Building a compiler for a mini programming language

CS 321 Languages and Compiler Design I, Winter Term 2016
Department of Computer Science, Portland State University

Primary learning objectives: Upon successful completion, students will be able to:

- Explain the phase structure of a typical compiler and the role of each phase.
- Use basic Java tools and language mechanisms relevant to compiler construction.

Introduction

These notes describe the construction of a simple compiler for a programming language that we will refer to as "mini". This language (and the code described here) was developed specifically for use in this course. As such, if you already know of another programming language called mini, or you find one elsewhere on the web, then please be aware that *it is probably not the same language!*

Our mini language is rather basic, lacking many of the features that you might expect to find in a more realistic programming language. Nevertheless, it is still rich enough to illustrate many of the key concepts of compiler structure that you will need to understand.

The full source code for our mini implementation is available from the class web site. You should download a copy, follow along, and experiment. Note, however, that we will be treating this code primarily as a set of high-level building blocks that can be plugged together to build a working compiler. Although you are welcome to take a deeper look at details of how the code works, this is not required or expected. In addition, it is important to note that later projects for this course will be based on different code bases. And while the underlying concepts in each case are the same, you may prefer to invest more of your time studying details of the code that is provided for later, more heavily weighted projects.

The mini language is an example of an *imperative* programming language, meaning that mini programs are written as sequences of *commands* or *statements*, each of which performs a specific action—such as updating the value of a variable or printing out a value—when the program is *executed*. The following listing shows a mini program for calculating the sum of the numbers from `1` to `10` and printing out the result:

00/test.mini

```
1  // A simple test
2  int i, t;
3  i = 0;
4  t = 0;
5  while (i<10) {
6    i = i + 1;
7    t = t + i;
8  }
9  print t;
```

This program uses two variables: a counter, `i`, that takes values from `0` to `10`; and a total, `t`, that is used to accumulate the sum of all the different values of `i`. If you have any doubt about what this program does or

how it works, you should try performing a dry run, tracing the values of the two variables at each step. If you conclude that the program should terminate after printing out a final result of `55`, then you are in good shape to carry on!

The accompanying software is made available in the form of a single zip file called `minitour.zip`. If you are using a Unix machine (such as one of the machines in the Linuxlab, or your own Linux or Mac OS X computer), and you have downloaded a copy of `minitour.zip` into your current working directory, then you can unpack and start inspecting its contents using the following commands:

```
$ unzip -q minitour.zip  # -q to suppress listing files
$ cd minitour
$ ls -C                  # -C to shown directory contents in columns
00              05              10              15              20
01              06              11              16              21
02              07              12              17              Index
03              08              13              18              minitour.pdf
04              09              14              19
$
```

This output indicates that there are actually 22 different versions of the software, numbered from `00` through to `21`. Each of these is stored in a different subdirectory, together with an `Index` file that provides a very brief summary of the new features that are added in each version. We will be walking through each of these changes in turn in the following sections, and you should use the numbers in the section headings of these notes to find the appropriate version of the software.

You might have noticed that the listing for our simple `mini` program was enclosed in a box with the label `00/test.mini` embedded in the middle of the top line. This indicates that you can find a machine readable copy of the code in that box by looking in the file `00/test.mini`. In many cases, we will only include fragments of code in these notes. This saves space, but also allows us to focus on the parts of the code that are most important. Note, however, that you can always use the line numbers, shown in the left margin, to relate the portion of code that is shown here to the full contents of the file.

Examples of command line interactions—such as the example at the top of this page showing how to unpack the `minitour.zip` file—will be indicated by a single vertical line on the left hand side. In these examples, the dollar symbol represents the operating system prompt. In practice, of course, you may see a different prompt symbol on your machine, depending on the details of how it has been configured.

Note that some familiarity with Java is assumed in these notes; you are welcome to ask the instructors for help in understanding specific details of course, but if you need further information, you should note that detailed documentation for Java is available from `http://docs.oracle.com/javase/6/docs/`.

```
-------------------------------------------------------------------------------
program : stmts                                    -- Programs
-------------------------------------------------------------------------------
stmt    : ";"                                      -- Empty statement
        | Id "=" expr ";"                          -- Assignment
        | "{" stmts "}"                            -- Block of statements
        | "while" "(" expr ")" stmt                -- While loop
        | "if" "(" expr ")" stmt [ "else" stmt ]   -- Conditional statement
        | "print" expr ";"                         -- Print statement
        | type Id { "," Id } ";"                   -- Variable declaration
stmts   : /* empty */                              -- Empty list of statements
        | stmts stmt                               -- Non-empty list of statements
-------------------------------------------------------------------------------
type    : "int"                                    -- The type of integer values
        | "boolean"                                -- The type of Boolean values
-------------------------------------------------------------------------------
expr    : lor                                      -- Expressions
lor     : lor "||" land                            -- Logical or
        | land
land    : land "&&" bor                            -- Logical and
        | bor
bor     : bor "|" bxor                             -- Bitwise or
        | bxor
bxor    : bxor "^" band                            -- Bitwise xor
        | band
band    : band "&" eql                             -- Bitwise and
        | eql
eql     : eql "==" rel                             -- Equality test
        | eql "!=" rel                             -- Not equal test
        | rel
rel     | rel "<" add                              -- Less than
        | rel ">" add                              -- Greater than
        | rel "<=" add                             -- Less than or equal
        | rel ">=" add                             -- Greater than or equal
        | add
add     : add "+" mult                             -- Addition
        | add "-" mult                             -- Subtraction
        | mult
mult    : mult "*" unary                           -- Multiplication
        | mult "/" unary                           -- Division
        | unary
unary   : "+" unary                                -- Unary plus
        | "-" unary                                -- Unary minus
        | "~" unary                                -- Bitwise negation
        | "!" unary                                -- Logical negation
        | primary
primary : IntLit                                   -- Integer literal
        | "true"                                   -- Boolean literal true
        | "false"                                  -- Boolean literal false
        | Id                                       -- Variable reference
        | "(" Expr ")"                             -- Parenthesised expression
-------------------------------------------------------------------------------
```

Figure 1: A grammar for the mini programming language

Simple command line application with arguments (00)

Java programs are structured as collections of classes. Any Java program that is intended to be used as a command line application should include at least one class with a `main` method that is declared as `public static void main(String[] args)`. As the name suggests, the array `args` will be initialized with the values of the arguments that are passed to the program on the command line.

Because we are interested in building a compiler, we will begin our development with a program consisting of a class called `Compiler` that takes the name of a file to be compiled as a command line argument:

00/Compiler.java

```
// 00 Simple command line program with arguments
public class Compiler {
  public static void main(String[] args) {
    if (args.length!=1) {
      System.out.println("This program requires exactly one argument");
    } else {
      System.out.println("Ok, we should look for an input called " + args[0]);
    }
  }
}
```

In its current form, of course, this program is a long way from actually functioning as a compiler, but it will still provide a good starting point for subsequent steps.

The standard Java compiler is a program called `javac`, and its primary function is to translate `.java` source files to corresponding `.class` files that can be executed on a Java virtual machine (JVM). The following transcript, for example, shows that a new `Compiler.class` file is created in the current directory when we compile the `Compile.java` source file shown above:

```
$ ls -C                    # List the files in this directory,
Compiler.java   test.mini
$ javac Compiler.java      # run the Java compiler,
$ ls -C                    # and note the new .class file in the listing
Compiler.class  Compiler.java   test.mini
$
```

Compiled Java class files can be executed using the `java` command with a first argument that specifies the name of the class that contains the required `main` method. Any subsequent command line arguments are packaged up in the array of strings that is passed as input to `main`. The following examples show how our initial `Compiler` behaves with zero, one, or two arguments respectively:

```
$ java Compiler            # run with zero arguments - an error!
This program requires exactly one argument
$ java Compiler one        # run with one argument   - success!
Ok, we should look for an input called one
$ java Compiler one two    # run with two arguments  - an error!
This program requires exactly one argument
$
```

Using an error handler (01)

Compilers typically have to deal with many different kinds of errors, including not only problesm with command line parameters, but also syntax errors, type errors, and others. The quality of the error messages that a compiler produces—measured intuitively by the extent to which they help programmers to locate and fix the problems in their code—can also have a significant impact on its usability in practice.

With this in mind, and before we add new compiler functionality to `Compiler.java`, we will modify our code to make use of a general library for flexible error handling, as shown in the following code:

01/Compiler.java

```
1  // 01 Using an error handler
2  import compiler.*;                                                    // <<<
3
4  public class Compiler {
5    public static void main(String[] args) {
6      Handler handler = new SimpleHandler();                            // <<<
7      if (args.length!=1) {
8        handler.report(new Failure("This program requires exactly one argument"));
9      } else {
10       System.out.println("Ok, we should look for an input called " + args[0]);
11     }
12   }
13 }
```

The particular library that we are using here is called `compiler` because it provides general functionality for compiler writers, particularly with respect to error handling. The code for this library is stored in a new `compiler` subdirectory and forms what is know in Java terminology as a *package*. The `import compiler.*;` declaration in Line 2 makes it easy to reference individual classes in the `compiler` package—such as `Handler` or `Failure`—simply by using their names. (Without the `import` line, it would still be possible to access these classes, but we would need to include the package name as a prefix to each class name, as in `compiler.Failure`.)

The specific parts of the `compiler` package used here are:

- The `Failure` class is used to construct an object that describes an error that should be reported to the user. In the form shown here, we use the syntax `new Failure(msg)` to capture a particular string message as part of the error. Later, we will see that there is also a form `new Failure(pos, msg)` that points the programmer to a specific position, `pos`, in the input program where the error has been detected.
- The `Handler` class represents objects that can be used to handle an error in some appropriate way. Different types of handler are appropriate in different contexts. For example, we might use a handler that displays errors on the console in a command line application, or a different handler that captures details about errors in a table as part of the GUI in an integrated development environment (IDE) like Eclipse or Visual Studio. An advantage of using the `compiler` package is that the compiler writer only needs to report an error to its handler, using a call of the form `handler.report(failure)` and can then leave the `handler` to take care of responding to the `failure` in some appropriate manner.
- The `SimpleHandler` class captures a simple strategy for error handling that displays reported errors on the console and keeps track of the total number of errors that have been reported. At a later point in the program, for example, we can check to see if any errors have been reported by testing if `handler.getNumFailures()>0`.

Using exception handling (02)

In some situations, it is possible to `report` an error to a `handler` without changing the overall flow of control. Suppose, for example, that we are scanning a program to check that each subexpression has an appropriate type, and that we find a loop `while (e) s` where `e` does not produce a `boolean` result. In this situation, we should certainly report the error related to `e`, but we can also continue scanning the body of the loop, `s`, for further errors. In practice, for example, developer productivity might be increased by a compiler that finds and reports multiple errors in a single run.

But there are some errors from which there is no obvious way to recover. For example, if the user does not properly specify the name of the file that they want to compile, then there is no sensible way to continue. In situations like this, it is appropriate to trigger a Java exception by using the `throw` construct to abort the current computation. The following variant of `Compiler.java` illustrates this approach:

02/Compiler.java

```
// 02 Using exception handling (try, catch, and throw)
import compiler.*;

public class Compiler {
  public static void main(String[] args) {
    Handler handler = new SimpleHandler();
    try {                                                                  // <<<
      if (args.length!=1) {
        throw new Failure("This program requires exactly one argument"); // <<<
      }
      System.out.println("Ok, we should look for an input called " + args[0]);
    } catch (Failure f) {                                                  // <<<
      handler.report(f);
    } catch (Exception e) {                                                // <<<
      handler.report(new Failure("Exception: " + e));
    }
  }
}
```

In this program, the enclosing `try{...}` construct ensures that any exception that occurs while the code between the opening and closing braces is being executed will be caught by one of the two `catch(...){...}` blocks. The first of these deals with `Failure` exceptions (i.e., compiler diagnostics generated by our code), while the second deals with other kinds of Java `Exception`. We do not expect to see exceptions of the latter kind in the current program, but later versions might trigger such exceptions if, for example, there is an I/O error such as a "file not found".

Overall, the behavior of this program is much the same as our original version of `Compiler.java`, modulo some small changes in the way that `SimpleHandler` displays error messages.

```
$ java Compiler            # run with zero arguments - an error!
ERROR: This program requires exactly one argument
$ java Compiler one        # run with one argument   - success!
Ok, we should look for an input called one
$ java Compiler one two    # run with two arguments  - an error!
ERROR: This program requires exactly one argument
$
```

Now we have the error handling infrastructure in place, it is time to turn our attention to adding some true compiler functionality!

Reading a sequence of integer codes from a file (03)

The standard Java libraries provide a simple way to read the contents of a file: use the file's name to create an associated `FileReader` object and then call its `read()` method repeatedly to obtain codes for each of the characters in the file. Note that each `read()` call returns an integer value, with different integer codes corresponding to different characters. For example, the lower case character `i` has code 105, the semicolon has code 59, the space has code 32, and the newline character has code 10. As a special case, the `read()` method returns the numeric code -1 when it reaches the end of the file. Using these ideas, we can create a new version of `Compiler.java` that adds a `.mini` suffix to the given program name, and then reads and prints a table showing the numeric codes for each of the characters in that file:

03/Compiler.java

```
13  // Read program:
14  FileReader reader = new FileReader(args[0] + ".mini");              // <<<
15  int n = 0;  // Number of characters read
16  int c;      // Code for individual characters
17  while ((c = reader.read()) != -1) {                                 // <<<
18    System.out.print("| " + c + "\t");
19    if ((++n % 8)==0) {
20      System.out.println("|");
21    }
22  }
23  System.out.println("|");
```

Note that we are only showing a fragment of the full source file here. However, other than the addition of an extra `import` declaration to provide access to the `FileReader` class, the rest of the code—including the exception handling infrastructure and the code to check for a single command line argument—is exactly the same as in the previous versions.

A simple test run produces the following output:

```
1   $ java Compiler test
2   | 47    | 47    | 32    | 65    | 32    | 115   | 105   | 109   |
3   | 112   | 108   | 101   | 32    | 116   | 101   | 115   | 116   |
4   | 10    | 105   | 110   | 116   | 32    | 105   | 44    | 32    |
5   | 116   | 59    | 10    | 105   | 32    | 61    | 32    | 48    |
6   | 59    | 10    | 116   | 32    | 61    | 32    | 48    | 59    |
7   | 10    | 119   | 104   | 105   | 108   | 101   | 32    | 40    |
8   | 105   | 60    | 49    | 48    | 41    | 32    | 123   | 10    |
9   | 32    | 32    | 105   | 32    | 61    | 32    | 105   | 32    |
10  | 43    | 32    | 49    | 59    | 10    | 32    | 32    | 116   |
11  | 32    | 61    | 32    | 116   | 32    | 43    | 32    | 105   |
12  | 59    | 10    | 125   | 10    | 112   | 114   | 105   | 110   |
13  | 116   | 32    | 116   | 59    | 10    |
14  $
```

Note, however, that we will trigger an exception if we attempt to read from a file that does not exist:

```
1   $ java Compiler nosuchfile
2   ERROR: Exception: java.io.FileNotFoundException: nosuchfile.mini (No suc
3   h file or directory)
4   $
```

Reading a sequence of characters from a file (04)

The previous version of `Compiler.java` displays the contents of the specified source file as a sequence of numeric codes. This reflects the way that the source program is encoded in the machine, but we can make some quick changes to obtain more human readable output. Specifically, the following variant uses a new function, `charAsString()`, defined elsewhere in `Compiler.java`, to generate a printable description for each individual character:

04/Compiler.java

```
13    // Read program:
14    FileReader reader = new FileReader(args[0] + ".mini");
15    int n = 0;  // Number of characters read
16    int c;      // Code for individual characters
17    while ((c = reader.read()) != -1) {
18      System.out.print("| " + charAsString(c) + "\t");                  // <<<
19      if ((++n % 8)==0) {
20        System.out.println("|");
21      }
22    }
23    System.out.println("|");
```

With this revision, a simple test run produces the following output, which makes it easier to see the relationship between the original source code (which we'll print out again at the end using a `cat` command), and the sequence of character codes that the `FileReader` produces:

```
1     $ java Compiler test
2     | /     | /     | ' '   | A     | ' '   | s     | i     | m     |
3     | p     | l     | e     | ' '   | t     | e     | s     | t     |
4     | \n    | i     | n     | t     | ' '   | i     | ,     | ' '   |
5     | t     | ;     | \n    | i     | ' '   | =     | ' '   | 0     |
6     | ;     | \n    | t     | ' '   | =     | ' '   | 0     | ;     |
7     | \n    | w     | h     | i     | l     | e     | ' '   | (     |
8     | i     | <     | 1     | 0     | )     | ' '   | {     | \n    |
9     | ' '   | ' '   | i     | ' '   | =     | ' '   | i     | ' '   |
10    | +     | ' '   | 1     | ;     | \n    | ' '   | ' '   | t     |
11    | ' '   | =     | ' '   | t     | ' '   | +     | ' '   | i     |
12    | ;     | \n    | }     | \n    | p     | r     | i     | n     |
13    | t     | ' '   | t     | ;     | \n    |
14    $
15    $ cat test.mini
16    // A simple test
17    int i, t;
18    i = 0;
19    t = 0;
20    while (i<10) {
21      i = i + 1;
22      t = t + i;
23    }
24    print t;
25    $
```

Reading a sequence of lines from an input source (05)

Some basic elements of mini programs—such as keywords, identifiers, and literals—can span multiple characters. This requires some form of buffering to keep track of previously read characters while we are in the process of recognizing these items. For this reason, our next step will be to move from viewing the source file as a sequence of characters to treating it instead as a sequence of lines, each of which serves as a multiple character buffer. (In particular, the complete text for every keyword, identifier, and literal in a mini program occurs as a substring of a single program line.)

This approach is supported by the `JavaSource` subclass of the `Source` class, both of which are included in the `compiler` package. In addition to buffering, subclasses of `Source` can be used to accommodate special features of particular programming languages (such as Unicode escapes in Java, or literate code in Haskell), or to accept input from other sources (such as an in-memory buffer, command line arguments, or the standard input device). The `JavaSource` class is specifically designed to handle the task of reading Java source code—or source code for other languages that use the same input conventions—from a text file:

05/Compiler.java

```
13  // Read program:
14  String     input  = args[0] + ".mini";
15  FileReader reader = new FileReader(input);
16  Source     source = new JavaSource(handler, input, reader);        // <<<
17  String line;
18  while ((line=source.readLine())!=null) {
19    System.out.println(source.getLineNo() + ":\t" + line);
20  }
```

This program works by passing a `FileReader` object, representing the input source file, as an argument to the `JavaSource` constructor, and then repeatedly calling the latter's `readLine()` method to read individual lines of input. This continues until `readLine()` returns a `null` result to signal that the end of the input file has been reached. As you can see on Line 19, the `source` object also supports a `getLineNo()` method to return the current line number in the input file.

Behind the scenes, `readLine()` takes care of a number of mundane details, from reading individual characters from the `FileReader` parameter, to miscellaneous special case details. As one example of the latter, `readLine()` handles the use of *Unicode escapes* in the input stream. This is a special feature of Java that allows unusual characters to be included as part of a program text by writing strings of the form `\uXXXX`. Here, `XXXX` denotes a four digit hexadecimal code that identifies a particular Unicode character. This can be useful if you want to use a specific character in your program—perhaps a special symbol or a character from an unusual alphabet—but you do not have a key for entering that character on your keyboard. For example, if the < key on your keyboard is broken, then you can still use the < symbol in your program by writing it as `\u003c`. (Let's just hope that you never break the backslash key!) The following variant of our previous `test.mini` file illustrates this (the Unicode escape is used on Line 5):

05/test.mini

```
1  // A simple test
2  int i, t;
3  i = 0;
4  t = 0;
5  while (i\u003c10) {
6    i = i + 1;
7    t = t + i;
8  }
9  print t;
```

When we run our new version of `Compiler.java`, however, we can see that the Unicode escape has been replaced, as intended, with the required < symbol:

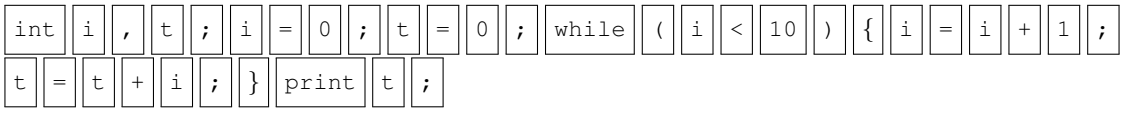
```
$ java Compiler test
1:      // A simple test
2:      int i, t;
3:      i = 0;
4:      t = 0;
5:      while (i<10) {
6:        i = i + 1;
7:        t = t + i;
8:      }
9:      print t;
$
```

As such, later stages of our compiler, which only use the output from the `source` object, will treat the input program in exactly the same way as if it had originally been written using the < character instead of the `\u003c` Unicode escape.

It is important to realize that this processing occurs prior to lexical analysis (which will be described in the next section), which means, for example, that Unicode escapes can be used anywhere that they are needed in a source program, including as part of an identifier, a comment, or a literal.

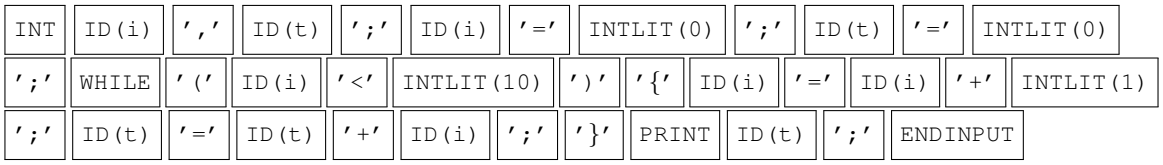
Reading a sequence of tokens from a lexer (06)

Lexical analysis is a process that translates source programs into sequences of *tokens*. In particular, this means gathering together groups of characters that represent items like keywords, identifiers, and literals into single logical entities, and filtering out characters that are part of comments or whitespace. From the perspective of lexical analysis, a first step in understanding the `test.mini` program that we have seen in previous sections is to view it as the following sequence of *lexemes*:



Each of the boxes shown here contains a small fragment of the source program text—arranged in order from left to right—including keywords, punctuation, operators, identifiers, and literals. Although there is no way to recover the exact form of the original program from this list (including details such as layout, whitespace, or comments), none of the essential details of the program have been lost.

The next step is to organize the set of all possible lexemes in to different *token* types. For example, we might have one token type `INT` that represents just the single keyword `int`, and other token types `ID` and `INTLIT` to represent the sets of all identifiers and all integer literals, respectively. In the latter cases, we might write `ID(x)` to describe an identifier with name `x` and `INTLIT(n)` to describe an integer literal with value `n`. From this perspective, the essential content of the `test.mini` source file can be represented by the following list of tokens:



Note that each of the boxes in this diagram includes a token code and, for identifiers and integer literals, an additional parenthesized attribute to capture the associated name or value. In addition, we have included an additional token, represented by the name `ENDINPUT` to mark the end of the input stream.

A software component that implements lexical analysis is referred to as a *lexical analyzer*, or *lexer* for short. (The term *scanner* is also quite widely used.) The lexer for `mini` is implemented by adding a new package called `lexer` that contains two files. The first, `MiniTokens.java`, specifies a numeric code for each of the different token types that can appear in a `mini` program:

06/lexer/MiniTokens.java

```
1  package lexer;
2
3  public interface MiniTokens {
4      // The following names are used to represent tokens that are (or
5      // can be) more than one character long.  Every token should have
6      // a distinct code, and these codes should not conflict with the
7      // single character token codes listed below.  In all other regards,
8      // however, the mapping of tokens to codes is arbitrary.
9      public int ENDINPUT = 0;  // Signals the end of the input stream
10     public int ID       = 1;  // An identifier
11     public int INTLIT   = 2;  // An integer literal
12     public int EQEQ     = 3;  // ==  (equality test)
13     public int NEQ      = 4;  // !=  (not equals)
14     public int LTE      = 5;  // <=  (less than or equal)
```

```
15        public int GTE      = 6;  // >=  (greater than or equal)
16        public int LAND     = 7;  // &&  (logical/short-circuiting and)
17        public int LOR      = 8;  // ||  (logical/short-circuiting or)
18        public int IF       = 9;  // keyword if
19        public int ELSE     = 10; // keyword else
20        public int WHILE    = 11; // keyword while
21        public int PRINT    = 12; // keyword print
22        public int INT      = 13; // keyword int
23        public int BOOLEAN  = 14; // keyword boolean
24        public int TRUE     = 15; // Boolean literal true
25        public int FALSE    = 16; // Boolean literal false
26
27        // The following single characters are also used as token codes:
28        //   ';', '{', '}', '=', '(', ')', '(', ')', '+', '-',
29        //   '~', '!', '*', '/', '<', '>', '&', '|', '^', ','.
30        // All of these characters have distinct numeric codes that are
31        // greater than 32, so there is no conflict with the symbolic
32        // token code names listed above.
33    }
```

Note that we have introduced symbolic names for token types like `ID` and `INT`, but that we use single character values like `'+'` or `'('` as token codes for the corresponding single character punctuation and operator symbols. In the end, however, all of these token codes are just simple numeric values. So long as we ensure that each different type of token has a distinct token code, the exact mapping of token types to numeric codes is not significant.

The second component in our lexer implementation is the `MiniLexer` class, whose primary methods are as follows:

- `nextToken()` reads the next token from a given source and returns the appropriate numeric code for that token.
- `getToken()` returns the numeric code for the most recently read token.
- `getPos()` returns the position of the most recently read token in the input stream; the compiler can use this information to refer back to specific positions in a source file when it detects an error.
- `getLexeme()` returns a string giving the text of the current token. This method can be used to find the name of an indentifier token following a call to `nextToken()` that returns `ID`.
- `getNum()` returns an integer that specifies the value of the most recently read integer literal. This method is intended to be used following a call to `nextToken()` that returns `INTLIT`.
- `tokenName()` returns a string that describes the current token.

Putting all of this together, we can build a new version of the running `Compiler.java` program that includes support for lexical analysis:

06/Compiler.java

```
14    // Read program:
15    String     input  = args[0] + ".mini";
16    FileReader reader = new FileReader(input);
17    Source     source = new JavaSource(handler, input, reader);
18    MiniLexer  lexer  = new MiniLexer(handler, source);                // <<<
19    while ((lexer.nextToken())!=MiniTokens.ENDINPUT) {
20      Position pos = lexer.getPos();
```

```
21      System.out.println(pos.coordString()
22                         + "\t  " + lexer.getToken()
23                         + "\t"   + lexer.tokenName());
24    }
25    System.out.println("ENDINPUT");
```

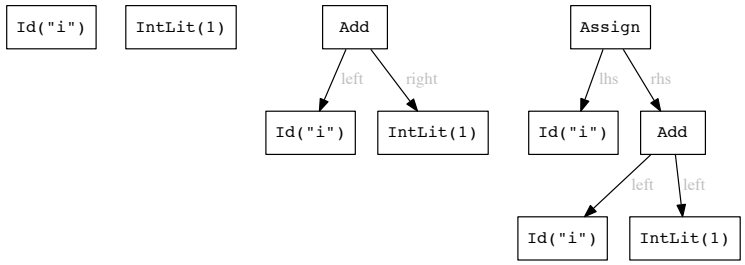
In this code, we construct a new `MiniLexer` object in Line 18, that will take its input, one line at a time, from the specified `source`. The main body of the program is a loop that makes repeated calls to `nextToken()`, printing out a description for each token that it finds, including the position of the token (a pair containing the column and line numbers for the first character in the token), the token code, and a description of the token. The loop eventually terminates when the lexer returns the `ENDINPUT` token code:

```
1   $ java Compiler test
2   (1, 2)    13    int keyword
3   (5, 2)    1     identifier, i
4   (6, 2)    44    comma
5   (8, 2)    1     identifier, t
6   (9, 2)    59    semicolon
7   (1, 3)    1     identifier, i
8   (3, 3)    61    = operator
9   (5, 3)    2     integer literal, 0
10  (6, 3)    59    semicolon
11  (1, 4)    1     identifier, t
12  (3, 4)    61    = operator
13  (5, 4)    2     integer literal, 0
14  (6, 4)    59    semicolon
15  (1, 5)    11    while keyword
16  (7, 5)    40    open parenthesis
17  (8, 5)    1     identifier, i
18  (9, 5)    60    < operator
19  (10, 5)   2     integer literal, 10
20  (12, 5)   41    close parenthesis
21  (14, 5)   123   open brace
22  (3, 6)    1     identifier, i
23  (5, 6)    61    = operator
24  (7, 6)    1     identifier, i
25  (9, 6)    43    + operator
26  (11, 6)   2     integer literal, 1
27  (12, 6)   59    semicolon
28  (3, 7)    1     identifier, t
29  (5, 7)    61    = operator
30  (7, 7)    1     identifier, t
31  (9, 7)    43    + operator
32  (11, 7)   1     identifier, i
33  (12, 7)   59    semicolon
34  (1, 8)    125   close brace
35  (1, 9)    12    print keyword
36  (7, 9)    1     identifier, t
37  (8, 9)    59    semicolon
38  ENDINPUT
39  $
```

As a quick cross check, for example, you can confirm that the first token in `test.mini` is an `int` keyword, in Column 1 and Line 2, with the numeric code 13 that corresponds to the definition of `INT` in `06/lexer/MiniTokens.java`.

Building an abstract syntax tree (07)

The next step is to build a *parser* that can turn the sequence of tokens produced by a lexer in to a data structure called an *abstract syntax tree*, or AST, that captures the essential structure of the program. The following diagram, for example, shows abstract syntax trees for the expressions `i`, `1`, `i+1`, and `i=i+1`.



The first two trees, on the left of the diagram, are single nodes. As the names suggest, `Id` nodes represent identifiers, and `IntLit` nodes represent integer literals. The third tree has an `Add` node at its root, indicating that it represents an addition, together with two subtrees that represent its left and right arguments. Finally, the last tree has an `Assign` node at its root and has subtrees that describe the left hand side (lhs) and right hand side (rhs) of the assignment.

There are two steps that we need to complete the implementation of a parser for the mini language. The first step is to define a set of classes that can be used to represent abstract syntax trees. There is one such class for each of the different forms of statement and expression that can appear in a mini program; these classes are stored in the `ast` subdirectory. The second step is construct a parser that will read tokens from the lexer and build the appropriate abstract syntax tree structure using the various `ast` classes. The code in the `MiniParser` class, which is included in the `parser` subdirectory, addresses this need.

In the following version of `Compiler.java`, we connect a `lexer` for mini to a new `MiniParser`, and then use the `parseProgram()` method to read an abstract syntax tree for the program. If this completes, without throwing an exception, then we can conclude that the input program was syntactically valid:

07/Compiler.java

```
16  // Read program:
17  String     input  = args[0] + ".mini";
18  FileReader reader = new FileReader(input);
19  Source     source = new JavaSource(handler, input, reader);
20  MiniLexer  lexer  = new MiniLexer(handler, source);
21  MiniParser parser = new MiniParser(handler, lexer);                    // <<<
22  Stmt       prog   = parser.parseProgram();                             // <<<
23  if (handler.hasFailures()) {                                           // <<<
24    throw new Failure("Aborting: errors detected during syntax analysis"); // <<<
25  }                                                                      // <<<
26
27  // Output result:
28  System.out.println("Syntax valid");
```

After all the work that is needed to reach this point, it is perhaps a little disappointing that the only output is a simple message indicating that the input program was valid:

```
1  $ java Compiler test
2  Syntax valid
3  $
```

Happily, we will see more interesting uses of abstract syntax trees in the next few sections!

Displaying abstract syntax trees using indentation (08)

In this section, we extend our code to print a description of the AST structures that result from a successful parse. The output relies on indentation to reflect tree structure: we use one line for each node and print each child node on a subsequent line with greater indentation than its parent.

08/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Output result:
28    new IndentOutput(System.out).indent(prog);                          // <<<
```

Running this program with our previous `test.mini` file as input produces the following output:

```
1   $ java Compiler test
2   Block
3     VarDecl
4       int
5       Id("i")
6       Id("t")
7     Assign
8       Id("i")
9       IntLit(0)
10    Assign
11      Id("t")
12      IntLit(0)
13    While
14      Lt
15        Id("i")
16        IntLit(10)
17      Block
18        Assign
19          Id("i")
20          Add
21            Id("i")
22            IntLit(1)
23        Assign
24          Id("t")
25          Add
26            Id("t")
27            Id("i")
28    Print
29      Id("t")
30  $
```

We can quickly extract several details here—for example, the main program is a `Block` whose body consists of a `VarDecl`, two `Assign`s, a `While`, and then a `Print`. This gives us some confidence (although no guarantee) that both the parser and the `indent()` implementation are working correctly.

Pretty printing abstract syntax trees (09)

For anything other than short programs, the indented output that is produced by the code described in the previous section is very hard to read. An alternative approach is to print out descriptions of abstract syntax trees in a form that more closely resembles the concrete syntax of the mini language. This approach is sometimes referred to as "pretty printing":

09/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Output result:
28    String output = args[0] + ".txt";
29    new TextOutput(output).toText(prog);                              // <<<
30    System.out.println("Pretty printed AST output: " + output);
```

This version of `Compiler.java` writes its output directly to a file, so we have to use separate commands to run the pretty printer and then look at the results. In the following transcript, we show the effects of running our pretty printer on a "flattened" `test.mini` where all of the tokens have been squeezed on to a single line:

```
1     $ cat flattest.mini       # display a flattened version of test.mini
2     int i,t;i=0;t=0;while(i<10){i=i+1;t=t+i;}print t;
3     $ java Compiler flattest  # pass it through the pretty printer
4     Pretty printed AST output: flattest.txt
5     $ cat flattest.txt        # and see how much prettier it looks!
6     int i, t;
7     i = 0;
8     t = 0;
9     while (i<10) {
10        i = i+1;
11        t = t+i;
12    }
13    print t;
14
15    $
```

As the pretty printed output here shows, our parser was still able to extract the true structure of the program from its obfuscated input, without the use of layout or spacing information that can make code easier for humans to read.

Drawing abstract syntax trees using dot (10)

Tectual output is good for viewing larger programs. For small programs, however, it can be very useful to have a way to visualize abstract syntax trees graphically, making it easy to see the underlying tree structure as directly as possible. We can accomplish this here by modifying our compiler to generate descriptions of abstract syntax tree in a language called `dot`, which is an integral part of the AT&T GraphViz tools. (See `http://www.graphviz.org`, or `http://web.cecs.pdx.edu/~mpj/cs321/graphviz/` for a short introduction that is more directly oriented to CS 321.)

Following the same pattern as the previous examples, we extend our compiler with a new `DotOutput` class that represent files containing `dot` code, and we add `toDot()` methods to the abstract syntax classes for each of the different forms of expression and statement. The resulting version of `Compiler.java` looks very similar to the code in the previous two sections using `IndentOutput` and `TextOutput`, respectively:

10/Compiler.java

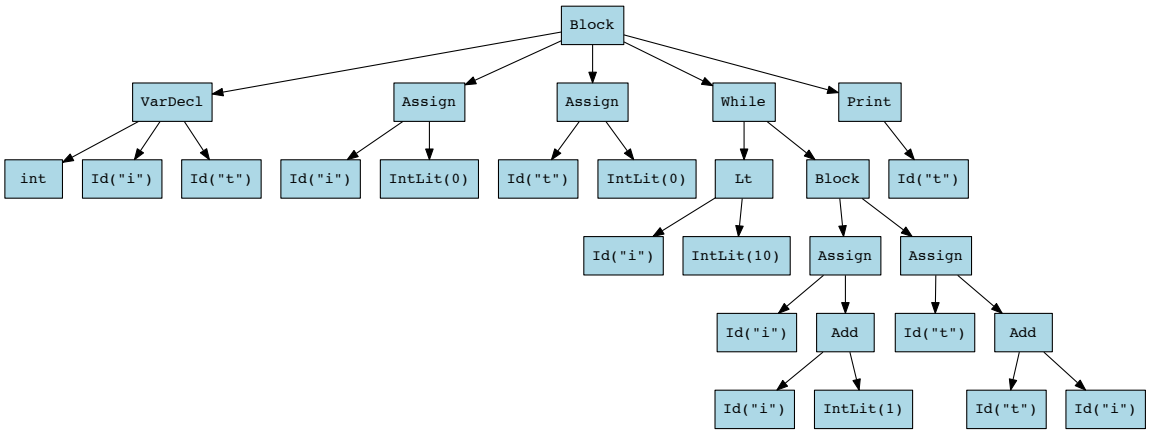
```
16  // Read program:
17  String     input  = args[0] + ".mini";
18  FileReader reader = new FileReader(input);
19  Source     source = new JavaSource(handler, input, reader);
20  MiniLexer  lexer  = new MiniLexer(handler, source);
21  MiniParser parser = new MiniParser(handler, lexer);
22  Stmt       prog   = parser.parseProgram();
23  if (handler.hasFailures()) {
24    throw new Failure("Aborting: errors detected during syntax analysis");
25  }
26
27  // Output result:
28  String output = args[0] + ".dot";
29  new DotOutput(output).toDot(prog);                              // <<<
30  System.out.println("AST in dot format output: " + output);
```

This time, however, the output that is produced is in `dot` format:

```
1   $ java Compiler test            # read the test.mini file
2   AST in dot format output: test.dot
3   $ cat test.dot                  # inspect the generated dot file
4   digraph AST {
5   node [shape=box style=filled fontname=Courier];
6   0[label="Block" fillcolor="lightblue"];0 -> 1;
7   1[label="VarDecl" fillcolor="lightblue"];1 -> 2;
8   2[label="int" fillcolor="lightblue"];1 -> 3;
9   3[label="Id(\"i\")" fillcolor="lightblue"];1 -> 4;
10  4[label="Id(\"t\")" fillcolor="lightblue"];0 -> 5;
11  5[label="Assign" fillcolor="lightblue"];5 -> 6;
12  6[label="Id(\"i\")" fillcolor="lightblue"];5 -> 7;
13  7[label="IntLit(0)" fillcolor="lightblue"];0 -> 8;
14  8[label="Assign" fillcolor="lightblue"];8 -> 9;
15  9[label="Id(\"t\")" fillcolor="lightblue"];8 -> 10;
16  10[label="IntLit(0)" fillcolor="lightblue"];0 -> 11;
17  11[label="While" fillcolor="lightblue"];11 -> 12;
18  12[label="Lt" fillcolor="lightblue"];12 -> 13;
19  13[label="Id(\"i\")" fillcolor="lightblue"];12 -> 14;
20  14[label="IntLit(10)" fillcolor="lightblue"];11 -> 15;
21  15[label="Block" fillcolor="lightblue"];15 -> 16;
22  16[label="Assign" fillcolor="lightblue"];16 -> 17;
```

```
17[label="Id(\"i\")" fillcolor="lightblue"];16 -> 18;
18[label="Add" fillcolor="lightblue"];18 -> 19;
19[label="Id(\"i\")" fillcolor="lightblue"];18 -> 20;
20[label="IntLit(1)" fillcolor="lightblue"];15 -> 21;
21[label="Assign" fillcolor="lightblue"];21 -> 22;
22[label="Id(\"t\")" fillcolor="lightblue"];21 -> 23;
23[label="Add" fillcolor="lightblue"];23 -> 24;
24[label="Id(\"t\")" fillcolor="lightblue"];23 -> 25;
25[label="Id(\"i\")" fillcolor="lightblue"];0 -> 26;
26[label="Print" fillcolor="lightblue"];26 -> 27;
27[label="Id(\"t\")" fillcolor="lightblue"];}
$
$ dot -Tpdf -otest.pdf test.dot # and produce a graphic in pdf format
$
```

It is possible to make sense of the contents of the `test.dot` file as a sequence of declarations that describe individual nodes in the tree (using declarations of the form `nodeNumber[...];`) and edges between nodes (using declarations of the form `nodeSrc -> nodeDest;`). However, this output is not really intended to be human readable, and is instead expected to be passed as an input to the `dot` command that we see in the last line of the above transcript. In general, `dot` can be used to produce output in a variety of graphical formats. With the specific command used above, however, an output in `pdf` format is requested, and this produces the following diagram:

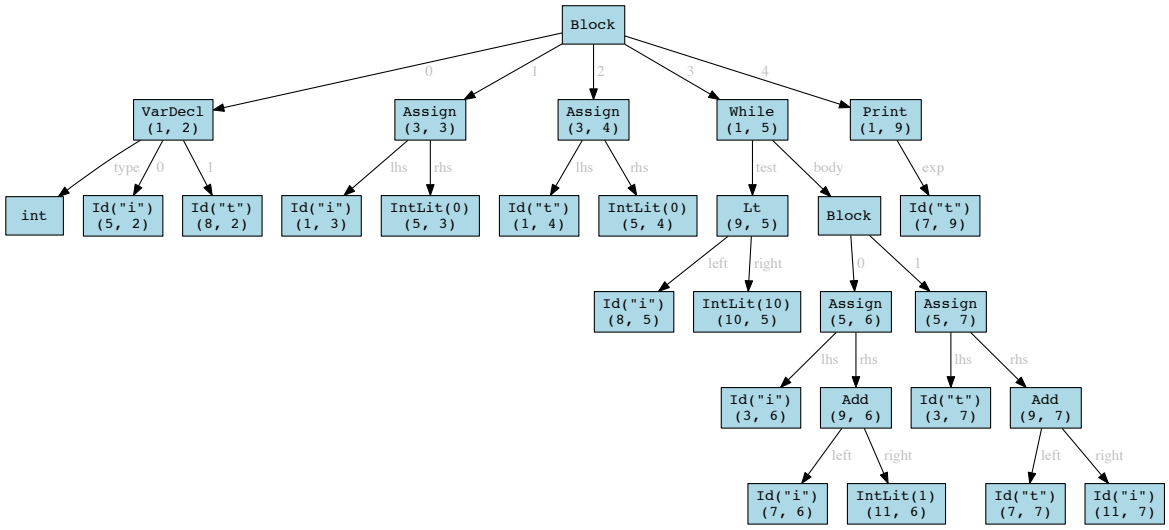


Diagrams like this quickly become hard to read as the size of the input program grows. Nonetheless, they are an excellent tool for those who are just beginning to learn about abstract syntax trees. In particular, if you ever struggle to figure out what the abstract syntax tree for a particular mini program looks like, then you might want to try typing the mini code in to a text file, and then running a sequence of commands like the ones shown here.

Drawing annotated abstract syntax trees using dot (11)

The basic infrastructure for drawing diagrams of abstract syntax trees that we saw in the previous section can be extended to capture extra details about those trees. For example, with some small modifications to the Version 10 code, we can arrange for each tree edge to be labeled with the name of the corresponding attribute.

Without any further changes to the version of `Compiler.java` that we used in the previous section, we can now repeat those same steps to obtain a more heavily annotated graphic showing the abstract syntax tree for `test.mini` (or for any other mini source file, for that matter!):



Of course, you might need a magnifying glass to see all those fine details on a printed page, but at least you can zoom in if you are viewing the diagram on a computer in `pdf` format!

Scope analysis (12)

Scope analysis traverses the abstract syntax tree to make sure that every variable that is used in the program has a corresponding definition.

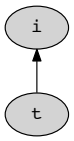
12/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);                          // <<<
29
30    // Output result:
31    System.out.println("Scope analysis did not detect any errors");
```

The set of variables that are in scope at each point in a program can be described by the elements of a linked list, which we will refer to as an *environment*. The environments are extended by variable declarations, but these variable declarations "go out of scope" at the end of the block in which they appear. Some examples:

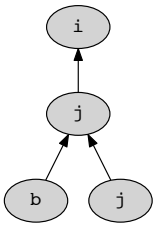
12/test.mini

```
1    // A simple test
2    int i, t;
3    i = 0;
4    t = 0;
5    while (i\u003c10) {
6      i = i + 1;
7      t = t + i;
8    }
9    print t;
```

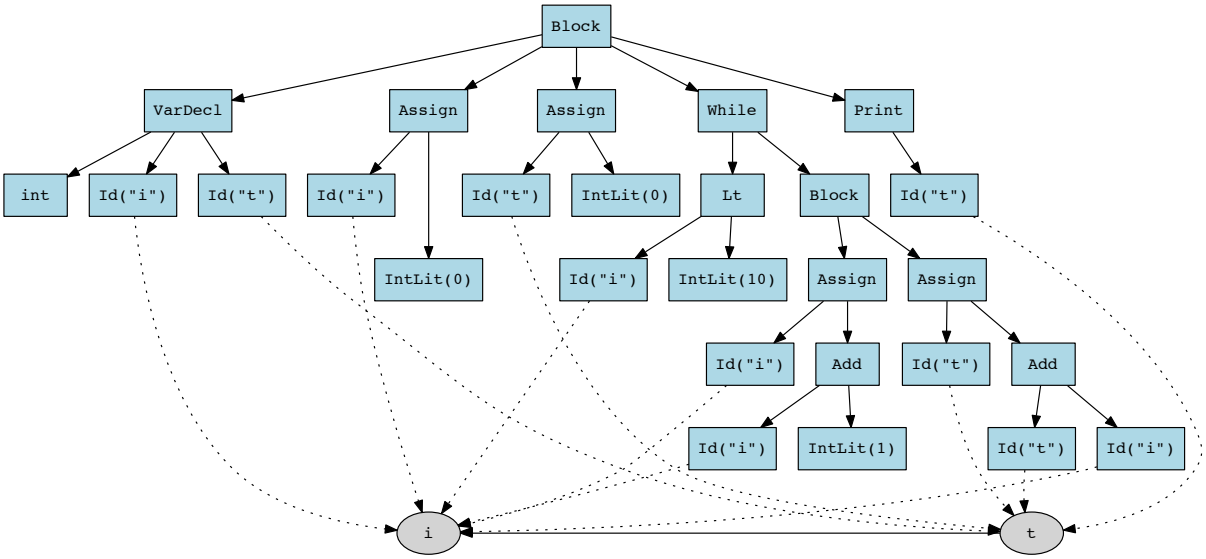


12/split.mini

```
1    int i, j;
2    if (i>0) {
3      boolean b;
4      print j;
5    } else {
6      int j;
7      print i;
8    }
```



Every time the scope analysis encounters an identifier in the AST, it searches for the most recent entry for a variable of that name in the current environment, and adds a pointer to that environment entry from the identifier node. These links are shown as dotted lines in the following diagram.



If scope analysis encounters an identifier that has no corresponding environment entry, then it should report an error.

Drawing environments using dot (13)

Using techniques similar to those that we used to draw abstract syntax trees, we can add a `dotEnv()` method to generate diagrams of the environment data structures that are produced by the scope analysis:

13/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29
30    // Output result:
31    String output = args[0] + "_env.dot";
32    new DotEnvOutput(output).dotEnv(prog);                          // <<<
33    System.out.println("Environment graph output: " + output);
```

The environment diagrams in the previous section were drawn using this version of `Compiler.java`.

Pretty printing to HTML (14)

We can adapt our pretty printer (Version 09) to produce HTML code instead of plain text.

14/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29
30    // Output result:
31    String output = args[0] + ".html";
32    new HTMLOutput(output).toHTML(prog);                              // <<<
33    System.out.println("HTML output: " + output);
```

Using the information provided by the environment data structures, we can distinguish between occurrences of identifiers that define variables, and occurrences of identifiers that correspond to uses of variables. By wrapping appropriate tags around each identifier, and attaching some simple Javascript code, we can arrange for the browser to highlight the definition of a variable when the mouse is over a place where that variable is used:

```
int i, t;
i = 0;
t = 0;
while (i<10) {
    i = i+1;
    t = t+i;
}
print t;
```

We can also arrange for the browser to highlight all uses of a variable when we hover over its definition:

```
int i, t;
i = 0;
t = 0;
while (i<10) {
    i = i+1;
    t = t+i;
}
print t;
```

(This requires a modification to scoping analysis to store a list of variable uses in each environment node.)

Type checking (15)

We can run a type checking analysis over the abstract syntax trees for mini programs. (The links from identifiers to environment objects that were added during scope analysis allow us to determine the type for each variable reference.)

15/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29    new TypeAnalysis(handler).analyze(prog);                          // <<<
30
31    // Output result:
32    System.out.println("Type analysis did not detect any errors");
```

Some quick tests:

```
1   $ java Compiler test    # Check the original test program
2   Type analysis did not detect any errors
3   $ cat bad.mini          # A different test case
4   print true+1;
5   $ java Compiler bad     # Will this one type check?
6   ERROR: "bad.mini", line 1, column 7
7   An expression of type int was expected
8   ERROR: Aborting: errors detected during type checking
9   $
```

Drawing type annotated abstract syntax trees with dot (16)

We can use the information produced during type checking to color code the nodes in AST diagrams:

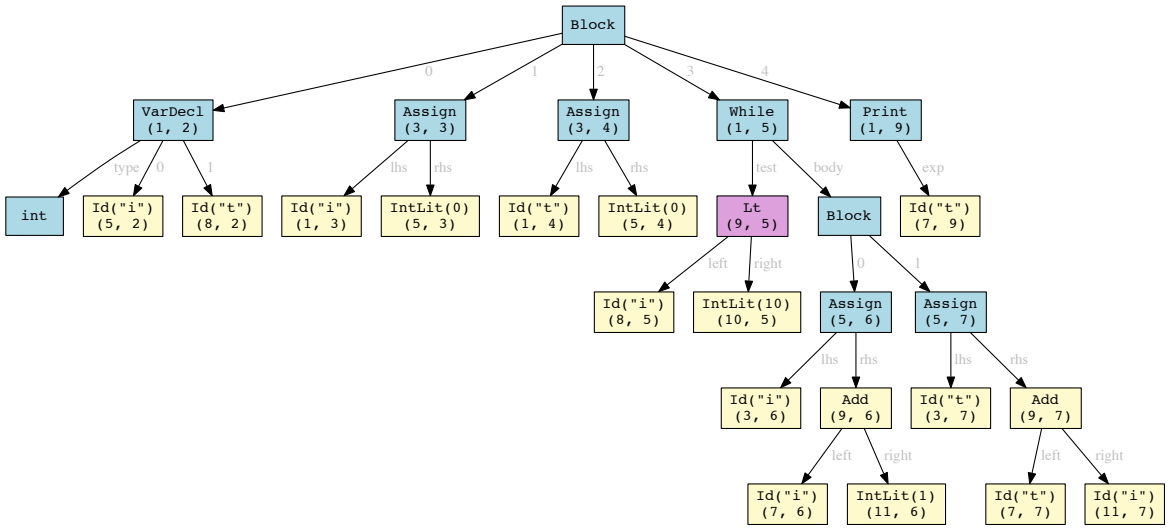
16/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29    new TypeAnalysis(handler).analyze(prog);
30
31    // Output result:
32    String output = args[0] + ".dot";
33    new DotOutput(output).toDot(prog);                                    // <<<
34    System.out.println("AST in dot format output: " + output);
```

Now it only takes a couple of steps to pass our running `test.mini` example through the compiler pipeline, and to generate a pdf version of the graph:

```
1   $ java Compiler test              # generate the dot file
2   AST in dot format output: test.dot
3   $ dot -Tpdf -otest.pdf test.dot   # produce a pdf version of the graph
4   $
```

The resulting, multicolor tree diagram is as follows (integer valued nodes are yellow, boolean nodes are plum, and statement nodes are blue):



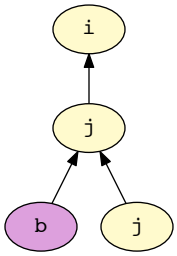
Drawing type annotated environments with dot (17)

We can extend the use of type-based color coding to environment diagrams, placing a call to `dotEnv()` after type checking:

17/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29    new TypeAnalysis(handler).analyze(prog);
30
31    // Output result:
32    String output = args[0] + "_env.dot";
33    new DotEnvOutput(output).dotEnv(prog);                              // <<<
34    System.out.println("Environment graph output: " + output);
```

The resulting environment diagram for the `split.mini` program that we described as part of Version 13 is now rendered as follows, clearly distinguishing the Boolean variable `b` from the other variables, all of which are integer-valued:



Initialization analysis (18)

In this section, we extend our compiler with another simple static analysis that we describe as *initialization analysis*. The purpose of this analysis is to detect places in a program where a variable might have been used without having been previously initialized. Problems of this kind may reflect an error in the logic of the source program, so it can be useful to detect and flag them as errors in the program. In our implementation, this requires one extra static analysis phase:

18/Compiler.java

```
16  // Read program:
17  String     input  = args[0] + ".mini";
18  FileReader reader = new FileReader(input);
19  Source     source = new JavaSource(handler, input, reader);
20  MiniLexer  lexer  = new MiniLexer(handler, source);
21  MiniParser parser = new MiniParser(handler, lexer);
22  Stmt       prog   = parser.parseProgram();
23  if (handler.hasFailures()) {
24    throw new Failure("Aborting: errors detected during syntax analysis");
25  }
26
27  // Analyze program:
28  new ScopeAnalysis(handler).analyze(prog);
29  new TypeAnalysis(handler).analyze(prog);
30  new InitAnalysis(handler).analyze(prog);                          // <<<
31
32  // Output result:
33  System.out.println("Initialization analysis did not detect any errors");
```

This is an example of what is known as a *forwards analysis*, using information that is known at the beginning of the program to compute information about later program points. In this particular case, the information that we want to track is the set of variables that are known to have been initialized at each program point. Initially–that is, before the program starts running—none of the variables have been initialized and so the set is empty. If we encounter an assignment of the form `x=e;` at a point where the set of initialized variables is I, then set of initialized variables immediately after the assignment will be $I \cup \{\texttt{x}\}$. However, if `e` contains any variable that is not listed in I, then we have detected a place where an uninitialized variable has been used, and we can report an error. This covers the treatment of assignment statements, but there are ways to handle each of the other statement forms too, leading to a full initialization analysis.

Of course, all of the variables in `test.mini` are properly initialized, so there are no errors to report:

```
1  $ java Compiler test
2  Initialization analysis did not detect any errors
3  $
```

However, it is not difficult to create a different program that triggers an initialization error:

```
1  $ cat tplus1.mini        # display an alternative test program
2  int t;
3  print t+1;
4  $ java Compiler tplus1   # and then try to compile it
5  ERROR: "tplus1.mini", line 2, column 7
6  The variable "t" may be used before it has been initialized
7  ERROR: Aborting: errors detected during initialization analysis
8  $
```

Simplification of arithmetic expressions (19)

In this section, we add a simple optimization pass that scans the (abstract syntax tree of the) input program and looks for opportunities to simplify the code by using transformations like the following:

- *constant folding*: for example, replacing `19+23` with `42`
- *identity laws*: for example, replacing `x+0` with `x`
- *associativity*: for example, replacing `(x+1)+2` with `x+(1+2)`, and then further simplifying that to `x+3`
- *commutativity*: for example, replacing `(1+x)+2` with `(x+1)+2`, and then further simplifying that to `x+3`

We add a call to the resulting `simplify()` phase after the various static analysis phases are complete (there is no point in attempting to simplify a program that does not even pass static analysis):

19/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29    new TypeAnalysis(handler).analyze(prog);
30    new InitAnalysis(handler).analyze(prog);
31
32    // Display program before simplification pass:
33    System.out.println("Program BEFORE simplification:");              // <<<
34    new TextOutput(System.out).toText(prog);
35
36    prog.simplify();                                                   // <<<
37
38    // Display program after simplification pass:
39    System.out.println("Program AFTER simplification:");               // <<<
```

Our running example, `test.mini`, does not contain any code that can be improved by the simplification algorithm. We can demonstrate its effects, however, with a new test program, `fortytwo.mini`, that includes some (gratuitious) opportunities for simplification:

```
1   $ java Compiler fortytwo
2   Program BEFORE simplification:
3   int i;
4   i = 21+21;
5   print (41*(7-6))+(i+1);
6   print ((((1+2)+3)+4)+5)+i;
7   print ((((i+1)+2)+3)+4)+5;
8
9   Program AFTER simplification:
```

```
10      int i;
11      i = 42;
12      print i+42;
13      print i+15;
14      print i+15;
15
16      $
```

Evaluation: an interpreter for the mini language (20)

We can build a *interpreter* for mini by adding an additional component after static analysis and optimization to run the input program:

20/Compiler.java

```
16    // Read program:
17    String     input  = args[0] + ".mini";
18    FileReader reader = new FileReader(input);
19    Source     source = new JavaSource(handler, input, reader);
20    MiniLexer  lexer  = new MiniLexer(handler, source);
21    MiniParser parser = new MiniParser(handler, lexer);
22    Stmt       prog   = parser.parseProgram();
23    if (handler.hasFailures()) {
24      throw new Failure("Aborting: errors detected during syntax analysis");
25    }
26
27    // Analyze program:
28    new ScopeAnalysis(handler).analyze(prog);
29    new TypeAnalysis(handler).analyze(prog);
30    new InitAnalysis(handler).analyze(prog);
31
32    // Optimization:
33    prog.simplify();
34
35    // Execute program:
36    System.out.println("Ready to run the program:");                    // <<<
37    prog.exec();                                                        // <<<
38    System.out.println("Done!");                                        // <<<
```

Quick test:

```
1    $ java Compiler test
2    Ready to run the program:
3    55
4    Done!
5    $
```

It is easy to calculate that `1+2+3+4+5+6+7+8+9+10 = 55`, and hence to confirm that our interpreter is working correctly, at least in this simple case.

Compilation (21)

In this section, we will complete our compiler pipeline, enabling the translation of valid mini source programs into executable programs.

Before we go further, there are two technical asides:

1. Our compiler produces code for computers that use the IA32 (32 bit) instruction set, which is broadly supported by recent Windows, Mac OS X, and Linux based laptops and desktops. Although the compiler itself will run on any computer that supports Java, it will not be possible to execute the compiled code on machines that do not support the IA32 instruction set.

2. The methods that we use to generate code vary in small ways between different platforms. Before attempting to compile and build this final version of `Compiler.java`, you should edit the code on Line 20 of `ast/IA32.java` to ensure that it has the correct platform setting for your computer. The currently supported platforms are: `LINUX`, `MACOSX`, and `WINDOWS`.

Our final version of `Compiler.java` replaces the code for executing programs that we saw in the interpreter (Version 20) with code for: generating an assembly language translation of the original source program (Lines 35–38); and invoking `gcc` to run the assembler and link the resulting object code with a simple runtime system (Lines 40–45) to produce an executable program:

21/Compiler.java

```
16  // Read program:
17  String     input  = args[0] + ".mini";
18  FileReader reader = new FileReader(input);
19  Source     source = new JavaSource(handler, input, reader);
20  MiniLexer  lexer  = new MiniLexer(handler, source);
21  MiniParser parser = new MiniParser(handler, lexer);
22  Stmt       prog   = parser.parseProgram();
23  if (handler.hasFailures()) {
24    throw new Failure("Aborting: errors detected during syntax analysis");
25  }
26
27  // Analyze program:
28  new ScopeAnalysis(handler).analyze(prog);
29  new TypeAnalysis(handler).analyze(prog);
30  new InitAnalysis(handler).analyze(prog);
31
32  // Optimization:
33  prog.simplify();
34
35  // Output compiled program:
36  String output = args[0] + ".s";
37  new IA32(output).generateAssembly(output, prog);                  // <<<
38  System.out.println("Assembly code output: " + output);
39
40  // Invoke assembler to produce executable:
41  Runtime.getRuntime()                                              // <<<
42         .exec("gcc -m32 -o " + args[0] + " " +                     // <<<
43                 args[0] + ".s runtime.c")                          // <<<
44         .waitFor();                                                // <<<
45  System.out.println("Executable program: " + args[0]);
```

The following sample run demonstrates three steps: (1) compiling `test.mini`; (2) inspecting the generated

assembly code; and (3) running the executable version:

```
$ java Compiler test    # (1) compile the test program
Assembly code output: test.s
Executable program: test
$ cat test.s            # (2) inspect the generated assembly code
        .file   "test.s"
        .globl  _Main_main
_Main_main:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $8,%esp
        movl    $0,%eax
        movl    %eax,-4(%ebp)
        movl    $0,%eax
        movl    %eax,-8(%ebp)
        jmp     l1
l0:
        movl    $1,%eax
        movl    -4(%ebp),%ecx
        addl    %ecx,%eax
        movl    %eax,-4(%ebp)
        movl    -4(%ebp),%eax
        movl    -8(%ebp),%ecx
        addl    %ecx,%eax
        movl    %eax,-8(%ebp)
l1:
        movl    $10,%eax
        movl    -4(%ebp),%ecx
        cmpl    %eax,%ecx
        jl      l0
        subl    $12,%esp
        movl    -8(%ebp),%eax
        pushl   %eax
        call    _print
        movl    %ebp,%esp
        popl    %ebp
        ret
$ ./test                # (3) run the compiled executable
print: 55
$
```

You are not expected to understand the details of the generated assembly code at this stage. You should observe that the final result that is produced by the executable is the same as the value that was produced by the interpreter in the previous section!