SYSC 3101
Programming Languages

Bison and Flex – Lab 1

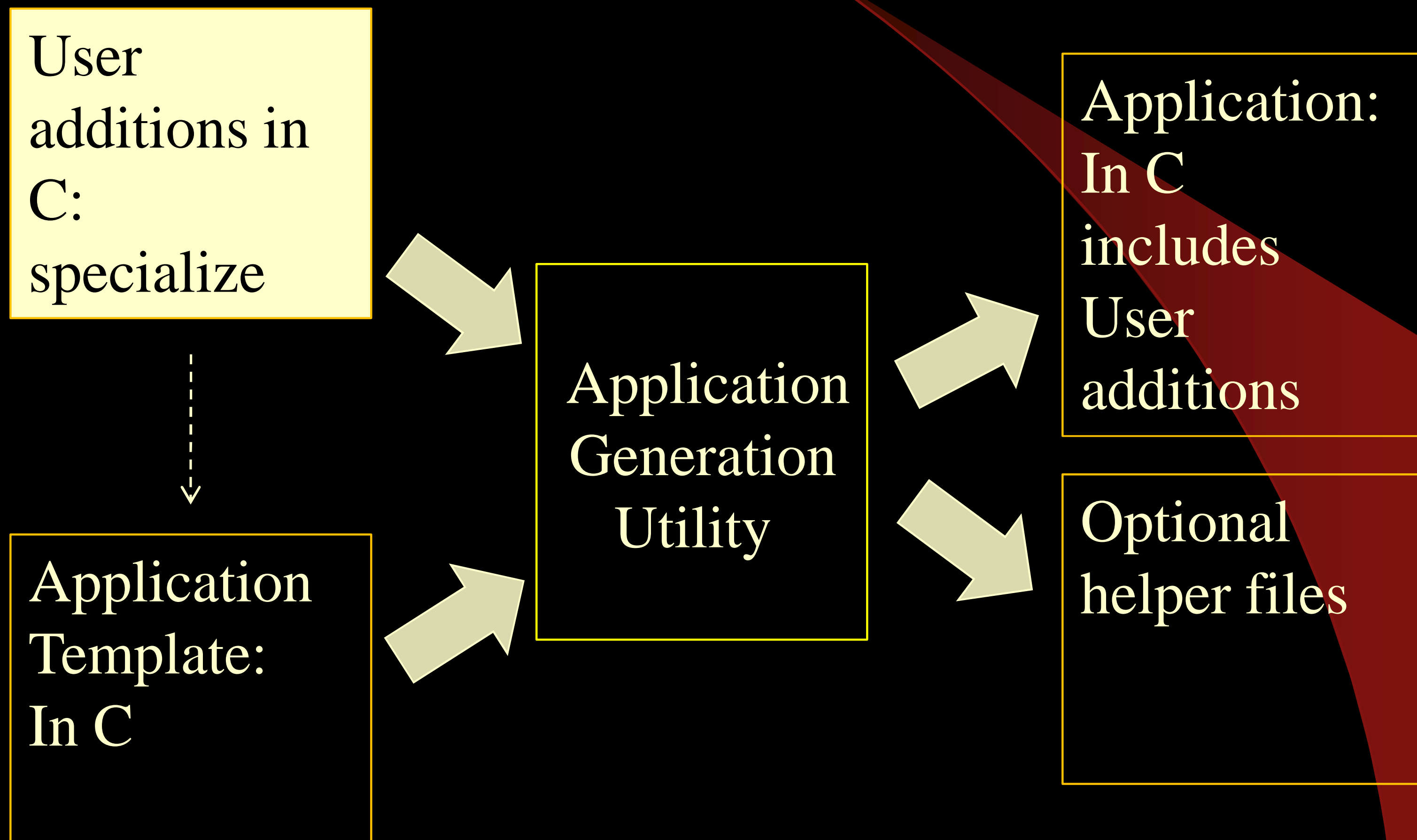
Winter 2011

Overview

- Bison: generates a C program = parser
  - Also generates C header for scanner program
- Flex: generates a C program = scanner
- Both accept input (files) that allows the generated program to be specialized
- What does it mean to "generate" ?
  - Parts of a C program: "top" (#include, extern), "body" (functions, statements)
  - Specialize: includes pieces to insert into template

Carleton UNIVERSITY

Concept



Bison Utility

- User Input = grammar (CFG) + actions (in C)
  - Perform action when rule is matched
- Output = Bison parser (in C), and header file for Scanner (in C)
- Parser file: C code for parser implementation
  - grammar + actions are inserted into the implementation code at appropriate spots
  - Provides access to parser: yyparse( )

Bison Parser

- Generated by Bison Utility (previous slide)
- Does not include a Scanner
- Parser calls function yylex() to scan
- User must provide yylex → Flex!
- Parser and scanner communicate using shared (global) variables
- Bison documentation is "thorough"

Issues yet to discuss (1)

- If parser is a function, where is main()?
- How to format the grammar and actions (i.e. input file formats)?
- Linkage to yylex?
- How to get token info back from yylex?
- How to report errors?

Bison Utility Input File

```
%{
/* put includes here – will be included at top
   of parser file */
%}
Bison declarations
%%
 Grammar rules
%%
User functions (optional)
```

Colour Code:
Bison syntax

Bison Declarations

Colour Code: **Bison syntax** **Grammar specific** **C syntax**

- Names and semantic values of **terminals**:

  %token <sem-val-type> NAME
- **Nonterminal** symbols and semantic values:

  %type <sem-val-type> SYMBOL
- <sem-val-type> above is optional (default = int)
- A **union** that includes all semantic values

```
%union{
        C-type sem-val-type;
}
```

Bison Declaration example:

terminals/nonterminals?

```
%token <anInt> INTEGER
%token <realNum> REAL
%type <anInt> S E
%union {
   int  anInt;
   double  realNum;
}
```

semantic values?

union?

Bison Grammar Rules

- Rule syntax   result : body ;
- result is a nonterminal (where are tokens defined?)
  - Every nonterminal symbol must appear as result of at least one rule
- Body → sequence of elements + action (optional)
- Default: start symbol for the grammar is the first nonterminal in the first rule
  - Can override with %start *symbol*

    in declarations part

Carleton UNIVERSITY

Grammar Rules: Body

- Consist of tokens, nonterminals, literals, actions
- literals are enclosed in single quotes
  - *eg:* '+' , '**n**' newline , '\'' single quote

Example: expr: primary ;

expr: primary '+' primary ;

2 rules (forms) for expr

Can condense as

```
expr: primary
    | primary '+' primary ;
```

Example (con’t from previous slide)

expr: primary
    | primary '+' primary ;

primary : NUM
    | IDENTIFIER
    | ‘(‘ expr ‘)’ ;

NUM & IDENTIFIER?

(indirect recursion)

Grammar Rule: Action

- Include after form part of body

  { C-statement(s); }

- Can have an action for each form in a Rule
- Perform the action when the rule is matched
- yyparse calls yyerror() routine whenever it detects an error in its input (i.e. input doesn't match any rule)

Subset of … See Bison documentation for more ...

Bison Bridge to Scanner

- Calls yylex() to scan next token
- Assumes call to int yylex(...some stuff here ...):
  - Returns token identifier (integer) for current token
  - Sets char * yytext: text of current token
  - Sets yyleng: length of text of current token
  - When appropriate ? sets yylval: the semantic value of the current token

Output from Bison Utility

- From cmd line ? : bison -d gram.y
- gram.y is input file
- Generates gram.tab.c → parser
  - #include in main.c
- -d generates gram.tab.h
  - Contains definitions for scanner bridge
  - #include this file in scanner input file

User defined name

Grammar Rules: Examples (no Actions)

```
exp: exp '+' exp ;          (direct recursion)
expseq1: exp
   | expseq1 ',' exp ;
```

```
input: /* empty */    ← ϵ !
   | input line ;
line: '\n'
   | exp '\n' ;
```

Grammar Rules with Actions

Example:

```
expr:  NUM  { printf("found a NUM expr\n"); }
  |  ID { printf("found an ID expr\n"); }
;
```

Rules and Semantic Values

- $ is special → semantic values
- $n → psuedo-variables: refer to the values returned by the components on the right hand side of the rules
- $$ → value returned by the left-hand side of a rule

More Rules with Actions

```
expr : '(' expr ')' { $$ = $2 ; } ;
 ↑      ↑    ↑    ↑
 $$     $1   $2   $3
```

- expr on the RHS is assigned the value of the expr on the LHS

What sorts of things have values at compile time?

A Complete .y Grammar file (gram3.y)

```
%{  #include <stdio.h> %}
%token NUM ;
%%
prog : sumnums ;
sumnums : NUM '+' NUM
   { printf( "Result is %d\n", $1 + $3 ); };
%%
int yyerror( char * s )
   {    return fprintf( stderr, "%s\n", s ); }
```

no %union or semantic types? default?

Invoke Bison Utility

- Cmd-prompt> bison –d gram3.y
- Will output:
  - gram3.tab.c
  - gram3.tab.h

Flex Scanner

- Concepts are similar to Bison
  - Same general layout to input file (slide 7)
  - Input file has .l (dot ell) extension
- Language = RE (no recursion in rules)
- Only give rules for tokens
- Return token identifier
- May return values through shared variables
  - yytext, yylval, yyleng, etc (see manual)

Rules

- Rule identifies specific RE and action for that RE
- Forms:

  REx action ;

  REy { action1; action2; }

Slightly different handling of ';' in compound actions!

Regular Expression Syntax

- text characters:
  - a - z, 0 - 9, *space*...
  - \n : newline
  - \t : tab
- operators: " \ [ ] ^ - ? . * + | ( ) $ / { } % < >
- n : treat next character as text character
- . : match anything (example slide 29)

Operators

- [...] : match anything within [ ] (example slide 29)
- ? : match zero or one time
  - eg: ab?c → ac, abc
- * : match zero or more times (example slide 29)
  - eg: ab*c → ac, abc, abbc...
- + : match one or more times (example slide 28)
  - eg: ab+c → abc, abbc...

More Operators

- (...) : group ...
  - eg: (ab)+ → ab, abab...
- | : alternation
  - eg: ab|cd → ab, cd
- {n,m} : repitition
  - eg: a{1,3} → a, aa, aaa

Actions

- ; → Null action
- ECHO; → printf("%s", yytext);
- {...} → Multi-statement action
- Actions can access:
  - yytext: → lexeme
    - C-String of matched characters
    - Make a copy if necessary! ← as an action!
  - yylen : length of the lexeme
  - yylval : semantic value (if computed by scanner)
  - … more!

Recall rule form:
RE action

A Complete .l Scanner File (gram3.l)

[recall Bison file gram3.y on slide 20]

```
%{
#include <stdio.h>
#include "gram3.tab.h"       ← output from Bison!
%}
%%
                            atoi( char *) ??

[0-9]+ {  yylval = atoi((char *)&yytext[0]);  ← yylval?
          return NUM;  } ← return?  (slide 14!)
```


Carleton UNIVERSITY

Complete file continued

has ablank in front of \n to include blank spaces in RE!

```
[ \n\t]* ;  /* get rid of whitespace */

. return *yytext;  /* for all else, just return the text */

%%
int yywrap() { return 1; }
```

← needed for end of file

Carleton UNIVERSITY

Invoking Flex

- Cmd-prompt> flex gram3.l
- Will output:
  - lex.yy.c ← always called this!

Main Program (main3.c)

```
#include <stdio.h>
#include <stdlib.h>
extern int yyerror();
extern int yylex();

#define YYDEBUG 1
#include “gram3.tab.c”        ← from Bison
#include “lex.yy.c”           ← from Flex
```

main.c continued

```
int main( )
{
/* yydebug = 1; */     ← uncomment this to get
                         extended output at runtime
   yyparse();          ← invoke parser
   return 0;
}
```

Compile main

- Cmd-prompt> gcc main.c
- Will output: a.exe ← executable!

- To run:
- Cmd-prompt> echo 2+3|a

Summary of CMD-line Invocation

Cmd-prompt> bison -d gram3.y

Cmd-prompt> flex gram3.l

Cmd-prompt> gcc main3.c

Cmd-prompt> echo 2+3|a ← Windows “magic”

Cmd-prompt> echo 2+w|a

If you install Bison & Flex on your machine, believe the instructions about not installing in a path with spaces in a folder name

→ “**Program Files**” contains a space ... **BAD**!