

# 6.035

Fall 2007

## Project 1 Scanner/Parser Project

Michal Karczmarek  
Computer Science and Artificial Intelligence Laboratory  
Massachusetts Institute of Technology

## Why 6.035

- Many disciplines are employed in a compiler
- Bridge abstraction layers
  - Between high-level language and architecture
  - Become more efficient programmers
- Learn to design and use some useful tools
  - Language recognition
  - Tree manipulation
  - Pattern recognition
  - Optimization and parallelization frameworks
- Build a large project in a team

Michal Karczmarek

2

6.035 ©MIT Fall 2007

## My Responsibilities

- I focus on the project
  - Designing
  - Helping you
- And grading
- No office hours
  - Email me or stop by my office
- Group meeting for each project Phase
  - First meeting is should be on or before this Friday, email me to schedule it!
- Check out the Google calendar.

Michal Karczmarek

3

6.035 ©MIT Fall 2007

## Project

- Design a complete optimizing compiler for our Decaf Language targeting x86-64.
- Open-ended
  - Except for first phase you are not going to be given much.
  - The design process is a very important aspect.
  - I am here to help.
- Compiler competition at the end of the semester.

Michal Karczmarek

4

6.035 ©MIT Fall 2007

## Preliminaries

- Everyone should be a member of a group!
- Each group will have a private locker.
  - /mit/6.035/groups/leXX
  - Saman, Chen and I have access.
- Check out the skeleton infrastructure.
  - Of course, you can decide to ignore it.
- Tools to become familiar with (from 6.170):
  - CVS (or some other version control system)
  - Apache Ant
  - Eclipse (or another IDE)

Michal Karczmarek

5

6.035 ©MIT Fall 2007

## Decaf Language

- Simple Imperative Programming Language
  - Array, expressions, methods, control flow
  - No: pointers, classes, floating point
- No explicit parallelism (change from last year)
  - You will be given an analysis package, which you can use to find parallelism
  - Leverage the multiple cores of x86-64

Michal Karczmarek

6

6.035 ©MIT Fall 2007

## Lexical Analysis (Scanning)

- Covert stream of input characters into tokens.
  - Each token is created without memory of previous tokens
- A token is treated as a unit by later passes.
- The scanner will:
  - Discard whitespace (not in a string or char literal)
  - Denote keywords, integer literals, string and char literals (using delimiters), operators, and identifiers.
  - Report sensible errors for lexically malformed programs. (ANTLR errors mostly OK)

Michal Karczmarek

7

6.035 ©MIT Fall 2007

## Lexical Analysis

- Example:

```
class Program { void main () {} }
```



```
TK_class ID("Program") LCURLY
TK_void ID("main") LPAREN RPAREN
LCURLY RCURLY RCURLY
```

- Don't generate the scanner by hand, use a scanner generator: ANTLR

Michal Karczmarek

8

6.035 ©MIT Fall 2007

## Structure of ANTLR Grammar

- Header
- Options (multiple levels)
- class
- Tokens
- Rules
  - Similar to RegExp
  - Uses: | ? +
  - protected
  - Only one accepted in a conflict – eliminate conflicts!

Michal Karczmarek

9

6.035 ©MIT Fall 2007

## Scanning Example

```
tokens { "class"; }
```

```
LCURLY options : "{" ;
RCURLY options : "}" ;
```

```
ID : ('a'..'z' | 'A'..'Z')+;
```

```
int          ID("int")
class        TK_class
class x { }  TK_class ID("x") LCURLY RCURLY
i n t       ID("i") ID("n") ID("t")
```

Michal Karczmarek

10

6.035 ©MIT Fall 2007

## Syntactic Analysis

- Regular Expressions have limited expressiveness.
- Use *recursive* context-free grammars to express the structure of a programming language.
- Any questions on in class material?
  - Grammars, derivations, ambiguity resolution
- In practice we use a parser generator to automate the construction of a parser: ANTLR
- Remove ambiguity from the language spec
  - Convert to LL(k) with no conflicts for grammar
  - Enforce operator precedence and associativity

Michal Karczmarek

11

6.035 ©MIT Fall 2007

## ANTLR File Structure

- Terminal and non-terminal definitions:

```
terminal      ID, PLUS, MINUS, MULT, DIV; //from lexer
non terminal   expr;
```
- Grammar

```
expr : ID | expr (PLUS | MINUS | MULT | DIV) expr;
```

  - Infinite recursion!
- Grammar

```
expr : ID | ID (PLUS | MINUS | MULT | DIV) expr;
```

OR

```
expr : ID | expr (PLUS | MINUS | MULT | DIV) ID;
```

Michal Karczmarek

12

6.035 ©MIT Fall 2007

## Semantic Actions

- Code to execute for a rule.
- Executed after the preceding terminal / non-terminal in the rule is recognized.
- A value can be passed "up" to the enclosing rule.
- For terminals: Value the scanner associated with the terminal is accessible.

```
program : TK_class name:ID
        {System.out.println ("got id: " +
        name.getText()); } LCURLY RCURLY;
```

Michal Kazemczak

13

6.035 ©MIT Fall 2007

## Eliminating Conflicts

- Intuition: The parser does not know what to do given the tokens already seen and the next tokens (lookahead).
- Increasing k can fix some problems (use with caution: k=3 is sufficient, much higher than that may indicate bad grammar)
- Fix infinite recursions due to left-recursion
- See Chapter 3 of Tiger Book or Dragon Book.
- To investigate the source of the conflict you should look at the parser states.
  - To enable output of parse states in ANTLR, see the ant build file.

Michal Kazemczak

14

6.035 ©MIT Fall 2007

## Shift/Reduce Conflict Example

- Intuition for shift/reduce conflicts:
  - Delay decision as much as possible
- $S : 0 S 0 \mid \epsilon$
- If you are parsing "00" (. is the current position):
  - 1) .00
  - 2) Shift 0 => 0.0
  - 3) What should we do, shift another 0 (so we would expect more 0S0) or reduce by  $S ::= \epsilon$ ?
- Resolution:
  - . Rewrite the grammar:
    - . Easy in this case:  $S : (0 0 S)?$ ;
    - . Of course  $S : (0 0)^*$ ; will work as well!

Michal Kazemczak

15

6.035 ©MIT Fall 2007