .equiv @_obj_size_
14 .equiv @error_div_zero, 2
15 .equiv @error_arg, 1
16 .equiv @error_oob, 3
17 .equiv @error_none, 4
18 .equiv @error_oom, 5
19 .equiv @error_nyi, 6
20 .equiv @listHeaderWords, 4
21 .equiv @bool.True, const_39
22 .equiv @bool.False, const_38
23
24 .data
25
26 .globl $object$prototype
27 $object$prototype:
28 .word @
```

```
binary_tree.result.txt
1 175
2 15
3 23
4 42
5
```

Compiler from ChocoPy to RISC-V (CS4200-B)

Executing RISC-V with simulator

ChocoPy: Language Manual and Reference

Start Reading!

ChocoPy v2.2: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen; v2 changes by Paul Hilfnger

University of California, Berkeley

November 23, 2019

Contents

1	Introduction	3
2	A tour of ChocoPy	3
2.1	The top level	4
2.2	Functions	4
2.3	Classes	5
2.4	Type hierarchy	6
2.5	Values	7
2.5.1	Integers	7
2.5.2	Booleans	7
2.5.3	Strings	7
2.5.4	Lists	7
2.5.5	Objects of user-defined classes	7
2.5.6	None	8
2.5.7	The empty list (<code>[]</code>)	8
2.6	Expressions	8
2.6.1	Literals and identifiers	8
2.6.2	List expressions	8
2.6.3	Arithmetic expressions	8
2.6.4	Logical expressions	8
2.6.5	Relational expressions	9
2.6.6	Conditional expressions	9
2.6.7	Concatenation expressions	9
2.6.8	Access expressions	9
2.6.9	Call expressions	9
2.7	Type annotations	9
2.8	Statements	10
2.8.1	Expression statements	10
2.8.2	Compound statements: conditionals and loops	10
2.8.3	Assignment statements	11
2.8.4	Pass statement	11
2.8.5	Return statement	11
2.8.6	Predefined classes and functions	11

Homework in WebLab

#spofax-users channel on research group Slack

- Where many Spofax users hang out
- Ask questions about Spofax
- (Not answers to assignments)
- No guarantee to quick response, or answer at all

Send me an email if you want an invitation

Program Transformation by Term Rewriting

Eelco Visser



CS4200 | Compiler Construction | November 12, 2020

This Lecture



Define transformations on abstract syntax trees (terms) using rewrite rules

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

Term rewrite rules define transformations on (abstract syntax) trees. Traditional rewrite systems apply rules exhaustively. This paper introduces programmable rewriting strategies to control the application of rules, the core of the design of the Stratego transformation language.

Note that the notation for contextual rules is no longer supported by Stratego. However, the technique to implement contextual rules still applies.

ICFP 1998

<https://doi.org/10.1145/291251.289425>

Building Program Optimizers with Rewriting Strategies*

Eelco Visser¹, Zine-el-Abidine Benissa¹, Andrew Tolmach^{1,2}
Pacific Software Research Center

¹ Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA

² Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA

visser@acm.org, benissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

We describe a language for defining term rewriting strategies, and its application to the production of program optimizers. Valid transformations on program terms can be described by a set of rewrite rules; rewriting strategies are used to describe when and how the various rules should be applied in order to obtain the desired optimization effects. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer as a whole, compared to traditional monolithic optimizer implementations. We illustrate the expressiveness of our language by using it to describe a simple optimizer for an ML-like intermediate representation.

The basic strategy language uses operators such as sequential composition, choice, and recursion to build transformers from a set of labeled unconditional rewrite rules. We also define an extended language in which the side-conditions and contextual rules that arise in realistic optimizer specifications can themselves be expressed as strategy-driven rewrites. We show that the features of the basic and extended languages can be expressed by breaking down the rewrite rules into their primitive building blocks, namely matching and building terms in variable binding environments. This gives us a low-level core language which has a clear semantics, can be implemented straightforwardly and can itself be optimized. The current implementation generates C code from a strategy specification.

1 Introduction

Compiler components such as parsers, pretty-printers and code generators are routinely produced using program generators. The component is specified in a high-level language from which the program generator produces its implementation. Program optimizers are difficult labor-intensive components that are usually still developed manually, despite many attempts at producing optimizer generators (e.g., [19, 12, 28, 25, 18, 11]).

*This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP '98 Baltimore, MD USA
© 1998 ACM 1-58113-024-4/98/0009...\$5.00

A program optimizer transforms the source code of a program into a program that has the same meaning, but is more efficient. On the level of specification and documentation, optimizers are often presented as a set of correctness-preserving *rewrite rules* that transform code fragments into equivalent more efficient code fragments (e.g., see Table 5). This is particularly attractive for functional language compilers (e.g., [3, 4, 24]) that operate via successive small transformations, and don't rely on analyses requiring significant auxiliary data structures. The paradigm provided by conventional rewrite engines is to compute the normal form of a program with respect to a set of rewrite rules. However, optimizers are usually not implemented in this way. Instead, an algorithm is produced that implements a *strategy* for applying the optimization rules. Such a strategy contains meta-knowledge about the set of rewrite rules and the programming language they are applied to in order to (1) control the application of rules; (2) guarantee termination of optimization; (3) make optimization more efficient.

Such an ad-hoc implementation of a rewriting system has several drawbacks, even when implemented in a language with good support for pattern matching, such as ML or Haskell. First of all, the transformation rules are embedded in the code of the optimizer, making them hard to understand, to maintain, and to reuse individual rules in other transformations. Secondly, the strategy is not specified at the same level of abstraction as the transformation rules, making it hard to reason about the correctness of the optimizer even if the individual rules are correct. Finally, the host language has no awareness of the transformation domain underlying the implementation and can therefore not use this domain knowledge to optimize the optimizer itself.

It would be desirable to apply term rewriting technology directly to produce program optimizers. However, the standard approach to rewriting is to provide a fixed strategy (e.g., innermost or outermost) for normalizing a term with respect to a set of user-defined rewrite rules. This is not satisfactory when—as is usually the case for optimizers—the rewrite rules are neither confluent nor terminating. A common work-around is to encode a strategy into the rules themselves, e.g., by using an explicit function symbol that controls where rewrites are allowed. But this approach has the same disadvantages as the ad-hoc implementation of rewriting described above: the rules are hard to read, and the strategies are still expressed at a low level of abstraction.

In this paper we argue that a better solution is to use explicit specification of *rewriting strategies*. We show how

Stratego/XT combines SDF2 and Stratego into toolset for program transformation.

This paper gives a high-level overview of the concepts.

The StrategoXT.jar is still part of the Spoofax distribution.

Program Transformation with Stratego/XT Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80089 3508 TB, Utrecht, The Netherlands
visser@acm.org
<http://www.stratego-language.org>

Abstract. Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation systems including parsing and pretty-printing. The framework addresses the entire range of the development process; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

1 Introduction

Program transformation, the automatic manipulation of source programs, emerged in the context of compilation for the implementation of components such as optimizers [28]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [13], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [21], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation a necessary operation in any scenario where structured data play a role. Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jstraca [31], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation

Lecture Notes in Computer Science 2003

https://doi.org/10.1007/978-3-540-25935-0_13

C. Lengauer et al. (Eds.): Domain-Specific Program Generation, LNCS 3016, pp. 216–238, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Spoofax combines SDF2 and Stratego into a language workbench, i.e. an IDE for creating language definition from which IDEs for the defined languages can be generated.

A distinctive feature of Spoofax is live language development, which supports developing a language definition and programs in the defined language in the same IDE instance.

Spoofax was developed for Eclipse, which is still the main development platform. However, Spoofax Core is now independent of any IDE.

Note that since the publication of this paper, we have introduced more declarative approaches to name and type analysis, which will be the topic of the next lectures.

OOPSLA 2010

<https://doi.org/10.1145/1932682.1869497>

The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats

Delft University of Technology

l.c.l.kats@tudelft.nl

Eelco Visser

Delft University of Technology

visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10...\$10.00

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a “build” button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Documentation for Stratego at
the metaborg.org website.

Spoofax
latest

Search docs

The Spoofax Language Workbench
Examples
Publications

TUTORIALS

Installing Spoofax
Creating a Language Project
Using the API
Getting Support

REFERENCE MANUAL

Language Definition with Spoofax
Abstract Syntax with ATerms
Syntax Definition with SDF3
Static Semantics with NaBL2

Transformation with Stratego

Stratego Tutorial/Reference (Old)
The Stratego Library
Concrete Syntax in Stratego Transformations

Docs » Transformation with Stratego

 Edit on GitHub

Transformation with Stratego

Parsing a program text results in an abstract syntax tree. Stratego is a language for defining transformations on such trees. Stratego provides a term notation to construct and deconstruct trees and uses *term rewriting* to define transformations. Instead of applying all rewrite rules to all sub-terms, Stratego supports programmable *rewriting strategies* that control the application of rewrite rules.

- [Stratego Tutorial/Reference \(Old\)](#)
- [The Stratego Library](#)
- [Concrete Syntax in Stratego Transformations](#)

 Previous

Next 

© Copyright 2016-2017, MetaBorg. Revision `ce45afc9`.

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

This paper defines the formal semantics of the full Stratego language, including its scoped dynamic rules feature.

These slides document most features of the base language without dynamic rules and without giving the formal semantics.

If you want to dig deeper

Fundamenta Informaticae 69 (2006) 123–178

IOS Press

Program Transformation with Scoped Dynamic Rewrite Rules

Martin Bravenboer, Arthur van Dam, Karina Olmos and Eelco Visser*

Department of Information and Computing Sciences

Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht

The Netherlands

visser@acm.org

Abstract. The applicability of term rewriting to program transformation is limited by the lack of control over rule application and by the context-free nature of rewrite rules. The first problem is addressed by languages supporting user-definable rewriting strategies. The second problem is addressed by the extension of rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are defined at run-time and can access variables available from their definition context. Rules defined within a rule scope are automatically retracted at the end of that scope. In this paper, we explore the design space of dynamic rules, and their application to transformation problems. The technique is formally defined by extending the operational semantics underlying the program transformation language Stratego, and illustrated by means of several program transformations in Stratego, including constant propagation, bound variable renaming, dead code elimination, function inlining, and function specialization.

1. Introduction

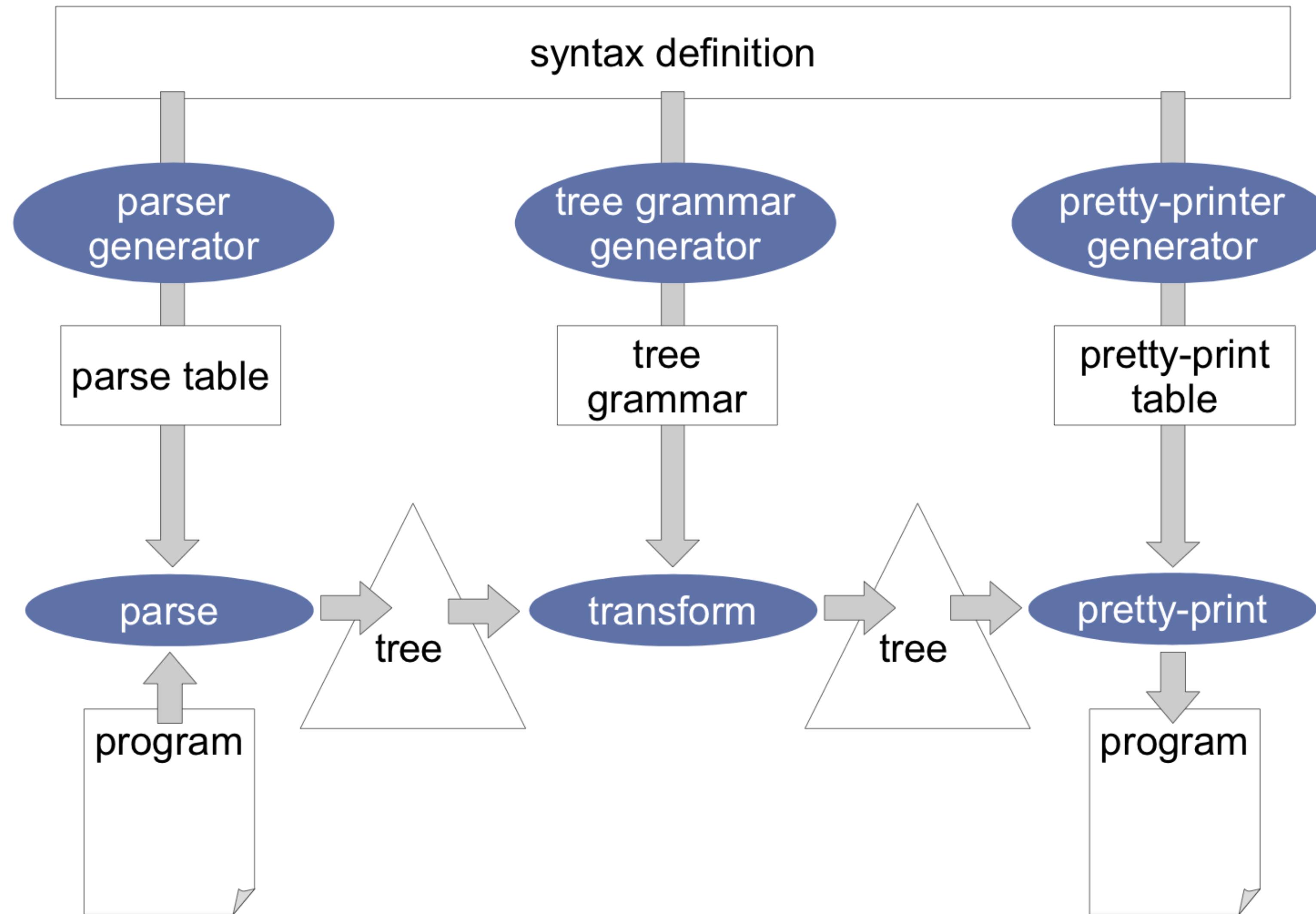
Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function C such that $C(P) > C(tr(P))$, i.e. the cost decreases as a result of applying the transformation [30, 29, 11]. The cost of a program can be measured in different dimensions such as performance, memory usage, understandability, flexibility, maintainability, portability, correctness, or satisfaction of requirements. Related to these goals, program transformations are applied in different settings; e.g. compiler optimizations improve performance [24] and refactoring tools aim at improving understandability [28, 14]. While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by *automating*

*Address for correspondence: Department of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

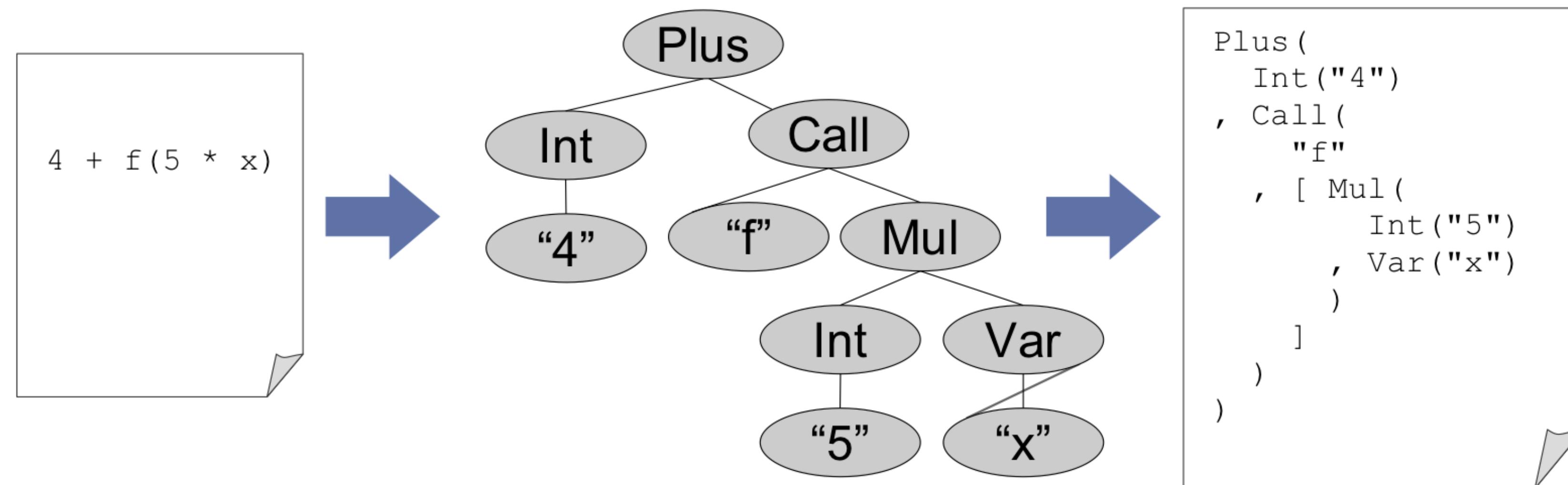
Transformation Architecture

This presentation uses the Stratego Shell for explaining the behavior of Stratego programs. The Stratego Shell is currently not supported by Spoofax

Architecture of Stratego/XT



Programs as Terms



Trees are represented as terms in the ATerm format

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

ATerm Format

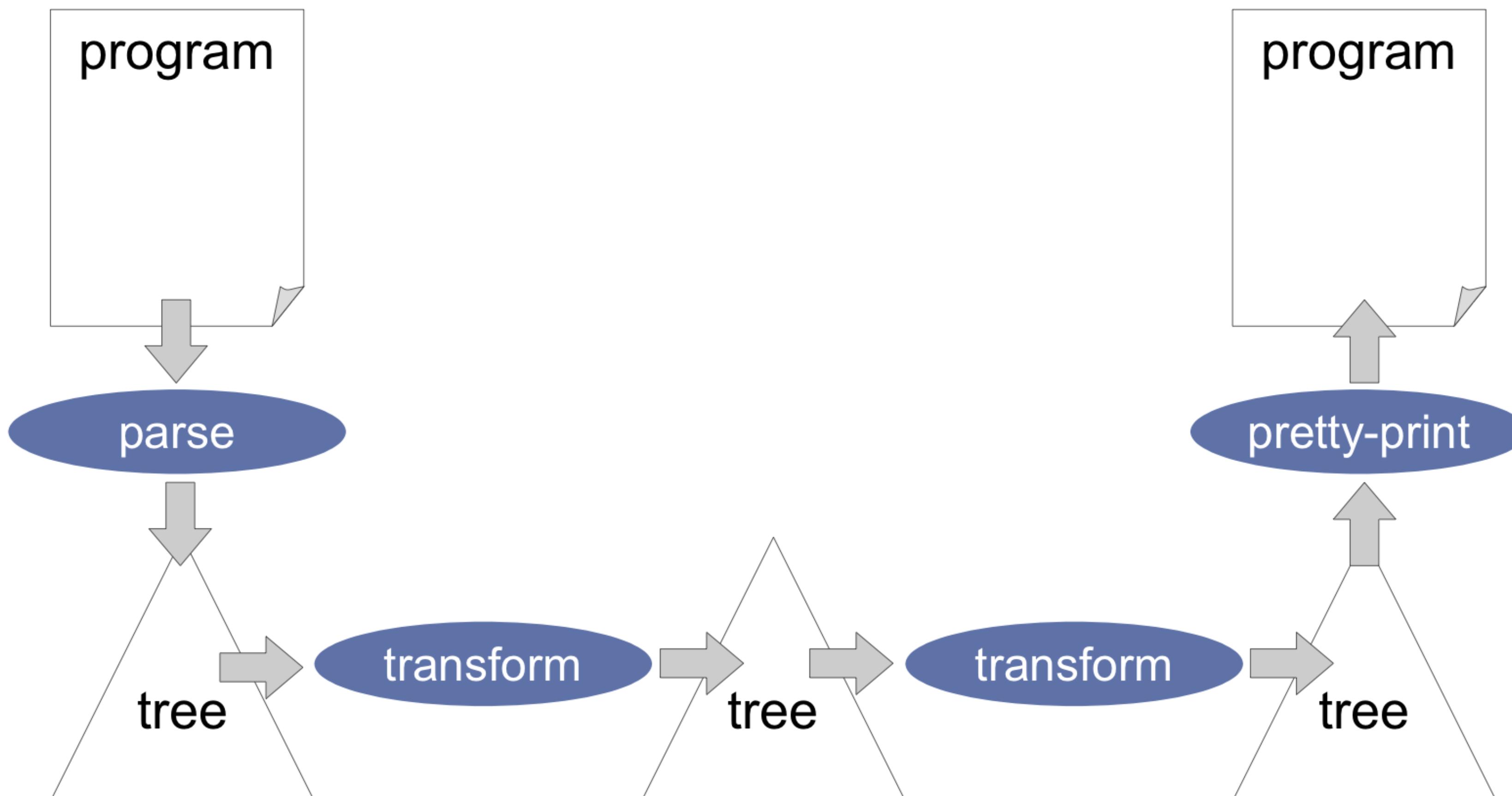
Application	Void(), Call(t , t)
List	[], [t , t , t]
Tuple	(t , t), (t , t , t)
Integer	25
Real	38.87
String	"Hello world"
Annotated term	$t\{t, t, t\}$

- Exchange of structured data
- Efficiency through maximal sharing
- Binary encoding

Structured Data: comparable to XML

Stratego: internal is external representation

How to Realize Program Transformations?



Conventional Term Rewriting

- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
- (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine

Strategic Term Rewriting

- Select rules to use in a specific transformation
- Select strategy to apply
- Define your own strategy if necessary
- Combine strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

<current term>

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>
stratego> <strategy expression>
<transformed term>
```

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

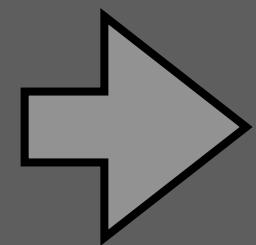
Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>
stratego> <strategy expression>
<transformed term>
stratego> <strategy expression>
command failed
```

Terms

Parsing: From Text to Terms

```
let function fact(n : int) : int =
    if n < 1 then 1 else (n * fact(n - 1))
in fact(10)
end
```



```
Let(
  [ FunDec(
      "fact"
    , [FArg("n", Tp(Tid("int")))]
    , Tp(Tid("int"))
    , If(
        Lt(Var("n"), Int("1"))
      , Int("1")
      , Seq(
          [ Times(
              Var("n")
            , Call(
                Var("fact")
              , [Minus(Var("n"), Int("1"))]
            )
          )
        ]
      )
    )
  ]
, [Call(Var("fact"), [Int("10")])]
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

Cons = [a-zA-Z][a-zA-Z0-9]*
context-free syntax

Term.App = <<Cons>(<{Term ", "}*>)>
```

```
Zero()
Succ(Zero())
Cons(A(), Cons(B(), Nil()))
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

Cons = [a-zA-Z][a-zA-Z0-9]*
```

context-free syntax

```
Term.App  = <<Cons>(<{Term ","}*>)>
Term.List = <[<{Term ","}*>]>
Term.Tuple = <(<{Term ","}*>)>
```

```
Zero()
Succ(Zero())
[A(), B()]
```

Syntax of Terms

```
module ATerms

sorts Cons Term

lexical syntax
  Cons      = [a-zA-Z][a-zA-Z0-9]*
  Cons      = String
  Int       = [0-9]+
  String    = "\"" StringChar* "\""
  StringChar = ~["\\n"]
  StringChar = "\\\" [\\\"]"

context-free syntax
  Term.Str  = <<String>>
  Term.Int   = <<Int>>
  Term.App   = <<Cons>(<{Term ","}*>) >
  Term.List  = <[<{Term ","}*>]>
  Term.Tuple = <(<{Term ","}*>)>
```

```
0
1
[A(), B()]
Var("x\\\")

Let(
  [ Decl("x", IntT()),
    Decl("y", BoolT())
  ]
, Eq(Var("x"), Int(0))
)
```

Syntax of A(nnotated) Terms

```
module ATerms

sorts Cons Term

lexical syntax
  Cons = [a-zA-Z][a-zA-Z0-9]*
  // more lexical syntax omitted

context-free syntax

  Term.Anno      = <<PreTerm>{<{Term ","}*}>>
  Term           = <<PreTerm>>

  PreTerm.Str    = <<String>>
  PreTerm.Int    = <<Int>>
  PreTerm.App    = <<Cons>(<{Term ","}*}>>
  PreTerm.List   = <[<{Term ","}*}>>
  PreTerm.Tuple  = <(<{Term ","}*}>>
```

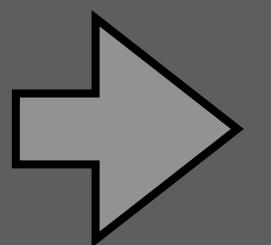
```
Var("x"){Type(IntT())}
```

Signatures

Signatures

context-free syntax

```
Exp.Uminus = [- [Exp]]  
Exp.Power = [[Exp] ** [Exp]]  
Exp.Times = [[Exp] * [Exp]]  
Exp.Divide = [[Exp] / [Exp]]  
Exp.Plus = [[Exp] + [Exp]]  
Exp.Minus = [[Exp] - [Exp]]  
Exp.CPlus = [[Exp] +i [Exp]]  
Exp.CMinus = [[Exp] -i [Exp]]  
Exp.Eq = [[Exp] = [Exp]]  
Exp.Neq = [[Exp] ◇ [Exp]]  
Exp.Gt = [[Exp] > [Exp]]  
Exp.Lt = [[Exp] < [Exp]]  
Exp.Geq = [[Exp] ≥ [Exp]]  
Exp.Leq = [[Exp] ≤ [Exp]]  
Exp.True = <true>  
Exp.False = <false>  
Exp.And = [[Exp] & [Exp]]  
Exp.Or = [[Exp] | [Exp]]
```



Signature declares argument and return types of term constructors

signature constructors

Uminus	: Exp → Exp
Power	: Exp * Exp → Exp
Times	: Exp * Exp → Exp
Divide	: Exp * Exp → Exp
Plus	: Exp * Exp → Exp
Minus	: Exp * Exp → Exp
CPlus	: Exp * Exp → Exp
CMinus	: Exp * Exp → Exp
Eq	: Exp * Exp → Exp
Neq	: Exp * Exp → Exp
Gt	: Exp * Exp → Exp
Lt	: Exp * Exp → Exp
Geq	: Exp * Exp → Exp
Leq	: Exp * Exp → Exp
True	: Exp
False	: Exp
And	: Exp * Exp → Exp
Or	: Exp * Exp → Exp

Signature is automatically generated
from SDF3 productions

Stratego compiler only checks *arity*
of constructor applications

Rewrite Rules

Desugaring

```
if x then  
  printint(x)
```

```
if x then  
  printint(x)  
else  
  ()
```

```
IfThen(  
  Var("x")  
, Call(  
  "printint"  
, [Var("x")])  
)
```

```
IfThenElse(  
  Var("x")  
, Call(  
  "printint"  
, [Var("x")])  
, NoVal()  
)
```

pattern matching

pattern instantiation

```
desugar: IfThen(e1, e2) → IfThenElse(e1, e2, NoVal())
```

Lists of Elselfs

signature

constructors

If : Exp * Exp * List(ElseIf) → Exp

ElseIf : Exp * Exp → ElseIf

IfThen : Exp * Exp * Exp → Exp

```
If(c, e1, [
  ElseIf(c2, e2),
  ElseIf(c3, e3),
  ...
])
```

Desugar :

```
If(c, e, []) → IfThen(c, e, NoVal())
```

Desugar :

```
If(c, e, [ElseIf(c2, e2) | es]) → IfThen(c, e, If(c2, e2, es))
```

More Desugaring

signature
constructors

PLUS: BinOp
MINUS: BinOp
MUL: BinOp
DIV: BinOp

EQ: RelOp
NE: RelOp
LE: RelOp
LT: RelOp

Bop: BinOp * Expr * Expr → Expr
Rop: RelOp * Expr * Expr → Expr

desugar: Uminus(e) → Bop(MINUS(), Int("0"), e)

desugar: Plus(e1, e2) → Bop(PLUS(), e1, e2)
desugar: Minus(e1, e2) → Bop(MINUS(), e1, e2)
desugar: Times(e1, e2) → Bop(MUL(), e1, e2)
desugar: Divide(e1, e2) → Bop(DIV(), e1, e2)

desugar: Eq(e1, e2) → Rop(EQ(), e1, e2)
desugar: Neq(e1, e2) → Rop(NE(), e1, e2)
desugar: Leq(e1, e2) → Rop(LE(), e1, e2)
desugar: Lt(e1, e2) → Rop(LT(), e1, e2)
desugar: Geq(e1, e2) → Rop(LE(), e2, e1)
desugar: Gt(e1, e2) → Rop(LT(), e2, e1)

desugar: And(e1, e2) → IfThenElse(e1, e2, Int("0"))
desugar: Or(e1, e2) → IfThenElse(e1, Int("1"), e2)

Constant Folding

```
x := 21 + 21 + x
```

```
x := 42 + x
```

```
Assign(  
  Var("x")  
, Plus(  
  Plus(  
    Int("21")  
, Int("21"))  
, Var("x"))  
)
```

```
Assign(  
  Var("x")  
, Plus(  
  Int("42")  
, Var("x"))  
)
```

```
eval: Plus(Int(i1), Int(i2)) → Int(i3)  
where <addS> (i1, i2) ⇒ i3
```

More Constant Folding

```
eval: Bop(PLUS(), Int(i1), Int(i2)) → Int(<addS> (i1, i2))

eval: Bop(MINUS(), Int(i1), Int(i2)) → Int(<subS> (i1, i2))

eval: Bop(MUL(), Int(i1), Int(i2)) → Int(<mulS> (i1, i2))

eval: Bop(DIV(), Int(i1), Int(i2)) → Int(<divS> (i1, i2))

eval: Rop(EQ(), Int(i), Int(i)) → Int("1")
eval: Rop(EQ(), Int(i1), Int(i2)) → Int("0") where not( <eq> (i1, i2) )

eval: Rop(NE(), Int(i), Int(i)) → Int("0")
eval: Rop(NE(), Int(i1), Int(i2)) → Int("1") where not( <eq> (i1, i2) )

eval: Rop(LT(), Int(i1), Int(i2)) → Int("1") where <ltS> (i1, i2)
eval: Rop(LT(), Int(i1), Int(i2)) → Int("0") where not( <ltS> (i1, i2) )

eval: Rop(LE(), Int(i1), Int(i2)) → Int("1") where <leqS> (i1, i2)
eval: Rop(LE(), Int(i1), Int(i2)) → Int("0") where not( <leqS> (i1, i2))
```

Application: Spoofax Builders

Spoofax Builders

```
rules

build :
  (node, _, ast, path, project-path) → result
with
  <some-transformation> ast ⇒ result
```

Builder gets AST as parameter

Tiger: AST Builder

```
module Syntax

menus

menu: "Syntax" (openeditor)

action: "Format"           = editor-format (source)
action: "Show parsed AST" = debug-show-aterm (source)
```

editor/syntax.esv

```
rules // Debugging

debug-show-aterm:
  (node, _, _, path, project-path) → (filename, result)
  with
    filename := <guarantee-extension(|"aterm")> path
  ; result   := node
```

trans/tiger.str

Builder gets AST as parameter

Application: Formatting

Tiger: Formatting Builder

```
module Syntax

menus

menu: "Syntax" (openeditor)

action: "Format"          = editor-format (source)
action: "Show parsed AST" = debug-show-aterm (source)
```

editor/syntax.esv

trans/pp.str

trans/pp.str

```
rules

editor-format:
  (node, _, ast, path, project-path) → (filename, result)
  with
    ext      := <get-extension> path
    ; filename := <guarantee-extension(|$[pp.[ext]]|)> path
    ; result   := <pp-debug> node
```

rules

```
pp-Tiger-string =
  parenthesize-Tiger
  ; prettyprint-Tiger-start-symbols
  ; !V([], <id>)
  ; box2text-string(|120)

pp-debug :
  ast → result
  with
    result := <pp-Tiger-string> ast
    <+ ...
    ; result := ""
```

Tiger: Parenthesize

```
module pp/Tiger-parenthesize

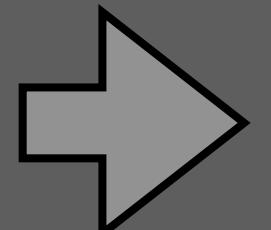
imports
    libstratego-lib
    signatures/-

strategies

parenthesize-Tiger =
    innermost(TigerParenthesize)
```

context-free priorities

```
Exp.And
> Exp.Or
> Exp.Array
> Exp.Assign
> ...
```



```
rules
TigerParenthesize :
Or(t_0, t_1) → Or(t_0, Parenthetical(t_1))
where <(?For( _, _, _, _)
+ ?While( _, _)
+ ?IfThen( _, _)
+ ?If( _, _, _)
+ ?Assign( _, _)
+ ?Array( _, _, _)
+ ?Or( _, _)
+ fail)> t_1

TigerParenthesize :
And(t_0, t_1) → And(Parenthetical(t_0), t_1)
where <(?For( _, _, _, _)
+ ?While( _, _)
+ ?IfThen( _, _)
+ ?If( _, _, _)
+ ?Assign( _, _)
+ ?Array( _, _, _)
+ ?Or( _, _)
+ fail)> t_0
```

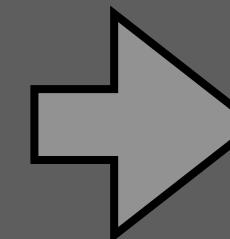
pp/Tiger-parenthesize.str

Tiger: Pretty-Print Rules

context-free syntax

```
Exp.If = <
  if <Exp> then
    <Exp>
  else
    <Exp>
>
```

syntax/Control-Flow.sdf3



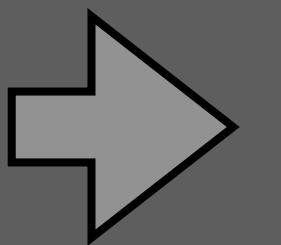
```
rules
  prettyprint-Tiger-Exp :
    If(t1_, t2_, t3_) → [ H(
      [S0pt(HS(), "0")]
      , [ S("if ")
        , t1_
        , S(" then")
      ]
      )
      , t2_
      , H(
        [S0pt(HS(), "0")]
        , [S("else")]
      )
      , t3_
    ]
    with t1_ := <pp-one-Z(prettyprint-Tiger-Exp)
          <+ pp-one-Z(prettyprint-completion-aux)> t1_
    with t2_ := <pp-indent(|"2")> [
      <pp-one-Z(prettyprint-Tiger-Exp)
      <+ pp-one-Z(prettyprint-completion-aux)> t2_ ]
    with t3_ := <pp-indent(|"2")> [
      <pp-one-Z(prettyprint-Tiger-Exp)
      <+ pp-one-Z(prettyprint-completion-aux)> t3_ ]
```

pp/Control-Flow-pp.str

Application: Desugaring

Tiger: Desugaring

```
function printboard() = (
  for i := 0 to N-1 do (
    for j := 0 to N-1 do
      print(if col[i]=j then " 0" else " .");
      print("\n")
    );
    print("\n")
)
```



```
function printboard() = (
  let
    var i : int := 0
  in
    while i < N - 1 do (
      let
        var j : int := 0
      in
        while j < N - 1 do (
          print(if col[i] = j then
                " 0"
              else
                " .");
          j := j + 1
        );
        print("\n")
      );
      i := i + 1
    );
    print("\n")
)
```

Expressing for in terms of while+

Tiger: Desugaring Builder

```
module Transformation

menus

menu: "Transform" (openeditor) (realtime)

action: "Desugar"      = editor-desugar (source)
action: "Desugar AST" = editor-desugar-ast (source)
```

editor/Transformation.esv

```
rules // Desugaring

editor-desugar :
  (node, _, _, path, project-path) → (filename, result)
  with
    filename := <guarantee-extension(|"des.tig")> path
  ; result   := <desugar-all; pp-Tiger-string>node

editor-desugar-ast :
  (node, _, _, path, project-path) → (filename, result)
  with
    filename := <guarantee-extension(|"atemp")> path
  ; result   := <desugar-all>node
```

trans/tiger.str

Tiger: Desugaring Transformation

```
module desugar
imports signatures/Tiger-sig
imports ...
strategies
  desugar-all = topdown(try(desugar))
rules
  desugar :
    For(
      Var(i)
    , e1
    , e2
    , e_body
    ) ->
    Let(
      [VarDec(i, Tid("int"), e1)]
    , [ While(
        Lt(Var(i), e2)
      , Seq(
          [ e_body
          , Assign(Var(i), Plus(Var(i), Int("1")))
          ]
        )
      )
    ]
  )
```

Application: Outline View

Tiger: Outline View

The screenshot shows the Tiger programming environment with two windows:

- queens.tig** window (left): Displays the source code for a program to solve the 8-queens problem. The code uses老虎 (Tiger) syntax, including `let`, `var`, `type`, `function`, `for` loops, and conditional statements like `if` and `try`. A specific section of the `printboard` function is highlighted with a blue selection bar.
- Outline** window (right): Shows a hierarchical outline of the program's structure. It includes:
 - A `let` block containing declarations for `N` (type `intArray`), `row`, `col`, `diag1`, and `diag2`.
 - A `fun printboard` block containing a nested `for i` loop.
 - A `for j` loop nested within the `for i` loop, which contains a `print` statement and an `if` condition.
 - A `fun try` block containing an `if` statement and a `try` block.

Tiger: Outline View Builder

```
module Syntax  
  
views  
  
outline view: editor-outline (source)  
  expand to level: 3
```

trans/outline.str

editor/syntax.esv

```
module outline  
  
imports  
  signatures/Tiger-sig  
  signatures/Types-sig  
  signatures/Records-sig  
  signatures/Variables-sig  
  signatures/Functions-sig  
  signatures/Bindings-sig  
  libspooftax/editor/outline  
  pp  
  
rules  
  
editor-outline:  
  (_, _, ast, path, project-path) → outline  
  where  
    outline := <simple-label-outline(to-outline-label)> ast
```

Tiger: Outline View Rules

rules

```
to-outline-label :  
    Let(_, _) -> ${[let]}
```

```
to-outline-label :  
    TypeDec(name, _) -> ${[type [name]]}
```

```
to-outline-label :  
    FunDec(name, _, t, _) -> ${[fun [name] : [<pp-tiger-type> t]]}
```

```
to-outline-label :  
    ProcDec(name, _, _) -> ${[fun [name]]}
```

```
to-outline-label :  
    VarDecNoType(name, _) -> ${[var [name]]}
```

```
to-outline-label :  
    VarDec(name, _, _) -> ${[var [name]]}
```

trans/outline.str

rules

```
to-outline-label :  
    Call(f, _) -> ${[f]()}]
```

```
to-outline-label :  
    If(_, _, _) -> ${[if]}
```

```
to-outline-label :  
    For(Var(name), _, _, _) -> ${[for [name]]}
```

// etc.

Outline View: Generic Strategy

```
module libspofax/editor/outline
imports ...
signature
constructors
  Node : Label * Children → Node
rules

  simple-label-outline(s1) =
    collect-om(to-outline-node(s1, fail), conc)

  custom-label-outline(s1, s2) =
    collect-om(origin-track-forced(s2) <+ to-outline-node(s1, s2), conc)

  to-outline-node(s1, s2):
    term → Node(label, children)
    where
      random := <next-random>;
      label := <origin-track-forced(s1; term-to-outline-label)> term;
      children := <get-arguments; custom-label-outline(s1, s2)> term

  term-to-outline-label =
    is-string
    <+ ?term{a}; origin-text; ?label; !label{a}
    <+ write-to-string // fallback
```

Term structure for outline nodes

Language-independent strategy for collecting outline nodes

Application: Completion

Tiger: Completion Rules

context-free syntax

```
Exp.If = <  
    if <Exp> then  
        <Exp>  
    else  
        <Exp>  
>
```

syntax/Control-Flow.sdf3



rules

```
suggest-completions(|completions):  
    Exp-Plhdr() -> <add-completions(  
        | ( "If"  
        , If(  
            <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
            , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
            , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
        )  
        )  
    ); fail> completions
```

completion/Control-Flow-cp.str

Combining Rewrite Rules with Strategies

Naming and Composing Strategies

Reuse of transformation requires definitions

1. Naming strategy expressions
2. Named rewrite rules
3. Reusing rewrite rules through modules

Simple strategy definition and call

- Syntax: $f = s$
- Name strategy expression s
- Syntax: f
- Invoke (call) named strategy f

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs = {e1,e2:(Plus(e1,e2) -> Plus(e2,e1))}
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Named rewrite rules (sugar)

- Syntax: $f : p_1 \rightarrow p_2 \text{ where } s$
- Name rewrite rule $p_1 \rightarrow p_2 \text{ where } s$
- Equivalent to: $f = \{x_1, \dots, x_n : (p_1 \rightarrow p_2 \text{ where } s)\}$
(with x_1, \dots, x_n the variables in p_1 , p_2 , and s)

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs : Plus(e1,e2) -> Plus(e2,e1)
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Example: Inverting If Not Equal

```
if(x != y)
    doSomething();
else
    doSomethingElse();
```

⇒

```
if(x == y)
    doSomethingElse();
else
    doSomething();
```

```
InvertIfNot :
  If(NotEq(e1, e2), stm1, stm2) ->
  If(Eq(e1, e2), stm2, stm1)
```

Modules with Reusable Transformation Rules

```
module Simplification-Rules
rules
    PlusAssoc :
        Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))

    EvalIf :
        If(Lit(Bool(True())), stm1, stm2) -> stm1

    EvalIf :
        If(Lit(Bool(False())), stm1, stm2) -> stm2

    IntroduceBraces :
        If(e, stm) -> If(e, Block([stm]))
        where <not(?Block(_))> stm
```

```
stratego> import Simplification-Rules
```

Rules define one-step transformations

Program transformations require many one-step transformations and selection of rules

1. Choice
2. Identity, Failure, and Negation
3. Parameterized and Recursive Definitions

Deterministic choice (left choice)

- Syntax: $s_1 \leftarrow s_2$
- First apply s_1 , if that fails apply s_2
- Note: local backtracking

```
PlusAssoc :  
    Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))  
EvalPlus :  
    Plus(Int(i), Int(j)) -> Int(k) where <addS>(i, j) => k
```

```
Plus(Int("14"), Int("3"))  
stratego> PlusAssoc  
command failed  
stratego> PlusAssoc <+ EvalPlus  
Int("17")
```

Composing Strategies

Guarded choice

- Syntax: $s_1 < s_2 + s_3$
- First apply s_1 if that succeeds apply s_2 to the result
else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- $s_1 <+ s_2$ always backtracks to s_2 if s_1 fails
- $(s_1 ; s_2) <+ s_3 \not\equiv s_1 < s_2 + s_3$
- commit to branch if test succeeds, even if that branch fails

```
test1 < transf1
+ test2 < transf2
+ transf3
```

Composing Strategies

Guarded choice

- Syntax: $s_1 < s_2 + s_3$
- First apply s_1 if that succeeds apply s_2 to the result
else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- $s_1 <+ s_2$ always backtracks to s_2 if s_1 fails
- $(s_1 ; s_2) <+ s_3 \not\equiv s_1 < s_2 + s_3$
- commit to branch if test succeeds, even if that branch fails

```
test1 < transf1
+ test2 < transf2
+ transf3
```

If then else (sugar)

- Syntax: if s_1 then s_2 else s_3 end
- Equivalent to: where(s_1) $< s_2 + s_3$

Composing Strategies

Identity

- Syntax: id
- Always succeed
- Some laws
 - $\text{id} ; s \equiv s$
 - $s ; \text{id} \equiv s$
 - $\text{id} \<+ s \equiv \text{id}$
 - $s \<+ \text{id} \not\equiv s$
 - $s_1 \< \text{id} + s_2 \equiv s_1 \<+ s_2$

Failure

- Syntax: fail
- Always fail
- Some laws
 - $\text{fail} \<+ s \equiv s$
 - $s \<+ \text{fail} \equiv s$
 - $\text{fail} ; s \equiv \text{fail}$
 - $s ; \text{fail} \not\equiv \text{fail}$

Composing Strategies

Identity

- Syntax: id
- Always succeed
- Some laws
 - $\text{id} ; s \equiv s$
 - $s ; \text{id} \equiv s$
 - $\text{id} \<+ s \equiv \text{id}$
 - $s \<+ \text{id} \not\equiv s$
 - $s_1 \< \text{id} + s_2 \equiv s_1 \<+ s_2$

Failure

- Syntax: fail
- Always fail
- Some laws
 - $\text{fail} \<+ s \equiv s$
 - $s \<+ \text{fail} \equiv s$
 - $\text{fail} ; s \equiv \text{fail}$
 - $s ; \text{fail} \not\equiv \text{fail}$

Negation (sugar)

- Syntax: $\text{not}(s)$
- Fail if s succeeds, succeed if s fails
- Equivalent to: $s \< \text{fail} + \text{id}$

Parameterized and recursive definitions

- Syntax: $f(x_1, \dots, x_n \mid y_1, \dots, y_m) = s$
- Strategy definition parameterized with strategies (x_1, \dots, x_n) and terms (y_1, \dots, y_m)
- Note: definitions may be recursive

Parameterized and recursive definitions

- Syntax: $f(x_1, \dots, x_n \mid y_1, \dots, y_m) = s$
- Strategy definition parameterized with strategies (x_1, \dots, x_n) and terms (y_1, \dots, y_m)
- Note: definitions may be recursive

```
try(s)          = s <+ id  
  
repeat(s)      = try(s; repeat(s))  
  
while(c, s)    = if c then s; while(c,s) end  
  
do-while(s, c) = s; if c then do-while(s, c) end
```

Traversal Strategies

Term Rewriting

- apply set of rewrite rules exhaustively

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules (operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances

Term Rewriting for Program Transformation

```
signature
  sorts Prop
  constructors
    False : Prop
    True : Prop
    Atom : String -> Prop
    Not : Prop -> Prop
    And : Prop * Prop -> Prop
    Or : Prop * Prop -> Prop
  rules
    DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
    DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
    DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
    DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
    DN : Not(Not(x)) -> x
    DMA : Not(And(x, y)) -> Or(Not(x), Not(y))
    DMO : Not(Or(x, y)) -> And(Not(x), Not(y))
```

Term Rewriting for Program Transformation

```
signature
  sorts Prop
  constructors
    False : Prop
    True : Prop
    Atom : String -> Prop
    Not : Prop -> Prop
    And : Prop * Prop -> Prop
    Or : Prop * Prop -> Prop
  rules
    DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
    DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
    DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
    DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
    DN : Not(Not(x)) -> x
    DMA : Not(And(x, y)) -> Or(Not(x), Not(y))
    DMO : Not(Or(x, y)) -> And(Not(x), Not(y))
```

This is a non-terminating rewrite system

Encoding Control with Recursive Rewrite Rules

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a ‘pure’ rewrite rule

$$p_1 \rightarrow p_2$$

with a functionalized rewrite rule:

$$f : p_1 \rightarrow p'_2$$

applying f recursively in the right-hand side

- Normalize terms $f(t)$ with respect to these rules
- The function now controls where rules are applied

Recursive Rewrite Rules: Disjunctive Normal Form

```
dnf  : True      -> True
dnf  : False     -> False
dnf  : Atom(x)   -> Atom(x)
dnf  : Not(x)    -> <not>(<dnf>x)
dnf  : And(x,y)  -> <and>(<dnf>x,<dnf>y)
dnf  : Or(x,y)   -> Or(<dnf>x,<dnf>y)
```

```
and1 : (Or(x,y),z) -> Or(<and>(x,z),<and>(y,z))
and2 : (z,Or(x,y)) -> Or(<and>(z,x),<and>(z,y))
and3 : (x,y)        -> And(x,y)
and  = and1 <+ and2 <+ and3
```

```
not1 : Not(x)    -> x
not2 : And(x,y)  -> Or(<not>(x),<not>(y))
not3 : Or(x,y)   -> <and>(<not>(x),<not>(y))
not4 : x          -> Not(x)
not  = not1 <+ not2 <+ not3 <+ not4
```

Functional encoding has two main problems

Overhead due to explicit specification of traversal

- A traversal rule needs to be defined for each constructor in the signature and for each transformation.

Separation of rules and strategy is lost

- Rules and strategy are completely *intertwined*
- Intertwining makes it more difficult to *understand* the transformation
- Intertwining makes it impossible to *reuse* the rules in a different transformation.

Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

language	# constructors
Tiger	65
C	140
Java 5	325
COBOL	300–1200

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

Programmable Rewriting Strategies

- Select rules to be applied in specific transformation
- Select strategy to control their application
- Define your own strategy if necessary
- Combine strategies

Idioms

- Cascading transformations
- One-pass traversal
- Staged transformation
- Local transformation

Strategic Idioms

Rules for rewriting proposition formulae

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom   : String -> Prop
    Not    : Prop -> Prop
    And    : Prop * Prop -> Prop
    Or     : Prop * Prop -> Prop
rules
  DAOL   : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR   : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL   : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR   : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN     : Not(Not(x))      -> x
  DMA   : Not(And(x, y))    -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

Strategic Idioms: Cascading Transformation

Cascading Transformations

- Apply small, independent transformations in combination
- Accumulative effect of small rewrites

```
simplify = innermost(R1 <+ ... <+ Rn)
```

disjunctive normal form

```
dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
```

conjunctive normal form

```
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

Strategic Idioms: One-Pass Traversal

One-pass Traversal

- Apply rules in a single traversal over a program tree

```
simplify1 = downup(repeat(R1 <+ ... <+ Rn))
simplify2 = bottomup(repeat(R1 <+ ... <+ Rn))
```

constant folding

```
Eval : And(True, e) -> e
Eval : And(False, e) -> False
Eval : ...
```

```
eval = bottomup(try(Eval))
```

Strategic Idioms: One-Pass Traversal

Example: Desugarings

```
DefN  : Not(x)      -> Impl(x, False)
DefI  : Impl(x, y)  -> Or(Not(x), y)
DefE  : Eq(x, y)    -> And(Impl(x, y), Impl(y, x))
Def01 : Or(x, y)    -> Impl(Not(x), y)
Def02 : Or(x, y)    -> Not(And(Not(x), Not(y)))
DefA1 : And(x, y)   -> Not(Or(Not(x), Not(y)))
DefA2 : And(x, y)   -> Not(Impl(x, Not(y)))
IDefI : Or(Not(x), y) -> Impl(x, y)
IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)
```

```
desugar = topdown(try(DefI <+ DefE))
```

```
impl-nf  = topdown(repeat(DefN <+ DefA2 <+ Def01 <+ DefE))
```

Staged Transformation

- Transformations are not applied to a subject term all at once, but rather in stages
- In each stage, only rules from some particular subset of the entire set of available rules are applied.

```
simplify =  
  innermost(A1 <+ ... <+ Ak)  
  ; innermost(B1 <+ ... <+ Bl)  
  ; ...  
  ; innermost(C1 <+ ... <+ Cm)
```

Local transformation

- Apply rules only to selected parts of the subject program

```
transformation =  
  alltd(  
    trigger-transformation  
    ; innermost(A1 <+ ... <+ An)  
  )
```

Except where otherwise noted, this work is licensed under

