Parser Implementation

Specification

Compiler Design Project 2

: implement a C-Minus parser using Bison

1. Use source code from C-Minus with LEX
2. Parses it with C-Minus grammar
3. Returns or prints abstract syntax tree(AST)

Basic Knowledge

Bison

- Input
    - CFG grammar
    - precedence and associativity
- Output
    - LALR(1) parser

C-Minus grammar

1. program → declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier ***ID ;*** | type-specifier ***ID[NUM] ;***
5. type-specifier → ***int*** | ***void***
6. fun-declaration → type-specifier ***ID (***params***)*** compound-stmt
7. params → param-list | ***void***
8. param-list → param-list ***,*** param | param
9. param → type-specifier ***ID*** | type-specifier ***ID[ ]***
10. compound-stmt → ***{*** local-declarations statement-list ***}***
11. local-declarations → local-declarations var-declarations | empty
12. statement-list → statement-list statement | empty
13. statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
14. expression-stmt → expression ***;*** | ***;***
15. selection-stmt → ***if (*** expression ***)*** statement | ***if (*** expression ***)*** statement ***else*** statement /* **Dangling Else problem */**
16. iteration-stmt → ***while (*** expression ***)*** statement
17. return-stmt → ***return ;*** | ***return*** expression ***;***
18. expression → var ***=*** expression | simple-expression
19. var → ***ID*** | ***ID [*** expression ***]***
20. simple-expression → additive-expression relop additive-expression | additive-expression

21. relop → ***<=*** | ***<*** | ***>*** | ***>=*** | ***==*** | ***!=***
22. additive-expression → additive-expression addop term | term
23. addop → ***+*** | ***-***
24. term → term mulop factor | factor
25. mulop → ***** | ***/***
26. factor → ***(***expression***)*** | var | call | ***NUM***
27. call → ***ID (***args***)***
28. args → arg-list | empty
29. arg-list → arg-list, expression | expression

Compilation Environment

Windows 11

Ubuntu 22.04.3 LTS

GNU/Linux 5.15.153.1-microsoft-standard-WSL2 ×86_64

Implementation

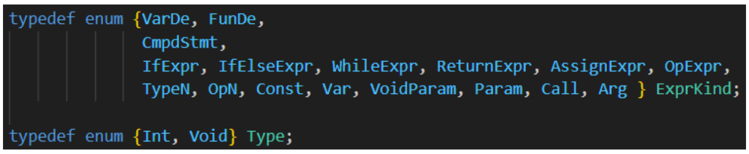
FILE `main.c`

- set `NO_ANALYZE` and `TraceParse` to TRUE, set `NO_PARSE` to FALSE

FILE `globals.h`

- Modify the syntax tree for parsing part
- Tiny parser had defined specific kinds of enumeration `NodeKind`, `StmtKind`, `ExpKind`, but I combined to one definition `ExprKind`. I put all kinds of non-terminals here.

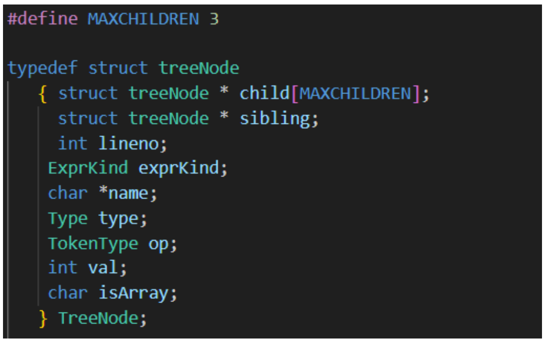
  For type checking, I changed `ExpType` according to C-Minus grammar `{int, void}`.



```
typedef enum {VarDe, FunDe,
              CmpdStmt,
              IfExpr, IfElseExpr, WhileExpr, ReturnExpr, AssignExpr, OpExpr,
              TypeN, OpN, Const, Var, VoidParam, Param, Call, Arg } ExprKind;

typedef enum {Int, Void} Type;
```

- Therefore I removed some attributes from `treeNode` struct, and untangled the union. I also add `char isArray` attribute to determine if the type of a variable is an array or not.



```
#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
    ExprKind exprKind;
    char *name;
    Type type;
    TokenType op;
    int val;
    char isArray;
   } TreeNode;
```

FILE `util.c`

- Modify `printTree` function to print C-Minus Syntax tree.

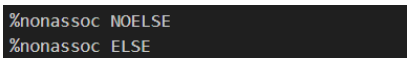
- According to the above, I removed `newStmtNode()` and `newExprNode()` functions and re-implemented `newTreeNode()` function to allocate and initialize new Node type according to the one I defined in `global.h` file. `isArray` is set to FALSE when the node is created.
- `void printNameAndType (TreeNode * tree)` function was created by separating the common part that prints the name and type. Here I checked `isArray` value and print whether it is an array or not. In addition, for the type part of function declarations, the output format is slightly different from that of others.
- And I completely rewrite the `printTree()` function under the definition of `ExprKind`, the output format and CFG grammar of C-Minus. The thing to note here is that I needed to check the child node for `ReturnExpr` to check if it has a value or not. On account of the definition that `TypeN` and `OpN` is a fake node to get a type information for other nodes, the parser will ignorer and not print anything.
- The file `util.h` is also changed.

FILE `cminus.y`

- Copy and paste the code from `yacc/tiny.y` and define my lexical rules based on C-Minus.
- For the declarations section, I defined terminal symbols of the grammar. They include tokens from the scanner and fake tokens to deal with ambiguity of the grammar.

  The declaration order in this section determines the precedence, so symbols are listed in a ascending order of priority. Errors, parentheses, brace,s curly braces, commas and semicolons have the highest priority. Comparison operators are defined as `%nonassoc` which means that they are non-associative. I defined binary operators `PLUS`, `MINUS`, `MUL`, `DIV` as left-associative operators(`%left`). As well, multiply and divide should precede addition and subtraction by declaration order. Token `ASSIGN` is right-associative(`%right`). After operators, there are `ID` and `NUM` token. Reserved words are given the lowest priority.

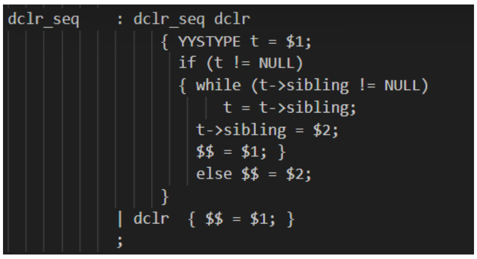
  As for the dangling else problem, I created one fake symbol `NOELSE` to set the priority of unmatched if statements. This token has a lower precedence than the token `ELSE`. Then, I used it with the `%prec` declaration in rules section.



```
%nonassoc NOELSE
%nonassoc ELSE
```

- Every name in the rules section that has not been declared in declarations section is a non-terminal. For example, `expr_stmt` is used to separate the expressions by `SEMICOLON`, and `empty` is for ε production rules.

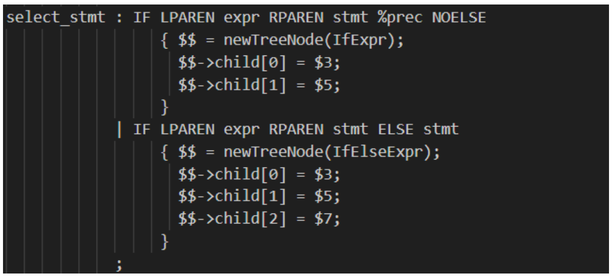
  I modified all production rules according to the given CFG grammar on C-Minus. `program` is the start symbol of the compiler. The action on all sequences(`dclr_seq`, `stmt_list`, `param_list`, `local_dclr`, `arg_list`) is implemented the same. The nodes in the list are connected by `sibling`.



```
dclr_seq    : dclr_seq dclr
                { YYSTYPE t = $1;
                  if (t != NULL)
                  { while (t->sibling != NULL)
                        t = t->sibling;
                    t->sibling = $2;
                    $$ = $1; }
                    else $$ = $2;
                }
            | dclr  { $$ = $1; }
            ;
```

  About `ID` and `NUM`, we need to use `tokenString` to get names or values of each token. Thus I defined non-terminals `id` and `num`, which create a new `treeNode` and save its `tokenString` as an attribute. Now that I could access the value of the token by `$`. Additionally, those non-terminals also save `lineno` info because we need a `lineno` value of variable or function declaration for semantic analysis after. One thing to be aware of is that these nodes (including `type`, `relop`, `add_op`, `mul_op`,) may not be `free`d later on.

  The precedence follows the patterns defined in the declarations section. However, I used `%prec NOELSE` and forced precedence for a production rule `select_stmt : IF LPAREN expr RPAREN stmt %prec NOELSE` in order to deal with dangling else problem.



```
select_stmt : IF LPAREN expr RPAREN stmt %prec NOELSE
                { $$ = newTreeNode(IfExpr);
                  $$->child[0] = $3;
                  $$->child[1] = $5;
                }
            | IF LPAREN expr RPAREN stmt ELSE stmt
                { $$ = newTreeNode(IfElseExpr);
                  $$->child[0] = $3;
                  $$->child[1] = $5;
                  $$->child[2] = $7;
                }
            ;
```

Execution

```
apt-get install bison
make cminus_parser // compile
./cminus_parser test.cm // execute
```

Result

- `test.1.txt`

```
/* A program to perform Euclid
   Algorithm to computer gcd */

int gcd (int u, int v)
{
    if (v == 0) return u;
    else return gcd(v,u-u/v*v)
    /* u-u/v*v == u mod v */
}

void main(void)
{
    int x; int y;
    x = input(); y = input();
    output(gcd(x,y));
}
```

```
result.1.txt

C-MINUS COMPILATION: ./test.1.txt

Syntax tree:
  Function Declaration: name = gcd, return type = int
    Parameter: name = u, type = int
    Parameter: name = v, type = int
    Compound Statement:
      If-Else Statement:
        Op: ==
          Variable: name = v
          Const: 0
        Return Statement:
          Variable: name = u
        Return Statement:
          Call: function name = gcd
            Variable: name = v
            Op: -
              Variable: name = u
              Op: *
                Op: /
                  Variable: name = u
                  Variable: name = v
                Variable: name = v
  Function Declaration: name = main, return type = vo
    Void Parameter
    Compound Statement:
      Variable Declaration: name = x, type = int
      Variable Declaration: name = y, type = int
      Assign:
        Variable: name = x
        Call: function name = input
      Assign:
        Variable: name = y
        Call: function name = input
      Call: function name = output
        Call: function name = gcd
          Variable: name = x
          Variable: name = y
```

- `test.2.txt`

```
void main(void)
{
    int i; int x[5];

    i = 0;
    while( i < 5 )
    {
        x[i] = input();

        i = i + 1;
    }

    i = 0;
    while( i <= 4 )
    {
        if( x[i] != 0 )
        {
            output(x[i]);
        }
    }
}
```

```
C-MINUS COMPILATION: sample/test.2.txt

Syntax tree:
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      Variable Declaration: name = i, type = int
      Variable Declaration: name = x, type = int[]
        Const: 5
      Assign:
        Variable: name = i
        Const: 0
      While Statement:
        Op: <
          Variable: name = i
          Const: 5
        Compound Statement:
          Assign:
            Variable: name = x
              Variable: name = i
            Call: function name = input
          Assign:
            Variable: name = i
            Op: +
              Variable: name = i
              Const: 1
      Assign:
        Variable: name = i
        Const: 0
      While Statement:
        Op: <=
          Variable: name = i
          Const: 4
        Compound Statement:
          If Statement:
            Op: !=
              Variable: name = x
                Variable: name = i
              Const: 0
            Compound Statement:
              Call: function name = output
                Variable: name = x
                  Variable: name = i
```

- `test.txt` : check array declarations, call expressions, compound statements, return expressions

```
int g(int h, int i){
  int k[4];
  k[2] = 2+1+3;
  return;

}
int skldf33(void){
 g(a,b);
 if(ds=0)
 {}
 return 1;
}
```

```
C-MINUS COMPILATION: sample/test.txt

Syntax tree:
  Function Declaration: name = g, return type = int
    Parameter: name = h, type = int
    Parameter: name = i, type = int
    Compound Statement:
      Variable Declaration: name = k, type = int[]
        Const: 4
      Assign:
        Variable: name = k
          Const: 2
        Op: +
          Op: +
            Const: 2
            Const: 1
          Const: 3
      Non-value Return Statement
  Function Declaration: name = skldf33, return type = int
    Void Parameter
    Compound Statement:
      Call: function name = g
        Variable: name = a
        Variable: name = b
      If Statement:
        Assign:
          Variable: name = ds
          Const: 0
        Compound Statement:
      Return Statement:
        Const: 1
```

- `if.test.txt` : check if else statements

```
void main(void)
{
  if (a <0) {}
    if(a==2)
      a = 3;
    else
      a = 4;
}
```

```
C-MINUS COMPILATION: sample/if.test.txt

Syntax tree:
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      If Statement:
        Op: <
          Variable: name = a
          Const: 0
        Compound Statement:
      If-Else Statement:
        Op: ==
          Variable: name = a
          Const: 2
        Assign:
          Variable: name = a
          Const: 3
        Assign:
          Variable: name = a
          Const: 4
```

- `op.test.txt` : check precedence and associativity of operations

```
int main(void)
{
  a = a + b * c;
  if (a < b)
    a = b = c;
}
```

```
C-MINUS COMPILATION: sample/op.test.txt

Syntax tree:
  Function Declaration: name = main, return type = int
    Void Parameter
    Compound Statement:
      Assign:
        Variable: name = a
        Op: +
          Variable: name = a
          Op: *
            Variable: name = b
            Variable: name = c
      If Statement:
        Op: <
          Variable: name = a
          Variable: name = b
        Assign:
          Variable: name = a
          Assign:
            Variable: name = b
            Variable: name = c
```

Reference

https://www.gnu.org/software/bison/manual/ Bison manual

https://efxa.org/2014/05/17/techniques-for-resolving-common-grammar-conflicts-in-parsers/ Dangling else handling