Building Compilers with Phoenix

Parser Generators: ANTLR

© 2008 Martin v. Löwis

History of ANTLR

- ANother Tool for Language Recognition
- Terence Parr's dissertation: Obtaining Practical Variants of LL(k) and LR(k) for k > 1
  - PCCTS: Purdue Compiler Construction Set
    - Master's Thesis: YUCC (1988), LL(1)"
  - written in C++
  - a.k.a. ANTLR v1
  - "linear approximate LL(k)"
- ANTLR v2: Rewrite in Java (1997)
  - can generate parsers in many other languages
- ANTLR v3: Rewrite (2007)
  - new parser algorithm: LL(*)
  - auto-backtracking for conflicting/ambiguous productions
  - ...

© 2008 Martin v. Löwis

LR Parsing

- **L**eft-to-right, preferring **R**ight-most derivations
- LR(k): Parser needs k tokens look-ahead
- Language L is said to be LR(k) if LR(k) parser exists
  - Many programming languages are LR(1)
- typically bottom-up parsing:
  - parser is a state machine, with an additional stack of nested non-terminals and terminals
  - as input is encountered, pieces are combined into non-terminals
- typically table-driven:
  - based on look-ahead and state: decide to shift/reduce/accept
- Many variants
  - LALR (look-ahead LR): yacc
  - SLR (simple LR)
  - GLR (generalized LR)

© 2008 Martin v. Löwis

LL parsing

- **L**eft-to-right, preferring **L**eft-most derivations
- LL(k): needs k tokens lookahead
- Programming languages are often not LL(k) for any k
- typically implemented as top-down parsers
- parser algorithm:
  - construct parser table up-front
    - for any non-terminal N and combination of k look-ahead tokens $T_1..T_k$, select production P
  - maintain stack S of non-terminals
  - Given look-ahead tokens and top of stack, select production
    - for non-terminal N, push N, then recurse
    - for terminal T, consume T
    - when production is complete, pop N

© 2008 Martin v. Löwis

LL(k)

- Sometimes, two or more tokens look-ahead necessary

```
decl → type ID ';'
     |  type ID '=' value ';'
type → 'int' | 'long' | 'float' | 'double'
```

- Problem: for alphabet A, $(\#A)^N$ different lookahead sequences are possible
  - algorithm considered impractical
  - problem with left-recursion remains
- Linear approximated LL(k):
  - Instead of checking all $(\#A)^N$ combinations, parser forms equivalence classes as sequence of token sets:
    - {'int','long','float','double'} {ID} {';'}
  - cannot parse all LL(k) grammars
  - experience: if "real-language" grammar is LL(k), l-a LL(k) can parse it

© 2008 Martin v. Löwis

LL(*)

- Sometimes, cannot parse with LL(k) for any fixed k

  method → type ID '(' args ')' ';'
  | type ID '(' args ')' '{' statements '}'
  args → args (',' arg)*
  args → type ID

- Solution: express prefixes of alternative as regular language
  - generate DFA to determine alternative
- Limitation: prefix language must not use recursive productions
  - maximum recursion depth configurable (-Xm), default 4

© 2008 Martin v. Löwis

Conflict resolution in ANTLR

- Syntactic predicates: prefix alternative with "try" production
  - alternatives tested in textual order, first matching alternative used
  - e.g. Java for vs. foreach

```
forControl : (forVarControl) => forVarControl
                    | forInit? ';' expression? ';' forUpdate?;
forVarControl: 'final'? annotation? type Identifier ':' expression;
```

- Semantic predicates: prefix alternative with boolean predicate (in parser's target language)
- Backtracking

© 2008 Martin v. Löwis

Semantic Predicates

- Example: classify identifiers in C

  declaration : decl_specs declarator ';' ;
  decl_specs : (storage_class | type_qual | type_spec)+ ;
  type_spec : 'void' | 'int' | ... | type_id;
  type_id : ID;
  declarator : ID | '*' declarator ;

  – Conflict: In "static const Foo", is Foo type_id or declarator?
  – Solution:
    - type_id: {isTypeName(input.LT(1).getText())}? ID ;
- Other application: optional language features

  statement : for_stmt | while_stmt
  | expr_stmt | ... | {have_aop()}? proceed_stmt

© 2008 Martin v. Löwis

Backtracking

- New in ANTLR v3
- option backtrack: per grammar or per non-terminal
- implicit syntactic predicates on all alternatives
  - optimization: omit predictive parsing if LL(*) already disambiguates the grammar
- memoization: avoid re-parsing the same pieces over and over again due to backtracking
  - option memoize

© 2008 Martin v. Löwis

ANTLR Parsers

- grammar files (.g): grammar name, global options, productions
- EBNF: alternatives (|), repeating (?, *, +)
- various annotations to describe semantic actions
  - return type declaration
  - tree construction (!, ^)
  - arbitrary semantic actions ({...})
- common options:
  - language (Java, C, C++, C#, Objective-C, Python, Ruby)
  - output (AST, template)
  - k (ANTLR v2)

© 2008 Martin v. Löwis

ANTLR Lexers

- Lexical rules: rule name starts with capital letter
- additional constructs for character ranges ('a'..'z')
- common option: greedy (default: true)
- token attributes: text, type, line, pos, ...
- channel: divide tokens into different channels
  - whitespace: set channel to Token.HIDDEN_CHANNEL

© 2008 Martin v. Löwis

Semantic Actions

- Return values and rule arguments
  - r [int a, String b] returns [int c, String d] :
  - ... { $c = $a; $d = $b }
- label parts of a production with symbolic names
  - expr : left=expr '+' right=term;
- Arbitrary semantic actions
- Creation of abstract syntax tree (output=AST)
  - default: flat list of all tokens
  - mark terminals with ^ to indicate they should be parent node of the production
  - mark terminals with ! to omit node from tree
  - classes used for nodes can be set with ASTLabelType option

© 2008 Martin v. Löwis

Generating with ANTLR

- generating without ANTLR: just process AST
- tree rewriting: write "tree parser" to convert one tree into another
- templating: StringTemplate library

© 2008 Martin v. Löwis