컴파일러 입문

Miscellaneous

목 차

Symbol Table [1/3]

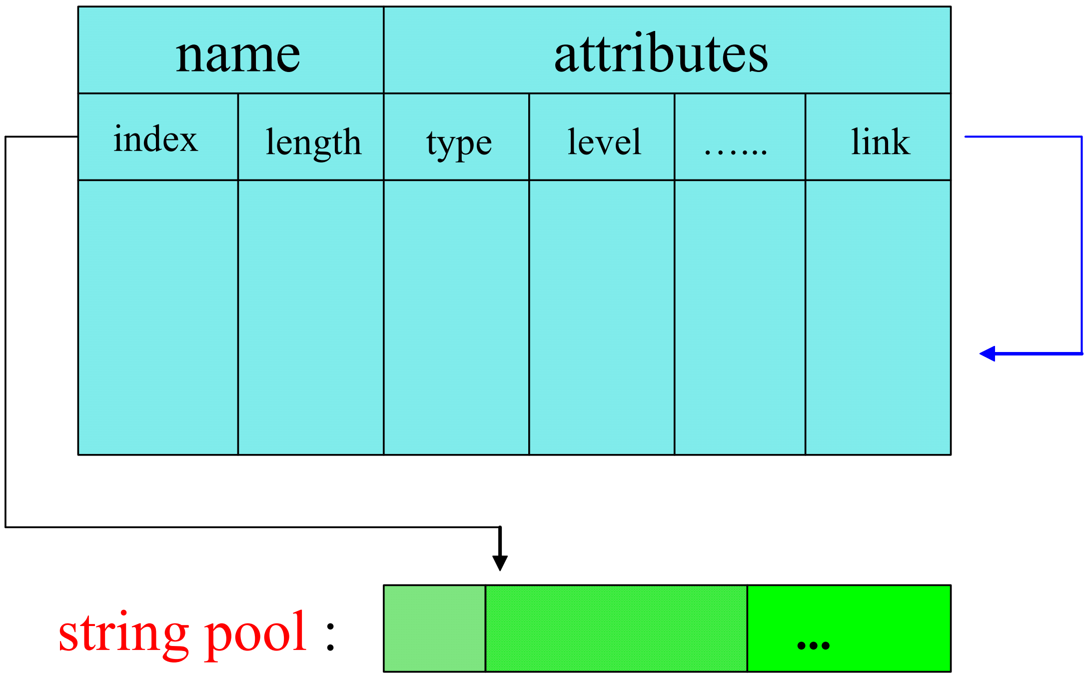
- Symbol tables(also called identifier tables or name tables) assist two important functions in the translation process: in checking semantic correctness and aiding in the proper generation of code. Both of these functions are achieved by inserting into, and retrieving from the symbol table, attributes of the variables used in the source program. These attributes are usually found explicitly in declarations or more implicitly through the context in which the variable name appears in the program.

- Symbol table actions :

  - insert
  - search(lookup)
  - delete

Symbol Table [2/3]

- Symbol table entries



- Attributes appearing in a symbol table are dependent on the **usage** of the symbol table.

Symbol Table [3/3]

- **Stack-Implemented Hash-Structured Symbol Table**

Text p. 534

- hash bucket
- symbol table
- level table

- set operation
- reset operation

YACC

Yet Another Compiler-Compiler

Stephen C. Johnson

July 31, 1978

목 차

Introduction [1/3]

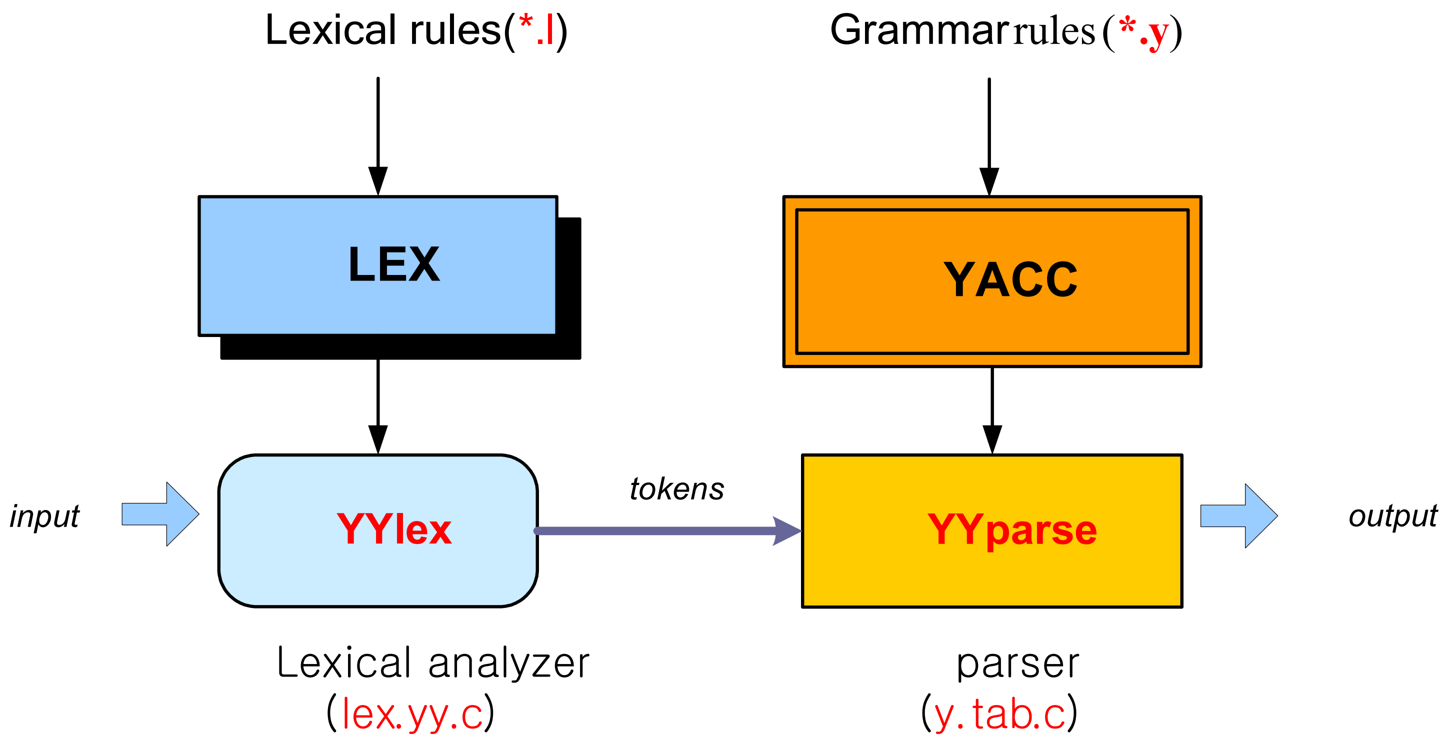
- Yacc provides a general tool for imposing structure on the input to a computer program.

cfg + actions → Yacc → C program

  - The class of specifications accepted is very general one, that is, LALR(1) grammar with disambiguating rules.
  - Base language : C

Introduction [2/3]

- **Model for Lex and Yacc**



Introduction [3/3]

- **lexical analysis**
  - the user must supply a lexical analyzer to read the input stream and communicate tokens(with values, if desired) to the parser.
  - A very useful tool for constructing lexical analyzer is lex.
    - token number : yylex() return value
    - token value : yylval(external variable).
- **Parser Actions**
  - An LR Parser : shift, reduce, accept, and error.
    - when a shift takes place, the parser calls a lexical analyzer to get a token and the external variable yylval is copied onto the value stack.
    - when a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. After return from the user code, the reduction is carried out.
  - In addition to the stack holding states, the value stack running in parallel with it holds the values from the lexical analyzer and the actions associated with rules.

Input Specification [1/2]

- **format** :

```
declarations
%%
rules
%%
programs
```

- **The declaration section**
  - It is optional part.
  - %token declares names representing tokens.
    ex) %token name1 name2 ...
  - %start declares the start symbol explicitly. By default, the start symbol is taken to be the left hand side of the first production rule in the rules section.

Input Specification [2/2]

- **The rule section**

  Form : A : RHS {action code} ;

  where, A : Left Hand Side of a production,
  RHS : Right Hand Side of a production rule,
  action code : C statements.

- The **program section** is copied into the **generated** program.

Rule section [1/4]

- grammar rules + actions
  - With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process.
- Grammar Rule Description

  \- form : A : RHS

  where, A : a nonterminal symbol,
  RHS : a sequence of names and literals.

  ex) BNF <expression> ::= <expression> + <term> ¦ <term>
  YACC expression : expression '+' term ¦ term

Rule section [2/4]

- A **literal** consists of a character enclosed in single quote "'" . As in C, all escape sequences are recognized.

  ex) '\n' newline '\b' backspace '\t' tab

  '\ooo' ooo in octal '\\' backslash

- The **names** used in the RHS of a grammar rule may represent **tokens** or **nonterminal** symbols. Names may be of arbitrary length, and may be made up of letters, dot "." , underscore "_" , and noninitial digits. Uppercase and lowercase letters are distinct.

- The vertical bar "¦" can be used to avoid rewriting the left hand side.

  ex)

```
A : B C D ;                A : B C D
A : E F ;     <------->        ¦ E F
A : G ;                        ¦ G ;
```

- **ε -production**

  ex) A -> ε <===> YACC A : ;

Rule section [3/4]

- **Action Description**
  - An action is an arbitrary C statements enclosed in curly braces { and }.

```
ex) expression : expression '+' term
                 { printf("addition expression detected\n"); }
                 ;
    expression : term
                 { printf("simple expression detected\n"); }
                 ;
```

  - In a real parser, these actions can be used to construct the parse tree(syntax tree) or to generate code directly.
  - YACC permits an action to be written in the middle of a rule as well as at the end.
  - YACC parser uses only names beginning with yy; the user should avoid such names.

Rule section [4/4]

- YACC provides a facility for associating values with the symbols used in a grammar rule.
  - **$$**, **$1**, **$2**, … represent the **values** of each grammar symbol.
  - Values can be passed to other grammar rules by performing an assignment in the action part to the pseudo variable **$$**.
- Parse tree construction
  - **node**(L,n1,n2) creates a node with label L, with the descendants n1 and n2, and returns the index of the newly created node.

```
ex) expr : expr '+' expr   { $$ = node('+',$1,$3); }
```

Ambiguity and Conflicts

- **Ambiguity**
  - A set of grammar rules is <u>ambiguous</u> if there is some input string that can be structured in **two** or more different ways.
- **Conflicts**
  - shift/reduce, reduce/reduce
  - Yacc invokes two disambiguating rules by default:
    - In a shift/reduce conflict, the default is to do the shift.
    - In a reduce/reduce conflict, the default is to do reduce by the **earlier** grammar rule.

Precedence

- %left, %right, %nonassoc

```
ex) %right '='
    %left '+' '-'
    %left '*' '/'
    %%
    expr  :  expr  '='  expr
          |  expr  '+'  expr
          |  expr  '-'  expr
          |  expr  '*'  expr
          |  expr  '/'  expr
          ;

           a = b = c * d - e - f * g
     <-->  a = (b = (((c * d) - e) - (f * g)))
```

Error Handling

- It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors.

- The token name error is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token “error“ is legal.

Example [1/4]

- **Problem**: a rudimentary desk calculator operating on integer values.
- **calc.l**

```
%{
/* LEX source for calculator program */
%}
%%
[ \t]        ;   /* ignore blanks and tabs */
[0-9]+       {yylval = atoi(yytext); return NUMBER;}
"mod"        return MOD;
"div"        return DIV;
"sqr"        return SQR;
 \n ¦ .      return yytext[0];  /* return everything else */
```

Example [2/4]

- execution sequence

```
% lex calc.l
% yacc calc.y
% cc y.tab.c -ll -o calc
% calc
1+1
2
3+4*5
23
(3+4)*5
35
sqr sqr 2+3
19
25 mod 7
4
(3))
syntax error
Try again
↑C
%
```

Example [3/4]

- **calc.y**

```
%{
/* YACC source for calculator program */
# include <stdio.h>
%}
%token NUMBER DIV MOD SQR
%left '+' '-'
%left '*' DIV MOD
%left  SQR
%%
comm : comm '\n'
       ¦ lambda
       ¦ comm expr  '\n' {printf("%d\n", $2);}
       ¦ comm error '\n' {yyerrok; printf(" Try again \n");}
       ;
expr   : '(' expr ')'  {$$ = $2;}
       ¦ expr '+' expr {$$ = $1 + $3;}
       ¦ expr '-' expr {$$ = $1 - $3;}
       ¦ expr '*' expr {$$ = $1 * $3;}
```

If an error is detected in the parse, the parser skips to a newline character, the error status is reset(**yyerrok**) and an appropriate message is output.

Example [4/4]

```
      ¦ expr MOD expr          {$$ = $1 % $3;}
      ¦ SQR  expr              {$$ = $2 * $2;}
      ¦ NUMBER
  ;
lambda: /* empty */
  ;
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{
  printf("%s\n", s);
}
main()
{
  return yyparse();
}
```