COMP 356
Programming Language Structures
Class 11: Parsers (Section 4.5 of *Concepts of Programming Languages*)

1 Overview of parsing algorithms

The **parser** (or **syntactic analyzer**) of a compiler organizes the stream of tokens (from the scanner) into parse trees according to the grammar for the source languages.

All grammars used in designing parsers must be unambiguous. However, as we will see below, the automatic parser generator we will use (called *CUP*) allows you to use an ambiguous BNF grammar, provided you give it some extra rules, called *disambiguating rules*. The disambiguating rules specify which choices to make when constructing the parse tree. Of course, the combination of the BNF and the disambiguating rules are equivalent to an unambiguous grammar, but the approach using disambiguating rules turns out to be easier.

Parsing algorithms that can parse any unambiguous BNF grammar are $O(n^3)$, where n is the length of the string being parsed. One well-known algorithm to achieve this is the Cocke-Younger-Kasami (CYK) algorithm (it has a good Wikipedia page, if you're interested in learning more about it.)

Commercial compilers use $O(n)$ (linear) parsing algorithms that can handle specific classes of grammars, called LL(1) and LR(1) grammars. These grammars are sufficient to describe the syntax of the vast majority of programming language features.

Parsers are classified as:

- **top-down** parsers, which build parse trees from the root to the leaves. These parsers are designed using LL(1) grammars.
- **bottom-up** parsers, which build parse trees from the leaves to the root. These parsers are designed using LR(1) grammars.

Bottom-up parsers build parse trees from the leaves to the root. The idea is to use the grammar rules "backward" - the parser matches against the R.H.S. of grammar rules and when some rule is matched, it is used to construct a subtree with the R.H.S of the rule as the children and the L.H.S. as the root.

A bottom-up parser uses a stack to keep track of the parse. At each parsing step, there are two possible actions:

- **shift** - push the next token from the input onto the stack
- **reduce** - if the sequence of grammar symbols on "top" of the stack matches the R.H.S. of a grammar rule, pop them all and replace them with the L.H.S. of that rule.

The parser is controlled by a table generated from the grammar. The tables are too large and complex to construct by hand, and so are generated using a tool such as yacc or CUP (in this course, we use CUP).

Bottom-up parsers are also called LR parsers, because they parse LR grammars.

The parsers generated by yacc and CUP are LALR(1) parsers, where LALR(1) stand for Look-Ahead LR(1). These parsers are less powerful than LR parsers, because

LL(1) grammars $\subset$ LALR(1) grammars $\subset$ LR grammars.

But the loss of parsing power is small and the parse tables for LALR(1) parsers are much smaller than those for LR parsers.

If the grammar used as input to yacc or CUP is ambiguous or otherwise not LALR(1), these tools will report **parsing conflicts** as error messages. The possible conflicts are:

- **shift-reduce conflicts** - at some step, the parser can't tell whether to shift a token onto the stack or do a reduction.
- **reduce-reduce conflicts** - at some step, the parser can't tell which of two reductions to perform. This can occur when two rules in the grammar have identical R.H.S.'s, but different L.H.S.'s.

2 Using CUP

CUP generates an LALR(1) parser (in Java) from an attribute grammar specification:

- this approach works well with synthesized attributes
- each time a rule is used in a reduction, the attribute of the L.H.S. of the rule is computed (via user-specified Java code) from the attributes of the R.H.S.
- bottom-up parsing ensures that the attributes of the R.H.S. are available when needed.

JFlex scanners can easily be made compatible with CUP parsers.

A CUP specification is placed in a file with extension .cup. For an example,see the provided file `example.cup`. To generate parser code, use the command

```
java [-cp "cup.jar;."] java_cup.Main example.cup
```

This generates:

- `parser.java` which contains the parser code
- `sym.java` which contains definitions of all of the tokens used as integer constants. These must match the tokens used in the scanner. (Note that in our previous lecture, `sym.java` was produced manually as part of the demo of JFlex. When you run the above command, `sym.java` gets overwritten with an automatically-generated file. The original (generated "by hand") is available as `sym.java.byhand`, in case you're interested.)

To compile and run the parser:

1. you need to have already produced some scanner code (`Yylex.java`) using JFlex, as described in the previous lecture
2. ensure that `Yylex.java` is in the same directory as `parser.java` and `sym.java`
3. compile by executing:

   ```
   javac [-cp "cup.jar;."] parser.java
   ```

4. run the parser by executing:

```
java [-cp "cup.jar;."] parser < input
```

where `input` is the file containing the source program.

Or, to enter the source program directly from the keyboard, just enter:

```
java [-cp "cup.jar;."] parser
```

and then type the source program and press Ctrl-D when finished.

2.1 Format of a CUP specification file

The format of a CUP specification file is given by
imports
directives
declarations of tokens and nonterminals
grammar rules and actions

For an example, see the file `example.cup` on the course resources page.

The *imports* are copied directly to the beginning of the generated code in parser.java. For example,

```
import java_cup.runtime.*;
```

is usually placed here to provide access to the needed library classes.

The *directives* supply user code to be placed in the indicated class in the generated code. For example:

`parser code {:` *code* `:}`

causes the code in the special brackets `{: :}` to be placed in class `parser`.

In the *declarations of tokens and nonterminals* section, all grammar symbols are declared as follows:

- the name used for a token must match the name assigned to the `sym` field of class `Symbol` when that token is recognized by the scanner.
- if a token has an attribute (placed in the `value` field of class `Symbol` by the scanner), the type of that token must be declared and must match the attribute value assigned by the scanner.
- if a nonterminal has an attribute, the type of that attribute must also be given. The majority of nonterminals will have attributes.
- by convention, token names are in all caps and nonterminal names are all lowercase. Note that nonterminal names are not put in angle brackets, $\langle\ \rangle$.

This section also includes any specifications of associativity and precedence of tokens. These specifications are actually disambiguating rules and permit parsing of many ambiguous grammars when used appropriately. The syntax of these specifications is:

`precedence` ⟨spec⟩ `terminal {, terminal};`

where:

⟨spec⟩ → `left` | `right` | `nonassoc`

The ⟨spec⟩ specifies associativity. If a token is declared as `nonassoc`, then it can not occur multiple times in one expression. For example, `1 < 2 < 3` is illegal in many programming languages.

Precedence rules:

- all tokens listed in the same declaration have the same precedence.
- if there are multiple precedence declarations, later declarations have higher precedence.
- all tokens not listed in a `precedence` declaration have lowest precedence

For example:

```
precedence left ADDOP
precedence left MULOP
```

declares that `MULOP` has higher precedence than `ADDOP`, and that both are left associative.

In many cases, these precedence and associativity declarations can be used to remove shift-reduce and reduce-reduce conflicts from the grammar.

The *grammar rules and actions* section is an attribute grammar for the language being parsed:

- the L.H.S. of the first rule is the start symbol
- the symbol `::=` is used in place of →
- nonterminal names are not in ⟨ ⟩ (the declarations specify which names are nonterminals and which are tokens)
- each rule ends in `;`
- each symbol on the R.H.S. of a rule can be labeled using `:` and a name. For example, `expr:l ADDOP:op expr:r`, where `l`, `op` and `r` are labels
- for tokens, the label is used to refer to the intrinsic attribute assigned by the scanner (stored in the `value` field of the `Symbol`)
- for nonterminals, the attribute value is computed using **actions** associated with each rule
- the attribute of the L.H.S. of the rule is always `RESULT`

The actions associated with each rule:

- are in special brackets `{: :}`
- contain Java code that can refer to attribute labels and `RESULT`
- typically compute the value of `RESULT` based on the attribute values of the grammar symbols on the R.H.S. of the rule. Hence, the attributes are synthesized attributes.
- are executed each time the rule is used in a reduction
- can easily be used to build parse trees or do other computation

2.2 Summary of commands for JFlex and CUP

The commands needed to compile and run a CUP parser with a JFlex scanner are as follows, assuming:

- file `example.lex` contains the JFlex specification
- file `example.cup` contains the CUP specification
- file `input` contains the source program
- JFlex and CUP are installed and the CLASSPATH is set correctly. If your CLASSPATH isn't set correctly, you can put `cup.jar` in the current directory and add `-cp "cup.jar;."` to each command.

command	comment
`java JFlex.Main example.lex`	creates Yylex.java
`java java_cup.Main example.cup`	creates sym.java and parser.java
`javac parser.java`	compiles everything
`java parser < input`	runs the parser

Table 1: Commands needed to compile and run a CUP parser with a JFlex scanner.

To add tokens, you must declare them in the .cup file and also add actions to the .lex file to recognize and return them. Also, you must run CUP (to generate constant declarations for the new tokens in sym.java) before recompiling anything.

2.3 In-class exercise: adding "repeat" functionality to the calculator

This exercise asks you to add some slightly silly functionality to the calculator, but it will demonstrate many of the concepts you need to understand.

The task is to add functionality that repeats the digits of a given integer before evaluating the integer as a number. The repeat functionality will be triggered by the letter "r". For example, if the calculator encounters `r235` in an expression, this will be interpreted as the numerical value 235235. As a further example, a complete input consisting of "`3.5 + r5;`" would evaluate to 58.5.

The steps required to complete this exercise are as follows:

1. Download the required files and run all the commands in Table 1. Edit input.txt and verify that the calculator can handle expressions like "`3*(2.5+1.5);`".
2. Add a new regular definition in `example.lex`. Specifically, define a new regular expression called "REPEAT" immediately after the line beginning "NUM".
3. Add a new action in `example.lex`. Specifically, add a line after the line beginning "{NUM}", returning a new symbol for matches to the REPEAT regular expression. I recommend using a `String` as the datatype of the value for this symbol.
4. You have finished modifying `example.lex`, so now run JFlex to check there are no errors.
5. Add a new terminal called REPEAT, in `example.cup`. Do this by adding a line after the line "terminal Double NUM;".

6. Add a new rule to the grammar in `example.cup`. Specifically, add a new line after the line containing "NUM:n". The line should start with "| REPEAT". Hints: the Java code that appears between the "{:" and ":}" will need to use (i) the `substring` method of the String class, and (ii) string concatenation.

7. Run all the remaining commands (see Table 1 above). Check that inputs such as "`r235;`" and "`3.5 + r5;`" are evaluated correctly.