Lecture Notes:
The WHILE and WHILE3ADDR Language

15-819O: Program Analysis
Claire Le Goues
clegoues@cs.cmu.edu

1 The WHILE Language

In this course, we will study the theory of analyses using a simple programming language called WHILE, along with various extensions. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties (to be discussed in a later lecture). It is a simple imperative language, with (to start!) assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

We use the following metavariables to describe different categories of syntax. The letter on the left will be used as a variable representing a piece of a program. On the right, we describe the kind of program piece that variable represents:

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

The syntax of WHILE is shown below. Statements S can be an assignment $x := a$, a skip statement, which does nothing (similar to a lone semicolon or open/close bracket in C or Java), and `if` and `while` statements, whose condition is a boolean predicate P. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators, abstractly represented by op_a. Boolean expressions include true, false, the negation of another boolean expression, boolean operators op_b applied to other boolean expressions, and relational operators op_r applied to arithmetic expressions.

$$
\begin{array}{rcl@{\qquad}rcl}
S & ::= & x := a & a & ::= & x \\
 & | & \text{skip} & & | & n \\
 & | & S_1;\ S_2 & & | & a_1\ op_a\ a_2 \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 & & & \\
 & | & \text{while } P \text{ do } S & op_a & ::= & + \mid - \mid * \mid / \\
\end{array}
$$

$$
\begin{array}{lcl}
P & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } P \\
 & | & P_1\ op_b\ P_2 \\
 & | & a_1\ op_r\ a_2 \\
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leq \mid = \mid > \mid \geq
\end{array}
$$

2 WHILE3ADDR: A Representation for Analysis

For analysis, the source-like definition of WHILE can sometimes prove inconvenient. For example, WHILE has three separate syntactic forms—statements, arithmetic expressions, and boolean predicates—and we would have to define the semantics of each separately to reason about it. A simpler and more regular representation of programs will help simplify certain of our formalisms.

As a starting point, we will eliminate recursive arithmetic and boolean expressions and replace them with simple atomic statement forms, which are called *instructions*, after the assembly language instructions that they resemble. For example, an assignment statement of the form $w = x * y + z$ will be rewritten as a multiply instruction followed by an add instruction. The multiply assigns to a temporary variable t_1, which is then used in the subsequent add:

$$
\begin{array}{l}
t_1 = x * y \\
w = t_1 + z
\end{array}
$$

As the translation from expressions to instructions suggests, program analysis is typically studied using a representation of programs that is not only simpler, but also lower-level than the source (WHILE, in this instance) language. Many Java analyses are actually conducted on byte code, for example. Typically, high-level languages come with features that are numerous and complex, but can be reduced into a smaller set of simpler primitives. Working at the lower level of abstraction thus also supports simplicity in the compiler.

Control flow constructs such as `if` and `while` are similarly translated into simpler `goto` and `conditional branch` constructs that jump to a particular (numbered) instruction. For example, a statement of the form `if` P `then` S_1 `else` S_2 would be translated into:

$$
\begin{array}{ll}
1: & \text{if } P \text{ then goto } 4 \\
2: & S_2 \\
3: & \text{goto } 5 \\
4: & S_1 \\
5: & \textit{rest of program...}
\end{array}
$$

How would you translate While statements?

This form of code is often called 3-address code, because every instruction has at most two source operands and one result operand. We now define the syntax for 3-address code produced from the WHILE language, which we will call WHILE3ADDR. This language consists of a set of simple instructions that load a constant into a variable, copy from one variable to another, compute the value of a variable from two others, or jump (possibly conditionally) to a new address n. A program P is just a map from addresses to instructions:[1]

[1]The idea of the mapping between numbers and instructions maps conceptually to Nielsens' use of *labels* in the

$$
\begin{array}{rcll}
I & ::= & x := n & \quad op ::= + \mid - \mid * \mid / \\
 & \mid & x := y & \quad op_r ::= < \mid = \\
 & \mid & x := y \; op \; z & \quad P \in \mathbb{N} \to I \\
 & \mid & \text{goto } n & \\
 & \mid & \text{if } x \; op_r \; 0 \text{ goto } n &
\end{array}
$$

Formally defining a translation from a source language such as WHILE to a lower-level intermediate language such as WHILE3ADDR is possible, but more appropriate for the scope of a compilers course. For our purposes, the examples above should suffice as intuition. We will focus instead on semantics and on formalizing program analyses.

3 Operational Semantics

To reason about the correctness of an analysis, we need a clear definition of what a program *means*. There are many ways of giving such definitions; the most common technique in industry is to define a language using an English document, such as the Java Language Specification. However, natural language specifications, while accessible to all programmers, are often imprecise. This imprecision can lead to many problems, such as incorrect or incompatible compiler implementations, but more importantly for our purposes, analyses that give incorrect results.

A better alternative is a formal definition of program semantics. We begin with *operational semantics*. which mimics, at a high level, the operation of a computer executing the program, including a program counter, values for program variables, and (eventually) a representation of the heap. Such a semantics also reflects the way that techniques such as dataflow analysis or Hoare Logic reason about the program, so it is convenient for our purposes.

There are two broad classes of operational semantics: *big-step operational semantics*, which specifies the entire operation of a given expression or statement; and *small-step operational semantics*, which specifies the operation of the program one step at a time.[2]

3.1 WHILE: Big-step operational semantics

Let's start with WHILE expressions, restricting our attention to arithmetic expressions for simplicity. What is the meaning of a WHILE expression? Some expressions, like a natural number, have a very clear meaning: The "meaning" of 5 is just, well, 5. But what about $x + 5$? The meaning of this expression clearly depends on the value of the variable x. We must *abstract* the value of variables as a function from variable names to integer values:

$$\mathcal{E} \in \mathit{Var} \to \mathbb{Z}$$

We use E to range over $\mathcal{E}$, denoting a particular program *state*. The meaning of an expression using a variable, like $x + 5$, involves "looking up" the variable value in the associated E, and substituting it in. Given a state, we can then write a *judgement* as follows:

$$< e, E > \Downarrow n$$

WHILE language specification in the text. This concept is akin to mapping line numbers to code.

[2]This is just an introduction, and does not cover all of the ways to specify or use operational semantics; I'm covering enough to support a conversation about abstract interpretation and correctness.

This means that the expression e evaluates to n in state E. This formulation is called *big-step* operational semantics; the $\Downarrow$ judgement relates an expression and its "meaning." We then build up the meaning of more complex expressions using *rules of inference* (also called *derivation* or *evaluation* rules). An inference rule is made up of a set of judgments above the line, known as premises, and a judgment below the line, known as the conclusion. The meaning of an inference rule is that the conclusion holds if all of the premises hold:

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

An inference rule with no premises is an axiom; axioms are always true. For example, integers always evaluate to themselves, and the meaning of a variable is its stored value in the state:

$$\frac{}{< n, E >\Downarrow n}\ ints \qquad\qquad \frac{}{< x, E >\Downarrow E(x)}\ vars$$

Addition expressions illustrate a rule with premises:

$$\frac{< e_1, E >\Downarrow n_1 \quad < e_2, E >\Downarrow n_2}{< e_1 + e_2, E >\Downarrow n_1 + n_2}\ add$$

How does the value of x come to be "stored" in E? For that, we must consider WHILE *Statements*. Unlike expressions, statements have no direct result. However, they can have *side effects*. That is to say: the "result" or *meaning* of a Statement is a *new state*. The judgement $\Downarrow$ as applied to statements and states therefore looks like:

$$< S, E >\Downarrow E'$$

We refer to the pair of a statement and a state ($< S, E >$) as a *configuration*; this will become important when we get to the next section. We can now write rules of inference for statements, bearing in mind that their *meaning* is not an integer, but a new state. The meaning of the `skip` statement, for example, is an unchanged state:

$$\frac{}{< \texttt{skip}, E >\Downarrow E}$$

Statement sequencing, on the other hand, does involve premises:

$$\frac{< s_1, E >\Downarrow E' \quad < s_2, E' >\Downarrow E''}{< s_1; s_2, E >\Downarrow E''}$$

The `if` statement involves two rules, one for if the boolean predicate evaluates to `true` (rules for boolean expressions not shown), and one for if it evaluates to `false`. I'll show you just the first one for demonstration:

$$\frac{< b, E >\Downarrow \texttt{true} \quad < s_1, E >\Downarrow E'}{< \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E >\Downarrow E'}$$

What should the second rule for `if` look like? Write it down!

This brings us to assignments, which actually update the state:

$$\frac{< e, E >\Downarrow n}{< x := e, E >\Downarrow E[x := n]}$$

To fully specify the semantics of a language, one must provide a judgement rule like this for every language construct. Clearly, these notes only include a subset, for brevity!

3.2 Small-step operational semantics

Big-step operational semantics has its uses. Among other nice features, it directly suggests a simple interpreter implementation for a given language. However, it is difficult to talk about a statement or program whose evaluation *does not terminate*. Nor does it give us any way to talk about intermediate states (so modeling multiple threads of control is out).

When we need this power, we can instead use a *small-step operational semantics*, sometimes called *structural* operational semantics. Small-step semantics specifies program execution one step at a time. Where big step semantics specifies program meaning as a function between a configuration and a new state, small step semantics models it as a step from one program configuration to another.

You can think of small-step semantics as a set of rules that we repeatedly apply to configurations until we reach a *final configuration* for the language ($< \texttt{skip}, E >$, in this case) if ever.[3] We write this new judgement using a slightly different arrow: $\rightarrow$. $< S, E > \rightarrow < S', E' >$ indicates one step of execution; $< S, E > \rightarrow^* < S', E' >$ indicates zero or more steps of execution. To be complete, we should also define auxiliary small-step operators $\rightarrow_a$ and $\rightarrow_b$ for arithmetic and boolean expressions, respectively; only the operator for statements results in an updated state (as in big step, above). The types are thus:

$$\begin{array}{lcl} \rightarrow & : & (\texttt{Stmt} \times E) \rightarrow (\texttt{Stmt} \times E) \\ \rightarrow_a & : & (\texttt{Aexp} \times E) \rightarrow \texttt{Aexp} \\ \rightarrow_b & : & (\texttt{Bexp} \times E) \rightarrow \texttt{Bexp} \end{array}$$

We can now again write the semantics of a WHILE program as rules of inference. Some rules look very similar to the big-step rules, just with a different arrow. For example, consider variables:

$$\frac{}{< x, E > \rightarrow_a E(x)}$$

Things get more interesting when we return to statements. Remember, small-step semantics express a single execution step. So, consider an `if` statement:

$$\frac{< b, E > \rightarrow_b b'}{< \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, E > \rightarrow < \texttt{if } b' \texttt{ then } S_1 \texttt{ else } S_2, E >}$$

$$\frac{}{< \texttt{if true then } s_1 \texttt{ else } s_2, E > \rightarrow < s_1, E' >}$$

I have again omitted the `if-false` case, as an exercise to the reader; suffice to say, there's one more `if` rule. Regardless, contrast these rules with the big-step rules, above. *What's the difference?*

3.3 Small-step semantics for WHILE3ADDR

The ideas behind big- and small-step operational semantics are consistent across languages, but the way they are written can vary based on what is notationally convenient for a particular language or analysis. WHILE3ADDR is slightly different from WHILE, so beyond requiring different rules for its different constructs, it makes sense to modify our small-step notation a bit for defining the meaning of a WHILE3ADDR program.

First, let's revisit the *configuration* to account for the slightly different *meaning* of a WHILE3ADDR program. As before, the configuration must include the state, which we still call E, mapping variables to values. However, a well-formed, terminating WHILE program was effectively a single

[3] Not all statements reach a final configuration, like `while true do skip`.

statement that can be iteratively reduced to `skip`; a WHILE3ADDR program, on the other hand, is a mapping from natural numbers to program instructions. So, instead of a statement that is being reduced in steps, the WHILE3ADDR c must includes a program counter n, representing the next instruction to be executed.

Thus, a configuration c of the abstract machine for WHILE3ADDR must include the stored program P (which we will generally treat implicitly), the state environment E, and the current program counter n representing the next instruction to be executed ($c \in E \times \mathbb{N}$). The abstract machine executes one step at a time, executing the instruction that the program counter points to, and updating the program counter and environment according to the semantics of that instruction.

This adds a tiny bit of complexity to the inference rules, because they must explicitly consider the mapping between line number/labels and program instructions. We represent execution of the abstract machine via a judgment of the form $P \vdash < E, n > \rightsquigarrow < E', n' >$ The judgment reads: "When executing the program P, executing instruction n in the state E steps to a new state E' and program counter n'."[4] To see this in action, consider a simple inference rule defining the semantics of the constant assignment instruction:

$$\frac{P[n] = x := m}{P \vdash < E, n > \rightsquigarrow E[x \mapsto m], n+1} \textit{ step-const}$$

This states that in the case where the nth instruction of the program P (looked up using $P[n]$) is a constant assignment $x := m$, the abstract machine takes a step to a state in which the state E is updated to map x to the constant m, written as $E[x \mapsto m]$, and the program counter now points to the instruction at the following address $n + 1$. We similarly define the remaining rules:

$$\frac{P[n] = x := y}{P \vdash < E, n > \rightsquigarrow E[x \mapsto E[y]], n+1} \textit{ step-copy}$$

$$\frac{P[n] = x := y \; op \; z \quad E[y] \; \mathbf{op} \; E[z] = m}{P \vdash < E, n > \rightsquigarrow E[x \mapsto m], n+1} \textit{ step-arith}$$

$$\frac{P[n] = \text{goto } m}{P \vdash < E, n > \rightsquigarrow < E, m >} \textit{ step-goto}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = true}{P \vdash < E, n > \rightsquigarrow < E, m >} \textit{ step-iftrue}$$

$$\frac{P[n] = \text{if } x \; op_r \; 0 \text{ goto } m \quad E[x] \; \mathbf{op_r} \; 0 = false}{P \vdash E, n \rightsquigarrow E, n+1} \textit{ step-iffalse}$$

[4] I could have used the same $\rightarrow$ I did above instead of $\rightsquigarrow$, but I don't want you to mix them up.