Parsing Fortress syntax

Conference Paper · January 2009

DOI: 10.1145/1596655.1596667 · Source: DBLP

1 author:

Sukyoung Ryu
Korea Advanced Institute of Science and Tec…
37 PUBLICATIONS **380** CITATIONS

Parsing Fortress Syntax

Sukyoung Ryu
Sun Microsystems Laboratories
35 Network Drive
Burlington, MA 01803, USA
sukyoung.ryu@sun.com

ABSTRACT

In this paper, we report our experience with parsing the syntax of 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.

Parsing the Fortress syntax is nontrivial due to its support for mathematical syntax and growable syntax. Mathematical syntax is highly ambiguous and growable syntax allows a program itself to define how it is parsed. Fortress currently runs entirely on the JVM, which requires internal representation of Fortress as Java classes. We describe our trials to parse the entire Fortress syntax, in the presence of constant changes of the language syntax and its internal representation. and the lessons we learned from the experience.

Categories and Subject Descriptors

D.3 [**Programming Languages**]; D.3.3 [**Language Constructs and Features**]

General Terms

Languages, Parsing

1. INTRODUCTION

Fortress [4] is a new programming language designed for high-performance computing (HPC) with high programmability. It provides mathematical syntax to enable scientists and engineers to write programs in a notation they are accustomed to. It also provides implicit parallelism where possible and built-in support for transactional memory. Fortress is a multi-paradigm programming language supporting object-oriented and functional programming and it is designed for growth by community participation and development. Two main features of Fortress make parsing Fortress programs especially hard: mathematical syntax and growable syntax.

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
Copyright 2009 ACM 978-1-60558-598-7 ...$10.00.

```
z = 0
r = x
ρ = r^T r
p = r
DO i = 1, 25
    q = A p
    α = ρ/(p^T q)
    z = z + α p
    ρ_0 = ρ
    r = r − α q
    ρ = r^T r
    β = ρ/ρ_0
    p = r + β p
ENDDO
compute residual norm explicitly: ‖r‖ = ‖x − A z‖
```

Figure 1: NAS "CG" conjugate gradient specification

```
conjGrad⟦Elt extends Number, nat N,
         Vec extends Vector⟦Elt, N⟧,
         Mat extends Matrix⟦Elt, N, N⟧ ⟧
       (A: Mat, x: Vec): (Vec, Elt) = do
    z: Vec := 0
    r: Vec := x
    ρ: Elt := r · r
    p: Vec := r
    for j ← seq(1 # 25) do
        q = A p
        α = ρ/(p · q)
        z += α p
        ρ_0 = ρ
        r −= α q
        ρ := r · r
        β = ρ/ρ_0
        p := r + β p
    end
    (z, ‖x − A z‖)
end
```

Figure 2: NAS "CG" conjugate gradient in Fortress

As much as possible, we allow Fortress syntax to look like the standard mathematical notation used by scientists in their work. We provide rich support for entering and displaying Unicode [17] characters for mathematical notation. Using mathematical notation reduces the overhead for scientists to learn Fortress, because it looks like the equa-

```
$g_1 = \langle 1,2,3,4,5 \rangle$
$g_2 = \langle 6,7,8,9,10 \rangle$
for $i \leftarrow g_1, j \leftarrow g_2$ do
    $println$ "(" $i$ ", " $j$ ")"
end
```

Figure 3: A parallel `for` loop in Fortress

```
$g_1.loop($fn $i \Rightarrow$
        $g_2.loop($fn $j \Rightarrow$
                $println$ "(" $i$ ", " $j$ ")" ))
```

Figure 4: A desugared version of the `for` loop in Figure 3

tions they already use in publications, or when writing on paper or a whiteboard. As an example of Fortress code, Figure 2 shows a function that performs a matrix/vector "conjugate gradient" calculation (based on the NAS "CG" conjugate gradient benchmark for parallel algorithms) [6]. The function *conjGrad* takes a matrix and a vector, performs a number of calculations, and returns two values: a result vector and a measure of the numerical error. We believe that having programs look like the equations that scientists actually use will reduce errors introduced in "translating" those equations into program code, and will enable scientists to more quickly find any errors that are introduced. Figure 1 presents the original specification of the conjugate gradient calculation to show how much the Fortress program looks like the problem specification.

While using mathematical notation may enhance programmability for the users of Fortress, supporting it is not trivial because mathematical notation is highly ambiguous. For example, juxtapositions of two numbers are multiplications but juxtapositions of a function and a number are function applications. Parsing the expression:

$$3x \sin x \cos 2x \log \log x$$

requires types of juxtaposed items. When x is a number and each of $\sin$, $\cos$, and $\log$ is a function over numbers returning a number, it is parsed as the following expression:

$$((((3x)(\sin x))(\cos(2x)))(\log(\log x)))$$

Properties of mathematical operators also affect parsing mathematical equations: operator fixity, precedence, and associativity. The same operator $+$ may be used as an infix operator as in "$3+5$" or as a prefix operator as in "$+4$". A more delicate problem is whitespace sensitiveness. Parsing the expression:

$$\{\, |x| \mid x \leftarrow mySet, 3|x \,\}$$

takes whitespace into account to recognize different uses of "$|$". Depending on the whitespace context, it may be an enclosing operator as in "$|x|$", an infix operator as in "$3|x$", or a separator used in the entire set comprehension expression.

Fortress is designed to grow over time to accommodate the changing needs of its users via a syntactic abstraction mechanism [5]. It is possible to add new language 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. In Figure 3, 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. The Fortress syntactic abstraction mechanism allows `for` loops such as this to be transformed into method calls and anonymous functions as in Figure 4.

However, parsing Fortress programs using new syntax is nontrivial because a program itself defines how it is parsed: the new syntax definitions in the program determines which syntax may occur in the program itself. In order to resolve this self-dependency problem, we take a two-step approach to parse Fortress programs: "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 without any uses of the new syntax by desugaring away the new syntax.

In addition to the design goals of Fortress, practical issues also make parsing Fortress programs particularly hard. Fortress is designed as a next-generation programming language which allows us to bring various lessons from many modern programming languages into Fortress. Experiments with combinations of various language features led to continuous language changes, which in turn required frequent changes of the Fortress parser. At the same time, the changes in the concrete syntax often required changes in the internal representation of the Fortress Abstract Syntax Tree (AST).

In this paper, we present our trials to parse the entire Fortress syntax in the presence of the above mentioned challenges. Section 2 describes the parsing methods that we have tried so far and Section 3 describes the pros and cons of the parser generator that we currently use. Section 4 presents the entire parser family of Fortress and Section 5 summarizes our lessons from the Fortress parser development.

2. FORTRESS PARSER TRIALS

We tried several parsing techniques roughly in the order of increasing expressiveness.

2.1 JavaCC: LL(k)

JavaCC (Java Compiler Compiler) [2] is a parser/scanner generator for the Java$^{\text{TM}}$ Programming Language [7]. JavaCC generates top-down parsers, which limits it to the LL(k) class of grammars. An LL(k) parser parses the input from left to right, and constructs a leftmost derivation of the sentence using k tokens of lookahead. We have found that parsing Fortress requires a high value of k which makes JavaCC unsatisfactory.

2.2 CUP: LALR(1)

CUP [1] is a LALR parser generator for the Java Programming Language. An LR parser parses the input from left to right, and constructs a rightmost derivation of the

- $\diamond$ All kinds of whitespace, in any quantity.
- $\square$ At least one whitespace character of any kind.
- $\not\hookleftarrow$ Whitespace containing no line-ending characters.
- $\hookleftarrow$ Whitespace containing at least one line-terminating character.
- $\emptyset$ Explicitly, no space at all.

Figure 5: Whitespace description in a Fortress syntax for a hand-written recursive descent parser

$\mathit{Expr5}_\sigma ::= \mathit{Expr6}_\sigma\ (\ \mathit{Op5}\ \mathit{Expr6}_\sigma\ )\ *$

$\mathit{Op5} ::= (\ \underline{\texttt{=}} \mid \underline{\texttt{<}} \mid \underline{\texttt{>}} \mid \underline{\texttt{>=}} \mid \underline{\texttt{<=}} \mid \underline{\texttt{/=}} \mid \underline{\texttt{<>}}\ )\ \diamond$

$\mathit{Expr6}_\sigma ::= \mathit{Expr7}_\sigma\ (\ \mathit{Op6}\ \mathit{Expr7}_\sigma\ )\ *$

$\mathit{Op6} ::= (\ \underline{\texttt{+}} \mid \underline{\texttt{-}}\ )\ \diamond$

$\mathit{BigExpr7}_\sigma ::= \underline{\texttt{SUM}}\ \diamond\ \underline{\texttt{[}}\ \diamond\ \mathit{DrawnFroms}\ \underline{\texttt{]}}\ \diamond\ \mathit{Expr7}_\sigma$

$\mathit{BigExpr8}_\sigma ::= \underline{\texttt{PRODUCT}}\ \diamond\ \underline{\texttt{[}}\ \diamond\ \mathit{DrawnFroms}\ \underline{\texttt{]}}\ \diamond\ \mathit{Expr8}_\sigma$

$\mathit{Expr7}_\sigma ::= \mathit{Expr8}_\sigma\ (\ \mathit{Op7}\ \mathit{Expr8}_\sigma\ )\ *$

$\mathit{Op7} ::= (\ \underline{\texttt{TIMES}} \mid \underline{\texttt{*}} \mid \underline{\texttt{/}} \mid \underline{\texttt{REM}} \mid \underline{\texttt{DIV}} \mid \underline{\texttt{MOD}}\ )\ \diamond$

$\mathit{Expr8}_\sigma ::= (\ \mathit{someUnary}\ )*\ \mathit{Expr8a}_\sigma$

$\mathit{someUnary} ::= (\ \underline{\texttt{-}} \mid \underline{\texttt{\#}}\ )\diamond$

Figure 6: A sample Fortress syntax for a hand-written recursive descent parser

sentence. An LALR parser or a lookahead LR parser is a specialized form of LR parser and it is known to be a good trade-off between the size of grammars it can handle and the size of parsing tables it uses. Most compiler-compilers such as yacc [11] generate LALR parsers. Again, we found that parsing Fortress requires much lookahead which makes CUP not sufficient for Fortress.

2.3 Hand-Written Recursive Descent

A recursive descent parser is a top-down parser generated from a set of (possibly) mutually-recursive procedures. Each procedure usually implements one of the productions of the grammar. Therefore, the grammar structure is closely reflected in the structure of the parser code. A recursive descent parser makes it relatively easier for the language designer to handle the subtle details of the language syntax.

We tried a hand-written recursive descent parser for a version of Fortress concrete syntax which is similar to LL with scanner hacks. Whitespace is explicit in the grammar as described in Figure 5. In order to handle operator uses, the expression grammar was layered by operator precedence. A sample syntax of this version of Fortress concrete syntax is shown in Figure 6. The entire syntax was not easy to understand, not adaptable for syntax changes, and the parser implementation was very difficult.

2.4 Elkhound: GLR(1)

Elkhound [13] is a Generalized LR (GLR) parser generator written in C++ [16], and can generate parsers in either C++ or OCaml [12]. GLR parsers can parse any context-free grammar. They support unlimited lookahead and am-

```
nonterm(program) compilation_unit {
fun merge(one,two) { amb one.node_span ‘‘program’’ }

-> w c:api w { node loc (‘API c) }
-> w c:component w { node loc (‘Component c) }
-> imps:imports_opt
   exps:exports_opt
   defs:defs_opt w
  { node loc (‘Component ( node loc
      { component_name = node loc
          (Str.split (Str.regexp ‘‘\\.’’)
                     (Filename.chop_extension
                         (Filename.basename
                              loc.Lexing.pos_fname)));
        component_imports = imps;
        component_exports = exps;
        component_defs = defs; } ) ) }

}
```

Figure 7: A sample Fortress syntax in Elkhound

```
nonterm(expr) expr {
fun merge(one,two) {
  amb_expr "expr" one two
}
  -> oe:op_expr { oe }
  -> t:tuple_expr { t }
  -> fe:flow_expr { node loc (‘FlowExpr fe) }
  -> fe:fn_expr { fe }
  -> oe:object_expr { oe }
  -> ae:assignment_expr { ae }
  -> te:type_ascription_expr { te }
}

nonterm(expr) no_newline_expr {
fun merge(one,two) {
  amb_expr "no_newline_expr" one two
}
  -> oe:no_newline_op_expr { oe }
  -> t:tuple_expr { t }
  -> fe:no_newline_flow_expr { node loc (‘FlowExpr fe) }
  -> fe:no_newline_fn_expr { fe }
  -> oe:object_expr { oe }
  -> ae:no_newline_assignment_expr { ae }
  -> te:no_newline_type_ascription_expr { te }
}

nonterm(expr) no_space_expr {
  fun merge(one,two)
    { amb_expr "no_space_expr" one two }
-> oe:no_space_op_expr { oe }
-> t:tuple_expr { t }
-> fe:no_space_flow_expr { node loc (‘FlowExpr fe) }
-> oe:object_expr { oe }
-> ae:no_space_assignment_expr { ae }
}

/* w = any whitesapce, or none */
nonterm(unit) w {
fun merge(one,two) { amb None "w" }
-> { () }
-> wr { () }
}
/* wr = nonempty whitespace */
nonterm(unit) wr {
fun merge(one,two) { amb None "wr" }
-> TOK_WHITESPACE w { () }
-> TOK_NEWLINE w { () }
}
```

Figure 8: Whitespace-sensitive expression productions in Elkhound

biguous grammars by parsing all possibilities in parallel and wait for one of the possibilities to win over the others.

We could parse most of the Fortress syntax in Elkhound but we never finished parsing the entire syntax as with the other parsing trials described in the previous sections. Figure 7 shows the Fortress grammar syntax in Elkhound and generated a Fortress parser in OCaml to be used by the Fortress interpreter in OCaml developed in parallel. Some syntactic restrictions, such as different modifiers on different language constructs, were not handled by the parser; instead, they were assumed to be checked in later phases of the interpreter/compiler. The final state of the parser was about 3,500 lines (counted by '`wc -l`') of Elkhound productions, about 300 lines of lexer, and about 1,000 lines of OCaml support code.

While Elkhound was the most powerful tool that we had tried until then, it had several issues for us. First, Elkhound did not have a good abstraction mechanism. Whitespace-sensitive productions were repeated with minor differences as shown in Figure 8. Second, the performance of the generated parser was not scalable. The parser could not parse 64×32 arrays, for example[1]. Finally, we had some changes in the Fortress interpreter implementation. The new developers took over the existing parser/interpreter implementation and tried to replace the old interpreter in OCaml with a new interpreter in Java. The original plan was to convert OCaml AST from the Elkhound/OCaml parser to some internal representation similar to s-expressions, then read them back into the Java interpreter. However, because the new developers had no experiences of programming in OCaml before, the parser was very slowly adapted to the Fortress syntax changes and the internal representation between OCaml AST and the Java AST was never completed.

2.5 Rats!: Packrat

Rats! [9] is a packrat parser generator, which is written in Java and can generate parsers in Java. Rats! grammars build on Parsing Expression Grammars (PEGs) [8]. Unlike Context-Free Grammars (CFGs), PEGs use "ordered" productions. Therefore, they are unambiguous and support localized changes, and they are also closed under composition, intersection, and complement. Packrat parsers memoize intermediate results, which guarantees linear time performance. They support unlimited lookahead by matching expressions without consuming inputs and integrate lexing and parsing which simplifies maintaining the parsers in the presence of constant syntax changes.

We rewrote the Fortress parser from scratch in Rats!, which turned out to be a real success. We could avoid using two additional implementing languages (and several tools required to support them) from the Fortress interpreter/-compiler code base: C++ and OCaml. We could reduce the number of the repeated (with minor differences) productions due to whitespace-sensitiveness by using the Rats! module system. The new keywords introduced in the new syntax could be easily introduced by the scannerless parsers generated by Rats!. Last but not least, the constant helps from the developer, Robert Grimm, who has been very responsive and open to any suggestions, made it possible for us to replace the existing Elkhound-generated parsers with Rats!-generated parsers.

[1]The test Fortress program is available at: `http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/tests/arrayBig.fss`.

Fortress.rats	root module
Compilation.rats	compilation units
Declaration.rats	top-level declarations
OtherDecl.rats	dimension/unit/... declarations
TraitObject.rats	trait/object declarations
Function.rats	function declarations
Variable.rats	variable declarations
Method.rats	method declarations
LocalDecl.rats	local declarations
Syntax.rats	new syntax declarations
Parameter.rats	function parameters
MethodParam.rats	method parameters
MayNewlineHeader.rats	headers with newline
NoNewlineHeader.rats	headers without newline
Type.rats	types with newline
NoNewlineType.rats	types without newline
Expression.rats	expressions with newline
NoNewlineExpr.rats	expressions without newline
NoSpaceExpr.rats	expressions without whitespace
DelimitedExpr.rats	expressions enclosed by keywords
Literal.rats	literals
NoSpaceLiteral.rats	literals without whitespace
Identifier.rats	identifiers
Keyword.rats	keywords
Symbol.rats	symbols
Spacing.rats	spaces and comments
Unicode.rats	unicode characters

Figure 9: Rats! grammar modules for the entire Fortress syntax

3. FORTRESS PARSERS IN RATS!

The Rats! parser generator has been very useful in the development of the Fortress parser. It is the first parser generator that generated parsers for the entire Fortress syntax. In this section, we discuss the pros and cons of using Rats! in the development of the open-sourced Project Fortress [3].

3.1 Advantages of Rats!

3.1.1 Module System

Rats! organizes grammars into modules to support conciseness and ease of modifiability. It allows the entire Fortress syntax to be split into separate units of related productions. As of revision 3641 in the Project Fortress repository, there are 27 Rats! modules as described in Figure 9. Splitting a huge set of productions into smaller sets of related productions separates concerns and makes it easier to debug the grammar productions.

Rats! allows modules to reuse existing modules with modifications, which avoids error-prone, duplicated productions with small changes. Among 27 modules, `TraitObject` and `MethodParam` modify `Param`, `NoNewlineExpr` and `NoSpaceExpr` modify `Expr`, and `NoNewlineType` modifies `Type`. Figure 10 shows that the `NoSpaceExpr` module modifies the `Expr` module by removing some productions of several nonterminals in the original `Expr` module.

Figure 10 also shows that the `NoSpaceEpxr` module is parameterized by four modules: `Expr`, `Keyword`, `Symbol` and

```
module com.sun.fortress.parser.NoSpaceExpr(
                              Expr, Keyword,
                              Symbol, Spacing);
modify Expr;
import Keyword;
import Symbol;
import Spacing;
...
Expr ExprFront   -= <Flow> , <Fn> ;
Expr NoSpaceExpr = ExprFront ;
...
PureList<PrecedenceOpExpr> TightInfixPostfix
                 -= <Primary>, <Prefix>, <Left> ;
```

Figure 10: An example Rats! module modification

$$
\begin{array}{l}
\textit{Comprehension} ::= \\
\quad \texttt{BIG}?\ \textit{LeftEncloser}\ \textit{StaticArgs}?\ \textit{Entry}\ \texttt{|} \\
\qquad\qquad\qquad\qquad \textit{GeneratorClauseList}\ \textit{RightEncloser}
\end{array}
$$

Figure 11: Comprehension syntax in EBNF

`Spacing`. Parameterized modules are very useful to combine each module with other modules with respect to whitespace requirements. For example, array elements are separated by spaces in Fortress: [1 2 3 4 5]. Therefore, expressions used in the array elements context should not include any spaces. This is represented as the following module instantiations in Fortress.rats:

```
instantiate com.sun.fortress.parser.NoSpaceLiteral(
                   MayNewlineHeader, NoSpaceExpr,
                   Symbol, Spacing)
```

The `NoSpaceLiteral` module includes the productions for array elements and it uses `NoSpaceExpr` instead of `Expr` or `NoNewlineExpr` to parse expressions in the array elements context.

3.1.2 Grammar Specification

Rats! productions are very close to the Fortress concrete syntax grammar in Extended BNF (EBNF) notation. The syntax of the following set comprehension expression:

$$\{\, |x| \mid x \leftarrow mySet, 3|x \,\}$$

is represented in EBNF as shown in Figure 11 and is implemented in the `DelimitedExpr` module as shown in Figure 12. The nonterminal `Comprehension` comes after the Java class `Expr`, which represents the AST node constructed by the action part "`{ ... }`" in Java code. Except the whitespace notations, such as `w` and `wr`, and labels on the symbols, such as `a1:` and `a2:`, to be used in the action part, the grammar specifications in Rats! are pretty close to the EBNF notation. Similar productions in the specification and the corresponding implementation reduce "translation" errors.

3.1.3 Global Parsing State

Rats! provides an interface for managing global parsing states to support parsing context-sensitive expressions. Global states are manipulated within lightweight transactions; four methods for transactions should be provided: `reset`, `start`, `commit`, and `abort`. The global state is reset by a

```
Expr Comprehension =
     (BIG w)? a1:LeftEncloser a0:hasW a2:StaticArgs?
     w a3:Entry wr bar wr a4:GeneratorClauseList w
     a5:RightEncloser
     { ... }
```

Figure 12: Comprehension syntax in Rats! productions

```
void Program = (Token w)+ ;

stateful void Token =
     ...
   / api        { yyState.left(...); }
   / case       { yyState.left(...); }
   / component  { yyState.left(...); }
   / typecase   { yyState.left(...); }
   / also       { yyState.left(...); }
   / end        { yyState.right(...); }
   / a1:LeftEncloser  { yyState.left(...); }
   / a1:RightEncloser { yyState.right(...); }
     ...
```

Figure 13: Global parsing state manipulation via the `stateful` feature

production which invokes `reset`. For each production that might modify the global state, a new transaction begins by invoking `start` and the transaction completes on a successful parse by invoking `commit` and on a failed parse by invoking `abort`.

We use this global parsing state for checking unmatched delimiters. As we discuss in Section 3.2 in more detail, syntax errors reported by the Rats!-generated parsers are often not informative. To provide better error messages, we run a delimiter checker over a Fortress program before actually parsing the program. The delimiter checker maintains a global stack of opening delimiters and closing delimiters to detect any unmatched delimiters. As shown in Figure 13, the nonterminal `Token` is marked as `stateful` to note that this production might modify the global state. For each opening token, such as `api`, `case`, and `LeftEncloser`, the token is pushed into the global stack: `yyState.left(...)`. For each closing token, such as `end` and `RightEncloser`, the token is checked to see whether it matches to the popped token from the global stack: `yyState.right(...)`.

3.2 Disadvantages of Rats!

3.2.1 Syntax Error Reporting

While Rats! provides several error handling facilities such as pretty printing grammars, type checking the semantic values of the productions, and tracking line and column numbers, syntax error messages from Rats!-generated parsers are often not helpful. Rats!-generated parsers automatically deduce error messages from nonterminal names. For example, a parse error within the production for `Comment` results in the error message "comment expected." Because Fortress syntax is whitespace sensitive, most productions include nonterminals for whitespaces which include `Comment`. As a result, almost every syntax error message was "comment expected." The error message problem is not only for the Fortress syntax. In the case of the C and Java parsers included in the

```
transient void InvalidSpace =
  ("\t" / "\u000b")
  { log(createSpan(yyStart,yyCount),
        "Tab characters are not allowed in " +
        "Fortress programs except in comments.");
  }
/ ("\u001c" / "\u001d" / "\u001e" / "\u001f")
  { log(createSpan(yyStart,yyCount),
        "An invalid whitespace character is used.");
  };
```

Figure 14: Error productions for better syntax error messages

```
module com.sun.fortress.parser.Field(
       Variable, Header, Type, Identifier, Spacing);
modify Variable;
...
VarDecl FldDecl = VarDecl ;
List<Modifier> VarMods := FldMods ;
List<Modifier> VarImmutableMods := FldImmutableMods ;


module com.sun.fortress.parser.AbsField(
       Variable, Header, Type, Identifier, Spacing);
modify Variable;
...
AbsVarDecl AbsFldDecl = AbsVarDecl ;
List<Modifier> AbsVarMods := AbsFldMods ;
```

Figure 15: Field declaration syntax in components and APIs using Rats! module system

Rats! distribution[9], most error messages are "symbol characters expected."

New Fortress programmers frequently request improved syntax error messages. To provide more precise and informative error messages, we added some ad-hoc error productions for common cases. For example, Figure 14 shows error productions for invalid space characters. The Rats!-generated parser recognizes the invalid space characters and reports the error with the source location and the corresponding error messages. However, error productions increase the size of productions, slow down the parsing performance, and decrease the readability of the grammar due to the added visual clutter.

3.2.2 Module System

While the Rats! module system improved the readability and productivity of the Fortress parser development, the precise productions described by the module system resulted in bad syntax error messages. For example, Fortress provides similar but different syntax for top-level variable, local variable, and field declarations in APIs and components. They have different sets of modifiers and declarations in APIs must have type annotations. We first used the Rats! module system to describe their syntax. The most elaborate syntax was described in Variable.rats and the other variants were described in each module, LocalDecl.rats, Field.rats, and AbsField.rats modifying Variable.rats as described in Figure 15.

While it reduced the size of copy-and-pasted code and it described the syntax of each case very precisely, the syntax

$$
\begin{array}{rcl}
\textit{IntExpr} & ::= & \textit{IntExpr} + \textit{SumExpr} \\
 & | & \textit{SumExpr} \\
\textit{SumExpr} & ::= & \textit{IntVal}
\end{array}
$$

Figure 16: Left-recursive production in EBNF

```
IntExpr IntExpr =
     seed:SumExpr list:IntExprTail*
     { yyValue = (IntExpr)apply(list, seed); };
constant transient Action<IntExpr> IntExprTail =
     SumIntExpr
constant inline Action<IntExpr> SumIntExpr =
     w plus w a1:SumExpr
     { yyValue = new Action<IntExpr>() {
           public IntExpr run(IntExpr base) {
               ...
           }};
     };
transient IntExpr SumExpr = IntVal ;
```

Figure 17: Transformed left-recursive production in Rats!

error messages from a variable declaration with invalid modifiers, for example, were mostly "comment expected" again. Because the Rats!-generated parsers commit to a result once the result for a production at a given input position has been determined, the occurrences of invalid modifiers in the front of a variable declaration prohibit the parser from recognizing the variable declaration. To provide better syntax error messages in these cases, we decided to sacrifice the preciseness of the grammar productions and check for the validity of the modifiers after parsing the program. Section 4 describes our current approach in more detail.

3.2.3 Left-Recursive Productions

Like other recursive descent parsers, Rats! productions do not support left-recursive productions in general. It supports "direct" left-recursions for void, text-only, and generic productions but because the Fortress parser creates its own Fortress AST, the left-recursive productions must be manually transformed to the corresponding right-recursive productions. For example, a simplified syntax of static integer expressions in Fortress presented in Figure 16 should be rewritten as in Figure 17 using the Rats! facilities to create left-recursive data structures. Manual transformations of left-recursive productions are not difficult but the transformed productions look very different from the original productions and there are no guarantees for the correctness of such transformations.

4. FORTRESS PARSER FAMILY

Based on the lessons from the series of our trials to parse the entire Fortress syntax, in the presence of constant changes of the language syntax and its internal representation, we ended up with a family of Fortress parsers. We describe each part of the Fortress parser family in this section.

4.1 Abstract Syntax Tree

The Fortress AST was first developed in OCaml but it was replaced by hand-written Java classes when the Elkhound parser generator was replaced by Rats! entirely. Even though

```
abstract Decl();
  abstract TraitObjectDecl(TraitTypeHeader header)
                        implements Generic;
      TraitDecl(List<BaseType> excludesClause,
                Option<List<BaseType>> comprisesClause,
                boolean comprisesEllipses);
      ObjectDecl(Option<List<Param>> params)
                implements ObjectConstructor;
  VarDecl(List<LValue> lhs, Option<Expr> init);
  FnDecl(FnHeader header, Id unambiguousName,
         Option<Expr> body,
         Option<Id> implementsUnambiguousName)
        implements Applicable, Generic;
  abstract DimUnitDecl();
      DimDecl(Id dimId, Option<Type> derived,
              Option<Id> defaultId);
      UnitDecl(boolean si_unit, List<Id> units,
               Option<Type> dimType,
               Option<Expr> defExpr);
  TestDecl(Id name, List<GeneratorClause> gens,
           Expr expr);
  PropertyDecl(Option<Id> name, List<Param> params,
               Expr expr);
  TypeAlias(Id name,
            List<StaticParam> staticParams,
            Type typeDef);
  GrammarDecl(Id name, List<Id> extendsClause,
              List<GrammarMemberDecl> members,
              List<TransformerDecl> transformers,
              boolean nativeDef);
```

Figure 18: Partial description of the Fortress AST node structure

Rats! supports the automatic generation of abstract syntax trees, each of the Fortress AST nodes includes specific information depending on its kind. There were 223 nodes in hand-written Java classes, which lacked consistent methods and it was very error prone and tedious to cope with constant language changes.

To overcome these shortcomings, we decided to replace existing Java classes representing Fortress AST nodes with automatically generated Java classes by the ASTGen tool [15]. ASTGen automatically generates Java source files for a composite class hierarchy and accompanying visitors from a concise definition of the hierarchy. Figure 18 provides a part of the Fortress AST node structure. From this definition of the hierarchy, ASTGen generates a Java source file for each node class.[2]

In order to generate the entire Fortress AST structure automatically by ASTGen, we had to clean up the Fortress AST node hierarchy. For example, we factored out many utility static methods from node classes and removed mutation of nodes. Because these changes were made in the internal representation of the Fortress syntax, they did not affect the expressiveness of the Fortress language. Even though it took quite a long time and much effort to replace the classes, the task immensely reduced the costs of maintaining and learning the Fortress AST structure. We could revise the Fortress AST structure more easily and correctly after the ASTGen migration. As a byproduct of the migration, we improved the output of many error messages in the interpreter to provide more specific exceptions with source location information whenever possible.

[2]The entire Fortress node hierarchy is defined at:
`http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/astgen/Fortress.ast`

```
grammar ForLoop extends { Expression, Identifier }
 Expr |:=
   for { $i$ : Id ← $e$ : Expr , ? SPACE }* do $block$ : Expr end ⇒
   ◁[ $for_2$ $i$ * *; $e$ * *; do $block$; end ]▷
 | $for_2$ $i$ : 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 $ia$ ⇒ ($for_2$ $ib$ * *; $eb$ * *;
                                   do $block$; end))) ]▷
       end
   end
end
```

Figure 19: A grammar definition of a simplified syntax of for loops in Fortress

4.2 Syntactic Abstraction

Fortress is intended to grow over time to accommodate the changing needs of its users. To support such growth, we designed and implemented the Fortress syntactic abstraction mechanism. Figure 19 presents a definition of a simplified syntax of `for` loops in Fortress. The main challenges have been (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. 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. [3] 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. Because grammar definitions introduce new syntax to the language, the program itself defines how to parse it and turn it into an AST. Therefore, parsing a Fortress source program to a corresponding AST 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 in an AST representation which describes how the program is to be trans-

[3]The brief description of the syntactic abstraction implementation is available at:
`http://projectfortress.sun.com/Projects/Community/wiki/SyntacticAbstractionCode`

```
stateful void Token =
     LiteralExpr
   / import w api // ignore the api keyword
   / ...
   / Encloser   // ignore vertical bars
   / a1:LeftEncloser  { yyState.left(a1); }
   / a1:RightEncloser { yyState.right(a1); }
   / a1:["]
     { yyState.quote(createSpan(yyStart,yyCount),
                     "\\\""); }
   / a1:[’]
     { yyState.quote(createSpan(yyStart,yyCount),
                     "’"); }
   / Numbers
   / OtherKeywords
   / OtherSymbols
   / QualifiedName
   / NumericSeparator
   / Op
   / (_ (!w))+
   ;
```

Figure 20: Delimiter checker before parsing Fortress programs

formed into an executable core Fortress program. Then, in the transformation stage, the parsed program is transformed to a program in core Fortress AST without macros which can be interpreted by the Fortress interpreter. The syntactic abstraction mechanism is built on top of the Rats! module system. By separating grammar extensions in their own modules, we were able to extend the core Fortress grammar in a modular way.

4.3 Delimiter Checker

In order to improve syntax error messages for unmatched delimiters, we developed a pre-parsing checker, "delimiter checker." [4] Because Fortress provides both conventional language keywords such as `do` and `end` and mathematical enclosing operators such as $\langle$ and $\rangle$, the delimiter checker should be able to handle them seamlessly. As Figure 20 shows, we maintain a global stack of opening delimiters and closing delimiters using the "stateful" feature of Rats!. The Rats!-generated parser serves as a delimiter checker maintaining the global stack. We first tried adding various error productions into the main Fortress parser to handle unmatched delimiters but a separate delimiter checker turned out to be much more concise and scalable, and provide better error messages.

4.4 Syntax Checker

As we described in Section 3.2.2, it is often the case that precise grammar productions prohibit the parser from recognizing alternative productions. To provide better syntax error messages in such cases, we decided to make the grammar productions intentionally less precise and check various syntax restrictions after parsing the program. We developed a post-parsing checker, "syntax checker" to verify the context-

```
public void forTraitDecl(TraitDecl that) {
    inTrait = true;
    Modifiers mods = NodeUtil.getMods(that);
    if (!Modifiers.TraitMod.containsAll(mods)) {
        log(that,
            mods.remove(Modifiers.TraitMod) +
            " cannot modify a trait, " +
            NodeUtil.getName(that));
    }
    if ( inApi && mods.isPrivate() ) {
        log(that,
            "Private trait " +
            NodeUtil.getName(that) +
            " must not appear in an API.");
    }
    super.forTraitDecl( that );
    inTrait = false;
}
```

Figure 21: Syntax checker after parsing Fortress programs

dependent syntax constraints. [5] As Figure 21 shows, it is implemented as a visitor over the parsed AST to check syntactic restrictions such as the followings:

- Declarations in APIs should not have any missing types.
- Declarations of traits, objects, functionals, variables, methods, and fields in components and APIs should have valid modifiers.
- A getter declaration should not have a parameter.
- A setter declaration should have a single parameter.

4.5 Pattern Matching

After parsing a given Fortress program, the Fortress parser family either reports syntax errors or generates the corresponding Fortress AST. The later phases of the Fortress interpreter/compiler, such as the disambiguation phase and the type checking phase, manipulate the AST. As Figure 21 presents, the most common programming style of manipulating the AST structures in the Java Programming Language is using the visitor design pattern [10]. Visitor patterns are surely better than a series of `instanceof` testing but they are very verbose and do not work well with renamed data structures. For example, if the AST node `TraitDecl` is renamed as `TraitDeclaration` and the first line in Figure 21 is incorrectly changed as follows:
`public void forTraitDecl(TraitDeclaration that) {`
the error is not detected by a Java compiler.

In addition to the Java Programming Language, we are using the Scala programming language [14] in the Fortress compiler development in order to take advantage of pattern matching, among other things. [6] As Figure 22 shows, using

[4]The implementation of the delimiter checker is available at: `http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/src/com/sun/fortress/parser/preparser`

[5]The implementation of the delimiter checker is available at: `http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/src/com/sun/fortress/parser_util/SyntaxChecker.java`

[6]The Fortress static checker implementation in the Scala programming language is available at: `http://projectfortress.sun.com/Projects/Community/browser/trunk/ProjectFortress/src/com/sun/fortress/scala_src/typechecker`

```
def excludes(first: Type, second: Type): Boolean =
    (first, second) match {
        case (SBottomType(_), _) => true
        case (_, SBottomType(_)) => true
        case (SAnyType(_), _) => false
        case (_, SAnyType(_)) => false
        case (SVarType(_,_,_), _) =>
            typeAnalyzer.exclusion(first, second)
        case (_, SVarType(_,_,_)) =>
            typeAnalyzer.exclusion(first, second)
        case (SArrowType(_,_,_,_),
              SArrowType(_,_,_,_)) => false
        case (SArrowType(_,_,_,_), _) => true
        case (_, SArrowType(_,_,_,_)) => true
        case (f@STupleType(_,_,_,_),
              s@STupleType(_,_,_,_)) =>
            NodeUtil.differentArity(f, s) ||
            typeAnalyzer.exclusion(f, s)
        case (STupleType(_,_,_,_) ,_) => true
        case (_, STupleType(_,_,_,_)) => true
        case _ =>
            typeAnalyzer.exclusion(first, second)
}
```

Figure 22: Pattern matching in the Scala programming language

pattern matching for the case analysis of data structures is very concise and provides good error messages in the presence of changes data structures.

5. CONCLUSION

Fortress is a growable, mathematically oriented, parallel programming language for scientific applications. Fortress is originally developed as part of Sun's work on a DARPA project for high productivity computing systems and it is now an open-source project with international participation. It has been our experience that defining the Fortress grammar, with the challenge of the constant language changes, is most naturally done in the context of a formalism supporting unlimited lookahead. As a result, the Fortress parser is implemented by Rats!, which builds on PEGs supporting unlimited lookahead and backtracking, and generates parsers in Java. The Fortress AST node classes are automatically generated by the ASTGen tool, which generates Java source files for node classes. The Fortress interpreter and the compiler (under development) are implemented in multiple programming languages including the Java Programming Language, Scala, and C, and it runs entirely on the JVM.

Acknowledgments

The author sincerely thanks the Fortress team for fruitful discussions and support. Special thanks to Robert Grimm for all his work and help with Rats!.

6. REFERENCES

[1] CUP - LALR Parser Generator for Java^{TM}. http://www2.cs.tum.edu/projects/cup/.

[2] Java Compiler Compiler^{TM} (JavaCC^{TM}) - The Java Parser Generator. https://javacc.dev.java.net/.

[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] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Foundations of Object-Oriented Languages*, 2009.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[7] Gilad Bracha, Guy Steele, Bill Joy, and James Gosling. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

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

[9] Robert Grimm. Rats! – an easily extensible parser generator. http://cs.nyu.edu/~rgrimm/xtc/rats.html.

[10] James W. Cooper. *Java^{TM}Design Patterns: A Tutorial*. Addison-Wesley, 2000.

[11] Stephen C. Johnson. Yacc: Yet another compiler-compiler, 1979.

[12] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System, release 3.08*. http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf, 2004.

[13] Scott McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, pages 73–88, 2004.

[14] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification*. http://scala.epfl.ch/docu/files/ScalaReference.pdf, 2004.

[15] Brian R. Stoler, Eric Allen, and Dan Smith. ASTGen. http://sourceforge.net/projects/astgen.

[16] Bjarne Stroustrup. *The C++ Programming Language (Special Edition ed.)*. Addison-Wesley, 2000.

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