Week 6: Book

Friday, February 25, 2022 7:38 AM

Parsing = Syntax Analyzer

Parsing: how a string of terminals can be generated by a grammar
↳ the parser must be able to construct the tree (in theory)

→ For any context-free grammar there is a parser (syntax analyzer) that takes at most $O(n^3)$ time to parse a string of n terminals.
↳ too slow = $O(n)$ (linear) parser exist in practice

- Parsing Methods: Classes, refer to the order in which nodes in the parse tree are constructed.
  - easy to write ← 1) top-down: construction starts at the ROOT and proceeds towards the LEAVES.
  - handles bigger grammars ← 2) Bottom-up: construction starts at the LEAVES and proceeds towards the ROOT.

2.6 Lexical Analyzer

↳ reads characters from the input and groups them into TOKEN OBJECTS
↳ We say TOKENS = TERMINALS since the parser ignores the info of the 2nd part of a token (attribute values)

TOKEN = ⟨token name, attribute value⟩
↳ a token is: a terminal with additional information

→ a sequence of input chars that is a single token = LEXEME.

2.6.1 Removal of White space and Comments

↳ Most languages allow arbitrary amounts of white space between TOKENS
↳ Comments are likewise ignored during parsing, so they are treated as white spaces

→ If white space is eliminated by the lexical analyzer, the parser will never have to consider it.
↳ the alternative of modifying the grammar to accept white space is not easy.

```
for ( ; ; peek = next input character ) {
    if ( peek is a blank or a tab ) do nothing;
    else if ( peek is a newline ) line = line+1;
    else break;
}
```

line → counter for errors

This code skips white space by reading input characters as long as it sees blank/tabs/lines
peek: the next input char

2.6.3 Constants

↳ integer constants can be allowed either by creating a terminal symbol ("num") for such constants OR by adding syntax of integer constants into the GRAMMAR.

↳ the job of collecting chars into integers → lexical analyzer = numbers treated as a unit in syntax analysis

→ When a sequence of digits appears in the input stream,

lexical analyzer → passes to the PARSER a TOKEN
↳ consisting of the terminal NUM along w/ integer attribute coming

from the digits

* example: the input 31+28+59 is transformed into:

⟨num, 31⟩⟨+⟩⟨num, 28⟩⟨+⟩⟨num, 59⟩

↳ the terminal '+' has no attributes, so ⟨+⟩

```
if ( peek holds a digit ) {
    v = 0;
    do {
        v = v * 10 + integer value of digit peek;
        peek = next input character;
    } while ( peek holds a digit );
    return token ⟨num, v⟩;
}
```

Figure 2.30: Grouping digits into integers

2.6.4 Recognizing keywords and identifiers

→ Keywords: fixed character strings used as punctuation marks or to identify constructs. i.e. FOR, DO, IF, etc.

→ char strings are also used as IDENTIFIERS to name variables, arrays and functions.
↳ grammars treat identifiers as terminals to SIMPLIFY the parser

everytime an identif. appears. ← expects terminal "id"

* example: on input

count = count + increment;

The parser works with terminal stream: id = id + id

the TOKEN for id has an attribute to hold the LEXEME.

The tuples for the input stream are:

⟨id, "count"⟩ ⟨=⟩ ⟨id, "count"⟩ ⟨+⟩ ⟨id, "increment"⟩ ⟨;⟩

→ Warning: Keywords generally satisfy the same rules for forming identifiers
↳ solution: keywords need to be 'reserved': cannot be used as identifiers
= A character string forms an identifier ONLY if it is not a keyword.

→ the LEXICAL analyzer solves 2 problems by using a table to hold char strings

the string table can be implemented using a hash table

① Single representation: by using a table, the compiler can use the pointers instead of the string: faster.

② Reserved words: initialize the table with them and their tokens
↳ when it reads a string (ID/KEYWORD) that could be an identifier, it first checks if it is in the table.
↳ returns a token with terminal "id".

```
if ( peek holds a letter ) {
    collect letters or digits into a buffer b;
    s = string formed from the characters in b;
    w = token returned by words.get(s);
    if ( w is not null ) return w;   ↳ check keyword list
    else {
        Enter the key-value pair (s, ⟨id, s⟩) into words
        return token ⟨id, s⟩;
    }
}
```

Distinguishing keywords from identifiers.

The code snippets so far can form a function scan() that returns token objects:

```
Token scan() {
    skip white space, as in Section 2.6.1;
    handle numbers, as in Section 2.6.3;
    handle reserved words and identifiers, as in Section 2.6.4;
    /* if we get here, treat read-ahead character peek as a token */
    Token t = new Token(peek);
    peek = blank /* initialization, as discussed in Section 2.6.2 */ ;
    return t;
}
```

2.7 Symbol Tables

↳ data structures used by compilers to hold info about CONSTRUCTS

↳ this info is collected incrementally by the analysis phase and used by the synthesis phase.

↳ Support multiple declarations of the same identifier in a program.

→ Entries: contain info about an identifier, like:

- its char string or LEXEME
- type
- position in storage

→ Scope: the scope of a declaration is the portion of a program to which the declaration applies.

↳ a program block has its own symbol table with an entry for EACH DECLARATION IN THE BLOCK.

→ The ENTRIES are created and used by lexical, syntax and semantic analyzer (analysis phase)

```
1   int x1 ;
2   x1
3       x1
4   x1
```

all occurences of x_1 are within the scope of the declaration on line 1 (subscript)

→ Symbol Table Interface : The symbol Table is a Hash Table object with the actions:

- put : put a new entry in the symbol Table. The hash table has key-value
  - key: is a string, or a reference to a string. Use reference to tokens for identifiers.
  - Value: is an entry of class Symbol.

could also return the index of the entry

```
public void put(String s, Symbol sym) {
    table.put(s, sym);
}
```

- get: an entry for an identifier by searching the chained tables (tables of all blocks), starting with the table of current block.

↳ returns either a symbol table entry or NULL.

returns the index of the entry for the string s | ∅

```
public Symbol get(String s) {
   for( Env e = this; e != null; e = e.prev ) {
      Symbol found = (Symbol)(e.table.get(s));
      if( found != null ) return found;
   }
   return null;
}
```

3 Lexical Analyzer

↳ the part of the compiler that reads the source text

↳ identifies LEXEMES.

↳ strips out comments / white space

blank, newline, tab, chars to separate TOKENS.

↳ correlates error messages w/ the source program

→ Parsing = Syntax analysis

→ 3 terms:

a) TOKEN: is a pair consisting of TOKEN NAME and an optional ATTRIBUTE VALUE.

- Token name: abstract symbol representing a lexical unit i.e. keyword or a char string of an identifier

↳ is what the parser processes.

b) PATTERN: is a description of the form that the LEXEMES of a TOKEN may take.
↳ a keyword as a token (name), the pattern is the sequence of chars that form it.
↳ for identifiers, the pattern is a more complex structure that is matched by many strings.

c) LEXEME: a sequence of characters in the source program that matches the PATTERN for a token and is identified by the lexical analyzer as an INSTANCE of that token.

Token	Informal Description	Sample Lexemes
if	characters i, f	`if`
else	characters e, l, s, e	`else`
comparison	< or > or <= or >= or == or !=	`<=, !=`
id	letter followed by letters and digits	`pi, score, D2`
number	any numeric constant	`3.14159, 0, 6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

i.e. the C statement

printf("Total = %d\n", score);

↳ lexemes matching the pattern for token "id" and "string" is a lexeme matching token "literal"

Some rules:
1. One token for each keyword
2. Tokens for operators (either for each or by classes) ← "comparison"
3. One token representing all identifiers
4. One or more tokens for constants: numbers, literal and strings
5. Token for each punctuation symbol, such as left and right parentheses, comma and semicolon

3.1.4. Lexical Errors

It is hard to tell with only the lexical analyzer if there is an error.
i.e.

fi(a == f(x)) . . .

lexical cannot tell if it is a misspelling for "if" or an undeclared function identifier.
↳ Since fi is a valid lexeme for the token "id", the lexical must return the token "id" to the parser and let other phase (parser) handle the error.

→ Suppose a situation arises where the lexical analyzer cannot proceed because none of the patterns for tokens matches any prefix of the remaining input
↳ the simplest recovery is PANIC MODE RECOVERY
delete successive characters from remaining input, until lexical finds a well-formed token at the beginning of what input is left.

→ other error-recovery actions are:
1. Delete one char from the remaining input.
2. Insert a missing char into the remaining input.
3. Replace a char by another char
4. Transpose two adjacent chars.

attempts to repair the input

the simplest → see whether a prefix of the remaining input can be a valid lexeme by 1 TRANSFORMATION

→ Regular Expressions are an important notation for specifying LEXEME PATTERNS.

→ alphabet: any finite set of symbols → string: finite set of symbols from an alphabet

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.
2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.
3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

→ ϵ : empty string

→ language: set of strings over an alphabet.

3.4.1 Recognition of Tokens: Transition Diagrams

↳ Transition diagrams have a collection of nodes/circles, called STATES

→ Each STATE represents a condition that could occur

scans until a pattern is found

STATE → all characters between the lexeme Begin and forward pointer

↳ EDGES are directed from one state to another.

→ Each EDGE is labeled by a symbol / set of symbols

i.e. If we're in state S and the next input symbol is "a",

we look for the edge out of state S labeled "a".

↳ if we find it, we advance the forward pointer

and enter the STATE were EDGE labeled "a" ends.

→ DETERMINISTIC: a diagram with never more than ONE EDGE out of a STATE with a given symbol in its labels.

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.
3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.