Lecture 11: SLLGEN tutorial

➡ Introduction

✦See EOPL Appendix-A (p.345)

- Powerful LL(1) parser generator
- Input:
  1. Specification of lexical tokens (lexemes) as regular expressions
  2. Specification of the PL BNF in a rule format
- Output: A function which parses PL input syntax and produces an AST representation of the input program
- See Wikipedia: http://en.wikipedia.org/wiki/Regular_expression

Scanning using SLLGEN

➡ Specifying Scanners

- Defines tokens of the PL which can appear in the input syntax
- Given by a regular expression rules
- Example (our PL tokens)

```
(define the-lexical-spec
  '((whitespace (whitespace)                                  skip)
    (comment    ("%" (arbno (not #\newline)))                 skip)
    (identifier (letter (arbno (or letter digit "_" "-" "?")))  symbol)
    (number     (digit (arbno digit))                         number)))
```

- Interpretation:

1. Any predefined “whitespace” characters are skipped
2. Any arbitrary string of characters following a “%” upto a newline are skipped
3. An identifier is a letter followed by an arbitrary number of contiguous digits, letters or specified punctuation characters
4. A number is any digit followed by an arbitrary number of digits

➡ Lexical Rule formalism

- Each lexical rule has 3 parts:
  1. a name for the token
  2. a pattern of characters expressed as a regular expression
  3. an action to be taken by the scanner
- Actions
  1. skip - ignore the input characters
  2. symbol - make a PL identifier
  3. number - make a literal number from the token string
  4. string - make a PL string literal from the token string

➡ Behaviour

- The scanner reads characters from the input stream
- Finds the longest possible match of input characters given the lexical rules
- Converts each such maximal string into a token (as per above rules)
- Scanners are built by calling the method sllgen:make-string-scanner

➡ Examples of Scanning

- Define a new scanner

```
(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))
```

- Given our lexical scanner rules above (called the-lexical-spec)

```
foo bar x = 3 % comment
```

- Produces (using just-scan method)

```
((identifier foo 1)
 (identifier bar 1)
 (identifier x 1)
 (literal-string28 "=" 1)
 (number 3 1))
```

- Note: the 3rd field is for debugging information (ignore)

Parsing using SLLGEN

➡ Specifying Parsers

- Second input parameter to the parser generator
- A grammar is a list of production rules
- Each rule has 3 parts:
  1. a *left-hand-side* (LHS) which specifies the non-terminal symbol of the corresponding PL BNF syntax rule (and the name of an abstract datatype)
  2. a *right-hand-side* (RHS) which is a list of terminal symbols (punctuation and keywords) plus other non-terminal symbols in the grammar
  3. a production name which will be the name of the abstract datatype variant for the LHS datatype.
- SLLGEN will automatically generate these abstract datatypes from the grammar (using sllgen:make-defined-datatypes)
- Note: the generated parser will produce abstract datatype variants as output
- To make a string parser from a set of lexical rules and grammar rules, we use the method sllgen:make-string-parser

➡ Examples

```
(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))

>(scan&parse "+ (x, 3)")
#<struct:a-program>

> (eval-program (scan&parse "+ (x, 3)"))
13
```

➡ Define the Read-Eval-Print loop

```
(define read-eval-print
  (sllgen:make-rep-loop "--> "
                        (lambda (pgm) (eval-program pgm))
                        (sllgen:make-stream-parser the-lexical-spec the-grammar)))
```

- Try it!