CPLR

An exploration in compiler construction

Contents

1 Introduction

CPRL is an exploration in the theory and practice of building compilers. Compiling a subset of the Ada programming language is merely an artifact of an application of ideas explored and implemented by CPRL. In fact, most of the implementation deals with issues that are independent of both the source language and target platform (referred to in GCC parlance as the target triple or more recently the target quadruple http://gcc.gnu.org/ml/gcc-help/2003-08/msg00169.html[1]). One of our hopes is that supporting new languages will actually be relatively easy and supporting new platforms will not be an excessive amount of work[2].

1.1 Design

Several design decisions pervade CPLR:

True to UNIX culture, every single component of a system should only serve one purpose and do that task really well. This is emphasized in “The Art of Unix Programming” by Eric Steven Raymond http://www.faqs.org/docs/artu but has clearly been articulated by countless more beginning with Ken Thompson’s vision of a “small but capable system.”

The bulk of CPLR is not the compiler itself but the set of libraries that are useful in building a compiler. The idea of producing a set of libraries useful for building compiler like tools rather than a monolithic compiler is heavily influenced by Chris Lattner’s graduate work that has culminated in the LLVM project http://llvm.org/[3]. Viewed from a more of a pedagogical angle, many of the concepts that emerge into what we call a compiler can be well explained, explored, and understood in isolation or abstraction[4].

These design philosophies afford many advantages. As mentioned earlier, new languages and new platforms are relatively easy to support. Exploring non-traditional compilation pipelines,

[1] The CPU_TYPE part of the GCC target specification is still not enough to disambiguate among instruction set architecture subtleties that may be of interest to a compiler’s code generator. Suppose a compiler is given a target of i686-pc-linux-gnu. Should the compiler emit code that uses the EMT_64 extensions? Is OK to use SSE3 or limit the optimizations to SSE2?

[2] With modern CPU architectures making heavy use of out-of-order execution there is much onus placed on the compiler to allocate registers and order instructions such as to maximize the processor’s resources. The burden placed on the compiler means that only so many optimizations can be performed in an architecture-agnostic fashion.

for example parser-less compilation, is also easy. GCC's single mindedness is a source of frustration for many programmers and the driving factor behind many ad-hoc tools; even though GCC has intimate knowledge of many languages not much can be done with it other than compiling programs. An Ada compiler is but one application of the CPLR libraries; possible applications need not even be compilers at all. A few examples include syntax highlighting or live marking of incorrect code in an editor.

That being said, CPLR is not exhaustive in its implementation of concepts. For example it makes little sense to implement more than one type of LL, LR, SLR, or LALR parser. Because of the separation of concerns though, if a reader desires to know more about a different form of parsing it would be rather straight forward to implement a different parser variant while still re-using the test suite and all other components as is. Likewise you may not even want to use Ada as the source language.

Mentioning sources of influence on compiler design without mentioning GCC is hard to do. GCC deserves credit in what it has accomplished socially for the free software movement. GCC's primary influence on CPLR is that a single compiler driver can be used for various source languages. Though CPLR only implements a subset of Ada, nothing in its design restricts it to a single source language. Beyond that, there are many aspects of GCC that can be improved upon. For example, there will be no attempt at trying to be compatible with GCC's command line options because most make no sense.

1.2 Python

CPLR is implemented in Python. We will try as much as possible to adhere to the Python coding standards - "PEP 8 -- Style Guide for Python Code" http://www.python.org/dev/peps/pep-0008/.

Python presents several advantages both practically and pedagogically. Python is a very rich language and comes ready with many data-types and programming constructs. Most of the Python style has evolved embracing a sense of flexibility of expression. You are not locked in to any particular style, for example object-orientation with Java, but rather can use whatever

[3] Chris Lattner is a very timid and humble person and would probably shy away from being named next to Ken Thompson and Donald Knuth. Nonetheless, his work has played important role in shaping current trends of compilers and compiler hackers in the wild.

[4] The human mind has a very limited short term capacity. The primary mechanism for dealing with a crippling amount of short term memory is abstraction. The classic example of this is phone numbers, which are much easier to remember than arbitrary 10 digit numbers because of a psychological phenomenon called chunking. Chunking is just an application of our mind's ability to abstract structure. The structure of phone numbers originated from work done at Bell Labs as a result of George A. Miller's well known 1956 paper "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information." To give another example, people have no trouble driving cars by pressing a pedal and turning a wheel while never giving an shred of thought to how those components affect the dynamics and chemical reactions involved in "actually" driving a car.

style is most practical. In a compiler there are modules that are well suited to be described as objects while others are purely procedural and others still are purely algorithmic in nature and are well suited to functional styles of expression. Since Python runs on many platforms writing a compiler in Python inherently means you are writing a cross-compiler. Furthermore, Python code can have both a prosaic and mathematical expressiveness to it, which is very well suited for conveying concepts.

Python doesn't come without its drawbacks though. Notably execution speed can possibly be an issue since Python is interpreted. It remains to be seen just how serious of an issue it turns out to be since there is always the option of writing performance critical sections in C. More interesting to the topic of compiler construction, perhaps, is that writing a bootstrapping compiler, for Ada in the case of CPLR, is not possible with Python.

1.3 Literate Style

CPLR is even less of a compiler than it is a document explaining various concepts at the core of computer science. Some argue that students do not really understand computer science until they understand compilers (http://steve-yegge.blogspot.com/2007/06/rich-programmer-food.html), this document merely reviews many topics core to computer science by applying them to a compiler. How better to explain concepts than through narrative[5]. For the purpose of example, the concepts are applied in this document as a compiler for a subset of the Ada programing language. While CPLR is not written in a traditional literate programming style it is influenced by Donald Knuth's idea of the program as a narrative http://www-cs-faculty.stanford.edu/~knuth/lp.html. You may be reading this text directly in the source tree, in which case the code is a live, runnable artifact. Likewise, you may be reading this text in a compiled document, in which case the document contains everything you need to create a working compiler for the subset of the Ada language we are interested in[6].

CPLR differs from traditional literate programming style in two significant ways. First, CPLR is written as prosaic source and later converted into a narrative document rather than as narrative document woven into a compilable source code. On many levels, it makes more sense to lay out a source tree with files that mimic the conceptual modules and object hierarchy of the system. Second, conversion from source to document is kept to a minimal amount of fuss. There is no code "weaving" or code block rearranging. The source code tree is walked in a manually defined order. Each file is included as a whole. Of course, source code and comments are prettied up so that the document reads more like a document than a listing of source code. Since this document represents the entirety of the source tree and this document is generated by code

[5] Oral tradition and narrative are as old as humanity itself. Possibly the only better learning tool is experience itself unless of course one doesn't have enough experience to start doing.

[6] In the practice of giving credit where it is due, the desire to produce a document that a compiler can be crafted from was also heavily influenced by "Physically Based Rendering: From Theory to Implementation" by Matt Pharr and Greg Humphreys when observing first hand what an incredible learning tool a text written in literate programming style can be.

found in the source tree, you guessed it, there is a chapter that details how this is done.

Now, follow along as we build a compiler[7].

[7] Due to time constraints the entire compiler wasn't in a state that lent itself to conversion into a narrative document in time for submission. The rest of this document will contain some text extracted from some of the more polished sources. Where certain source modules didn't have proper documentation and formating they are simply described.

2 Through The Looking Glass

2.1 #/cplr:

"You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes." *Morpheus*

Our exploration begins top-down. We will never be very far from the Ada compiler example; never too deep into theory.

The main entry point is nothing more than a simple wrapper script. The wrapper is only responsible for setting up the environment so that all the CPLR libraries are found by the Python interpreter and handing execution off to the CPLR driver.

Design:

- The cplr script is the only component that is concerned with launching the compiler.
- The cplr wrapper script is the only component that has a UNIX bias.
- There is the added benefit that Python caches the bytecode of modules after they have been interpreted providing faster subsequent startup time. This would not happen if we invoke the main driver directly as a Python script.

Since the rest of the system is written in Python when porting CPLR to another platform the only thing that needs to be changed is this wrapper script.

```
cd ‘dirname $0‘
export PYTHONPATH=./src/lib:$PYTHONPATH
```

It may be useful for debugging or deployment pruposes to control how the python interpretter is invoked. Options can be set in the PYTHON_OPTS enviroment variable or hard coded in this script for deployment.

```
PYTHON_OPTS="$PYTHON_OPTS "
```

By default CPLR prevents all python exceptions from reaching the interpreter and instead prints error summaries when unnexpected situations arrise. When debugging CPLR it may be useful to get extra information about abnormal behaviour and raised exceptions. Simply set the the CPLR_DEBUG environment variable to any value, for example: `export CPLR_DEBUG=1`

```
python $PYTHON_OPTS src/apps/compiler/driver.py $*
```

2.2 Source-tree layout

At the top-level things are fairly straight-forward.

The two main scripts of interest are `cplr`, which invokes the compiler proper and `run`, which provides infrastructure tools like running the tests.

Any file with the extension ".txt" can be assumed to be reStructuredText suitable for processing by docutils.

`#` - Refers to the project root (the directory that contains this file)

`#/src` - Contains all the project sources. More explanations are available for each of the components.

`#/doc` - Contains documentation artifacts that get included in the final composition and directives on how to assemble the documentation from the source.

`#/build` - Where built artifacts get placed. For example the PDF of the final composed documentation. The directory will never be placed under version control since its contents can always be re-created from the source tree contents.

`#/toolchain` - A self contained environment. It will contain any third party tools or libraries needed to run CPLR itself or generate the documentation. By including the toolchain under version control anyone who checks out a working branch immediately has a working environment without needing to install software on their systems.

3 Design Specifics

Time to dive into the specifics of the CPLR design. One of the advantages of using Python as a source language is that is provides for a pseudo-bootstrapping experience. In particular the lexical and syntactic definitions are expressed in Python and are handed off to factory methods that can create the DFAs for the lexer and parser on the fly.

3.1 Patterns

I won't go into great lengths about all the design patterns used but will just mention some in passing.

Abstract factories are used to separate the language specifications from the other libraries. The Ada specific considerations are completely contained within the Ada front end specification. Components like the longest match lexer and the SLR parser know nothing about Ada and could just as easily be created using different language specifications.

The visitor pattern is very handy for tasks syntax directed translation. Each task can be written as it's own visitor that traverses the parse tree however it pleases. Once the definition of the abstract syntax tree nodes is complete there is no need to add to it if ever a new type of traversal is required. A new visitor can be implemented. An abstract implementation of the visitor pattern in Python can be found in `#/src/lib/util/visitor.py`. A specialized, abstract visitor for traversing abstract syntax trees is implemented in `#/src/lib/compilertools/ast_visitor.py` Concrete visitor implementations will be discussed later when talking about abstract syntax tree traversals and transformations.

In a pure object oriented paradigm a pattern called the command pattern was devised to work around the need to use functional units. The command pattern requires objects to implement a method of a certain type / name. This can either be done by convention in a dynamic typed language or enforced at compile time, for example using a Java Interface. In Python, however, everything is an "object" including functions. Any user defined type can implement the `__call__()` method and it will be called when an instance of the object is used in the context of Python's function call syntax. This makes the call sites of objects that implement this version of the command pattern very elegant since they appear to be calling functions. The only distinction is in the implementation itself. This will come in handy with translator objects

3.2 Translator objects

As hinted at previously, the overall design of a compiler can be modeled as a series of transformations. Each transformation consumes something that it considers its source and produces a result. The nature of compilation, thus lends itself to a pipeline architecture - much like Unix pipes and processes. In particular we've defined what we like to call a translator object pattern. In Python objects are strongly but dynamically typed and behavior is often defined by convention. For example, a file-like object is a Python object that, among other things, implements the `read()` and `write()` methods.

In the case of the translator object the convention is that it should be callable with a source object and return a result. (i.e.: it should implement `__call__(self, source): return (result, issues)`). A translator object is a functional unit that maps a source to a result. The `source` can be whatever type a concrete translator expects. For example, the ByteScanner expects a file-like object, the SLRParser expects a stream of lexical tokens. The only restriction on the `result` is that it can be converted to a string representation. Why this is crucial will become apparent shortly. Finally, `issues` is a possibly empty array of objects that represents problems that might have arisen during the translation. This flexibility of only expecting things by convention is great when prototyping since none of these objects are expected to inherit from specific base types. As the design hardens and the requirements become more well-defined it might be sensible to enforce stricter type policies.

You'll notice that no restrictions were placed on what an `issue` is. This is left up to each individual caller and call-ee to agree upon since they are the only entities in the system that need to produce and interpret the meaning of each issue. If the caller only cares about printing the issue for the user then the only thing that really matters is being able to convert the issue to a human readable string. There is a distinction made between fatal and non-fatal issues - analogous to compiler errors and warnings. If a translator returns `None` then it is considered to have encountered a fatal issue during translation. Note that this is very different returning an instance that evaluates to the empty string, such as the object code for an empty compilation. As mentioned before, this looseness in the conventions is very helpful when doing an early prototype. As it stands right now, phases of the CPLR pipeline return instances that derive from the error types found in `#/src/lib/compilertools/error.py`.

3.3 Pipelines

Taking a step back though, it makes sense to combine translator units into sequences. This is implemented by `#/src/lib/util/pipeline.py`. You'll quickly notice that because of the loose conventions, a pipeline is also a translator object. Translator objects can thus be defined recursively. The entire compiler is really nothing more than a translator object itself. The recursiveness stops in our design with the Compiler object.

If the pipeline encounters a fatal error as it's evaluating each translation phase it must also return a fatal error. In particular a pipeline will short-circuit instead of evaluating the remaining phases and will return None. This is helpful to prevent a cascade of errors being printed to the user.

3.4 The compiler proper

The idea of separating the compiler into a set of libraries has paid off well. What could be considered “the compiler” is nothing but glue code, a driver, and argument parsing and can be found in `#/src/apps/compiler/`. Another way to view this is that the `compiler.py` module provides the convenience of knowing how to bring the libraries together in just the right way and the `driver.py` module provides the convenience of wrapping the compiler object in a form that can be invoked on a command line. It can probably be argued that `compiler.py` should be part of the `compilertools` library.

The `Compiler` object itself though is still just a translator object in disguise; it has been given a bit more of a structured API so that a user can control the compilation but it still takes a source and returns a result that can be converted to a string representation. Then `driver.py` simply needs to write the string representation of the result to the output file.

3.5 Utilities

The `#/src/lib/util` package contains convenience routines where Python lacks certain expressiveness. The two most significant utilities are the Visitor and Pipeline implementations. The rest is fairly uninteresting in the context of compiler design.

3.6 Language tools

If we were to view the CPLR architecture as a layer diagram, the modules in `#/src/apps/compiler/` would sit at the very top, `#/src/lib/util/` would be at the very bottom. In the middle you would find `#/src/lib/lang/` upon which the modules in `#/src/lib/compilertools/` are built.

The `lang` package is an attempt to separate the formal language tools needed to build a compiler from the issues of compilation proper. A nice example would be the implementation of lexical analysis. The `lang/reg` package is concerned purely with regular language issues. In particular, it can build DFAs out of the regular composition operators (symbol, concatenation, alternation, and kleene-closure) along with a few convenience operations. This is used directly by `compilertools/stages/_languages/ada/syntax.py` to build a set of DFAs that can recognize the Ada83 lexical elements. Finally everything is brought together by the `compilertools/stages/_languages/ada/anaylis_ada.py` module when it creates a LongestMatchLexer (`compilertools/lex/lexer.py`). The lexer really only cares that it's created with a list of `DFA X Type` pairs. Upon matching a particular lexical element the right token type is instantiated. In other words the `reg` package only deals with recognizing regular languages. The lexer has the responsibility of behaving like a transducer consuming characters and emitting a token stream.

A similar split exists between aspects of context free grammar and parser generation versus the actual parser, which is more of a driver that iterates over the token stream using the parse tables. The split is a little blurry in its current implementation and that could be cleaned up a little.

3.7 The front end

What makes up an Ada83 front end is completely specified in the package `compilertools/stages/_languages/ada`. In particular the `analysis_ada.py` module within that package is nothing more that a factory that builds the right components needed for the Ada front end using the Ada specific definitions. For example, the DFAs that make up the lexer could have been built with any set of expressions.

3.8 Of bytes and scanners

The idea behind `compilertools/scanner.py` was to abstract a few things related to file IO away from lexical analysis proper. The ByteScanner is the most trivial example of this. The scanner can validate the input character stream to make sure, for example, that only the valid range of Ada83 characters are present. It also means that the lexer doesn't have to be concerned with seeking through a file and determining positions of tokens. There are many examples where this might come in handy - I will discuss two. First, the input may not be in a file at all. In the spirit of creating a library of compiler tools, a user might very well want to invoke the lexer on a string or character array. This shouldn't require a change to the lexer. Second, once the input is passed the scanner you can make the assumption that everything is in unicode. While this isn't an issue with Ada it is with other languages and in particular, makes the interpretation of string literals much easier. In multi-byte representations there isn't a one-to-one correspondence between file seek position and character count. The scanner abstracts character position away and makes error reporting of multi-byte source files easier.

3.9 Lexer

The lexer has already been discussed in previous sections so I'll only mention it here. We decided to implement is a longest match scheme, using priorities to disambiguate when there was a tie. Priority is specified by order in the list of DFAs used to create the lexer. For Ada this simple approach turned out to work fine since Ada is very strict about delimiters. The main place where priority mattered was in preferring reserved word tokens over identifier tokens. No "hacks" were needed in the lexer or syntax definition proper. For more difficult to parse languages this may not be an option. Instead, shortest match may be a more viable alternative if we wish to keep a clean separation between compilation tools and language specifications. This is considerably more involved to implement correctly as the token stream needs to be handled in a non-destructive way and markers added to the lexical specification.

The lexer deserves a word about Python's performance. For the size of programs we were compiling in our tests execution speed was not an issue. The only time speed becomes a major issue is in the NFA to DFA conversion of the automaton that recognize the language elements. We complete the full conversion up-front rather than on demand as certain tools like gnu-grep do. This means that building the full lexical definition can be time very consuming. We work around this by caching each DFA that's built.

3.10 Parser

For the purposes of syntax analysis we implemented an SLR parser and parser generator. Without having a lot of experience writing grammars it's hard to say if the simplicity of SLR cost us in terms of writing an SLR grammar and if we would have been better off writing a LALR parser. In other words we just don't have the experience to subjectively quantify if, for a language like Ada, writing a LALR grammar is significantly easier than writing an SLR grammar. That being said there are more complicated languages like C that require an LR(1) table construction. For example, parsing an assignment statement that involves pointer dereferencing needs the extra lookahead.

That being said, LR parsing schemes in general, in our opinion, feel more natural as they are operating on the token stream. In the case of our SLR parser, a parse tree is easily built as tokens are shifted and rules are reduced. The algorithm is very straight forward (`compiler-tools/parse/slr.py`). The only concrete argument that we've found in favour of LL style parsing algorithms is that a parser generator can generate code in a recursive decent format. The generated code is arguably easier to reason about than the tables generator by an LR parser generator. While this is true, you are still emitting *new* code that is possibly prone to errors and must still be compiled or interpreted. This almost seems like the antithesis of abstraction. Conversely, in the LR scheme the code responsible for generating the tables and driving the syntactic analysis is set and can be well tested. There is little chance of a grammar introducing an error in the parser. Likewise, the hierarchy of elements can still be inspected either by looking at the grammar itself or by looking at the definitions for the parse tree nodes (`compiler-toos/stages/_languages/ada/nodes.py`).

Again, just like with the lexer, the parser itself doesn't inherently know about the language it is parsing. It is just consuming tokens and reducing the input by creating new node instances. What parse tree node types to create is completely specified by `_languages/ada/grammar.py`.

Finally, one last word about parsing. The ada specific nodes are defined in such a way that they automatically create an abstract syntax tree as they are instantiated. This meshes well with the bottom up nature of LR parsing.

3.11 Error reporting

Reporting issues to the user has been touched on just in passing so far. In the current implementation issues within the compiler libraries are very lightweight. The `compilertools/error.py` error types are sub-classed to be more specific for example a `SyntaxError` in `compiler-tools/parse/slr.py`. Errors themselves, in most cases, represent nothing more than two seek positions in the original input stream. There is no assumption made that, for example, an error will be destined to be displayed to the user in a certain way. Callers can then interpret issues, possibly re-reading portions of the source buffer.

Error reporting puts a spotlight on the duality between the compiler as a tool that is meant to quickly generate correct code from correct sources (eg: the compiler in a build farm) and the compiler as a user facing tool that is meant to be invoked iteratively as a program source is being developed. In other words we see a design conflict between throughput and responsiveness akin to the design tradeoffs that must be made in an operating system.

In the case of CPLR, the design takes the middle ground. Issues are tracked in a minimal way while the compilation is proceeding. They contain enough information to be elaborated on in the event that a) errors arise at all in the compilation and b) the application of the compiler desires to elaborate the errors.

In particular the compiler libraries end up being fairly memory efficient. Once the input has been tokenized the entire source is jettisoned and the tokens simply retain offset information about where they where originated. Similarly, syntactic nodes can determine where they originated in the source by examining the extents of the tokens they are composed of.

An attempt was made at a more sophisticated error reporting mechanism. In particular, since we originally started with an SLR parser we attempted do construct a simple bounded context style parser that is capable of correctly reporting all errors before bailing out (this is based on the paper "An LR Substring Parser for Noncorrecting Syntax Error Recovery"). We were not able to make this work properly in all cases so the compiler currently uses the SLR parser and dies after the first incorrect token. The source for the simple bounded context parser can be found in `compilertools/parse/sbc.py`. Once this module actually works properly, the only change required to use it is to switch the parser constructor in `compilertools/stages_languages/ada/analysis_ada.py`. In a lot of cases, reporting the first incorrect token produces error reports that are understandable. They do get a little confusing, however, in certain cases. For example a procedure with an empty body will report an error at the `end` token:

```
procedure hello is
begin
   -- an empty body
end hello;
```

No attempt was made at error recovery in the form of source program correction. We might even posit that the value of such a technique is arguable. You are in fact allowing an incorrect sources to be compiled an run. At the very least if such a transformation is made a warning should be issued in place of where the error would have been reported. Other than in the very early stages of prototyping new code, it is unclear whether such a type of user interface is desirable. Unless the group of people working on a project have a strict policy of turning all warnings into errors before checking in code this can lead to programmer laziness and portability problems down the line. A good example of this is how GCC reports certain C or C++ errors as warnings and the problems that arise when the same code base is used to build a product using Microsoft's VC compiler. Where source correcting has a clear advantage is in the context of and editor or IDE. Again, the specifics of the design interaction in such an interface must be very clearly thought out and very tastefully implemented. Thoughts of Microsoft Word's auto-correct feature spring to mind.

3.12 Attributes, types, and symbols

Every transformation up to now has been for two reasons. First, is to make sure the program is syntactically correct. Second, is to get the program into a form that makes it easy to start making assertions about the validity of the program. To make an analogy with logic, once we reach this phase we know that the program is well formed but we don't yet know if it's valid.

To call this phase “semantic analysis” is a misnomer. The compiler will never actually create any form of in-memory representation that it can use to infer the meaning of each statement on the program and its effect on the system. That being said, the abstract syntax tree does capture the full meaning of the program but doesn’t try to make sense out of it. An apt analogy would be Searle’s Chinese Room thought experiment - the meaning of the program is never destroyed or altered but we always have a very local context. Much could probably be said about this meta-linguistic awareness and the philosophy of natural languages.

One way to view the program in the AST state is to think of the internal nodes as capturing the behaviour of the program while the leaf nodes (tokens) provide the parameterization. We need to traverse the tree to ensure that the program says something meaningful rather than syntactically correct gibberish.

In particular, we must check that the types of expressions agree, extract values out of tokens like string and integer literals, and ensure that symbols that are used exist in the current program scope.

This phase is implemented as a series of visitors in `compilertools/stages/_languages/ada/`: `symbol_collector.py` and `type_constructor`). Each visitor traverses the AST and annotates nodes with attributes as needed. It’s worth noting that this isn’t the first traversal of the AST we’ve completed so far (and it isn’t the last either). The actual act of building the AST is of course an inherent traversal. In fact the entire structure of the AST is actually captured by node attributes - each node has a dictionary of child nodes. Because there is so much in common between finding symbols, determining symbol scope, and inferring symbol types; it doesn’t make much sense to separate these into their own visitors.

One nice thing about this design is that it allows the compiler to sanely report what parts of Ada are supported. The grammar and parser might be written to check the complete Ada83 syntax but the visitor can later properly report to the user that it can’t handle a certain type of AST node and hence doesn’t support feature ’X’.

The first step is to collect all type information for the program. This visitor starts at the root AST node and traverses the tree only looking for type declarations. Types are defined by two string values. First, all types have a unique name. Second, the information about the arrangement of values the type represents along with possible constraints on the type are expressed by the type construction. Since Ada uses a scheme known as name equivalence, which means that if two types have different names they will be treated as different even though they may structurally represent the same information, this visitor keeps builds a map from type names to type construction. The root node is annotated with the type information for the program. Type names are used immediately in the next phase of compilation to check type equivalence of symbols and expressions. The type construction is carried through with the AST for later phases of the compilation where the information will be needed for proper code generation and in some cases (eg: range checking) run time information.

Before moving on to symbol collection, it is worth mentioning the Ada standard namespace. A naive approach to type creation would involve kicking off this phase with a non-empty type dictionary. The dictionary could be hard coded with the built-in Ada type declarations. A cleaner approach would instead have the compiler construct the type information from scratch by supplying it with an Ada source file with content similar to the Ada83 LRM appendix C “Predefined Language Environment”. This approach makes the compiler infinitely more adaptable to changes in the default language library. The types defined by the language are no different then than a separately compiled user-defined module that the compiler implicitly includes in the compilation.

The next step is to collect all symbols and verify that scoping rules aren't being violated. For example, in the statement `var := func(param)` if either `func` or `param` aren't in the scope of the assignment statement then even though the statement is syntactically correct it isn't valid.

The symbol table maps the mangled symbol name to type name plus decoration. The rule for name mangling global is just an extension of C name mangling - the global namespace is '_' and program subunits are prefixed by `parentName_`. For example, the symbol name for `procedure proc` within `package pack` would be `_pack_proc`. The type name values can be used as a key in the previously created type construction table. The type decorators capture information that is not directly relevant to the type, for example, if the given symbol is a constant. Type checking in Ada then boils down to checking that the symbol's type actually exists in the type construction mapping. If the type hasn't been declared by the time it is used a compiler error should be raised. Next, the type names can be checked for simple string equality. Finally, the validity of assignments, for example, can be verified by checking the decorators.

3.13 Code generation

Ideally we would have liked to have a proper intermediate representation. There are many benefits to having an IR. An IR promotes clean architectural separation and encourages languages dependent issues to remain in the front end while code generation issues and platform specific tweaks are the concern of the back end. More importantly, an IR makes the compiler portable since it avoids the multiplicative explosion involved when supporting multiple languages and platforms. However, while certain optimizations are language dependent or at the very least operate at the AST level, others like code flow analysis, static single assignment form, liveliness analysis etc. can all be done on the intermediate representation. Thus optimizing at this level benefits all front ends that can emit code to that type of intermediate form.

Finding the right boundary between a feature-rich and simple IR is no easy task. At the very least the IR should have the feature set of a virtual RISC machine, most likely with infinite registers so that the front end doesn't need to be concerned with the platform specific details of register allocation.

Due to time constraints, we weren't able to make use of an IR. The very beginnings of an intermediate representation tree-language can be found in `compilertools/ir.py`. The general idea was that the front end would have as its last phase a visitor that would translate a fully annotated parse tree to a tree of IR nodes. The IR tree can then be generated to platform code by a synthesis phase. A very straight forward translation is one to CVM a C-macro language similar to what was presented in class `compilertools/stages/_platforms/cvm/`.

One of the difficulties with implementing a compiler around an IR is that it becomes hard to rapidly prototype since it enforces a clean separation of language analysis from code generation. You couldn't, for example, easily pass a chunk of assembly or C code through the IR barrier. Every phase up to and including parsing is rather easy to unit test. Afterwards, it becomes harder to unit test the results and more desirable to run tests against the output and results from the compiled program. One might be tempted to implement a quick and dirty code generation visitor that converts directly from the annotated parse tree to straight-line assembly, or in our case C macros and possibly snippets of C code like calls to the standard library. This is actually what we have ended up doing with CPLR.

3.14 Linking

The compiler proper (`#/src/apps/compiler.py`) doesn't actually handle linking the result into an executable. The compiler always simply returns the result of the transformations requested. If no environment flag is given to stop compilation at an intermediate phase then the compiler will return an Obj instance (`compilertools/stages/obj.py`) containing the object code and global symbol information.

Linking is handled by `driver.py`, the abstract `cmpilertools/stages/link.py`, and the concrete implementation in `compilertools/stages/_platforms/link_cvm.py`. In the case of CPLR as it currently stands, the object code is really just a bunch of C macros. Linking, then, is actually an invocation of GCC to properly compile the C program into a true executable.

Because of this, CPLR is actually cross platform. The only requirements are a Python 2.2 or later interpreter, GCC, and a platform that is supported by the Boehm conservative Garbage Collector. The garbage collector library is built the first time the linker is invoked and statically linked to the generated executables. This means the resulting executables can be copied and run on any system of an equivalent architecture (cpu-type, lib-C version etc.).

4 Reflections

4.1 The compiler API

We would say that the compiler API is one of CPLR's strongest points.

Writing the Compiler API in a way that returns intact result objects that can be converted to a string turns out to be very powerful. If you step back from viewing the compiler as a whole - a unit whose sole purpose is to convert a source file on disk to an executable program on disk - you realize that you have an entity that knows how to translate program sources into several different forms.

You can then apply the compiler to a variety of situations. In fact, `driver.py` is nothing more than an application that prints issues to standard out and writes the string value of the compilation result to a file. It would be trivial (on the order of about 100 lines of Python) to instead invoke the compiler to stop after parsing and interpret the issues to do error highlighting in an IDE. In fact `compilertools/pretty_print.py` and `driver.py` already take a stab at this in order to print compiler errors and warnings to the user. The flexibility doesn't stop there. Though the object returned can be converted to a string but it is a full Python object. If you ask the compiler to stop after parsing, the result object *is* the AST where each node is a subtype of the Node object defined in `compilertools/parse/tree.py`. Similarly, if you stop after symbols have been collected you will have the AST with each node annotate with attribute values and an actual Python dictionary object mapping symbol names to type information. It doesn't take much to realize that without changing the compiler or any of its libraries in any way you can build tools that perform more in depth semantic analysis, for example checking for potential buffer overruns (though this is less of a concern in Ada).

Likewise, there is nothing set in stone about the compiler needing to start from a plain source file. The compiler simply wraps a pipeline of translator objects. The API could be augmented to not only allow stopping after any phase but also starting at any phase by taking an array slice of the pipeline. In fact, `util/pipeline.py` is already designed with the in mind. The most obvious example is support for pre-compiled headers or more generally, distributed compilation. You could persist the intermediate result of any translation phase to disk and pick up at that point later on. A more involved example would be supporting program translation at the AST level. You build a tool that invokes the compiler to get an AST, does whatever refactorings it needs and then feeds the AST back into the compiler to get a running program. This example would require a little more work to support in a generic way since the current AST is very Ada specific. Another example making use of such a feature would be distributed compilation.

4.2 A compiler as a set of libraries

By far the most difficult part of designing a compiler as a set of general purpose libraries that, for the most part, don't make any language or platform assumptions is designing the top level user interface. In the case of CPLR this is "simply" a command line. This surely isn't the first time the idea of a multi-lingual, multi-platform compiler has been tossed around. Of course we didn't actually attempt to implement different language front ends or different platform code generators. The design was always conscious, though, to keep things generic and keep language and platform specifics isolated and collected. One of the nice parts about the experience is the extra thought you must give to different command line options for example. Because the different phases are collected and put together on the fly only after language and platform have been inferred, different options may not even apply. Instead, you must think through what the generic top-level operations are. A good example is that no matter what language or platform is being compiled for the notion of stopping compilation after a certain phase is almost universal. So instead of taking the GCC approach (-s, -c, etc.) CPLR simply has --stop-after="named-phase". The value is kept in an environment and only evaluated by the modules that know about constructing the pipeline. This also has the advantage that it's simpler to read and remember what the compiler is doing when invoked.

It may be worth briefly mentioning localization of the command line interface. No real design effort was made to internationalize any of the compiler libraries. for example, error types hard code english strings. If we were to want to internationalize the compiler, a similar approach to the command line arguments would have to be taken. Because the set of libraries that make up the compiler is dynamically bound, each library would have to provide it's own localized strings and error messages. You couldn't simply return error codes to the caller since the number and value of such error codes is open ended.

4.3 Compiler compilers and duplication of code

Originally, proceeding by creating lexical and syntactic language definitions that made reference to separately defined Python Node and Token types seemed to be a straightforward approach. The language definitions could be kept concise and separate from the Python cruft of defining types, their initializers, and attributes. Additionally, creating visitors as separate entities clearly expressed the nature of each transformation. Towards the end of the experience; after you've jumped countless times between `grammar.py` and `nodes.py`, after having written several visitors that, while they do in fact clearly express their transformations, come with the tedium of duplication; you start to see the allure of compiler compilers that allow you to annotate a grammar with rules expressed in the target language. It would be very convenient for the definitions in `nodes.py` to be generable from the grammar. Likewise, while we used the unimplemented visitor methods to report unsupported language features in a finished product the assurance of knowing there are no unimplemented methods is a greater win. Otherwise you must rely on a very complete test coverage to generate all possible classes of parse trees such that all Node types will be visited.

4.4 Parallelizing development tasks

Another challenge with starting a compiler from scratch for the first time is figuring out just how to parallelize development and keep each other unblocked. The experience of developing a compiler is very different from, say, writing an OS at least until there is enough of the compiler present to get off the ground. It is fairly straight forward to agree upon the set of lexical tokens that allow parallel development of the lexer and parser. However, the set of parse tree nodes is highly dependent on the grammar so it is hard to start working on later transformations. Likewise, without having a clear picture on what the AST would end up looking like, it was hard to come up with any meaningful intermediate form. Part of the problem was in fact that the solutions, while partially outlined in class and texts, were unknowns for the most part. As we made progress the API for each phase became clearer.

4.5 ASTs and program transformations

One interesting avenue of abstract syntax trees is program transformations (for example: some of the work being done at Sun on Jackpot). The problem with transforming at the AST level is that it is very language specific.

along the lines of being language agnostic, one part of compiler design that has piqued our interest is the feasibility of making the AST less language specifc by extracting common language elements (eg: statements, expressions, functions, assignment...). Where do you draw the line? Is it in fact possible to generically say that transformation alpha on assignments is valid in any language. Alternatively, you could convert languages to an intermediate form and do the transformations at the level. This is similar to an optimizing JIT and approaches like Java bytecode, Microsoft's CLR, or LLVM. At this level, it seems very cumbersome, if intractable in certain situations, to support language level style refactoring. Often times there is not enough information contained at the IR level to apply a reverse transformation. Alternatively, you could design a completely separate high-level representation that is language agnostic. A high level VM if you will. In it would could define a mapping to and from various source languages and a set of well defined features the environment supports, functions, objects, abstract evaluation.

So how would you go about writing a symbol renaming, re-factoring algorithm without rewriting it for each language you're interested it? We don't really have concrete any concrete solutions. We simply wanted to finish off by mentioning one of the places where this whole experience has taken (left) us.