Scanners and Parsers

Summary

- What is a programming language
- What makes a language successful
- Why study programming languages
- Programming language views
- Types of languages
- Phases of compilation

Types of languages

- Imperative
  - Von Neumann (Fortran, Pascal, BASIC, C, …)
  - Object-oriented (C++, Java, Smalltalk, …)
- Declarative
  - Functional (Scheme, Lisp, ML, …)
  - Logic, constraint-based (Prolog, VisiCalc, …)

Phases of Compilation

- Scanning (lexical analysis)
- Parsing (syntax analysis)
- Semantic analysis
- Intermediate code generation
- Machine-independent code improvement
- Machine-specific code improvement
- Target code generation

Overview

specifying Patterns

A scanner must recognize various parts of the language’s syntax
Some parts are easy – e.g., white space, keywords, operators, comments
Other parts require more notation – e.g., identifiers, numbers
→ Need a powerful notation to specify these patterns

A scanner must recognize various parts of the language’s syntax
Some parts are easy – e.g., white space, keywords, operators, comments
Other parts require more notation – e.g., identifiers, numbers
→ Need a powerful notation to specify these patterns
Regular Expressions

- Defines a set of strings over the characters contained in some alphabet, defining a language
- Three basic operations
  - Union – e.g., a|b
  - Concatenation – e.g., ab
  - Closure – (Kleene closure) – e.g., a* - where a can be a set concatenated with itself zero or more times

Finite Automata

- A finite collection of states
- A set of transitions between those states
- An alphabet
- A start state, S0
- One or more final states

Deterministic vs. Non-deterministic – single-valued vs. multi-valued transitions, no epsilon transitions

RE <-> DFA

Recognizers

Identifier
- letter → (a|b|c | ... |z|A|B|C | ... |Z)
- digit → (0|1|2|3|4|5|6|7|8|9)
- id → letter(letter|digit)*

Automated RE->NFA

- Build NFA for each term, connect them with ε moves
  - Concatenate - ab
  - Union – a|b
  - Kleene Closure – a*

Thompson’s Construction

- Each NFA has a single start state and a single final state
- An ε-move always connects two states that were start or final states
- A state has at most 2 entering and 2 exiting ε-moves, and at most 1 entering and 1 exiting move on a symbol in the alphabet
Automated DFA->RE

for i = 1 to N
  for j = 1 to N
    $R_{ij}^0 = (a(s_i, a) = s_j)$
    if (i == j)
      $R_{ij}^0 = R_{ij}^0 \cup \epsilon$
  for k = 1 to N
    for i = 1 to N
      for j = 1 to N
        $R_{ij}^k = R_{ik}^{k-1} R_{kk}^{k-1} R_{kj}^{k-1} \cup \epsilon$

$L = U R_{ij}^N$

$\epsilon S_F$

NFA->DFA

- Subset construction algorithm
  - Each state in DFA corresponds to a set of states in NFA
  - $q_0 \leftarrow \text{closure}(n_0)$
  - Initialize Q with $\{q_0\}$
  - While Q is still changing
    - For each $q \in Q$
      - For each character $a \in \Sigma$
        - $T[q, a] \leftarrow \text{closure}(\text{move}(q, a))$
        - $T[q, a] \leftarrow t$
        - If $t \notin Q$ then
          - Add t to Q

DFA Minimization

$P \leftarrow P \{S_F, (S - S_F)\}$
While (P is still changing)
  $T \leftarrow 0$
  For each set $p \in P$
    For each $\alpha \in \Sigma$
      Partition p by $\alpha$
      into $p_1, p_2, p_3, ..., p_k$
      $T \leftarrow T \cup p_1, p_2, p_3, ..., p_k$
    If $T \neq P$ then
      $P \leftarrow T$

Implementing Scanners

- Ad-hoc
- Semi-mechanical pure DFA
- Table-driven DFA

Code for Recognizer/Scanner

```java
char char_class = read_char();
delay = S0, "code for 50";
token_value = "" /* empty string */
while (not done)
  class = char_class(next_char());
  switch (class)
  {
    case S1: /* building an id */
      token_value = token_value + char;
      char = read_char();
      break;
    case S2: /* accept state */
      token_value = token_value;
      done = true;
      break;
    case S3: /* error */
      token_type = error;
      done = true;
      break;
  }
return(token_type);
```

Tables Driving the Recognizer

<table>
<thead>
<tr>
<th>char_class</th>
<th>a-z</th>
<th>A-Z</th>
<th>0-9</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next_state</th>
<th>class</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>S1</td>
<td>S1</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>digit</td>
<td>S3</td>
<td>S1</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>other</td>
<td>S3</td>
<td>S2</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

To change language, we can just change tables
Error Recovery

- E.g., illegal character
- What should the scanner do?
  - Report the error
  - Try to correct it?
- Error correction techniques
  - Minimum distance corrections
  - Hard token recovery
  - Skip until match

Scanner Summary

- Break up input into tokens
- Catch lexical errors
- Difficulty affected by language design
- Issues
  - Input buffering
  - Lookahead
  - Error recovery
- Scanner generators
  - Tokens specified by regular expressions
  - Regular expression → DFA
  - Highly efficient in practice

Overview

- Front end
  - Scanner
  - Parser

- Back End

- Source code
- Intermediate language
- Machine code

- Front end
  - Source code
  - Scanner
  - Parser

- Back End

- Intermediate language
- Tokens
- Parser

- Machine code

- Errors

Parser

- Check input for grammatical correctness
- Determine syntax of token stream
- Constructs intermediate representation
- Produces meaningful error messages

Needs mathematical model of syntax → grammar

Needs algorithm for testing syntax → parsing

Context Free Grammars

- Adds recursion/allows non-terminals to be expressed in terms of themselves
- Can be used to count/impart structure – e.g., nested parentheses
- Notation – grammar G(S,N,T,P)
  - S is the start symbol
  - N is a set of non-terminal symbols (LHS)
  - T is a set of terminal symbols (tokens)
  - P is a set of productions or rewrite rules
    \[ P : N \rightarrow N U T \]

Advantages of CFGs

- Precise syntactic specification of a programming language
- Easy to understand, avoids ad hoc definition
- Easier to maintain, add new language features
- Can automatically construct efficient parser
- Parser construction reveals ambiguity, other difficulties
- Imparts structure to language
- Supports syntax-directed translation
Derivations

- A sequence of application of the rewrite rules is a derivation or a parse (e.g., deriving the string $x + 2 - y$)
- The process of discovering a derivation is called parsing

Types of derivations

- Leftmost derivation
  - The leftmost non-terminal is replaced at each step
- Rightmost derivation
  - The rightmost non-terminal is replaced at each step
- Ambiguous grammar — one with multiple leftmost (or multiple rightmost) derivations for a single sentential form

Types of parsers

- Top-down (LL) parsers
  - Left to right, leftmost derivation
  - Starts at the root of the derivation tree and fills in
  - Predicts next state with n lookahead
- Bottom-up (LR) parsers
  - Left to right, rightmost derivation
  - Starts at the leaves and fills in
  - Start with input string, end with start symbol
  - Starts in a state valid for legal first tokens
  - Change state to encode possibilities as input is consumed
  - Use a stack to store both state and sentential form

Top-Down Parsing

- At a node labeled A, select a production with A on its LHS and for each symbol on its RHS, construct the appropriate child
- When a terminal is added that does not match the input, backtrack
- Find the next node to be expanded (must have a label in NT)

Calculator Example

- All variables are integers
- There are no declarations
- The only statements are assignments, input, and output
- Expressions use one of four arithmetic operators and parentheses
- Operators are left associative, with the usual precedence
- There are no unary operators

Regular Expressions

id → letter (letter | digit)*
literal → digit digit*
read, write, “=”, “+”, “-”, “*”, “/”, “(“,”)”
$$ /* end of input */$
Grammar for Calculator

<pgm> → <stmtlist> $$
<stmtlist> → <stmtlist> <stmt> | ε
<stmt> → id := <expr> | read <id> | write <expr>
<expr> → <term> | <expr> <addop> <term>
<term> → <factor> | <term> <multop> <factor>
<factor> → ( <expr> ) | id | literal
<addop> → + | -
<multop> → * | /

Eliminating Left Recursion

• A grammar is left recursive if there exists A in NT such that A → Aδ for some string δ
• Transform the grammar to remove left recursion

<foo> → <foo> δ
| µ
<foo> → µ <bar>
<bar> → δ <bar> | ε
where <bar> is a new non-terminal

Eliminating Common Prefixes

foo → bar δ
| bar ( µ )
→ foo → bar footail
footail → δ | ( µ )

LL Grammar for Calculator

<pgm> → <stmtlist> $$
<stmtlist> → <stmtlist> <stmt> | ε
<stmt> → id := <expr> | read <id> | write <expr>
<expr> → <term> <termtail>
<termtail> → <addop> <term> <termtail> | ε
<term> → <factor> <factortail>
<factortail> → <multop> <factor> <factortail> | ε
<factor> → ( <expr> ) | id | literal
<addop> → + | -
<multop> → * | /

Example non-LL Grammar Construct

stmt → if condition then_clause else_clause
| other_stmt
then_clause → then stmt
else_clause → else stmt | ε
→ Ambiguous – allows dangling else to be paired with either then in
if A then if B then C else D

Fix – Bottom-up parsing (LR)

stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if condition then balanced_stmt
| else balanced_stmt
then balanced_stmt → if condition then stmt
| else unbalanced_stmt
unbalanced_stmt → if condition then stmt
| else balanced_stmt

OR
Use special disambiguating rules → use production that occurs first in case of conflict
**Parser Construction**

- Recursive descent parsing
  - Top-down parsing algorithm
  - Built on procedure calls (may be recursive)
  - Write procedure for each non-terminal, turning each production into clause
  - Insert call to procedure A() for non-terminal A and to match(x) for terminal x
  - Start by invoking procedure for start symbol S

**Predictive (Table-Driven) Parsing**

- Actions –
  - Match a terminal
  - Predict a production
  - Announce a syntax error
- Push as yet unseen portions of productions onto a stack
- Use –
  - FIRST(A)
  - FOLLOW(A)

**The FIRST Set**

- FIRST(α) is the set of terminal symbols that begin strings derived from α
- To build FIRST(X):
  - If X is a terminal, FIRST(X) is {X}
  - If X=ε, then ε ∈ FIRST(X)
  - If X=Y₁ Y₂ ... Yₖ, then put FIRST(Y₁) in FIRST(X)
  - If X is a non-terminal and X=Y₁ Y₂ ... Yₖ, then a ∈ FIRST(X) if a ∈ FIRST(Y₁) and ε ∈ FIRST(Yⱼ), for all 1≤j≤k

**The Follow Set**

- For a non-terminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form
- To build FOLLOW(B) for all B -
  - Starting with goal, place eof in FOLLOW(<goal>)
  - If A→αBβ, then put (FIRST(β)-ε) in FOLLOW(B)
  - If A→αB, then put FOLLOW(A) in FOLLOW(B)
  - If A→αB and ε ∈ FIRST(β), then put FOLLOW(A) in FOLLOW(B)

**Using FIRST and FOLLOW**

- For each production A→α and lookahead token
  - Expand A using the production if token ∈ FIRST(α)
  - If ε ∈ FIRST(α), expand A using the production if token ∈ FOLLOW(A)
  - All other tokens return error
- If there are multiple choices, the grammar is not LL(1) (predictive)

**LL(1) Grammars**

A Grammar G is LL(1) if and only if, for all non-terminals A, each distinct pair of productions A→α and A→β satisfy the condition FIRST(α) ∩ FIRST(β) = Φ, i.e.,

For each set of productions A→α₁α₂...αₙ

- FIRST(α₁), FIRST(α₂), ..., FIRST(αₙ) are all pairwise disjoint
- If αᵢ →ε for any i, then FIRST(αᵢ) ∩ FOLLOW(A) = Φ, for all j≠i
Bottom-Up Parsers

- End with start symbol
- Apply productions in reverse to input, replacing RHS of production with LHS non-terminal
- Final result is a rightmost derivation, in reverse
- Bottom-up parsers can use stack and lookahead to choose production
- LR(k) parsers see the entire RHS before choosing a production → more powerful than LL(k)

Definitions –
- The handle is defined as the combination of
  1) RHS to be replaced
  2) Its position
- Replacement step is called a reduction

Shift-Reduce Parsers

- Actions
  - Shift – next input symbol shifted onto stack
  - Reduce – right end of handle is on top of stack; locate left end of handle within stack; pop handle off stack and push appropriate non-terminal LHS
  - Accept – terminate parsing and signal success
  - Error – initiate/call an error recovery routine

Skeleton Parser

```
push all
s = top of stack
while (1)
  if action[s,token] = "shift s" then
    push s
    push s
  else if action[s,token] = "reduce A" then
    pop 2 * |β| symbols
    s = top of stack
    push A
    push goto[s,A]
  else if action[s,token] = "accept" then
    return
  else error
```

Shifts, Reduces, and Accept for a valid string, where A is the length of the input string and f is the length of the reverse rightmost derivation

LR(1) Items

- An LR(k) item is a pair [α,β], where α is a production from G with α at some position in the rhs and β is a lookahead string containing k symbols (terminals or eof)
  - The • indicates how much of an item we have seen at a given state in the parse
  - [A • XY • Z,a] indicates that the parser has seen a string derived from XY and is looking for one derivable from Z, and that a is in FOLLOW(A)
  - A • XYZ generates 4 LR(0) items

LR(1) Machine

- Definitions
  - Closure of [A • αX•β,a] contains itself and any items of form [B • αX•β, FIRST(b)], repeating for new items
  - goto(X) of [A • αX•β,a] contains the closure of [A • αX•β,a]
- LR(1) DFA construction
  - Begin with closure of start symbol [S • •,eof]
  - For each state, calculate goto(X) for all grammar symbols X, generating states
    - Each state is a set of LR(1) items
  - Repeat above step for newly generated states

LR(1) Table Construction

- If S appears on RHS, create augmented grammar G' by adding S • S
- Construct the collection of sets of LR(1) items for G'
  - For each state Si and each item in state Si
    - If [A • αX•β,a] in Si and goto[S] = Sj, then set action[i,a] to "shift f" (a must be a terminal)
    - If [A • αX•β,a] in Si, then set action[i,a] to "reduce A • α"
    - If [S • •,eof] in Si, then set action[i, eof] to "accept"
  - For all non-terminals
    - if goto(Si,A) = Sj, then set goto[i, A] to j
  - All other entries in action and goto are set to "error"
- The initial state of the parser is the state constructed from the set containing the item [S • •,eof]
Error Recovery

• Phrase-level
  – Delete tokens until something likely to follow expression (in FIRST or FOLLOW for expression or eof)
  – Context-specific FOLLOW sets
  – Avoid deleting statically defined “starter” symbols
• Exception-based
  – Small set of contexts to which to back out in case of error
  – Re-raise exception to pop to next handler on seeing starter symbol that it shouldn’t delete
  – Delete input tokens until parsing can recommence

Error Recovery Example

stmt_list ← stmt
| stmt_list; stmt

- Throw out the erroneous statement, synchronize at “;” or eof, invoke yyerror(“syntax error”)
- Other natural places for errors – lists, missing parentheses or brackets, extra operator or missing operator