Attributes

Additional information about a token's lexeme.

- Stored in variable yyval
- Type of attributes (usually a union) specified by YYSTYPE
- Additional variables:
  - yytext: Lexeme (Actual text string)
  - yyleng: length of string in yytext
  - yylineno: Current line number (number of \n seen thus far)
- * enabled by %option yylineno

Examples

| for  | Sequence of f, o, r |
| "||" | C-style OR operator (two vert, bars) |
| .*  | Sequence of non-newline characters |
| "*/" | Sequence of characters except * and / |
| "[" ]" | Sequence of non-quote characters |
| (letter)"."(letter)|(digit)|"." | C-style identifiers |

A Complete Example

```c
#include <stdio.h>
#include "tokens.h"
%
digit [0-9]
hexdigit [0-9a-f]
%
"."{digit}+ { return(PLUS); }
"-"{digit}+ { return(MINUS); }
{digit}{digit}+ { return(INTEGER_CONSTANT); }
{digit}+{digit}+ { return(FLOAT_CONSTANT); }
. { return(SYNTAX_ERR); }
%
```
Implementing a Scanner

transition : state × Σ → state

algorithm scanner() { 
currentstate = start state;
while (1) {
c = getc(); /* on end of file, */
if defined(transition(currentstate, c))
currentstate = transition(currentstate, c);
else
  return s;
}
}

Implementing a Scanner (contd.)

Implementing the transition function:

- Simplest: 2-D array.
  Space inefficient.
- Traditionally compressed using row/column equivalence.
  (default on (f)lex)
  Good space-time tradeoff.
- Further table compression using various techniques:
  — Example: RDM (Row Displacement Method):
    Store rows in overlapping manner using 2 1-D arrays.
    Smaller tables, but longer access times.

Priority of matching

What if an input string matches more than one pattern?

"if" { return(TOKEN_IF); }
(letter)* { return(TOKEN_ID); }
"while" { return(TOKEN_WHILE); }

- A pattern that matches the longest string is chosen.
  Example: if1 is matched with an identifier, not the keyword if.
- Of patterns that match strings of same length, the first (from
  the top of file) is chosen.
  Example: while is matched as an identifier, not the keyword while.

Constructing Scanners using (f)lex

- Scanner specifications: specifications, 1

(f)lex

specifications, 1 → lex, yy, c

- Generated scanner in lex yy, c

(g)cc

lex yy, c → executable

- yywrap(): hook for signalling end of file.
- Use -lf1 (flex) or -ll (lex) flags at link time to include
default function yywrap() that always returns 1.
Grammars

The syntactic structure of a language is defined using grammars.

- Grammars (like regular expressions) specify a set of strings over an alphabet.
- Efficient recognizers (like DFA) can be constructed to efficiently determine whether a string is in the language.
- Language hierarchy:
  - Finite Languages (FL)
    - Enumeration
  - Regular Languages (RL ⊆ FL)
    - Regular Expressions
  - Context-free Languages (CFL ⊆ RL)

Context-free Grammars

- Terminal Symbols: Tokens
- Nonterminal Symbols: set of strings made up of tokens
- Productions: Rules for constructing the set of strings associated with nonterminal symbols,
  - Example: $Smt \rightarrow \text{while } Expr \text{ do } Stmt$

Start symbol: nonterminal symbol that represents the set of all strings in the language.

Lexical Analysis: A Summary

Convert a stream of characters into a stream of tokens.

- Make rest of compiler independent of character set
- Strip off comments
- Recognize line numbers
- Ignore white space characters
- Process macros (definitions and uses)
- Interface with symbol (name) table.

Parsing

A.k.a, Syntax Analysis

- Recognize sentences in a language.
- Discover the structure of a document/program.
- Construct (implicitly or explicitly) a tree (called a parse tree) to represent the structure.
- The above tree is used later to guide translation.
The Two Sides of Grammars

Specify a set of strings in a language.

Recognize strings in a given language:
- Is a given string $x$ in the language?
  - Yes, if we can construct a derivation for $x$
- Example: Is $id + id \in \mathcal{L}(E)$?
  
  $id + id \iff E + id$
  $\iff E + E$
  $\iff E$

Derivation

$E$ derives $id + id$:

$E \iff E + E$
$\iff E + id$
$\iff id + id$

- $\alpha A \beta \iff \alpha \gamma \beta$ iff $A \rightarrow \gamma$ is a production in the grammar.
- $\alpha \xrightarrow{\dagger} \beta$ if $\alpha$ derives $\beta$ in zero or more steps.
  - Example: $E \xrightarrow{\dagger} id + id$
- Sentence: A sequence of terminal symbols $w$ such that $S \xrightarrow{\dagger} w$ (where $S$ is the start symbol)
- Sentential Form: A sequence of terminal/nonterminal symbols $\alpha$ such that $S \xrightarrow{\dagger} \alpha$

Example

$E \Rightarrow E + E$
$E \Rightarrow E - E$
$E \Rightarrow E * E$
$E \Rightarrow E / E$
$E \Rightarrow (E)$
$E \Rightarrow id$

$\mathcal{L}(E) = \{id, id + id, id - id, \ldots, id + (id * id) - id, \ldots\}$

Context-free Grammars

Production: rule with nonterminal symbol on left-hand side, and a (possibly empty) sequence of terminal or nonterminal symbols on the right-hand side.

Notations:
- Terminals: lower case letters, digits, punctuation
- Nonterminals: Upper case letters
- Arbitrary Terminals/Nonterminals: $X, Y, Z$
- Strings of Terminals: $u, v, w$
- Strings of Terminals/Nonterminals: $\alpha, \beta, \gamma$
- Start Symbol: $S$
Ambiguity

A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

Example: id + id * id

![Parse Trees](image)

Disambiguation

Express Preference for one parse tree over others.

Example: id + id * id

The usual precedence of * over + means:

![Preferred Parse Tree](image)

Derivations

- Rightmost derivation: Rightmost nonterminal is replaced first:
  
  \[ E \Rightarrow E + E \]
  
  \[ \Rightarrow E + id \]
  
  \[ \Rightarrow id + id \]

  Written as \( E \xrightarrow{\text{rm}} id + id \)

- Leftmost derivation: Leftmost nonterminal is replaced first:
  
  \[ E \Rightarrow id + E \]
  
  \[ \Rightarrow id + id \]

  Written as \( E \xrightarrow{\text{lm}} id + id \)

Parse Trees

Graphical Representation of Derivations

\[
\begin{align*}
E & \Rightarrow E + E & E & \Rightarrow E + E \\
\Rightarrow id + E & & \Rightarrow id + id \\
\Rightarrow id + id & & \Rightarrow id + id
\end{align*}
\]

A Parse Tree succinctly captures the structure of a sentence,