

6.035
Fall 2006

Project 1
Scanner/Parser Project

Michael Gordon

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

My Responsibilities

- I focus on 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.

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.

Preliminaries

- Everyone should be a member of a group!
- Each group has a private locker.
 - /mit/6.035/groups/leXX
 - Saman 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)

Decaf Language

- Simple Imperative Programming Language
 - Array, expressions, methods, control flow
 - No: pointers, classes, floating point
- Includes optional parallelism: `forpar`
 - To leverage the multiple cores of x86-64
 - Programmer (not compiler) encodes and verifies parallelism
 - Treat just like a `for` loop for now

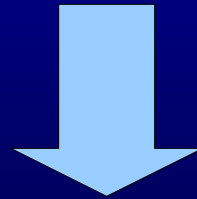
Lexical Analysis (Scanning)

- Covert stream of input characters into 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.

Lexical Analysis

- Example:

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



```
CLASS ID("Program") LBRACE VOID  
  ID("main") LPAREN RPAREN LBRACE  
RBRACE RBRACE
```

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

Structure of JFlex File

- Macros
 - Ex: `Identifier = [a-zA-Z][0-9a-zA-Z]*`
- List of lexical rules and actions
 - Regular Expressions
 - Longest / top-most match accepted
 - Action executed upon match of an expression
 - Return token type and optional attribute to parser.
 - `ytext()` returns string of current match

Scanning Example

```
<YYINITIAL> {  
    "int"          { return symbol(sym.INT); }  
    {Identifier}  { return symbol(sym.ID, new String(yytext())) ;}  
    {WhiteSpace}  { /* ignore */ }  
}
```

int

intintint

int q s

i n t



INT

ID("intintint")

INT ID("q") ID("s")

ID("i") ID("n") ID("t")

JFlex Lexical States

- If a rule has associated states, it can only be matched when the scanner is in that state.
- YYINITIAL is the initial state.
- To change state, call `yybegin (STATE) ;` from the action.
- Ex:

```
<YYINITIAL> expr1      { yybegin (S1) ; ... }  
<S1>         expr2      { ... }  
<S1>         expr1      { yybegin (YYINITIAL) ; ... }
```

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: Java CUP
- Remove ambiguity from the language spec
 - Convert to LALR(1) with no conflicts for grammar
 - Enforce operator precedence and associativity

Java Cup File Structure

- Terminal and non-terminal definitions:

```
terminal          PLUS, MINUS, MULT, DIV; //from scanner
non terminal      expr;
```

- Precedence and associativity definitions:

- disambiguates shift/reduce conflicts

- $8 + 4 * 7 \Rightarrow$ Can reduce $8 + 4$ or shift $*$

- in order of increasing precedence

```
precedence left  PLUS, MINUS;
precedence left  MULT, DIV;
```

- Grammar

```
expr ::= expr PLUS expr | expr MINUS expr
      expr MULT expr | expr DIV expr ;
```

Semantic Actions

- Code to execute for a rule.
- Executed after the last terminal / non-terminal for a rule is recognized.
- A value can be passed “up” to the enclosing rule.
- For terminals: Value the scanner associated with the terminal is accessible.
- Example (ignoring types):

```
expr ::= int:v1 PLUS int:v2 { : RESULT = v1 + v2; : } ;  
int  ::= INT_LITERAL:v { : RESULT = v; : } ;
```

Eliminating Conflicts

- Intuition: The parser does not know what to do given the tokens already seen and the next token (lookahead).
- Conflicts arise when there is a choice between:
 - shifting or reducing (shift/reduce)
 - reducing by 2 rules (reduce/reduce).
- 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 CUP, see the ant build file.

Shift/Reduce Conflict Example

- Intuition for shift/reduce conflicts:
 - Delay decision as much as possible
- $S ::= 0S0 \mid \epsilon$
- If you are parsing "00" (. is the current position):
 - 1) .00
 - 2) Shift 0 \Rightarrow 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: $S ::= 00S \mid \epsilon$

Errors

- Cup includes an `error` token that reduces and synchronizes to an error:

```
statement ::=  
    ... |  
    error SEMICOLON { : /* report error if needed : } ;
```

- Use it to report an error to the user and continue.