The Bison Parser Generator

- As with the scanner, implementing a parser used for a real language can involve a great deal of work and required care to get many details right.

- Luckily, as with scanners, tools called ***parser generators*** exist to automate much of this work. Parser generators typically allow a compiler writer to specify the syntax of a language in the form of a context-free grammar, automatically generating a working parser based on this specification.

- One of the most popular parser generators is Bison, which is an open-source successor of the parser generator Yacc.

- Bison works very much like Flex, allowing a compiler writer to specify a language's syntax as a context-free grammar with C/C++ actions attached to the grammar's productions to be executed when the productions are matched in a source program.

- Indeed, as we will see, Bison and Flex are designed to be able to work together, and Bison also outputs C/C++ code implementing an executable parsing function, which can be compiled into an application and called to parse input source code.

- In fact, the format of a Bison parser specification is very similar to the format for a Flex scanner specification, using similar syntax and broken down into the same sections (a Bison parser specification is usually written in a file with a `.y` extension):

```
%{
Prologue
%}
Declarations
%%
Rules
%%
Epilogue
```

- Let's look at how to write a parser in Bison. We'll do this in the context of a specific parsing task. In particular, we'll write a parser for the simple language we looked at when learning about Flex. As a refresher, here are the syntax features of this language:
    - The language allows only semicolon-terminated assignments, e.g.:

      ```
      a1 = b + 2;
      ```

    - The operations available (in addition to assignment) are: `+`, `-`, `*`, `/`.
    - Expressions may contain any combination of identifiers and numbers.
    - An identifier in the language consists of a lower-case letter and an optional single digit, e.g. `x` or `x2`.
    - A number can be an integer or a floating point number. All numbers are unsigned.
    - Expressions may be grouped in parentheses `()`.
    - All whitespace is ignored.

- Our goal will be to write a parser for this language that executes all of the specified computations and keeps track of the value of each variable. At the end of the parse, our parser will print out the values of all of the stores variables.

The parser prologue

- Let's start writing our parser with the prologue. Just like in a Flex scanner specification, the prologue contains C/C++ code that is copied directly to the top of the output parser file.

- We can start our prologue with some basic include statements:

```
%{
#include <iostream>
#include <map>
%}
```

- Here, we're including the `map` header to get access to C++'s `map` data structure, which we'll use to implement a basic symbol table that maps variable names to their numerical values.

- We can declare our symbol table next as a global variable in the generated parser file:

  ```
  std::map<std::string, float> symbols;
  ```

- Here, we're mapping strings (the variable names) to `float` values (the numerical value of the corresponding variable).

- Next we'll include a prototype for the function `yyerror()`. This is a function that we *must* define. Its purpose is simply to report an error, given a message describing the error:

  ```
  void yyerror(const char* err);
  ```

- We'll actually define `yyerror()` later, in the epilogue, so we just put its prototype here.

- In addition, we'll include a prototype of the scanning function, `yylex()`, which will come from our Flex scanner (defined elsewhere, which is why we include the `extern` keyword):

  ```
  extern int yylex();
  ```

- We'll talk more later about how to interface between our Flex scanner and our Bison parser.

- For now, this is all we'll need in our prologue.

Bison declarations

- Just like with Flex, the declarations section of the parser specification is an opportunity to specify how our parser will behave. We'll use this section to make several specifications for the parser we're writing.

Declaring types for grammar symbols

- Importantly, the declarations section is where we'll declare all of the terminals and nonterminals we'll use in our grammar and associate a specific data type with each one.

- The terminals we'll use will be the syntactic categories output by our scanner. For this language, we have several terminal symbols: `IDENTIFIER`, `NUMBER`, `EQUALS`, `PLUS`, `MINUS`, `TIMES`, `DIVIDEDBY`, `SEMICOLON`, `LPAREN`, and `RPAREN`.
    - By convention, terminal symbols are specified in all capital letters in a Bison specification, while nonterminals are specified in all lower-case.

- As we'll see, we can hook our scanner up to return to the parser the lexeme value associated with each word in the source code file, in addition to the syntactic category. In fact, we can return lexemes of different data types for different syntactic categories.

- This will be useful for the language we're parsing. In particular, for identifiers, it will be useful to have the scanner return the lexeme as a string corresponding to the text of the identifier, while for numbers, it will be useful to have the scanner return the lexeme as a numerical value (e.g. `float` values).

- For the other syntactic categories in our language, the lexeme does not convey any additional information beyond the syntactic category, so these can simply be represented with an integer code equivalent to the syntactic category itself.

- Thus, we need to be able to represent lexemes with three different data types, depending on the syntactic category of the lexeme: `string`, `float`, and `int`.

- We can tell Bison that we want to use these three different data types by specifying a `%union` declaration:

```
%union {
     float value;
     std::string* str;
     int token;
}
```

    - Note that we're using *pointers* to strings here because the `%union` declaration results in a `union` C/C++ data type being created, and types with non-trivial constructors are not allowed in a `union`.

- Note that if we only had a single data type (e.g. string) that worked for all of our different syntactic categories, we could have instead used a declaration like this:

```
%define api.value.type { std::string* }
```

- With our data types specified, we can assign the specific type we'll use with each terminal symbol. We do this using `%token` declarations.
    - Bison refers to nonterminals as "tokens".

- For example, here's a `%token` declaration to tell Bison we want to treat identifiers as strings:

```
%token <str> IDENTIFIER
```

- Here, `IDENTIFIER` is the specific syntactic category we'll use for identifiers. This will actually become defined as an integer value we can use later to refer to identifiers.

- Importantly, note that in the `%token` declaration above, we associate the `std::string*` data type with the `IDENTIFIER` syntactic category by specifying the *name* of the `std::string*` within the `%union` declaration above, i.e. `str`.

- Following this pattern, here's a `%token` declaration for our `NUMBER` syntactic category, again using the name `value` from the `%union` declaration above:

```
%token <value> NUMBER
```

- We can declare many tokens at the same time. For instance, here are `%token` declarations covering the rest of our nonterminals, again using the name `token` of the `int` data type from our union above:

```
%token <token> EQUALS PLUS MINUS TIMES DIVIDEDBY
%token <token> SEMICOLON LPAREN RPAREN
```

- Now is an opportunity to also declare the nonterminal symbols we'll use in our grammar and to associate a type with those, if we'd like.

- For this language, we'll use the following three nonterminals:
    - `expression` – we'll use this nonterminal to represent algebraic expressions in the language, i.e. the right-hand sides of assignment statements. Since our ultimate goal here is to track the numerical values

being assigned to variables, we'll want to be able to compute the numerical value associated with an expression. For that reason, we'll want to assign a numerical type (i.e. `float`) to expressions.
    - `statement` – we'll use this nonterminal to represent a single assignment statement. As we'll see, the specific type associated with a statement will not be important.
    - `program` – finally, we'll use this nonterminal to represent an entire program, i.e. a collection of statements. Again, the specific type associated with a program will not be important.

- Given these three nonterminals and their type requirements, we can use a `%type` declaration to assign types to the specific nonterminals. Since `expression` is the only nonterminal for which we need a type, it's the only one we'll include in the declaration. As with the `%token` declarations above, we use the *name* (`value`) corresponding to the `float` type in our `%union` declaration:

```
%type <value> expression
```

Declaring operator precedence and associativity

- Next, we'll specify some declarations to indicate the precedence and associativity of the operators in our language. Bison will use these declarations to help decide how to apply grammar productions to match the source code.

- There are four operators in the language for which we'll want to specify precedence and associativity: `PLUS`, `MINUS`, `TIMES`, and `DIVIDEDBY`.

- In this case, the `PLUS` and `MINUS` operators have the same precedence, and the `TIMES` and `DIVIDEDBY` operators have the same precedence. All four operators are left-associative, which means that, in the absence of parentheses, operations of the same precedence should be executed from the left.
    - For example, in the expression `5 - 4 + 3`, the operation `5 - 4` would be computed first, and then 3 would be added to the result. The operations are executed from the left.

- We can start by using a `%left` declaration to declare that both `PLUS` and `MINUS` are left-associative:

```
%left PLUS MINUS
```

- Here, since `PLUS` and `MINUS` appear in the same declaration, they will be assigned the same precedence.

- To assign `TIMES` and `DIVIDEDBY` a *higher* precedence than `PLUS` and `MINUS`, we can place their `%left` declaration *below* the one for `PLUS` and `MINUS` in the declarations:

```
%left TIMES DIVIDEDBY
```

- If we had operators whose order of precedence was higher than than `TIMES` and `DIVIDEDBY`, we could add them in associativity declarations below the one for `TIMES` and `DIVIDEDBY`. Bison treats operators declared later as having higher precedence.

- If we had any operators that were right-associative (i.e. ones that should be executed from the right, such as negation), we could use Bison's `%right` declaration.
    - Bison also has a `%nonassoc` declaration to declare that an operator is non-associative, i.e. that it is a syntax error to find that operator used twice "in a row."
    - The `%precedence` declaration can also be used to indicate precedence for an operator without assigning associativity. This means that associativity-related conflicts for for this operator will be reported as compile-time errors.

Declaring a start/goal symbol

- As our last declaration, we'll specify that the `program` nonterminal will be the start/goal symbol for our grammar:

```
%start program
```

Defining our grammar

- At this point, we can move on to the rules section of the parser specification and start to define the grammar for our language.

- At this point, we'll only specify the grammar, and we'll hold of on associating actions with individual productions until later.

- The grammar we define for our language will be pretty simple. We'll start by writing productions for the `program` nonterminal:

```
program
      : program statement
      | statement
      ;
```

- This syntax should be pretty easy to understand, given our previous experience writing grammars. In particular, we have two productions for the `program` nonterminal: one that says a program is a program followed by a single statement and one that says a program is just a single statement.

- Importantly, note that the use of whitespace in this rule is simply a matter of code style. The rule could all have been written on a single line (terminated by a semicolon) or broken differently into lines (e.g. including the right-hand side of the first production on the same line as the `program` on the left-hand side).

- Also, note that we're using left recursion in the first production for `program` here. In fact, because Bison is an LR(1) parser (actually LALR(1)), left recursion is actually preferable to right recursion.

- In our language, we have only one type of statement, an assignment statement. We can write a rule for recognizing a statement as follows:

```
statement
      : IDENTIFIER EQUALS expression SEMICOLON
      ;
```

- Again, this rule should be straightforward to understand: a statement consists of an identifier, followed by an equals sign, followed by an expression, and terminated by a semicolon.

- Finally, we can write a rule for recognizing expressions. Because we have already specified the precedence and associativity of the operators we'll use in expressions, the productions we write for expressions can be straightforward:

```
expression
     : LPAREN expression RPAREN
     | expression PLUS expression
     | expression MINUS expression
     | expression TIMES expression
     | expression DIVIDEDBY expression
     | NUMBER
     | IDENTIFIER
     ;
```

Compiling our parser

- Before we move on, let's look at how we use Bison to compile our parser, which we should be able to do now that we've specified our grammar.

- Assuming our parser is specified in a file called `parser.y`, we can use the `bison` tool like so to generate C++ code for our parser in `parser.cpp`:

  ```
  bison -o parser.cpp parser.y
  ```

- In our situation, though, we want to be able to integrate our parser with the scanner we wrote previously for this language using Flex. We can make this easier by using the `-d` option to tell `bison` to output a header file containing definitions that can be used in the scanner specification:

  ```
  bison -d -o parser.cpp parser.y
  ```

- This will result in two files being produced: `parser.cpp` and `parser.hpp`.

- If you open `parser.hpp` to look at it, you will see that it contains definitions of integer codes corresponding to our grammar's terminal symbols and other useful definitions.

Updating our scanner to work with our parser

- With `parser.hpp` generated, we can now turn our attention briefly to our scanner to make some small changes so that the scanner can work together with our parser.

- When we use `bison` to generate code for our parser, the key thing it generates is a function called `yyparse()`, which is the function we can call to run our parser, similar to the way we called `yylex()` to run our Flex-generated scanner.

- In fact under the default setup, `yyparse()` will now become the only function we call, and `yyparse()` will call `yylex()` every time it needs a new token from the input source code.

- The main change we have to make in our scanner specification, then, is to have the scanner *return* the syntactic category of each word it recognizes in the input source code.
    - In some situations, we'll also want to return the lexeme associated with the recognized word.

- Assuming we have the source code from our old scanner for this language, we can start with the scanner's initial declarations.

- In particular, we'll start by including the `parser.hpp` file that was generated by `bison`, so we have available (among other things) the integer codes for the terminal symbols in our grammar:

```
#include "parser.hpp"
```

- We can actually remove much of the other code in the initial declarations, since we'll no longer need to keep a running list of all of the lexeme/syntactic category pairs.

- With that done, we can now go about remaking the scanner to return token representations.

- We can start with the `IDENTIFIER` token. Our scanner already included a rule for matching identifiers, using the pattern `[a-z][0-9]?`. Our action for this rule must now return two things to the parser: the lexeme associated with the recognized identifier and its associated syntactic category.

- There is a specific Bison/Flex paradigm for accomplishing this. Specifically, our parser code will contain a global variable called `yylval`. This is used to store the lexeme value associated with a specific word recognized in the source code.

- Indeed, if you look in parser.hpp, you'll see a declaration for `yylval`:

  ```
  extern YYSTYPE yylval;
  ```

- Just above that, you'll see the `YYSTYPE` defined as the `union` we specified with a `%union` declaration in our parser specification, containing the fields `value`, `str`, and `token`.

- Combining all of this information together, the first step of the action in our Flex rule for recognizing identifiers will be to save the string value of the identifier lexeme (which, as you probably remember, we can get from `yytext`) in `yylval.str`:

  ```
  yylval.str = new std::string(yytext, yyleng);
  ```

- Remember, because of the constraints placed on us by the %union declaration, we need to use a `std::string*` type here, which is why we dynamically allocate a new `std::string`.

- With the lexeme value for our identifier successfully stored in `yylval`, we can now simply return the integer code corresponding to our `IDENTIFIER` syntactic category:

  ```
  return IDENTIFIER;
  ```

- We can update the action for our `NUMBER` rule similarly, simply storing the numerical value of the recognized number token in `yylval.value`:

  ```
  yylval.value = atof(yytext);
  return NUMBER;
  ```

  - Don't forget to `#include <cstdlib>` in the initial scanner declarations to be able to use `atof()`.

- For the remainder of our scanner rules, we can simply set `yylval.token` to the same integer code we return for the syntactic category, e.g. for the `PLUS` operator:

  ```
  return (yylval.token = PLUS);
  ```

- Finally, in our catch-all rule for unrecognized symbols, we can simply return the ASCII integer value of the invalid character:

```
return (yylval.token = yytext[0]);
```

- We can also get rid of the definition of the `main()` function in our scanner's user code section, since we'll be writing a `main()` function elsewhere now.

Associating actions with our grammar

- Now that we have a more clear picture about what our scanner will be returning for each token it recognizes, we're ready to assign actions to the productions in our grammar.

- To understand how to write actions, remember that Bison is essentially building a parse tree based on the input tokens from the source code.

- Bison is a bottom-up parser. This means that it starts building the parse tree from the leaves up. At each iteration, finds symbols on the upper fringe of the parse tree that match the right-hand side of some grammar production and adds a parent node in the tree corresponding to the left-hand side of that production.

- In other words, when Bison matches a production, the semantic values of the symbols on the right-hand side of the production are already known, and we can use those values to compute the semantic value for the symbol on the left-hand side of the production.

- Within an action in Bison, we can refer to the semantic value of the n'th symbol on the right-hand side of a production with the notation `$n`. In other words, we can refer to the semantic value of the first symbol on the right-hand side of the production as `$1`, the second symbol as `$2`, and so forth.

- In addition, we can refer to the semantic value of the left-hand side of the production as `$$`.

- Typically in an action, we will use some or all of the values `$1`, `$2`, etc. to compute and assign a value to `$$`.

- Let's see how this works in the context of the parser we're writing. We'll start by looking at the rules for the `expression` nonterminal.

- In the grammar we defined above, the first production for `expression` was `expression → LPAREN expression RPAREN`.

- Remember, when this production is matched, it means the parser has already found (and presumably has semantic values for) symbols matching the right-hand side of the production.

- Given that we know the semantic values of the symbols on the right-hand side, for this production we simply want to assign the value of the `expression` on the right-hand side to the `expression` on the left-hand side.
    - In this case, the parentheses don't affect the *value* of the `expression` on the left-hand side once we know the value of the entire `expression` inside the parentheses.

- Thus, our complete rule for this production, including the action, will look something like this:

```
LPAREN expression RPAREN { $$ = $2; }
```

- The action here is just C/C++ code that uses Bison notation to assign the value of the `expression` on the right-hand side of the production (which is the second symbol on the right-hand side) to the `expression` on the left-hand side.

- The next few rules for `expression` will do something similar:

```
expression PLUS expression { $$ = $1 + $3; }
expression MINUS expression { $$ = $1 - $3; }
expression TIMES expression { $$ = $1 * $3; }
expression DIVIDEDBY expression { $$ = $1 / $3; }
```

- In each of these rules, we already know the semantic values of both `expression` symbols on the right-hand side, which are of `float` type, as we specified with our `%union` declaration from above.

- These rules also tell us that a token output by the scanner matched a specific operation.

- Thus, we simply combine all of this information in the actions for these rules to compute the semantic value of the `expression` symbol on the left-hand side of the production by applying the appropriate operation to the values of the `expression` symbols on the right-hand side.

- So far, so good. To understand how to write the action for the next rule, remember that we updated the scanner so that when it recognizes a `NUMBER`, it not only returns the `NUMBER` token, it sets `yylval.value` to the numerical value of the recognized number, storing it there as a `float` value.

- Thus, back in the parser, each `NUMBER` nonterminal will have the appropriate numerical semantic value assigned to it.
    - Remember that we declared `NUMBER` tokens as having a floating point type with a `%token` declaration.

- This means our production `expression → NUMBER` can also have a simple action associated with it. In particular, the action will simply assign the semantic value of the left-hand side's `NUMBER` token (which is a `float` value that comes from the scanner) to the `expression` on the left-hand side:

```
NUMBER { $$ = $1; }
```

- Our action for the final `expression` rule, `expression → IDENTIFIER`, will be similar.

- In this case, when the scanner recognizes an `IDENTIFIER`, it dynamically allocates a new `std::string` into which it stores the lexeme associated with the `IDENTIFIER`, and it assigns a pointer to this string to `yylval.str`. This pointer becomes the value associated with the `IDENTIFIER` in the parser.

- There are two extra twists in the case of an `IDENTIFIER`, though. The first is that, when we encounter an identifier in an expression, we want to use the *numerical* value associated with that identifier.

- For now, we'll assume that we've successfully kept track of the numerical value associated with each identifier and stored it in our symbol table, `symbols`.
    - We'll also make the assumption, for now, that each identifier has a numerical value assigned to it before it's used in an expression.

- The second twist when dealing with an `IDENTIFIER` token is that we don't want to leak the memory that was allocated by the scanner to store the associated lexeme (i.e. the `std::string`). To ensure that we don't, we need to free that memory after we're done with it.

- In this case, we will no longer need the lexeme for the `IDENTIFIER` after the action for the `expression → IDENTIFIER` rule is executed, so we can free its associated memory within that action.

- Putting this all together, our action for this rule will look like this:

```
IDENTIFIER { $$ = symbols[*$1]; delete $1; }
```

- Here, we simply look up the numerical value associated with the recognized identifier (dereferencing the pointer stored in `$1` to get the actual string value of the identifier lexeme) and assign it as the semantic value of the `expression` on the left-hand side of the rule. We then free the memory allocated to the lexeme string.

- We've now assigned an action to every `expression` rule, and after any of those rules is matched, the `expression` symbol on the left-hand side of the rule will be assigned a numerical semantic value.

- With our work on the rules for the `expression` symbol finished, let's move on to look at the rules for the `statement` symbol.

- Remember, the only type of statement in our language is an assignment statement that assigns the value of an `expression` (which, as we've established, is numerical) to an `IDENTIFIER`.

- The grammar rule for the `statement` symbol is `statement → IDENTIFIER EQUALS expression SEMICOLON`.

- As before, by the time this rule is matched, the scanner will have assigned a `std::string*` value to the `IDENTIFIER` token in the rule's right-hand side. We will also have already computed the numerical value of the `expression` on the right-hand side.

- Thus, all we need to do in the action for this rule is store the numerical value assigned to the `expression` into the symbol table under the string associated with the `IDENTIFIER`. In addition, after this rule, we'll be finished with the `std::string*` that was allocated for the `IDENTIFIER` by the scanner, so we can free it:

```
IDENTIFIER EQUALS expression SEMICOLON { symbols[*$1] = $3;
     delete $1; }
```

- These are all of the actions we need to assign to make our parser work. The `program` symbol is only needed to make sure the parser continues to read statements as long as they're present in the source code being parsed. For our current parsing goals, no action needs to be taken for any of the rules associated with the `program` symbol.

The epilogue

- The epilogue section of the parser specification file is where we write code to be copied verbatim to the bottom of the generated parser file.

- In this case, the only thing we need to include in the epilogue is a definition for `yyerror()`, which we prototyped in the prologue.

- `yyerror()` is a function that is called by the generated parser when it encounters a syntax error. It is passed a string containing an error message (often just "syntax error").

- Our definition of this function can simply print out the error message:

```
void yyerror(const char* err) {
     std::cerr << "Error: " << err << std::endl;
}
```

Tying everything together with a driver program

- Finally, in a separate `.cpp` file, we can implement a simple driver program that calls the parser and, if the parse succeeds, print out the ending values for all of the variables in the symbol table:

```
#include <iostream>
#include <map>

extern int yyparse();

extern std::map<std::string, double> symbols;

int main(int argc, char const *argv[]) {
  if (!yyparse()) {
    std::cout << std::endl << "Symbol values:"
      << std::endl;
    std::map<std::string, double>::iterator it;
    for (it = symbols.begin(); it != symbols.end(); it++) {
      std::cout << it->first << " : " << it->second
        << std::endl;
    }
    return 0;
  } else {
    return 1;
  }
}
```

- Here, we rely on the fact that `yyparse()` returns 0 to signify success and non-zero to signify that an error occurred. We also rely on the symbol table being defined globally within the parser.

Handling errors

- One important part of parsing is giving users informative error messages when their programs contain syntax errors. Bison provides some mechanisms to help do this.

- As a first step towards providing informative error messages, we'll enable location tracking in our parser by including the following directive:

```
%locations
```

- This directive will make Bison define a new structure type `YYLTYPE` to hold location information, including information about line and column:

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

- The `%locations` directive will also cause Bison to define a new global variable `yylloc` of type `YYLTYPE`, in which locations of tokens can be tracked.

- Unfortunately, the `%locations` directive doesn't make Bison actually *perform* any location tracking. This will be up to us.

- The easiest way to track locations is in the scanner. Specifically, for every token recognized in the scanner, we will update `yylloc`.

- To do this, we can define the macro `YY_USER_ACTION` in the scanner specification. This is a special macro that, if defined, is executed before every single action in the scanner.

- To keep things simple, our `YY_USER_ACTION` definition can simply set the value of `yylloc`'s `first_line` and `last_line` fields to the current line number, as indicated by `yylineno`:

```
#define YY_USER_ACTION \
     yylloc.first_line = yylloc.last_line = yylineno;
```

- If we wanted to, we could do something more complicated, updating line and column numbers based on `yytext` and `yyleng`.

- With this action in place, we can obtain the location of any symbol from inside an action in our parser. Specifically, the location of the n'th symbol in a rule is stored in `@n`, similar to the way the semantic value of that symbol is stored in `$n`.

- For example, we could use the following simple rule to print the location of each identifier recognized as an expression:

```
expression : IDENTIFIER
     {
          std::cout << @1.first_line << std::endl;
     }
```

- We can combine our new knowledge of the location of each symbol in a rule with use of Bison's `error` symbol to perform some basic error reporting.

- Normally, when a Bison parser encounters a syntax error, it simply calls `yyerror()` and then immediately fails.

- However, we can have our parser *recover* from errors and continue parsing by placing the `error` symbol at an appropriate location in the right-hand side of a rule.

- One general way to indicate to the user the line number of each erroneous statement would be to include a second production for our `statement` symbol that looked like this: `statement` → `error SEMICOLON`.

- Then, we could define an action for this rule that looked something like this to report the erroneous statement to the user:

```
std::cerr << "Error: bad statement on line "
     << @1.first_line << std::endl;
```

- Importantly, after recovering from the error via the `error` token, our parser will no longer return 1 indicating a parsing failure. As an alternative, we could maintain a global boolean flag, e.g. `_error`, and set that flag to `true` when an error is encountered:

```
_error = true;
```

- Our `main()` function could then be adapted to check the `_error` flag to determine whether or not to print out the symbols at the end of the parse.

- We can take our error handling one step further. In particular, one common error our parser might run into is one where a user tries to use a variable before it is defined.

- To handle errors of this type, we can modify our action for the `expression → IDENTIFIER` production to make sure the identifier is defined in the symbol table before accessing its value there:

```
{
  if (symbols.count(*$1)) {
    $$ = symbols[*$1];
  } else {
    std::cerr << "Error: unknown symbol " << *$1
      << " on line " << @1.first_line << std::endl;
    _error = true;
    YYERROR;
  }
  delete $1;
}
```

- Here, we use the `YYERROR` macro to "throw" a syntax error, ultimately resulting in our `error` production for the `statement` symbol being matched.

- We could do something similar, e.g. to recognize division by zero in our `expression → expression DIVIDEDBY expression` production.

- Because of the way we used the `error` token in the production for the `statement` symbol, all errors in a source program will be reported to the user instead of just the first one.

- We can also turn on verbose parsing errors, so that the error messages passed to `yyerror()` are (slightly) more informative:

```
%define parse.error verbose
```

Push parsing

- In some situations, it may be appropriate to implement a ***push parser***.

- Up to now, we've only seen ***pull parsing***, where the parser "pulls" a token from the scanner every time it needs one by calling the scanning function `yylex()`.

- A push parser reverses this role. Specifically, in a push-parsing setup, the scanner “pushes” a token to the parser every time a new token is ready.

- Push parsing can be useful, for example, when the parser is integrated into the main event loop in an interactive application, or when the scanner can generate multiple tokens in a single action.

- To turn our parser above into a push parser requires a few changes. First, we must use parser declarations to indicate that we want to use a push parser:

```
%define api.pure full
%define api.push-pull push
```

    - The first of these declarations turns our parser into a pure (reentrant) parser by making variables like `yylval` and `yylloc` local to the parsing function instead of global. This is almost always needed when using push parsing.

- After making these changes, we’ll need to change the prototype of `yyerror()` slightly to allow it to take a location as an argument as well:

```
void yyerror(YYLTYPE* loc, const char* err);
```

- Most of the remaining changes will be in our scanner specification.

- First, since `yylval` and `yylloc` are now local to the parser instead of global, we’ll need to make versions of these variables within the scanner (e.g. in the initial declarations) that we can use there:

```
YYSTYPE yylval;
YYLTYPE yylloc;
```

- In addition, we’ll have to create a new parser state object of type `yypstate*` to keep track of the current state of the parser:

```
yypstate* pstate = yypstate_new();
```

- This object will be passed to the parser each time we make a push-parsing call.

- Finally, within the actions of the scanner, we'll have to invoke the push parser by calling `yypush_parse()` instead of returning tokens.

- `yypush_parse()` takes four arguments:
  - A `yypstate*` representing the parser's current state.
  - An integer code indicating the syntactic category of the token being pushed (e.g. `IDENTIFIER` or `LPAREN`).
  - A `YYSTYPE*` holding the semantic value of the current token (i.e. `&yylval`).
  - A `YYLTYPE*` holding the location of the current token (i.e. `&yylloc`).

- `yypush_parse()` will return an integer status code which will be `YYPUSH_MORE` if the parser is expecting more input.

- Putting this all together, we can define a simple macro `PUSH_TOKEN()` to call `yypush_parse()` and check its return value, returning from the scanning function if the status is anything other than `YYPUSH_MORE`:

```
#define PUSH_TOKEN(token) do {                  \
    int s = yypush_parse(pstate, token, &yylval, &yylloc);
\
    if (s != YYPUSH_MORE) {                     \
      yypstate_delete(pstate);                  \
      return status;                            \
    }                                           \
  } while (0)
```

- Importantly, note that before calling `PUSH_TOKEN()`, we'll have to make sure `yylval` and `yylloc` are updated appropriately.

- Then, in our scanner's actions, we can simply replace all of the `return` statements with calls to `PUSH_TOKEN()`, e.g.:

```
[a-z][0-9]? {
              yylval.str = new std::string(yytext, yyleng);
              PUSH_TOKEN(IDENTIFIER);
            }
```

- Finally, we'll need to make sure to include an `<<EOF>>` rule in the scanner that pushes a 0 token and then returns (from `yylex()`) the status returned from `yypush_parse()`, which will be 0 if the parser was expecting the end of the file:

```
<<EOF>>  {
           int s = yypush_parse(pstate, 0, NULL, NULL);
           yypstate_delete(pstate);
           return s;
         }
```

    - Note that we must free the memory allocated to `pstate` here.

- After making these changes in the scanner, we simply need to modify our `main()` function to call `yylex()` instead of `yyparse()`. After doing so, we should be able to run our parsing program as we did before.