CS323 Project Midterm Report

C-like Compilers in Rust

2024.11.18

Ben Chen, Jiarun Zhu, Yicheng Xiao

Contents

1 Software Stack and Project Goal 3
1.a Software stack and Project Goal 4
1.a.a Why not Rust? 4
1.a.b Project Goal 4
2 Specification and Core Feature 5
2.a Lexer 6
2.b Parser 9
2.b.a Error Recovery 14
2.c AST 15
3 The Design of the Compiler 17
3.a Design Principles 18
4 Implementation Progress 19
4.a Progress 20
4.b Schedule 21

1 Software Stack and Project Goal

Software stack and Project Goal

Using `flex` and `bison` requires different tool chain and binding different files together, we are hoping to implement a software tool chain that can integrate Lexer, Parser, Semantic Analysis and the Intermediate Result Generation(Optimization) together.

1.a.a Why not Rust?

Rust is famous for its powerful compiler, by using Rust to build a compiler, we can dive more deeply into this amazing language. Also, it is easier to test using Rust by only a simple `cargo test`!

```
rust            1.82.0
logos           0.14.2
lalrpops        0.22.0
llvm_ir         0.11.1 (Support llvm-IR)
```

1.a.b Project Goal

We are trying to build a compiler frontend system to generate LLVM-IR result. And use the LLVM backend system, we can build a compiler thoroughly.

2 Specification and Core Feature

Lexer

By using `Logos`, a Rust-supported lexer, our language supports the following tokens. We are still planning to add more tokens when applying more features.

Operators

`>`	OpGreaterThan	`=`	OpAssign	`^`	OpPow
`<`	OpLessThan	`+`	OpPlus	`&&`	OpAnd
`<=`	OpLessThanEqual	`-`	OpMinus	`||`	OpOr
`>=`	OpGreaterThanEqual	`*`	OpMul	`!`	OpNot
`==`	OpEqual	`/`	OpDiv	`++`	OpIncrement
`!=`	OpNotEqual	`%`	OpMod	`--`	OpDecrement

Punctuation

`.`	Dot	`[`	LeftBracket	`{`	LeftBrace
`,`	Comma	`]`	RightBracket	`}`	RightBrace
`:`	Colon	`(`	LeftParen		
`;`	Semicolon	`)`	RightParen		

Lexer (ii)

Keywords

`if`	KeywordIf	`for`	KeywordFor	`break`	KeywordBreak
`else`	KeywordElse	`return`	KeywordReturn	`continue`	KeywordContinue
`while`	KeywordWhile				

Declaration

`enum`	DeclarationEnum	`struct`	DeclarationStruct	`fn`	DeclarationFunctio

Type

`bool`	TypeBool	`char`	TypeChar	`string`	TypeString
`int`	TypeInt	`float`	TypeFloat	`null`	TypeNull

Lexer (iii)

Literals

<table>
<tr><td>`r"true|false"`</td><td>LiteralBool: bool</td><td>`r"'.'"` `r"'\\u[0-9a-fA-F]{1,6}'"`</td><td>LiteralChar: char</td><td>`r"-?[0-9]+"` `r"-?0[xX][0-9a-fA-F]+"`</td><td>LiteralInt: i32</td></tr>
<tr><td>`r"-?(?:0|[1-9]\d*)?\.\d+(?:[eE][+-]?\d+)?"`</td><td>LiteralFloat: f32</td><td>`r#""([^"\\]|\\["\\bnfrt]|\\x[0-9a-fA-F]{2}|\\u[a-fA-F0-9]{1,6})*""#`</td><td>LiteralString: String</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Identifiers

`r"[a-zA-Z_$][a-zA-Z0-9_$]*"`	Identifier: String

Comment

`r"//[^\n]*\n?"`	Line Comment
`r"/*"`	Block Comment

Parser

By using `LALRPOP`, a Rust Parser Generator, our language will support a superset of SPL with the following syntax for now. Also, we tuned the original grammar to utilize `LALRPOP`'s features.

Term and Computation Expression (returns a value excluding bool or variable itself)

```
// Below are all left associative for now
CompExpr -> Term | // Priority 0
   CompExpr ^ CompExpr | CompExpr % CompExpr // Priority 1
   CompExpr * CompExpr | CompExpr / CompExpr // Priority 2
   CompExpr + CompExpr | CompExpr - CompExpr // Priority 3
   Identifier OpIncreament | Identifier OpDecreament

Term -> LiteralInt | LiteralFloat | LiteralChar | LiteralString | Identifier |
   LeftParen CompExpr RightParen

Specifier -> TypeInt | TypeFloat | TypeChar | TypeString | Null
   // And struct...
```

Parser (ii)

Conditional Expression (returns a boolean since if/loop only accept condition)

```
CondExpr -> CondTerm |
    CompExpr OpEqual CompExpr |
    CompExpr OpNotEqual CompExpr |
    CompExpr OpLessThan CompExpr |
    CompExpr OpGreaterThan CompExpr |
    CompExpr OpLessThanEqual CompExpr |
    CompExpr OpGreaterThanEqual CompExpr |
    CondExpr OpEqual CondExpr |
    CondExpr OpNotEqual CondExpr |
    CondExpr OpAnd CondExpr |
    CondExpr OpOr CondExpr |
    OpNot CondExpr

CondTerm -> LiteralBool | Identifier | LeftParen CondExpr RightParen
// Only Bool variable
```

Parser (iii)

Assignment Statement & Local Declaration

```
AssignStmt -> Identifier OpAssign CompExpr Semicolon |
     Identifier OpAssign CondExpr Semicolon

Defs -> Def Defs | $ // *
Def -> Specifier Decs Semicolon

Decs -> Dec | Dec Comma Decs // +
Dec -> VarDec | VarDec OpAssign CompExpr | VarDec OpAssign CondExpr
```

Statements (If/else, while/for)

```
Stmts -> Stmt Semicolon Stmts | $ // *

ForInit -> Decs Comma ForInit | $ // *
```

Parser (iv)

```
Stmt -> AssignStmt | Def | LeftBrace Stmts RightBrace |
    | CondExpr | CompExpr | // In case someone writes this...
    | KeywordReturn CompExpr Semicolon
    | KeywordReturn CondExpr Semicolon // Or: Expr -> CompExpr | CondExpr
    | KeywordIf LeftParen CondExpr RightParen Stmt
    | KeywordIf LeftParen CondExpr RightParen Stmt KeywordElse Stmt
    | KeywordWhile LeftParen CondExpr RightParen Stmt
    | KeywordFor LeftParen ForInit Semicolon CondExpr Semicolon AssignStmt
        Semicolon Stmt
```

Function Declaration

```
FuncDec -> Type Identifier LeftParen ParaDecs RightParen
    LeftBrace Stmts RightBrace
```

Parser (v)

```
ParaDecs -> ParaDec Comma ParaDecs | ParaDec | $

ParaDec -> Specifier Identifier
```

Global Definitions (Variable and Function)

```
ExtDefs -> ExtDef ExtDefs | $ // *

ExtDef -> FuncDec | Def
```

The Program

```
Program -> ExtDefs
```

Parser (vi)

2.b.a Error Recovery

`LALRPOP` provides error recovery by doing so.

```
Term: Box<tree::CompExpr> = {
    <n: "int"> => Box::new(tree::CompExpr::Value(tree::Value::Integer(n))),
    // ...
    "(" <CompExpr> ")",
    ! => { errors.push(<>); Box::new(tree::CompExpr::Error) },
}

#[test]
fn test_error_recovery() {
    assert_compexpr_parse("2 + * 5", "(2: i32 + (MissingTermError * 5: i32))");
    assert_compexpr_parse("2 + * 5 *",
    "(2: i32 + ((MissingTermError * 5: i32) * MissingTermError))");
}
```

Semantic action as building syntax tree. Every parsing result returns a node, represented by an object in Rust. We first define what a node looks like.

```
// ast.rs
pub enum CompExpr {
    Value(Value),
    Variable(Variable),
    UnaryOperation(UnaryOperator, Box<CompExpr>),
    BinaryOperation(Box<CompExpr>, BinaryOperator, Box<CompExpr>),
    Error
}
```

When facing an input, it will returns something in a recursive form, just like a tree.

```
// grammar.lalrpop
pub CompExpr: Box<tree::CompExpr> = {
```

AST (ii)

```
    Term,
    <lhs:CompExpr> "^" <rhs:CompExpr> => {
      Box::new(tree::CompExpr::BinaryOperation (lhs, tree::BinaryOperator::Pow, rhs))
    },
    <lhs:CompExpr> "%" <rhs:CompExpr> => {
      Box::new(tree::CompExpr::BinaryOperation (lhs, tree::BinaryOperator::Mod, rhs))
    },
}

#[test]
fn test_expr() {
    // Test if evaluation order is correct
    assert_compexpr_parse("2 + 4 * 5", "(2: i32 + (4: i32 * 5: i32))");
    // Test expression with bracket
    assert_compexpr_parse("(2 + 4) * 5", "((2: i32 + 4: i32) * 5: i32)");
}
```

3 The Design of the Compiler

Design Principles

We try to implement our language from easy to hard. First, we are planning to implement C-basic features. Then, trying to dive deep into the principle of Rust compilers, we are trying to realize macro and lifetime features.

The **macro** will be parsed first in parser. To simplify the problem, we will only focus on macro translation instead of analyzing functional macros like Rust compiler does.

The **lifetime parameter** will be implemented using the natural feature `lalrpop` has. By locating the declaration of variable and function, we might be able to analyze the lifetime of variables and try to avoid some of the memory issues and de-referencing issues we mentioned in the former presentation. However, the notion of lifetime must come with ownership, therefore, our realization of lifetime checking might be a rather simple one.

4 Implementation Progress

Progress

- ✅ Simple Lexer
- ❌ More complicated Lexer
- ✅ Simple Parser
- ❌ More complicated parser
- ✅ Simple AST structure
- ❌ AST with visitor mode
- ❌ Semantic Analysis
- ❌ Intermediate Result Generation
- ❌ Intermediate Result Optimization

Schedule

- ✅ Simple Lexer
- 📅 `12.1` More complicated Lexer
- ✅ Simple AST structure
- 📅 `12.1` AST with visitor mode
- ✅ Simple Parser
- 📅 `12.1` More complicated parser
- 📅 `12.1` Semantic Analysis
- 📅 `12.22` Intermediate Result Generation
- 📅 `12.22` Intermediate Result Optimization