1 The İzmir language

The İzmir language is a very simple untyped\(^1\) language with integer values and global variables.

The İzmir language is designed to be easy to compile. The code linked from https://www.gnu.org/ghm/2022/#workshop contains a working parser, and is designed to be completed with:

- a compiler generating İzmirVM code;
- a working İzmirVM virtual machine, generated by Jitter.

The build system is already given and does not need to be modified.

1.1 İzmir syntax

The İzmir language features expressions and statements: an expression serves to compute a value: every expression one result. A statement does not compute a result, but has an effect: either changing the value of a variable or printing a value.

An İzmir-language program is a sequence of statements.

1.1.1 Expressions

Let \( n \) be an integer number such as 3, -1 or 42.
Let \( b \) be the Boolean constant true or false.
Let \( x \) be a variable name such as \( x \), \( y \) or \( foo \).

Any number is an expression:
\[
e ::= n
\]

Any Boolean constant is an expression:
\[
e ::= b
\]
Notice that Boolean constants are effectively integers, and can be freely mixed and combined with them.

Any variable is also an expression:
\[
e ::= x
\]

Given two expressions, their sum is an expression:
\[
e ::= e + e
\]

The same holds for subtraction, multiplication, division and remainder:
\[
e ::= e - e
\]
\[
e ::= e \ast e
\]
\[
e ::= e / e
\]

\(^1\)There is no difference between integers and Booleans: an expression such as false + 3 is considered to be correct.
Given one expression its negative version is also an expression:
\[ e ::= - e \]

Boolean constants (true and false) are expressions:
We can also use logic operators to build expressions. Given an expression its logical negation is also an expression:
\[ e ::= \text{not } e \]

Given two expressions their logical conjunction (logical “and”) and logical disjunction (logical “or”) are also expressions:
\[ e ::= e \text{ and } e \]
\[ e ::= e \text{ or } e \]

Comparison operators between integers build Booleans values. Comparison operators are also used to build expressions:
\[ e ::= e = e \]
\[ e ::= e \neq e \]
\[ e ::= e < e \]
\[ e ::= e > e \]
\[ e ::= e \leq e \]
\[ e ::= e \geq e \]

1.1.2 Statements
The empty statement skip, which does nothing, is a statement:
\[ s ::= \text{skip;} \]

The assignment statement, which evaluates an expression and assigns it to a variable, is a statement:
\[ s ::= x := e ; \]

The printing statement, which evaluates an expression and prints it to the standard output, is a statement:
\[ s ::= \text{print } e ; \]

Given two statements, their sequential composition (which means executing one after the other) is also a statement:
\[ s ::= s \quad s \]
Notice that, since every statement ends with a semicolon character (’;’), there will always be a semicolon separating the first and the second statement in a sequential composition.

Given an expression and a statement we can build from them a while loop by using the expression as the guard and the statement as the body: the while statement execution consists in executing the body repetedly, as long as the guard evaluates to a true result:
\[ s ::= \text{while } e \text{ do } s \text{ end;} \]
1.2 Compilation rules of the *İzmir* into the *İzmirVM* virtual machine

The style of compilation presented here is *compositional*: compiling a language phrase consists in compiling all of its subphrases, plus occasionally some additional work.

1.2.1 Compiling expressions

We compile a constant by pushing it onto the stack:

\[ \text{pushconstant } n \]
\[ \text{pushconstant } 1 \]
\[ \text{pushconstant } 0 \]

If the variable \( x \) is held in the register \( r_x \) we compile the expression \( x \) by pushing the value of the register \( r_x \):

\[ \text{pushregister } r_x \]

Unary-operator expressions are compiled by first compiling the sub-expression, with one more instruction after it; the one instruction after it pops one element from the stack and pushes another element in its place:

\[ \text{unaryminus} \]
\[ \text{not} \]

Binary-operator expressions are compiled by first compiling the left sub-expression, then compiling the right sub-expression, and finally emitting one more instruction after them; the one instruction after them pops two elements from the stack and replaces them with a new element, which is the result of some computation:

\[ \text{plus} \]
\[ \text{minus} \]
\[ \text{times} \]
\[ \text{divided} \]
\[ \text{remainder} \]
\[ \text{equals} \]
\[ \text{different} \]
\[ \text{less} \]
\[ \text{greater} \]
\[ \text{lessorequal} \]
\[ \text{greaterorequal} \]

1.2.2 Compiling statements

The translation of an empty statement is empty:

\[ \text{skip} \]

The translation of a printing statement consists in first translating the expression, then emitting a print instruction that pops the result and prints it:

\[ \text{print } e \]

The translation of an assignment to a variable \( x \) held in a register \( r_x \) consists in first translating the expression, then popping the result into the register:
\[ r := e \]
\[ = [e] \]
\[ \text{pop } r_x \]

The translation of the sequential composition of two statements is the translation of the first statement followed by the translation of the second statement:
\[ [s_1; s_2] \]
\[ = [s_1] \]
\[ [s_2] \]

The translation of a \textbf{while} loop is as follows:
\[ [\text{while } e \text{ do } s \text{ end;} ] \]
\[ = b \$check \]
\[ $\text{beginning:}$ \]
\[ [s] \]
\[ $\text{check:}$ \]
\[ [e] \]
\[ \text{bnz } $\text{beginning}$ \]

The labels shown here as $\$beginning$ and $\$check$ must be fresh (in the sense of never previously used).

\subsection{1.2.3 Compiling programs}

A program is compiled by compiling each statement inside it, one after the other.