

LAB 9 – Recursive Descent Parser Program

An Advanced Introduction to Unix/C Programming



Dennis
Ritchie



Ken
Thompson



Linus
Torvalds



Richard
Stallman



Brian
Kernighan

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands

Scanner / Parser

Scanner reads one character at a time and returns a token.

Parser reads one token at a time and matches the token with the context free grammar.

TOKEN

Scanner reads one character at a time and returns a token.

A token is defined as:

```
typedef union
{
    long    integral;
    double  floating_point;
    char    *string;
    char    op;
} TOKEN_VALUE;
```

```
typedef struct token {
    TOKEN_TYPE    type;    // IDENTIFIER_TOKEN, INT_TOKEN, LPAREN_TOKEN, ADDOP_TOKEN...
    TOKEN_VALUE    val;    // token->val.op, token->val.string, token->val.integral, ...
} TOKEN;
```

AST Node

Parser reads one token at a time and matches the token with the grammar.

```
typedef struct node
{
    NODE_TYPE    type;    // PROGRAM_NODE, ASSIGN_STMT_NODE, NUMBER_NODE
    union
    {
        char      *identifier;    // firstVar
        NUMBER     number;        // type=INT_TYPE, value=100
        char       op;            // +, -, *, /, %
    } data;
    struct node    *leftChild;    // Points to left child AST Node
    struct node    *rightChild;  // Points to right child AST Node
} NODE;
```

input.txt

```
john@oho:~/LAB9.1line$ cat input.txt
firstVar = 100;
print (firstVar);
second2var = 0.15;
print (second2var);
repeat (2)
    firstVar = firstVar + 1;
print (+firstVar);
repeat(10)
    print (-101.725) ;
```

Grammar

```
<program> ::= <statement> <EOF> |  
            <statement> <program>  
  
<statement> ::= <assignStmt> | <printStmt> |  
               <repeatStmt>  
<assignStmt> ::= <ident> = <expr> ;  
<printStmt> ::= print <expr> ;  
<repeatStmt> ::= repeat ( <expr> ) <statement>  
  
<expr> ::= <term> | <expr> <addOp> <term>  
<term> ::= <factor> | <term> <multOp> <factor>  
<factor> ::= <ident> | <number> | <addOp> <factor> |  
            ( <expr> )  
  
<ident> ::= <letter> | <ident> <letter> | <ident> <digit>
```

```
<number> ::= <integer> | <float>  
  
<integer> ::= <digit> | <integer> <digit>  
<float> ::= <digit> . | <digit> <float> | <float> <digit>  
  
<addOp> ::= + | -  
<multOp> ::= * | / | %  
  
<letter> ::= a-z | A-Z | _ | $  
<digit> ::= 0-9
```

Grammar Used for: firstVar = 100;

<program> ::= <statement> <EOF>

<statement> ::= <assignStmt>

<assignStmt> ::= <ident> = <expr> ;

<expr> ::= <term>

<term> ::= <factor>

<factor> ::= <number>

<ident> ::= <letter> | <ident> <letter>

<number> ::= <integer>

<integer> ::= <digit> | <integer> <digit>

<letter> ::= a-z | A-Z | _ | \$

<digit> ::= 0-9

Grammar used for firstVar = 100;

cat 1.txt; run

```
john@oho:~/LAB9.1line$ ls
1.txt  eval.c  eval_test.c  parse.c  print.c  print_test.c  sample_code.c  scan.h
a.out  eval.h  input.txt    parse.h  print.h  run           scan.c
```

```
john@oho:~/LAB9.1line$ more run
gcc -g parse.c scan.c eval.c eval_test.c
```

```
john@oho:~/LAB9.1line$ cat 1.txt
firstVar = 100;
```

```
john@oho:~/LAB9.1line$
```


parse.h

scan.h

```
john@oho:~/LAB9.CODE$ cat parse.h
#ifndef __PARSE_H
#define __PARSE_H

#include <stdio.h>
#include <stdlib.h>
#include "scan.h"

typedef enum
{
    PROGRAM_NODE,
    STATEMENT_NODE,
    ASSIGN_STMT_NODE,
    REPEAT_STMT_NODE,
    PRINT_STMT_NODE,
    EXPR_NODE,
    TERM_NODE,
    FACTOR_NODE,
    IDENT_NODE,
    NUMBER_NODE
} NODE_TYPE;

typedef enum
{
    INT_TYPE,
    FLOAT_TYPE
} NUMBER_TYPE;

typedef union
{
    long integral;
    double floating_point;
} NUMBER_VALUE;

typedef struct
{
    NUMBER_TYPE type;
    NUMBER_VALUE value;
} NUMBER;
```

```
typedef struct node
{
    NODE_TYPE type;
    union
    {
        char *identifier;
        NUMBER number;
        char op;
    } data;
    struct node *leftChild;
    struct node *rightChild;
} NODE;

TOKEN *getNextToken(TOKEN
**currToken);

NODE *program();
NODE *statement();
NODE *assignStmt(TOKEN
**currToken);
NODE *repeatStmt(TOKEN
**currToken);
NODE *printStmt(TOKEN
**currToken);
NODE *expr(TOKEN **currToken);
NODE *term(TOKEN **currToken);
NODE *factor(TOKEN **currToken);
NODE *ident(TOKEN **currToken);
NODE *number(TOKEN
**currToken);

void freeParseTree(NODE **node);

void error(char *errorFormat, ...);
```

```
john@oho:~/LAB9.CODE$ cat scan.h
#ifndef __SCAN_H
#define __SCAN_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef enum
{
    NO_TOKEN_TYPE, // 0
    INVALID_TOKEN, // 1
    REPEAT_TOKEN, // 2
    PRINT_TOKEN, // 3
    IDENT_TOKEN, // 4
    INT_TOKEN, // 5
    FLOAT_TOKEN, // 6
    ASSIGNMENT_TOKEN, // 7
    LPAREN_TOKEN, // 8
    RPAREN_TOKEN, // 9
    ADD_OP_TOKEN, // 10
    MULT_OP_TOKEN, // 11
    SEMICOLON_TOKEN, // 12
    EOF_TOKEN // 13
} TOKEN_TYPE;

typedef union
{
    long integral;
    double floating_point;
    char *string;
    char op;
} TOKEN_VALUE;

typedef struct token
{
    TOKEN_TYPE type;
    TOKEN_VALUE val;
} TOKEN;

TOKEN *scanner();
void freeToken(TOKEN **);
void printToken(TOKEN **);
void ungetToken(TOKEN **);

#define BUF_SIZE 128
```

main.c

```
#include "eval.h"

int main(int argc, char **argv)
{
    freopen(argv[1], "r", stdin);

    NODE *fullProgram = program();      ← Call program();
    printf("Done parsing...\n");
    evalProgram(fullProgram);
    freeParseTree(&fullProgram);
    cleanUpSymbolTables();

    exit(0);
}
```

TOKEN *getNextToken

```
TOKEN *getNextToken(TOKEN **token)
{
    freeToken(token);
    TOKEN* tok = scanner();
    // uncomment the line below if desired for debugging purposes.
    printToken(&tok); fflush(stdout);
    return tok;
}
```

NODE *program()

```
NODE *program()                                // <program> ::= <statement> <EOF>
{
    printf("ENTER: program()\n");
    NODE *node = calloc(sizeof(NODE), 1);
    node->type = PROGRAM_NODE;
    node->leftChild = statement();
    if (node->leftChild != NULL)
    {
        printf("program(): node->rightChild = program();\n");
        node->rightChild = program();
    }
    else
    {
        printf("program(): leftChild = free(node); node=NULL\n");
        free(node);
        node = NULL;
    }
    return node;
}
```

NODE *statement()

```
NODE *statement()                                // <statement> ::= <assignStmt>
{
    TOKEN *token = getNextToken(NULL);
    if (token == NULL) {
        return NULL;
    }
    else if (token->type == EOF_TOKEN) {
        freeToken(&token);
        return NULL;
    }

    NODE *node = calloc(sizeof(NODE), 1);
    node->type = STATEMENT_NODE;

    switch(token->type) {
        case IDENT_TOKEN:
            node->leftChild = assignStmt(&token);
            break;
        default: // see the TOKEN_TYPE enum to see which numbers mean what...
            error("60 Invalid token at start of statement : ");
            printToken(&token);
            fflush(stdout);
    }
    return node;
}
```

NODE *assignStmt()

```
NODE *assignStmt(TOKEN **currToken)                // <assignStmt> ::= <ident> = <expr>;
{
    NODE *node = calloc(1, sizeof(NODE));
    node->type = ASSIGN_STMT_NODE;
    node->leftChild = ident(currToken);

    if ((*currToken)->type != ASSIGNMENT_TOKEN) {    // Must be an equal sign, =
        error("Missing ASSIGNMENT_TOKEN, =.");
    }

    *currToken = getNextToken(currToken);
    node->rightChild = expr(currToken);

    *currToken = getNextToken(currToken);
    if ((*currToken)->type != SEMICOLON_TOKEN) {      // Must be a semicolon, ;
        error("Missing SEMICOLON_TOKEN in assignment statement.");
    }
    freeToken(currToken);
    return node;
}
```

NODE *ident()

```
NODE *ident(TOKEN **currToken)
{
    NODE *node = calloc(1, sizeof(NODE));
    node->type = IDENT_NODE;

    if ((*currToken)->type != IDENT_TOKEN) {
        error("Identifier not found.");
    }

    node->data.identifier = strdup((*currToken)->val.string);

    printf("147 ident(): node->leftChild = %p\n", node->leftChild);
    printf("148 ident(): node->rightChild = %p\n", node->rightChild);

    *currToken = getNextToken(currToken);

    return node;
}
```

NODE *expr()

```
NODE *expr(TOKEN **currToken)    // <expr> ::= <term>
{
    // TODO
    NODE *node = calloc(1, sizeof(NODE));
    node->type = EXPR_NODE;
    node->leftChild = term(currToken);
    return node;
}
```


NODE *term()

```
NODE *term(TOKEN **currToken)
{
    // TODO
    NODE *node = calloc(1, sizeof(NODE));
    node->type = TERM_NODE;

    node->leftChild = factor(currToken);

    if ((*currToken)->type == MULT_OP_TOKEN) {
        node->data.op = (*currToken)->val.op;
        *currToken = getNextToken(currToken);
        node->rightChild = term(currToken);
    }

    return node;
}
```

NODE *factor()

```
NODE *factor(TOKEN **currToken)
{
    NODE *node = calloc(1, sizeof(NODE));
    node->type = FACTOR_NODE;

    node->leftChild = number(currToken);

    return node;
}
```

NODE *ident()

```
NODE *ident(TOKEN **currToken)
{
    NODE *node = calloc(1, sizeof(NODE));
    node->type = IDENT_NODE;

    if ((*currToken)->type != IDENT_TOKEN) {
        error("Identifier not found.");
    }

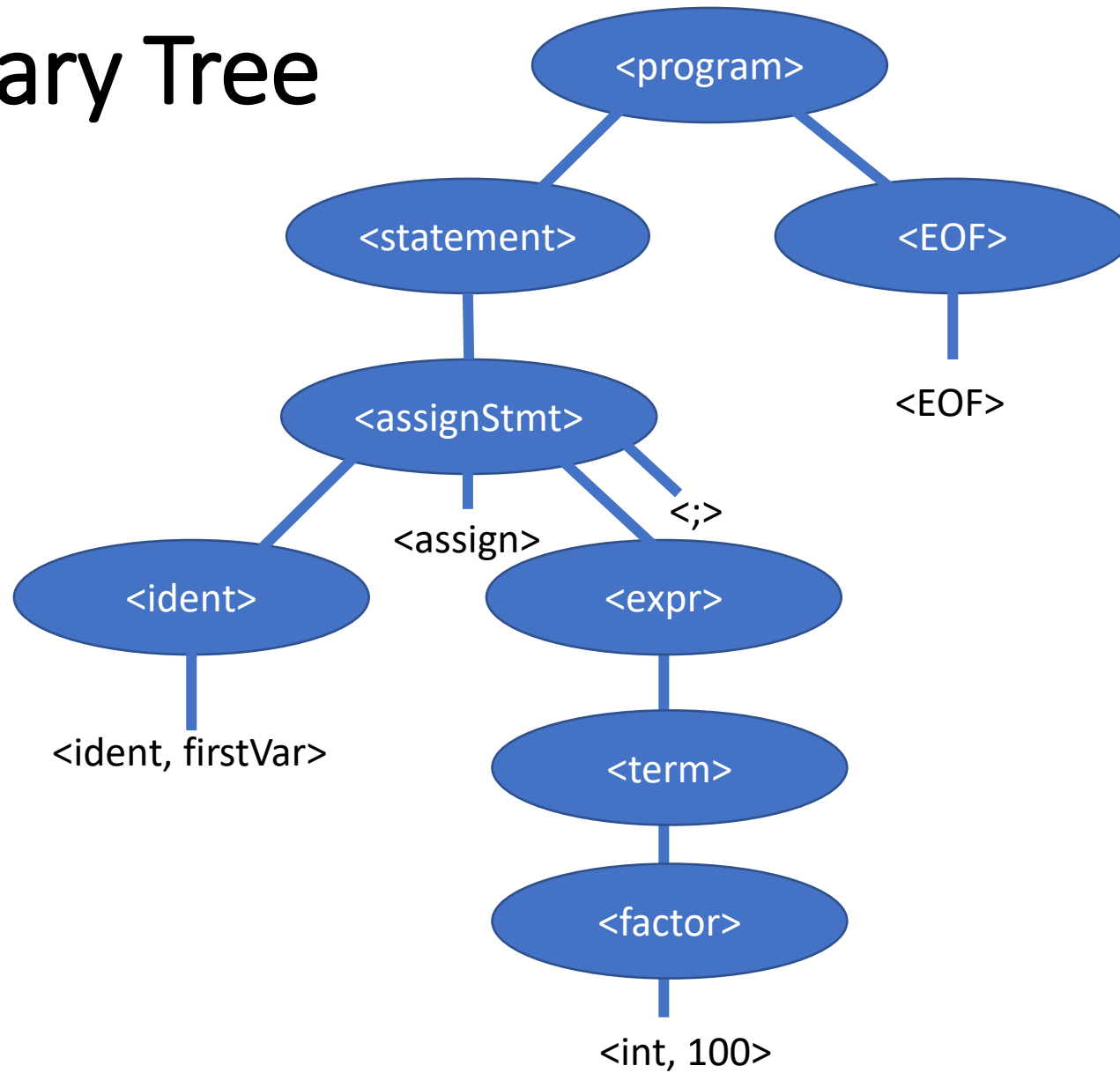
    node->data.identifier = strdup((*currToken)->val.string);

    printf("147 ident(): node->leftChild = %p\n", node->leftChild);
    printf("148 ident(): node->rightChild = %p\n", node->rightChild);

    *currToken = getNextToken(currToken);

    return node;
}
```

AST Binary Tree



a.out < 1.txt

```
john@oho:~/LAB9.1line$ cat 1.txt
firstVar = 100;

john@oho:~/LAB9.1line$ run; a.out < 1.txt
ENTER: program()
<IDENT, firstVar>
147 ident(): node->leftChild = (nil)
148 ident(): node->rightChild = (nil)
<ASSIGN>
<INT, 100>
<SEMICOLON>
program(): node->rightChild = program();
ENTER: program()
<EOF>
program(): leftChild = free(node); node=NULL
Done parsing...

john@oho:~/LAB9.1line$
```