Micro/Ex Compiler

A robust compiler implementation for the Micro/Ex programming language that generates optimized three-address intermediate code. Built using Lex/Flex and Yacc/Bison, this compiler features strong type checking, control structures, and comprehensive error handling.

Features

Core Language Support - Integer and float variable declarations - Fixed-size array support - Multiple variable declarations - Type checking and validation

Control Structures - FOR loops (TO/DOWNTO) - IF-THEN-ELSE statements - Nested control structures - Print statements

Expression Support - Arithmetic operations (+, -, *, /) - Comparison operators (>=, <=, >, <, ==, !=) - Array element access - Unary minus operator

Project Structure

- microex-lex-compiler/
  - src/ # Source code
    - include/ # Header files
      - microex.h # Common declarations
    - lexer.l # Lexical analyzer
    - parser.y # Parser and code generator
  - docs/ # Documentation
    - compiler-guide.md # Technical documentation
    - Report.md # Project report
    - Report.pdf
  - tests/ # Test files
    - test_declarations.microex
    - test_assignments.microex
    - test_for_loops.microex
    - test_complete.microex
    - test_pdf_example.microex
  - build/ # Generated files
    - lex.yy.c
    - y.tab.c
    - y.tab.h
  - Makefile # Build system
  - README.md # Project overview

Prerequisites

- GCC compiler
- Flex (lexical analyzer generator)

- Bison (parser generator)
- Make build system
- (Optional) Pandoc for documentation
- (Optional) clang-format for code formatting

Quick Start

```
git clone <repository>
cd microex-lex-compiler
```

1. **Build the compiler**

   ```
   make clean && make && make test
   ```

2. **Run tests**

   ```
   make test              # Run all tests
   make test-declarations # Test declarations only
   make test-assignments  # Test assignments only
   make test-for-loops    # Test FOR loops
   make test-complete     # Test full implementation
   make test-all
   ```

3. **Generate documentation**

   ```
   make docs
   ```

Usage

1. **Write your Micro/Ex program** (example.microex):

   ```
   Program Example
   Begin
       declare i as integer;
       declare a, b as float;

       a := 3.14;
       b := 2.0;

       FOR (i := 1 TO 10)
           a := a * b;
       ENDFOR

       IF (a >= 100.0) THEN
           print(a);
       ENDIF
   End
   ```

2. **Compile your program**:

```
./microex example.microex
```

3. **View the generated three-address code**:

```
START Example

Declare i, Integer
Declare a, Float
Declare b, Float

F_STORE 3.14,a
F_STORE 2.0,b

I_STORE 1,i
lb&1:
F_MUL a,b,T&1
F_STORE T&1,a

INC i
I_CMP i,10
JLE lb&1

F_CMP a,100.0
JL lb&2
CALL print, a
lb&2:

HALT Example
```

Make Targets

Target	Description
`make`	Build the compiler
`make clean`	Remove generated files
`make test`	Run all tests
`make docs`	Generate documentation
`make format`	Format source code
`make install`	Install to /usr/local/bin
`make debug`	Build with debug symbols
`make help`	Show available targets

Development

Adding New Features

1. **Update Grammar**: Modify `src/parser.y`

2. **Update Lexer**: Modify `src/lexer.l`
3. **Add Tests**: Create new test file in `tests/`
4. **Update Documentation**: Modify files in `docs/`

Running Tests

```
# Run specific test
make test-declarations

# Run all tests
make test

# Debug build and test
make debug && make test
```

Code Formatting

```
# Format all source files
make format
```

Documentation

- **compiler-guide.md**: Complete technical documentation
- **Example programs**: See `tests/` directory
- **Generated documentation**: Run `make docs`

Contributing

1. Fork the repository
2. Create a feature branch
3. Make your changes
4. Add tests for new features
5. Submit a pull request

License

MIT License - See LICENSE file for details

Support

For bug reports and feature requests, please: 1. Check existing issues 2. Create a new issue with detailed description 3. Include example code and error messages

Authors

G36maid miku65434@gmail.com

Micro/Ex Compiler Technical Guide

Table of Contents

Language Specification

Lexical Structure

Keywords

```
Program  Begin    End      declare  as       integer  float
FOR      TO      DOWNTO   ENDFOR   IF       THEN     ELSE
ENDIF    print
```

Operators

```
:=    // Assignment
+     // Addition
-     // Subtraction/Unary minus
*     // Multiplication
/     // Division
>=    // Greater than or equal
<=    // Less than or equal
>     // Greater than
<     // Less than
==    // Equal to
!=    // Not equal to
```

Special Symbols

```
;     // Statement terminator
,     // List separator
[     // Array index open
]     // Array index close
(     // Expression/parameter open
)     // Expression/parameter close
```

Identifiers

- Begin with a letter (A-Z, a-z)

- Can contain letters, digits (0-9)
- Case-sensitive
- Maximum length: 50 characters

Literals

- Integer literals: Sequence of digits (e.g., 123, 42)
- Float literals: Digits with decimal point (e.g., 3.14, 0.5)

Grammar (EBNF)

```
program         → 'Program' identifier 'Begin' declaration_list statement_list 'End'

declaration_list → (declaration)*
declaration     → 'declare' identifier_list 'as' type ';'
identifier_list → identifier (',' identifier)*
type            → 'integer' | 'float'
array_decl      → identifier '[' INTEGER_LITERAL ']'

statement_list → (statement)*
statement      → assignment_stmt
               | for_stmt
               | if_stmt
               | print_stmt

assignment_stmt → (identifier | array_access) ':=' expression ';'
array_access   → identifier '[' expression ']'

for_stmt       → 'FOR' '(' identifier ':=' expression direction expression ')'
                 statement_list
                 'ENDFOR'
direction      → 'TO' | 'DOWNTO'

if_stmt        → 'IF' '(' condition ')' 'THEN'
                 statement_list
                 ('ELSE' statement_list)?
                 'ENDIF'

print_stmt     → 'print' '(' expression (',' expression)* ')' ';'

expression     → term (('+' | '-') term)*
term           → factor (('*' | '/') factor)*
factor         → INTEGER_LITERAL
               | FLOAT_LITERAL
               | identifier
               | array_access
```

```
              | '(' expression ')'
              | '-' factor

condition     → expression ('==' | '!=' | '>' | '>=' | '<' | '<=') expression
```

Compiler Architecture

Phase 1: Lexical Analysis (microex.l)

1. **Token Recognition**
   - Keywords and operators
   - Identifiers and literals
   - Error detection for invalid characters
2. **Symbol Management**
   - Maintenance of symbol table
   - String literal pool
   - Line number tracking

Phase 2: Syntax Analysis (microex.y)

1. **Grammar Implementation**
   - LALR(1) parsing
   - Precedence rules for operators
   - Error recovery strategies
2. **Semantic Actions**
   - Type checking
   - Scope management
   - Code generation triggers

Phase 3: Code Generation

1. **Three-Address Code**
   - Instruction selection
   - Register allocation
   - Label management
2. **Optimization**
   - Constant folding
   - Dead code elimination
   - Common subexpression elimination

Three-Address Code Reference

Data Movement Instructions

```
I_STORE src,dest       // Integer assignment
F_STORE src,dest       // Float assignment
I_LOAD  src,dest       // Integer load
F_LOAD  src,dest       // Float load
```

Arithmetic Instructions

```
I_ADD   op1,op2,dest   // Integer addition
I_SUB   op1,op2,dest   // Integer subtraction
I_MUL   op1,op2,dest   // Integer multiplication
I_DIV   op1,op2,dest   // Integer division
I_UMINUS op,dest       // Integer unary minus

F_ADD   op1,op2,dest   // Float addition
F_SUB   op1,op2,dest   // Float subtraction
F_MUL   op1,op2,dest   // Float multiplication
F_DIV   op1,op2,dest   // Float division
F_UMINUS op,dest       // Float unary minus
```

Control Flow Instructions

```
J       label          // Unconditional jump
JE      label          // Jump if equal
JNE     label          // Jump if not equal
JG      label          // Jump if greater
JGE     label          // Jump if greater/equal
JL      label          // Jump if less
JLE     label          // Jump if less/equal

I_CMP   op1,op2        // Integer comparison
F_CMP   op1,op2        // Float comparison
```

Program Structure Instructions

```
START   name           // Program start
HALT    name           // Program end
Declare name,type      // Variable declaration
CALL    name,args...   // Procedure call
```

Symbol Table Management

Symbol Table Entry Structure

```c
typedef struct {
    char name[50];          // Symbol name
    char type[20];          // Data type
    int array_size;         // Array size (0 for scalar)
    int declared;           // Declaration flag
} symbol_t;
```

Operations

1. **Symbol Insertion**

    - Check for duplicates
    - Type validation
    - Array bounds verification
2. **Symbol Lookup**
    - Type checking
    - Array bounds checking
    - Undeclared variable detection
3. **Scope Management**
    - Single scope (global)
    - Name collision prevention
    - Declaration tracking

Error Handling

Lexical Errors

- Invalid characters
- Malformed numbers
- Identifier length exceeded
- Unterminated strings

Syntax Errors

- Missing semicolons
- Mismatched parentheses
- Invalid statement structure
- Incorrect array declarations

Semantic Errors

- Type mismatches
- Undeclared variables
- Array bounds violations
- Invalid operations

Error Recovery

1. **Panic Mode**
    - Skip to next semicolon
    - Resynchronize at statement boundary
2. **Error Messages**
    - Line number
    - Error context
    - Suggested fix

Optimization

Implemented Optimizations

1. **Constant Folding**

   ```
   x := 2 + 3  →  x := 5
   ```

2. **Common Subexpression Elimination**

   ```
   t1 := a + b
   t2 := a + b  →  t2 := t1
   ```

3. **Dead Code Elimination**

   ```
   if (0) then   →   (removed)
      x := 1
   endif
   ```

Future Optimizations

1. Strength Reduction
2. Loop Invariant Code Motion
3. Register Allocation
4. Peephole Optimization

Development Guide

Adding New Features

1. **New Operators**
   - Add token in microex.l
   - Update grammar in microex.y
   - Implement semantic actions
   - Add code generation rules
2. **New Control Structures**
   - Define syntax in grammar
   - Implement label management
   - Add code generation patterns
   - Update error handling
3. **New Data Types**
   - Extend symbol table
   - Add type checking rules
   - Implement conversion rules
   - Update code generation

Testing

1. **Unit Tests**
   - Lexical analysis

- Parsing
- Code generation
- Error handling

2. **Integration Tests**
   - Complete programs
   - Error cases
   - Optimization verification
3. **Test Files**
   - test_declarations.microex
   - test_assignments.microex
   - test_for_loops.microex
   - test_complete.microex

Best Practices

1. **Code Organization**
   - Modular design
   - Clear commenting
   - Consistent naming
   - Error checking
2. **Memory Management**
   - Symbol table cleanup
   - Temporary storage
   - String handling
   - Error recovery
3. **Documentation**
   - Inline comments
   - API documentation
   - Error messages
   - Usage examples