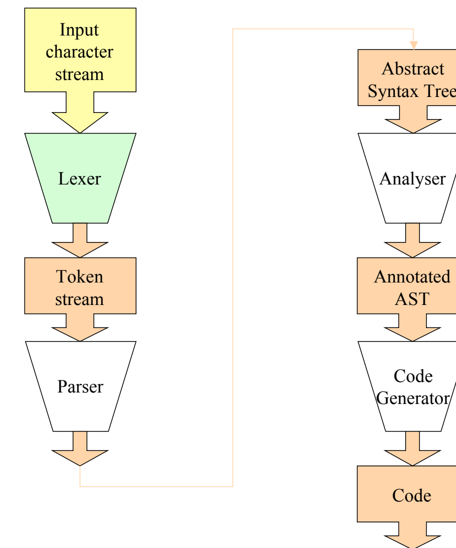
Automated Tools

Easier ways to create parsers

The Compilation Task



Automated?

- The initial stages of compilation are very mechanistic
  - Lexer
    - Text to stream of tokens
      - Pattern matching
  - Parser
    - Application of grammar rules
    - Recovery from errors
  - AST Generation
    - Data structure for subsequent stages

Automated?

- The final stages of compilation are language dependant
  - Semantic Analysis
    - Unfortunately often ad-hoc and hard to codify
  - Code Generation
    - Transformation of AST
    - Dependant on the target machine/language

Lexer Generators

- Transforms a set of regular expressions into a finite state automaton.
- Attempts to find match rules that consume the maximum number of characters
- Jlex
  - Produces Java
- Lex/Flex
  - Produces C

Parser Generators

- YACC and Bison
  - Produces C
  - Bottom up
    - make choices after consuming all the tokens associated with the choice
    - Use BNF grammars rather than EBNF grammars

Combined Approaches

- ANTLR ANTLR
  (ANother Tool for Language Recognition)
  - Supports three stages
    - Lexer, Parser and Tree walker
  - Produces C/C++, Java or Sather
  - LL(*k*)
- JavaCC
  - Supports two stages
    - Three with JJTree
    - Lexer and Parser
  - LL(*k*)

What is JavaCC

- JavaCC stands for "the Java Compiler Compiler“
  - a parser generator and lexical analyzer generator.
  - hand-crafting a lexer and a parser is difficult
- www.experimentalstuff.com/Technologies/JavaCC/

JavaCC Overview

- Generates a top down parser.
  - Hence ideal for generating a parser which is LL(*k*).
- Generates a parser in Java.
  - Hence can be integrated with any Java based application.
- Token specification and grammar specification structures are in the same file.

Types of Productions in JavaCC

- There can be four different kinds of Productions.

1. Regular Expressions
   - Used to describe the tokens (terminals) of the grammar.
2. Token Manager Declarations
   - The declarations and statements are written into the generated Token Manager (lexer) and are accessible from within lexical actions.

Types of Productions in JavaCC

- There can be four different kinds of Productions.

3. EBNF
   - Standard way of specifying the productions of the grammar.
4. Java Code
   - For something that is not context free or is difficult to write a grammar for.

An Example Grammar

```
start    ::=    expr ";"
expr     ::=    (addOp)? term (addOp term)*
addOp    ::=    "+" | "-"
term     ::=    factor (multop factor)*
multop   ::=    "*" | "/"
factor   ::=    primary ( "**" primary)?
primary  ::=    number |
                "(" expr ")" |
                ident ( "(" rvalue ( "," rvalue )* ")" )?
rvalue   ::=    expr | string
number   ::=    <INTEGER_LITERAL> |
                <REAL_LITERAL>    |
                <HEX_LITERAL>
ident    ::=    <IDENTIFIER>
string   ::=    <STRING>
```

Terminals

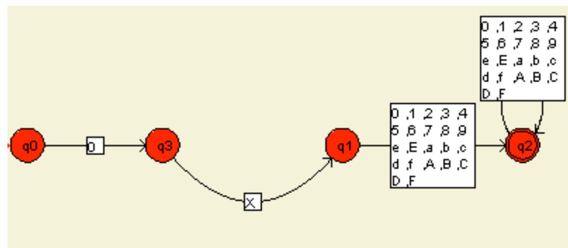
- `REAL_LITERAL: (["0"-"9"])+ "." (["0"-"9"])+ (["e","E"] ["+","-"] (["0"-"9"])+)?`
- `INTEGER_LITERAL: ["1"-"9"] (["0"-"9"])* (["e","E"] ["+","-"] (["0"-"9"])+)?`
- `HEX_LITERAL: "0X" (["0"-"9","a"-"f","A"-"F"])+`
- `LETTER: ["a"-"z","A"-"Z"]`
- `DIGIT: ["0"-"9"]`
- `IDENTIFIER: <LETTER> (("_")? (<LETTER>|<DIGIT>))*`

How does the Lexer work?

- The JavaCC lexer produces a Deterministic Finite State Automata
- DFA = (Q, ∑, T, Q_0,F)
  - a set of states Q,
  - a set of symbols (the alphabet) ∑
  - a transition function mapping Qx ∑ to Q
  - a start state Q_0 that belongs to Q
  - and a set of states F included in Q named the final states
- Deterministic?

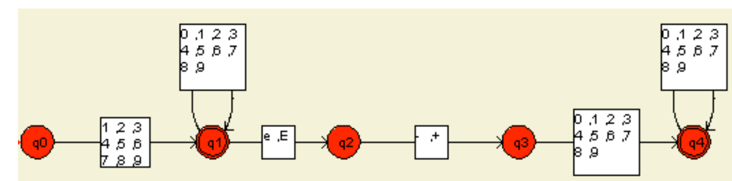
DFA for HEX_LITERAL

- `HEX_LITERAL: "0X" (["0"-"9","a"-"f","A"-"F"])+`



DFA for INTEGER_LITERAL

- `["1"-"9"] (["0"-"9"])* (["e","E"] ["+","-"] (["0"-"9"])+)?`



- http://www.cs.virginia.edu/~nc2y/dfa/

Structure of a JavaCC Definition

JavaCC

```
PARSER_BEGIN(ParserName)

Class ParserName definition

PARSER_END(ParserName)

Explicit Token definitions

Grammar Rules
```

JLex

```
Java code to access Yylex
   class
%%
%{
  Additional Yylex
   declarations
%}

Pragmas

General Token definitions
%%
State based Token definitions
```


JavaCC Lexer features

- SKIP:
  - When a production is applied NO token object is created
- TOKEN:
  - When a production is applied a token object is created and passed to the parser
- SPECIAL_TOKEN:
  - When a production is applied a token object is created and NOT passed to the parser

JavaCC Lexer features

- MORE:
  - A token is not created but the characters recognised will be part of the next created token
- STATE
  - Explicit lexical state
- Java Code
  - Not normally necessary

Parser Class Definition

```
PARSER_BEGIN(Expression)

class Expression {
  public static void main(String
    args[]) {
    System.out.println("Reading from
    standard input...");
    Expression t = new
    Expression(System.in);
    try {
      t.start();
      System.out.println("Thank you.");
    } catch (Exception e) {

    System.out.println(e.getMessage());
      e.printStackTrace();
    }
  }
}

PARSER_END(Expression)
```

- With Jlex define
  - A Lexer driver class
  - A token class
- With JavaCC define
  - A Parser class

Token Definition 1

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}
```

Definition of token separators (white space)

Token Definition 2

```
TOKEN :
{
  < REAL_LITERAL: (["0"-"9"])+ "." (["0"-"9"])+
                  (["e","E"] ["+","-"] (["0"-"9"])+)?>
|
  < INTEGER_LITERAL: ["1"-"9"] (["0"-"9"])*
                     (["e","E"] ["+","-"] (["0"-"9"])+)?>
|
  < HEX_LITERAL: "0X" (["0"-"9","a"-"f","A"-"F"])+ >
}
```

NUMBER LITERALS

Token Definition 3

```
MORE :
{
  "\"" : WithinString
}
<WithinString> TOKEN :
{
  <STRING: "\""> : DEFAULT
}
<WithinString> MORE :
{
  <"\"\"">
}
<WithinString> MORE :
{
  <~["\n","\r"]>
}
```

String Literal

- DFA?

Token Definition 4

```
TOKEN :
{
  < IDENTIFIER: <LETTER>  (("_")? (<LETTER>|<DIGIT>))* >
|
  < #LETTER: ["a"-"z","A"-"Z"] >
|
  < #DIGIT: ["0"-"9"] >
}
```

Hidden Token

Identifier

- DFA?

Questions

- Create a DFA for the following definition of Real numbers

```
(["0"-"9"])+ "." (["0"-"9"])+ (["e","E"] ["+","-"]
(["0"-"9"])+)?
```

- Create a single DFA for Integer, Real and Hex numbers
- Describe two ways how could you implement the 'consume the maximum number of characters' requirement.