COMP2212 Programming language concepts

Lecture 4 - Lexing and parsing
Processing syntax

- We have seen how to use grammars to derive program strings.
- Lexing and parsing goes the other way: Given a string, does there exist a derivation?
  - program string
  - identify tokens
    - lexer, using regular expressions
  - produce abstract syntax
    - parser, producing a parse tree
Lexical analysis (lexing)
Lexing

- Essentially pattern matching on text
- Identify **tokens** and interface with parser
- eg.
Lexing

- typically tokens are identified with regular expressions
- a lexer constructs an automaton, which is executed as individual characters are read
- recall from theory of computing: power of regular expressions is the same as power of finite automata
Example (alex)

```haskell
{ module Main (main) where

%wrapper "basic"

$digit = 0-9   -- digits
$alpha = [a-zA-Z]  -- alphabetic characters

tokens :-

$white+  ;
"--".* ;
let { \s -> Let }
in { \s -> In }
$digit+ { \s -> Int (read s) }
[\=\+\-\*\/\()\]  { \s -> Sym (head s) }
$alpha [$alpha $digit \_ \']*  { \s -> Var s }

-- Each action has type :: String -> Token

-- The token type:
data Token =
Let |
In |
Sym Char |
Var String |
Int Int
 deriving (Eq,Show)

main = do
  s <- getContents
  print (alexScanTokens s)
}
```
Parsing
Parsing

- **top-down** - parse tree is built from the root down
  - *recursive descent* parsers
    - LL grammars (leftmost derivations)
- **bottom-up** - parse tree is built from the leaves up
  - similar to pushdown automata
  - LR grammars (rightmost derivations)
Recursive-descent parsing

- typically one procedure for each production rule of the grammar

eg

\[\text{<expr>} \rightarrow \text{<term>} \{(\ + \ | \ -) \text{<term>}\}\]

\[\text{<term>} \rightarrow \text{<factor>} \{(\ * \ | \ /) \text{<factor>}\}\]

\[\text{<factor>} \rightarrow \text{id} \ | \ \text{int\_constant} \ | \ (\text{<expr>})\]

```c
void expr() {
    printf("Enter <expr>\n");
    /* Parse the first term */
    term();
    /* As long as the next token is + or -, get the next token and parse the next term */
    while (nextToken == ADD_OP || nextToken == SUB_OP) {
        lex();
        term();
    }
    printf("Exit <expr>\n");
}
```

```c
void term() {
    printf("Enter <term>\n");
    /* Parse the first factor */
    factor();
    /* As long as the next token is * or /, get the next token and parse the next factor */
    while (nextToken == MULT_OP || nextToken == DIV_OP) {
        lex();
        factor();
    }
    printf("Exit <term>\n");
    /* End of function term */
}
```

Example from Sebesta, “Programming Language Concepts”, 10th ed
Example continued

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )

```c
void factor() {
    printf("Enter <factor>\n");
    /* Determine which RHS */
    if (nextToken == IDENT || nextToken == INT_LIT) /* Get the next token */
        lex();
    /* If the RHS is ( <expr>), call lex to pass over the */
    /* left parenthesis, call expr, and check for the right */
    else { /* It was not an id, an integer literal, or a left */
        if (nextToken == LEFT_PAREN) { /* parenthesis */
            lex();
            expr();
            if (nextToken == RIGHT_PAREN)
                lex();
            else /* End of if (nextToken == ... */
                error();
        } /* End of if (nextToken == ... */
    /* It was not an id, an integer literal, or a left */
    else /* parenthesis */
        error();
} /* End of else */
printf("Exit <factor>\n");
} /* End of function factor */
```
Top-down LL parsing

❖ Consider a left recursive grammar
  ❖ \( A \rightarrow A + B \)
  ❖ or \( A \rightarrow BaA, B \rightarrow Ab \)
❖ what goes wrong with a recursive descent parser?
❖ it is possible to convert a left recursive grammar to one where there is no left recursion
❖ cf. Greibach normal form
Example (parsing)

https://hackage.haskell.org/package/parsc

- parsing library in Haskell that uses recursive descent
- examples in Friday’s lecture
Top-down LL parsing

- Basic problem: we need to be able to determine the RHS in a production based on the next token
  - left factoring - convert a grammar into one that passes this requirement
- LL(k) grammars - k refers to the lookahead
- LL(1) - can determine the RHS with 1 token lookahead
Bottom-up LR parsing

- Uses a stack and works similarly to a deterministic pushdown automaton
- The statespace and transition functions are precomputed - this is known as an LR parsing table
- Linear time parsing (in the size of the input string)
- Requires preprocessing, typically done by a tool (such as yacc)
lex and yacc

- Classic C tools for lexing and parsing
  - lex - builds lexers
    - (Alex - a Haskell variant)
  - yacc (Yet Another Compiler Compiler) - builds parsers
    - (Happy - a Haskell variant)
Coursework choices

❖ it’s ok to roll your own recursive descent parser
❖ use a parsing library like Parsec, other libraries also available (Google is your friend)
  ❖ Advantages: it’s a library, so you can use it within your code, many useful features
  ❖ Disadvantages: you have to pay more attention to grammar design (e.g. precedence, issues with left recursion, etc.)
❖ you can use a parser generator like Happy.
  ❖ Advantages: specify grammar at high level, help with precedence, associativity and
  ❖ Disadvantages: a separate program, which requires pre-processing, so new syntax to learn and slightly more complicated compilation.
❖ In either case, you can use Alex for lexing, or roll your own lexer