Bison and PP2

Bison is a parser generator for context-free grammars, as flex is a scanner generator for regular expressions. Bison is actually the GNU version of an older UNIX tool, yacc, "yet another compiler compiler", as one of the goals of GNU projects is to provide open-source implementations of existing UNIX programs. By default, bison generates an LALR parser.

How bison works

When constructing compilers, bison is usually used with flex. Flex recognize tokens, passes the sequence of tokens to bison, and bison parses this the sequence of tokens. Like flex, bison is designed to generate a parser in C. Many of the features you have seen in flex, such as yylval, token types (e.g., T_Int), yylloc, are usually declared in bison or by bison. Bison invokes flex by calling function yylex to get the sequence of routines.

Traditionally, bison source files have extension name of ".y". Suppose we have a file called parser.y, we can compile it with,

$ bison –dvty parser.y

This command generates the C source code for the parser, y.tab.c, and a header file, y.tab.h, for both the parser and flex. The definition of yylval, and token types (e.g., T_Int), should be defined in paser.y, and are passed on to y.tab.h by bison.

With the parser source code, we can then compile the parser with

$ flex scanner.l // build the scanner
$ gcc -o paser y.tab.c lex.yy.c -ll -ll // generate the parser

Note that in the above command also requires the output c file, lex.yy.c, from flex to build the parser.

In Decaf, we have already provided the Makefile to call bison and flex, so you don't have to manually invoke these commands. This section is mostly FYI.

Bison file format

Your input file is organized as follows (note the intentional similarities to `flex`):

```
%{
Declarations
%}
Definitions
%%
Rules and Actions
%%
User subroutines
```

The optional Declarations and User subroutines sections are used for ordinary C code that you want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines

to the bottom.

The Definitions section is where you configure various parser features such as defining token codes, establishing operator precedence and associativity, and setting up the global variables (e.g., yylval) used to communicate between the scanner and parser.

The Rules and Actions section is where you specify the grammar rules. As in `flex`, you can associate an action with each pattern (this time a production), which allows you to do whatever processing is needed as you reduce using that production.

Writing the Definition section – Symbol Attributes and Tokens

The parser allows you to associate attributes with each grammar symbol, both terminals and nonterminals. For terminal symbols, the global variable **`yylval`** is used to communicate the particulars about the token just scanned from the scanner to the parser. For nonterminals, you can explicitly access and set their attributes using the attribute stack.

By default, **`YYSTYPE`** (the type of yylavl) is just an union. Usually you want to different information for various symbol types, so a union type can be used instead. You indicate what fields you want in the union via the **`%union`** directive.

```
%union{
   int          integerConstant;
   bool         boolConstant;
   char         *stringConstant;
   Decl         *decl;
}
```

For example, the above YYSTYPE defines a union type that use field “integerConstant” of yylval to store value of an integer constant token and use field “boolConstant” for the value of a boolean constant token. The field “decl” is used for non-terminal symbol Decl (declaration), with more explanations in the later. You should expand this union type to include the types of the values for all of the types of tokens.

The above line would be included in the definitions section. It is translated by **`bison`** into C code as a **`YYSTYPE typedef`** for a new union type with the above fields. The global variable **`yylval`** is of this type, and parser stores variables of this type on the parser stack, one for each symbol currently on the stack.

In the Definition section, you will also have to define all the tokens are that used in your grammar and in flex (e.g., T_Int). When defining each token, you can identify which field of the union is applicable to this token type by preceding the token name with the fieldname enclosed in angle brackets. The fieldname is optional (for example, it is not relevant for tokens without attributes).

```
%token   <integerConstant> T_IntConstant
%token   <doubleConstant> T_DoubleConstant
```

The above token definition says T_InConstant tokens should use field “integerConstant” of yylval, while T_DoubleConstant tokens should use field “doubleConstant” of yylval.

To set the attribute for a nonterminal, use the **%type** directive, also in the definitions section. This establishes which field of the union is applicable to instances of the nonterminal:

%type <decl> Decl

The above type definition specifies that for non-terminal token Decl, it value is accessed using the “decl” field of yylval.

For tokens without values, you can simply declare them without field names. For example, to declare keywords, you can write,

%token T_Int T_For

You will need to add all necessary tokens to your parser.y.

Writing the Definition section – Associativity and Precedence

For simplicity, you may want to define expression with grammar
E -> E ‘+’ E | E ‘-‘ E | E ‘*’ E | E ‘/’ E (note => is not a bison way of writing grammar)

While this grammar is simple, it is ambiguous as we have discussed in class, due to its lacking of precedence and associativity. So add the precedence and associativity, you can write the following in the definition section,

```
%left   '+' '-'
%left   '*' '/'
```

This two lines specify that +, - * and / are all left associative. And the fact that + and – are declared before * and / makes them having low precedence than * and /.

Another way to set precedence is by using the **%prec** directive. When placed at the end of a production with a terminal symbol as its argument, it explicitly sets the precedence of the production to the same precedence as that terminal. This can be used when the right side has no terminals or when you want to overrule the precedence given by the rightmost terminal.

Even though it doesn’t seem like a precedence problem, the dangling else ambiguity can be resolved using precedence rules. Think carefully about what the conflict is: Identify what the token is that could be shifted and the alternate production that could be reduced. What would be the effect of choosing the shift? What is the effect of choosing to reduce? Which is the one we want?

Using **bison**'s precedence rules, you can force the choice you want by setting the precedence of the token being shifted versus the precedence of the rule being reduced. Whichever precedence is higher wins out. The precedence of the token is set using the ordinary **%left**, **%right**, or **%nonassoc** directives. The precedence of the rule being reduced is determined by the precedence of the rightmost terminal (set the same way) or via an explicit **%prec** directive on the production.

In Decaf, some operators may have no associativity, for example '.' and '['. You will also have to handle unary operator (such as negative sign" specially. For more information on this topic, please read bison's documentation at http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html.

Writing Rules and Actions

Bison rules separates the left-hand and right-hand of the rules with ":" instead of an arrow. You can use "|" to separate right-hands. Your rules should also finish with an ";". Each right-hand to occupy one row. For example, for productions E -> E '+' E | E '-' E | E '*' E | E '/' E, we can writhe

```
E : E '+' E
  | E '-' E
  | E '*' E
  | E '/' E
  ;
```

One of your main jobs is write all rules for decaf.

Your second job is to add actions to construct the abstract syntax tree. To do that you need to access the values (attributes) of your terminal and non-terminal symbols. To access a grammar symbol's attribute from the parse stack, there are special variables available for the C code within an action. **`$n`** is the attribute for the nth symbol of the current right side, counting from **`1`** for the first symbol. The attribute for the nonterminal on the left side is denoted by **`$$`**. If you have set the type of the token or nonterminal, then it is clear which field of the attributes union you are accessing. If you have not set the type (or you want to overrule the defined field), you can specify with the notation **`$<fieldname>n`**. A typical use of attributes in an action might be to gather the attributes of the various symbols on the right side and use that information to set the attribute for the nonterminal on the left side.

A similar mechanism is used to obtain information about symbol locations. For each symbol on the stack, the parser maintains a variable of type **`YYLTYPE`**, which is a structure containing four members: first line, first column, last line, and last column. To obtain the location of a grammar symbol on the right side, you simply use the notation `@n`, completely parallel to **`$n`**. The location of a terminal symbol is furnished by the lexical analyzer via the global variable **`yylloc`**. During a reduction, the location of the nonterminal on the left side is automatically set using the combined location of all symbols in the handle that is being reduced. To access the location of the **n**'th symbol, use **`@n`**.

The actual class definitions for AST tree nodes are already provided to you. Therefore, you only need to create objects based on these classes and connect these objects to form the AST. For example, consider the declaration-list grammar given in the parser.y,

```
DeclList : DeclList Decl      { ($$=$1)->Append($2); }
         | Decl               { ($$ = new List<Decl*>)->Append($1); }
         ;
```

The first rule, DeclList -> DeclList Decl, says that a declaration-list derives a (shorter) declaration-list with a declaration (of class, function, variable or interface). The action of reducing with this rule, is ($$=$1)->Append($2);. $$ is the value of DeclList on the left-hand of the rule, the $1 is the value of the first symbol (DeclList) on the right-hand, and $2 is the value of the second symbol (Decl) on the right-hand. The meaning of ($$=$1)->Append($2); is appending the last declaration to the list of nodes of the right-hand declaration-list, and assign this list to be the value of the left-hand declaration-list.

Another example is to create a variable declaration,

```
Variable :   Type T_Identifier    { $$ = new VarDecl(new Identifier(@2, $2), $1); }
             ;
```

This is essentially rule, Variable -> Type T_Identifier, a line of variable declaration with type and variable name. Again, $$ represents the value of Variable, $2 represents the value of T_Identifier, and $1 represents the value of Type. @2 represents the location (row and column number in yylloc) of T_Identifier. The action of this rule is then; 1) create new Identifier object with the location and value of T_Identifier with new Identifier(@2, $2); 2) create ad new VarDecl object with the T_Identifer object and the value of the Type (the new VarDecl function); 3) assign the new VarDecl object to Variable.

The final rule for a variable declaration statement then can be,

```
VarDecl  :   Variable ';'      {$$ = $1;}
             ;
```

Conflict Resolution

What happens when you feed **bison** a grammar that is not LALR(1)? **bison** reports any conflicts when trying to fill in the table, but rather than just throwing up its hands, it has automatic rules for resolving the conflicts and building a table anyway. For a shift/reduce conflict, **bison** will choose the shift. In a reduce/reduce conflict, it will reduce using the rule declared first in the file. These heuristics can change the language that is accepted and may not be what you want. Even if it happens to work out, it is not recommended to let **bison** pick for you. You should control what happens by explicitly declaring precedence and associativity for your operators.

For example, ask **bison** to generate a parser for this ambiguous expression grammar that includes addition, multiplication, and exponentiation (using `'^'`).

```
%token T_Int
%%

E   : E '+' E
    | E '*' E
    | E '^' E
    | T_Int
    ;
```

When you run **`bison`** on this file, it reports: `conflicts: 9`

```
shift/reduce
```

In the generated **`y.output`**, it tells you more about the issue:

```
...
State 6 contains 3 shift/reduce conflicts.
State 7 contains 3 shift/reduce conflicts.
State 8 contains 3 shift/reduce conflicts.
...
```

If you look through the humanreadable **`y.output`** file, you will see it contains the family of configuring sets and the transitions between them. When you look at states 6, 7, and 8, you will see the place we are in the parse and the details of the conflict. Each state has productions which can tell which rules are causing errors and should be adjusted.

A common source of reduce/reduce conflicts are due to the use of ε-rules. For example, the follow grammar,

```
Formals   :   FormalList              { $$ = $1; }
              |   /* empty */         { $$ = new List<VarDecl*>; }
              ;

FormalList:   FormalList ',' Variable  { ($$=$1)->Append($3); }
              |   Variable           { ($$ = new List<VarDecl*>)->Append($1); }
              ;
```

is better than,

```
Formals   :   FormalList              { $$ = $1; }
              ;

FormalList:   FormalList ',' Variable  { ($$=$1)->Append($3); }
              |   Variable               { ($$ = new List<VarDecl*>)->Append($1); }
              |   /* empty */          { $$ = new List<VarDecl*>; }
              ;
```

The second grammar may cause reduce/reduce conflicts.