MiniPascal compiler documentation

Timo Mäki
Code generation (spring 2016)
MiniPascal compiler

May 22, 2016

1 Architecture

The program is written in C#6, and a .NET 4.6.1 compatible environment is expected for the execution of the compiler, and the resulting CIL bytecode. The program is build using Visual Studio 2015, and executed on command line, eg. "MiniPascal.exe myProgram.txt". The compiler includes a test suite that uses NUnit test framework. The tests are run using NUnit3 test adapter for Visual Studio 2015. The test adapter must downloaded to visual studio as an extension. The NUnit framework can be downloaded through Nuget with package restoring. The tests builds an executable, which output is read by the tests. Note that the tests may fail sometimes, if the process read access is not released in time. A rerun fixes the problem. Tests also do not test all static analysis errors, though most tests still fail with incorrect input.

The lexer is defined by three types, *Scanner*, *TokenConstruction*, *IScanner-State*. The Scanner keeps reading the input, the scanner state is moved according to input and the TokenConstruction holds the build tokens. The scanner states form a finite state machine.

The input stream for the scanner is called *SourceStream*. Its backing storage differs when run with the commandline and when run with the test runner.

The parser is defined by a partial class, composed of three files. *Parser-Statements* contains grammar rules for statements, *ParserExpressions* contains grammar rules for expressions. *Parser* contains the rest, like types, blocks and helper methods.

Most grammar rules contain their own class for results. For example, there is a class for If statement, while etc.

Most of the operators (and .size) are their own classes, and are attached to their *SimpleType*. *MiniPascalType* contains the SimpleType and a flag whether the type is array.

Semantic analysis involves the *Scope* class. It contains the identifiers of the current, and the previous, scope. The analysis is started by a check of indentifiers, that they are declared and unique in the scope. Second it performs the type check of the operations. No operation is allowed for two different types, although certain operations may return a different type than the original types. The scope class is discarded after the analysis.

Like semantic analysis, code generation is performed in the statement and expression classes themselves. The *CILEmitter* class holds the necessary Reflection.Emit types for generation. For example, it stores the variable and procedure names. In general, there is no need to import Reflection namespace outside this class.

2 Known missing featurs

The compiler aborts when it counters an error. The implementation does not support using the predefined indentifiers as indentifiers.

3 Language implementation-level decisions

For all programs, the limitations of the Common Language Runtime apply. In variable declarations, integers and reals are set to 0, booleans to false and strings to empty string literal. Same applies for arrays, expect for strings which are set to null. Integers and reals are 32 bit. Booleans are internally integers.

For string comparison, the BCL String.Compare is used. The "a" < "b" comparison returns true, since a is before b in alphabetic ordering. For Booleans the integer comparision is used. String.Equals is used for string equality comparison. Integers, reals and booleans uses the CIL instructions for the previous. String.Concat for string concatenation.

All arguments of method calls are evaluated left to right. All equal operations are also evaluated left to right (eg. in 1 + 2 + 3, the 1 + 2 is first counted).

Method writeln is a series of calls to Console.Write. Writeln does not automatically add new line character sequence (since it is not required, and is quite easy to add).

Read is treated as a series of assigments. It calls to Console.ReadLine, and parses the result when needed. Real parsing uses InvarianCulture from System.Globalisation. Parse failures raise an unhandled exception from the BLC library.

Assert is an If statement, calling Environment.Exit(-1) when failing.

Functions are static methods in the emitted code (and procedures are functions with void as return type). All variables are allocated to the stack, and all previously allocated variables are passed by reference to the method. A scope can have up to 256 variables at the same time.

4 Major problems

The largest implementation issue was the by-reference variables of methods. Particulary, the assigments, loads and arguments had to be done differently, depending wheter the variable was local stack, local parameter, array, reference parameter or reference scope. The solution was to ensure all possible variables are always available. Variables are accessed through a series of if-else statements (like in Assigment statement or variable operand).

5 Semantic analysis

- Assigment type
- Operator type
- Operator exists for type
- Declared identifier
- Unique identifier
- Array index integer type
- While/If expression boolean type
- Method call parameter count match
- Reference only for variables
- Size for arrays only

Notably, there is no check that return is called, or that its type is correct.

6 Modified grammar

1. Simple statement should lead into new reference start rule, which would then split to call and assigment, since both of them can start with an identifier.
2. Factor should contain same kind of reference start rule as previously, for call and variable.
3. If statement should contain only: ”if” <Boolean expr> ”then” <statement> [”else” <statement>].
4. In factor, size should be added to the end of every statement, or create a new rule where first a factor is read, and optionally size.

7 Abstract Syntax Tree

1. AbtractSyntaxTree has ScopedProgram
2. ScopedProgram has IStatements
3. IStatement is implemented by Assert, Assigment, Read, Declaration, If, NoReturn Call, Print, Procedure, Return, While
   (a) Assert has IExpression
   (b) Assigment has an IExpression and identifier
   (c) Declaration has identifiers and their type
   (d) If has IExpression and two IStatements
   (e) NoReturnCall has a Call
   (f) Print has IExpressions
   (g) Procedure has a ScopedProgram, MiniPascalType, identifier and variables with their identifiers, types and reference flag
   (h) Read has assigment statements
   (i) Return has an IExpression
   (j) While has an IExpression and a ScopedProgram
4. IExpression is implemented by Expression and SignedExpression
   (a) Expression has IExpressions or IOperands and their operators
   (b) SignedExpression is Expression and has a sign
5. IOperand is implemented by Voolean, Integer, String and Real literals, Call, ExpressionOperand, Size, Unary, VariableOperand and Reading
   (a) Call has an identifier and IExpressions
   (b) Reading has variable and its type
   (c) ExpressionOperand has IExpression
   (d) Size has and IOperand
   (e) Unary has an IOperand and operator
   (f) VariableOperand has identifier and its type

Code Generation Project 2016: *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 Mini-Pascal; 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 all 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 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 considerably more design work.

1. Simplified C, using only the lowest-level parts of C, as a sort of portable assembler. The restrictions are the following. (1) No structured control statements (such as, **if**, **while**, **for**) are allowed. Instead, use only **goto** statements or simple **if-goto** statements for altering the sequence of execution to other parts of the program. (2) Expressions 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 generally useful for script engines and compilers.

3. Design and build your own software tools to generate CIL, either in symbolic 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 LLVM intermediate representation) or a high-level programming language (such as *JavaScript*), 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 Java 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 choises.
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).

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 23 o'clock (11 p.m.) on Sunday 22 of May, 2016. After the appointed deadline, the maximum points to be gained for a delivered work diminishes linearly, decreasing two (2) points per each hour late.

The project must be stored in a zip form. This zip should include all relevant files (including comprehensive sample of source and targets programs), contained in a directory that is named according to your unique department user name. The deliverable zip file must contain (at least) the following subfolders.

```
  <username_ . . >
    ./doc
    ./src
```

Compress (zip) the whole folder and deliver its *address* to the exercise assistant.

When naming your project (.zip) and document (.pdf) files, always include your CS user 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: username_proj_2016_05_22.zip
document: username_doc_2016_05_22.pdf

When delivering by e-mail, send (cc) extra copies of the mail message to (1) yourself and (2) juha.vihavainen (at) helsinki.fi. You can then easily confirm that your mail message was really transferred and contained valid data. These extra copies also serve as backups and affirmations in case of any mix ups or failures in delivery. More detailed instructions and the requirements for the assignment are given in the exercise group sessions. 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 the teaching assistant Jiri Hamberg.

Syntax of Mini-Pascal (Spring 2016)

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. Mini-Pascal allows nested functions and procedures, i.e. subroutines which are defined within another. A nested function is invisible outside of its immediately enclosing function (procedure / block), but can access preceding visible local objects (data, functions, etc.) of its enclosing functions (procedures / blocks). Within the same scope (procedure / function / 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 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 usually checked at run time.
5. By default, variables in Pascal are not initialized (with zero or otherwise); so they may initially contain rubbish values.
6. A Mini-Pascal program can print numbers and strings via the predefined special routines *read* and *writeln*. The stream-style IO makes conversion of values from their text representation to their 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 syntax notation for Mini-Pascal

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).

Context-free grammar

<program> ::= "**program**" *<id>* "**;**" *<block>* "**.**"

<declaration> ::= "**var**" *<id>* { **,** *<id>* } "**:**" *<type>* |
"**procedure**" *<id>* "(" parameters ")" "**;**" *<block>* |
"**function**" *<id>* "(" parameters ")" "**:**" *<type>* "**;**" *<block>*

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

<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**"

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