Code Generation Project 2020: Mini-Pascal Compiler Documentation

Vili Lipo, 014814253

2020-05-24 19:01:49+03:00

1 Introduction

Few months before the world seemed like a very different place than now and I decided that it would be a good time to learn the Rust Programming Language while doing this project. Well life happened and it was not that great of an idea. Despite all of that here I present to you my implementation of Mini-Pascal written in Rust and it produces simplified C-code.

I was learning Rust while I was doing this project, so it took me longer than I thought to implement the features. That is why, I unfortunately did not have time to implement "var"-parameters. Otherwise I think that the compiler is quite feature-complete and as Rust enforces safety in many ways I think that it has quite robust error handling too.

I also did some quite basic rookie mistakes while doing this project, like almost finishing it completely, before I realized that the my AST-struture is not idiomatic Rust at all, and then I reimplemented the whole parser and compiler back-end to use the new AST. This was very time consuming as I was experimenting with different AST-structures with a compiler that had everything but subroutines implemented. I should have implemented only like integer expressions and then experimented with different data structures.

In the end I am quite happy with the final data-structures, even though those still ended up making the semantical analysis part of my program very verbose. The whole project is over 4700 lines long and, I hope that it is not too overbearing to read, and I fully recognize that no one would consider the src/typefolder.rs to be clean code.

2 Architecture

The architecture of this compiler mostly follows the directions given in the lecture materials, but instead of visiting and mutating one AST structures, there is two distinct structures. The semantic analysis produces the TypedAst structure, that is then processed by the CodeGenVisitor. During the semantic analysis a symbol table is used to store variable information like type and unique address. This data is then added to the final AST-representation and the CodeGenVisitor does not need a symboltable at all. UML-diagram of the whole architecture can be seen in figure 1.

3 Instructions and testing

To build the application on Linux platform the Rust toolchain must be installed. Then the application can be run with `cargo run examples/test.minipascal target/test.c`. Alternatively it can be built with `cargo build` and then the binary is `target/debug/mini-pascal-compiler`. If there is issues in the build I also included a binary of the application in the `bin` folder with the name `vmpc` (Vili's Mini-Pascal Compiler). It should work on amd64 linux platforms with no extra dependencies. The test inputs are in the folder `examples`. There is Rust unit tests for the source and scanner modules, but due to lack of time the rest of the application is only tested via the files in examples.

4 Mini-Pascal token patterns

The token patterns can be observed in figure 2. Semicolons, brackets and braces have special meanings but those are not grouped in any way so I did not list them in the regular expressions. Nevertheless those are recognized by the scanner as tokens. Scanner does not give any special attention to predefined identifiers.

5 Context free grammar and techniques used to solve any problems

The Mini-Pascal context free grammar can be observed in figure 3. Here a call or variable construction is created to solve the violation of LL (1) rule. Now as the language syntax does not allow to assign a value to a call, in the parsing function the return type of call or variable is checked and based on that we see if the assign can be constructed. If a call is not followed by ":=" a call statement

Figure 1: Uml of the Architecture

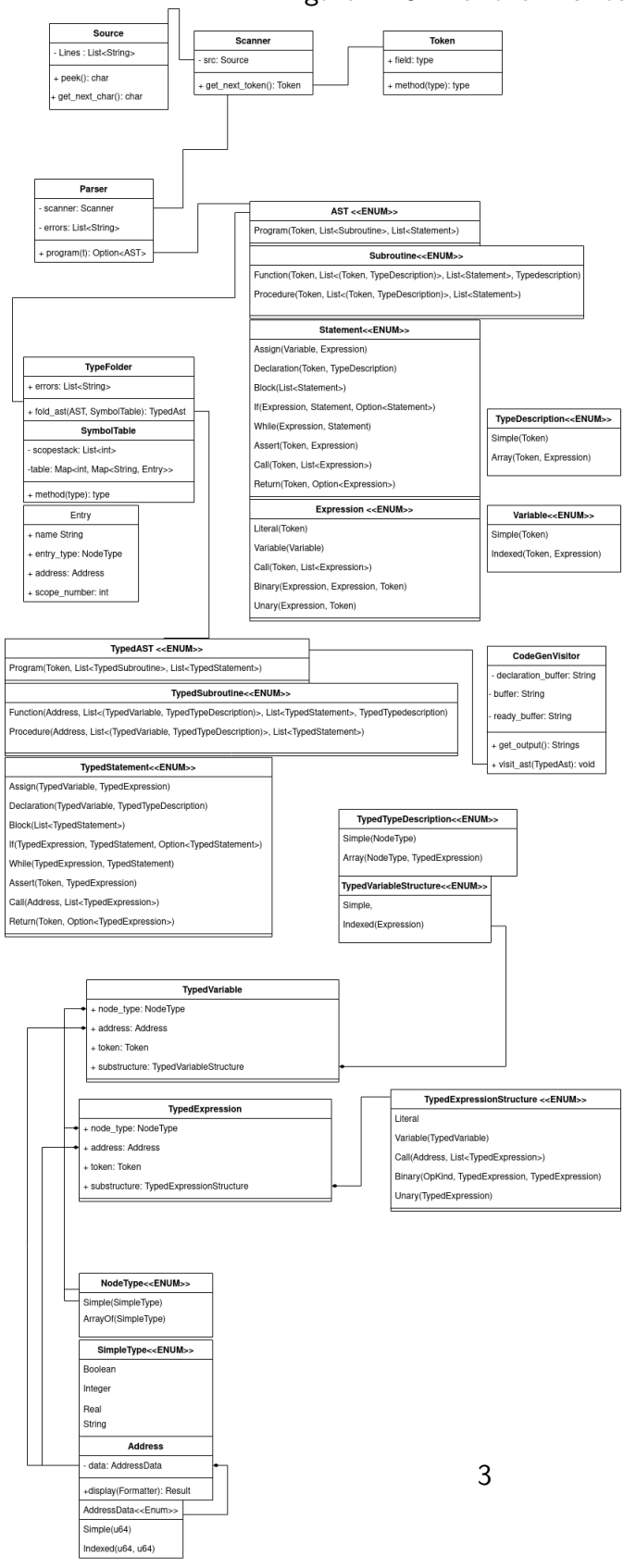

Figure 2: Token patterns

```
<integer> ::= <digit>*
<real> ::= <digit>*\.<digit>*
<real> ::= <digit>*\.<digit>*e^<digit>*
<string_literal> ::= "[^"]*"
<identifier> = ([a-z] | [A-Z])([a-z] | [A-Z] | _ | [0-9])*
<operator> ::= + | - | / | * | < | > | <> | <= | >=
<operator> :== and | or | not
<eof> ::=
<keyword> ::= var | if | then | else | of
<keyword> ::= while | do | begin | end | array
<keyword> ::= procedure | function | assert | return
```

is returned. So no actual backtracking is used, as the scanner is not made to reverse at any point. In factor this does not create any issues as all of the variants of call or variable are legal variants of factor.

The parser does not recognize writeln or read as statements. Those are parsed as regular calls. Those are recognized by the TypeFolder, so that user might redefine the identifier if he/she wishes to do so.

6 AST-specifications

For my first AST-produced by the parser I think the source code is the most easy way to explain it. I think that it is better than UML, as Rust provides very good tools for this kind of structure. The code is in figure 4. The AST is represented as enums with different variants for each structure. We see that AST-contains a variant program that has three parts, Token produced by the scanner, a list of subroutines and a list of statements being the main block. Then Subroutine is an enum with two variants function and procedure. Both of them have token, and parameters that are a list of tuples that have a token for the name and a type-description enum for the type construction. Functions have one additional type-description for the function output type.

In Rust to support recursive enum structures, the recursive definitions must be wrapped in Box objects. This is used in the Statement and Expression enums. In statement the If variant has `Option<Box<Statement> >` as its third field as the else branch may or may not be present.

Instead of decorating this structure by mutating it, the TypeFolder object folds

Figure 3: Context free grammar

```
<program> ::= "program"<id>";" {<procedure> | <function>}<block>"."
<procedure> ::= "procedure" <id>"("<parameters>")"";"<block>";"
<function> ::= "function" <id>"("<parameters>")"":"<type>";"<block>";"
<var-declaration> ::= "var"<id>{"," <id>} ":" <type>
<parameters> ::= ["var"]<id> ":" <type> { "," ["var"]<id> ":" <type> }
                | <empty>
<type> ::= <simple type> | <array type>
<simple type> ::= <type id>
<array type> ::= "array" "["[<expr>]"]" "of" <simple type>
<block> ::= "begin" <statement> {";"<statement>} [";"] "end"
<statement> ::= <simple statement>
                | <structured statement>
                | <var-declaration>
<simple statement> ::= <assignment_or_call> | <return statement>
<assignment_or_call> ::= <call_or_variable> (":=" <expr> | <empty>)
<arguments> ::= <expr> {"," expr } | <empty>
<return statement> ::= "return" [<expr>]
<structured statement> ::= <block> | <if statement> | <while statement>
<if statement> :== "if" <expr> "then" <statement> [ "else" <statement>"]
<while statement> ::= "while" <expr> "do" <statement>
<expr> ::= <simple expr> [ <relational operator> <simple expr> ]
<simple expr> ::= [<sign>] <term> { <adding operator> <term> }
<term> ::= <factor> {<multiplying operator> <factor>}
<factor> ::= <call_or_variable> | <literal>
<factor> ::= "("<expr>")" | "not" <factor>
<call_or_variable> ::= <id><call_or_variable_tail>
<call_or_variable_tail> ::= [<expression>]
                          | (<parameters>)
                          | "."size"
                          | <empty>
```

Figure 4: AST source code

```
pub enum AST {
    Program(Token, Vec<Subroutine>, Vec<Statement>),
}

pub enum Subroutine {
    Function(
        Token,
        Vec<(Token, TypeDescription)>,
        Vec<Statement>,
        TypeDescription,
    ),
    Procedure(Token, Vec<(Token, TypeDescription)>, Vec<Statement>),
}

pub enum Statement {
    Assign(Box<Variable>, Expression),
    Declaration(Token, TypeDescription),
    Block(Vec<Statement>),
    If(Expression, Box<Statement>, Option<Box<Statement>>),
    While(Expression, Box<Statement>),
    Assert(Token, Expression),
    Call(Token, Vec<Expression>),
    Return(Token, Option<Expression>),
}

pub enum Expression {
    Literal(Token),
    Variable(Box<Variable>),
    Binary(Box<Expression>, Box<Expression>, Token),
    Unary(Box<Expression>, Token),
    Call(Token, Vec<Expression>),
}

pub enum TypeDescription {
    Simple(Token),
    Array(Token, Expression),
}

pub enum Variable {
    Indexed(Token, Expression),
    Simple(Token),
}
```

this AST to TypedAst structure that contains the type-infomation and addressing information needed by the code generation. TypeFolder does most of the heavy lifting in the backend of this compiler. As a result the CodeGeneration visitor does not even need to access the symboltable or mutate the created TypedAst structure.

Using the folder pattern instead of the visitor pattern made the TypeFolder more verbose than its visitor counterpart. I implemented most of the application using a mutable data-structure first, before I refactored it to use the new data-structure. This is because there is quite a lot of code handling the building of the new typed nodes. Also compared to some more object oriented languages polymorphism in Rust felt a bit awkward and so this enum structure warrants use of large match-operator case switches in the visitor and folder code. The code base is quite verbose for that reason. The benefits of this are the that code-generation part can have very straightforward control-flow and still be completely safe.

In the TypedAst datastructure the TypedExpression is made in to a struct, that holds all of the important information. In its field substructure it contains a value of enum TypedExpressionStrucure that again models the different types of Expressions with types. This same idiom is also applied to typed variables. The TypedAst structure is then just visited by the CodeGeneration visitor to produce the output C-code.

In TypedAst the Read and Write statement variants are introduced. The TypeFolder can determine with its symbol table if user has redefined the identifier to mean some other value, or if it still means the special statement.

7 Implementation level decisions

My implementation of Mini-Pascal has static scope, since it felt bit more natural in the unlimited register machine idiom. If I had modelled some runtime stack, then dynamic scope might have been a good alternative.

Var-parameters are not implemented to the final version due to lack of time. So instead arrays are passed as references to subroutines and other types as values. Strings are char-pointers in my compiled C, but due to every string concatenation allocating a new memory section it does not mutate the existing string. Strings have maximum length of 512 characters, but for literals only the needed amount of memory is allocated. There is no garbage collection whatsoever.

Recursive calls work. A subroutine can call subroutines that were declared before it but not the ones that are declared after it. The change to declare functions before analysing their bodies is doable, but not trivial in the current

folding code.

Predefined identifiers are added to the symbol table when it is created. There entries for type identifiers, special subroutines and predefined values are added to top scope. Then in TypeFolder if a user shadows any of these names in lower scopes the new entries will appear first in a lookup operation. Some predefined identifiers like type identifiers, read and writeln are regarded as special in symbol table and the semantic analysis takes this in to account when producing the TypeAst-structure. In the TypedAst structure there exists statement variants for read and write, that do not exist in the untyped AST. This is because the symbol table information is needed to determine if the user has shadowed those special identifiers.

Booleans are printed as integers 0 meaning false and 1 true. This could be fixed by extending the runtime. Boolean values true and false are stored in global variables `r1` and `r2`, in the generated c-code. This is done so that the code can be uniform for all variables and that no special case has to be done for these predefined ones.

8 Semantic analysis

The TypeFolder does a large amount of semantic analysis. It type checks all expressions, assignments and calls. Basically for all expressions except the relational ones, if the operator is applicable to the arguments and the arguments are of the same type, then the expression has the same type as the arguments. For the relational operations the sides are checked in the same way but the type of the expression is boolean.

For statements, in if, while and assert statements the conditions are checked to be of the type boolean.

In assign statements the left hand side variable must have matching type to the right hand side expression. For declarations it is checked that the variable is not declared twice in same scope. Then the type identifier is checked to be a correct type identifier in the symboltable. In declarations the array size expression is checked to be of the type integer. Same goes for array variable indexes.

For read statements it is checked that all of its arguments are variables. Write statement accepts all arguments.

For functions the TypeFolder checks that a function has a return statement that returns a value of correct type.

Calls are checked so that procedure calls can only exist as statements and function calls can only exist as expressions. It is also checked that the argument types match the parameter types.

Figure 5: Complete list of semantical checks

1. Subroutines
   (a) Function must return a correct type
   (b) Parameters have valid types
2. Expressions
   (a) Both sides have the same type
   (b) Operator is applicable to the type
3. Variables
   (a) Variable is declared before use
   (b) Variable is not declared twice in same scope
   (c) Array variable index must be integer
4. Type identifiers
   (a) Identifier used as a type must be a valid type
   (b) Array Type length expression must be integer
5. Statements
   (a) If condition must be a boolean
   (b) While condition must be a boolean
   (c) Assert condition must be a boolean
   (d) Assignment types must match, and the variable must be declared before assignment.
   (e) Call
      i. Only identifiers that are either functions or procedures can be called
      ii. Argument types must match the parameter types for the subroutine
      iii. Only procedure calls are accepted as statements, and function calls as expressions
   (f) Read statement parameters must be variables

9 Major problems and solutions

I explained how I solved the LL (1) violations in the third section. In addition to the biggest issues was how to efficiently model the AST without redundant code in the data structure or the visitors and folders. I found that implementing static scope using the lecture materials was quite straight forward. Maybe attending the "Ohjelmointikielten Periaatteet" course helped me in that task, as scoping was one of the primary study interests on that course.

First I was bit unsure how using goto for flow-control would affect the code-generation. Mostly I worried how to ensure that variables are declared only once. Then I designed the three buffer system, where the code is generated to three separate string buffers. The first is for declarations, and second is for the main body of the code. Then when a subroutine or main block is complete the both of the buffers are pushed to the ready buffer the declarations first.

Using this method the gotos can not cause duplicate declarations in C-runtime. This actually reminded me from a code style rule from my C-programming course in Lappeenranta University Of Technology, where the lecturer demanded that all variables are declared at the start of a function. Now it clicked that in that way the code would resemble the assembly generated by C-compiler and thats why the lecturer preferred to have it that way.

The specification for I/O-operations was a positive suprise to me, as it was easy to match the desired behaviour using C-standard library functions 'printf' and 'scanf'. So the format strings are just generated based on the argument types and the argument addresses are passed to the C-functions.

Arrays were a bit difficult to figure out, as the size is determined in runtime C, so dynamic memory management had to be used. I decided to save the size for every array in a variable called `{address}_size`. This variable is set the value of the size expression at the array declaration, and then the `.size` function was trivial to implement.

My enum-based data-structure essentially made me use a lot of `match` statements in a switch-like fashion in the code base. Many would argue that this is not clean code at all, and that the switches should neatly contained in some factory that produces some polymorphic objects that would behave nicely during code generation. I found doing that too difficult as I am a Rust novice. I don't know if such polymorphic structure would even be idiomatic Rust as the Rust compiler written in Rust uses these enum-based structures in its own abstract syntax model. As a whole I found working with Rust to be challenging but still very rewarding.

The memory management in the generated code is abysmal. No attempt at garbage collection is done and the string concatenation is especially offending in this category, as every step concatenation reserves a fresh section for the result.

If this is done in a loop every iteration reserves more memory that is never freed. This could be fixed by rewriting assignment for strings. Now the char pointer of the result is set as the value of the left hand side variable. If `strcpy` would be used to assign a value, then the expressions would not need to reserve new memory every time those are run.

10 Error handling strategies

For error handling I heavily utilize the Rust Option-enum, that has variants Some(<T>), and None. So if a parsing method is successful it returns Some(expression) for example. If it fails it returns None. Then the caller can check that what variant it received by using this if let statement:

```
if let Some(expression) = self.expression() {
  // do something with expression
}
```

or the match statement:

```
match self.expression() {
  Some(expression) => {
      // do somethign with expression
      },
  None => println("Could not parse expression!")
}
```

This success based control-flow is heavily used in the parser, and TypeFolder. This might have been the case that I found the new "Silver bullet" of safe code and overused it so much that it actually falls to the anti-pattern category.

In the parser I only create a error message from the lowest level where the error occurs. So when a parsing-method calls a another parsing-method that returns None the none is just forwarded upwards. If the error is the result of expecting a certain token and something else appears, then an error message is added to "parser.errors" and None is returned. This usually leads to re-synchronization at statement level. Especially bad

If the syntax is so broken that the parser can not procedure a sensible AST-from it naturally no semantic analysis can be done.

The TypeFolder has quite similar approach to error handling. So if a folding method cannot produce a correctly typed node, it returns None that is passed along, usually to statement level. At the source of the problem an error message is appended to TypeFolders error list.

There is minimal error handling in CodeGenVisitor, as the TypedFolder takes care that the TypedAst-passed to CodeGenVisitor is safe to compile. Trying to write an array makes `printf` format string to be "Error array printing not implemented". In addition to that if the visitor fails to read a C-runtime file an error message is printed. This is the extent of error handling in the CodeGeneration visitor.

Worst issues for the parser are when there is issues in subroutine declarations. If there is issue there the parser will most likely parse the subroutine body as the main body. That can also lead to quite confusing error messages. The first error message should provide the information on what exactly went even in that case.

11 Work hour log

Date	Duration	Task
23.3	3h	Creating module Source
23.3	1h	Initial testing of Source
24.3	4h	Creating Scanner
24.3	2h	Initial testing of Scanner
26.3	5h	Initial recursive parser with just ints and main block
26.3	3h	Initial AST
27.3	1h	PrintVisitor
30.3	8h	Writing SemanticVisitor and CodeGeneration for ints
31.3	4h	Adding booleans and reals
3.4 - 5.5	Focusing on other courses	
5.5	4h	Adding strings
6.5	3h	Adding actually simple arrays
8.5	2h	Adding string arrays
10.5	2h	Implementing size
11.5	4h	assert and parameters
13.5	3h	Parsing subroutines
14.5	3h	Refactoring AST and parser
15.5	2h	Refactoring AST and printvisitor
15.5	2h	SemanticVisitor to be TypeFolder
16.5	10h	Refactoring AST and implementing TypeFolder.
17.5	8h	Refactoring AST, Typefolder and CodeGeneration.
18.5	1h	Reimplementing code generation
18.5	1h	Created Address data-structure
19.5	2h	Reimplementing code generation
21.5	8h	Parsing parameters, Subroutine semantic analysis
22.5	3h	Subroutine semantic analysis, parameter analysis
23.5	2h	Writing report
23.5	6h	Implementation of subroutine code generation, testing
25.6	6h	Wrapping up.
Total	103h	Total