Dept. of Computer Science at Sogang University

COMPILER (CSE 4120)

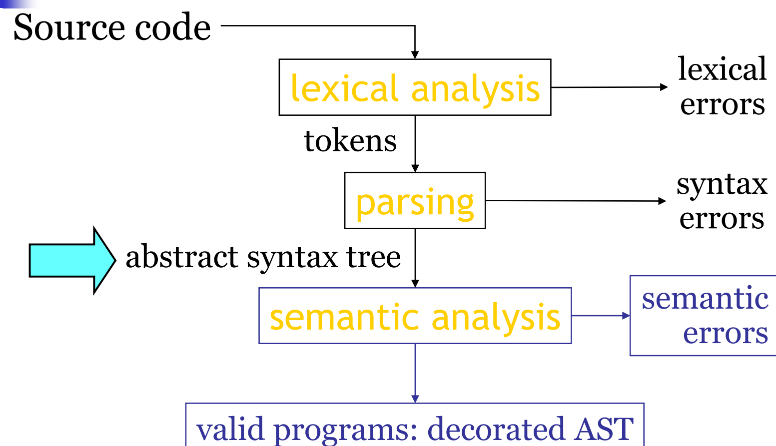
(Lecture 7: Building Abstract Syntax Tree)

Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

Semantic Analysis



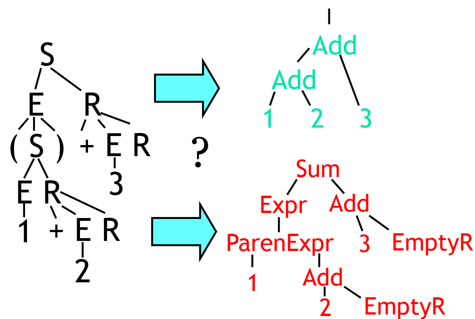
Concrete Parse Trees

- Represent the concrete syntax of the source language
- Inconvenient to use directly
  - Many of punctuation tokens are redundant and convey no information
    - Useful in the input string, but once the parse tree is built, the structure of the tree conveys the structuring information more conveniently
  - The structure of the parse tree depends on too much on the grammar
    - Left-factoring, elimination of left-recursion, elimination of ambiguity introduce extra nonterminal symbols and extra grammar productions
  - These details should be confined to the parsing phase and should not clutter the semantic analysis phase
    - ***Abstract Syntax*** is used for separating these two phases

Concrete Parse Trees (An Example)

$$S \rightarrow E\,R$$
$$R \rightarrow \varepsilon \mid +\,E\,R$$
$$E \rightarrow \text{num} \mid (\,S\,)$$

(1 + 2) + 3



Abstract Syntax Tree

- Abstract Syntax are tree data structure
- Advantages:
  - We get rid of improper symbols of the source program (concrete syntax), only token with semantics values remains.
  - Parse trees depend too much on the grammar (intermediate non-terminal) whereas abstract syntax has not these intermediate node.
- Abstract syntax is the interface between parsing and semantics analysis
- Abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation

Syntax of the TINY Language

program → *stmt-sequence*
stmt-sequence → *stmt-sequence* **;** *statement* | *statement*
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* **end**
| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier** **:=** *exp*
read-stmt → **read identifier**
write-stmt → **write** *exp*
exp → *simple-exp comparison-op simple-exp* | *simple-exp*
comparison-op → **<** | **=**
simple-exp → *simple-exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → * | **/**
factor → **(** *exp* **)** | **number** | **identifier**

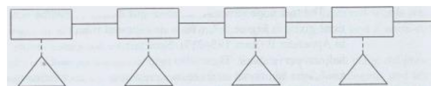
Syntax Tree Structure for the Tiny Compiler

- C declarations for a TINY syntax tree node

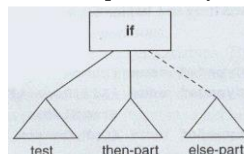
```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
   } TreeNode;
```


Syntax Tree Structure for the Tiny Compiler

- Visual description of the syntax tree structure
  - Rectangular boxes indicate statement nodes
  - Round and oval boxes indicate expression nodes
  - A sequence of statements connected by sibling fields

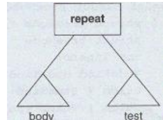


  - An if-statement (with potentially three children)

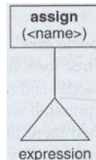


Syntax Tree Structure for the Tiny Compiler

- Visual description of the syntax tree structure (Cont'd)
  - A repeat-statement

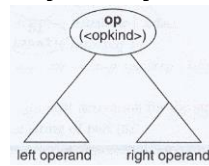


  - An assign-statement



- Visual description of the syntax tree structure (Cont'd)
  - A write-statement

  - An operator expression



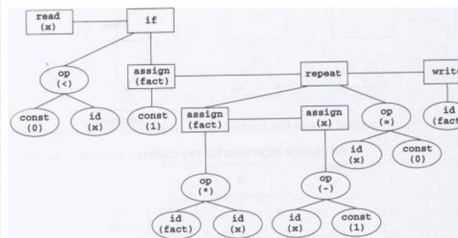
Dept. of Computer Science at Sogang University

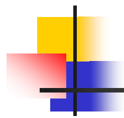
Syntax Tree Structure for the Tiny Compiler

- Sample program in the TINY language

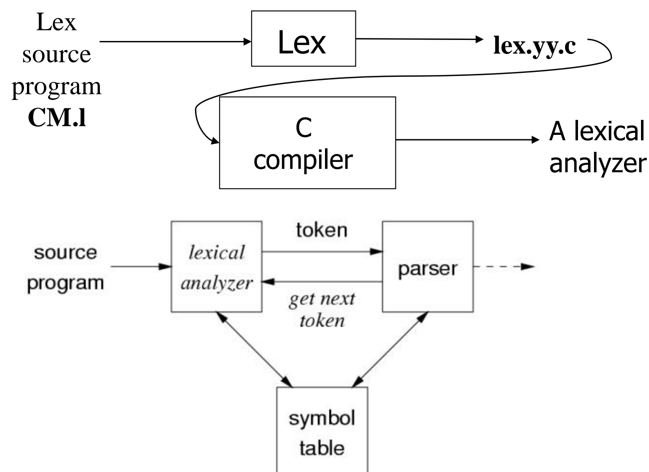
```
{ Sample program
  in TINY language-
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

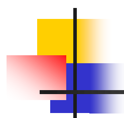
- Syntax tree for the sample TINY program





Interaction of Lexical Analyzer with Parser



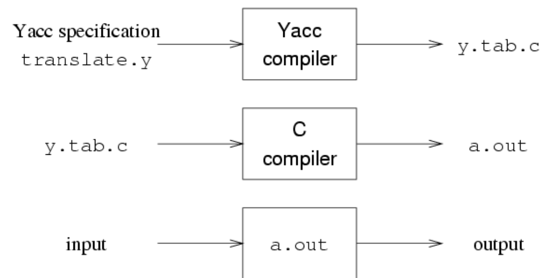


Parser Generators

- Yacc: LALR parser generator
  - yacc *filename*.y generates **y.tab.c, ytab.c,** or **<filename>.tab.c**
  - To use (f)lex with yacc, one specifies the –d option to yacc to instruct it to generate the “y.tab.h” containing definitions of all the “%tokens” appearing in the yacc input
    - **yacc -d filename.y → y.tab.c, y.tab.h(or ytab.h)**
    - insert the following statement into the definitions section of your lex source: **#include "y.tab.h"**
    - If the file ***y.tab.h*** is included in the **lex** source, then **yacc** must be run before the **lex**-generated file is compiled.
  - verbose option: -v flag
    - yacc -v filename.y → **y.output**
    - **y.output** contains a textual description of the LALR(1) parsing table
      - **An example: Figure 5.12 in pp 233 and 234 of the textbook**

Parser Generators

- cc y.tab.c –ly



Compiling

- The file *y.tab.c* produced by **yacc** can be compiled with **cc** as follows:
  - **cc y.tab.c -ly**
  - The **-ly** option causes the object file generated to be linked with the **yacc** library.
  - If you supply your own versions of **main**() and **yyerror**(), link the module containing these routines before the library.
- A routine named **yylex**() must be provided to do lexical analysis and return tokens to the parser. This routine can be supplied by the user or generated from a **lex** specification. A lexical analyzer generated **lex** can be linked to the parser using the following command line:
  - **cc y.tab.c lex.yy.c -ly -ll**
- *lex.yy.c* is the C file that **lex** produces. If you are using the library version of *main*(), the **-ly** option must appear before **-ll**, so that the version in the **yacc** library is used.

Yacc Source Program
declarations
%%
translation rules
%%
supporting C-routines

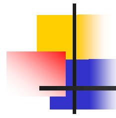
Yacc Basic

- Declarations
  - C declarations delimited by %{ and %}
  - Declarations used by the translation rules or procedures
    - ```
      %{
          #include <ctype.h>
          #include <stdio.h>
      %}
      %token DIGIT | PLUS | MINUS | TIMES |
      %left PLUS MINUS /* same precedence and left associative */
      %left TIMES
      ```

Yacc Source Program
declarations
%%
translation rules
%%
supporting C-routines

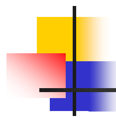
Yacc Basic

- Translation rules
  - Rules consists of a grammar: $A \rightarrow r_1 \mid r_2 \mid \ldots \mid r_n$
  - Yacc-input format
    - ```
      A  :  r1  {semantic action 1}
         |  r2  {semantic action 2}
        ...
         |  rn  {semantic action n}
         ;
      ```
    - Semantic actions: fragments of C program code attached to grammar productions
- Supporting C-routines
  - yylex(): lexical analyzer (mandatory)

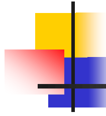


Semantic Actions

- Special Symbols: $$ $i
  - $$: the attribute value associated with the left *nonterminal*
  - $i: the value associated with the *i*th grammar symbol (terminal or nonterminal)



```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp       { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp     : exp '+' term   {$$ = $1 + $3;}
        | exp '-' term   {$$ = $1 - $3;}
        | term    {$$ = $1;}
        ;

term    : term '*' factor {$$ = $1 * $3;}
        | factor  {$$ = $1;}
        ;

factor        : NUMBER        {$$ = $1;}
              | '(' exp ')'   {$$ = $2;}
              ;

%%

main()
{ return yyparse();
}
```

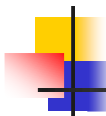


Dept. of Computer Science at Sogang University

Generation of a TINY Parser Using Yacc

- Semantic actions associated with each of the grammar rules
  - Actions represent the construction of the syntax tree corresponding to the parse tree at that point
    - New nodes need to be allocated by calls to **newStmtNode** and **newExpNode** from **util** package
    - Appropriate child nodes of new tree node need to be assigned
    - Example:
      - **write_stmt : WRITE exp**

```
write_stmt  :  WRITE exp
                    { $$ = newStmtNode(WriteK);
                      $$ → child[0] = $2;
                    }
              ;
```

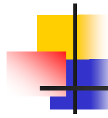
Dept. of Computer Science at Sogang University

Generation of a TINY Parser Using Yacc

```
/* Function newStmtNode creates a new statement
 * node for syntax tree construction
 */
TreeNode * newStmtNode(StmtKind kind)
{ TreeNode * t = (TreeNode *malloc(sizeof(TreeNode));
  int i;
  if (t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->nodekind = StmtK;
    t->kind.stmt = kind;
    t->lineno = lineno;
  }
  return t;
}
```

```
/* Function newExpNode creates a new expression
 * node for syntax tree construction
 */
TreeNode * newExpNode(ExpKind kind)
{ TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
  int i;
  if (t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->nodekind = ExpK;
    t->kind.exp = kind;
    t->lineno = lineno;
    t->type = Void;
  }
  return t;
}
```

Dept. of Computer Science at Sogang University



Generation of a TINY Parser Using Yacc

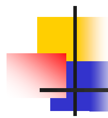
- Semantic actions associated with each of the grammar rules (Cont'd)
  - Example:
    - **program : stmt_seq**

```
program : stmt_seq
            { savedTree = $1;};
```

    - **assign_stmt**

```
assign_stmt : ID {  savedName = copyString(tokenString);
                    savedLineNo = lineno;  }
              ASSIGN exp
                  { $$ = newStmtNode(AssignK);
                    $$ → child[0] = $4;
                    $$ → attr.name = savedName;
                    $$ → lineno = saveLineNo; };
```


Dept. of Computer Science at Sogang University



Generation of a TINY Parser Using Yacc

- Semantic actions associated with each of the grammar rules (Cont'd)
  - Example

```
exp  :  simple_exp LT simple_exp
          { $$ = newExpNode(OpK);
            $$ → child[0] = $1;
            $$ → child[1] = $3;
            $$ → attr.op = LT;
          }
       | simple_exp EQ simple_exp
          { $$ = newExpNode(OpK);
            $$ → child[0] = $1;
            $$ → child[1] = $3;
            $$ → attr.op = EQ;
          }
```