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

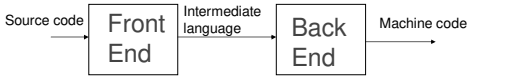
Types of languages

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

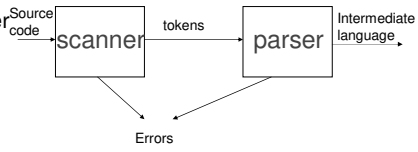
Phases of Compilation

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

Overview



- Front end
  - Scanner
  - Parser



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

Regular Expressions

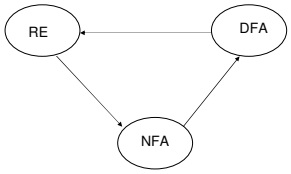
- Defines a set of strings over the characters contained in some alphabet, defining a language
- Three basic operations
  - Union – e.g., a|b
  - Concatenation – e.g., ab
  - Closure – (Kleene closure) – e.g., a* - where a can be a set concatenated with itself zero or more times

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



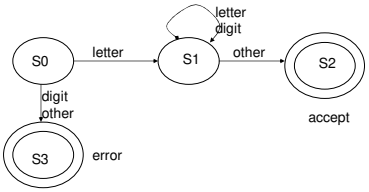
Recognizers

Identifier

letter → (a|b|c| … |z|A|B|C| … |Z)

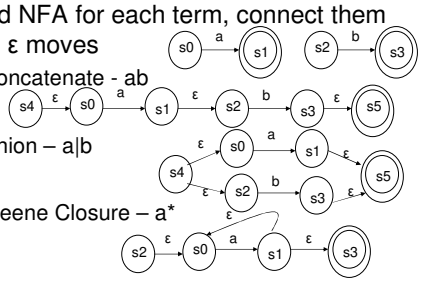
digit → (0|1|2|3|4|5|6|7|8|9)

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

NFA->DFA

- Subset construction algorithm
  - Each state in DFA corresponds to a set of states in NFA

q0 ← ε-closure(n0)
initialize Q with {q0}
while (Q is still changing)
  for each qi ε Q
    for each character α εΣ
      t← ε-closure(move(qi, α))
      T[qi, α] ← t
      if t ∉ Q then
        add t to Q

DFA Minimization

$P \leftarrow P \{S_F, \{S - S_F\}$
while (P is still changing)
  T ← 0
  for each set p ε P
    for each α ε Σ
      partition p by α
        into p1, p2, p3, … pk
      T← T U p1, p2, p3, … pk
  if T ≠ P then
    P ← T

Implementing Scanners

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

Code for Recognizer/Scanner

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

Tables Driving the Recognizer

char_class

	a-z	A-Z	0-9	other
value	letter	letter	digit	other

next_state

class	S0	S1	S2	S3
letter	S1	S1	--	--
digit	S3	S1	--	--
other	S3	S2	--	--

To change language, we can just change tables

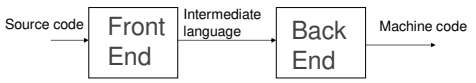
Error Recovery

- 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

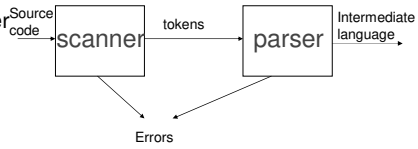
Scanner Summary

- 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



Parsers

- 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 ser of productions or rewrite rules
    (P : N → 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> <add op> <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> → <stmt> <stmtlist> | 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
| other_stmt
unbalanced_stmt → if condition then stmt
| if condition then balanced_stmt else unbalanced_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$\leftarrow\varepsilon$, then $\varepsilon \in$ FIRST(X)
  - If X$\leftarrow$Y1 Y2 … Yk then put FIRST(Y1) in FIRST(X)
  - If X is a non-terminal and X$\leftarrow$Y1 Y2 … Yk, then a ε FIRST(X) if a ε FIRST(Yi) and $\varepsilon \in$ FIRST(Yj), for all $1 \leq j < i$

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$\leftarrow\alpha$Bβ, then put {FIRST(β)-ε} in FOLLOW(B)
  - If A$\leftarrow\alpha$B, then put FOLLOW(A) in FOLLOW(B)
  - If A$\leftarrow\alpha$Bβ and $\varepsilon \in$ FIRST(β), then put FOLLOW(A) in FOLLOW(B)

Using FIRST and FOLLOW

- For each production A$\leftarrow\alpha$ and lookahead token
  - Expand A using the production if token $\in$ FIRST(α)
  - If $\varepsilon \in$ FIRST(α), expand A using the production if token $\in$ 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 $\leftarrow \alpha$ and A$\leftarrow\beta$ satisfy the condition FIRST(α) $\cap$ FIRST(β) = Φ, i.e.,

For each set of productions A$\leftarrow \alpha 1|\alpha 2|\ldots|\alpha n$

- FIRST(α1), FIRST(α2), …, FIRST(αn) are all pairwise disjoint
- If $\alpha_i \leftarrow\varepsilon$ for any i, then FIRST(α_j) $\cap$ FOLLOW(A) = Φ, for all $j \neq i$

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 s0
token = next_token()
 while (1)
        s = top of stack
        if action[s,token] = "shift si" then
                push token
                push si
                token = next_token()
        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()
```

Results in *k* shifts, *l* reduces, and 1 accept for a valid string, where *k* is the length of the input string and *l* 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 a • 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← α•Bβ,a] contains itself and any items of form [B ← •γ, FIRST(βa)], repeating for new items
  - goto(X) of [A← α•Xβ,a] contains the closure of [A← αX•β,a]
- LR(1) DFA construction
  - Begin with closure of start symbol [S ← •α,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← α•aβ,b] in Si and goto[Si] = Sj, then set action[i,a] to "shift j" (a must be a terminal)
  - If [A← α•,a] in Si, then set action[i,a] to "reduce A← α"
  - If [S'←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'← •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

can be augmented with error

stmt_list ← stmt
| error
| 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