pdfauthor=Nicolas Hiillos, pdftitle=Course Project Report, pdfkeywords=, pdfsubject=, pdfcreator=Emacs 28.1 (Org mode 9.6), pdflang=English

Course Project Report

Nicolas Hiillos

May 16, 2022

Instructions

cmake and **gcc** are required to compile the interpreter. After `./build.sh` is run, to compile a mini-pl progmram file, run `./build/mini-pl [filename]`, the resulting file is named `out.wat`. For testing purposes, the user can use `./build/mini-pl -s [filename]` to run merely the scanner, or `./build/mini-pl -p [filename]` to run the scanner+parser. Both commands print a readable result. Example programs are provided in `./test/`.

`./run.sh [path to mini-pl program]` can be run after `build.sh` in the project root. This script compiles the program to WAT, runs wat2wasm, and serves the wasm env on localhost:8080/env.html using python3.

Implementation details

I decided to make WAT (WebAssembly text format) the compile target. I chose this over the binary WASM format to ease readability.

Scanner

The compiler uses a simple ad-hoc scanner and recognizes the following tokens. The main functionality of the scanner consists of a long switch statement which applies these regular expressions one character at a time, looking ahead as necessary.

```
COMMENT = "// | {* ... *}"
STR_LIT = """ <chars> """
PLUS = "+"
MINUS = "-"
MUL = "*"
DIV = "/"
```

```
MOD = "%"
EQ = "="
NEQ = "<>"
LT = "<"
GT = ">"
LTE = "<="
GTE = ">="
LEFT_PAREN = "("
RIGHT_PAREN = ")"
LEFT_BRACKET = "["
RIGHT_BRACKET = "]"
ASSIGN = ":="
DOT = "."
COMMA = ","
SEMICOLON = ";"
COLON = ":"
OR = "or"
AND = "and"
NOT = "not"
IF = "if"
THEN = "then"
ELSE = "else"
OF = "of"
WHILE = "while"
DO = "do"
BEGIN = "begin"
END = "end"
VAR = "var"
ARRAY = "array"
PROCEDURE = "procedure"
FUNCTION = "function"
PROGRAM = "program"
ASSERT = "assert"
RETURN = "return"
INT_LIT = <digits>
REAL_LIT = <digits>'.' <digits> [ 'e' [ <sign> ] <digits>]
ID = <letter> { <letter> | <digit> | '_' }
SCAN_ERROR = ""
SCAN_EOF = EOF
```

The scanner handles errors by outputting tokens of type `ERROR` for unterminated strings and unexpected characters. These error tokens should then be handled by the parser.

Parser

The parser implements each of the productions (everything enclosed in <>) in the CFG below as a class which is used to make up the syntax tree.

```
<program> ::= "program" <id> ";" { <procedure> | <function> } <main-
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>
<array type> ::= "array" "[" [<integer expr>] "]" "of" <simple type>
<simple type> ::= <type id>
<block> ::= "begin" <statement> { ";" <statement> } [ ";" ] "end"
<statement> ::= <simple statement> | <structured statement> | <var-
declaration>
<empty> ::=
<simple statement> ::= <assignment statement> | <call> | <return statement> |
            <read statement> | <write statement> | <assert statement>
<assignment statement> ::= <variable> ":=" <expr>
<call> ::= <id> "(" <arguments> ")"
<arguments> ::= expr { "," expr } | <empty>
<return statement> ::= "return" [ expr ]
<read statement> ::= "read" "(" <variable> { "," <variable> } ")"
<write statement> ::= "writeln" "(" <arguments> ")"
<assert statement> ::= "assert" "(" <Boolean expr> ")"
<structured statement> ::= <block> | <if statement> | <while statement>
<if statement> ::= "if" <Boolean expr> "then" <statement> |
                   "if" <Boolean expr> "then" <statement> "else" <statement>
<while statement> ::= "while" <Boolean expr> "do" <statement>
<expr> ::= <simple expr> |
           <simple expr> <relational operator> <simple expr>
<simple expr> ::= [ <sign> ] <term> { <adding operator> <term> }
<term> ::= <factor> { <multiplying operator> <factor> }
```

```
<factor> ::= <call> | <variable> | <literal> | "(" <expr> ")" |
             "not" <factor> | <factor> "." "size"
<variable> ::= <variable id> [ "[" <integer expr> "]" ]
<relational operator> ::= "=" | "<>" | "<" | "<=" | ">=" | ">"
<sign> ::= "+" | "-"
<negation> ::= "not"
<adding operator> ::= "+" | "-" | "or"
<multiplying operator> ::= "*" | "/" | "%" | "and"
```

Error handling

The parser enters a panicmode upon reading an unexpected token. Panicmode exits at the end of a line, but a permanent error flag is raised, preventing moving on from parsing.

AST

After parsing, the parse tree is simplified by the `ParseTreeWalker` class using the visitor pattern. The new AST is then traversed by the `Decorator` class, which performs type checks on expressions and verifies that variables are in scope. The following classes represent the AST.

```
class IRNode {
public:
  mutable std::string type = "void";
  mutable std::string name;
  mutable std::list<std::string> errors;
  mutable std::list<std::string> errors_in;
  mutable std::list<std::string> errors_out;
  mutable std::map<std::string, std::string> symtab; // name,type
  virtual void accept(IRVisitor *v) = 0;
  void appendError(const std::string s) const { errors.emplace_back(s); }
  void appendError_in(const std::string s) const { errors_in.emplace_back(s); }
  void appendError_out(const std::string s) const {
    errors_out.emplace_back(s);
  }
};

class Program : public IRNode {
public:
  std::list<Function *> functions;
  Scope *scope;
  void accept(IRVisitor *v) override { v->visitProgram(this); }
};

class Function : public IRNode {
public:
  Scope *scope;
  void accept(IRVisitor *v) override { v->visitFunction(this); }
};

class Statement : public IRNode {
public:
  void accept(IRVisitor *v) override { v->visitStatement(this); }
};

class Scope : public IRNode {
public:
  std::list<Statement *> statements;
  void accept(IRVisitor *v) override { v->visitScope(this); }
};

class If : public Statement {
public:
  Scope *scope1;
  Scope *scope2;
  void accept(IRVisitor *v) override { v->visitIf(this); }
};

class While : public Statement {
public:
  Scope *scope;
  void accept(IRVisitor *v) override { v->visitWhile(this); }
};

class Declare : public Statement {
public:
  std::list<std::string> names;
  void accept(IRVisitor *v) override { v->visitDeclare(this); }
};
```

```
class Assign : public Statement {
public:
  Expr *expr;
  void accept(IRVisitor *v) override { v->visitAssign(this); }
};

class Call : public Statement {
public:
  std::list<Expr *> args;
  void accept(IRVisitor *v) override { v->visitCall(this); }
};

class Return : public Statement {
public:
  Expr *expr;
  void accept(IRVisitor *v) override { v->visitReturn(this); }
};

class Read : public Statement {
public:
  std::list<std::string> names;
  void accept(IRVisitor *v) override { v->visitRead(this); }
};

class Assert : public Statement {
public:
  Expr *expr;
  void accept(IRVisitor *v) override { v->visitAssert(this); }
};

class Expr : public IRNode {
public:
  void accept(IRVisitor *v) override { v->visitExpr(this); }
};

class Literal : public Expr {
public:
  std::string value;
  void accept(IRVisitor *v) override { v->visitLiteral(this); }
};
```

```
class BinaryOp : public Expr {
public:
  std::string op;
  Expr *left;
  Expr *right;
  void accept(IRVisitor *v) override { v->visitBinaryOp(this); }
};

class UnaryOp : public Expr {
public:
  std::string op;
  Expr *left;
  void accept(IRVisitor *v) override { v->visitUnaryOp(this); }
};

class Variable : public Expr {
public:
  Expr *index;
  void accept(IRVisitor *v) override { v->visitVariable(this); }
};
```

Semantic checks

The semantic checks currently implementd:

- check that Variable is not already in scope upon declaraion
- check that Variable is in scope upon assignment
- check that Variable is in scope upon read
- check that Variable is of the same type as the Expr upon assignment
- check Expr operands are the same type
- check Expr evaluates to Boolean for assert, if, and while

Error handling

As `Decorator` traverses the AST, any error strings of semantic rule violations are passed up to the root node. If the root node contains any errors, they are reported to the user, preventing progressing to code generation.

Standard library

Since module linking is currently not possible in WASM, I decided to simply add all the standard library functions to the same module as the output program. IO and memory management are handled by Javascript and WASM functions in `src/wasmlib/wasmlib.js` and `src/wasmlib/wasmlib.wat`.

Functions

Procedures are simply functions with return type "void".

Simplifications

- procedures are syntactic sugar for functions that return type "void"
- strings are always 20 bytes long

Not implemented

- Real numbers
- Arrays
- functions/procedures
- malloc/free

*

Reflection

- I can not get the parser to properly recognize blocks and while loops. (segfaults)
- I started working on this project way too late. This led to me running out of time while implementing code generation. Much time was also wasted on bugs.
- The JS and WASM standard library and runtime environment is mostly not finished.
- I had a hard time deciding what classes and data to have in the simplified/decorated AST. My choice of strings to represent types was not a good choice.

- Along with the course topics, I learned a lot about WASM working on this project, which was a personal goal of mine. However I fear the quality of the project suffered due to be being overly ambitious and managing my time poorly.

Work log

Date	Hours	Progress
<2022-05-13 Fri>	4	Initialize repo and begin planning
<2022-05-14 Sat>	10	Fix parser bugs
<2022-05-15 Sun>	16	Get parser working with new language
<2022-05-16 Mon>	12	Code generation
TOTAL HOURS	42	

Appendix A (MiniPL description)

Syntax and semantics of Mini-PL (Spring 2022)

Mini-Pascal is a simplified (and slightly modified) subset of Pascal. Generally, the meaning of the features of Mini-Pascal programs are similar to their semantics in other common imperative languages, such as C.

1. A Mini-Pascal program consist of series of functions and procedures, and a main block. The subroutines may call each other and may be (mutually) recursive. Within the same scope (procedure, function, or block), identifiers must be unique but it is OK to redefine a name in an *inner* scope.
2. A **var** parameter is passed *by reference*, i.e. its address is passed, and inside the subroutine the parameter name acts as a synonym for the variable given as an argument. A called procedure or function can freely read and write a variable that the caller passed in the argument list.
3. Mini-Pascal includes a C-style **assert** statement. If an assertion fails the system prints out a diagnostic message and halts execution.
4. The Mini-Pascal operation *a.size* only applies to values of type **array of** *T* (where *T* is a simple type). There are only one-dimensional arrays. Array types are compatible only if they have the same element type. Arrays' indices begin with zero. The compatibility of array indices and array sizes is checked at run time (usually).
5. By default, variables in Pascal are not initialized (with zero or otherwise); so they may initially contain rubbish (random) values.
6. A Mini-Pascal program can print numbers and strings via the predefined special routines *read* and *writeln*. The stream-style input operation *read* makes conversion of values from their text representation to appropriate internal numerical (binary) representation.
7. Pascal is a case non-sensitive language, which means you can write the names of variables, functions and procedures in either case.
8. The Mini-Pascal *multiline comments* are enclosed within curly brackets and asterisks as follows: "`{* . . . *}`".
9. Note that the names *Boolean*, *false*, *integer*, *read*, *real*, *size*, *string*, *true*, *writeln* are treated in Mini-Pascal as "predefined identifiers", i.e., it is allowed to use them as regular identifiers in Mini-Pascal programs.

The arithmetic operator symbols '+', '-', '`*`', and '/' represent the following functions, where T is either "*integer*" or "*real*".

```
"+" : (T, T) -> T            // addition
"-" : (T, T) -> T            // subtraction
"*" : (T, T) -> T            // multiplication
"/" : (T, T) -> T            // division
```

The operator '%' represents integer modulo operation. The operator '+' *also* represents string concatenation:

```
"%" : (integer, integer) -> integer           // integer modulo
"+" : (string, string) -> string              // string concatenation
```

The operators "**and**", "**or**", and "**not**" represent Boolean operations:

```
"or"  : (Boolean, Boolean) -> Boolean         // logical or
"and" : (Boolean, Boolean) -> Boolean         // logical and
```

```
  "not" : (Boolean) -> Boolean                     // logical not
```

The relational operators "=", "<>", "<", "<=", ">=", ">" are overloaded to represent the comparisons between two values of the same type, with the obvious meanings. They can be applied to values of the types *int*, *real*, *string*, *Boolean*.

Context-free grammar for Mini-PL

The syntax definition is given in so-called *Extended Backus-Naur* form (EBNF). In the following Mini-Pascal grammar, the use of curly brackets "{ ... }" means 0, 1, or more repetitions of the enclosed items. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "**bold**"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as "**:=**"). Note that the syntax given below also specifies the precedence of operators (via productions defined at different hierarchical levels).

<program> **::=** "**program**" *<id>* "**;**" { *<procedure>* | *<function>* } *<main-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>*

<array type> **::=** "**array**" "**[**" [*<integer expr>*] "**]**" "**of**" *<simple type>*

<simple type> **::=** *<type id>*

<block> **::=** "**begin**" *<statement>* { "**;**" *<statement>* } ["**;**"] "**end**"

<statement> **::=** *<simple statement>* | *<structured statement>* | *<var-declaration>*

<empty> **::=**

<simple statement> **::=** *<assignment statement>* | *<call>* | *<return statement>* |
<read statement> | *<write statement>* | *<assert statement>*

<assignment statement> **::=** *<variable>* "**:=**" *<expr>*

<call> **::=** *<id>* "**(**" *<arguments>* "**)**"

<arguments> **::=** expr { "**,**" expr } | *<empty>*

<return statement> **::=** "**return**" [expr]

<read statement> **::=** "*read*" "**(**" *<variable>* { "**,**" *<variable>* } "**)**"

<write statement> **::=** "*writeln*" "**(**" *<arguments>* "**)**"

<assert statement> **::=** "**assert**" "**(**" *<Boolean expr>* "**)**"

<structured statement> **::=** *<block>* | *<if statement>* | *<while statement>*

<if statement> **::=** "**if**" *<Boolean expr>* "**then**" *<statement>* |
"**if**" *<Boolean expr>* "**then**" *<statement>* "**else**" *<statement>*

<while statement> **::=** "**while**" *<Boolean expr>* "**do**" *<statement>*

<expr> **::=** *<simple expr>* |
<simple expr> <relational operator> <simple expr>

<simple expr> **::=** [*<sign>*] *<term>* { *<adding operator> <term>* }

<term> **::=** *<factor>* { *<multiplying operator> <factor>* }

<factor> **::=** *<call>* | *<variable>* | *<literal>* | "**(**" *<expr>* "**)**" | "**not**" *<factor>* | *<factor>* "**.**" "*size*"

<variable> **::=** *<variable id>* ["**[**" *<integer expr>* "**]**"]

<relational operator> **::=** "=" | "<>" | "<" | "<=" | ">=" | ">"

<sign> **::=** "+" | "-"

<negation> **::=** "**not**"

<adding operator> **::=** "+" | "-" | "**or**"

<multiplying operator> **::=** "*" | "/" | "**%**" | "**and**"

Lexical grammar

<id> **::=** *<letter>* { *<letter>* | *<digit>* | "**_**" }

<literal> **::=** *<integer literal>* | *<real literal>* | *<string literal>*

<integer literal> **::=** *<digits>*

<digits> **::=** *<digit>* { *<digit>* }

<real literal> **::=** *<digits>* "." *<digits>* ["**e**" [*<sign>*] *<digits>*]

<string literal> **::=** "\"" { *< a char or escape char >* } "\""

<letter> **::=** a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
p | q | r | s | t | u | v | w | x | y | z | A | B | C |
D | E | F | G | H | I | J | K | L | M | N | O | P
| Q | R | S | T | U | V | W | X | Y | Z

<digit> **::=** 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol or keyword> **::=** "+" | "-" | "*" | "%" | "=" | "<>" | "<" | ">" | "<=" | ">=" |
"(" | ")" | "[" | "]" | "**:=**" | "." | "," | ";" | ":" | "**or**" |
"**and**" | "**not**" | "**if**" | "**then**" | "**else**" | "**of**" | "**while**" | "**do**" |
"**begin**" | "**end**" | "**var**" | "**array**" | "**procedure**" |
"**function**" | "**program**" | "**assert**" | "**return**"

<predefined id> **::=** "*Boolean*" | "*false*" | "*integer*" | "*read*" | "*real*" | "*size*" |
"*string*" | "*true*" | "*writeln*"

Appendix B (Project description)

Code Generation Project 2022: *Mini-Pascal compiler*

Implement a compiler for the Mini-Pascal programming language. The work has two parts: firstly, a front end making a full syntactic and semantic analysis of a Mini-Pascal program; secondly, a back end that generates target code. The language analyzer must correctly recognize and process all valid (and invalid) Mini-Pascal programs. It should report syntactic and semantic errors, and then continue analyzing the rest of the source program. It must construct an AST and do a semantic analysis on this internal representation. If the given program was found free from errors, the back-end generates target code that can be subsequently executed. Note that you cannot use any ready-made automatic tools to generate target code: the task of this project is to *implement* the algorithms for such a tool.

Implementation requirements and grading criteria

You are expected to properly use and apply the compiler techniques taught and discussed in the course lecture materials and exercises. C# is used as the implementation language by default. Generating C as the target language is recommended as a reliable and probably the easiest choice (the option 1 below). However, you can choose an alternative target language among the following list of options, but these may need extra study and design work.

1. Simplified C, using only the lowest-level features of C, as a sort of portable assembler. The restrictions are the following. (a) Structured control statements (such as, **if**, **while**, **for**) are not allowed. Instead, use unconditional **goto** statements or simple conditional **if-goto** statements for altering the sequence of execution to other parts of the program.

   (b) Expressions, too, must be in a very simplified form. Only one call operation **or** one primitive operation is allowed per expression. Expressions may not contain parentheses (but for a call to enclose its list of arguments) nor the conditional-expression operator ("**?:**").
2. The .NET *System.Reflection.Emit* namespace contains built-in API classes that allow a C# program to emit metadata and Microsoft common intermediate language (CIL) and optionally generate a PE file (.exe) on disk. These ready-made classes are useful for script engines and compilers.
3. Design and build your own software tools to generate CIL, either in symbolic (textual) or in binary form.
4. JBC (Java Byte Code), in binary format (i.e., a Java class file). JVM includes no official symbolic assembly code.
5. A target platform and target language of your choice. This may be a low-level target language, such as the *WebAssembly* bytecode (binary or text format) or a more high-level programming language, such as *JavaScript* or *asm.js*, or some other appropriate generally available target language.

The emphasis of the grading is the quality of the implementation: its overall architecture, clarity, and modularity. Pay attention to programming style and commenting. Grading of the assignment will consider (undocumented) bugs, level of completion, and its overall success (solves the problem correctly). The evaluation will particularly cover technical advice and techniques given by the course. You must make sure that your compiler system can be run and tested on the development tools available at the CS department.

Documentation

Write a report on the assignment, as a document in PDF format. The title page of the document must show appropriate identifications: the name of the student, the name of the course, the name of the project, and the date and time of delivery.

Describe the overall architecture of your language processor with UML diagrams. Explain the diagrams. Clearly describe the testing process and the design of test data. Tell about possible shortcomings of your program (if well documented and explained they may be partly forgiven). Give instructions how to build and run your compiler. The report must include the following parts:

1. The Mini-Pascal token patterns as *regular expressions* or, alternatively, as *regular definitions*.
2. A *modified context-free grammar* that is more suitable for recursive-descent parsing, and techniques (backtracking or otherwise) used to resolve any remaining syntactic problems. These modifications must not affect the language that is accepted.
3. Specify *abstract syntax trees* (AST), i.e. the internal representation for Mini-Pascal programs. You can use UML diagrams or alternatively give a syntax-based definition of the abstract syntax.
4. *Language implementation-level decisions*. Many programming languages have left items (e.g. evaluation orders, or data representations for values) to an implementation. A language may also allow but does not require for a specific feature. For example, C and C++ do not specify how numbers should be represented (can use efficient native machine-based values), or in what order some list of expressions are evaluated. Usually evaluation proceeds in left-to-right order, but an implementation may have the freedom to use whatever ordering it may prefer for optimizations. Identify any such relevant issues as related to Mini-Pascal and its definition, and specify the decicions made for your own implementation. Explain your choices.
5. *Semantic analysis*. Make a comprehensive list of all the semantics rules and checks needed for Mini-Pascal programs. You can use this list when you design your implementation and inputs for testing.
6. The major problems concerning *code generation*, and their solutions. What were the most problematic or demanding issues (language constructs, features, behaviour) when translating from the source language to the target language. Explain your solutions and discuss how well they worked out.
7. *Error handling* strategies and solutions used in your Mini-Pascal implementation (in its scanner, parser, semantic analyzer, and code generator).
8. Include your work hour log in the documentation. For each day you are working on the project, the log should include: (1) date, (2) working time (in hours), (3) the description of the work done. And finally, the total hours spent on the project during the project course.

For completeness, include this original project definition and the Mini-Pascal specification as appendices of your document. You can refer to them when explaining your solutions.

Delivery of the work

The final delivery is due at 12:00 on Monday 16 of May, 2022.

The work should be returned to the course Moodle page, in a zip form. This zip (included in the e-mail message) should contain all relevant files, within a directory that is named according to your name. The deliverable zip file must contain (at least) the following subfolders.

```
<user>
  ./doc
  ./src
```

When naming your project (.zip) and document (.pdf) files, always include your name and the packaging date. These constitute nice unique names that help to identify the files later. Names would be then something like:

project zip: `user_proj_2022_03_14.zip`
document: `user_doc_2022_03_14.pdf`

More detailed instructions and the requirements for the assignment are given in the exercise group. If you have questions about the folder structure and the ways of delivery, or in case you have questions about the whole project or its requirements, please contact me.