Project1: Build A Small Scanner/Parser

Introducing Lex, Yacc, and POET

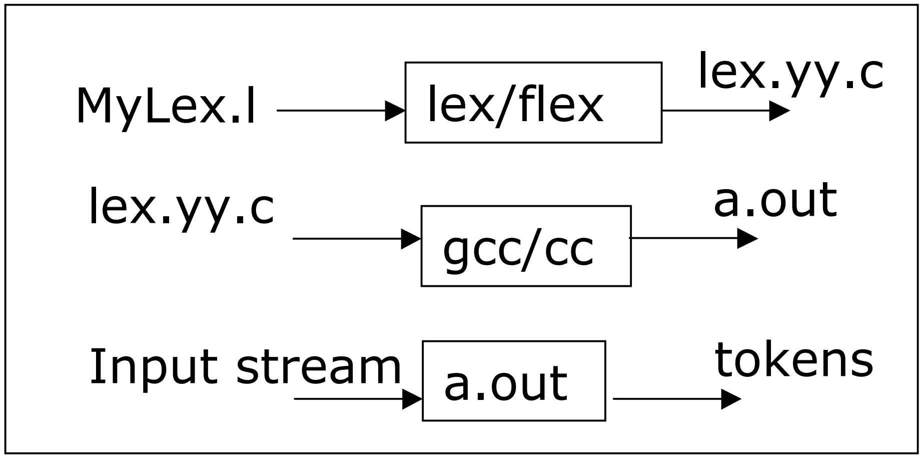
Project1:
Building A Scanner/Parser

- Parse a subset of the C language
  - Support two types of atomic values: int float
  - Support one type of compound values: arrays
  - Support a basic set of language concepts
    - Variable declarations (int, float, and array variables)
    - Expressions (arithmetic and boolean operations)
    - Statements (assignments, conditionals, and loops)
- You can choose a different but equivalent language
  - Need to make your own test cases
- Options of implementation (links available at class web site)
  - Manual in C/C++/Java (or whatever other lang.)
  - Lex and Yacc (together with C/C++)
  - POET: a scripting compiler writing language
  - Or any other approach you choose --- must document how to download/use any tools involved

This is just starting…

- There will be two other sub-projects
  - Type checking
    - Check the types of expressions in the input program
  - Optimization/analysis/translation
    - Do something with the input code, output the result
- The starting project is important because it determines which language you can use for the other projects
  - Lex+Yacc ===> can work only with C/C++
  - POET ==> work with POET
  - Manual ==> stick to whatever language you pick
- This class: introduce Lex/Yacc/POET to you

Using Lex to build scanners



- Write a lex specification
  - Save it in a file (MyLex.l)
- Compile the lex specification file by invoking lex/flex

  ```
  lex MyLex.l
  ```

  - A lex.yy.c file is generated by lex
  - Rename the lex.yy.c file if desired (> mv lex.yy.c MyLex.c)
- Compile the generated C file

  ```
  gcc -c lex.yy.c  (or gcc -c MyLex.c)
  ```

The structure of a lex specification file

declarations

```
N1 RE1
…
Nm REm
%{
typedef enum {…} Tokens;
%}
% Lex configurations
```

Token classes

```
%%
P1  {action_1}
P2  {action_2}
……
Pn  {action_n}
```

Help functions

```
%%
int main() {…}
```

- Before the first %%
  - Variable and Regular expression pairs
    - Each name Ni is matched to a regular expression
  - C declarations

    ```
    %{
    typedef enum {…} Tokens;
    %}
    ```

    - Copied to the generated C file
  - Lex configurations
    - Starts with a single %
- After the first %%
  - RE {action} pairs
    - A block of C code is matched to each RE
    - RE may contain variables defined before %%
- After the second %%
  - C functions to be copied to the generated file

Example Lex Specification(MyLex.l)

```
cconst   '([^\']+|\\\')'
sconst  \"[^\"]*\"

%pointer

%{
  /* put C declarations here*/
%}

%%
foo { return FOO; }
bar { return BAR; }
{cconst} { yylval=*yytext;
                  return CCONST; }
{sconst} { yylval=mk_string(yytext,yyleng);
                  return SCONST; }
[ \t\n\r]+   {}
.         { return ERROR; }
```

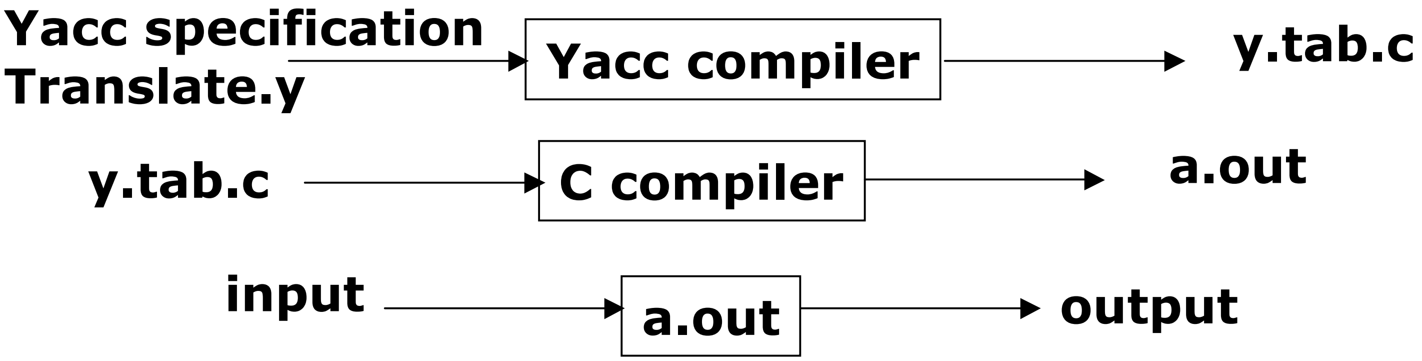
Each RE variable must be surrounded by {}

Exercise

- How to recognize C comments using Lex?
  - “/*"([^“*”]|(“*”)+[^“*”“/”])*(“*”)+”/”

YACC: LR parser generators

- Yacc: yet another parser generator
  - Automatically generate **LALR parsers (more powerful than LR(0), less powerful than LR(1))**
  - Created by S.C. Johnson in 1970's

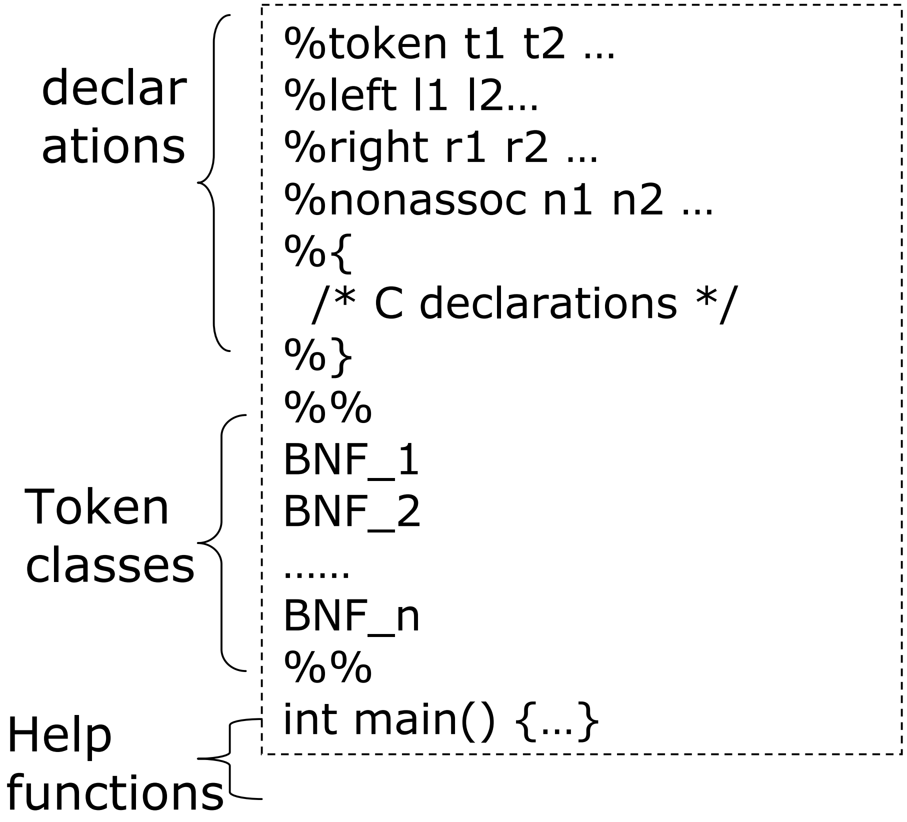


- Compile your yacc specification file by invoking yacc/bison

```
yacc Translate.y
```

  - A y.tab.c file is generated by yacc
  - Rename the y.tab.c file if desired (> mv y.tab.c Translate.c)
- Compile the generated C file: gcc -c y.tab.c (or gcc -c Translate.c)

The structure of a YACC specification file



```
%token t1 t2 …
%left l1 l2…
%right r1 r2 …
%nonassoc n1 n2 …
%{
  /* C declarations */
%}
%%
BNF_1
BNF_2
……
BNF_n
%%
int main() {…}
```

- Before the first %%
  - Token declarations
    - Starts with %token %left %right %nonassoc …
    - In increasing order of token precedence
  - C declarations

        %{
        typedef enum {…} Tokens;
        %}

    - Copied to the generated C file
- After the first %%
  - BNF or BNF + action pairs
    - An optional block of C code is matched to each BNF
    - Additional actions may be embedded within BNF
- After the second %%
  - C functions to be copied to the generated file

Example Yacc Specification

```
%token NUMBER
%left ‘+’  ‘-’
%left ‘*’  ‘/’
%right UMINUS

%%
expr : expr ‘+’ expr
       | expr ‘-’ expr
       | expr ‘*’ expr
       | expr ‘/’ expr
       | ‘(‘ expr ‘)’
       | ‘-’ expr %prec UMINUS
       | NUMBER
      ;
%%
   #include <lex.yy.c>
```

- Assign precedence and associativity to terminals (tokens)
  - Precedence of productions = precedence of rightmost token
  - left, right, noassoc
  - Tokens in lower declarations have higher precedence
- Reduce/reduce conflict
  - Choose the production listed first
- Shift/reduce conflict
  - In favor of shift
- Can include the lex generated file as part of the YACC file

Debugging output of YACC

- Invoke yacc with debugging configuration

  yacc/bison -v Translate.y

  - A debugging output y.output is produced

Sample content of y.output

```
state 699

   code5  ->  code5 . AND @105 code5   (rule 259)
   code5  ->  code5 . OR @106 code5   (rule 261)
   replRHS  ->  COMMA @152 code5 . RP   (rule 351)

   OR          shift, and go to state 161
   AND         shift, and go to state 162
   RP          shift, and go to state 710
```

The POET Language

- Questions to answer
  - Why POET?
  - What is POET?
  - How POET works?
  - POET in our class project
- Resources
  - ttp://bigbend.cs.utsa.edu

The POET Language

- Why POET?
  - Conventional approach: yacc + bison

The POET Language

- Why POET?
  - Conventional approach: yacc + bison

Source => token => AST => AST’ => …

Lex: *.lex
Syntax: *.y
AST: ast_class.cpp
Driver: driver.cpp, Makefile, …

The POET Language

- Lex + yacc
  - Separate lex and grammar file
  - flex, bison, gcc, makefile, …
  - Mix algorithms with implementation details
  - Difficult to debug

In a word: Complicated!

The POET Language

- Why poet
  - Combine lex and grammar in to one syntax file
  - Integrated framework
  - Interpreted
    - Dynamic typed
    - Debugging
  - Transformation oriented
    - Code template
    - Annotation
    - Advanced libraries

Less freedom but fast and convenient!

The POET Language

- What is POET?
  - Parameterized Optimizations for Empirical Tuning
  - Language
  - Script language

bigbend.cs.utsa.edu/wiki/POET

The POET Language

- Hello world!

```
<eval
   PRINT "Hello, world!“
```

- />

The POET Language

- Another example

```
<eval
  a = 10;
  b = 20;
  errmsg = "a should be larger than b!";
  if (a > b) {
    PRINT("a+b is" ^ (a+b));
   } else {
    ERROR errmsg;
  }
/>
```

The POET Language

- What is POET?
  - Grammar
    - C: arithmetic, control flow, variables, functions, …
    - PHP: dynamic typed, XML-style code template, …
  - Goal
    - Source to source transformation
  - Feature
    - Interpreted
    - Built-in libraries specialized for compilers
    - Annotation

The POET Language

- How POET works?
  - Source-to-source transformation
    - SED: sed
    - AWK: word
    - GREP: line
    - POET: AST node
  - Source1=>AST1=>AST2=>Source2
    - Source <=> AST: grammar, annotation
    - AST1 <=> AST2: C like transformation code

The POET Language

- Advantages
  - Grammar
    - Interpreted
    - Dynamic typed, debugging, …
  - Framework
    - Lex + Syntax => Grammar

      *.lex, *.y => grammar.pt
    - Split algorithm out of implementation detail

- Disadvantages
  - Performance
  - Learning curve
  - Freedom VS convenience

The POET Language

- POET and our class project
  - Driver
  - Grammar

```
pcg driver.pt
 –syntaxFile grammar.code
 –inputFile input.c
```

PCG: interpreter (mac, linux, windows, …)

The POET Language

- Driver.pt

```
<input to=inputCode from="input.txt" />
<eval PRINT inputCode />
```

- Grammar.code

```
<define Exp INT | BinaryExp />

<code BinaryExp pars=(left:Exp, right:Exp,
  op:"+"|"-"|"*"|"/")>
@left@ @op@ @right@
</code>
```

The POET Language

- POET and our class project
  - Built-in binaries
    - poet/lib/Cfront.code

NO: Direct use Cfront.code

YES: copy, rewrite, ask questions, …

Thanks!

The POET Language

- POET is a scripting compiler writing language that can
  - Parse/transform/output arbitrary languages
    - Have tried subsets of C/C++, Cobol, Java; Fortran
  - Easily express arbitrary program transformations
    - Built-in support for AST construction, traversal, pattern matching, replacement,etc.
    - Have implemented a large collection of compiler optimizations
  - Easily compose different transformations
    - Built-in tracing capability that allows transformations to be defined independently and easily reordered
- Supported data types
  - strings, integers, lists, tuples, associative tables, **code templates(AST)**
- Support arbitrary control flow
  - loops, conditionals, function calls, recursion
- Predefined library of code transformation routines
  - Currently support many compiler transformations

POET: Describing Syntax of Programming Languages

Example code templates for C

```
<code FunctionCall pars=(func,args) >
@func@(@args@)
</code>

<code FunctionDecl pars=(type:Type,
name:Name,
                                    params :
TypeDeclList) >
@type@ @name@(@params@)
</code>

<code FunctionDefn pars=(decl : FunctionDecl,
                                      body : StmtList) >
@decl@
{
  @body@
}

</code>
```

- Syntax of input/output languages expressed in a collection of code templates
  - Defines the grammar of a target language
  - Defines the data structure (AST) used to store the input code
- Each code template is a combination of BNF+AST
  - Code template name: lhs of BNF production
  - Code template body: rhs of BNF production
  - Code template parameters: terminals/non-terminals that have values (need to be kept in AST)
- Top-down predictive recursive descent parsing of the input

An Example Translator Using POET

```
<parameter inputFile message="input file name"/>
<parameter outputFile  message="output file name"/>

<code StmtList/>   <<* StmtList is a code template
<input from=(inputFile) syntax=“InputSyntax.code”  parse=StmtList
   to=inputCode/>    <<* start non-terminal is StmtList
<********* For project1, stop here ****************>
<eval  …… your operations to the input code ……/>

<output to=(outputFile) syntax=“OutputSyntax.code”
   from=resultCode/>
```

To run your POET code (MyParser.pt)

```
> POET/src/pcg -pinputFile=<myTestFile> -LPOET/lib MyParser.pt
```

To start you on the syntax definitions

```
include utils.incl  <<*utilities to help you
<*** content of InputSyntax.code **>
<define TOKEN  (("+" "+") ("-" "-") ("=""=") ("<""=") (">""=") ("!""=")
   ("+""=") ("-""=") ("&""&") ("|""|") ("-"">") ("*""/") CODE.INT_UL
   CODE.FLOAT CODE.Char CODE.String)/>
<define PARSE CODE.StmtList/>
<define KEYWORDS ("case" "for" "if" "while" "float")/>
<define BACKTRACK FALSE/>

<code Comment pars=(content:(~"*/")...) >
/*@content@*/
</code>
<code StmtList pars=(content) parse=LIST(Stmt,"\n") />
<code Stmt parse=(content:StmtBlock|WhileStmt|IfElseStmt|ExpStmt)/>
```

<*For more details, see the POET tutorial **>**