SWEN430 - Compiler Engineering (2018)

Lecture 3 - Parsing I

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Assignment 1

Extend the While parser/interpreter to add:

- Comments: line and block comments, as in Java.
- Constant declarations
  `const PI is 3.14.15`
- Switch statements
- Assignment to string elements

Recap: Scanner (Lexer, Tokeniser)

- Read input as a sequence characters/lines
- Output a sequence of tokens/lexemes/symbols
- Report lexical errors (illegal symbols)
- Strip out comments and white space
- Localises handling of white space, reading/counting lines, and (possibly) checking for end of file, proving a simpler interface for the parser.

Scanner design

- Set of tokens designed to form a *regular language*.

  Actually, a left-to-right deterministic regular language.

- Regular languages can be defined using *regular expressions*, using sequence, alternation, repetition and optional constructs.

- E.g.: $\textit{arithOp} = + \mid - \mid * \mid /$

  $\textit{relOp} = \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{!=}$

  $\textit{ident} = \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^*$

  $\textit{number} = \textit{digit}\ \textit{digit}^*\ [\ .\ \textit{digit}^*\ ]$

- What if you want to use brackets etc. in the language being defined?

Scanner design

- Regular languages can be recognised using *finite state acceptors*.
- A *finite state acceptor* is an abstract machine which:
  - has a finite amount of memory (the machine is in one of a finite number of states, had has no other memory)
  - at each step determines what to do based solely on the current state and the next input
  - accepts the input if it reaches the end of the input in an *accepting state*, and otherwise rejects the input.

Scanner design

A symbol/token/lexeme usually consists of:

- a *symbol type*, such as *semicolon*, *identifier*, or *number*;
- the actual text of the symbol;
- other information, such as line/character number, needed for diagnostics

Example

Input:

```
Test.java

class Test {
    public static void main(String[] args) {
        System.out.println("Hello␣World"); } }
```

Possible scanner output:

```
(keyWd, "class", 1, 1), (ident, "Test", 1, 7),
(lBrace, "{", 1, 12), (keyWd, "public", 2, 5),
(keyWd, "static", 2, 12), (keyWd, "void", 2, 19),
(keyWd, "main", 2, 25), (lParen, "(", 2, 29),
(keyWd, "String", 2, 30), (lBrack, "[", 2, 36),
(rBrack, "[", 2, 37), (ident, "args", 2, 39),
(rParen, ")", 2, 43), (lBrace, "{", 2, 45), ...
```

Scanner design

- The scanner is based on a *finite state acceptor*.
- Read one character at a time, and decide what to do next based on the next character, i.e. one character look ahead (or maybe 2 or 3).
- May be table driven or generated from a set of regular expressions (or a regular grammar), or hand coded — in which case the FSA is implicit in the program code.
- Often the most time consuming part of a compiler, so must be fast.
- May run scanner over the entire input and create a token string which is passed to the parser;

  or call scanner as a `nextToken` method from the parser;

  or run scanner and parser in parallel, e.g. as coroutines.
- Ex: Look at the code for the While lexer.

Scanner design

- Basic outline:

```
while there is more input do
  ch <- getNextChar
  case what kind of token can ch start of
    number: scanNumericCont;
    string: scanStringConst;
    ident: scanIdentifier;  // includes keywords
    operator: scanOperator;
    whiteSpace: scanWhiteSpace;
    otherwise: error
```

- Each method scans one kind of token and advances over all of the characters in that token.
- Sometimes look for white space before/after each token.
- Adding an *eof* char avoids checking everywhere for end of input:

```
ch <- getNextChat
while ch neq eof do
  case what kind of token can ch start of
```

Scanner design

Code for individual tokens is based on the structure of the RE:

- Sequence: $e_1\ e_2\ \ldots\ e_n$

  Look for occurrences of e_1, e_2, ..., e_n in turn.

- Alternation: $e_1 \mid e_2 \mid \ldots \mid e_n$

  If next char can start e_1, look for an occurrence of e_1
  ...
  else, if next char can start e_n, look for an occurrence of e_n
  else signal error.

- Repetition: e^*

  While next char can start e, look for an occurrence of e.

- Optional: $[e]$

  Exercise.

Scanner design

How can we be sure to make the right decision based just on the next character?

- In an alternation $e_1 \mid e_2 \mid \ldots \mid e_n$:

  No character can start more than one e_i.

  E.g. Can't have $ab|ac$. Rewrite as ...

- In a sequence, $e_1\ e_2\ \ldots\ e_n$:

  No character can start e_i (for $i < n$) and also start e_{i+1}.

  E.g. can't have $(a,)^* a$. Rewrite as ...

  Note: This means that any two identifiers/numbers must be separated by white space! Keywords are treated as a special case of identifiers.

- What about optional: $[e]$?

  Ex.

Parser design

- The set of syntactically valid programs is designed to be a (deterministically parsable) *context-free language*, defined by a form of *context-free grammar*.
- The parser is based on a *push-down automaton* — essentially an FSA with extra memory in the form of an unbounded stack.
- Read one token at a time and decide what to do next based on that token, i.e. one symbol look ahead (or sometimes ...).
- May build a parse tree from root down to leaves (top-down, LL(k); or from leaves up to root (bottom-up, LR(k)).
- May be table driven or generated from a context-free grammar, or hand coded — in which case the PDA is implicit in the program code.
- Lots of tools for generating scanners and parsers (yacc and lex, bison, antlr, ...).

Recursive Descent Parsing

- We'll use a form of top-down hand coded parser called *recursive descent* or *predictive parsing*.
- Simple but powerful deterministic parsing method — uses one (or more) lookahead symbols to determine what to look for next.
- Grammars and languages for which recursive descent works are called LL(k), where k is the number of lookahead symbols needed.
- Most programming languages structures turn out to be LL(1).
- For each nonterminal *N* in the grammar, define a method *parseN* to recognise an instance of *N* as a prefix of the input, and build an AST for it.
- Logic of the parse methods reflects the structure of the grammar. Can be coded directly from the grammar once it is in LL(1) form.
- Easy to extend to do error analysis/reporting/recovery, semantic checking and building AST.

Recursive Descent Parsing

- A *Context-Free Grammar* (CFG) is a set of rules of the form

$$N \longrightarrow A_1 \mid \ldots \mid A_n$$

  where, N is a *non-terminal*, and $A_1, \ldots, A_n$ are strings of *terminals* and/or non-terminals.

- Terminals are symbols that actually appear in a program
  Non-terminals are names of structural components of a program

- The parser method for such a rule (ignoring tree building) is:

```
parseN()
  if nextSym can start A1 then recognise A1
  ...
  else if nextSym can start An then recognise An
  else error(nextSym can't start N)
```

Recursive Descent Parsing

- If A_i is $X_1 \ldots X_m$
- Then `recognise Ai` is:

```
recognise X1;
...
recognise Xm;
```

- Where `recognise Xj` calls `parseXj` if X_j is a nonterminal, and looks for terminal X_j otherwise.
- This needs some more plumbing to handle errors and build AST.
- In practice, we can simplify the parser a bit.

Recursive Descent Parsing

- Example: $$\begin{array}{lcl} E & \longrightarrow & N \mid V \mid \texttt{(}E\texttt{+}E\texttt{)} \\ N & \longrightarrow & [\texttt{0-9}]^{+} \\ V & \longrightarrow & [\texttt{a-zA-Z\_}]^{+}[\texttt{a-zA-Z0-9\_}]^{*} \end{array}$$

  Note that this uses Unix regular expression notation to define sets of terminals.

- Parser is:

```
parse E
  if nextSym is N then advance
  else if nextSym is V then advance
  else if nextSym is '(' then
         parse('(');
         parseE;
         parse('+');
         parseE;
         parse(')');
  else error(nextSym can't start E)
```

When does it work? LL(1) conditions

- Must be able to decide what to do on basis of next input symbol.
- So, given *N* $\longrightarrow$ *A* | *B*, no symbol that can start an *A* can also start a *B*.
- Define *first* sets:

  Let γ be a sequence of terminal and non-terminal symbols. Then *first*(γ) is the set of all terminal symbols which begin a string derived from γ.
- Code `nextSym can start N` as `nextSym in first(N).`
- We can now state the **Choice Condition**:

  For any rule *N* $\longrightarrow \alpha \mid \beta$, it must hold that *first*(α) $\cap$ *first*(β) $= \emptyset$.

LL(1) — first() sets

- Example: $$\begin{array}{ll} S \longrightarrow T\ \mathtt{a} \mid U\ \mathtt{b} & (1) \\ T \longrightarrow \mathtt{d}\ T \mid \mathtt{e} & (2) \\ U \longrightarrow \mathtt{c}\ U \mid \mathtt{f} & (3) \end{array}$$

- For rule (2): $\mathit{first}(dT) = \{d\}$ and $\mathit{first}(e) = \{e\}$.

- For rule (3): $\mathit{first}(cU) = \{c\}$ and $\mathit{first}(f) = \{f\}$.

- For rule (1): $\mathit{first}(Ta) = \mathit{first}(T) = \mathit{first}(dT) \cup \mathit{first}(e) = \{d, e\}$,
  and $\mathit{first}(Ub) = \mathit{first}(U) = \mathit{first}(cU) \cup \mathit{first}(f) = \{c, f\}$.

- Does this satisfy the choice condition?

- Ex: Write (or find) a recursive definition of *first*.

LL(1) — first() sets

- What are the first() sets for these grammars?

  1. $S \longrightarrow \mathtt{a}\ S \mid \mathtt{a}$

  2. $T \longrightarrow \mathtt{c}\ T \mid \mathtt{d}$
     $U \longrightarrow \mathtt{c}\ U \mid \mathtt{e}$

  3. $S \longrightarrow T\ \mathtt{a} \mid U\ \mathtt{b}$
     $S \longrightarrow T\ \mathtt{a} \mid U\ \mathtt{c}$
     $T \longrightarrow \mathtt{d}\ T \mid \mathtt{e}$
     $U \longrightarrow \mathtt{c}\ U \mid \epsilon$

- Do they satisfy the choice condition?

Questions to ponder

- How do we extend the Choice Condition to $N \longrightarrow \alpha_1 | ... | \alpha_n$?
- Are there any cases the Choice Condition doesn't handle?

  i.e. anywhere else the parser has to make a choice?
- How can we extend this to handle extended BNF grammars, where we can write:
  - $[\alpha]$ to mean that α is optional.
  - α^* to mean that α is repeated 0 or more times.
  - α^+ to mean that α is repeated 1 or more times.