Compilers

Project 2: Types and Variables

2nd Bachelor Computer Science
2023 – 2024

Kasper Engelen
kasper.engelen@uantwerpen.be

For the project of the Compilers course you will develop, in groups of 3 students, a compiler capable of translating a program written in (a subset of) the C language into the LLVM intermediate language, and into MIPS instructions. You can also work alone, but this will make the assignment obviously more challenging. The project will be completed over the semester with weekly incremental assignments.

Concretely, you will construct a grammar and lexer specification, which is then turned into Python code using the ANTLR tool. The rest of the project will consist of developing custom code written in Python 3. This custom code will take the parse tree and turn it into an abstract syntax tree (AST). This AST will then be used to generate code in the LLVM IR language and in the MIPS assembly language. At various stages of the project you will also have to develop utilities to provide insight into the internals of the compiler, such as visualising the AST tree.

On Blackboard you can find two appendices: one in which an overview of the project is given, and one which contains some information on how to use ANTLR.

1 Types and Variables

The goal of this assignment is to extend your parser to support variables, as well as float and character types.

1.1 Grammar

Extend your grammar to support the following features:

- (mandatory) Add an `int main() { ... }` function. Function calling mechanisms such as arguments and return values do not yet have to be supported, that will come in a later assignment. A missing `main` should result in an error.
- (mandatory) Extra reserved keywords need to be supported: `const`, `char`, `int`, and `float`.
- (mandatory) Literals are now no longer limited to integers: literals of **any type** (integer, floating point, character) can now be part of expressions. **Note:** only "simple" floating point literals in the fixed point format need to be supported. Scientific notation is not necessary.
- (mandatory) Variables.
  There should be support for variables. This includes variable declarations, variable definitions, assignment statements, and identifiers appearing in expressions.
  Note: A variable name (identifier) can contain only letters (both uppercase and lowercase letters), digits and underscore. The first character of an identifier should be either a letter or an underscore. There is no rule on how long an identifier can be.

- (mandatory) Pointers.
  Support for pointers, as well as support for the unary pointer operators `*` (dereference) and `&` (address). Variable declarations and definitions need to support pointers as well.

- (mandatory) Constants.
  Support for constant variables, including const pointers and the `const` keyword. Only const pointers of the following format need to be supported: `const int*`, `const float*`, `const char*`. This means that the pointer *itself* is mutable (i.e. it can be re-assigned). The value that it points to, however, is *constant*.

- (mandatory) Implicit conversions.
  Consider the following order on the basic types:

  `float isRicherThan int isRicherThan char`

  Implicit conversions of a richer to a poorer type (e.g., assignment of an `int` to a `char` variable) should cause a compiler warning indicating possible loss of information.

- (mandatory) Explicit conversions.
  Another extension could be support for explicit casts (i.e., the cast operator). This enables the programmer to indicate that they are aware of possible information loss. Hence the compiler should not yield a warning anymore.

- (mandatory) Pointer arithmetic.
  Pay attention to the size of the types that are pointed to: incrementing a `char*` is different than incrementing an `int*`! Examples:

  - Assignment: `p = 0`,
  - Addition, subtraction, multiplication: `p + q`, `p + 2`, `q - 5`, `p*4`, etc.
  - Increment, decrement: `p++`, `q--`, etc.
  - Comparison: `p < end`, `p >= 0`, `p == 0`, etc.

- (mandatory) Increment/Decrement Operations.
  Support for the unary operators `++` and `--`, both the prefix and suffix variants.

- (optional) Const casting.
  Make it possible to have a non-const pointer to a const value:

```
const float f = 0.789;
const float* f_ptr = &f;
float* non_const_f_ptr = f_ptr;
*non_const_f_ptr = 3.1492;
// f is now equal to 3.1492
```

  Note: if you implement this trick, you will also have to modify your **constant propagation**. Even though `f` is a constant here, it will be modified later on and can thus only be propagated to a limited extent.

Example input file:

```
int main () {
    const int x = 5*(3/10 + 9/10);
    float y = x*2/( 2+1 * 2/3 +x) +8 * (8/4);
    y = x + y;
    int z;
    float* flt_ptr = &y;
    char ch = 'x';
    const int* ptr_to_int = &z;
    ptr_to_int = &x; // this is allowed. The pointer now points to variable x
    *ptr_to_int = 33; // this is NOT allowed!
}
```

1.2 Abstract Syntax Tree

You should construct an **explicit** AST from the Concrete Syntax Tree (CST) generated by ANTLR. Define your own datastructure in Python to construct the AST, such that the AST does not depend on the ANTLR classes.

1.3 Visualization

To show your AST structure, provide a listener or visitor for your AST that prints the tree in the Graphviz dot format. This way it can easily be visualized. For a reference on the dot format, see `http://www.graphviz.org/content/dot-language`. There are useful tools to open `dot` files such as `xdot` on Ubuntu, and Visual Studio Code on MacOS.

1.4 Constant Propagation

When a variable is `const` or used immediatly after being assigned, it can be evaluated at compile time. Hence, most compilers will not actually generate machine code (assembler) for the variable lookup in this situation. Rather, they will replace the variable node in the AST with a literal node containing the result.

Extend your optimization visitor to first replace identifiers in expressions with their value, if it is known at compile-time, before performing constant folding. Note that you should support constant folding if you want to support constant propagation (see project 1).

It can be useful to implement a flag that allows you to turn off the constant folding, so that during testing you can more easily test the compiler.

2 Error Analysis

2.1 Syntax Errors

The compiler is allowed to stop when it encounters a syntax error. An indication of the location of the syntax error should be displayed, but diagnostic information about the type of error is optional (and non-trivial).

2.2 Semantic Errors

For semantical errors, it is necessary to output more specific information about the encountered error. For example, for usage of a variable of the wrong type, you might output:

```
[ Error ] line 54, position 13: variable x has type y while it should be z
```

When in doubt, the GNU C Compiler with options `-ansi` and `-pedantic` is the reference.

Implement a semantic analysis visitor for your AST that checks for semantic errors. These errors include, but are not limited to:

- Use of an undeclared variable.
- Redeclaration or redefinition of an existing variable.
- Operations or assignments of incompatible types.
- Assignment to an rvalue.
- Assignment to a `const` variable.

The semantic analysis visitor will make use of a *symbol table*. Define your own datastructure in Python to construct the symbol table. Keep in mind that the symbol table also needs to consider scopes, even though the current inputfiles only have a global scope.