01/18/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)
Jenna Wise
`jlwise@andrew.cmu.edu`

1 Learning Objectives

After this recitation you should be able to ...

- Relate and differentiate
  - Concrete syntax and abstract syntax
  - The WHILE abstract syntax and the WHILE3ADDR abstract syntax
  - Big-step operational semantics and small-step operational semantics
  - Program representation, program semantics, and syntactic analysis
- Formulate and use a syntactic analysis
- Prove that a program, starting in a particular state, will evaluate to a particular program state or configuration
- Set up the Soot analysis infrastructure in your environment and successfully compile and run an analysis

2 Reminders

- Homework 1 is due next **Thursday, January 24, 2019 at 11:59pm**. Instructions and the starter code can be found on the course website
- Email me your **Andrew ID** and **GitHub username** at **jlwise@andrew.cmu.edu**, so I can set-up your private GitHub repository where you will submit your code for the coding homework assignments

3 Weekly Lecture Review

3.1 Program Representation

3.1.1 Concrete Syntax

Defines the rules by which programs can be expressed as strings of characters (surface level of a language; what the programmer sees; tied to a particular representation)

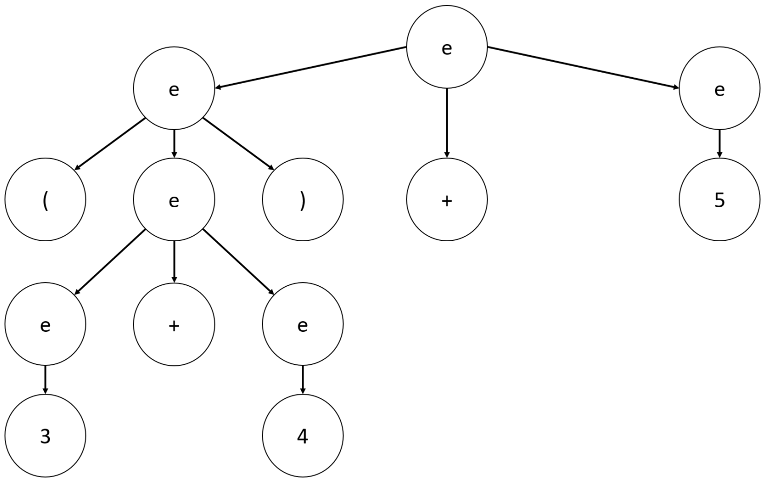
Example:
The same sum expression can be encoded in many different ways, which must be reflected in the concrete syntax (infix vs. prefix vs. postfix).

$$\begin{array}{lcl} e & ::= & n \mid (e) \mid e + e \\ e & ::= & n \mid (e) \mid + \; e \; e \\ e & ::= & n \mid (e) \mid e \; e \, + \end{array}$$

where n is an integer literal

Example parse tree for:

$(3 + 4) + 5$



3.1.2 Abstract Syntax

Defines the deep structure of a language, independent of any particular representation or encoding (the way programs look like to the evaluator/compiler)

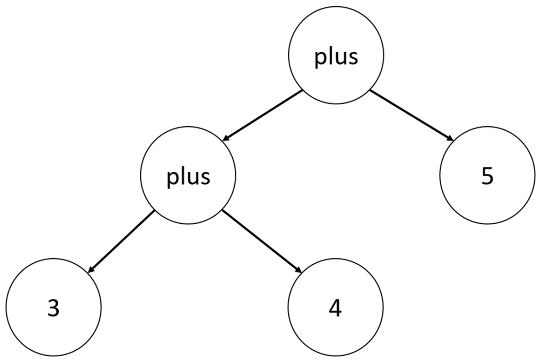
Example:
The abstract syntax representation for the sum expression is independent of any particular encoding; a sum expression always has two operand expressions as its significant parts regardless of its concrete syntax representation.

$$e \;::=\; n \mid \text{plus}(e, e)$$

where n is an integer literal

Example abstract syntax tree for:

$\text{plus}(\text{plus}(3, 4), 5)$



3.1.3 WHILE Abstract Syntax

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
P	boolean predicates (BExp)

$$
\begin{array}{lll}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } P \text{ do } S
\end{array}
\qquad
\begin{array}{lll}
P & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } P \\
 & | & P_1\ op_b\ P_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lll}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lll}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \ \mid\ \leq \ \mid\ = \\
 & | & > \ \mid\ \geq \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

3.1.4 WHILE3ADDR Abstract Syntax

I	instructions
x, y	program variables (Vars)
n	number literals

$$
\begin{array}{lll}
I & ::= & x := n \\
 & | & x := y \\
 & | & x := y\ op\ z \\
 & | & \text{goto } n \\
 & | & \text{if } x\ op_r\ 0 \text{ goto } n
\end{array}
\qquad
\begin{array}{lll}
op & ::= & + \mid - \mid * \mid / \\
op_r & ::= & < \ \mid\ \leq \ \mid\ = \\
 & | & > \ \mid\ \geq \\
P & \in & \mathbb{N} \to I
\end{array}
$$

The WHILE3ADDR abstract syntax is a simpler more uniform version of the WHILE abstract syntax. Programs written in the WHILE3ADDR language are easier to analyze than if they were written in the WHILE language. You will be writing analyses to process programs written in JIMPLE, an extended version of WHILE3ADDR, for your first homework.

3.2 Syntactic Analysis

A syntactic analysis processes a program representation, often to determine errors in programs.

Example: Bad shift analysis

Processes programs written in JIMPLE, a typed 3-address intermediate representation suitable for analyzing Java programs (the analyses that you will write for your homework may also analyze programs written in JIMPLE).

```
For each instruction I in the program
    if (I is a shift instruction)
        if (type of the left operand of I is int
             && the right operand of I is a constant
             && value of constant < 0 or > 31)
            warn("Shifting by less than 0 or more than 31 is
               meaningless")
```

3.3 Program Semantics

A program's semantics define its meaning. There are three main classes of formal semantics denotational, operational, and axiomatic. Yesterday's lecture covered operational semantics and its two classes: big-step operational semantics and small-step operational semantics.

Operational semantics mimics, at a high level, the operation of a computer executing the program.

Both big-step and small-step operational semantics depend on a program state $E \in \mathit{Var} \to \mathbb{Z}$, which maps program variables to their corresponding integer values.

3.3.1 Big-step Operational Semantics

Big-step operational semantics specifies the entire operation of a given expression or statement. It requires that inference rules consist of the $\Downarrow$ judgment, which specifies program meaning as a function between a program configuration ($\langle S, E\rangle$) and a new state (E') (all together: $\langle S, E\rangle \Downarrow E'$).

For example for the WHILE language, example inference rules defining the entire operation of a variable expression and an assignment statement are:

$$\frac{}{\langle x, E\rangle \Downarrow E(x)}\ \textit{big-var} \qquad \frac{\langle e, E\rangle \Downarrow n}{\langle x := e, E\rangle \Downarrow E[x \mapsto n]}\ \textit{big-assign}$$

3.3.2 Small-step Operational Semantics

Small-step operational semantics specifies the operation of a program one step at a time. It requires that inference rules are repeatedly applied to configurations until a final configuration ($\langle \texttt{skip}, E\rangle$) is reached (if it can be reached). The inference rules consist of the $\langle S, E\rangle \to \langle S', E'\rangle$ or $\langle S, E\rangle \to^* \langle S', E'\rangle$ judgments, which indicates one step of execution or zero or more steps of execution respectively.

For example for the WHILE language, example inference rules defining a single step operation of a variable expression and an assignment statement are:

$$\frac{}{\langle x, E\rangle \to_a E(x)}\ \textit{small-var}$$

$$\frac{\langle e, E\rangle \to_a e'}{\langle x := e, E\rangle \to \langle x := e', E\rangle}\ \textit{assign-congruence} \qquad \frac{}{\langle x := n, E\rangle \to \langle \texttt{skip}, E[x \mapsto n]\rangle}\ \textit{assign-int}$$

3.3.3 Derivations & Proof Techniques

We can prove that concrete program expressions will evaluate to particular values and concrete program statements will evaluate to particular program states (in the case of big-step operational semantics) or configurations (in the case of small-step operational semantics). This is done by chaining together rules of inference into *derivations*, for example:

$$\dfrac{\dfrac{\langle 4, E_1\rangle \Downarrow 4 \quad \langle 2, E_1\rangle \Downarrow 2}{\langle 4 * 2, E_1\rangle \Downarrow 8} \qquad \langle 6, E_1\rangle \Downarrow 6}{\langle (4 * 2) - 6, E_1\rangle \Downarrow 2}$$

Therefore, we have proven that $(4 * 2) - 6$ evaluates to 2 starting in the program state E_1.

We will not review proof techniques using operational semantics, because we will review it in significant detail next recitation in relation to `hw2`.

4 Homework Help

4.1 Soot Introduction

Soot is a Java optimization framework for Java bytecode implemented in Java. It supports four different intermediate representations: Java Bytecode, BAF, JIMPLE, and GRIMP. Soot provides a Java API for writing optimizations and analyses on Java bytecode in these forms. You will analyze programs written in the JIMPLE intermediate representation, which is a typed and compact 3-address code representation of bytecode.

4.2 Homework Environment Set-up & Execution

4.2.1 Set-up

- Install the Java Development Kit version 7 or 8 (if necessary)
  - Make sure it is on your path when using the command-line or is usable by Eclipse or IntelliJ (if you are using either IDE)
- Install `ant` (if necessary)
  - Both Eclipse and IntelliJ ship with `ant`, so you don't need to take any additional actions to install it when using either IDE
  - The Andrew Unix machines, `unix.andrew.cmu.edu`, have both the JDK and `ant` already installed
- Download hw1.zip from the course website and unzip it
  - You can unzip it directly to your cloned GitHub repository
  - Import it into Eclipse or IntelliJ (if using either IDE)
- Execute the `build.xml` file in the `hw1` directory to make sure everything works
  - From the command-line, run `ant build`; the project should build successfully and report: `BUILD SUCCESSFUL`

  - From Eclipse or IntelliJ, run the `ant` build file in the way appropriate for the IDE you are using (it is okay if it both builds the project and executes the tests in this step); if this step is successful you may see either `BUILD SUCCESSFUL` or `BUILD FAILED` depending on if you just built the project or if you both built the project and executed the tests respectively

4.2.2 Execution

- Execute the `build.xml` file in the `hw1` directory to both build the project and run the tests
  - From the command-line, run `ant`
  - From Eclipse or IntelliJ, run the `ant` build file in the way appropriate for either IDE to both build the project and execute the tests
  - After compilation succeeds, you will get a `BUILD FAILED` message indicating the tests failed (you will make the tests pass by completing the assignment)

4.2.3 Running Soot Manually from the Command-line

All of the commands below should be run in the `hw1` directory. All of the commands except for the first one should be run only after running `ant build`.

- To test-run Soot, run:

```
java -cp lib/soot-trunk.jar soot.Main --help
```

- To produce JIMPLE output for a class such as `edu.cmu.se355.hw1.Shifty`, use:

```
java -cp lib/soot-trunk.jar soot.Main -cp "build:lib/rt.jar" -f
    J edu.cmu.se355.hw1.Shifty
```

- To run the example PrintAnalysis on the same class, use:

```
java -cp "build:lib/soot-trunk.jar" edu.cmu.se355.hw1.
   PrintAnalysisMain -cp "build:lib/rt.jar" -p jap.
   printanalysis on edu.cmu.se355.hw1.Shifty
```

- The main analysis files for bad shift analysis and unread field analysis also specify unit tests, using the JUnit framework. The unit tests are run by the ant build script (`build.xml`), but you can also run them manually. The command for the bad shift analysis unit test is:

```
java -cp "build:lib/soot-trunk.jar:lib/junit-4.11.jar:lib/
   hamcrest-core-1.3.jar" org.junit.runner.JUnitCore edu.cmu.
   se355.hw1.ShiftAnalysisMain
```

*****Note:** The above command lines have been tested on Linux, but they will be different if you are using Windows: you will need to replace / with \ and replace : with ;

4.3 Helpful Resources

- Soot Wiki
- The Soot Survivor's Guide
- Sable Thesis detailing Soot, includes a useful description of the JIMPLE intermediate representation
- Soot's Javadocs