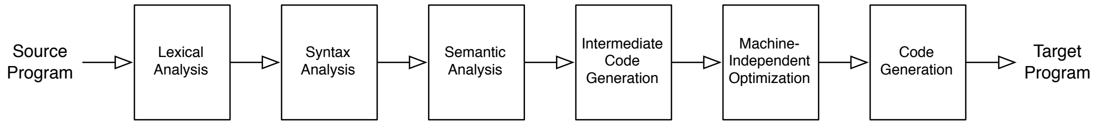
CS 301 Spring 2016
First Week Lab

Tiny Compiler: Front End

Plan

Updated after class to number exercises and improve headings. The content is the same.



This week in lab, we build a compiler that translates a tiny calculator language to x86 machine code, using the major stages of a "real" compiler. Our goal is to see the architecture of a working compiler at an accessible scale. We use this as a rough road map for the rest of the semester, when we explore each stage more deeply (and build a much larger compiler). I provide some code (tricky or design-heavy parts) and you implement the rest. By the end of the week, you will read or write every line of this compiler implementation. (Another goal is to demystify that black box early on!) Today, we start with an activity that will guide you through building the front end for our compiler.

Directions:

- Work through this activity in groups of about 3 (2 or 4 OK if needed). Please make sure there is at least one CS 251 alum in each group. Pick a group of people you do not know or have not worked with (much) previously. We will switch groups on Friday. This will give you a chance to meet classmates you do not know (well) and think about potential tutorial groups and project groups for the semester.
- As you work, check in with me after each Exercise if you are unsure about your answers.
- I will not collect anything from you, but write your thoughts down for reference. Naturally, you will want to write down the programming parts too.
- If we do not finish today (likely?), please try to finish what you can (with or without your group) by Friday, but don't spend more than another hour or two. If you are curious or stumped by something feel free to stop by my first week office hours (`http://cs.wellesley.edu/~bpw/`) or make an appointment. We will review this part on Friday and patch up any loose ends, then start another activity to build the back end of the compiler (intermediate code, optimization, code generation).
- Refer to Scala documentation on the tools page as needed:
  `https://cs.wellesley.edu/~cs301/tools.html#scala`

Setup

We assume you have the Eclipse Scala IDE installed: `http://scala-ide.org`

If you have Mercurial configured:

1. Fork the repository on Bitbucket:
   `https://bitbucket.org/wellesleycs301/cs301-labs/fork`.
2. Clone to your machine:

```
hg clone ssh://hg@bitbucket.org/your_bitbucket_username/cs301-labs
```

If you do not have Mercurial configured yet:

1. Download the tarball: `https://cs.wellesley.edu/~cs301/lab/cs301-tiny.tar.gz`
   You can do this without browser with (that's a capital Oh):

   ```
   curl -O https://cs.wellesley.edu/~cs301/lab/cs301-tiny.tar.gz
   ```

2. Unpack it:

   ```
   tar xfz cs301-tiny.tar.gz
   ```

Both: When you have acquired the code:

1. Start up Scala IDE and select the workspace `cs301-labs` that you just acquired.
2. Continue and then click through any welcome messages or "go to workbench".
3. Choose File > New > Scala Project.
4. Enter `tiny` as the Project name.
5. Click Finish.

Compiler Architecture

Before we define the TINY language, let us examine the high-level architecture of the compiler. In the `tiny` project in Eclipse, open up `Compiler.scala` in the `tiny` package in the `src` directory. Each stage of the compiler that appears in this code is defined in a file of the same name. `AST.scala` and `IR.scala` define data structures to represent TINY programs in the compiler.

Exercise 1. Read the main method of the compiler. Take a look around. Skim header comments in each of the Scala files. Is this the compiler structure we expect?

Program Representation. Pause for a moment and consider that, according to this code, it appears that programs are just more data. Like other data, they can be manipulated, computed upon, transformed, and so on. Like other data, their *representation* is independent from their *meaning*. In fact, this compiler will deal with the same program represented as:

- a source code file (a sequence of bytes on disk)
- a source code string (a sequence of characters)
- a stream of tokens (multi-character symbols)
- a tree
- several graphs (which also happen to be linked lists here)
- an assembly code string (and assembly files and executable machine code files outside the compiler)

Exercise 2. Read some code and describe at least 5 differences between Scala and a language with which you are familiar (*e.g.*, Java). (No need to write this down, just describe to each other.) **We will check in with the whole class here to share everyone's tips about Scala.**

TINY Language Definition

Informal Definition. Our *source language*, TINY[1], is a small calculator[2] language with simple arithmetic expressions, mutable variables with assignment, user input, and output (printing). The following program takes two integer inputs i_1 and i_2 from the user and prints the results of $2i_2 + 7$ and $i_1 + i_2 + 104$.

```
x = ( 4 + input ) ;
y = input ;
print ( ( 7 ) + ( y + y) ) ;
print ( ( x + ( y + 3 ) ) + 97 ) ;
```

Syntax. Programs are sequences of statements. Statements are either assignments or prints, ending in semicolons. Assignments have a source expression (right hand side of equal sign) and a destination variable (left hand side). Variable names are strings of letters other than `print` or `input`. Expressions are either literal integers, variable references, parenthesized infix additions of two expressions, or input calls. Expressions may be wrapped in unneeded parentheses. *Symbols (even parentheses) must be separated by at least one space or newline.*

Semantics. "It works like you think it does." Evaluation proceeds eagerly in order, one statement at time. Expressions are evaluated eagerly left to right. The `input` expression evaluates by waiting for the user to type an integer input, then returning the result.

More Formal Syntax Definition. The syntax of TINY is given by the following *grammar* definition.

$$
\begin{aligned}
P &::= S+ \\
S &::= Var\ \texttt{=}\ E\ \texttt{;}\ \mid\ \texttt{print}\ E\ \texttt{;} \\
E &::= Num\ \mid\ Var\ \mid\ \texttt{(}\ E\ \texttt{+}\ E\ \texttt{)}\ \mid\ \texttt{input}\ \mid\ \texttt{(}\ E\ \texttt{)}
\end{aligned}
$$

In this grammar definition, the symbols ::= and | show the structure of the grammar definition itself. The serif + indicates "one or more of the preceding symbol." Symbols in `monospace font` are *terminals* that appear concretely in the string representation of a program. *Nonterminals* like P describe an abstract class of possible strings. Each nonterminal has one or more *productions* or forms that it may take.

Read the above definition recursively:

- A program P is a sequence of one or more statements S.
- A statement S is one of:
  - an assignment, with a variable name followed by `=` followed by an expression E and `;`.
  - a print, with `print` followed by an expression E and `;`.
- An expression E is one of:
  - a literal integer (any string of digits)
  - a variable name (any string of letters other than `print` or `input`)
  - a `(` followed by an expression E_1 followed by a `+` followed by an expression E_2 followed by a `)`.
  - an `input` call

Exercise 3. Write a TINY program that takes 3 inputs i_1, i_2, and i_3 and prints $2(i_1 + i_3) + i_2 + 7$. Save it in `tests/exercise.tiny`.

[1]It stands for *Treetop Investigators Needlessly Yammering*, *Truly Inimitable Northern Yew*, or whatever else you would like.
[2]Perhaps *adding machine language* is a better description, since its only arithmetic operation is addition.

Lexing and Parsing By Hand

Lexical Tokens. The code you just wrote is a string of characters, but you do not think of it quite this way. You see deeper structure. At the shallowest level, you see words and meaningful symbols, not individual characters. It's `print`, not `p`, `r`, `i`, `n`, `t`. Congratulations! Your brain has already accomplished lexical analysis, transforming that stream of individual characters into a stream of higher-level meaningful tokens.

Abstract Syntax Trees. But your brain does not stop there. You do not see just `print` followed by `(`, then `5`, `+`, `2`, `)`, `;`. You see *structure.* In fact, you see *a tree* because *programs are trees!* The grammar shows the true structure of programs and we can visualize this structure as a tree. Every type of TINY expression, statement, or program is a type of node in the tree.

For example a number, `301`, is a leaf node. It is an "interesting" part of the program (*e.g.*, not just a parenthesis), but it has no subparts, no children. The expression `( 3 + 4 )` is an addition node (or `+` for short) with two children, `3` on the left and `4` on the right, both leaves. `print ( 1 + 2 ) ; print 5 ;` is a program node with two statement children... and so on.

Notice that the terminals `print`, `(`, `+`, `)`, `;` are not children of the node. They are just uninteresting artifacts of the concrete syntax that are no longer needed with the tree structure in place. We keep them around as convenient, succinct labels to describe a node's type, but we could describe the node type in any way we want.

Exercise 4. Draw trees for the programs and program pieces given in the previous paragraphs. Label nodes with the part of their syntax that is not part of a child. Program nodes have no concrete syntax "of their own." Label them P.

Exercise 5. Draw a tree for the TINY program you wrote earlier. First, rewrite the program as one long line. You will write the tree above the program. Leaves will stay where they are. Inner nodes will be lifted up directly above where they appear in the source code. Build the tree from the bottom left, working left-to-right through the program, one node at a time. Only add an inner node to your tree once all of its children are completely built. You may end up with a few small independent trees that you later adopt as children under one node.

Congratulations! You just did manual parsing (syntax analysis) on top of your manual lexical analysis. These steps convert an unstructured linear string of characters into the meaningful hierarchical program structure they encode. Now it's time to teach them to the computer.

AST Representation

Let's start work on the front end of our TINY compiler. Before implementing parsing, we need a target representation: ASTs.

Exercise 6. Read through `AST.scala`, keeping in mind your earlier notes about Scala. `Seq` is a generic ordered sequence type, so the type `Seq[Stmt]` is the type of a sequence of `Stmt`s. Sketch the class hierarchy built in `AST.scala`. Does it look anything like the grammar we gave? Are there any types of programs, statements, or expressions that are in the grammar but missing from the Scala definitions?

Scala's case classes allow you to construct instances without the `new` keyword. (For CS 251 alums, they are much like constructors of ML datatypes. They allow easy pattern-matching.) To construct a `print` statement carrying the expression `301`, we just write `Print(Num(301))`, which results in a `Print` object whose `expr` is a `Num` object whose `value` is `301`.

Exercise 7. Write the AST for the program `print ( 1 + 2 ) ; print 5 ;` as a Scala expression, using the AST types defined in `AST.scala`.

Scanner

Now it is time to start implementing lexing and parsing in `Parse.scala`. For lexical analysis (a.k.a. lexing a.k.a. scanning), we use a `java.util.Scanner` to consume the program source code and break it up into lexical tokens. Starting with your assignment for next week's tutorial meetings, you will learn some principles behind the `Scanner` as well as techniques for more efficient lexing.

The `Scanner` has a few methods we use:

- `hasNext(p)` takes a `String` describing a simple pattern. Its return value indicates whether or not the pattern `p` describes the next token in the source code input. These patterns are *regular expressions*, which you may have used before. More in the upcoming assignment!
- `next(p)` consumes the next token, using the pattern `p` to define what part of the input comprises the token, and returns that token. `next(b)` assumes `hasNext(p)` is true (so always check first).
- There are also integer-specific versions `hasNextInt()` and `nextInt()` that check for and get the next integer encoded by the source code, respectively.

By default, `Scanner`s distinguish the boundaries of tokens by whitespace. (Our TINY language requires whitespace between tokens to make this easy.) When specifying patterns for tokens, some characters have special meaning. Parentheses and plus must be escaped if you want a literal paren or plus character: `"\\("` or `"\\+"`.

Exercise 8. Take a quick look through `Parse.scala` to see the `Scanner` in action in some provided code. What pattern gets used for recognizing variable-name tokens?

Recursive Descent Parser

`Parse.scala` has the framework of a *recursive descent parser*, one of the more intuitive ways to structure a parser by hand. The provided parser handles a subset of the TINY language. Specificaly, it handles assignment statements, integer literal expressions, and inputs. You will extend it to handle the other types of statements or expressions.

Exercise 9. Write a TINY program using this restricted subset of the language and run the compiler. It should print the AST it has parsed. (Currently, it does nothing more.)

The Eclipse Scala IDE will automatically compile the Scala code (*i.e.*, compile the compiler...) each time you save. To run the compiler on a test program in `tests/this-program-is.tiny`, open a terminal, change directory to your workspace and run the command:

```
scala -cp bin tiny.Compiler tests/this-program-is.tiny
```

Exercise 10. Read over the provided methods in `Parse.scala`. How do they correspond to the grammar for TINY? How do the different branches in each method correspond to the grammar? (Consider only the subset of the grammar corresponding to the cases we support so far.) Does this *recursive* descent parser seem to use recursion? How? How is its use of recursion related to the grammar? This one is important. Make sure you understand before moving on.

Exercise 11. Extend the parser to handle `print` statements in `parseStmt`. Think carefully about how you order this case with respect to the case for assignment statements. Write a couple simple programs to test your extension.

Exercise 12. Extend the parser to handle parenthesized addition expressions in `parseExpr`. Write a few simple programs to test your extension.

Exercise 13. Extend the parser to handle "extra parentheses" expressions (the last production of E in the grammar) in `parseExpr`. Before coding, think carefully about how you will distinguish these from addition expressions. Remember, you have to read in tokens in order. If you want to see a later token, you have to consume everything in between first. The expressions `( 1 + 2 )` and `( 1 )` look identical for the first couple tokens. How can you teach the parser to tell them apart? Write a few simple programs to test your extension.

Congratulations! You have now tranformed a linear blob of text into a hierarchical data structure that captures the true structure of a program. You taught the computer to *understand* the structure behind the syntax of TINY programs.