Syntactic Analysis

in
JavaCUP

Parsing methods

- Last week we talked about different parsing methods: Top-down methods (such as LL(1)) and Bottom-up methods.
- I mentioned several different bottom-up methods. All these methods uses a stack, and differs in when and how symbols are pushed on the stack.
- One such method is called LALR(1).

LALR(1)

- LALR(1) is probably the most practical parse method for programming languages.
- All "reasonable" language constructs can be handled by a LALR(1) parser.
- The parser can be quite compact and efficient.
- This is the method used by parser generators such as Yacc, Bison, MLYacc, Yecc, SableCC, and most importantly for you by JavaCUP.

JavaCUP

- JavaCUP, just like JLex, takes an input file describing a grammar and produces a Java program that recognizes strings in the grammar.
- JavaCUP generates two files Parser.java and Tokens.java.
- If the grammar in the input file is not LALR(1) an error message is produced.

A JavaCUP parser

```
package expression;
import java_cup.runtime*;

terminal PLUS, MINUS, TIMES;
terminal DIV, LPAR, RPAR, INT;
non terminal Expr, Term, Factor;

start with Expr;
```


A JavaCUP parser

```
Expr ::=   Expr PLUS Term
     |     Expr MINUS Term
     |     Term
     ;
Term ::=   Term TIMES Factor
     |     Expr DIV Factor
     |     Factor
     ;
Factor ::= INT
       |   LPAR Expr RPAR
       ;
```

Precedence

- We can simplify the parser by using precedence.
- This is done by adding two lines to the file.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
```

- Left means left associative, that is, 1-2-3 is parsed as (1-2)-3. (i.e., –4 and not 2)
- Lowest precedence comes on the first row, each following row has higher precedence. So 1+2*3 is parsed as 1+(2*3). (i.e., 7 and not 9)

Using Precedence

```
package expression;
import java_cup.runtime*;
terminal PLUS, MINUS, TIMES;
terminal DIV, LPAR, RPAR, INT;
non terminal Expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
start with Expr;
```


Using Precedence

```
Expr ::= Expr PLUS Expr
       |   Expr MINUS Expr
       |   Expr TIMES Expr
       |   Expr DIV Expr
       |   INT
       |   LPAR Expr RPAR
       ;
```


Shift-reduce Conflicts

- With an ambiguous grammar the parser generator can not decide whether to shift (push a token on the stack) or reduce the contents on the stack to a non terminal.
- E.g., `Expr ::= Expr PLUS Expr`
- Three solutions:
  1. Use precedence directives.
  2. Rewrite the grammar.
  3. Leave it as it is: shift will be chosen.

Not the right solution for the example!

Reduce-reduce Conflicts

- Reduce-reduce conflicts are not so common and often indicate a problem with the language rather than the grammar.
- They generally has to be solved by rewriting the grammar.
- (They can come from trying to solve semantic issues by syntactic rules.)

Syntax vs. Semantics

- The syntax describes the structure of a sentence, and the semantics describes its meaning.
- Example:

  We do not want to allow integer arithmetic on boolean values:

  1 + true.

Wrong solution

```
Expr ::= IExpr | BExpr
IExpr ::= IExpr PLUS IExpr
       |  INT
BExpr ::= BExpr OR BExpr
       |  BOOL
```

Now we have solved the problem

```
1 + true
```

Since `+` takes two `IExpr` and `true` is a `BExpr`.

Wrong solution (cont)

- But now we also want to allow variables: "`1 + x`", and "`true or x`"
- We add ID to the grammar.

```
Expr ::= IExpr | BExpr
IExpr ::= IExpr PLUS IExpr | INT | ID
BExpr ::= BExpr OR BExpr | BOOL | ID
```

- But now ID can be either IExpr or BExpr, and the parser has now way of knowing which interpretation is correct.
- Right Solution™: Defer this distinction until the semantic analysis. (Part IV)

Error Recovery

- After detecting an error the parser should be able to continue processing the input.
- This processing is only for finding more errors, not for generating code.

Error Recovery in JavaCUP

- Introduce a special symbol `error`.
- The author of a parser can use error in productions. For instance:

```
Statements ::= ε
             | Statement "." Statements
             | error "." Statements
             ;
```


Error Recovery (cont)

- If the parser encounters an error, it will pop the stack until it gets into a state, where `error` is legal.
- At this point it shifts `error` onto the stack.
- Then, the input tokens are skipped, until the next input token is one that can legally follow the new state.

Error Recovery (cont)

- This scheme is very dependent on a good choice of error productions.
- Assume a production `Statements = error "."`
- If the parser encounters an error inside a statement, it will pop the stack until it expects a `statements`.
- At this point it shifts `error` onto the stack.
- Then, the input tokens are skipped, until "." is found.

Where to put `error`

- It is a good idea to have a terminal after error to ensure termination.
- Examples:

```
CondStatement ::= …
  IF Expr LBRACE error RBRACE
…
Expr ::= …
        | LPAR error RPAR
```


Abstract Syntax

- The parser is responsible for the syntactical analysis of a program, but it should also produce an internal representation of the program for the rest of the compiler to work with.
- A common structure used for the internal representation of the parsed program is as an *Abstract Syntax Tree (AST)*.
- ASTs will be presented next week.

The Assignment

- Assignment 2, Syntactic analysis:
- Use JavaCUP to write a parser for Expressive.
- Start with a parser that only does syntactic analysis. Not generating an AST. The AST generation should be added later. Next week you will get the AST data structure.

Deadline

- A working parser handed in before 2.5.03 will give a ~10% bonus on the exam.
- A full working compiler handed in before 13.6.03 will give another ~20% bonus on the exam.
- A working interpreter handed in before the exam (date to be announced) gives yet another ~10% bonus on the exam.
- You will need at least 50% on the exam to get the grade 3.

Advice

- You will not be able to finish the project by just working during the scheduled lab time! (Unless perhaps if you have written a compiler before.)
- Compared to the rest of the project the scanner is really tiny and simple.
- Do not hesitate to ask me or Nick for help if you get stuck. (Email)
- To get 50% on the exam without any bonus will probably be quite hard. (But very easy with the bonus.)

The parser

- A framework is available on the web page. You will need to complete the file expressive.cup.
- You will need your expressive.lex file from the first assignment. (But not Tokens.java)
- Don't use the grammar from the specification straight off.
- Try using the actions to print information about the parsing in order to see how your parser is working.

Testing

- There will be some test files with both correct and incorrect Expressive programs available on the web page.
- A small interpreter working on the AST will also be provided so that you can test your parser.