Compile and execute relational expressions, logical expressions, if/else statements, multiple statement bodies, and while loops.

Program

You should write a driver program that takes a text file as a command line argument and then processes it using your compiler.

The compiler should parse the input for a sequence of statements and generate a code tree representing that expression.

The code tree is then used to generate machine code instructions to be executed later.

Logical Values

Your compiler should recognize the keywords "true" and "false".

Hint1: These should be recognized at the lowest level of the expression parser, along with integer constants, string constants, and variables.

Hint2: Since the token type for these keywords is an identifier, the test for these keywords will need to take place before variables are recognized.

You should have a new code tree node type for logical (boolean) constants.

There should be storage in the node for the value (true or false),

just like you have internal storage for integer constants.

The result type of the constant expression should be a boolean type.

Your nodes should be able to handle boolean return values as well as the int4 and string values they currently handle.

The code generation for the print() function should be modified to handle boolean values as well.

A boolean variable type for the language will NOT be implemented at this time.

Implementation

The code generation for these constants should simply place a 1 or 0 into the accumulator for true and false respectively. Hint: Immediate values are not needed.

Relative Expressions

These are the six standard operators for comparing numeric values: <, <=, >, >=, =, ~=. Only comparisons of int4 expressions need be implemented.

The arithmetic operators took two int4 values and returned a int4 value. The relative operators take two int4 values and return a logical value. If the types of the operands are not both int4s, then an error should occur.

With respect to the expression processing hierarchy, relative expressions are higher than arithmetic expressions. So, parsing these operators will involve parsing an arithmetic expression, and then, if appropriate, the relative operator and another arithmetic expression. This means that it's entirely possible that a relative expression may return the value of an arithmetic expression if there are no following relational operators.

The relational operators all have the same precedence level.

Implementation

In a machine with 32-bit registers, there is a purity to representing true as a 32-bit "1", and false as a 32-bit "0". And logical values stored naturally in 32-bit memory locations should likely be represented this way. However, for the comparisons and testing needed in the code, this often means 32-bit immediate values in the code, which is inefficient.

In the Intel architecture, much can be done by dealing only with the lower 8 bits of the register. Immediate values, when needed, can be stored in a single byte.

Rather than push the result of the operator back on to the stack, the convention will be "the result can be found as a 0 or 1 value in the AL register".

The machine code generation for each operator is roughly the same: 1) generate the machine code for the left-hand arithmetic expression. 2) If it has not already been done, this value should be pushed on to the stack. 3) generate the machine code for the right-hand arithmetic expression. The value should be placed in a register. 4) The previously stored left-hand value should be restored from the stack into a different register. 5) The CMP instruction should be issued to compare the two registers. The flags will be set appropriately as a result of this instruction, but the registers themselves will not be changed. 6) The appropriate SETcc instruction should be issued to set the AL register to 0 or 1 depending on the result of the comparison. This final instruction is the only difference between the implementations of all the relational operators.

Logical Expressions

There are three standard operators for comparing logical values: or (|), and (&), and not (!). These operators all take logical values as operands and return a logical value.

This implies that the operators can be placed in sequence, e.g. a | b | c, or a & b & c.

With respect to the expression processing hierarchy, logical expressions are higher than relative expressions.

However, the logical operators have their own internal precedence. Logical not has precedence over logical and, which has precedence over logical or. Short-circuit Evaluation

As discussed in class, short-circuit evaluation of binary logical operators means that instead of evaluating both operands before performing the logical operation,

the first operand is evaluated and then based on the value, it is determined whether or not the second operand should be evaluated.

An important point to make is that when the first operand is evaluated, its value alone may determine the result of the operator (this is the short-circuit).

Equally important is the fact that when the second operand must be evaluated, its value alone will determine the result of the operator.

Consequently, the operator itself will not need to be evaluated. This is more important than it sounds.

Some examples will help. In evaluating expr1 | expr2, if expr1 is true, then the result of the or operator is true, regardless of the value of expr2. But if expr1 is false, then the result of the or operator is true if and only if expr2 is true. The or operator itself does not actually need to be evaluated.

Likewise, in evaluating expr1 & expr2, if expr1 is false, then the result of the and operator is false, regardless of the value of expr2. But if expr1 is true, then the result of the and operator is true if and only if expr2 is true. The and operator itself does not actually need to be evaluated.

Implementation

The need to evaluate sequences of logical operators, along with their precedence adds some additional challenges to the parsing.

However, the structure of this parsing is almost identical to some of the arithmetic parsing that has already been implemented.

Parsing addition sequences leads to calling the parsing for multiplication sequences, which ultimately leads to calling the parsing for arithmetic negation.

In a like fashion, parsing or sequences leads to calling the parsing for and sequences, which leads to calling the parsing for logical negation.

The structure for parsing or sequences is similar to that for parsing addition sequences.

The structure for parsing and sequences is similar to that for parsing multiplication sequences.

The structure for parsing logical negation is similar to parsing arithmetic negation.

The parsing routine for logical negation should call the routine for parsing relative expressions.

Short-circuit Evaluation

Although it seems more advanced, short circuit evaluation for ands and ors turns out to be easier to code.

1)Generate the code for evaluating the left-hand operand. The result of the operand is in the AL register.

2) Generate a TST instruction, comparing the AL register with 1. Because only the AL register is being used, an 8-bit immediate operand for the TST is sufficient.

3) Depending on which operator is being implemented, generate a JE(JZ) or JNE(JNZ) instruction. The jump should be over the code of the second operand, yet to be generated. 4) Generate the code for evaluating the right-hand operand, which should leave its result in the AL register. 5) Finalize the jump offset as needed.

For simplicity, you may assume that the jumps will all be 32-bit jumps.

Logical negations can be implemented using an XOR instruction.

Decisions

The syntax for decisions is the keyword if followed by a logical expression inside of parentheses.

This is followed by the body of the decision.

This can then be followed by an optional else statement, which is also followd by its own decision body.

The code tree node for a decision may have two or three children. The first child is the logical expression. The second child is the body of the decision. The third child is the body of the else clause, if it exists.

Decision Bodies

The bodies of the decisions can come in one of two forms, either as a single statement or as a statement block. Statement blocks are simply sequences of statements enclosed in curly braces. After parsing the expression (including parentheses) of an if statement, the value of the current token (left curly brace or not) will indicate whether a statement or a statement block should be parsed. The same logic works for the body of the else statement.

The parsing of the if statement as described naturally enables the structure of if()...else if()...else if()...else. You will not need to do anything additional to take advantage of this structure.

Implementation

The machine code generation of the decision is slightly different, depending on whether or not there is an else clause.

If there is no else, then 1) Generate the code for the expression. 2)Generate an appropriate TST instruction. 3) Generate a conditional jump. The jump should happen if the result of the expression is false. 4) Generate the code for the decision body. 5) Adjust the jump in 3) to jump over the code of 4).

If the decision has an else, then much of the generation is as above, but with subtle differences. 1) Generate the code for the expression. 2) Generate the appropriate TST instruction. 3) Generate a conditional jump. The jump should happen if the result of the expression is false. 4) Generate the code for the decision body. 5) Generate a nonconditional jump. This is to jump over the body of the elseclause. 6) Adjust the jump in 3) to jump over the code of both 4) and 5). 7) Generate the code for the else clause. 8) Adjust the jump of 5) to jump over the code of 7).

Again, for simplicity, you may assume that all jumps are 32-bit relative jumps.

While loops

After implementing decision statements, while loops are fairly simple. The syntax for decisions is the keyword while followed by a logical expression inside of parentheses. This is followed by the body of the loop. As with decisions, the body may a single statement, or a statement block enclosed in curly braces.

The parsing of the loop body follows the same guidelines as that for the ifstatement.

The code tree node for the while loop has two children. The first is the expression. The second is the body.

Implementation

The machine code generation of the while loop is fairly straightforward. In order to reduce the number of jumps, the loop is turned into a bottom driven loop.

1) Generate a non-conditional jump to jump to the location of the decision. 2) Generate the code of the loop body. 3) Adjust the jump in 1) to jump over the code of 2). 4) Generate the code of the decision expression. 5) Generate an appropriate TST instruction. 6) Generate a conditional jump. The jump should happen if the result of the expression is true. The jump should go backward over the code of 6) 5) 4) and 2) to the start of the loop body in 2).

Code Tree

In addition to the node types you have previously implemented, you should add node types for a logical constant (internal value of true or false), nodes for each of the 6 relational operators, nodes for the three logical operators, a node type for if statements, and a node type for while statements.