# ssCC

a Parser Code Generator

~~ssCC~~

# SiCC

Simple Compiler Compiler

# The Goal of a Compiler Compiler

To create a compiler for a specific language based on the language's token and grammar definition.

Takes care of the dirty work of having to analyze input.

No need to implement tedious tokenizing and parsing of input every time you need to create a language, simply define what the desired language "looks like".

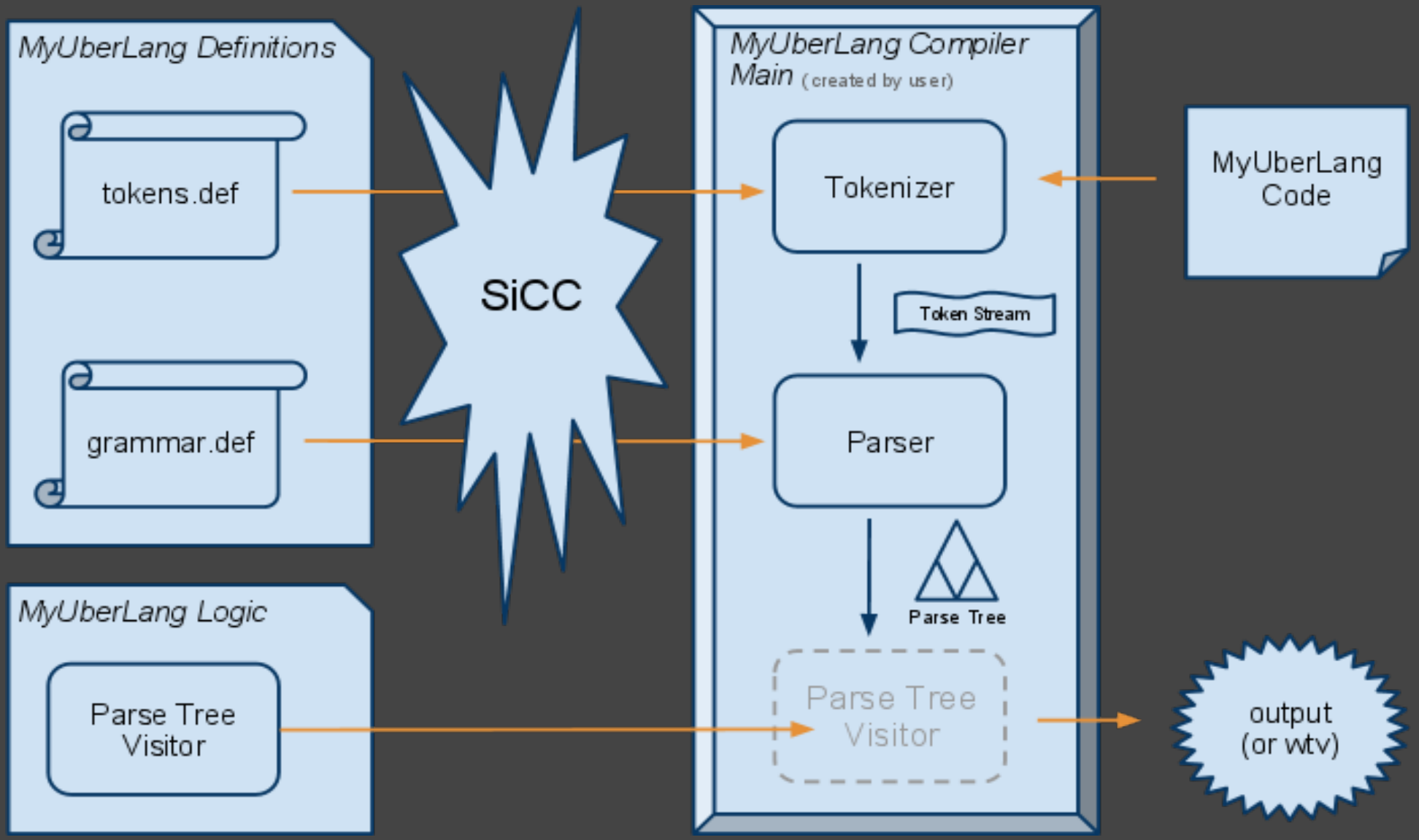
# How SiCC Works

Let's create a new language: MyUberLang

- The user creates token and grammar definitions for MyUberLang and feeds them to SiCC
- SiCC uses the definitions to build a tokenizer and a parser, and outputs them as Java classes
- Along with SiCC's output, the user provides extra Java classes which traverses the parse tree in order to implement MyUberLang's "logic"
- The total of generated and user-supplied classes can now be used to compile or interpret code written MyUberLang

# How SiCC Works

*now with diagrams!*



# Strictly Speaking...

*(you might have noticed)*

...SiCC is not a "compiler compiler" but rather a "tokenizer and parser code generator".

The code that SiCC generates will tokenize and build a parse tree, but it does not know what to do next, it cannot compile.

It's up to you to create classes that use the generated parse tree to meet your needs, such as interpreting or compiling.

# Overview

*checklist...*

- **SiCC** - Takes care of dirty work, creates tokenizer and parser
- **Token Definition File (txt)** - Food for SiCC, makes tokenizer
- **Grammar Definition File (txt)** - Food for SiCC, makes parser
- **Visitor implementation (Java)** - Traverse parse tree, gives meaning
- **Main class (Java)** - Connect everything together

# The SiCC Command

SiCC [options] definitions

*options* *(combination of the following)*

--package *pakagename*
includes all generated files in the given Java package

--prefix *prefix*
adds the given prefix to all generated classes

*definitions* *(one of the following)*

*token.def grammar.def*

--tokenizer-only *tokens.def*

--parser-only *grammar.def*

# The SiCC Command

*examples...*

The basic command

```
sicc  myuberlang.tokens.txt  myuberlang.grammar.txt
```

Include all generated classes in the Java package "uberpack"

```
sicc  --package uberpack  myuberlang.tokens.txt  myuberlang.grammar.txt
```

Prefix all generated classes with Uber (such as UberTokenizer.java)

```
sicc  --prefix uber  myuberlang.tokens.txt  myuberlang.grammar.txt
```

Generate only a tokenizer, which will belong to the "ubertok" package

```
sicc  --package ubertok  --tokenizer-only  myuberlang.tokens.txt
```

# Token Definition File

A simple text file containing regular definitions.

One definition per line in the format `tokenname: definition`

Token names must be of only alphanumeric characters ('a' to 'b' and '0' to '9') and must start with a letter.

Definitions are written as regular expressions.

Internal token definitions are written `:tokenname: definition`
and will not be turned into tokens by the generated tokenizer, but can be embedded in other token definitions.

Embedding a token is only allowed if the token to be embedded has been defined above the current definition.

Comments may be written by starting with a pound sign: #

# Token Definition File

*operators and special characters...*

*** Match zero or more**

**+ Match one or more**

**? Match one or none**

**| Match the pattern on either side, much like an OR**

**\ Escape, matches the next character (used to match operators, such as \+ or \[ or \:)**

**\n Matches a newline characters**

**\r Matches a carriage return character**

**\t Matches a tab character**

**\s Matches a space character**

**() Group patterns**

**[abc] Character class, matches any character within the brackets**

**[^abc] Negative character class, matches any character that is not within the brackets**

**:tok: Uses the pattern of the named token to match**

# Token Definition File

*examples...*

```
# matching "khan", "khaan", "khaaan", "khaaaan"...
tok1: kha+n

# matching "fun" or "sun"
tok2: [fs]un

# matching "wild cats" or "wild dogs", note the use of \s
tok3: wild \s (cats | dogs)

# matches a quoted string
tok5: " [^"]* "

# matching a variable name
# (alphanumeric, starting with a letter)
:alpha: [ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]
:digit: [0123456789]
tok4: :alpha: (:alpha: | :digit:)*
```

# Token Definition File

*special SiCC token definitions...*

*SKIP*

definition of what the generated tokenizer may ignore, usually used for ignoring whitespace and comments

*EOF*

not defined in the token definition file, but is automatically returned by the generated tokenizer when the end of the input has been reached, which is used by the parser

# Grammar Definition File

A simple text file containing a context-free grammar.

One definition per line in the format `Rule -> definition`

Rule names must be of only alphanumeric characters ('a' to 'b' and '0' to '9') and must start with a letter.

Definitions are written as (simpler) regular expressions.

Only token and rule names may be used in the definition.

The first rule is considered the starting rule and becomes the root of the parse tree.

Comments may be written by starting with a pound sign: #

# Grammar Definition File

*operators and special tokens...*

*note*: A much smaller set of operations compared to token defintions.

*** Match zero or more**

**? Match one or none**

**| Match the pattern on either side, much like an OR**

**() Group patterns**

**\0 Epsilon, an 'empty' match**

**[>1] Multiple child flag, must be places at the end of the definition, signals that the rule should be included in the parse tree only if the node has more than one child**

# Grammar Definition File

*examples...*

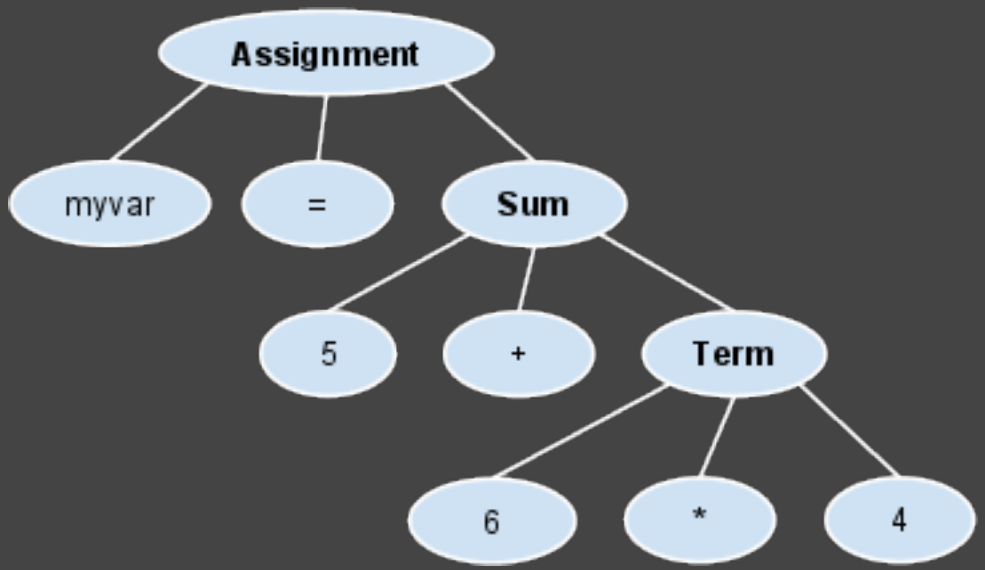
```
# matching a phone number of the form "(613) 555-1234"

PhoneNumber -> AreaCode space FirstPart dash

SecondPart AreaCode -> leftparen threedigits rightparen

FirstPart -> threedigits

SecondPart -> fourdigits

# matching a person's name, optional title and middle names

FullName -> Title FirstName MiddleNames LastName

Title -> profession | maritalstatus | \0

FirstName -> name

MiddleNames -> name*

LastName -> name
```

# Grammar Definition File

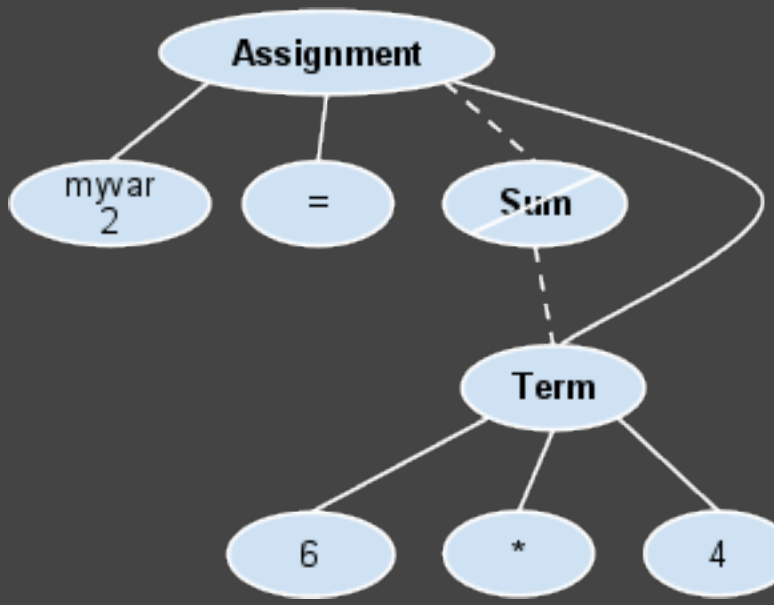
*examples (continued)...*

```
# using the [>1] indicator

Assignment -> var eq Sum

Sum -> Term (plus Term)* [>1]

Term -> number (multiply number)* [>1]
```

**myvar = 5 + 6 * 4**




**myvar2 = 6 * 4**



# SiCC Generated Classes

| | |
|---|---|
| ASTNode | Base parse tree node class |
| ASTToken | A superclass of ASTNode, represents a token in the parse tree |
| AST___Node | A superclass of ASTNode, one created for each grammar rule |
| iTokenizer | An interface implemented by Tokenizer |
| Parser | The main parsing class, takes a Tokenizer and outputs a parse tree |
| Token | A token outputed from Tokenizer |
| Tokenizer | The main tokenizing class, reads in a character stream and outputs a stream of Tokens |
| Visitor<X,Y> | An interface that uses the visitor pattern, used to traverse the parse tree |

# Traversing the Parse Tree

*parse tree nodes...*

A class named AST___Node is created for every rule defined.

ex: ASTBlockNode, ASTStatementNode, ASTSumNode

An ASTToken class is also created to represent tokens.

All of these classes are superclasses of the base ASTNode and make up the generated parse tree.

# Traversing the Parse Tree

*selected variables and methods of ASTNode...*

```
public class ASTNode {

  private ASTNode parent;

  private Vector<ASTNode> children = new Vector<ASTNode>();

  private String name, value;


  public Vector<ASTNode> getChildren() { return children; }

  public ASTNode getChild(int i) { return children.get(i); }

  public int numChildren() { return children.size(); }

  public String getName() { return name; } // rule or token name

  public String getValue() { return value; } // only used for ASTTokens

  public ASTNode getParent() { return parent; }

  public <X,Y> X accept(Visitor<X,Y> visitor, Y data) {
    return visitor.visit(this, data);
  }
}
```

# Traversing the Parse Tree

*the Visitor interface...*

A generic Java interface called Visitor is also created:

```
public interface Visitor<X,Y>
```

The interface defines the following method for ASTNode and each of its AST___Node superclasses:

```
public X visit(AST___Node node, Y data);
```

In your implementation of the Visitor interface, the class types and uses of X and Y are of your choosing, they are meant as helpers.

# Traversing the Parse Tree

*visiting the tree...*

As you might have noticed, ASTNode defines an `accept(Visitor v)` method, which calls the Visitor's `visit(ASTNode n)` function with itself as the argument.

The parse tree is visited in this way.

In your implementation of Visitor, each call to `visit(AST___Node n)` will usually include recursive `accept(this)` to each of the node's children, along with the "logic" needed to handle the node.

# Putting It All Together

*creating a Main class...*

## Basically:

```
class SuperApp {

    public static void main(String args[]) {

        // Create a tokenizer from where the input is coming
        Tokenizer tokenizer = new Tokenizer(new InputStreamReader(System.in));

        // The parser needs a tokenizer, so pass it in
        Parser parser = new Parser(tokenizer);

        // Simply call the parser's parse() method, which returns the root node
        ASTEquationNode rootnode = parser.parse();

        // Create a visitor
        InterpretorVisitor() interpretor = new InterpretorVisitor();

        // Start the traversal by visiting the root node, the output type and
meaning          // depends on your Visitor implementation
        String output = interpretor.visit(rootnode, null);

    }

}
```

# That's All There Is To It!


.....yeah ok, it's best to learn by examples