444 Compilers — Project #2 — Parser

Project #2: LL Parser

Introduction

The objective of this assignment is to write a backtracking-free LL Predictive Parser. This parser will make use of output from the Lexer constructed in the previous project to obtain the token stream of the program being compiled. (Note, this also allows hand-built lexer-format files to be input by the Parser.)

The Simple LL Predictive Parser consists of three parts: the simple parser machine, the LL parse table for the grammar involved, and the prediction stack. The simple version of the parser machine is a very short piece of code because it is table-driven.

During parsing, the LL Parser creates a Parse Tree (PST) of Node objects. Each Node contains (wraps) the symbol (terminal or non-terminal) it was built for. Each node containing a symbol is built as the RHS of a selected rule is pushed onto the prediction stack. Once the PST is complete, the Parser outputs the tree as text in a simple serialized tree format. Finally, the Parser converts the PST to an AST. This AST is then output using the same serialized tree format.

Team

The team may contain from two to four members. Pick a three-letter name for the team. (If two teams pick the same TLA, I will disambiguate them.)

Optionally, for teams, establish tasks and each day briefly discuss the Agile 3Qs: 1) What was completed yesterday?, 2) What is planned to complete today?, and 3) What obstacles are in the way?. Bread up a task so that some part can be completed each day. If you expect to have down days (i.e., no-work days), tell your team up front.

LL Parse table

The LL parse table is straight-forward to build (by hand) once the grammar has been converted to simple LL-compatible form. This involves several preparatory steps with the grammar:

1. Convert the grammar rules to simple rules because only simple rules go in table cells
2. LRE (Left Recursion Elimination), to avoid LL infinite recursion
3. Disappearing Non-Terminal analysis to add direct epsilon rules for ghostable Non-Ts
4. First Set construction, to see which simple rules go into which LL table cells
5. Fill the LL Table cells
6. If there are cell conflicts, use Left Factoring to avoid more than one simple rule in a cell
7. Follow Set construction, to see which epsilon rules go into which LL table cells
8. ID which RHS symbol will "own" the rest in the AST (or leave the mom Non-T as the owner)

This will typically leave a number of cells unfilled. If the LL Parser machine looks up one of these empty cells, the Parser (step M3E) emits an error and stops. The error should indicate the token (i.e., column header) and the top-of-stack symbol (i.e., the rule LHS) and the token's line number (if known).

Issues

- Match a prediction stack node's symbol against an input stream's token; via token kind.
- Build a node for a symbol; several nodes can be built for the same symbol.
- Mom node should know the rule used to build its kids; and have easy access to them for treewalking
- Table cell should have access to its rule
- Rule should have easy access to its RHS symbols
- Leaf node should know the token it was built for; for 1) variable name and 2) error line # info
- Should be easy to tell if a node's symbol is a terminal or a non-terminal
- Need an "input stream" easy to use (e.g., read each text-token into an array of Token refs)

- How to do Rule #2 Add-a-Trick for a parser, because of the massive bug hunt potential for all of this.

Tree Serialization

Tree Serialization is done via a treewalk in pre-order (mom before kids).

Each node in a tree is serialized as follows (with a space between each part).

1. Output a '(' parenthesis, and "Node:"
2. Output the node's local field information (see below for this)
3. Output each child node, recursively, if any
4. Output a ')' parenthesis – do this using a post-order lollypop

An object's local information should look like the following:

1. Output a '(' parenthesis
2. Output the object's class or data type name (e.g., "Rule:")
3. Output the object's ID number
   Note: Each object should get a unique ID number on creation (eg, counter, or memory location)
   For a given object type, these ID numbers don't have to be sequential, just unique.
4. For each local field, output a tuple of the field name, "=", the field value, and ";"
   If the value is a pointer (or reference), then output that object's class name and unique ID
5. Output a ')' parenthesis

Indentation, line wrapping, etc. are all optional. The easiest solution is to output a newline before every '(' parenthesis.

A6 Grammar

For this project we will use the A6 grammar, specified in the Appendix.

Deliverables

1. In a pdf document, show any new epsilon rules; show each LHS rules before and after for **LRE**; show before and after for any **Left Factored** rules, show the **First Set** for each rule, show the **Follow Set** for each LHS non-terminal symbol, and for each non-trivial rule show the RHS owner node to be hoisted (or indicate the LHS will be kept in place because no RHS seemed appropriate).
2. A6-table: The A6 LL Parser table as a spreadsheet or an importable text CSV file.
3. LL-Parser: The LL Parser machine's source code, which you used to test the parser table on the example A6 programs from the Lexer project.
4. Readme: the README.txt file (e.g., title, contact info, files list, installation/run info, bugs remaining, features added, citations of work referred to).
5. Five output runs of your parser, one for each of the 3 Lexer's sample programs, and two of your own choosing. Two outputs for each: one for the PST and one for the AST. Note, if you like, you can use hand-built input to your Parser, or you can use someone else's Lexer output as input (but cite it).

Academic Rules

Correctly and properly attribute all third party material and references, if you used any.

Submission & Grading

Same as for project #1.

Appendix -- A6 Grammar

Pgm = kwdprog Vargroup Fcndefs Main
Main = kwdmain BBlock
BBlock = brace1 Vargroup Stmts brace2

Vargroup = kwdvars PPvarlist | eps
PPvarlist = parens1 Varlist parens2
Varlist = Varitem semi Varlist | eps
Varitem = Vardecl | Vardecl equal Varinit
Varitem = Classdecl | Classdef
Vardecl = Simplekind Varspec
Simplekind = Basekind | Classid
Basekind = int | float | string
Classid = id
Varspec = Varid | Arrspec | Deref_id
Varid = id
Arrspec = Varid KKint
KKint = bracket1 int bracket2
Deref_id = Deref id
Deref = aster

Varinit = Expr | BBexprs
BBexprs = brace1 Exprlist brace2 | brace1 brace2
Exprlist = Expr Moreexprs
Moreexprs = comma Exprlist | eps

Classdecl = kwdclass Classid
Classdef = Classheader BBclassitems
BBClassitems = brace1 Classitems brace2
Classheader = Classdecl Classmom Interfaces
Classmom = colon Classid | eps
Classitems = Classgroup Classitems | eps
Classgroup = Class_ctrl | Varlist | Mddecls
Class_ctrl = colon id // in {public, protected, private}
Interfaces = colon Classid Interfaces | eps

Mddecls = Mdheader Mddecls | eps
Mdheader = kwdfcn Md_id PParmlist Retkind
Md_id = Classid colon Fcnid

Fcndefs = Fcndef Fcndefs | eps
Fcndef = Fcnheader BBlock

Fcnheader = kwdfcn Fcnid PParmlist Retkind
Fcnid = id
Retkind = Kind
PParmlist = parens1 Varspecs parens2 | PPonly
Varspecs = Varspec More_varspecs
More_varspecs = comma Varspecs | eps
PPonly = parens1 parens2

Stmts = Stmt semi Stmts | eps
Stmt = Stasgn | Fcall | Stif
Stmt = Stwhile | Stprint | Strtn

Stasgn = Lval equal Expr
Lval = Varid | Aref | Deref_id
Aref = Varid KKexpr
KKexpr = bracket1 Expr bracket2

Fcall = Fcnid PPexprs
PPexprs = parens1 Exprlist parens2 | PPonly

Stif = kwdif PPexpr BBlock Elsepart
Elsepart = kwdelseif PPexpr BBlock Elsepart
Elsepart = kwdelse BBlock | eps

Stwhile = kwdwhile PPexpr BBlock
Stprint = kprint PPexprs

Strtn = kwdreturn Expr | kwdreturn

PPexpr = parens1 Expr parens2
Expr = Expr Oprel Rterm | Rterm
Rterm = Rterm Opadd Term | Term
Term = Term Opmul Fact | Fact
Fact = Basekind | Lval | Addrof_id | Fcall | PPexpr
Addrof_id = ampersand id
Oprel = opeq | opne | Lthan | ople | opge | Gthan
Lthan = angle1
Gthan = angle2
Opadd = plus | minus
Opmul = aster | slash | caret