COM 3351 October 14, 1993
Dr. Wand Readings: EOPL, Chap. 11 (much altered)

Lecture 3: Syntax: Scanning and Parsing

Lexical Scanning

Programs are just strings of characters. Need to group these characters into meaningful units. This grouping is divided into two stages: *scanning* and *parsing*.

Scanning is dividing the sequence of characters into words, punctuation, etc. These units are called *lexical items*, *lexemes*, or most often *tokens*. Refer to this as the *lexical structure* of the language.

Parsing is organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing (diagramming) an English sentence into clauses, etc. Refer to this as the *syntactic* or *grammatical* structure of the language.

Typical pieces of lexcical specification:

- Any sequence of spaces and newlines is equivalent to a single space.
- A comment begins with `%` and continues until the end of the line.
- An identifier is a sequence of letters and digits starting with a letter, and a variable is an identifier that is not a keyword.

What's in a token?

Data structure for token consists of two pieces:

- A *class*, a Scheme symbol that describes what kind of a token you've found. The set of classes is part of the lexical specification. Typical examples are: `identifier`, `number`, `lparen`, `rparen`, `begin`, `end`, `if`.
- A piece of data describing the particular token. For punctuation and keywords, like `lparen`, `rparen`, `if`, there is no data, so we just use `#f`. For an identifier, the datum would be a symbol (the name of the identifier). For a number, the datum would be the number described by the number literal.

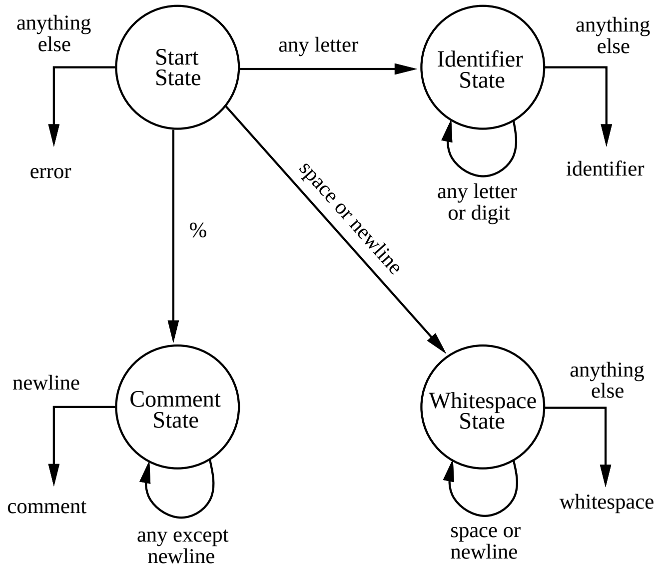
  In a language that didn't have symbols, we might use a string (the name of the identifier), or an entry into a hash table indexed by identifiers (a *symbol table*) instead. Using Scheme spares us these annoyances.

Job of the scanner is to go through the input and analyze it to produce these tokens. Generally the scanner is a subroutine that, when called, produces the "next" token of the input.

Could write scanner from scratch, but that's tedious and error-prone. Better idea: write down the lexical specification in a specialized language.

Specification via finite automata

Automaton for the sample specification above:



Start in start-state, then collect characters in buffer as you follow the arrows. Quit when you fall off the graph.

Specification via regular expressions

Automata are nice, but still tedious to express in textual form. Another approach: *regular expressions*

$$
\begin{aligned}
R \quad ::= \; & \langle\text{character}\rangle \mid R\,R \mid R \cup R \mid R^* \\
& \mid \neg R \mid R \cap R
\end{aligned}
$$

$$\langle\text{whitespace}\rangle = (\langle\text{space}\rangle \cup \langle\text{newline}\rangle)\ (\langle\text{space}\rangle \cup \langle\text{newline}\rangle)^*$$

$$\langle\text{comment}\rangle = \texttt{\%}\ (\neg\langle\text{newline}\rangle)^*$$

$$\langle\text{identifier}\rangle = \langle\text{letter}\rangle\ (\langle\text{letter}\rangle \cup \langle\text{digit}\rangle)^*$$

Specification in SLLGEN

SLLGEN is a package I put together in about two days this summer. It generates scanners and parsers.

The scanners are specified by automata, but you can put regular expressions on the transitions. Here's our example:

```
(define automaton-0
  '(()                                    ; no keywords
    (start-state
      ((#\space #\newline)                ; list means OR
       (arbno (#\space #\newline))        ; ARBNO means star
       whitespace)                        ; emit whitespace token
      (#\%
        comment-state)                    ; go to comment-state
      (alphabetic
        (arbno (alphabetic numeric))
        identifier)
      (#\^ end-marker))                   ; need an end marker for testing
    (comment-state
      (#\newline comment)
      (any comment-state))))
```

Testing the Scanner

```
Chez Scheme Version 4.1b
Copyright (c) 1991 Cadence Research Systems

> (load "~/.chezrc")
loading chez-init.s
finished loading chez-init.s
> (load "~wand/pub/plangs/sllgen.scm")
> ;; [Here's where I defined automaton-0]
> (sllgen:string->token-list  automaton-0 "foo bar % here is a comment
      baz")
((sllgen:token identifier foo)
 (sllgen:token whitespace #f)
 (sllgen:token identifier bar)
 (sllgen:token whitespace #f)
 (sllgen:token comment #f)
 (sllgen:token whitespace #f)
 (sllgen:token identifier baz))
>
```

The SLLGEN scanner specification language

```
automaton ::= ((keyword ...)
               (state-name (regexp ... outcome) ...)
                                     {last regexp may be ELSE}
               ...)

regexp ::=  character          {matches the Scheme character given}
          | ALPHABETIC | NUMERIC | WHITESPACE
          | ANY                {matches any character}
          | (regexp ...)       {OR of these regexps}
          | (ARBNO regexp)     {STAR of the regexp}

outcome ::= #f | state-name | IDENTIFIER | NUMBER | symbol
```

An automaton is a list. First item in the list is a list of keywords (identifiers that are not variables). Each other item is the specification of a state.

Each state specification consists of the name of the state (a Scheme symbol), followed by a list of transitions.

Each transition looks like `(regexp1 regexp2 ... outcome)` : The corresponds to the regular expression `regexp1 regexp2 ...` , ie concatenation of regular expressions, see `identifier` example in `automaton-0`.

Last transition can be `else`; this is different from `any` because it does *not* absorb the character. This is used internally to build `arbno`'s.

An outcome can be one of the following:

- `#f`. This means this is the end of a token, but don't emit the token. This is used for whitespace and comments.
- A symbol that is the name of a state. Don't emit a token now, but go to that state.
- The symbol `identifier`. Convert the characters in the buffer into a Scheme symbol. Check to see if this symbol is one of the symbols in the keywords list. If so, emit a token with that keyword as its lexical class. Otherwise, emit an `identifier` token, with the symbol as its datum.
- The symbol `number`. Convert the characters in the buffer into a Scheme number. Emit a token with the class `number` and that number as its datum.
- Any other symbol. Is assumed to be the lexical class of a punctuation (eg `lparen`, `rparen`, etc.) Emit a token with that symbol as its class and `#f` as its datum.

(Look at `sllgen:cook-token` in `sllgen.scm` to see the details).

Another SLLGEN example

```
(define sllgen:automaton-2
  '((proc if then else let set! in =)   ; keywords
    (start-state
      (whitespace #f)
      ((alphabetic  #\* #\+ #\- #\/ #\! #\= #\:)
       (arbno (numeric alphabetic #\* #\+ #\- #\/ #\! #\= #\:))
       identifier)
      ((numeric #\-)
       (arbno numeric)
       number)
      (#\( lparen)
      (#\) rparen)
      (#\^ end-marker)
      (#\% comment-state))
    (comment-state
      (#\newline #f)
      (any comment-state))))
```

This is the scanner we'll use for most of our mini-languages.

Parsing

Parsing is organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing (diagramming) an English sentence into clauses, etc. Refer to this as the *syntactic* or *grammatical* structure of the language.

Output of parser is a *tree*, called *parse tree* or *abstract syntax tree* (AST). Mostly we'll call these AST's.

Example:

```
(define-record compound-command (cmd1 cmd2))
(define-record while-command (exp cmd))
(define-record assignment-command (var exp))
(define-record var-expression (var))
(define-record sum-expression (exp1 exp2))

;;; Example:  input:

begin x := foo; while x do x := (x + bar) end


;;; output:

(compound-command
  (assignment-command x (var-expression foo))
  (while-command
    (var-expression x)
    (assignment-command x (sum-expression
                            (var-expression x)
                            (var-expression bar)))))
```

Specification via context-free grammars

We've seen lots of grammars so far. These are called *context-free* grammars. (Why context-free? To find out, go take automata theory). The informal notation we have used most of the time to write down these grammars is called Backus-Naur Form (BNF).

Another Example:

$$
\begin{array}{lll}
\langle\text{command}\rangle & ::= & \texttt{begin}\ \langle\text{command}\rangle\ \texttt{;}\ \langle\text{command}\rangle\ \texttt{end} \\
 & | & \texttt{while}\ \langle\text{expression}\rangle\ \texttt{do}\ \langle\text{command}\rangle \\
 & | & \langle\text{identifier}\rangle\ \texttt{:=}\ \langle\text{expression}\rangle \\
\langle\text{expression}\rangle & ::= & \langle\text{identifier}\rangle \\
 & | & \texttt{(}\ \langle\text{expression}\rangle\ \texttt{+}\ \langle\text{expression}\rangle\ \texttt{)}
\end{array}
$$

Let's give names to the pieces of this grammar. We've got:

- **Non-terminals**. These are the syntactic classes being defined, like ⟨command⟩ and ⟨expression⟩.
- **Terminal Symbols**. These are the lexical items being processed, like ⟨identifier⟩, `begin`, etc. If we were being more careful, we would show the lexical classes explicitly, by writing, say `assign-sym` rather than `:=`.
- **Productions**. These are the rules that show how a non-terminal symbol can evolve into a string. We have 5 of these in our sample grammar.

A non-terminal can produce a string by applying a production, eg:

```
expression -> ( expression + expression )     sum-expression
           -> ( identifier + expression )     var-expression
           -> ( X + expression)
           -> ( X + ( expression + expression))  sum-expression
           -> ( X + ( identifier + expression))  var-expression
           -> ( X + ( Y          + expression))
           -> ( X + ( Y          + identifier))  var-expression
           -> ( X + ( Y          + Z         ))
```

We associate a *name* with each production; these become the record-types for our AST's; the AST documents the productions that were used to build the string. We don't have to keep non-terminal names or punctuation in the AST, because this information is already in the production.

Specifying grammars in SLLGEN

SLLGEN includes a sublanguage for specifying grammars. Here's our example again:

```
(define simple-while-grammar
  '((command
      (begin command semicolon command end)
      compound-command)
    (command
      (while expression do command)
      while-command)
    (command
      (identifier assign-sym expression)
      assign-command)
    (expression
      (identifier)
      var-expression)
    (expression
      (lparen expression plus-sym expression rparen)
      sum-expression)))
```

We'll also need a lexical specification:

```
(define automaton-for-while
  '((begin end while do)                     ; keywords
    (start-state
      (whitespace #f)
      (alphabetic
        (arbno (alphabetic numeric))
        identifier)
      (numeric (arbno numeric) number)
      (#\+  plus-sym)
      (#\:  assign-sym-state)
      (#\%  comment-state)
      (#\;  semicolon)
      (#\(  lparen)
      (#\)  rparen)
      (#\^  end-marker))
    (assign-sym-state
      (#\= assign-sym)
      (else (arbno (alphabetic numeric)) identifier))
    (comment-state
      (#\newline #f)
      (any comment-state))))
```

Protocol of my errors

```
(define test-while
  (sllgen:string->tree automaton-for-while simple-while-grammar))

Error in caddr: incorrect list structure (command (identifier assign-sym
expression)).

;;; oops -- no name specified for this production the first time I
;;; tried it

> (define test-while
    (sllgen:string->tree automaton-for-while simple-while-grammar))
> (test-while "begin x := foo; while x do x := (x + bar) end")

Error in sllgen:apply-actions: looking for do in ((term while) (non-term
expression) (term do) (non-term command) (reduce while-command)), found
(sllgen:token identifier do).

;;; oops -- (token identifier do) -- I forgot to put DO in the
;;; keywords list!

> (test-while "begin x := foo; while x do x := (x + bar) end")

Error in sllgen:apply-actions: looking for do in ((term while) (non-term
expression) (term do) (non-term command) (reduce while-command)), found
(sllgen:token identifier do).

;;; oops -- forgot to redefine TEST-WHILE
```

```
> (define test-while
    (sllgen:string->tree automaton-for-while simple-while-grammar))

> (test-while "begin x := foo; while x do x := (x + bar) end")
(compound-command
   (assign-command x (var-expression foo))
   (while-command
      (var-expression x)
      (assign-command
         x
         (sum-expression (var-expression x) (var-expression bar)))))
```

The SLLGEN grammar specification language

```
grammar ::= (production ...)       ; nonterm of first prod is
                                   ; start symbol.

production ::= (lhs rhs action)

lhs ::= symbol                     ; a symbol that appears in a
                                   ; lhs is a non-terminal, all others
                                   ; are considered terminals
rhs ::= (rhs-item ...)

rhs-item ::= symbol | (ARBNO rhs-item ...)

action ::= symbol | GOTO | EMIT-LIST
```

A grammar is a list of productions.

Important: The parser must be able to figure out which production to use knowing only (1) what production it's looking for and (2) the first symbol (token) of the string being parsed. Grammars in this form are called $LL(1)$ grammars (SLLGEN stands for Scheme LL(1) parser GENerator). This will be good enough for our purposes. SLLGEN will warn you if your grammar does not meet this restriction.

Each production consists of a left-hand side (a non-terminal symbol), a right-hand side (a list of rhs-item's) and an action.

The right-hand side of a production is a list of symbols or `arbno`'s. A symbol that appears as a left-hand side is assumed to be a non-terminal; all others are assumed to be terminal symbols.

The action is a symbol that will appear as the record-type of the AST node for this production. The special symbols `goto` and `emit-list` are used internally to process `arbno`'s, but can be used manually; see EOPL, pp. 399–401.

What goes in the AST? The subtree for each non-terminal, of course. Also the data fields of any tokens that are not `#f`, ie identifiers and numbers.

An `arbno` generates a single item in the AST, which will be a *list* of the data for the things inside the `arbno`. Example: Change our grammar to say

```
(define while-grammar-2
  '((command
      (begin (arbno command semicolon) end)
      compound-command)
    ...))
```

This makes a compound command a sequence of semicolon-*terminated* commands. Then our ASTs look like:

```
> (define test-while
    (sllgen:string->tree automaton-for-while while-grammar-2))
> (test-while "begin x := foo; y := bar; z := uu; end")
(compound-command
   ((assign-command x (var-expression foo))
    (assign-command y (var-expression bar))
    (assign-command z (var-expression uu))))
```

The `compound-command` has 1 data field, which contains a *list* of the commands inside the begin-end.

Another example

This will illustrate the use of `arbno`. We'll use `sllgen:automaton-2` as our lexical spec. The grammar is

```
(define sllgen:grammar-2
  '((expression
      (number)
      lit-exp)
    (expression
      (identifier)
      var-exp)
    (expression
      (let (arbno declaration) in expression)
      let-exp)
    (expression
      (lparen expression (arbno expression) rparen)
      app-exp)
    (declaration
      (identifier = expression)
      decl)))

(define sllgen:test-2 '*)

(define sllgen:gen-2
  (lambda ()
    (set! sllgen:test-2
      (sllgen:string->tree sllgen:automaton-2 sllgen:grammar-2))))


; > (sllgen:gen-2)
; > (sllgen:test-2 "let x = (+ y z) u = (f v) in (g x y z)")
; (let-exp
;    ((decl x (app-exp (var-exp +) ((var-exp y) (var-exp z))))
;     (decl u (app-exp (var-exp f) ((var-exp v)))))
;    (app-exp (var-exp g) ((var-exp x) (var-exp y) (var-exp z))))
```