CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Today

- Class will focus on PR02:
  - structure of Bison's .y file
  - yylex and yyparse
  - the union
  - symbol tables (read esp. section 2.7.1)
  - general advice

Structure of Bison's .y file

http://dinosaur.compilertools.net/bison/bison_6.html#SEC34

"A Bison grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

Comments enclosed in '/* ... */' may appear in any of the sections."

grammar.y

```
%{
#include <stdio.h>

/* EXTERN DECLARATIONS */
extern char * yytext;
extern int yylex();

/* FORWARD DECLARATIONS */
void yyerror(const char* p);

%}

/* DIRECTIVES */
%error-verbose


/* TOKENS */
%token ID 101

/* ASSOCIATIVITY AND PRECEDENCE DECLARATIONS */
%right ...
%left  low precedence operators
%left ...
%left  high-precedence operators


/* SYNTAX TREE NODE TYPE DECLARATIONS */
%union{
   struct Basic basic;
   struct ConstantValue k;
   struct ExpressionTypeInfo t;
}

%type <basic> ID
%type <k> C_INTEGER
%type <t> expression

%start program

%%

 /* GRAMMAR RULES W/ACTIONS */

program
   : definition_list sblock  {}
   ;

%%

void yyerror(const char* p){
    // do something reasonable
}
```

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

yylex and yyparse

yylex is defined in lexer by Flex, called by yyparse.

yyparse is defined in parser by Bison, called by your code.

"the union"

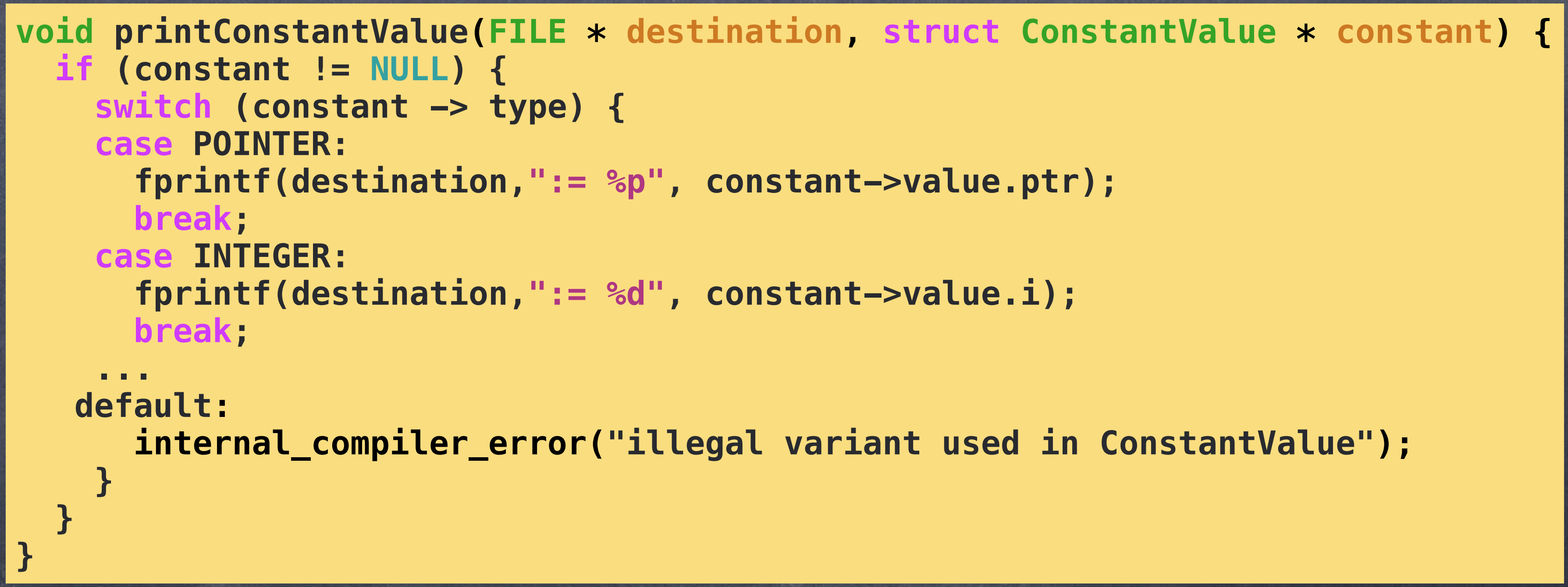
```
/* SYNTAX TREE NODE TYPE DECLARATIONS */
%union{
   struct Basic basic;
   struct ConstantValue k;
   struct ExpressionTypeInfo t;
}

%type <basic> ID
%type <k> C_INTEGER
%type <t> expression
```

other possible unions

```
enum ConstantType { POINTER, INTEGER, BOOLEAN, CHARACTER, STRING };

struct ConstantValue {
  struct SymbolTableEntry * actualType;
  int lineNo;
  int colNo;
  enum ConstantType type;
  union {
    void * ptr;
    int i;
    bool b;
    char c;
    char * s;
  } value;
  ...
  ...
};
```



```
void printConstantValue(FILE * destination, struct ConstantValue * constant) {
  if (constant != NULL) {
    switch (constant -> type) {
    case POINTER:
      fprintf(destination,":= %p", constant->value.ptr);
      break;
    case INTEGER:
      fprintf(destination,":= %d", constant->value.i);
      break;
    ...
   default:
      internal_compiler_error("illegal variant used in ConstantValue");
    }
  }
}
```

Suggestive - your code need not do exactly this.

symbol tables

One table per scope

Solid interface functions (constructors, accessors and mutators)

Good encapsulation and information hiding

Flexible design

```
/******************************************************************************
 Types
 ******************************************************************************/

struct SymbolTable;
struct SymbolTableList;
struct SymbolTableEntry;

/* Every symbol table entry must denote either a TYPE, a FUNCTION, or a
   VARIABLE.

   The type EntryCategory is used to express the kind of symbol table entry:

     TYPE is used for entries that denote types
     FUNCTION is used for entries that denote functions
     VARIABLE is used for entries that denote variables
*/
enum EntryCategory { TYPE, FUNCTION, VARIABLE };

/* Every type belongs to one of the following categories:

   PRIMITIVE is used for primitive types (such as integer, character,
   Boolean)

   PRODUCT is used for Cartesian products of types (i.e. structs/records)

   SUM is used for union (or sum) types; alpha does not currently support
   this category of type.

   MAPPING is used for mapping types: function types and array types

   UNDEFINED is used for expressions whose type is ill-defined
*/
enum TypeCategory { UNDEFINED, MAPPING, PRIMITIVE, PRODUCT, SUM };
```

```c
/******************************************************************************
 Constructors
   These functions build new values of the type indicated by the return type
   specification.
 ******************************************************************************/

/* Build and return a pointer to a new SymbolTable.  Every symbol table has a
   unique parent, except the top-level symbol table.  The top-level symbol
   table is created by the call:

       newSymbolTable(NULL)

*/
struct SymbolTable* newSymbolTable(struct SymbolTabe* parent);

/* Build and return a pointer to a new SymbolTableList.  The SymbolTableList
   has one member, table.
 */
struct SymbolTableList* newSymbolTableList(struct SymbolTable* table);

/* Build and return a pointer to a new SymbolTableEntry, of the indicated
   category.
 */
struct SymbolTableEntry* newSymbolTableEntry(enum EntryCategory category);
```

```
/******************************************************************************
 Mutators
******************************************************************************/

void addEntryToSymbolTable(struct SymbolTable* table, struct SymbolTableEntry* entry);

void addChildToSymbolTable(struct SymbolTable* parent, struct SymbolTable* child);
```

```
/******************************************************************************
 Accessors
 ******************************************************************************/

struct SymbolTable* getSymbolTable(void);

struct SymbolTable* getParent(struct SymbolTable* table);

struct SymbolTableList* getChildren(struct SymbolTable* table);

struct SymbolTableList* getRestOfChildren(struct SymbolTableList* list);

struct SymbolTable* getFirstOfChildren(struct SymbolTableList* list);

struct SymbolTableEntry* getEntryInSymbolTable(struct SymbolTable* table, char* name, bool ancestorSearch);

char * getName(struct SymbolTableEntry* entry);

enum EntryCategory getEntryCategory(struct SymbolTableEntry* entry);

enum TypeCategory getTypeCategory(struct SymbolTableEntry* entry);

struct SymbolTableEntry* getType(struct SymbolTableEntry* entry);

bool hasInit(struct SymbolTableEntry* entry);

int_least32_t makeSymbolTableID(int lineNumber, int colNumber);

struct SymbolTable* getSymbolTable(struct SymbolTableEntry* entry);
```

General Advice

Start last week 😉 (but really - don't delay)

Successful teamwork: communication and collaboration

Contribute and allow contributions

Develop incrementally (and develop test cases!)

Use Kanban board