Project 1
Scanner/Parser Project

Michael Gordon & Jim Sukha
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
Our Responsibilities

• We focus on the project
  – Designing
  – Helping you
• And grading
• No office hours
  – Email us or stop by our offices
• Group meeting for each project Phase
  – First meeting should be on or before this Friday, email to schedule it!
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.
  – We are here to help.

• Compiler competition at the end of the semester.
Preliminaries

- Everyone should be a member of a group!
- Each group will have a private locker.
  - /mit/6.035/groups/leXX
  - The course staff 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)
Decaf Language

- Simple Imperative Programming Language
  - Array, expressions, methods, control flow
  - No: pointers, classes, floating point

- No explicit parallelism
  - You will be given an analysis package, which you can use to find parallelism
  - Leverage the multiple cores of x86-64
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)
Lexical Analysis

• Example:

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

```plaintext
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
Structure of ANTLR Grammar

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

tokens { "class"; }

LCURLY options { paraphrase = "{"; } : "{";
RCURLY : "}";

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

int ID("int")
class TK_class
class x { } TK_class ID("x") LCURLY RCURLY
int ID("i") ID("n") ID("t")
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
Terminal and non-terminal definitions:

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

Grammar

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

- Infinite recursion!

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

- Still left recursion.

Grammar

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

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

- If the lookahead $k$ is too small, parser can match more than one rule.
- If $k = 1$, then the follow rules lead to ambiguity:
  
  LT: "<";
  LTEQ: "<=";
Conflict Example

• If the lookahead $k$ is too small, parser can match more than one rule.

• If $k = 1$, then the follow rules lead to ambiguity:

  LT: "<";
  LTEQ: "<=";

• Two possible solutions:
  – Increase lookahead to $k = 2$.
  – Left factor into a single rule.

  `LT: '<' ('='? { setType(LTEQ); } );`