September 8, 2005

6.035 Scanner/Parser Project

6.035 Scanner/Parser Project

Punyashloka Biswal

MIT EECS

Today

- Project information
- Scanner/Parser
  - What? Why?
  - Tools

Project Information

- Keep work in 6.035 group "locker"
- Will get group information later today
- Group members and staff have full access—others have none
- Use `cvs`—good for snapshots!
- Java 1.5.0 memory issues on MIT server: `-Xmx64M`
- Recent version of Apache Ant in 6.170 locker (`add -f 6.170`)
- Code from this lecture at 6.035 course server

Scanner

- Converts stream of characters into stream of tokens
- **Token:** sequence of characters that can be treated as unit.
- Sequence of tokens is all that matters to compiler.

- Discard comments, whitespace
- Use punctuation to define some tokens (e.g. string literals)
- Make almost everything else into an identifier

Scanner Example

```
class Program {
  // uninformative comment
  void main() {
    callout("printf", "%d", 42);
  }
}
```

becomes

```
CLASS IDEN("Program") LBRACE VOID IDEN("main") LPAREN RPAREN
LBRACE CALLOUT LPAREN STR("printf") COMMA STR("%d") COMMA NUM
(42) RPAREN SEMI RBRACE RBRACE
```

Scanner Generation

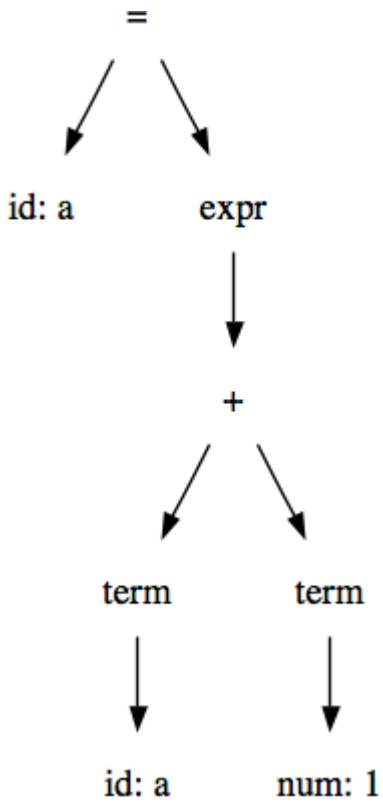
- ~~Write by hand~~ Use **JLex**
- Scanner generator: lexical specification (`.lex`)→ scanner automaton (`.java`)
- Interoperation with parser: use token names from CUP
- Example rules:

```
if              { return tok(sym.IF,NULL); }
[a-z][a-z0-9]*  { return tok(sym.ID,yytext()); }
```

- Regular expressions can do a lot, but they are never recursive!

Parser

- Converts stream of tokens into structural entities in language
- **Parse trees** capture structure of language
- `a = a + 1` becomes →
- Later phases of compiler operate on parse trees (aka concrete sytax trees) rather than text of program



Parser Generation

- **CUP** generates parser (`parser.java` and `sym.java`)
- … from context-free grammar:

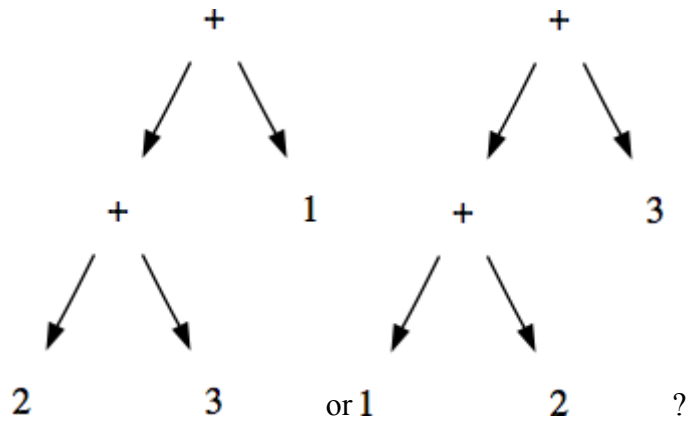
```
terminal INT, ID, PLUS, ASSIGN;
non terminal assign, expr;
start with assign;

assign := ID ASSIGN expr;
expr := expr PLUS expr;
expr := INT;
expr := ID;
```

- **Terminals** correspond directly to tokens
- **Non-terminals** are built up of terminals and other non-terminals

Parser Generation II

- Conflicts — when two different rules could apply at the same time
- e.g. how to parse `1 + 2 + 3`:



- Can resolve using precedence directives
- More general methods might be clearer to read and use

Errors

- **Recovery** — want to report **as many** errors as possible in single pass
- **Reporting** — want to report error in a way that is **helpful** to user.
  - Accurate position in code
  - Specific message
- More important in real world than in 6.035… but will prove very helpful in debugging your compiler
- In scanner — store information about lexical location and **keep** it for use in later phases
- In parser — use error rules to replace broken parse tree chunks with markers so that parsing can continue
- It's okay to give up after a particular phase if that one fails; e.g. don't try parsing if there are scanner errors

Pragmatics — Scanner

- Command line: `java JLex.Main` *`file`*
- Using `ant`

```
<target name="scanner" depends="init">

  <java classname="JLex.Main"
        classpathref="project.class.path">
    <arg value=${src}/minimal.lex />
  </java>

  <move file="${src}/minimal.lex.java"
        tofile="${genfiles}/Ylex.java" />

</target>
```

Pragmatics — Parser

- Command line: `java java_cup.Main <` *`file`*
- Using `ant`

```
<target name="parser" depends="init">

  <java classname="java_cup.Main"
        classpathref="project.class.path"
        input="${src}/minimal.cup"/>

  <move todir="${genfiles}">
    <fileset dir=".">
      <include name="parser.java" />
      <include name="sym.java" />
    </fileset>
  </move>

</target>
```

Consider

- …starting early
- …delineating individual responsibilities within team
- …using source-control (`cvs`)
- …using a build system (`ant` or `make`)
- …documenting your code
- …having fun!