GRAMPA: An Esoteric Programming Language to Simulate Parallel Computing

Contact: [email]@grinnell.edu

Reilly Grant
Grinnell College
[grantrei]

Tristan Knoth
Grinnell College
[knothtri17]

Chris Kottke
Grinnell College
[kottkech17]

Abstract

Parallel computing is an important way to process over very large problems. However, developing parallel applications is extremely difficult. We develop Generalized Rick And Morty ProgrAmming (GRAMPA), an esoteric imperative programming language supporting a simple forking model in order to simulate parallelism and introduce students shared memory and other rudimentary parallel computing concepts. The language supports only a limited set of commands, and includes syntax is based on the popular cartoon Rick and Morty in order to present parallel computing concepts in a fun and accessible manner.

1. Introduction

As parallelism becomes the most important paradigm for large-scale information processing, developing parallel applications is becoming an increasingly important skill for any developer. Students who begin to think about splitting problems up between processors earlier in their computer science education will potentially see more success learning more sophisticated parallelism paradigms later on. To this end, the popular TV show Rick and Morty presents the perfect medium through which to introduce students to rudimentary parallelism concepts. We develop GRAMPA, a simple Turing Complete imperative language with syntax based on references to Rick and Morty that supports a simple model of forking across shared memory. In Rick and Morty, the main characters travel between dimensions. The show's clear conceptual connections to multithreading may help alleviate the pain of learning to parallelize simple algorithms. GRAMPA's syntax is intended to look something like written natural language.

2. Prior Work

GRAMPA relies on a number of existing technologies, particularly the Haskell Parsec library. Our parser is built in Haskell using Parsec, which allows users to combine parsers via monads.... Didn't really know what else to put in this section.

3. Examples

4. Parsing GRAMPA

Using Parsec, we can build a recursive descent parser for GRAMPA by combining parsers for different types of expressions and statements in our language. Given a grammar for our language, we can define parsers for each variable in the grammar and thus recursively parse the entire language.

Grammar

The following Context-Free Grammar defines the GRAMPA syntax, and indicates the parsing hierarchy. The variable STRING refers to any string consisting only of chars, and INT refers to any integer.

$$
\begin{aligned}
S &\rightarrow UNIV \\
UNIV &\rightarrow \text{universe } STRING\ STMT \text{ destroy universe } \mid UNIV\ UNIV \\
STMT &\rightarrow PORTAL \mid IF \mid DECL \mid PRINT \mid WHILE \mid STMT\ ST \\
EXPR &\rightarrow OP1 \\
OP1 &\rightarrow AND \mid OR \mid OP2 \\
OP2 &\rightarrow NUMEQ \mid NUMLT \mid NUMGT \mid OP3 \\
OP3 &\rightarrow ADD \mid SUB \mid OP4 \\
OP4 &\rightarrow MUL \mid DIV \mid MOD \mid TERM \\
TERM &\rightarrow BASE \mid PARENA \mid PARENB \mid STRING \\
BASE &\rightarrow INT \mid BOOL \\
BOOL &\rightarrow \text{right} \mid \text{wrong} \\
PORTAL &\rightarrow \text{lets grab our } STRING \text{ and portal out of here} \\
IF &\rightarrow \text{if } OP1 \text{ then } STMT \text{ otherwise } STMT \text{ wubulubadubdub} \\
DECL &\rightarrow STRING \text{ means } EXPR \\
PRINT &\rightarrow \text{show me } STRING \\
WHILE &\rightarrow \text{while } OP1 \text{ do this for grandpa } STMT \text{ thanks Summer} \\
AND &\rightarrow OP2 \text{ and } OP1 \\
OR &\rightarrow OP2 \text{ or } OP1 \\
NUMEQ &\rightarrow OP3 \text{ is the same as } OP3 \\
NUMLT &\rightarrow OP3 \text{ is less than } OP3 \\
NUMGT &\rightarrow OP3 \text{ is greater than } OP3 \\
ADD &\rightarrow OP4 \text{ plus } OP3 \\
SUB &\rightarrow OP4 \text{ minus } OP3 \\
MUL &\rightarrow TERM \text{ times } OP4 \\
DIV &\rightarrow TERM \text{ divided by } OP4
\end{aligned}
$$

$$MOD \rightarrow TERM \text{ mod } OP4$$
$$PARENA \rightarrow \text{you gotta } OP3 \text{ Morty}$$
$$PARENB \rightarrow \text{you gotta } OP1 \text{ Morty}$$

Given the context-free grammar above, we define parsers for each of the individual substitution rules. For example, to parse a multiplication, we look for a $TERM$ on the left side of the expression, a ”times” to indicate that we are multiplying two expressions, and an $OP4$ on the right hand side of the expression. In Haskell, this is implemented as follows:

Algorithm 1: Multiplication Parser

```
whitespace
e1 ← termParser
whitespace
string ”times”
whitespace
e2 ← op4Parser
whitespace
return $ EBin Mul e1 e2
```

The sequence of instructions above is wrapped in a ”do” block to create a parser for multiplication. `whitespace` is a parser designed to consume all whitespace. As such, our language is completely whitespace insensitive as long as one separates commands by any amount of whitespace. `string` is a parser built into Parsec, which parses a specific string. In this way, we combine parsers for different types of expressions and recursively parse the entire document.

Parsec also gives the user the option to ”try” a parser. If one uses ”try” when calling a specific parser, the parser will only consume input if it successfully parses the entire expression. Thus, if it encounters an error it will attempt again to parse the exact same string for a different type of expression. This feature allows us to combine parsers for different types of expressions quite easily. For example, we parse statements, which have a number of forms, in the following way:

```
stmt = try sPortal <|> try sIf <|> try sDec <|>
try sPrint <|> try sWhile
```

Thus, if the parser does not find an ”if” statement, it can move on to look for a variable declaration or a print statement, for example. ”Trying” different parsers in this way allows us to combine parsers like the multiplication parser described above and recursively parse the entire context-free grammar.

As a final note, one should observe that our order of operations is implicitly embedded in the grammar, and therefore the parser. The relevant portion of the grammar begins with the $EXPR \rightarrow OP1$ substitution. Here, we parse operations in ascending precedence. We parse the terminals and parenthesized expressions last in order to ensure that they are leafs in the parse tree and are therefore evaluated first.

4.1 Left Associativity

Parsing a file using the process described above results in a right-associative parse tree. This means that the expression $1 + 2 + 3$ is parsed as $1+(2+3)$ rather than $(1+2)+3$, which is what one would expect. In many cases this difference is nontrivial. For example, we need to ensure that $16/4/2$ evaluates to $(16/4)/2 = 2$ rather than $16/(4/2) = 8$. To rectify this problem, we transform the abstract syntax tree generated by our parser to reorder operations and guarantee left-associativity.

5. GRAMPA Code Generation

6. Parallelism in GRAMPA

7. Conclusion

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...