Growing a Syntax

Eric Allen
Sun Microsystems
eric.allen@sun.com

Ryan Culpepper *
Northeastern University
ryanc@ccs.neu.edu

Janus Dam Nielsen * †
Aarhus University
jdn@brics.dk

Jon Rafkind *
University of Utah
rafkind@cs.utah.edu

Sukyoung Ryu
Sun Microsystems
sukyoung.ryu@sun.com

Abstract

In this paper we present a macro system for the Fortress programming language. Fortress is a new programming language designed for scientific and high-performance computing. Features include: implicit parallelism, transactions, and concrete syntax that emulates mathematical notation.

Fortress is intended to grow over time to accommodate the changing needs of its users. Our goal is to design and implement a macro system that allows for such growth. The main challenges are (1) to support extensions to a core syntax rich enough to emulate mathematical notation, (2) to support combinations of extensions from separately compiled macros, and (3) to allow new syntax that is indistinguishable from core language constructs. To emulate mathematical notation, Fortress syntax is specified as a parsing expression grammar (PEG), supporting unlimited lookahead. Macro definitions must be checked for well-formedness before they are expanded and macro uses must be well encapsulated (hygienic, composable, respecting referential transparency). Use sites must be parsed along with the rest of the program and expanded directly into abstract syntax trees. Syntax errors at use sites of a macro must refer to the unexpanded program at use sites, never to definition sites. Moreover, to allow for many common and important uses of macros, mutually recursive definitions should be supported.

Our design meets these challenges. The result is a flexible system that allows us not only to support new language extensions, but also to move many constructs of the core language into libraries. New grammar productions are tightly integrated with the Fortress parser, and use sites expand into core abstract syntax trees. Our implementation is integrated into the open-source Fortress reference interpreter.To our knowledge, ours is the first implementation of a modular hygienic macro system based on parsing expression grammars.

* Work partially done while visiting Sun Microsystems Laboratories.

† Supported by the SIMAP Project under the Danish Strategic Research Council NABIIT-program.

FOOL '09 24 January, 2009, Savannah, Georgia, USA.
ACM .

1. Introduction

A programming language can be thought of as a vocabulary of words and a set of rules that define how to combine words into meaningful constructs [19]. One goal of language design is to create a vocabulary and a set of rules that allow the programmer to express ideas clearly and concisely. But the set of concepts needed over the lifetime of a programming language is difficult to anticipate and is often dependent on the development of new systems that programs must interact with (for example, new domain-specific languages, new programming platforms, new virtual machines, etc.).

The Fortress programming language is intended to grow over time to accommodate the changing needs of its users [4]. One mechanism to allow for such growth is syntactic abstraction: It is possible to add new syntactic constructs to the language in libraries, defining new constructs in terms of old ones. In this manner, the language can gracefully adapt to unanticipated needs as they become apparent. Parsing of new constructs can be done alongside parsing of primitive constructs, allowing programmers to detect syntax errors in use sites of new constructs early. Programs in domain-specific languages can be embedded in Fortress programs and parsed along with their host programs. Moreover, the definition of many constructs that are traditionally defined as core language primitives (e.g., for loops) can be moved into Fortress' own libraries, thereby reducing the size of the core language.

Designing such a syntactic abstraction mechanism for Fortress is hard. In part, this difficulty is due to our design goal of growability: New syntax should be indistinguishable from old syntax from the user's perspective. This requirement imposes several constraints on Fortress macros: Macro definitions must be checked for well-formedness before they are expanded; otherwise syntax errors in the definition of a macro might not be exposed until the macro is used. Macro uses must be well encapsulated (hygienic, composable, respecting referential transparency); otherwise it would be impossible to safely use a macro without understanding the innards of its implementation. Use sites must be parsed along with the rest of the program and expanded directly into abstract syntax trees; otherwise new syntax would be distinguishable from old in the sense that use site errors of new syntax are not signaled alongside those of old syntax. Similarly, syntax errors at use sites of a macro must refer to the unexpanded program at use sites, never to definition sites. And because many desirable macro definitions require recursion, Fortress macros must support recursive and mutally recursive definitions.

Difficulties also arise from another design goal of Fortress: The concrete syntax of Fortress is designed to emulate mathematical

```
g1 = ⟨ 1, 2, 3, 4, 5 ⟩
g2 = ⟨ 6, 7, 8, 9, 10 ⟩
for i ← g1, j ← g2 do
    println "(" i ", " j ")"
end
```

Figure 1. A parallel for-loop in Fortress

```
g1.loop(fn i ⇒
        g2.loop(fn j ⇒
                println "(" i ", " j ")" ))
```

Figure 2. A desugared version of the for-loop in Figure 1

notation as closely as possible. For example, operator precedence in Fortress is not transitive. Moreover, juxtaposition itself is treated as a mathematical operator, whose meaning is overloaded based on the types of the arguments. Juxtaposition of numeric variables denotes multiplication, as in the expression:

$$a(m + n)$$

Juxtaposition of a function with an argument denotes function application, as in the expression:

$$\sin x$$

It has been the experience of the Fortress design team that defining a grammar with these properties is most naturally done in the context of a formalism supporting unlimited lookahead. Fortress syntax is defined in the formalism of Parsing Expression Grammars (PEGs), which support unlimited lookahead, are unambiguous and are closed under union [16]. Consequently, any macro system for Fortress must work in the context of a core syntax defined as a PEG. To make the macro system as expressive as possible, we also allow new syntax to be defined via arbitrary PEGs. Thus, combination of new syntax with old involves composition of PEGs, and rules for resolving conflicts between separately defined syntax must be provided.

In this paper, we present a macro system for Fortress that meets these design challenges. Our system works in the context of a core syntax based on mathematical notation, and new syntax is indistinguishable from old. Indeed, we have found that constructs currently defined as part of Fortress core syntax can be moved to macro definitions in libraries. One such example is the Fortress `for` loop. In Figure 1, we define two variables g_1 and g_2, each referring to a list. Next, the `for` loop introduces the variables i and j which iterate through the two lists, and all pairs of values in the cross product of the two lists are printed. Our system allows `for` loops such as this to be transformed into method calls and anonymous functions as in Figure 2.

2. Contributions and Outline

The main contributions of this paper are:

- Design of a hygienic macro system in the context of a core language based on parsing expression grammars
- A design for composing, and resolving conflicts among, separately compiled macros in the context of parsing expression grammars
- Explanation of the low-level mechanics of a working implementation of our system, available as open source in the Fortress reference implementation
- Definition of a core calculus for the Fortress macro system

The rest of the paper is organized as follows. Section 3 describes the design objectives of our system in detail. Section 4 describes how new syntactic abstractions can be described in the presented system through an example. Section 5 presents a formal treatment of the system semantics. In Section 6 the implementation of the system is presented and discussed. In Section 7 the macro system is evaluated by discussing various syntax extensions implemented in the system. A description of related work is given in Section 8, and we conclude in Section 9 along with discussing future work.

3. A Growable Language

A growable language is one of the key ideas of Fortress. The Fortress syntactic abstraction mechanism is designed to support the language growth with the goals described in this section in mind. Throughout this paper, we use the term *macro* to refer to a particular language extension.

New syntax indistinguishable from the core syntax It would be desirable if a macro syntax is indistinguishable from the core language syntax so that the uses of the macro do not introduce any visual clutter to the language. We could have chosen to enclose all macro uses in special brackets which would have made the parsing and recognition of macros easier. However, the result would have been a language in which extensions stand out conspicuously from core syntax, reminiscent of user-defined function definitions in APL [21]. 'Instead, we aim to design an extensible language with no apparent differences between the core syntax and macro uses, such as Scheme [2] and Lisp [23]. However, unlike the S-expressions of Scheme and Lisp, the Fortress core syntax emulates mathematical notation, which poses additional challenges in achieving this goal.

Composition of independent macros Composability of independent macros is especially important to support modular development of a system. Contributions from different development teams must be self-contained and must not interfere with each other. For example, suppose that one contribution provides a grammar with the following macro:

$$\text{Expr} \mid:= \ macro_1\ e : \text{Expr} \ \Rightarrow \langle [e] \rangle$$

which extends ($\mid:=$) an existing nonterminal Expr representing an expression with a new syntax "$macro_1\ e$" where e is an expression, and translates ($\Rightarrow$) the new syntax to e. Now suppose that another contribution provides a grammar with the following macro:

$$\text{Expr} \mid:= \ macro_2\ e : \text{Expr} \ \Rightarrow \langle [e^2] \rangle$$

which extends Expr with "$macro_2\ e$" where e is an expression, and translates it to e^2. Then a grammar extending both grammars can use both macros as expressions, even in combination with each other:

$$macro_1(macro_2\ \ 7)$$
$$macro_2(macro_1\ \ 7)$$

Expansion into other macros Macros may be defined in terms of other helper macros. For example, the following macro:

$$\text{Expr} \mid:= \ macro_3\ e : \text{Expr} \ \Rightarrow \langle [macro_4\ e] \rangle$$

is defined in terms of the following helper macro:

$$\text{Expr} \mid:= \ macro_4\ e : \text{Expr} \ \Rightarrow \langle [e] \rangle$$

Recursion and case dispatch Many interesting macros can be nicely written using recursive applications of defined macros. The capability of recursively defining macros (including mutual recursion) and dynamically dispatching on macros improves expressiveness as we discuss in Section 8. For example, the following macro:

```
grammar ForLoop extends { Expression, Identifier }
 Expr |:=
   for { i : Id ← e : Expr , ? Space }∗ do block : Expr end ⇒
   ◁[ for2 i ∗∗; e ∗∗; do block; end ▷
 | for2 i : Id∗; e : Expr∗; do block : Expr; end ⇒
  case i of
    Empty ⇒
      ◁[ block ▷
    Cons(ia, ib) ⇒
      case e of
        Empty ⇒◁[ throw Unreachable ▷
        Cons(ea, eb) ⇒
          ◁[ ((ea).loop(fn ia ⇒ (for2 ib ∗∗; eb ∗∗;
                                 do block; end)))▷
    end
  end
end
```

Figure 3. A grammar definition of a simplified syntax of `for` loops in Fortress

```
Expr|:= macro5 i : Id∗ do block : Expr end ⇒
        case i of
            Empty ⇒ ◁[ block ▷
            Cons(first, rest) ⇒
                ◁[ do println first
                      macro5 rest do block end
                   end ▷
        end
```

defines a new syntax "$macro_5\ i$ `do` *block* `end`" where i is a sequence of identifiers (Id$*$) and *block* is an expression, and its translation depends on the actual value bound to i. If i is an empty sequence (Empty), the input sequence matched by the new syntax is translated to *block*. Otherwise, it prints the first identifier in i (*first*) and recursively applies the new syntax to the remaining identifiers in i (*rest*).

Similar syntax for definition and use of a macro If the syntax for defining a macro resembles the syntax for using the macro, it would be easy for a programmer to learn how to use the macro just by looking at its definition. Key features to enable this similarity between the definition site and use site of a macro include:

- Tokens appear in a macro definition where they appear in a use site of the macro, in contrast to other syntax specification languages [22, 28] where the terminal tokens are specified in a separate section.
- Whitespace characters between tokens are interpreted as optional Fortress whitespace characters, making the specification of a macro less bloated compared to using another character sequence to represent the specification of whitespace.

4. Syntactic Abstraction by Example

The Fortress syntactic abstraction mechanism provides a way to extend the core language syntax. Before formally describing the syntactic abstraction mechanism, we first provide some examples of how to define and use macros. Figure 3 presents a definition of a simplified syntax of `for` loops in Fortress.

A programmer can define a new syntax with a grammar definition. A grammar is a self-contained language extension which can be composed with other grammars in a modular way. An example grammar ForLoop is defined in Figure 3. A grammar contains a set of *nonterminal* definitions and extensions, and may extend any number of other grammars, since it often makes sense to use macros from different grammars to implement new macros, or to create a new language extension by composing different languages. A new nonterminal is defined as:

NewNonterminal ::= ...

and an existing nonterminal is extended as:

ExistingNonterminal |:= ...

The available nonterminals in an extended grammar may be used as a basis for a nonterminal definition or extension. The ForLoop grammar extends an existing nonterminal Expr which defines all expression constructs in Fortress. It inherits the nonterminals provided by the extended grammars Expression and Identifier so that it can use the nonterminals in it. The Expr and Id nonterminals are defined in the Expression and Identifier grammars respectively and thus are available in ForLoop. We present the precise definition of nonterminal availability in Section 5.1.2.

A nonterminal definition or extension has an "ordered sequence" of alternative *variants*. For example, ForLoop extends Expr with two variants: the `for` loop variant and its helper for_2. A variant consists of a *pattern*, followed by $\Rightarrow$, and a *transformation expression*. A transformation expression is either an expression enclosed by $\lhd$ and $\rhd$, or a case expression with branches, whose right-hand sides are transformation expressions.

A pattern is defined by a sequence of parts. The pattern language is based on Parsing Expression Grammars (PEGs) and is described in Section 5. A part is either a simple part or operations over parts. Different kinds of parts make up the pattern for the `for` loop variant:

`for` { i : Id $\leftarrow$ e : Expr , ? Space }$*$ `do` *block* : Expr `end`

The first symbol is the terminal `for`, followed by a symbol group, another terminal `do`, a nonterminal reference "*block* : Expr", and then another terminal `end`. A terminal is any sequence of Unicode [33] characters that is not the name of a nonterminal available in the containing grammar. A symbol group is a sequence of symbols inside curly braces. A symbol or a symbol group may be followed by an option operator ? or a repetition operator $+$ (one or more repetition) or $*$ (zero or more repetition). A nonterminal reference such as "*block* : Expr" is a reference to any nonterminal available in the containing grammar optionally prepended by a label followed by ":". The whitespace between symbols represents optional Fortress whitespace. When a space is required, the nonterminal Space specifies the required space. The pattern language supports unlimited lookahead in the same way as PEGs, by means of the semantic predicates: $\wedge$ (*must* match) and $\neg$ (must *not* match), both of which do not consume input.

A transformation expression gives meaning to the macro by providing a translation of the pattern. A translation is expressed either as an expression enclosed by $\lhd$ and $\rhd$, or a case expression over a pattern variable. The transformation expression of the `for` loop variant:

$\lhd$[$for_2\ i \ast\ast; e \ast\ast;$ `do` *block*; `end` $\rhd$

uses the syntax-unfold operator, $\ast\ast$, to expand the lists of identifiers and expressions as arguments to the for_2 variant. The syntax-unfold operator is similar to the ellipsis operator [29] found in Scheme [2], and expands into the syntax matched by its arguments. The for_2 variant is translated into core Fortress syntax by recursively traversing the identifier and expression lists. The transformation expression uses two nested case dispatches on the list structure of the pattern variables i and e. The actual value bound to a pattern variable is compared to the left-hand side of each case clause: an empty list is matched to Empty and a nonempty list is matched to Cons. If a list is nonempty, the first argument of Cons contains

```
g1.loop(fn i ⇒
        for j ← g2 do
            println "(" i ", " j ")"
        end)
```

Figure 4. One unrolling of the for loop from Figure 1

the head of the list and the second argument contains the rest of the list. For each pair of head elements from an identifier list i and an expression list e, we call the *loop* method on the head of the expression list, ea, with an anonymous function whose argument is the head of the identifier list, as the input to the *loop* method. The two lists of identifiers and expressions are guaranteed to have the same length by the structure of the `for` variant.

Although the transformation is expressed in terms of Fortress concrete syntax, the act of transforming a use site at parse time is not performed on the concrete syntax. Instead, the transformation expression is parsed into Fortress Abstract Syntax Tree (AST) nodes, containing *placeholder* nodes to represent pattern variables, and the act of transforming a use site is performed by substituting AST nodes in for the placeholders. An alternative approach would have been a multi-staged system in which transformation expressions consist of explicit user-defined computations of resulting AST nodes. Although such an approach is more flexible, it is also tedious and error prone. Moreover, it is difficult to check that syntax trees constructed through explicit computation are well-formed. Earlier work has shown that although a template-based approach is not as flexible as an approach based on explicit computation, it is usually possible to express the transformations desired in practice [5, 8].

The `for` loop example in Figure 3 shows that our syntactic abstraction mechanism satisfies the five goals described in Section 3:

- New syntax indistinguishable from the core syntax: As Figure 1 shows, the use of the `for` macro is indistinguishable from other Fortress language constructs.
- Composition of independent macros: The `for` macro extends the grammars, Expression and Identifier, so that it can use the nonterminals, Expr and Id, defined in them.
- Expansion into other macros: The `for` macro is defined in terms of the for_2 macro.
- Recursion and case dispatch: The for_2 macro is recursively defined using two nested case dispatches. Using the for_2 macro, the `for` loop in Figure 1 can be rewritten as in Figure 4. One more application of the macro will desugar the `for` loop as in Figure 2.
- Similar syntax for definition and use of a macro: The syntax for defining the for_2 macro:

  $for_2\ i : \mathrm{Id}*; e : \mathrm{Expr}*;$ `do` $block : \mathrm{Expr};$ `end`

  and the use of it:

  $for_2\ i * *; e * *;$ `do` $block;$ `end`

  is very similar because tokens appear in the same place in both definition and use sites of the macro and implicit whitespace specification avoids visual clutter.

5. Syntax Normalization

In this section, we describe how a Fortress source program using macros is turned into a Fortress AST without macros which can be interpreted by the Fortress interpreter. For presentation brevity, we focus on a core subset of Fortress, *Core Fortress*. The abstract syntax of Core Fortress is presented in Figure 5.

Program	::=	*Grammar** *Expr*	
Grammar	::=	`grammar` *GrammarName* `extends` { *GrammarName** } (*Definition* | *Extension*)* `end`	
Definition	::=	*NTName* ::= *Variant**	
Extension	::=	*NTName* |:= *Variant**	
Variant	::=	*Pattern* ⇒ *Action*	
Pattern	::=	*Part**	
Part	::=	*SimplePart* ?	
	|	*SimplePart* $*$	
	|	*SimplePart* $+$	
	|	*SimplePart*	
	|	¬ *Part*	
	|	∧ *Part*	
SimplePart	::=	*PatternVar* : *BasePart*	
	|	*BasePart*	
BasePart	::=	*NTName*	
	|	*Terminal*	
	|	[*Char*:*Char*]	
Action	::=	⧼[*Term*]⧽	template
	|	`case` *PatternVar* `of` Empty ⇒ *Action* Cons(*PatternVar*, *PatternVar*) ⇒ *Action* `end`	
Expr	::=	n	number
	|	s	string
	|	x	variable
	|	`fn` x ⇒ *Expr*	function expression
	|	*Expr Expr*	function application
	|	(*Expr*)	parenthesized
Term	::=	*PatternVar*	
	|	*Term* $**$	
	|	n	
	|	s	
	|	x	
	|	`fn` x ⇒ *Term*	
	|	*Term Term*	
	|	(*Term*)	

Figure 5. Abstract syntax for Core Fortress

A Core Fortress program consists of a possibly empty sequence of grammar definitions followed by an expression, called the *main expression*. Grammar definitions introduce new syntax to Core Fortress. A grammar definition includes a sequence of nonterminal definitions or extensions and it may extend other grammars to use nonterminals defined in the extended grammars in its definition. A nonterminal definition or extension has an ordered sequence of alternative variants and a variant consists of a pattern and a transformation expression (*Action* in Figure 5). A transformation expression is either a term enclosed by ⧼[and]⧽, called a *template*, or a `case` expression over a pattern variable. The nonterminals *GrammarName*, *NTName*, *PatternVar*, *Terminal*, and *Char* range over Unicode strings. An expression is a number, string, variable, function expression, function application, or parenthesized expression. A term is a pattern variable, an ellipses term, number, string, variable, function term, function application term, or parenthesized term. The metavariables n, s, and x range over numbers, strings, and variables, respectively.

Because grammar definitions introduce new syntax to the language, the program itself defines how to parse it and turn it into an AST. We call the entire process from parsing a source program to creating a corresponding AST *syntax normalization*. Syntax normalization consists of two stages: *parsing* and *transformation*. In the parsing stage, the source program represented as a Unicode string is turned into what we call a *parsed* program. Then, in the transformation stage, the parsed program is transformed to a program in Core Fortress AST. Each stage is described in detail in the subsequent subsections.

5.1 Parsing

The parsing stage transforms a source program in a Unicode string into a parsed program in an AST representation which describes how the program is to be transformed into an executable Core Fortress program. The transformation is nontrivial because a program itself defines how it is parsed: the grammar definitions in the program determines which syntax may occur in the grammar definitions and the main expression.

In order to resolve this self-dependency problem, we take a two-step approach to parse Fortress programs. In the first step, we parse all the grammars except for the action part of each variant and the main expression. The action parts and the main expression are just parsed as Unicode strings. This step is a standard parsing which generates an AST where expressions are represented as strings. In the second step, we parse each action part and the main expression. In order to parse the action parts, we need to compute what we call the set of *available macros*. In Section 5.1.2, we show how to compute the set of available macros for a given grammar, and how to derive a PEG from which a PackRat parser can be generated (for example by using Rats! [17, 18]) and then used to parse the action parts. The action parts and the main expression are parsed into *node expressions*. A node expression describes how macros are invoked to turn a macro syntax into a Core Fortress syntax. The construction of node expressions is described in Section 5.1.3. This two-step approach works because the grammar definitions appear before the main expression. The approach also has the advantage that we break the dependency in parsing recursive macros.

The parsing stage relies on *Parsing Expression Grammars* (PEGs) [16] because they have some advantages over usual Context-Free Grammars (CFG) [10] based parsing formalisms such as LL and LALR(k). PEGs are unambiguous, are closed under union, and integrates lexing with parsing. We need the closure property because we want to combine PEGs for different grammars and the integrated lexing and parsing makes it easy to achieve our goal of "similar syntax for definition and use". Furthermore, the parsers based on PEGs allow a linear execution time compared to the generalized CFG parsers. We briefly introduce PEGs in Section 5.1.1 along with a description of our pattern language which is effectively a variant of *Parsing Expressions*.

5.1.1 Parsing Expression Grammars

A Parsing Expression Grammar (PEG) is a 3-tuple (N, T, s), where N is a finite set of nonterminal definitions, T is a finite set of terminal symbols, $s \in N$ is the start nonterminal definition. A nonterminal definition is a pair (n, cs) where n is a name and *cs* is a list of prioritized alternatives, each alternative being a *parsing expression* [16]. The terminal symbols and the nonterminal names are disjoint. If e, e_1, and e_2 are parsing expressions then so are the empty string ϵ, a terminal symbol $a \in T$, a nonterminal name n, a sequence $e_1\ e_2$, an optional expression $e?$, zero-or-more repetitions e^*, a not-predicate $\neg e$, and a and-predicate $\wedge e$.

The main difference between PEGs and CFGs is the lack of ambiguity in PEGs resulting from the prioritized alternatives. In a CFG, the two nonterminal definitions $A = a \mid ab$ and $A = ab \mid a$ are equivalent. However, in a PEG, the second alternative of the former would never succeed because the first alternative is always taken if the input string to be recognized starts with a.

The pattern language shown in Figure 5 is based on parsing expressions with some differences. If $Part_i$ $(1 \leq i \leq n)$ of a sequence of parts, $(Part_1, Part_2, \ldots, Part_n)$, corresponds to a parsing expression e_i, then the sequence of parts corresponds to the sequence of parsing expressions with optional Fortress whitespace in between: $(e_1, w, e_2, w, \ldots, w, e_n)$. Here w refers to the nonterminal defining the optional whitespace in Fortress. This small difference reduces the visual clutter when writing language extensions. We also allow character classes to be specified(e.g. the character class of ";" and the lowercase letters from "a" to "z" $[;a{:}z]$). Character classes can be trivially, but tediously, encoded in parsing expressions by explicit enumeration of the alternatives.

```
grammar A
  Nt::=
    macroA ⇒ ...
end
grammar B extends {A}
  Nt|:=
    macroB ⇒ ◁[... macroA ...]▷
end
grammar C extends {A}
  Nt|:=
    macroC ⇒ ...
end
```

Figure 6. Grammars including multiple nonterminal extensions

```
grammar D extends {B}
  Nt|:=
      macroD ⇒ ◁[... macroB ...]▷
    | B.Nt

(* This definition is illegal,
   because macroA is not propagated by the grammar B:
  Nt|:=
      macroD2 ⇒ ◁[... macroB ... macroA ...]▷
*)
end
```

Figure 7. Grammars should behave as usual module systems

5.1.2 Combining Grammars

In order to create a parser and parse the action parts in a given grammar, we need to compute the set of available macros. The set of available macro definitions for a grammar consists of the macros defined by the nonterminal definitions and extensions in the grammar and inherited from the extended grammars. However, the multiple "inheritance" of grammars in combination with the prioritized alternatives of PEGs makes it nontrivial to compose nonterminal extensions.

For example, Figure 6 shows three grammars A, B, and C, each of which defines a single macro. We write $\triangleleft[\ldots macroA \ldots]\triangleright$ to indicate a template where the macro $macroA$ occurs somewhere. Grammar B and C extend grammar A which contains the definition of the nonterminal Nt.

The interesting question is what the semantics of a nonterminal extension should be. It seems natural to expect that B is able to use all the macros defined in A since it extends it. However, what

```
grammar E extends {B, C}
  Nt|:=
      macroE ⇒ ⟨[... macroB ... macroC ...]⟩
end
```

Figure 8. Multiple extensions of a single nonterminal

if a grammar such as D in Figure 7 extends B? Because D does not extend A directly, we do not allow $macroA$ to be available in the grammar D as the comment enclosed by "(*" and "*)" in Figure 7 shows. This behavior is consistent with the Fortress component system where a component does not reexport all of its imports by default. A grammar may explicitly propagate an inherited macro by referring to the nonterminal extension defining the macro. The second alternative of the nonterminal extension in D gives an example of explicit propagation of an inherited macro: $B.\mathrm{Nt}$. Whenever $macroD$ is available, $macroB$ is also available.

Grammar B overrides the definition of the Nt nonterminal in grammar A, but the override does not affect grammar C which also extends A. It is only possible to completely override a nonterminal if all uses of the grammar containing the overridden nonterminal is changed to the grammar containing the overriding nonterminal. In the example above C would have to extend B instead, and there is no guarantee that another grammar which extends A could be created.

A grammar may extend multiple grammars. Multiple extension induces a lattice structure in the extension relation. One key issue with multiple extension is how to combine grammars when there is a "diamond" shape in the lattice as shown in Figure 8. The grammar E extends B and C both of which extend the same grammar A but define different extensions for the same nonterminal Nt. One approach would be to collect the macros along the paths from B and C to A following the extension relation, and then concatenate them:

```
Nt|:=
    macroE ⇒ ⟨[... macroB ... macroC ...]⟩
  | macroB ⇒ ⟨[... macroA ...]⟩
  | macroA ⇒ ...
  | macroC ⇒ ...
  | macroA ⇒ ...
```

The order of $macroB$ and $macroC$ could be reversed, but it has the same problem as this approach: the syntax defined in A might shadow $macroC$ because the variants are ordered. It is an unfortunate behavior because we intend to support a user-defined syntax on top of the existing Fortress syntax. Both $macroB$ and $macroC$ should have higher priority than $macroA$.

Our solution to the shadowing problem is to collect the macros along the paths till the lowest common ancestor of the immediately extended grammars in the extension relation and concatenate them, and then recursively collect and concatenate the macros along the paths to the end of the extension relation. For example, we resolve Nt in E as follows:

```
Nt|:=
    macroE ⇒ ⟨[... macroB ... macroC ...]⟩
  | macroB ⇒ ⟨[... macroA ...]⟩
  | macroC ⇒ ...
  | macroA ⇒ ...
```

One thing to note is that the ordering of macros must be specified. The ordering should be deterministic and preferably the grammar writer should be able to control it. Therefore, we have chosen to use the order in which the extended grammars appear in the `extends` clause.

```
NodeExpr ::= PatternVar
           | case PatternVar of
               Empty ⇒ NodeExpr
               Cons(PatternVar, PatternVar) ⇒ NodeExpr
             end
           | Transformer (NodeExpr...)
           | Ellipses (NodeExpr, NodeExpr...)
           | NodeConstructor(NodeExpr...)
```

Figure 9. Abstract syntax for node expressions

Using the solution, we build a PEG from which a parser can be generated and used to parse the action parts and the main expression. The PEG for a grammar is:

- formed from all definitions from all transitively-extended grammars and the current grammar,
- modified by the extensions from the current grammar, if any, and
- modified by the implicit extensions for all nonterminals in the inherited extensions that are not extended in the current grammar.

The first step of constructing the PEG is to gather all of the nonterminal definitions from all the transitively-extended grammars. A variant in a nonterminal definition is either a pair of a syntax pattern and a transformer or a reference to an extended grammar's extension (propagation of an extension). For the former, the pair is simply added to the PEG. For the latter, we retrieve the extensions added by that grammar, process them, and add them to the PEG. The next step is to apply the extensions. The extensions consist of the explicit extensions (defined in the grammar) and the implicit extensions inherited from the extended grammars. The PEG for parsing the main expression is obtained in the same way as for a grammar which extends all the preceding grammars, but containing no definitions nor explicit extensions.

5.1.3 Construction of Node Expressions

Using the parser generated by a PEG as described in Section 5.1.2, the action parts in the grammars and the main expression are parsed into node expressions. A node expression is a representation of how macro invocations are applied to obtain a Core Fortress AST. The abstract syntax of node expressions is presented in Figure 9 where a sequence of *NodeExpr* is represented as $\overline{NodeExpr}$. A node expression is a pattern variable, a case dispatch, a macro invocation represented as a reference to a *Transformer* applied to a number of node expression arguments, an ellipses node represented as a reference to an Ellipses applied to a node expression (the first argument), or a Fortress expression represented as a reference to a *NodeConstructor* applied to a number of node expression arguments. We use *Transformer* and *NodeConstructor* to range over templates and Core Fortress expressions, respectively. The second argument of the Ellipses node is initialized with its first argument and changes during the evaluation of the node.

A node constructor is created when an input string is matched by the Core Fortress syntax. A transformer is created whenever a macro use is parsed by matching the macro definition against the macro use. A macro definition is matched in the same way as a parsing expression is matched against an input string: each part is matched in the order it appears. If the macro definition contains any references to other nonterminals, then we parse with respect to the nonterminals and if they succeed the result is a node expression, and we continue parsing the rest of the macro definition. If all the parts in a macro definition are matched, then a transformer

is created with the node expressions from any nonterminals as arguments.

Consider an example from Section 3, where $macro_1$ is applied to the literal 7:

$$macro_1 \;\; 7$$

and parsed against the macro definition:

$$\text{Expr} \;|:=\; macro_1\, e : \text{Expr} \;\Rightarrow\; \lhd[e]\rhd$$

First, we see that there are two parts in the macro definition: $macro_1$ and "$e : Expr$". The first part is a terminal part which must be matched verbatim in the input sequence and the other part is a binding part where the pattern variable e gets bound to the result of parsing $Expr$. Parsing the action part of $macro_1$ returns the node expression consisting of a pattern variable reference and is transformed to a *Transformer* T_1, where T_1 is a fresh name. Then, parsing the macro use requires parsing 7 against the *Expr* nonterminal, which is matched by the Core Fortress syntax and the result is a node constructor for the Core Fortress abstract syntax 7. We bind e to 7 in the environment and pass 7 as an argument to the transformer T_1, resulting in the node expression $T_1(7)$ which can be passed on to the transformation phase described in Section 5.2.

5.2 Transformation

Transformation is the evaluation of node expressions to construct a Core Fortress AST without macros which can be interpreted by the Fortress interpreter. Given a parsed program which contains a set of grammars and a main expression, we transform it to an executable Core Fortress program by evaluating the node expression first in the main expression and consequently in transformers.

We define the evaluation of node expressions in terms of a small-step operational semantics [26]. We define the semantics using a transition system. The transition system consists of a set of configurations $\sigma \in \Sigma$, a set of terminal configurations $\mathrm{T} \subseteq \Sigma$, and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. A configuration is a node expression along with two environments: $\Upsilon, \Gamma \vdash NodeExpr$. The transformer environment Υ maps transformer names to transformer definitions and the node environment Γ maps pattern variables to node expressions. We write "$\Gamma\left[\overline{v} \mapsto \overline{n'}\right]$" to denote an extension of Γ with the new bindings "$\overline{v} \mapsto \overline{n'}$". The transformer environment is initialized to contain all the transformers from all the grammars and does not change during evaluation. This is possible because the parsing has already resolved the scope of the transformers introduced by the grammars. The node environment is initially empty. A value is a node constructor application, the arguments of which are also values.

The evaluation of a node expression is inductively defined by the transition relations in Figure 10. The metavariables t ranges over transformer names, v ranges over pattern variables, n ranges over node expressions, and c ranges over node constructors. For brevity, we write "$t\ \overline{v}.n$" for a transformer definition of name t, which takes the pattern variables $\overline{v}$ and has the body n. If the node expression is a pattern variable, we look it up in the node environment. If the node expression is a case-dispatch, we first evaluate the condition v in the current environments. The result is either an empty list Empty or a nonempty list $\mathrm{Cons}(hd, tl)$ with the first element hd and the rest of the elements tl. If it is an empty list, we evaluate n_1 in the current environments. If it is a nonempty list, we extend the node environment with the bindings for the pattern variables v_1 and v_2 to hd and tl respectively, and continue the evaluation. If the node expression is a macro invocation, we look up the transformer definition in the transformer environment, evaluate the macro arguments, and extend the node environment with the bindings from the pattern variables to their actual values, and continue evaluating the body. If the node expression is an

[Pattern Variable]

$$\frac{\Gamma(v) = n}{\Upsilon, \Gamma \vdash v \rightarrow \Upsilon, \Gamma \vdash n}$$

[Case Empty]

$$\frac{\Upsilon, \Gamma \vdash v \rightarrow \Upsilon, \Gamma \vdash \mathrm{Empty}}{\begin{array}{l}\Upsilon, \Gamma \vdash \texttt{case}\ v\ \texttt{of} \qquad\qquad \rightarrow \Upsilon, \Gamma \vdash n_1 \\ \quad \mathrm{Empty} \Rightarrow\ n_1 \\ \quad \mathrm{Cons}(v_1, v_2)\ \Rightarrow\ n_2 \\ \texttt{end}\end{array}}$$

[Case Cons]

$$\frac{\Upsilon, \Gamma \vdash v \rightarrow \Upsilon, \Gamma \vdash \mathrm{Cons}(hd, tl)}{\begin{array}{l}\Upsilon, \Gamma \vdash \texttt{case}\ v\ \texttt{of} \qquad \rightarrow \Upsilon, \Gamma\left[v_1 \mapsto hd\right]\left[v_2 \mapsto tl\right] \vdash n_2 \\ \quad \mathrm{Empty} \Rightarrow\ n_1 \\ \quad \mathrm{Cons}(v_1, v_2)\ \Rightarrow\ n_2 \\ \texttt{end}\end{array}}$$

[Macro Invocation]

$$\frac{\Upsilon(t) = t\ \overline{v}.n \qquad \Upsilon, \Gamma \vdash \overline{n} \rightarrow \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash t(\overline{n}) \rightarrow \Upsilon, \Gamma\left[\overline{v} \mapsto \overline{n'}\right] \vdash n}$$

[Ellipses First]

$$\frac{\begin{array}{c}PV(n'') \neq \emptyset \\ |\overline{n'}| + 1 = i \leq size(n) \qquad v_j \in PV(n) \qquad |\Gamma(v_j)| > 1 \\ \Gamma' = \Gamma\left[\overline{v}_j \mapsto \overline{\Gamma(v_j).nth(i)}\right] \qquad \Upsilon, \Gamma' \vdash n'' \rightarrow \Upsilon, \Gamma' \vdash n'''\end{array}}{\Upsilon, \Gamma \vdash \mathrm{Ellipses}(n, \overline{n'}n'') \rightarrow \Upsilon, \Gamma \vdash \mathrm{Ellipses}(n, \overline{n'}n''')}$$

[Ellipses Middle]

$$\frac{\begin{array}{c}PV(n'') = \emptyset \\ |\overline{n'}| + 1 = i - 1 < size(n) \qquad v_j \in PV(n) \qquad |\Gamma(v_j)| > 1 \\ \Gamma' = \Gamma\left[\overline{v}_j \mapsto \overline{\Gamma(v_j).nth(i)}\right] \qquad \Upsilon, \Gamma' \vdash n \rightarrow \Upsilon, \Gamma' \vdash n'''\end{array}}{\Upsilon, \Gamma \vdash \mathrm{Ellipses}(n, \overline{n'}n'') \rightarrow \Upsilon, \Gamma \vdash \mathrm{Ellipses}(n, \overline{n'}n''')}$$

[Ellipses Last]

$$\frac{PV(n'') = \emptyset \qquad |\overline{n'}| + 1 = size(n)}{\Upsilon, \Gamma \vdash \mathrm{Ellipses}(n, \overline{n'}n'') \rightarrow \Upsilon, \Gamma \vdash \overline{n'}n''}$$

[Node Constructor]

$$\frac{\Upsilon, \Gamma \vdash \overline{n} \rightarrow \Upsilon, \Gamma \vdash \overline{n'}}{\Upsilon, \Gamma \vdash c(\overline{n}) \rightarrow \Upsilon, \Gamma \vdash c(\overline{n'})}$$

Figure 10. Operational semantics for node expressions

ellipses node, the first argument of the node is replicated by the number of times equal to the length of a pattern variable within the first argument. The second argument of the ellipses node keeps track of the intermediate results during the evaluation of the ellipses node. For brevity, we write "$PV(n)$" for a set of pattern variables in the node expression n, "$size(n)$" for the length of a pattern variable within the node n, and "$l.nth(i)$" for the i-th element of the list l. If the node expression is a node constructor invocations, we construct the corresponding Core Fortress node.

The evaluation of a node expression may either fail, diverge, or result in an executable Core Fortress node. The evaluation may diverge if there is an infinite recursion in the definition of a transformer.[1] We do not consider nontermination a problem, because it

[1] Transformation described in this section is similar to Wand's[24] algorithm. The transformation function implements τ while the parser implements β and D.

will mainly happen when a macro does not deconstruct input, and programmers writing programs with recursive functions are used to deal with this kind of problems.

5.3 Hygiene

Fortress macros are hygienic, meaning any bindings introduced by a macro are fresh. Our hygienic system is a simplification of Clinger's algorithm [11]. Since our macro system cannot introduce macros itself, nor can a macro be hidden through a lexically bound variable, we need not concern ourselves with much of the complexity of the original hygiene algorithm.

The hygienic transformation works as follows. Before a macro is invoked, no renaming is performed. After a macro invocation, a flag is set which indicates that renaming should take place for the transformation of the current node and all the children nodes. During hygienic transformation, identifiers are looked up in a syntactic environment and replaced with the renamed identifiers they are bound to. The initial syntactic environment is the identity environment. Once the flag indicating that renaming should occur is set, each language construct binding an identifier generates a unique identifier and maps the original bound identifier to the new unique identifier. Each binding construct introduces a new syntactic environment which maintains a link to the previous environment. Identifiers not found in the immediate environment are recursively looked up in the parent environment until a mapping is found.

Pattern variables in a macro definition are never renamed since they are syntactic entities being introduced into the macro. In this way, we prevent variables passed to the macro from being renamed by virtue of the fact that they share names with bound identifiers introduced by the macro. Consider the following macro definition:

$$\text{Expr}|:= \mathit{foo}\ e : \text{Expr}\ \Rightarrow \lhd \texttt{fn}\ d \Rightarrow d + e \rhd$$

If the macro is invoked as:

$$\mathit{foo}\ d$$

then the result after hygienic transformation should be:

$$\texttt{fn}\ d_1\ \Rightarrow\ d_1\ +\ d$$

6. Implementation

The syntactic abstraction mechanism is built on top of Rats!, the underlying parser technology used to parse Fortress programs. A key aspect of Rats! is its composition framework based on modules. By separating grammar extensions in their own module, we were able to extend the core Fortress grammar in a modular way.

A new parser is generated for each modified or defined nonterminal in an extension grammar. All the nonterminals in the grammar are compiled into Rats! syntax and added into a new module, $\texttt{M}_{\text{user}}$. A nonterminal that is extended originally has productions $p_1 \ldots p_n$ and is modified so that $\texttt{M}_{\text{user}}$.`nonterminal` is inserted before p_1. All productions that refered to `nonterminal` will now implicitly refer to $\texttt{M}_{\text{user}}$.

The macro system is logically broken up into two stages: parsing, and transformation. The *first* stage consists of parsing the grammar declaration into an AST which contains transformation nodes instead of the template bodies. This stage has two steps: parsing the grammar declaration and parsing the template bodies. The core fortress parser is used to parse the grammar declarations and then those results are used to construct a new parser that is used to parse the templates. The system constructs a new parser for each unparsed template body and replaces the grammar AST node with a transformation node which contains the parsed template body. When a component is parsed, the grammars that it imports define the PEG used to parse the component body. The resulting AST will have nodes that specify a macro invocation. In stage *two*(transformation) the AST will be run through a macro processing engine which converts macro invocation nodes into their corresponding template bodies.

Template bodies consist of regular Fortress syntax, macro invocations, pattern variables, and ellipses nodes. During transformation, the macro system will dispatch according to these four cases. For regular Fortress syntax the engine will recursively dispatch on child nodes but otherwise do no additional processing unless a macro has already been invoked in which case the hygiene rules will be applied. Macro invocations will cause the engine to look up the function defined for that macro and apply it to the parameters of the macro. Pattern variables are looked up in the current macro's environment and replaced with a corresponding expression. Ellipses nodes cause the engine to replicate the immediate child node of the ellipses node a number of times equal to the length of a pattern variable that is also inside the ellipses node. The replicated nodes are then spliced into the parent node.

7. Evaluation

We have evaluated the design of our syntactic abstraction system by implementing the `for` loop example shown in Figure 3, and also a grammar that recognizes regular expressions and one recognizing XML. These examples show that the system is suitable for language extensions and domain specific languages by allowing their care-free implementation in general. The grammars for XML and regular expressions can be found as part of the open-source Fortress interpreter [3].

The for loop example demonstrates a weakness of the macro system in that some macros must be split up to accomodate the parser. The for loop cannot be written directly because when the ellipses are expanded the intermediate commas are missing and so the parser would not recognize the syntax as a for loop macro. Instead, the body of the for loop is written in such a way that no syntactic markers are needed and a separate macro is invoked with this simpler body. This generally means that most macros will need two forms and macro writers will have to know about both forms when invoking another macro inside their template. A solution to this problem would be to add operations over lists, so that the identifiers and the corresponding expressions in the for loop example could be zipped, and expanded with the commas in place. In general it would be useful if one could mark local helper macros as such, e.g. by using visibility modifiers like `private` in Java.

The system has a number of limitations. Syntax extensions are imported at the component level, and so we do not support lexically scoped syntax extensions like Scheme [2] or OMeta [37] does. Support for lexical scoping could be added later. A number of other systems [18, 37] based on PEGs allow semantic predicates in parsing expressions. A semantic action is a boolean-valued expression which must be true in order for a parsing expression to match. Semantic actions are useful in some situations, but in our exprience we haven't had any need for implementing any of our examples yet.

Currently the specification of syntax is not separated from the specification of semantics(actions), e.g. like in Bracha's [6]. Such a separation can be useful, e.g. for visualizing the source code in IDEs, or supporting different transformations. Explicit separation of the syntax and transformation is possible and we would like to investigate this in the future.

The current implementation uses Rats! to generate a parser for each transformation expression. The overall parsing of a Fortress program performs comparable to the parsers generated by Rats!, that is linear time. However there is the overhead of generating a grammar and invoking Rats! on that grammar for each transformation expression, which currently slows down the parsing first time a macro is used. The following times are for free because the gen-

erated parser is cached. The size of the transformed Core Fortress program is in worst case exponential in the size of the input.

Much of the syntactic abstraction system functionality falls out from the mechanisms inspired by Scheme systems, however the uniform and elegant structure of s-expressions cannot be reused in the context of Fortress, due to the many nuances of concrete syntax of Fortress. This prevented us from writing procedural macros efficiently, the amount of code required to construct a Fortress AST makes them an unattractive path to take.

8. Related work

Our system for syntactic abstraction incorporates ideas from two main lines of research: extensible grammars for meta-programming; and Lisp and Scheme macro systems.

Meta-programming systems and extensible grammar systems provide frameworks for constructing languages, extending languages with domain-specific notations, and translating between languages. Typically, a meta-program manipulates a separate object-program. Sometimes the meta-program and object-program are written in different languages.

In contrast, Lisp and Scheme macros affect subsequent parts of the same program that contains the macros. Using macros, programmers can write syntax extensions and domain-specific languages as libraries, with no need for preprocessors or custom compilers. Fortress takes this approach of supporting extension from within, rather than making it a separate facility with separate tools.

8.1 Extensible Syntax and Meta-programming

There are numerous languages, libraries, and frameworks designed for creating language tools, from simple parser generators [22] to comprehensive language definition frameworks. In this section we discuss a few systems that offer special support for language extension or translation.

Rats! [17, 18] is a tool for producing packrat [16] parsers from modular grammars. Using the module system a programmer can create new languages by applying modifications to existing grammars. Rats! provides parsing support for Fortress syntactic abstraction system.

OMeta [38] is an embedded language(in COLA [12] or Squeak Smalltalk [32]) which combines pattern matching and parsing. OMeta is based on PEGs which are generalized to work on arbitrary datatypes not only streams of characters, which leads to the use of PEGs for pattern matching. The encapsulation of grammars in OMeta is also inspired by object-oriented constructs but in contrast to grammars in Fortress they are encapsulated in a "class" like construct which only supports extension of a single grammar, the reason is mainly for avoiding nameclashes. Single extension is not a good match for expressing grammar extension since it is often the case that a grammar extends multiple other grammars. This is solved in OMeta though the use of the "foreign" production which effectively dispatches to separate grammars. Fortress solves this issue by allowing a grammar to extend multiple other grammars. "foreign" in Ometa also does not allow the programmer to selectively provide nonterminals to the user.

Katahdin [30] is a platform for specification and implementation of programming languages. A language is specified using a language where both the syntax and semantics are mutable at runtime. Katahdin is based on PEGs with some modifications, the choice is not prioritized but ordering is resolved based on a longest match strategy. Katahdin allows whitespace everywhere unless not explicitly disallowed, similar to our approach. However they go a step further and allow the programmer to redefined whitespace locally. A similar approach would benefit the Fortress syntactic abstraction system.

MetaBorg [8, 7, 27, 35] is a method of adding domain-specific notation support to general-purpose languages. The method relies on a syntax definition framework called SDF [20]. SDF permits definitions of arbitrary context-free grammars, which enables modular development of syntax. Another component in the MetaBorg tool chain is a term-rewriting language called Stratego. SDF and Stratego together support the construction of program transformers; SDF is used to describe the syntax and embed it into a Stratego program that specifies the transformation rules. The MetaBorg tools support language extension, but they constitute an external mechanism rather than an internal, integrated extension mechanism.

Silver [39] is a language description framework based on attribute grammars designed for the modular construction and composition of domain-specific languages. The semantics of the language is defined by its nonterminals' attributes. Silver supports a feature called forwarding [40] that allows the semantics of one syntactic form to be defined in terms of a translation into another syntactic form.

Cardelli et al. [9] devised an extensible syntax system based on grammar modifications. Like Lisp and Scheme but unlike most meta-programming systems, their system allowed programs to manipulate their own syntax; a programmer could introduce a new form and then use it in the same program.

Translations in their system were given in terms of concrete syntax patterns, allowing syntactic abstractions to be built incrementally. Like Scheme's hygienic macro systems, their system also preserves lexical scoping via renaming, but they have an explicit mechanism for requesting a fresh variable name.

Their system is more restrictive than ours in several ways. First, they use an LL(1) parser, and consequently their parser is efficient, but they cannot handle many of the syntaxes supported by Fortress. Their system performs transformations as part of the parsing process, and translations are analyzed to assure the termination of parsing. In contrast, our system has a separate translation pass, and while parsing is guaranteed to terminate, the translation pass might not.

8.2 Lisp and Scheme Macros

Fortress borrows several ideas from Lisp and Scheme macros. Fortress syntax extensions are part of Fortress programs, not external metadata requiring special handling. We also take heed of Scheme's consideration for proper lexical scoping. Finally, the semantics of grammars is designed to support the same kind of modularity patterns as Scheme's module systems do.

In 1986, Kohlbecker et al. [25] introduced *hygienic* macro expansion to prevent the inadvertent variable captures that sometimes occur in naive macro expansion. The original hygiene algorithm handled only top-level macros. Researchers in the Scheme community extended hygiene to cover locally defined macros [11, 14], first-order modules [15, 31, 36], and dynamically-linked components with macros in the signatures [13].

The original algorithm worked with top-level macros implemented as Lisp functions on syntax trees, and it involved scanning the result of every macro call to rename newly introduced identifiers. The algorithm developed by Clinger and Rees [11] eliminated the repeated code scanning by restricting macros to use fixed templates rather than arbitrary code to compute their results.

Our syntactic abstraction system implements hygiene in a manner that combines features of the Kohlbecker algorithm and the Clinger and Rees algorithm. Fortress macros occur only in grammars defined in APIs, so we do not need the machinery for locally defined macros, and Fortress macros specify their translations in terms of templates, so we do not need deep code scans.

The Fortress grammar system is designed to support modular syntax extensions. A common pattern in Scheme is to define one

macro in terms of several auxiliary macros but only export the main macro. In our system, a syntax extension can be defined with the help of other extensions, but clients of the one extension are not forced to accept the auxiliary extensions. In Scheme, this is done through the scoping of the macro name; in our system, grammars control the extent of changes to nonterminals.

Finally, the entire syntactic abstraction system resides ultimately in the APIs. Components are separately compiled; a component can be compiled given only the relevant APIs. Syntactic abstractions can refer to types, variables, and functions declared in the API, and the references they produce in client components are resolved based on the components the client is linked to. The Scheme component system with macros in the signatures [13] is arranged the same way and has similar hygiene properties.

9. Conclusion and Future Work

A growable language is one of the main ideas of the Fortress programming language. The Fortress syntactic abstraction mechanism serves a key role to support the language growth. It allows a user-defined syntax to be indistinguishable from core Fortress syntax, provides composition of independent macros, and supports mutually recursive macros using dynamic dispatches. The Fortress syntactic abstraction mechanism is so flexible that various language constructs can be moved from the core language syntax into libraries using macros.

Type checking macros is an interesting future direction. Because the syntactic abstraction mechanism described in this paper is a template-based approach, it is possible to perform a more precise type checking than a multi-staged system where the transformation is represented as an explicit construction of the AST nodes. The goal of type checking macros would be to provide the static guarantee that if a macro definition is well typed then there is no type error at the use sites of the macro unless the user of the macro provides an input of a wrong type.

Acknowledgments

The authors sincerely thank the Fortress team for fruitful discussions and support. Thanks to Robert Grimm for answering our questions about Rats!, and thanks to the anonymous reviewers for suggestions on improving the paper.

References

[1] Annika Aasa, Kent Petersson, and Dan Synek. Concrete syntax for data objects in functional languages. In *LISP and Functional Programming*, pages 96–105, 1988.

[2] H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams IV, D.P. Friedman, E. Kohlbecker, G.L. Steele Jr., D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[3] Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project Fortress Community website. `http://www.projectfortress.sun.com`.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0. `http://research.sun.com/projects/plrg/fortress.pdf`, March 2008.

[5] Claus Brabrand and Michael I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming Journal (SCP)*, 68(1):2–20, 2007.

[6] Gilad Bracha. Executable grammars in newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[7] Martin Bravenboer, Ren De Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE05*. Springer Verlag, 2005.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[9] Luca Cardelli and Florian Matthes. Extensible syntax with lexical scoping. Technical report, Research Report 121, Digital SRC, 1994.

[10] Noam Chomsky. Three models for the description of language. *IEEE Transactions*, 2(3), September 1956.

[11] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM.

[12] The cola programming language. `http://piumarta.com/software/cola/`.

[13] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In Robert Glck and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2005.

[14] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992.

[15] Matthew Flatt. Composable and compilable macros: you want it when? In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[16] 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.

[17] Robert Grimm. Practical packrat parsing. Technical report, New York University, 2004.

[18] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[19] Jr. Guy L. Steele. Growing a language. Keynote talk, OOPSLA, 1998. Also published at *Higher-Order and Symbolic Computation* 12, 221–236, 1999.

[20] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[21] Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.

[22] Stephen C. Johnson. Yacc: Yet another compiler-compiler. *Unix Programmer's Manual*, 2b, 1979.

[23] Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, 1984.

[24] E. E. Kohlbecker and M. Wand. Macros-by-example. In *POPL '87: Proceedings of the 14th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–84, New York, NY, USA, 1987. ACM.

[25] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[26] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[27] Jonathan Riehl. Assimilating metaborg:: embedding language tools in languages. In *International Conference on Generative Programming and Component Engineering*, pages 21–28, New York, NY, USA, 2006. ACM.

[28] Sablecc. `http://sablecc.org`.

[29] Scheme frequently asked questions. `http://community.schemewiki.org/?scheme-faq-macros`.

[30] Christopher Graham Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, Department of Computer Science, University of Bristol, United Kingdom, May 2007.

[31] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, and Jonathan Rees (Editors). Revised^6 report of the algorithmic language Scheme, September 2007. Available at `http://www.r6rs.org`.

[32] Squeak smalltalk. `http://www.squeak.org/`.

[33] The Unicode Consortium. *The Unicode Standard, Version 5.0*. Addison-Wesley, 2006.

[34] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[35] Eelco Visser and Eelco Visser. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, pages 299–315. Springer-Verlag, 2002.

[36] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, 1999.

[37] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

[38] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2007. ACM Press.

[39] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[40] Eric Van Wyk, Oege De Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction, volume 2304 of LNCS*, pages 128–142. Springer-Verlag, 2002.