Lecture Notes:
The WHILE Language and WHILE3ADDR Representation

15-819O: Program Analysis
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

Lecture 2

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 assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

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

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

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.

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

2 WHILE3ADDR: A Representation for Analysis

We could define the semantics of WHILE directly—and indeed we will do so when studying Hoare Logic.[1] For program analysis, however, the source-like definition of WHILE is somewhat inconvenient. Even a simple language such as WHILE can be complex to define. 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. A simpler and more regular representation of programs will help make our formalism simpler.

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 $w = x * y + z$

[1]The supplemental Nielson et al. text also defines the semantics of the WHILE language given here.

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:

$$\begin{aligned} t_1 &= x * y \\ w &= t_1 + z \end{aligned}$$

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 language. 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:

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

The translation of a statement of the form while P do S is similar:

$$\begin{array}{rl} 1: & \text{if not } P \text{ goto } 4 \\ 2: & S \\ 3: & \text{goto } 1 \\ 4: & \textit{rest of program...} \end{array}$$

This form of code is often called 3-address code, because every instruction is of a simple form with at most two source operands and one result operand. We now define the syntax for 3-address code produce 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 is just a map from addresses to instructions:

$$
\begin{array}{lll}
I & ::= & x := n \\
 & | & x := y \\
 & | & x := y \mathit{ op } z \\
 & | & \text{goto } n \\
 & | & \text{if } x \mathit{ op}_r \ 0 \text{ goto } n \\
\mathit{op} & ::= & + \mid - \mid * \mid / \\
\mathit{op}_r & ::= & < \ \mid \ = \\
P & \in & \mathbb{N} \rightharpoonup I
\end{array}
$$

Formally defining a translation from a source language such as WHILE to a lower-level intermediate language such as WHILE3ADDR is possible, but it is more appropriate for the scope of a compilers course. For the purposes of this course, the examples above should suffice as intuition. We will proceed by first formalizing program analysis in WHILE3ADDR, then having a closer look at its semantics in order to verify the correctness of the program analysis.