CS 4850 Section 03
Spring Semester 2023

Pint Programming Language

SP-19 Design and Develop a Programming Language

Project Members: Nicholas Luchuk, Manjote Singh
Project Owner: Sharon Perry
April 29, 2023
https://pint-plang.github.io/
Video Presentation: https://youtu.be/b2DyNZ-iNU0

Table of Contents

1. Abstract

The goal of this project is to design and develop a programming language called Pint. This will be facilitated by using ANTLR4 to generate a parser. Pint will use an innovative concept called "type conditions" to be efficient at run time without sacrificing safety at compile time. With type conditions, Pint will prevent invalid or unsafe abuses of array indexing, function calls, and similar constructs by enforcing boundary checks, input validation, and other restrictions.

2. Version Control

We used GitHub to manage this project.

The main repository is here: https://github.com/pint-plang/pint-lang.

3. Project Report

The Project Report will describe what we did, how we did it, what we learned, and the challenges we faced.

3.1. Design

From day one we began designing by asking the question "What kind of language do we want to make?" and elaborating on that. For instance, the first thing we asked was for each member to come up with something he likes about existing languages or wishes were in existing languages. After some discussion, we settled on the idea of type conditions as a feature that

satisfied both of our ideas. Type conditions are essentially extra restrictions that can be applied to types; for instance, a percent may be represented as an integer that must be between zero and one hundred, and with type conditions, the compiler can enforce that relationship. Type condition can be imposed on variables by a feature called type narrowing: in regions of the code where a variable can be verified to meet a type condition, the typechecker "narrows" its type to consider it to have that condition. For instance, in the body of if x > 0 then body, the type of x gets narrowed to int when it > 0.

After we settled on type conditions, we started working on the grammar. We thought about what functionality we would need, and then asked what we wanted it to look like. For instance, we knew we would need conditional branching, so we asked what we wanted that to look like, and settled on `if condition then do one thing else do another`, and consequently the grammar rule <ifExpr> -> 'if' <cond> 'then' <thenBody> ['else' <elseBody>]. We also wrote down the detailed semantics (that is, behavior) of each grammar rule, and tried to cover all the edge cases so that implementation would be as simple as possible. That said, when we did end up implementing them, we found many more little things that we had not considered, but which did not pose significant setbacks.

While a few small things changed during implementation, there was one major design flaw we ran into that we had to fix. Originally arrays were mutable, meaning that you could change individual elements of them without creating entirely new arrays; however, this allowed users to change the properties of arrays that might be restricted by type conditions, such that the type conditions would become false without the compiler knowing about it. Our solution was to make arrays immutable, meaning that they cannot be changed without making new arrays. To offset this heavy restriction, we borrowed features called slicing and spreading from other languages to allow expressive ways to manipulate arrays even without mutability. We will not explain slicing or spreading here, but as they are existing ideas in other programming languages, they should not be hard to research.

3.2. ANTLR

From previous research in the subject, we were aware of the ANTLR tool. ANTLR specifies a format for language grammars, and given a file of a grammar in that format, will generate a lexer, parser, and concrete syntax tree (CST) based on that grammar. Knowing this, our first step towards implementation was to translate the grammar we had constructed to the format specified by ANTLR, and in so doing, we got the first half of the code for free: ANTLR has proven more than adequate, and as such we have had no reason to give almost any thought to the lexer or parser (though for various reasons, we did end up making our own abstract syntax tree (AST) to use on top of the ANTLR's CST). The one consideration we did have to give was learning how ANTLR resolves grammatical ambiguity: lexically from top to bottom in the grammar file. As we learned this before running into any issues with ambiguity, the lexer and parser never posed any problems for us.

3.3. Interpreter

The interpreter was originally intended to only exist in the prototype, but we ran out of time and ended up leaving it as-is. As such, it is much less polished than we would like, but it does function essentially as intended. The interpreter operates by performing recursive descent on the CST, which is mostly a post-order traversal. When traversal reaches a most nodes, it first evaluates all their children (hence post-order), and then calculates a result for itself; for instance, if an addition node, `a + b`, is evaluated, it first evaluates `a` and `b`, then it adds their results and returns that is its result. Where the interpreter differs from post-order is that some nodes may have more complicated traversals; for instance, an if node, `if c then b`, evaluates `c` and then only evaluates `b` if `c` is true.

Of special note is that the interpreter does minimal type checking: it essentially acts as though Pint were a dynamically typed language, and will allow certain programs that are not

considered valid for Pint. For example, the interpreter will evaluate the expression `|if c then 42 else "Hello World!"|`, and will return either `42` or `12` depending on the value of `c` (note that `|x|` returns the absolute value if `x` is an integer or the length if `x` is a string).

3.4. Typechecker

The typechecker runs before the interpreter and verifies that all the types are correct. If the typechecker fails, the interpreter does not run, thereby preventing illegal scenarios like the one mentioned in the interpreter section. Originally, the typechecker and interpreter were being developed at the same time, but the typechecker took significantly longer than expected, and was the main source of delays for the project.

We deliberately excluded type conditions from the initial version of the typechecker, because we knew they would be the most complicated part and we wanted to get the easy part done. It took longer than expected, but otherwise went as planned. We then tried to add type conditions into the existing typechecker. However, afterwards when we were adding finishing touches, we found a fatal bug and had to scramble to redo the condition checking from scratch.

The typechecker functions similarly to the interpreter, but with some important differences. Firstly, the typechecker operates on the AST rather than the CST so that we could embed the type information into it for later use (note that, because we're still using the interpreter, it's not actually getting used later). Secondly, the typechecker does a proper post-order traversal, because we need to check the type of all code, not just the code that will be run by the interpreter. Finally, rather than evaluating the tree, it just determines each node's type; the main difference is that functions do not actually get called and loops get checked exactly once, rather than zero or more times.

In order to implement type conditions, we had to implement two things: type narrowing, and checking when a verified condition meets a required one. Type narrowing was relatively

simple: we were already keeping track of what variables exist and what their types are, so we just found all the variables in each if or while condition, and modified their types to meet that condition inside the appropriate scope. The condition checking was more difficult, and was the part that we had to redo. Originally, we essentially just kept a copy of the AST and checked for equality; however that had some significant flaws. Now, we have a modified version of the AST that replaces variables with "inputs," that could represent variables or other expressions that might meet a condition. This allows the typechecker to recognise different ways of writing a given condition as representing the same condition; for instance, it can tell that `x < y` is the same as `y > x`.

Because of the difficulties posed by type conditions, we have concluded that we should have started by implementing the type conditions, and only built out the rest of the language once type conditions were working soundly. This would have allowed us to anticipate problems in the rest of the language (such as the mutability issue); additionally, even if we had not been able to get a whole language working, at least the most important part would also be the most complete, rather than being something we scramble to get working at the last minute.

3.5. Incomplete Features

There were a number of features in various stages of development that we did not end up having time to add. For instance, we originally planned to have a small but mostly complete standard library, containing functions for string manipulation, file I/O, math, and a few other miscellaneous tasks. This would also have served as an excellent example of how users would interact with type conditions.

As mentioned above, the interpreter was originally intended only to be part of the prototype. We had planned to make the final product a compiled language by means of the LLVM project. LLVM is a compiler infrastructure tool that translate abstract semantics set out by

the user (in this case, us) into machine-native binaries, e.g. .exe on Windows. Work began on integrating LLVM before the typechecker was finished, and there is a branch in the repository with a partially working version, but due to the complication with the typechecker, we had to postpone indefinitely, and we never got back to it.

We also had a few stretch goals that we knew we might not get to, such as optional types; for instance, `int?` could be any integer value or `null`; it could only be used as an `int` by narrowing its type, thereby enforcing null checking (while still circumventing the need for any kind of NullPointerException). We also briefly considered making Pint object oriented, which would make implementing Kotlin-like smart casting trivial via type narrowing.