OBJ-SUBPCL TRANSPILER

Full term project report

Orfeas – Ioannis Zafeiris
2250

MYE020 – Compilers II

Table of Contents

Introduction

This report details the process we followed during the development of a full-term project for the course “Compilers II” at University of Ioannina, Computer Science and Engineering branch. We will be detailing the tools we used, the techniques we devised, and going step-by-step from the first stage of our project until the last.

Objective

Our objective for this product was to initially obtain an understanding of a provided Pascal EBNF grammar, adapt it in order to support basic object-oriented language principles, and create a fully-featured lexer and parser using the Flex & Bison tools. Afterwards, we had to design a specification for object-oriented principles in the C language (a language that was not built to accommodate such), and using our parser, implement a transpiler that would parse “Objective Pascal” code and produce compilable C code using our previously defined specification.

Languages & Tools

FLEX

Our first tool of choice is Flex (Fast Lexical Analyzer). Flex is the (free) successor to Lex and it is used to generate programs that perform pattern-matching on text. In simpler terms, Flex is a lexical analyzer. Flex works by reading input files (or from standard input if no files are given) for a description of the scanner code to generate. The description code is in the form of rules using standard regular expressions combined with C code. Flex can generate C code by default, which in turn defines the function "yylex()", which can be used to run the scanner. This C code can be integrated into a different application or compiled as-is and linked with the Flex runtime library to produce an executable. When executed, it analyzes its input for occurrences of the previously defined regular expressions and whenever it finds one it executes the corresponding C code.

To get a better understanding of what Flex rules look like, here is a sample description file which counts the words and the lines in a file:

```
%{
      /* This is a C-code block where we can define our */
      /* variables and other code necessary for our lexer. */
      int words = 0;
      int lines = 0;
}%

%%

/* Scan for a sequence of characters containing characters from */
/* a to Z and at the end of the it increment the word count. */
[a-zA-Z]+    { ++words; }

/* Scan for the new line character and when found increment the */
/* new line count. */
\n           { ++lines; }

%%
```

We will be using Flex in order to tokenize input Objective Pascal code for use in our parser.

Bison

Another tool we will be using is Bison. Bison is an open-source GNU implementation of Yacc (Yet Another Compiler Compiler), which provides full backwards compatibility as well as a variety of additional features and extensions. Similar to Yacc, Bison is a general-purpose parser generator, which given an annotated context-free grammar (which consists of rules along with corresponding C code), it generates parsing code employing LALR parser tables. Bison is extensively used in the industry, as it allows for the development of a wide range of language parsers, rom those used in simple desk calculators, to complex programming languages. Bison works in conjunction with Flex to provide the full lexer/parser experience.

Here is an example of a very simple calculator implemented in Flex and Bison:

```
%%

[ \t\n]+     { /* Ignore */ }
[0-9]+       { yyval = strtol(yytext, NULL, 10); return(NUM); }
.            { /* Ignore */ }

%%
```

```
%token NUM

%left '+' '-'
%left '*' */'

%%

expr
       : NUM               { $$ = $1; }
       | expr '+' expr     { $$ = $1 + $3; }
       | expr '-' expr     { $$ = $1 - $3; }
       | expr '*' expr     { $$ = $1 * $3; }
       | expr '/' expr     { $$ = $1 / $3; }
       ;

%%
```

We will be using Bison in combination with Flex in order to parse input Objective Pascal code and generate the necessary structures for our transpiler.

C++

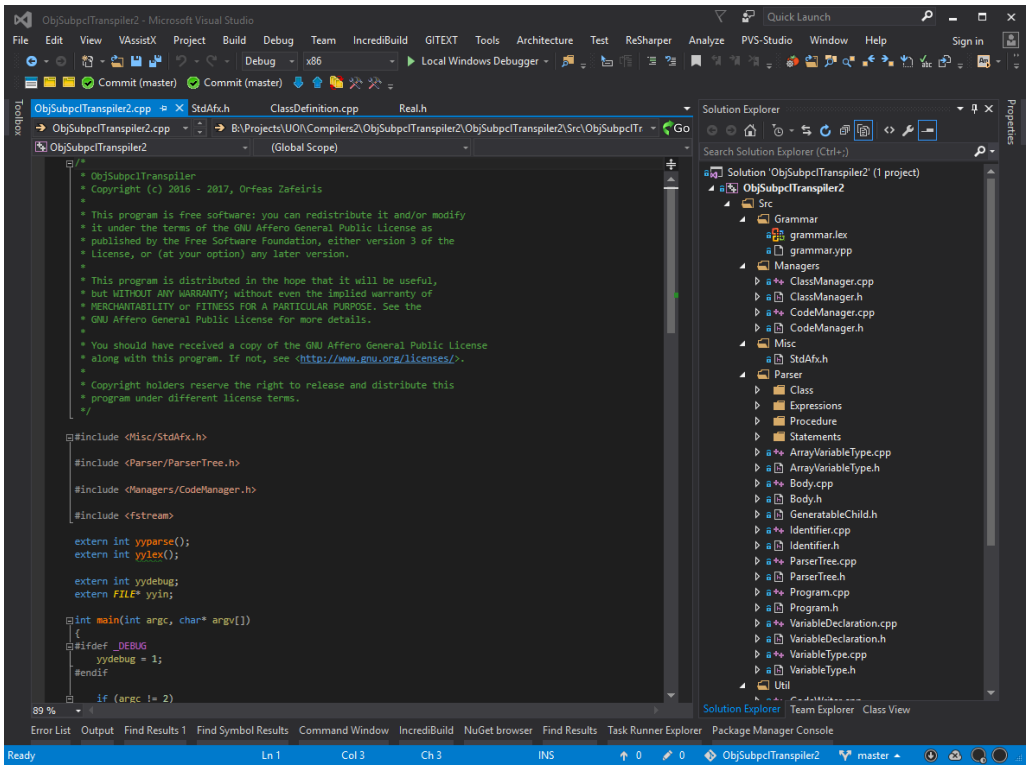
The language of our choice for implementing this project will be C++. C++ is a language that has expanded on the concepts of the C language by providing imperative, object-oriented, and generic programming features, while also providing facilities for low-level memory manipulation. Its performance, versatility, and portability are only some of the reasons why C++ is so widely adopted as the de-facto object-oriented programming language.

C++ also provides an extensive set of standard libraries. This standard library includes useful data structures such as vector, lists, maps, sets, queues, arrays, and more, along with a very useful set of algorithms for performing operations such as searching, sorting, and filtering. On top of that, it also provides platform-agnostic utilities for performing operations such as measuring time, creating and managing threads, generating random numbers, and more.

We specifically selected C++ as it can interface directly with the C/C++ code produced by the Flex and Bison tools, and because it will allow us to write code portable to any platform that has a C++ compiler. Even though we will be using Windows and the MSVC compiler as our development platform, the standardization of the C++ language and its libraries will allow us to very easily ensure that our code can compile and run on any system.

Microsoft Visual Studio

Finally, the development environment of our choice will be Microsoft's Visual Studio 2015. Visual Studio is an integrated development environment (IDE) and supports a variety of programming languages for both editing and debugging out of the box, along with advanced project management capabilities. On top of that, it also provides IntelliSense, which is a smart code completion and suggestion service which aims to make the development experience much better. Visual Studio also comes bundled with compilers and other tools for the various languages it supports.



Since our project will be written in C++, we will be utilizing the various advanced features of Visual Studio in order to improve our development experience and efficiency.

The Grammar

We begin by analyzing the provided Pascal EBNF grammar. It should be noted that this grammar has been modified to exclude some features of pascal, such as string literals, boolean expressions, nil types, and return types.

The Initial Grammar

Below you can see the initial grammar as it was provided to us:

```
program      ::= “program” id “;” body “.”

body         ::= (local)* block

local        ::= “var” (id (“,” id)* “:” type “;”)+
              |  header “;” body “;”

header       ::= “procedure” id “(“ [formal (“;” formal)*] “)”

formal       ::= [“var”] id (“,” id)* “:” type

type         ::= “integer”
              |  “real”
              |  “array” “[“ integer-const “]” “of” type

block        ::= “begin” stmt (“;” stmt)* “end”

stmt         ::= e
              |  l-value “:=” expr
              |  block
              |  call
              |  “if” expr “then” stmt [“else” stmt]
              |  “while” expr “do” stmt
              |  “return”

expr         ::= l-value
              |  r-value

l-value      ::= id
              |  l-value “[“ expr “]”
              |  “(“ l-value “)”
```

```
r-value      ::= integer-const
               |  real-const
               |  “(“ r-value “)”
               |  call
               |  unop expr
               |  expr binop expr

call         ::= id “(“ [expr (“,” expr)*] “)”

unop         ::= “not”
               |  “+”
               |  “-“

binop        ::= “+”
               |  “-“
               |  “*“
               |  “/“
               |  “div“
               |  “mod“
               |  “or“
               |  “and“
               |  “=“
               |  “<>“
               |  “<“
               |  “<=“
               |  “>“
               |  “>=“
```

Required Changes

Even though the grammar itself may seem good as-is, there's a few changes that we will need to perform in order to make it compatible with the characteristics of the C language.

This grammar allows nesting of procedure definitions, something that we can't support in C. We will be changing the grammar as follows to remove that feature:

```
body                ::= (local)* block

local               ::= "var" (id ("," id)* ":" type ";")+
                      |  procedure

procedure           ::= header ";" procedure-body ";"

procedure-body      ::= [formal (";" formal)*] block
```

After these changes, our final grammar will look like this:

```
program      ::= "program" id ";" body "."

body         ::= (local)* block

local        ::= "var" (id ("," id)* ":" type ";")+
               |  procedure

procedure    ::= header ";" procedure-body ";"

procedure-body ::= [formal (";" formal)*] block

header       ::= "procedure" id "(" [formal (";" formal)*] ")"

formal       ::= ["var"] id ("," id)* ":" type

type         ::= "integer"
               |  "real"
               |  "array" "[" integer-const "]" "of" type

block        ::= "begin" stmt (";" stmt)* "end"
```

```
stmt          ::= e
                | l-value ":=" expr
                | block
                | call
                | "if" expr "then" stmt ["else" stmt]
                | "while" expr "do" stmt
                | "return"

expr          ::= l-value
                | r-value

l-value       ::= id
                | l-value "[" expr "]"
                | "(" l-value ")"

r-value       ::= integer-const
                | real-const
                | "(" r-value ")"
                | call
                | unop expr
                | expr binop expr

call          ::= id "(" [expr ("," expr)*] ")"

unop          ::= "not"
                | "+"
                | "-"

binop         ::= "+"
                | "-"
                | "*"
                | "/"
                | "div"
                | "mod"
                | "or"
                | "and"
                | "="
                | "<>"
                | "<"
                | "<="
                | ">"
                | ">="
```

Making Pascal Object-Oriented

Our next step will be designing object-orientation semantics for Pascal, creating a new syntax, defining the necessary changes to the grammar, and incorporating all these changes to the previously defined structures.

Required Functionality

For the purposes of this project we will be implementing the following functionality in Pascal:

- The ability to define classes
  - Each class must have a constructor
  - Each class can have zero, one, or more member variables
  - Each class can have zero, one, or more methods
  - A variable can have a class value or pointer type
- Polymorphism
  - A class can inherit the properties and methods of another class
  - A class can define abstract methods
  - An inheriting class can override abstract methods
  - A class can define dynamic methods
  - An inheriting class can override dynamic methods
  - An inheriting class can call dynamic methods of its parent

The new Syntax

Based on the specifications above we will be defining a new syntax along with the necessary rules for parsing it and using it.

Some examples of our new Object Pascal language syntax are presented below.

```
(* Define a new class with abstract properties *)
class AbstractClass
var a : real;
    b : real;
begin
      procedure constructor(x : real);
      begin
            a := x;
      end;

      abstract procedure calculateB();
end;
```

```
(* Define a class that inherits and implements AbstractClass *)
class RealClass extends AbstractClass
begin
	procedure constructor(x : real);
	begin
		super.ctor(x);
	end;

	procedure calculateB();
	begin
		b := a * 2.5;
	end;
end;
```

```
(* Define a class with dynamic methods *)
class DynamicClass
var a : real;
begin
	procedure constructor();
	begin
	end;

	dynamic procedure calculateA(b : real);
	begin
		a := b * 2.5;
	end;
end;

(* Define a class that overrides DynamicClass' methods *)
class OverrideClass extends DynamicClass
begin
	procedure constructor();
	begin
	end;

	(* This will override the parents' calculateA() *)
	dynamic procedure calculateA(b : real);
	begin
		a := b * 3.5;
	end;
end;
```

Other than defining classes, we also need a way of defining new variables that are of a specific class. We will be supporting two methods of doing so:

1. Value variables: Class-type variables that get initialized during definition
2. Pointer variables: Class-type variables that are initialized later by the user

Here's an example of how the syntax for these new variable types works:

```
(* Initialize variable x of type SomeClass with arguments 1, 2 *)
var x : SomeClass(1, 2);
```

```
(* Initialize pointer variable x of type SomeClass *)
var x : $SomeClass;

begin
	(* Then instantiate the variable like so *)
	x := new SomeClass(1, 2)
end
```

Finally, we need a way to access class members and methods for defined class-type variables. The syntax for doing so will be as follows:

```
var x : SomeClass();

begin
	(* Access member variable a of object x *)
	x.a;

	(* Call method b of object x *)
	x.b();
end
```

Additions to the Grammar

In order to support the syntax we detailed above, we will be introducing several new rules to our grammar. These rules will not only make the syntax specification more concise, but also allow our parser to work as intended.

Below you can see the rules we will be adding:

```
class-def    ::= class-head [formal (“;” formal)*] class-body “;”

class-head   ::= “class” id [“extends” id]

class-body   ::= “begin” class-ctor (class-proc)* “end”

class-ctor   ::= ctor-head “;” procedure-body “;”

ctor-head    ::= “procedure” “constructor” “(“ [formal (“;”
                 formal)*] “)”

class-proc   ::= procedure
              |  abstract-proc
              |  dynamic-proc

abstract-proc ::= “abstract” header “;”

dynamic-proc ::= “dynamic” procedure

class-type   ::= id “(“ [expr (“,” expr)*] “)
             |  “$” id
```

Incorporating the Grammar Rules

Now that we have added these rules, it's time to incorporate them in the main grammar. We will also be modifying the variable declaration rules in order to allow for class-type variables.

We will also be introducing another change to how our Objective Pascal programs will have to be structured. After our changes, users will have to follow the following structure:

```
program id;
      [variable declarations]
      [procedure declarations]
      [class declarations]
      [main block]
.
```

With all that in mind, let's look to the grammar changes we will be performing:

```
body          ::= ["var" (id ("," id)* ":" type ";")+] (procedure)*
                  (class-def)* block

type          ::= "integer"
                | "real"
                | "array" "[" integer-const "]" "of" type
                | class-type

stmt          ::= e
                | l-value ":=" expr
                | l-value ":=" "new" id "(" [expr ("," expr)*] ")"
                | block
                | call
                | "if" expr "then" stmt ["else" stmt]
                | "while" expr "do" stmt
                | "return"

l-value       ::= id
                | l-value "[" expr "]"
                | l-value "." id
                | "(" l-value ")"

call          ::= id "(" [expr ("," expr)*] ")"
                | l-value "." id "(" [expr ("," expr)*] ")"
```

After all these changes, our final grammar will look like this:

```
program      ::= “program” id “;” body “.”

body         ::= [“var” (id (“,” id)* “:” type “;”)+] (procedure)*
                 (class-def)* block

procedure    ::= header “;” procedure-body “;”

procedure-body ::= [formal (“;” formal)*] block

header       ::= “procedure” id “(“ [formal (“;” formal)*] “)”

formal       ::= [“var”] id (“,” id)* “:” type

type         ::= “integer”
              |  “real”
              |  “array” “[“ integer-const “]” “of” type
              |  class-type

block        ::= “begin” stmt (“;” stmt)* “end”

stmt         ::= e
              |  l-value “:=” expr
              |  l-value “:=” “new” id “(“ [expr (“,” expr)*] “)”
              |  block
              |  call
              |  “if” expr “then” stmt [“else” stmt]
              |  “while” expr “do” stmt
              |  “return”

expr         ::= l-value
              |  r-value

l-value      ::= id
              |  l-value “[“ expr “]”
              |  l-value “.” id
              |  “(“ l-value “)”

r-value      ::= integer-const
              |  real-const
              |  “(“ r-value “)”
              |  call
              |  unop expr
              |  expr binop expr
```

```
call          ::= id “(“ [expr (“,” expr)*] “)”
                | l-value “.” id “(“ [expr (“,” expr)*] “)”

unop          ::= “not”
                | “+”
                | “-“

binop         ::= “+”
                | “-“
                | “*“
                | “/“
                | “div“
                | “mod“
                | “or“
                | “and“
                | “=“
                | “<>“
                | “<“
                | “<=“
                | “>“
                | “>=“

class-def     ::= class-head [formal (“;” formal)*] class-body “;”

class-head    ::= “class” id [“extends” id]

class-body    ::= “begin” class-ctor (class-proc)* “end”

class-ctor    ::= ctor-head “;” procedure-body “;”

ctor-head     ::= “procedure” “constructor” “(“ [formal (“;”
                  formal)*] “)”

class-proc    ::= procedure
                | abstract-proc
                | dynamic-proc

abstract-proc ::= “abstract” header “;”

dynamic-proc  ::= “dynamic” procedure

class-type    ::= id “(“ [expr (“,” expr)*] “)
                | “$” id
```

Making C Object-Oriented

Now that we have established what our Objective Pascal language variant has to look like it's time to make the necessary design decisions in order to effectively transpile it to C. By definition, C is not an object-oriented language. For that reason, we will have to exploit some of its features in order to emulate basic OOP paradigms.

Drawing Inspiration

As mentioned before, C was not designed to be an object-oriented language. In order to design an efficient object-oriented abstraction on top of it for the purposes of our transpiler we will draw inspiration from how the C++ language is structured.

As we mentioned previously, C++ is hugely influenced by C while also being mostly backwards compatible with it. This means that if we look into detail how compilers handle OOP functionality in C, we will most likely be able to translate those into plain C code.

Classes and their Methods

Classes in C++ are very similar to C structures. They are simple datatypes that contain a sequence of member variables and describe an object. One of the most basic differences is that in C++, classes provide access modifiers for their methods and members, something that's not possible with plain C structs. However, that's handled entirely by the compiler and does not actually affect the final compiled code in any way.

Another thing that's different in C++ is that classes can have their own methods defined. In the background, the way this is handled is by introducing a new calling convention during compilation, namely "__thiscall" (for x86 systems; x86_64 systems use the "__fastcall" calling convention which is fairly similar). This convention requires that the user passes the object pointer the method is being called upon to the calling method (either by pushing it to stack last in GCC-based compilers, or via the ecx/rcx register in MSVC). Then, when the user accesses other members or functions using the (usually implied) "this" pointer, that previously passed object pointer is used.

Take this example (compiled using MSVC) for instance:

```
class SomeClass
{
public:
	void SomeMethod()
	{
		a = 123;
	}

private:
	int a;
};

...

x->SomeMethod(); // Where x is an object of type SomeClass
```

This code produces the following asm code:

```
// x->SomeMethod();
mov    ecx, [x]
call   ?SomeMethod@SomeClass@@QAEXXZ

// void SomeMethod();
...
mov    dword ptr [ecx + 0], 123 // this->a = 123;
                  // (a is at offset 0 of the structure)
...
```

From the produced x86 assembly above you can see that when a user calls a method on a class object, the compiler automatically stores a pointer to that object in the ecx register and then uses that register as the "this" pointer where needed.

Handling Inheritance

C++ also provides built-in inheritance, something that we also want to be able to handle in our transpiled C code. Inheritance in C++ is achieved because during compilation, the compiler basically prepends the parent class structures to the child classes. This makes it so the memory layout of the structure takes into consideration not only the members of the inheriting class, but also the inherited class.

At a first glance, this would be tricky to incorporate into C without requiring additional accessors, duplicating code, or performing unnecessary casts. However, we are lucky, since Microsoft introduced a new feature into their C compiler (which has since been adopted by other compilers like GCC as well) which allows users to define anonymous structures as structure members.

Take the following two structures for example:

```
struct ParentStruct
{
	int a;
};

struct ChildStruct
{
	int b;
};
```

We want to make ChildStruct inherit the members of ParentStruct and then access them like so:

```
ChildStruct t;
t.a = 1;
t.b = 2;
```

Anonymous structure members allow us to very easily implement this by performing the following simple change to the ChildStruct structure:

```
struct ChildStruct
{
	struct ParentStruct;
	int b;
};
```

Now, every time we're defining and using a ChildStruct data structure we will also have access to the member variables of the ParentStruct structure while also preserving the same memory layout one would expect in C++ from inheriting classes.

Providing Dynamic and Abstract Methods

We previously discussed the need to provide abstract and dynamic overridable methods in our Objective Pascal implementation. C++ provides a very similar feature called virtual methods.

Virtual Methods are inheritable and overridable methods for which dynamic dispatch is facilitated, which in turn allow for runtime polymorphism. Additionally, C++ provides Pure Virtual Methods, which offer the same functionality as abstract methods in the sense that they need to be implemented by inheriting classes. Since our design is very similar to that of C++, we will be utilizing a similar design.

For each class with virtual methods, the C++ compiler generates what's called a Virtual Method Table (or vtable). This table contains pointers to all virtual methods the class provides. When the compiler creates the underlying data structure for the class, it prepends a hidden member called the Virtual Table Pointer which points to the actual vtable of the class. Inheriting class the override the value of this pointer and make it point to their own vtables, which in turn use their own (possibly overridden) methods. When a caller attempts to call a virtual method of a class object, instead of directly calling the function stub, the compiler uses the vtable pointer to perform a runtime invocation of said method.

Let's look at this mechanism into more detail:

```
class A
{
public:
	virtual void X();
}

class B : public A
{
public:
	virtual void X() override;
};
```

For these two classes, the compiler will generate the following vtables (pseudocode):

```
struct A_vtbl_t
{
	void (__thiscall* X)(A*);
};

static A_vtbl_t A_vtbl
{
	.X = A.X
};

static A_vtbl_t B_vtbl
{
	.X = B.X
};
```

On top of that, it will modify the memory layout of A to add the hidden member variable we discussed before:

```
class A
{
	A_vtbl_t* vptr;
	…
};
```

Then, in the generated constructors for A and B, it will assign the corresponding value to that hidden member variable:

```
A::A()
{
	this->vptr = &A_vtbl;
}

B::B()
{
	this->vptr = &B_vtbl;
}
```

Finally, when someone calls the method "X" on an object of type "A" or "B", instead of directly jumping to its address, the compiler will instead generated code that does something like this:

```
// This:
obj->X();
// Gets "converted" to this:
obj->vptr->X(obj);
```

Parsing Pascal

Now that we have established our design parameters it's time to start implementing our parser.

Writing the Lexer

The first step is writing our lexer rules for Flex. This is a fairly simple process in which we take any pre-defined words and sequences, convert them into regular expressions, and assign them a unique token.

We will begin by defining some basic sequences for our lexer:

```
newline      (\r?\n)
digit        [0-9]
white_space  [ \t](
alpha        [A-Za-z]
alpha_num    ({alpha}|{digit})
id           {alpha}{alpha_num}*
unsigned_int {digit}+
```

And after that, we can define our tokens like so:

```
%%

and          return(AND);
array        return(ARRAY);
begin        return(BEGINT);
div          return(DIV);
...

%%
```

Special handling is needed for ids and integers. In this case, we will be storing the alphanumeric value of the token in a standard C++ string, like so:

```
{unsigned_int} yylval.string = new std::string(yytext, yyleng);
                    return(UNSIGNED_INT);

{id}           yylval.string = new std::string(yytext, yyleng);
                    return(IDENTIFIER);
```

Converting the Grammar

Now that our lexer rules are complete, it's time to start writing the grammar rules. To do so, we will start by converting the EBNF grammar we defined before to a syntax that Bison fully understands. This process mostly consists of replacing string literals with previously defined lexer tokens, splitting out repeated blocks into separate rules, and introducing new alternatives for rules that have optional elements.

For example, the following EBNF grammar rule:

```
formal        ::= ["var"] id (“,” id)* “:” type
```

Will be converted to this:

```
formal
       : VAR id_seq COLON type
       | id_seq COLON type
       ;

id_seq
       : id
       | id_seq COMMA id
       ;
```

It is also necessary to define all of the tokens that we previously used for the Flex rules, like so:

```
%token AND ARRAY BEGINT DIV ...
```

Fixing the Grammar

Our grammar might now seem ready, but it's not. Running it through the Bison generator we can immediately notice that we have several conflicts in our hands. Conflicts are fairly common when writing Bison rules, especially when those rules directly derive from EBNF. Bison is kind enough to provide us with detailed information about all the conflicts present in our grammar, along with basic info about their causes.

```
State 7 conflicts: 2 shift/reduce
State 12 conflicts: 2 shift/reduce
...
```

We can then proceed to fixing those issues directly. By referring to the output file, we can see which rule each state corresponds to, and what part of it is causing the conflicts.

Most issues can be fixed by performing fairly simple changes, like removing empty statements from repeated rules and instead introducing them as alternatives in parent rules.

For instance, these rules are producing some conflict issues.

```
class
       : CLASS id BEGINT class_vars class_body END SEMICOLON
       | CLASS id EXTENDS id BEGIN class_vars class_body END
         SEMICOLON
       ;

class_vars
       :
       | VAR var_def
       | class_vars VAR var_def
       ;
```

We can change them to look like this in order to alleviate them:

```
class
       : CLASS id BEGINT class_vars class_body END SEMICOLON
       | CLASS id BEGINT class_body END SEMICOLON
       | CLASS id EXTENDS id BEGIN class_vars class_body END
         SEMICOLON
       | CLASS id EXTENDS id BEGIN class_body END SEMICOLON
       ;

class_vars
       : VAR var_def
       | class_vars VAR var_def
       ;
```

However, there are some rules that require more attention. Because of the way Bison parses files, the can be ambiguity when it comes to order-dependent operators. For instance, if Bison sees something like “a + b + c” it needs to know whether to interpret it as “(a + b) + c” or “a + (b + c)”. Since Bison cannot make any assumptions about grouping itself, we need to aid it by providing hints on order preference for our various tokens:

```
%left OR
%left AND
%left PLUS MINUS
%left STAR SLASH
%left DIV MOD
%right NOT
%right THEN ELSE
%nonassoc GT LT GTE LTE EQUAL NOTEQUAL
%nonassoc UMINUS UPLUS
```

Additionally, rules that depend on such operators need to be grouped together as a whole, instead of only grouping the operators and applying them on different rules.

For example, the following rules:

```
r_value
       : unop expr
       | expr binop expr
       ;

unop
       : NOT
       | UPLUS
       | UMINUS
       ;

binop
       : PLUS
       | MINUS
       ;
```

Need to be converted to follow the pattern seen below:

```
r_value
       : unop
       | binop
       ;

unop
       : NOT expr
       | UPLUS expr %prec UPLUS
       | UMINUS expr %prec UMINUS
       ;

binop
       : expr PLUS expr
       | expr MINUS expr
       ;
```

The Parser Tree

Even though our Bison grammar description is now complete, the parser itself doesn't do anything. We mentioned before that code can be associated with rules in both Bison and Flex, a fact we will exploit in order to generate our transpiler.

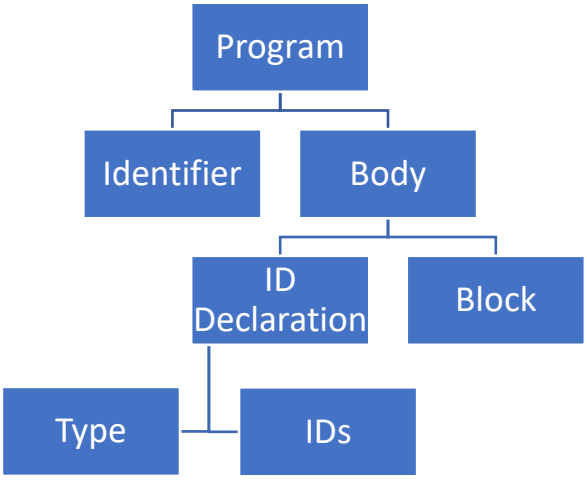
Using Bison and combining it with our own code, we will be constructing a Parser Tree. Each rule in the grammar will correspond to a type that can be added to our tree (in this case a C++ class), which will in turn be able to have its own children (also being parser tree types).

To get a better understanding of what our Parser Tree will look like, take a look at the following example:

```
program test;
	var x : real;

	begin
	end
.
```

The code above will approximately produce the following tree:



Since we mentioned that almost every rule will correspond to a C++ class, here's an example of what a rule and its corresponding class looks like:

```
class_def
       : class_header VAR id_decl class_body SEMICOLON
       | class_header class_body SEMICOLON
       ;
```

```
class ClassDefinition
{
public:
       ClassDefinition(ClassHeader* p_Header,
           VariableSeq* p_Variables, ClassBody* p_Body);

public:
       ClassHeader* m_Header;
       VariableSeq* m_Variables;
       ClassBody* m_Body;
};
```

After the class has been defined in C++, we will change the rule to generate an object of that class type whenever it's encountered:

```
class_def
       : class_header VAR id_decl class_body SEMICOLON
         { $$ = new ClassDefinition($1, $3, $4); }
       | class_header class_body SEMICOLON
         { $$ = new ClassDefinition($1, nullptr, $2); }
       ;
```

We will also need to define a union (a data structure which temporarily holds our parsed types), and a union member association for our rules, as seen below:

```
%union {
       ClassDefinition* class_def;
       ...
}

%type <class_def> class_def
```

It should be noted that for rules that describe sequences we will be using the built-in “std::vector” type, making a new instance of it, and pushing type instances to it every time we encounter one:

```
// In our C++ code we have ProcedureSeq defined as follows:
// typedef std::vector<Procedure*> ProcedureSeq;

procedure_seq
	: procedure { $$ = new ProcedureSeq(); $$->push_back($1); }
	| procedure_seq procedure { $$->push_back($2); }
	;
```

Putting it all Together

Finally, it's time to put everything we just did together and finish implementing our parser. With the C++ side of the rule types being fully implemented, our final Bison parser descriptor will look something like this:

```
%token AND ARRAY BEGINT ...
%token <string> UNSIGNED_INTEGER
%token <string> IDENTIFIER

%union {
       std::string* string;
       Identifier* id;
       ...
}

%type <id> id
...

%left PLUS MINUS
...

%start file

%%

file
       : program { Parser::ParserTree::SetProgram($1); }
       ;

program
       : PROGRAM id SEMICOLON body DOT
         { $$ = new Program($2, $4); }
       ;

...
```

Transpiling to C

Walking the Tree

As we mentioned before, our parser now generates a Parser tree that holds information about all parsed types and their data. In order to transpile to C we will have to walk that tree.

Starting from the root, we will be delegating code generation tasks to various tree elements. From then, those tree elements can decide to either delegate code generation to their children, or handle it themselves. This design choice is necessary because of the dependencies between multiple parsed types.

Managing Content

Simply walking the tree and delegating tasks to its elements isn't enough though. We also need to provide additional utility functions that will allow us to handle code generation and possible implementation issues much more easily.

For starters, we will be implementing a global code generation manager. This manager will be responsible of handling all code writing operations, along with making sure that the code formatting is maintained as expected. During code generation, parser types will be able to call upon that manager, along with their own specific attributes, in order to append their generated code.

Additionally, we will be implementing a global class manager. The class manager will expose easy-to-use utilities for registering, managing, and looking up classes. Using that manager will allow us to ensure input sanity and avoid unexpected compilation errors in the produced code, while also providing a reference for code generation.

Finally, some parser types will be expanded with additional utilities to provide necessary aid during transpilation. For instance, we will be adding member, method, and virtual method lookup functions in our ClassDefinition file, which will allow us to determine how we need to generate variable/method accessors while generating code for expression types.

Generating Code

It is now time to generate our final C code. We will be incorporating everything we mentioned above, all the design principles and methodology, to ensure that the output code is as correct as possible.

Even though the purpose of this report isn't to detail the code generation implementation for every single parser type, let us look at a simple example using the "binop" rule.

The "binop" rule maps to the C++ class "BinaryExpression", whose definition can be seen below:

```
namespace BinaryExpressions
{
	enum type
	{
		Plus,
		Minus,
		...
	};
}

class BinaryExpression :
	public IExpression
{
public:
	BinaryExpression(IExpression * left, IExpression * right,
			BinaryExpressions::type type);

public:
	virtual std::string ToString() override;

private:
	IExpression* m_Left;
	IExpression* m_Right;
	BinaryExpressions::type m_Type;
};
```

Now let's take a closer look at the "ToString()" function, which is the function that generates code for this specific expression:

```
std::string BinaryExpression::ToString()
{
	std:string s_FinalString = "";

	// Serialize the left expression.
	s_FinalString += m_Left->ToString();

	// Serialize the operator.
	switch (m_Type)
	{
		case BinaryExpressions::Plus:
			s_FinalString += " + ";
			break;

		case BinaryExpressions::Minus:
			s_FinalString += " - ";
			break;

		...
	}

	// Serialize the right expression.
	s_FinalString += m_Right->ToString();

	return s_FinalString;
}
```

From the example above you can see that the BinaryExpression type not only generates code for itself, but also delegates code generation to some of its children in a type agnostic way. Since we can't know what expression our children will specifically be, we just trust them to be able to generate code for themselves, similar to how we do, and only handle the portions that are specific to us.

Following a very similar pattern, we can very efficiently implement the entire transpiler, with some extra handling for special cases that can arise here and there.

After generation is done, we simply get the final code buffer from our global code generation manager and write it to a file for the user to compile using his C compiler of choice.

Conclusion

Adapting the pascal language grammar to be object oriented, designing OOP semantics for C, and creating a fully functional transpiler from “Objective Pascal” to C, was definitely no simple task. However, by clever utilization of technologies, tools, techniques, and other resources, we were able to significantly reduce the time it would’ve taken to transition from concept to reality, by streamlining the development process. By having a fully functional transpiler we can now very easily and efficiently write “Objective Pascal” code that can be compiled on virtually any hardware and software environment, without having to actually write a pascal compiler/interpreter for it, since there is a C compiler for virtually any platform imaginable!