MITOCW | MIT6_004S17_11-02-04_300k

A modern compiler starts by analyzing the source program text to produce an equivalent sequence of operations expressed in a language- and machine-independent intermediate representation (IR).

The analysis, or frontend, phase checks that program is well-formed, i.e., that the syntax of each high-level language statement is correct.

It understands the meaning (semantics) of each statement.

Many high-level languages include declarations of the type - e.g., integer, floating point, string, etc. - of each variable, and the frontend verifies that all operations are correctly applied, ensuring that numeric operations have numeric-type operands, string operations have string-type operands, and so on.

Basically the analysis phase converts the text of the source program into an internal data structure that specifies the sequence and type of operations to be performed.

Often there are families of frontend programs that translate a variety of high-level languages (e.g, C, C++, Java) into a common IR.

The synthesis, or backend, phase then optimizes the IR to reduce the number of operations that will be executed when the final code is run.

For example, it might find operations inside of a loop that are independent of the loop index and can moved outside the loop, where they are performed once instead of repeatedly inside the loop.

Once the IR is in its final optimized form, the backend generates code sequences for the target ISA and looks for further optimizations that take advantage of particular features of the ISA.

For example, for the Beta ISA we saw how a CMOVE followed by an arithmetic operation can be shorted to a single operation with a constant operand.

The analysis phase starts by scanning the source text and generating a sequence of token objects that identify the type of each piece of the source text.

While spaces, tabs, newlines, and so on were needed to separate tokens in the source text, they've all been removed during the scanning process.

To enable useful error reporting, token objects also include information about where in the source text each token was found, e.g., the file name, line number, and column number.

The scanning phase reports illegal tokens, e.g., the token "3x" would cause an error since in C it would not be a

legal number or a legal variable name.

The parsing phase processes the sequence of tokens to build the syntax tree, which captures the structure of the original program in a convenient data structure.

The operands have been organized for each unary and binary operation.

The components of each statement have been found and labeled.

The role of each source token has been determined and the information captured in the syntax tree.

Compare the labels of the nodes in the tree to the templates we discussed in the previous segment.

We can see that it would be easy to write a program that did a depth-first tree walk, using the label of each tree node to select the appropriate code generation template.

We won't do that quite yet since there's still some work to be done analyzing and transforming the tree.

The syntax tree makes it easy to verify that the program is semantically correct, e.g., to check that the types of the operands are compatible with the requested operation.

For example, consider the statement x = "bananas".

The syntax of the assignment operation is correct: there's a variable on the left-hand side and an expression on the right-hand side.

But the semantics is not correct, at least in the C language!

By looking in its symbol table to check the declared type for the variable x (int) and comparing it to the type of the expression (string), the semantic checker for the "op =" tree node will detect that the types are not compatible, i.e., that we can't store a string value into an integer variable.

When the semantic analysis is complete, we know that the syntax tree represents a syntactically correct program with valid semantics, and we've finished converting the source program into an equivalent, language-independent sequence of operations.