Compiler Compiler Tutorial

CSA2010 – Compiler Techniques

Gordon Mangion

Topics

- Prerequisites
- Compiler modules
- Javacc
- Semantic Analysis
- Code generation
- Examples
- The assignment (VSL)

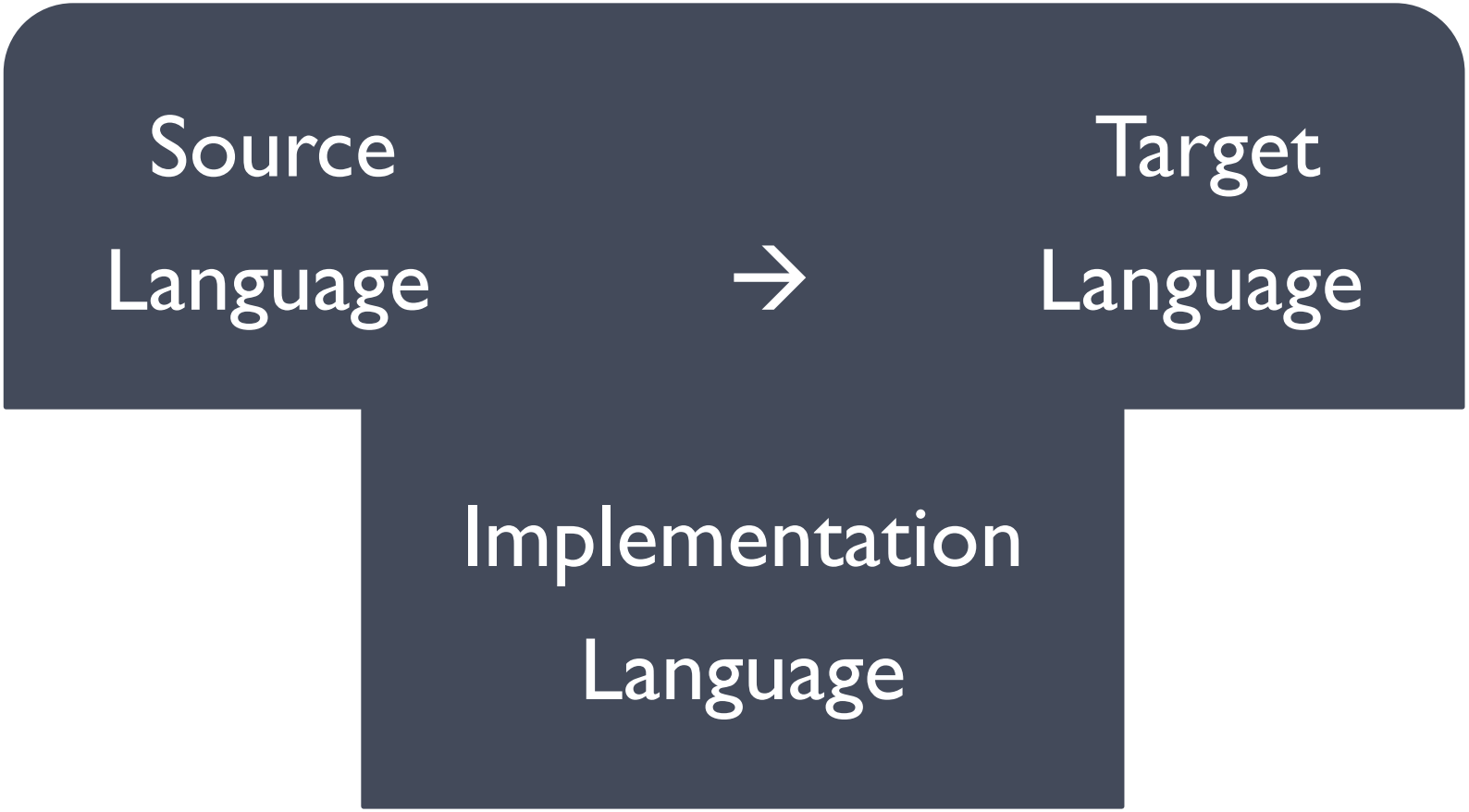
Prerequisites

- Java
- Regular Expressions
  - ? + * …
- Production rules and EBNF
- Semantics

Compilers

- What is a compiler?
- Architecture of a Compiler
- Where to start from?

Tombstone Diagram



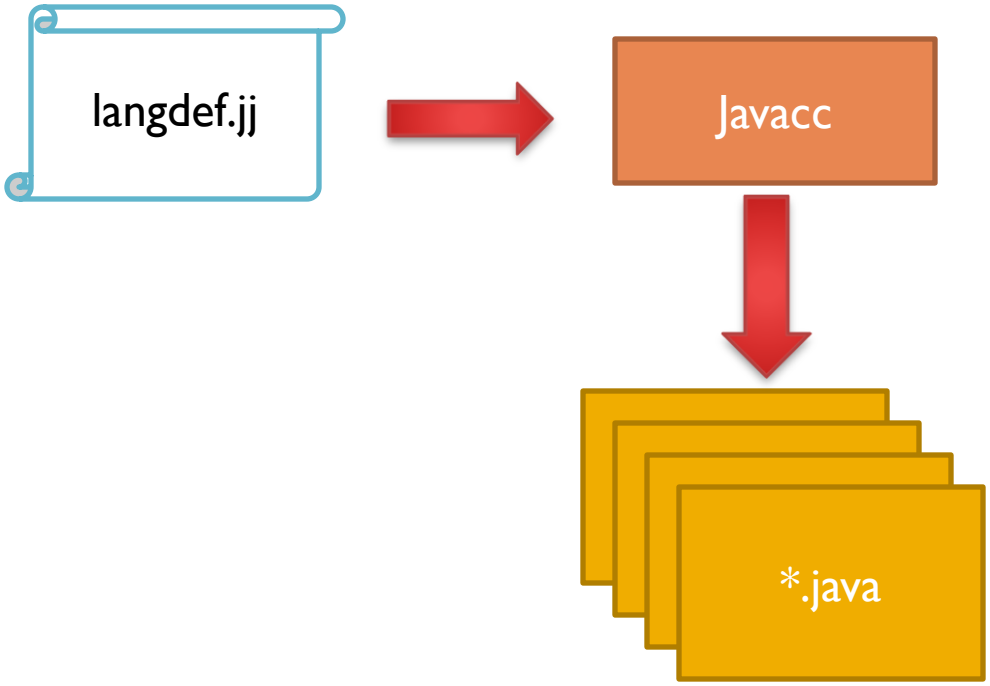
Program Generators?

- Source-to-Executable
- Reverse Polish Notation
- Source-to-Source
  - Preprocessors

Javacc

- A compiler that creates Parsers / Compilers
- The source code is a definition file
  - Grammar definition
  - Inline Java code
- The target is java-based parser
- Recognizer?

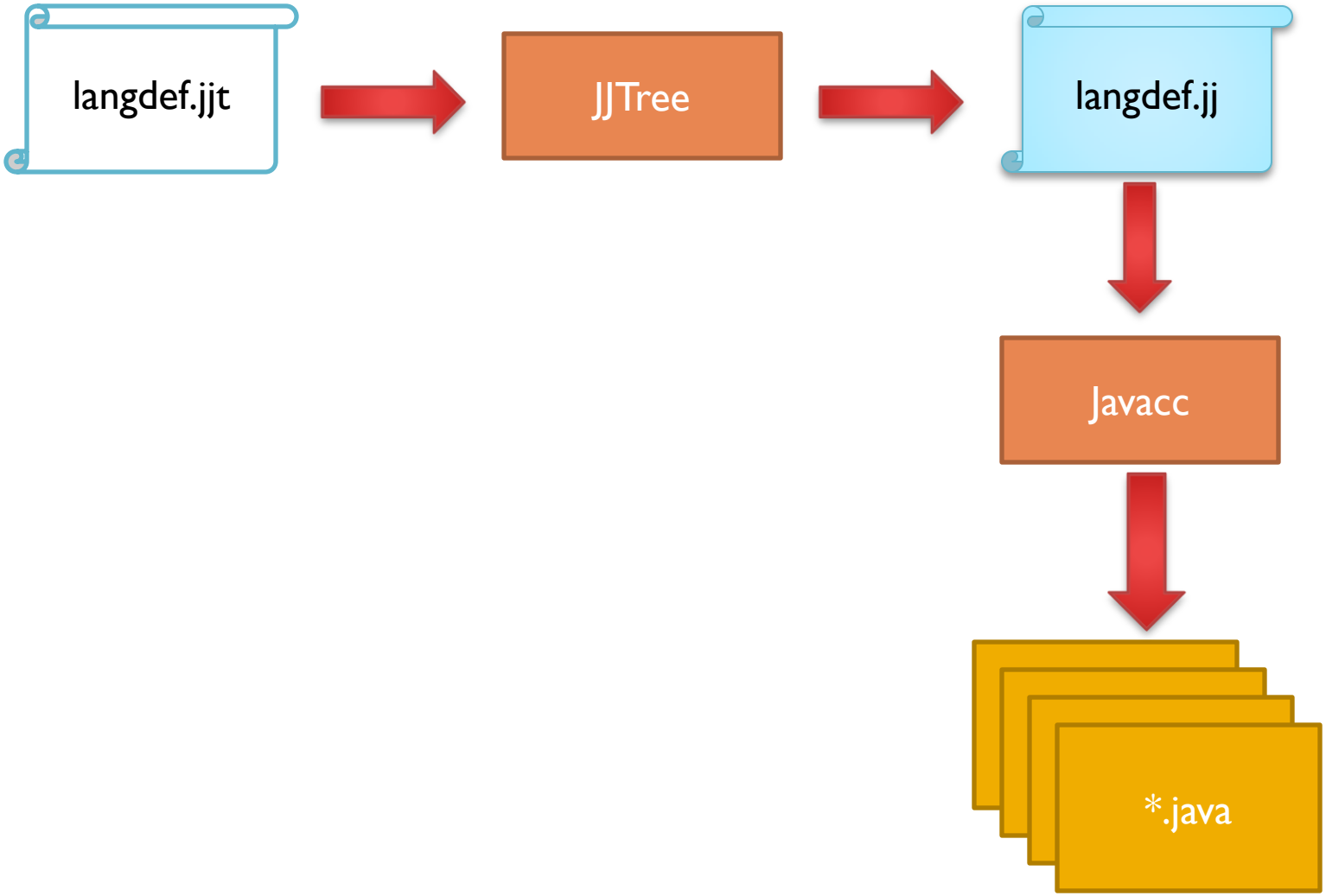
Javacc



JJTree

- From a recognizer to a parser
- Preprocessor to Javacc
- Produces Parse Tree building actions
- Creates ".jj" files from ".jjt" files

JJTree



Installing Javacc

- https://javacc.dev.java.net/
  - Javacc.zip / Javacc.tar.gz
- Eclipse plugin
  - Preferences->Install/Update->Add
    - http://eclipse-javacc.sourceforge.net/
  - Help->New Software

.jjt Grammar Files

- Four (4) sections
  - Options
  - Parser
  - Tokens
  - Production Rules

.jjt Options

```
options
{
   BUILD_NODE_FILES=false;
   STATIC=false;
   MULTI=true;
   …
}
```

.jjt Parser block

PARSER_BEGIN(**parser_name**)

…

public class **parser_name**

{

…

}

…

PARSER_END(**parser_name**)

.jjt Tokens

- Lexeme to ignore

```
SKIP :
{
  " "
  | "\t"
  | "\n"
  | "\r"
  | <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

.jjt Tokens

- Tokens that are to be returned

```
TOKEN :
{
  <PLUS:“+”>
  |<TRUE: "true" >
  | <FALSE: "false" >
  | <LETTER: (["A"-"Z"] | ["a"-"z"]) >
  | <STRING_LITERAL: "\"" (~["\"","\r","\n"])* "\"" >
  | <#DIGIT: [“0”-”9”]>
}
```

.jjt Production Rules

- Assume the following:

Header → "**Script**" <STRING_LITERAL>

or

Header ::= "**Script**" <STRING_LITERAL>

.jjt Production Rules

- Assume the following:

Header ::= “Script” <STRING_LITERAL>

```
void Header():
{
  <SCRIPT> <STRING_LITERAL>
}
```

.jjt Production Rules

Header ::= “Script” <STRING_LITERAL>

```
void Header(): { int n = 0; n++;}
{
  <SCRIPT> <STRING_LITERAL>
  { n++; }
}
```

.jjt Production Rules

- Calling Non-Terminals

```
void Integer(): {}
{ … }

void Header(): { }
{
   <SCRIPT>
   Integer()
}
```

.jjt Production Rules

- Token matching - Actions

```
void Header(): { int n = 0; n++; }
{
  <SCRIPT>{ n = 1; }
  <STRING_LITERAL>
  { n++; }
}
```

.jjt Production Rules

- Getting Token value

```
void Number()
{ Token t;
  int n = 0; }
{
  t=<DIGIT>{ n = integer.parseInt(t.image);
  }
}
```

Lookahead

- Consider the following java statements
  - public int name;
  - public int name[];
  - public int name() { … }

Building the AST

```
options
{
   STATIC=false;
   MULTI=true;
   BUILD_NODE_FILES=true;
   NODE_USES_PARSER=false;
   NODE_PREFIX="";
   …
}
```

Building the AST

```
SimpleNode Start() : {}
{
  Expression()
  ";"
  <EOF>
  {
      return jjtThis;
  }
}
```

Parsing

```
PARSER_BEGIN(MyParser)
...
MyParser parser = new MyParser(System.in);
try {
   SimpleNode n = parser.Start();
   System.out.println(“Parsed!");
} catch (Exception e) {
      System.out.println("Oops!");
      System.out.println(e.getMessage());
}
...
PARSER_END(MyParser)
```

Worksheet

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Plus ::= “+”

Minus ::= “**-**”

Op ::= Plus | Minus

Expression ::= Number { Op Number }

Example

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Factor ::= Expression | Number

Term ::= Factor [“*” | “/” Factor]

Expression ::= Term { “**+**” | “**-**” Term }

Start ::= Expression