UCC: Ultimate Civic Compiler

Jos Bonsink (10172920) & Mustafa Karaalioglu (10217665)

April 10, 2014

Contents

1 Introduction

In order to learn as much as possible from all aspects of compiler construction, the decision was made to build everything from scratch. C++ was the language of choice for the reasons that memory management would be easier, templated code could be used to reduce boiler plate code and if required, the original framework could be mimicked.

2 Lexicographic Analysis

Instead of using a generated scanner, whether table-driven or direct-coded, we decided to build a hand-coded scanner for the compiler. The choice to implement a hand-coded scanner proved to be an intelligent one. Understanding the usage of a third party tool would have taken more time than to implement our own scanner. The resulting scanner is easy to manage. Additional reserved words and symbols can easily be added to existing data structures. The scanner does not have any dependencies, since it does not rely on tools like lex.

The scanner does not simulate a deterministic finite automaton, but instead uses a more direct approach by pattern matching. While character-by-character scanning using a DFA comes from clean algorithmic formulations, it is far less natural to think and program in such way. An example of pattern matching a word is as follows:

```
char* wordPattern = "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_";
...
if(first >= 'A' && first <= 'Z') TokenizeWord(...);
...
void TokenizeWord(...)
{
    end = line.find_first_not_of(wordPattern, start);
    token.readString = line.substr(start, end - start);
}
```

This is basically all that is required to tokenize words and it looks and feels very similar to what we as human beings do when we extract words out of text. This direct translation into code makes it easy to write and comprehend. Tokenizing numbers and symbols looks very similar.

3 Syntactic Analysis

Compiler compilers that generate parsers based on an analytic grammar are available. Yacc and Bison generate table driven parsers, where the parser logic is contained in the parser program's data and not so much in the code. It is comparable with a black box. To really understand the complexities that arise when designing a parser, we decided to write a parser by hand.

There are two parser types, the top-down parser and the bottom-up parser. Bottom-up parsers support a somewhat larger range of language grammars than the deterministic top-down parsers. From a programming standpoint, the top-down parser is easier to implement. The extra language support of the bottom-up parser was not required, so the choice fell on a top-down parser.

The efficiency of a top-down parser depends critically on its ability to pick the correct production each time that it expands a nonterminal. The cost of parsing rises when wrong choices are made. For some grammars, the worst case behaviour is that the parser does not terminate. To avoid these kinds of problems certain transformations have been made to the original Civic grammar.

The original grammar used left recursion. With left-recursion, the top-down parser can loop indefinitely. To avoid this problem the left-recursive grammar has been reformulated so that it uses right recursion. Not all grammars are backtrack free, including the original grammar. Backtracking has been eliminated by left factoring the rules.

Expression parsing is a complex matter. The original grammar treats all of the arithmetic operators in the same way, without any regard for precedence. Precedence levels can be encoded into the grammar. But this greatly expands the amount of rules and thus the clarity of the parser. We decided to use the Shunting-yard algorithm [1] to parse mathematical expressions, thus limiting the grammar size.

The implementation of Shunting-yard was more complex than expected. Extending the original algorithm with support for function calls proved problematic. An alternative to Shunting-yard is the precedence climbing method [3]. The idea is that you "climb down" the precedence levels and create a tree along the way. Unlike the Shunting-yard algorithm this algorithm is not a pure operator-precedence parser, therefore implementing function call support was a trivial matter and thus we revised our earlier decision to use Shunting-yard.

For the most part the grammar could be directly translated into code, which made it easy to write as shown below.

```
bool FunHeader()
{
    return ParenthesesL() && Param() && Params() && ParenthesesR();
}
```

However, this elegance was mostly lost when incorporating error logging and abstract syntax tree generation. Parsing errors are handled by throwing exceptions, which allows the error logging to be handled at a higher level. The once so elegant code turned into something far less comprehendable as exemplified below.

```
bool Parser::LocalFuns()
{
    if(Void() && Id(true) && LocalFun(true) && LocalFuns())
    {
        return true;
    }
    else if(Type())
    {
        if(ArrayExpr()) throw ParseException("Unexpected array
            expression(variable declarations should precede function
            definitions)"]);
        else if(Id(true))
        {
            if(LocalFun()) return LocalFuns();
            else if(AssignOpt() || Semicolon()) throw
                ParseException("Variable declaration should precede
                function definitions");
            else(LocalFun(true));
        }
    }

    return true;
}
```

Finding a way to reduce this complexity proved to be too challenging

without prior experience of writing parsers.

4 Context-Sensitive Analysis

There are two techniques for gathering context-sensitive knowledge: an automated approach based on attribute grammars [2] and an ad hoc approach that relies on similar concepts. It was briefly mentioned in the previous section that abstract syntax tree generation was incorporated in the parser. This is the result of implementing the ad hoc syntax-directed translation.

The ad hoc approach is more commonly used in compilers and is more flexible than attribute grammars. The ad hoc approach does not restrict the programmer on how to gather context sensitive information. In contrast to the attribute-grammar where the information is saved in attributes assigned to production rules.

The attribute approach has some problems. Not all languages can be cleanly mapped onto onto the attribute-grammar paradigm, particularly those languages with a complex pattern of information flow can be difficult to express as attribute grammars.

4.1 Intermediate Representation

In order to perform any context-sensitive analysis or assembly code generation, an abstract syntax tree is used. The tree is generated during parsing, where tokens are pushed onto a stack until an AST node can be made. Nodes have a shared base class that contains a list of children, allowing for a tree like structure. Static polymorphism is used to allow for a clean and relatively type safe way of dealing with nodes. Another great benefit of using static polymorphism is the automatic registration of nodes. The example below shows how each new node class automatically obtains a unique family identifier.

```
class BaseNode
{
public:
    uint32_t Family() const;

protected:
    uint32_t family_ = ~0;
```

```
    static uint32_t familyCounter_;
};

template<class T>
struct Node : public BaseNode
{
    Node(){ family_ = Family(); }

    static int32_t Family()
    {
        static uint32_t family = (1 << BaseNode::familyCounter_++);
        return family;
    }
};

struct UnaryOp : public Node<UnaryOp> {};
struct BinaryOp : public Node<BinaryOp> {};
```

Because the UnaryOp and BinaryOp classes inherit from a templated class, the C++ compiler will create two classes of the templated Node class, one for UnaryOp and one for BinaryOp. As the Family() method is static, all instantations of a node will share the same method that is constructed at compile time. Because UnaryOp and BinaryOp each get their own version of the templated Node class, they now have a static Family() method that is unique to them. If we now write the following code:

```
UnaryOp u1, u2;
BinaryOp b1, b2;
```

Both u1 and u2 will be of the same family with number 1, and b1 and b2 will be of the same family with number 2.

With some additions to our base nodes we can do the following:

```
BaseNode* node = new BinaryOp();
if(node->Family() == BinaryOp::Family()) print("This evaluates to
    true!");
```

This is a very powerful tool that we can leverage to traverse nodes of a specific type and even to get some form of type safety when casting nodes from a

base type to a parent type. Consider the following:

```
template<class T>
T* StaticCast(BaseNode *node)
{
    return node->Family() == T::Family() ? static_cast<T>(node) :
        nullptr;
}
```

We can now only cast a node to a non-null pointer if it's of the same family, guaranteeing us that we'll never cast to the wrong type and modify unintended bytes.

This technique is heavily used to create tree traversals for different node types with type safety and without code duplication.

4.2 Symbol table

To perform context analysis the compiler has to derive information about the various entities manipulated by the program being translated. It must discover and store many distinct kinds of information. These facts have been recorded directly into the AST. For example the AST contains information about function parameter types and the return type of functions. The advantage of this approach is that it uses a single representation for the code being compiled. The disadvantage of this approach is that a lot of tree traversals have to be used to find some specific information. To eliminate this inefficiency, we have added pointers to identifier nodes to link back to the corresponding declaration. We have achieved this by using a symbol table to match an identifier with the correct declaration.

Since Civic allows a program to declare names at multiple levels, one symbol table is not enough. We need a sheaf of tables. A new scope is initialized by the following code:

```
void SymbolTable::Sheaf::InitializeScope()
{
    ++level;

    if (tables.size() == 0 || tables.find(level) == tables.end())
        tables[level] = Table();
```

```
    tables[level].Clear();
}
```

The symbol table is built during a single pass of the AST. The first items that are inserted in the level zero symbol table are the global declarations and definitions. As these should be accessible throughout the entire file. A new scope is created when a function definition is encountered. The functions arguments are the first items to be added to the table. The function body contains variable declarations and possibly more function definitions. Before traversing into the bodies of the inner functions, every function definition has to be added to the current table. Since inner functions should be able to call each other, they have to be able to find each others records in the symbol table. Finally the variable decelerations are added before traversing into inner functions and thus creating new symbol tables. A scope can be finalized by the following code:

```
void SymbolTable::Sheaf::FinalizeScope()
{
    if (level > 0) --level;
}
```

4.3 Analysis

The context analysis is performed along side the creation of the symbol table. Checks are implemented to prevent all illegal statements. We chose to do almost al the checks during one pass to make use of the symbol table. The symbol contains records for every function and variable that can be accessed in the current scope. Each record contains information about the item in question. A records looks like this:

```
class Record
    {
    public:
        Record() { }
        Record(bool immutable, Nodes::Type node, Nodes::NodePtr
            ptr) : immutable(immutable), initialized(false),
            type(node), decPtr(ptr) { }
```

```
        bool immutable;
        bool initialized;
        Nodes::Type type;
        Nodes::NodePtr decPtr;
        std::vector<Nodes::Param> params;

        std::vector<int> arrayDimensions;
        std::vector<std::string> dim;
    };
```

When an identifier cannot be found in the table of the current level or tables of higher levels, we can conclude that it does not exist. When a record of an identifier can be found, the analyser has access to a lot of information like the type of the identifier. Thus enabling type checking. The analyser does not only check for errors but also prepares the AST for the assembler. Additional information is added to the AST like operator types.

5 Code generation

Before the code generation process is started, tables are made for all the different nodes to map either their frame or variable index. This way the required information can easily be retrieved during node traversal. After the table generation has been completed, the AST is traversed breadth-first for all function definitions and statements. Expressions are traversed depth-first.

6 Conclusion

Writing a compiler from scratch is a great learning experience, but something that should only be done once in a lifetime. Writing a correct parser and generating an AST is not only a lot of work, but is also hard to engineer well. The templated node system for the AST made it a lot easier to write traversals to modify, replace or add nodes. With built-in type-safety and memory management the amount of programming errors was most likely reduced significantly. Unfortunately the compiler is not perfect, as it is lacking support for arrays and in some occasions generates the wrong assembly.

References

[1] Dr. E.W. Dijkstra. An algol 60 translator for the x1, 1961. [Online; accessed 2-April-2014].

[2] Jukka Paakki. Attribute grammar paradigmsa high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995.

[3] Martin Richards and Colin Whitby-Strevens. *BCPL: the language and its compiler*. Cambridge University Press, 1981.