Syntactic Analysis

Syntactic analysis, or parsing, is the second phase of compilation: The token file is converted to an abstract syntax tree.

Compiler Passes

Analysis of input program (front-end)
- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis

Synthesis of output program (back-end)
- Code Generation
- Optimization
- Intermediate Code Generation

Syntactic Analysis / Parsing

- Goal: Convert token stream to abstract syntax tree
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for remainder of analysis
- Three Part Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing / building ASTs
  - Study the miniJava Implementation

Context-free Grammars

- Compromise between
  - REs, which can’t nest or specify recursive structure
  - General grammars, too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse; but also allow restrictions for speed
- Not perfect
  - Cannot capture semantics, as in, “variable must be declared,” requiring later semantic pass
  - Can be ambiguous
- EBNF, Extended Backus Naur Form, is popular notation

CFG Terminology

- Terminals -- alphabet of language defined by CFG
- Nonterminals -- symbols defined in terms of terminals and nonterminals
- Productions -- rules for how a nonterminal (lhs) is defined in terms of a (possibly empty) sequence of terminals and nonterminals
  - Recursion is allowed!
- Multiple productions allowed for a nonterminal, alternatives
- Start symbol -- root of the defining language

EBNF Syntax of initial MiniJava

Program ::= MainClassDecl | ClassDecl
MainClassDecl ::= class ID 
  public static void main 
  ( String [] ID ) 
  { Stmt }
ClassDecl ::= class ID [ extends ID ] 
  { ClassVarDecl } { MethodDecl }
ClassVarDecl ::= Type ID 
MethodDecl ::= public Type ID 
  { [ Formal , Formal ] 
  } 
  { Stmt | return Expr ; }
Formal ::= Type ID
Type ::= int | boolean | ID
Initial miniJava [continued]

Stmt ::= Type ID ;
    | { Stmt } ;
    | if ( Expr ) Stmt else Stmt
    | System.out.println ( Expr ) ;
    | ID = Expr ;
Expr ::= Expr Op Expr
    | ! Expr
    | Expr . ID ([ Expr , Expr ])
    | ID | this
    | Integer | true | false
    | ( Expr )
Op ::= * | + | / | < | <= | >= | > | == | != | && | !

Derivations and Parse Trees

Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals

Parsing: inverse of derivation
– Given a sequence of terminals (a k a tokens) want to recover the nonterminals representing structure

Can represent derivation as a parse tree, that is, the concrete syntax tree

Ambiguity

• Some grammars are ambiguous
  – Multiple distinct parse trees for the same terminal string

• Structure of the parse tree captures much of the meaning of the program
  – Ambiguity implies multiple possible meanings for the same program
Resolving Ambiguity

- Option 1: add a meta-rule
  - For example “else associates with closest previous if”
  - Works, keeps original grammar intact
  - Ad hoc and informal

Resolving Ambiguity [continued]

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... | if (Expr) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if (Expr) Stmt | if (Expr) MatchedStmt else UnmatchedStmt
```

- Formal, no additional rules beyond syntax
- Sometimes obscures original grammar

Resolving Ambiguity Example

```
Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... | if (Expr) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if (Expr) Stmt | if (Expr) MatchedStmt else UnmatchedStmt
```

```
if (e₁) if (e₂) s₁ else s₂
```

Resolving Ambiguity [continued]

Option 3: redesign the language to remove the ambiguity

```
Stmt ::= ... | if Expr then Stmt end | if Expr then MatchedStmt else Stmt end
```

- Formal, clear, elegant
- Allows sequence of Stmts in then and else branches, no {,} needed
- Extra end required for every if

Another Famous Example

```
E ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /
```

```
a + b * c : a + b * c
```

Resolving Ambiguity (Option 1)

Add some meta-rules, e.g. precedence and associativity rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Preced</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postfix ++</td>
<td>Highest</td>
<td>Left</td>
</tr>
<tr>
<td>Prefix -</td>
<td>Right</td>
<td></td>
</tr>
<tr>
<td>** (Exp)</td>
<td>Right</td>
<td></td>
</tr>
<tr>
<td>* /, %</td>
<td>Left</td>
<td></td>
</tr>
<tr>
<td>** ==</td>
<td>&lt;</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example:
```
E ::= E Op E | - E | E ++
     | ( E ) | id
Op ::= + | - | * | / |
     | ** | == | < | && |
     | ||
```


Removing Ambiguity (Option 2)

Option 2: Modify the grammar to explicitly resolve the ambiguity

Strategy:
• create a non-terminal for each precedence level
• expr is lowest precedence non-terminal, each non-terminal can be rewritten with higher precedence operator, highest precedence operator includes atomic exprs
• at each precedence level, use:
  – left recursion for left-associative operators
  – right recursion for right-associative operators
  – no recursion for non-associative operators

Redone Example

Operators precedence Example

Designing a Grammar

Concerns:
– Accuracy
– Unambiguity
– Formality
– Readability, Clarity
– Ability to be parsed by a particular algorithm:
  • Top down parser ==> LL(k) Grammar
  • Bottom up Parser ==> LR(k) Grammar
– Ability to be implemented using particular approach
  • By hand
  • By automatic tools

Parsing Algorithms

Given a grammar, want to parse the input programs
– Check legality
– Produce AST representing the structure
– Be efficient
• Kinds of parsing algorithms
  – Top down (LL(1), Recursive Descent)
  – Bottom up (LR(1), Operator Precedence)

Top Down Parsing

Build parse tree from the top (start symbol) down to leaves (terminals)
• Pick a production & try to match the input
• Bad “pick” ⇒ may need to backtrack
• Some grammars are backtrack-free (predictive parsing)

Basic issue: when “expanding” a non-terminal with some r.h.s., how to pick which r.h.s.?

E.g.

Solution: look at input tokens to help decide
Predictive Parser
Predictive parser: top-down parser that can select rhs by looking at most k input tokens (the lookahead)
Efficient:
- no backtracking needed
- linear time to parse
Implementation of predictive parsers:
- recursive-descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub nonterminals, recursively
  - typically written by hand
- table-driven parser
  - PDA like table-driven FSA, plus stack to do recursive FSA calls
  - typically generated by a tool from a grammar specification

LL(k) Grammars
Can construct predictive parser automatically / easily if grammar is LL(k)
- Left-to-right scan of input, Leftmost derivation (replace leftmost NT at each step)
- k tokens of look ahead needed, ≥ 1
Some restrictions:
- no ambiguity (true for any parsing algorithm)
- no common prefixes of length ≥ k:
  - If ::= if Test then Stmts end |
    if Test then Stmts else Stmts end
- no left recursion:
  - E ::= E Op E |
  - a few others (First() and Follow() rules – see text.)
Restrictions guarantee that, given k input tokens, can always select correct rhs to expand nonterminal. Easy to do by hand in recursive-descent parser

Eliminating common prefixes
Can left factor common prefixes to eliminate them
- create new nonterminal for different suffixes
- delay choice till after common prefix
- Before:
  - If ::= if Test then Stmts end |
    if Test then Stmts else Stmts end
- After:
  - If ::= if Test then Stmts IfCont
  - IfCont ::= end | else Stmts end

Eliminating Left Recursion
- Can Rewrite the grammar to eliminate left recursion
- Before
  - E ::= E + T | T
  - T ::= T * F | F
  - F ::= id | ...
- After
  - E ::= T ECon
  - ECon ::= + T ECon | ε
  - T ::= F TCon
  - TCon ::= * F TCon | ε
  - F ::= id | ...

Building Top-down Parsers
Given an LL(1) grammar and its FIRST & FOLLOW sets
- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs’s
  - Each returns true on success and throws an error on false
  - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser
Improving matters
- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Recursive Descent Parsing Example
A couple of routines from the expression parser
- Parse()
  - token ← next_token();
  - if (Expr()) = true & token = EOF
    then next compilation step;
  - else report syntax error;
  - return false;
- Factor()
  - if (token = Number) then
    token ← next_token();
  - else if (token = Identifier) then
    return true;
  - else report syntax error;
  - return false;
- Expr()
  - if (Term()) = false
    then return false;
  - else return ECon();;
- ECon, Term, and TCon are constructed in a similar manner.
Building Top-down Parsers

Strategy
• Encode knowledge in a table
• Need a row for every NT and a column for every T
• Use a standard "skeleton" parser to interpret the table

Bottom Up Parsing

Construct parse tree for input from leaves up
– reducing a string of tokens to single start symbol (inverse of deriving a string of tokens from start symbol)

“Shift-reduce” strategy:
– read ("shift") tokens until seen r.h.s. of "correct" production
– reduce handle to l.h.s. nonterminal, then continue
– done when all input read and reduced to start nonterminal

LR(k)

• LR(k) parsing
  – Left-to-right scan of input, Rightmost derivation
  – k tokens of look ahead
• Strictly more general than LL(k)
  – Gets to look at whole rhs of production before deciding what to do, not just first k tokens of rhs
  – can handle left recursion and common prefixes fine
• Still as efficient as any top-down or bottom-up parsing method
• Complex to implement
  – need automatic tools to construct parser from grammar

LR Parsing Tables

Construct parsing tables implementing a FSA with a stack
• rows: states of parser
• columns: token(s) of lookahead
• entries: action of parser
  • shift, goto state
  • reduce production "X ::= RHS"
  • accept
  • error

Algorithm to construct FSA similar to algorithm to build DFA from NFA
• each state represents set of possible places in parsing
LR(k) algorithm builds huge tables

LALR-Look Ahead LR

LALR(k) algorithm has fewer states ==> smaller tables
– less general than LR(k), but still good in practice
– size of tables acceptable in practice
• k = 1 in practice
  – most parser generators, including yacc and CUP, are LALR(1)

Global Plan for LR(0) Parsing

• Goal: Set up the tables for parsing an LR(0) grammar
  – Add S$ : = S$ to the grammar, (i.e. We will be solving the problem for a new grammar with a terminator)
  – Compute parser states by starting with state 1 containing added production, S$ : = .S$
  – Form closures of states and shifting to complete diagram
  – Convert diagram to transition table for PDA
  – Step through parse using table and stack
LR(0) Parser Generation

- **Key idea**: simulate where input might be in grammar as it reads tokens
- “Where input might be in grammar” captured by **set of items**, which forms a state in the parser’s FSA
  - LR(0) item: \( \text{lhs} ::= \text{rhs} \), production, with a dot in rhs somewhere marking what’s been read (shifted) so far.
  - Example:
    - Initial item: \( S' ::= . \ S \$ \)
    - (LR(k) item: also add \( k \) tokens of lookahead to each item)

Closure

The initial state in the FSA is the **closure** of initial item.

The initial state in the FSA is the **closure** of initial item.

**Closure of an item**:

If the dot is before non-terminal, then:
1. Add all productions for that non-terminal, and
2. Put a dot at the start of the RHS of each production.

Initial item (1):
\[
S' ::= . \ S \$
\]
Initial state (1):
\[
S' ::= . \ S \$
S ::= . \ S
S ::= . \ \{ \ L \}
\]

Example

Grammar:
\[
S' ::= S \$
S ::= \text{beep} \mid \{ \ L \}
L ::= S \mid L ; S
\]

State (1):
\[
S' ::= S \$
S ::= \text{beep}
S ::= \{ \ L \}
\]

State (2) (reached on transition that shifts S):
\[
S' ::= S \$
\]

State (3) (reached on transition that shifts beep):
\[
S ::= \text{beep}
\]

State (4) (reached on transition that shifts \{ \}):
\[
S ::= \{ \ L \}
L ::= S
L ::= L ; S
S ::= \text{beep}
S ::= \{ \ L \}
\]

State Transitions (Shifting)

Given a set of items, compute new state(s) for each symbol (terminal and non-terminal) after dot
- state transitions correspond to shift actions
A new item is derived from an old item by shifting the dot over the symbol
- then do closure on this item to compute new state

Accepting & Reducing

Other than shifting symbols there are two other actions we might take:
- accepting:
  - at the end of a successful parse
- reducing:
  - applying a production to symbols on our stack that match the RHS of the production.
Accepting Transitions

If a state has an item with the dot before the $, e.g.:
S' ::= S . $
then we will add a transition from this state labeled $ that goes to the accept action (in the transition table).

For example, State (2):
S' ::= S . $
has a transition labeled $ to the accept action.

Reducing States

If state has an item with a dot at the end, e.g.:
lhs ::= rhs .
then it has a reduce lhs ::= rhs action.

For example, state (3):
S ::= beep .
has a reduce S ::= beep action
We will add this in our transition table as the action to take when in this state regardless of the next symbol.

Hmm.....Conflicting Actions?
– what if other items in this state shift?
– what if other items in this state reduce differently?

Example

Grammar:
S' ::= $ S
S ::= beep | ( L )
L ::= S | L ; S

S' ::= S . $
S ::= . beep
L ::= . beep
S ::= . ( L )

Rest of the States, Part 1

State (4): on beep, shift and goto State (3)
State (4): on (, shift and goto State (4)
State (4): on S, shift and goto State (5)
State (4): on L, shift and goto State (6)
State (5):
L ::= S .
reduce L ::= S
State (6):
L ::= L .
S ::= . beep
reduce S ::= S

Rest of the States (Part 2)

State (7):
L ::= ( L ) .
reduce S ::= ( L )
State (8):
L ::= L ; S
S ::= . beep
L ::= ( L )
State (8): on beep, shift and goto State (3)
State (8): on (, shift and goto State (4)
State (8): on S, shift and goto State (9)
State (9):
L ::= L ; S .
reduce L ::= L ; S

LR(0) State Diagram
Building Table of States & Transitions

Create a row for each state
Create a column for each terminal, non-terminal, and S
For every "state (i): if shift X goto state (j)" transition:
  • if X is a terminal, put "shift, goto j" action in row i, column X
  • if X is a non-terminal, put "goto j" action in row i, column X
For every "state (i): if $ accept" transition:
  • put "accept" action in row i, column $
For every "state (i): lhs ::= rhs." action:
  • put "reduce lhs ::= rhs" action in all columns of row i

Table of This Grammar

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>S</th>
<th>L</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s.g4</td>
<td>s.g3</td>
<td>g2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s.g4</td>
<td>s.g3</td>
<td>g5</td>
<td>g6</td>
</tr>
<tr>
<td>4</td>
<td>s.g4</td>
<td>s.g3</td>
<td>g7</td>
<td>g8</td>
</tr>
<tr>
<td>5</td>
<td>reduce L ::= S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s.g7</td>
<td>s.g8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>reduce S ::= { L }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s.g4</td>
<td>s.g3</td>
<td>g9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>reduce L ::= L ; S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution of Parsing Table

• Parser State:
  – stack of:
    • states, (initialized to state "1") and
    • shifted/reduced symbols, (initially empty)
  – unconsumed tokens, (initialized to input tokens)
• To run the parser, repeat these steps:
  – Do action(S, x) where S is the state on top of stack, and x is the next unconsumed token.
  – If the action was a goto(S), push state S onto the stack
  – If action (S, x) is empty, report syntax error

Actions

shift: push the next unconsumed token onto the stack
goto: push this state on the stack
reduce: LHS ::= RHS
  – Pop pairs of symbols and states from top of stack equal to the number of symbols in RHS
  – See what state I have uncovered (= uncovered_state)
  – Push LHS onto the stack
  – Push the state: action (uncovered_state, LHS) onto stack
  – (Would also build parse tree for LHS from RHS subtrees at this time.)
accept: done parsing, return parse tree

Example

<table>
<thead>
<tr>
<th>State</th>
<th>L</th>
<th>S</th>
<th>L</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>beep</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>accept</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems In Shift-Reduce Parsing

Can write grammars that cannot be handled with shift-reduce parsing

Shift/reduce conflict:
  • state has both shift action(s) and reduce actions
Reduce/reduce conflict:
  • state has more than one reduce action
Shift/Reduce Conflicts

LR(0) example:
\[ E ::= E + T | T \]
State:
\[ E ::= E . + T \]
\[ E ::= T . \]
- Can shift +
- Can reduce \( E ::= T \)

LR(k) example:
\[ S ::= \text{if } E \text{ then } S | \]
\[ \text{if } E \text{ then } S \text{ else } S | \ldots \]
State:
\[ E ::= \text{if } E \text{ then } S . \]
\[ \text{else } E ::= \text{if } E \text{ then } S . \text{ else } S \]
- Can shift else
- Can reduce \( S ::= \text{if } E \text{ then } S \)

Avoiding Shift-Reduce Conflicts

Can rewrite grammar to remove conflict
- E.g. Matched Stmt vs. Unmatched Stmt
Can resolve in favor of shift action
- Try to find longest r.h.s. before reducing
  works well in practice
  yacc, jflex, et al. do this

Reduce/Reduce Conflicts

Example:
\[ \text{Stmt ::= Type } \text{id } | \text{ LHS = Expr } | \ldots \]
\[ \ldots \]
\[ \text{LHS ::= id } | \text{ LHS [ Expr ] } | \ldots \]
\[ \ldots \]
\[ \text{Type ::= id } | \text{ Type [ ] } | \ldots \]
State:
\[ 
\begin{align*}
\text{Type ::= id .} \\
\text{LHS ::= id .}
\end{align*}
\]
- Can reduce Type ::= id
- Can reduce LHS ::= id

Avoid Reduce/Reduce Conflicts

Can rewrite grammar to remove conflict
- Can be hard
  - E.g. C/C++ declaration vs. expression problem
  - E.g. MiniJava array declaration vs. array store problem
Can resolve in favor of one of the reduce actions
  - But which?
  - yacc, CUP, et al. Pick reduce action for production
    listed textually first in specification

Abstract Syntax Trees

The parser’s output is an abstract syntax tree (AST) representing the grammatical structure of the parsed input
- ASTs represent only semantically meaningful aspects of input program, unlike concrete syntax trees which record the complete textual form of the input
  - There’s no need to record keywords or punctuation like (), ;, else
  - The rest of compiler only cares about the abstract structure

AST Node Classes

Each node in an AST is an instance of an AST class
- IfStmt, AssignStmt, AddExpr, VarDecl, etc.
Each AST class declares its own instance variables holding its AST subtrees
- IfStmt has testExpr, thenStmt, and elseStmt
- AssignStmt has lhsVar and rhsExpr
- AddExpr has arg1Expr and arg2Expr
- VarDecl has typeExpr and varName
AST Class Hierarchy

AST classes are organized into an inheritance hierarchy based on commonalities of meaning and structure.

- Each "abstract non-terminal" that has multiple alternative concrete forms will have an abstract class that’s the superclass of the various alternative forms.
  - Stmt is abstract superclass of IfStmt, AssignStmt, etc.
  - Expr is abstract superclass of AddExpr, VarExpr, etc.
  - Type is abstract superclass of IntType, ClassType, etc.

AST Extensions For Project

New variable declarations:
- StaticVarDecl

New types:
- DoubleType
- ArrayType

New/changed statements:
- IfStmt can omit else branch
- ForStmt
- BreakStmt
- ArrayAssignStmt

New expressions:
- DoubleLiteralExpr
- OrExpr
- ArrayLookupExpr
- ArrayLengthExpr
- ArrayNewExpr

Automatic Parser Generation in MiniJava

We use the CUP tool to automatically create a parser from a specification file, Parser/minijava.cup.

The MiniJava Makefile automatically rebuilds the parser whenever its specification file changes.

A CUP file has several sections:
- introductory declarations included with the generated parser
- declarations of the terminals and nonterminals with their types
- The AST node or other value returned when finished parsing that nonterminal or terminal
- precedence declarations
- productions + actions

Terminal and Nonterminal Declarations

Terminal declarations we saw before:
/* reserved words: */
terminal CLASS, PUBLIC, STATIC, EXTENDS;...
/* tokens with values: */
terminal String IDENTIFIER;
terminal Integer INT_LITERAL;

Nonterminals are similar:
nonterminal Program Program;
nonterminal MainClassDecl MainClassDecl;
nonterminal List/*<...>*/ ClassDecls;
...nonterminal List/*<Stmt>*/ Stmts;
nonterminal Stmt Stmt;
nonterminal List/*<Expr>*/ Exprs;
nonterminal List/*<Expr>*/ MoreExprs;
nonterminal Expr Expr;
...
nonterminal String Identifier;

Precedence Declarations

Can specify precedence and associativity of operators
- equal precedence in a single declaration
- lowest precedence textually first
- specify left, right, or nonassoc with each declaration

Examples:
preceision left AND_AND;
preceision nonassosc EQUALS_EQUALS,
EXCLAIM_EQUALS;
preadence left LESS, LESS_EQUAL,
GREATER_EQUAL, GREATER_THAN;
preadence left PLUS, MINUS;
preadence left STAR, SLASH;
preadence left EXCLAIM;
preadence left PERIOD;

Productions

All of the form:

LHS ::= RHS1 {: Java code 1 :}
| RHS2 {: Java code 2 :}
| ...
| RHSn {: Java code n :};

Can label symbols in RHS with: var suffix to refer to its result value in Java code

E.g.: Expr ::= Expr:arg1 PLUS Expr:arg2

| RESULT = new AddExpr( arg1, arg2, arg1left); |
| INT_LITERAL(value.intValue(), valueleft); |
| Expr:rcvr PERIOD Identifier:message OPEN_PAREN
Expr:args CLOSE_PAREN

| RESULT = new MethodCallExpr( rcvr, message, args, rcvrleft); |
| ... ; |
Error Handling

How to handle syntax error?
Option 1: quit compilation
  + easy
  - inconvenient for programmer
Option 2: error recovery
  + try to catch as many errors as possible on one compile
  - difficult to avoid streams of spurious errors
Option 3: error correction
  + fix syntax errors as part of compilation
  - hard!!

Panic Mode Error Recovery

When finding a syntax error, skip tokens until reaching a "landmark"
  - landmarks in MiniJava: ;, }, }
  - once a landmark is found, hope to have gotten back on track
In top-down parser, maintain set of landmark tokens as recursive descent proceeds
  - landmarks selected from terminals later in production
  - as parsing proceeds, set of landmarks will change, depending on the parsing context
In bottom-up parser, can add special error nonterminals, followed by landmarks
  - if syntax error, then will skip tokens till seeing landmark, then reduce and continue normally
  - E.g.Stmt ::= ... | error ; | { error |
     Expr ::= ... | ( error )

67
68