A language that compiles to C

Patrick Schönberger
05.08.21

Table of contents

Introduction

Language example

```
func main(argc : int, argv : char**) : int {
  var s1 : N1::N2::S1;
  var s2 : N1::N2::S1;
  var s3 : N1::N2::S1;
  s1.i1 = 123;
  s1.i2 = 456;
  s1.m1(s2.m1(s3.m1(89)));

  N1::N2::f1();

  var s4 : S1<int>;
  s4.t1 = 123;
  s4.t2 = 456;
  s4.m1();

  generic1<int>(1, 2);
  generic1<double>(3.4, 5.6);

  return 0;
}
```

Why?

- C still widely used today
- Pros:
  - Naturally fast
  - Available virtually anywhere
  - Many libraries available
- Cons:
  - Can be cumbersome to write
  - No unified build process
  - Not memory safe

What?

- Create a language that can be compiled to readable C code
- Same basic functionality
- Add some features which are easily represented in C:
  - Namespaces
  - Order-independent declarations
  - Struct methods
  - Generics
- These are implemented the way they would usually be when written by hand

How?

- Create a parser using ANTLR4
- Transform the resulting parse tree into an internal representation of the program
- Analyse the IR to resolve namespace access, method calls etc.
- Generate C code from the IR

The language

Basic functionality

- Shares basic concepts with C:
  - 8 basic types (including void)
  - Statically typed
  - Structs (no enums/unions yet)
  - Functions
  - Raw pointers
  - Same basic operators
- Since the resulting code is compiled by a C compiler, they have the same semantics

Namespaces

- Can contain anything that can be declared at the global level in C
  - Variables
  - Structs
  - Functions
- These can have identical names
- Explicitly address namespace using `::`
- Compiled to `namespace_funcName();`

```
namespace N1 {
  func f1() : void {
    puts("Hello\n");
  }
  namespace N2 {
    struct S1 {
      i1 : int;
      i2 : int;
      i3 : int;
    }

    func f1() : void {
      var s1 : N1::N2::S1;
      s1.m1(123);
    }
  }
}
```

Order-independent declarations

- Collect all declared structs and functions
- Simply forward declare all of them

```
func f1(): void {
  f2();
}

func f2(): void {
  printf(“Hello\n”);
}
```

Struct methods

- Function declared inside structs get passed an implicit this
- Method calls are transformed into the appropriate function calls based off of the variable type
- Method calls are always translated, basically Uniform Function Call Syntax
- Allows calling methods on non-struct types

```
struct S1 {
  abc(): S1 { }
  xyz(): S1 { }
}

var s: S1;
s.abc.xyz();
```

Generics

- Structs and functions can be declared generic
- One version will be generated for each type instantiation
- Has to be considered when selecting a function for a method call

```
struct S1<T> {
  t1: T;
  t2: T;

  m1() : T {
    return this->t1 + this->t2;
  }
}

func generic1<A>(a1 : A, a2 : A) : A {
  return a1 + a2;
}

var s: S1<int>;
s.m1(); // returns int

generic1<double>(1.5, 2.5);
```

How the compiler works

Parser

- Automatically generated using ANTLR4
- Produces C++ classes for lexing and parsing
- One type for each rule
  - `FuncDeclContext`
  - `FuncContext`
  - `ParameterContext`
- Can be traversed manually or by writing a listener/visitor class

```
funcDecl: 'func' func;

func: funcName genericDecl?
      '(' parameter ')'
      (':' type)
      (body | ';');

parameter: (var (',' var)*)?;
```

Internal representation

- 1 to 1 mapping of language semantics
- Parse tree is manually traversed
- Single Source of Truth approach, everything defined only once, anything else is a pointer

```
struct Function
{
  std::string name;
  Type returnType;
  std::vector<Variable> parameters;
  bool defined;

  std::vector<std::string> genericTypeNames;
  std::vector<std::vector<Type>>
genericInstantiations;

  Body body;
};
```

Translation to C

- Recursively print C equivalent
- Only print what is specific to a concept
  - A “body” is printed separately from a function because it also applies to e.g. an if statement
- Overloaded << operator
- Generic instantiations are collected by visiting functions calls/type declarations
- All additional features can be generated ad-hoc by collecting information from the IR
- Generics are generated using a global map

```
if (m->type == TypeModifierType::Pointer)
{
  sstr << "*(" << s << ")";
}
else
{
  sstr << "(" << s << ")[";
  if (m->_staticArray)
    sstr << m->_arraySize;
  sstr << "]";
}
```

Evaluation

What works

- Basic functionality fully working
- Parser and IR modular enough to easily add missing features
- Single Source of Truth allows transforming IR
- Figuring out the type of expressions is robust
- Code generation for advanced features works well for most intended use cases

What’s missing

- User often has to be specific
  - No type deduction
  - No hierarchical selection of namespace members
- No generic methods and no nested type instantiation (`S1<S2<int> >`)
- No multiple input/output files
- No inclusion of C libraries
- Operator precedence not parsed (although present thanks to C compiler)
- Selection of appropriate method calls can be wrong for chained calls (due to namespace problem)

Conclusion

Lessons learned

- More generic structure for AST preferable
  - Would alleviate many problems
- Think about what is necessary for the desired features
  - Collection of function calls/type declarations
  - Addressing specific nodes in a given subtree
- Visitor pattern makes a lot of sense for ASTs

Outlook

- Add missing base functionality (enums/unions, const/static decl, bitfields…)
- Rewrite IR as more traditional AST
- Complete generics (methods, nested types)

Later:

- Additional compilation targets
- Add interpreter
- Custom parser(?)