Pascal compiler: Design Specification

Final project of “Compile System Design”

Team Member

Shijia Wei 3130000026
Qi Zhu 3130000037
Wen Zhang 3130000014
Yuheng Zhou 3130000011
Contents

1 Introduction ................................................................. 2
  1.1 Document Outline .................................................... 2
  1.2 Background ............................................................ 2
  1.3 Compiling Pipeline ................................................... 2
  1.4 Workload Distribution of the Group .............................. 3
2 System Design ............................................................. 3
  2.1 Implement of Lex and Yacc .......................................... 3
  2.2 Construct Symbol Tables ............................................ 3
  2.3 Code Generation ..................................................... 4
3 Special Feature ............................................................. 6
  3.1 Sophisticated Error Handler ........................................ 6
  3.2 Optimization on Const values ..................................... 7
  3.3 Debug Mode ........................................................... 9
4 Test Results ................................................................. 9
  4.1 Record ................................................................. 9
  4.2 Array ................................................................. 10
  4.3 Recursive Call ....................................................... 10
  4.4 Cascade Function ................................................... 11
  4.5 Calculation .......................................................... 12
  4.6 Final Test Case ...................................................... 13
5 Appendix: Using Instructions ............................................ 13
1 Introduction

Compiler plays an important role in the field of computer science and industry for its ability to convert source code of high-level language to lower-level assembly program. This time we use C language to design a prototype compiler for Pascal language, aiming to obtain a better understanding of the concept of compiling.

1.1 Document Outline

This document is written to introduce the manual instruction, the design process and the attractive feature of our system. The manual instruction includes the method making executive program, using it to generate assembly code in x86 architecture and run the assembly program to get the result. The implementation design is divided into four parts: lexical analysis, syntax analysis, semantic analysis and code generation. Also, the exciting requirements that we achieved in error handling part and debugging mode is displayed in this document.

1.2 Background

As what Prof. Li recommends, Pascal language is used as the source language of our project. For its conciseness and conscientiousness, Pascal is widely used in college education these years, and the grammar it provides comes to be easier to deal with compared to C/C++ or many other languages.

Also, we use x86 assembly language in Solaris Style (also called AT&T Style) as destination language, because it is more convenient to run this assembly code in Linux or Mac OS. Here are the overall circumstance of the project.

<table>
<thead>
<tr>
<th>Source language</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination language</td>
<td>X86 Assembly - Solaris Style</td>
</tr>
<tr>
<td>Project design language</td>
<td>C Language</td>
</tr>
<tr>
<td>Develop environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>Lexical analyzer</td>
<td>Flex</td>
</tr>
<tr>
<td>Syntax analyzer</td>
<td>GNU Bison</td>
</tr>
<tr>
<td>Linker</td>
<td>GCC</td>
</tr>
</tbody>
</table>

Table 1: Project environment

1.3 Compiling Pipeline

In order to handle a variety of requirements of a compiler, the whole process is segmented into four steps. These four steps, lexical analysis, syntax analysis, semantic analysis and code generation, will operate sequentially to form a pipeline of this compiler.

- **Lexical Analysis**
  As the first step of compiling, lexical analyzer will generate a scanner for Pascal language to take out tokens from the source code.

- **Syntax Analysis**
  After tokens are got, Yacc can generate a Finite State Automaton to construct a parse tree to represent the hierarchy of the program for further use. Syntax errors like wrong spellings of reserved words should be returned in this stage.

- **Semantic Analysis**
  Based on the syntax tree, this part recognize the names of different variables, their types and values to construct symbol tables for every functions or procedures. Semantic errors like variable-type mismatch should be returned in this stage.

- **Code Generation**
  It converts the result of semantic analysis to x86 assembly code,
1.4 Workload Distribution of the Group

In this project of compiling system, two members are responsible for the front-end of the system while the other two focus on the back-end. The detailed work each members do is showed below.

- Yuheng Zhou (3130000011): Using Lex and Yacc for syntax tree construction, tree nodes generating and optimizing. He is also the draft person of slides and this document.
- Qi Zhu (3130000037): Using Lex and Yacc for syntax tree construction, set the mechanism of error handling, part of work of symbol table.
- Wen Zhang (3130000014): Write the program of code generation with Shijia Wei, part of work of symbol table, responsible for testing.
- Shijia Wei (3130000026): Write the program of code generation with Wen Zhang, put up with the idea of constant optimization, technical consultant for all group members.

In conclusion, the workload of this project is equally distributed on four group members.

2 System Design

2.1 Implement of Lex and Yacc

The main aim for the two steps at the beginning of the pipeline is to obtain a syntax tree of the program. To get this, we first use tiny.l (the same naming method as the textbook) to turn words into token. For clarity, the name of token is the string “TOKEN,” added with the upper case of the word.

In the stage of syntax tree construction, we optimize the storage mechanism of a tree node. Since some kinds of program instructions have unpredictable numbers of children, two mechanisms are adopt in this system, according to different kinds of instructions.

For example, “while” clauses showed below have and only have 2 children in the syntax tree. So we can just record the pointers of these two nodes in the parent node.

However, there are some clauses whose numbers of children node is unpredictable, such as variable declaration. For the declaration showed below, the number of integers a user define can be as much as he or she wants. Then the left-most child and the sibling child of each are record rather than storing all of them in parent node.

Although nodes of syntax tree have four different types: statement, declaration, type and expression. Each type also have several subtypes, for example, statement is divided into if_statement, while_statement, etc. And all these types and subtypes can be represented by an integer, whose higher digits means the type and lower digits means the subtypes.

2.2 Construct Symbol Tables

One important thing in symbol table construction is the life scope of variables, this problem can be handled by using multiple symbol tables and recording their nested levels. For example, if
the main routine has two functions called A and B, then the nested levels of A and B are both 1, and the nest level of the main routine is 0.

```plaintext
1 var
2  j, z: integer;  {level = 0}
3 begin
4   A();  {level = 1}
5   ...  {get out of A, level = 0}
6   B();  {level = 1}
7   ...  
8 end.
```

The variables defined in the main routine are stored in Table 0. When function A is compiled, a new table called Table 1 is constructed to store variables of function A, the a record called currentNestLevel is changed into 1. At that time, only Table 1 can be accessed to get the information of variable. And when the compiler pointer gets out of function A, the currentNestLevel is set to 0 and Table 1 will be discarded, since all of these variables are out of their scope.

## 2.3 Code Generation

### 2.3.1 Main Steps

The output of code generation stage is a assembly code written in the AT&T x86 format. We then use gcc to perform the linking work. The output can run directly on the Linux system such as Ubuntu. The pipeline in the code generation part can be divided into several steps.

- **Set up headers**
  At the beginning of the program, a number of constant data and codes which will be used later will be inserted.

- **Generate code for the given tree node**
  The first tree node given to this step is the root of the syntax tree. In this step, the program check the type of the given tree node and pass the node to the certain subprogram to handle. This step may be executed due to procedure-procedure, procudure-function, function-procedre and function-function nest declaration.

- **Generate detail code for given tree node is subprogram**
  If the node is a declaration of a function or a procudre, the symbol table will be generated simultaneously. According to the semantic information stored in the syntax tree and symbol

<table>
<thead>
<tr>
<th>zero</th>
<th>type</th>
<th>subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 bit</td>
<td>4bit</td>
<td>4bit</td>
</tr>
</tbody>
</table>

Table 2: Encode method of NodeType
table, the subprogram will emit certain code. Several run-time error detection such as type checking will be executed at this step.

- Generate the sibling tree node of given tree node
  If the given tree node has a sibling, the next tree node to be processed will be set to that sibling node and the program will switch to the second step. If not, the program will switch to the final step.

- Generate ending part
  When all the code should be generated in this tree node is omitted, the program will insert the ending part such stack operations.

2.3.2 Statements and System Calls

We implement the following grammar: assign, for, if, repeat, while, case and label. We design a set of instructions for each of these grammars. We also implement the read and write syscall. The data types that are supported include read include integer, real and char. The data types that are supported include integer, real, char and string.

2.3.3 Types

The types we actually support are integer and real. The expression and calculation of variables of these two types and be handled directly. The char and bool are treated as integer. However, the constant string is also supported in output.

We also implement the record feature and array feature. Non-zero subscript beginning of the array is allowed. The position of a element in an array is recalculated according to the range input in the original code. The nest definition of record is not supported.

2.3.4 Expressions

Our program can handle expressions that include const values and variables. The const real values will be added to data segment of the assembly code. Every const real value will be assigned an unique label. Then flds can be used to load real values. The const integer value are loaded into memory directly.

The type of the expression are traced during code generation. All the operations are transformed to binary operation. The integer-integer operand pair are handled by integer assembly instructions while ingeter-real and real-real operand pair are handled by float point assembly instructions.

The type conversions supported includes bool→char, bool→int, bool→real, char→int, char→real, int→real. If the conversions not listed above are detected during code generation, a error signal will be generated and reported later.

2.3.5 Procedure and Function Calls

Recursive calls of procedures and functions are supported in our project. Also, the variables declared in the “upper” layers can be used in the active procedure or function.

In our program, %ecx performs as fp of the function or procedure. When a funtion or procedure is called, all the parameters are pushed into the stack first. Then the value of %ecx will be pushed into stack acting as an access link. Then the value of %esp will be given to %ecx to set new fp. Finally, the assembly instruction call is used. The space for local variables will be reserved first, then the execution of the procedure or function will start. When the function or procedure ends, assembly instruction ret will be executed. Then all the parameters will be popped out. The values of parameters that have var label will be given to the original variables act as inputs.

We also implement the nest variable feature. The variable declared in the “upper” function or
procedure can be used in the active procedure or function. The layers between two procedures will be fetched from symbol table. Then the access link will be used to find out the fp of the “upper” procedure. Finally, the offset of that variable stored in symbol table will be added to the fp to get the actual memory address of that variable.

3 Special Feature

3.1 Sophisticated Error Handler

3.1.1 Error type

Our pascal compiler support various error detection listed below.

- Type mismatch
  In assignment statements and expressions, the type of left value and right value may be different and can not be automatically casted in code generator.

- Undeclared variable
  Undeclared variables in the program should be detected immediately, especially the left value, which would access unallocated memory later.

- Const value modification
  Modification of const violate the basic idea of const type, we locate this error carefully among every scope.

- Variable redeclaration
  We distinguish variables redeclaration in same scope.

- Function redeclaration
  We distinguish functions and procedures redeclaration in same scope as well.

3.1.2 Error Handler

We implement a sophisticated error handler, which report all errors in the source file. Error handler has two parameters: error type and corresponding syntax tree node and it can output the type of error and specific line number in code. Error handler will mark these incorrect syntax tree nodes as invalid ones, so code generator will be prevented from accessing these nodes at the next time. In this way, we easily handle the cascade errors in code generator. Then our error handler will count and report all of the errors.

All of error detections happen in code generation, since we easily find all distinctive errors in this period, details are as follows.

- Expression
  In expression operation nodes, according our design above, we support type cast in code

| Parameters |
| Access Link |
| Return Address |
| Local Variables |
| Local Temps |
| Parameters of next procedure |

Table 3: Procedure activation record
generator. So only the expression nodes whose running type beyond `real` should be treated as suspected nodes. Additionally, those nodes' running type will be changed into common base type in this period. The mismatched nodes will call error handler and mark as `invalid` nodes.

In expression identifier nodes, we check the undeclared variables in variable list of symbol table. Also, non-array identifiers are detected spontaneously.

- **Assignment statement**
  In assignment statement, we consider type mismatch, undeclared variable and const modification errors. The undeclared variable and type mismatch errors are handled in the same way mentioned above. The difference is that we support return value of function as the right value of assignment statement, so there might be type mismatch between the return value and left value. We search both function list and variable list in symbol table, in order to detect all possible errors.

### 3.1.3 Optimization on error detection

We introduce two major optimizations in our error handler. First, we support the assignment of enumeration variables, which causes undeclared variable error before. Obviously, our parser can not distinguish identifier from enumeration name. In code generator, once we meet an identifier, if the left value is an enumeration variable, we check wether the left value is an enumeration name or not through symbol table. Eventually, we handle the assignment of enumeration rather than reporting undeclared variable errors.

The second optimization is reducing inappropriate errors, because we suffer from tremendous redundant error reports. Actually, it’s caused by our simplification in detecting errors in code generator. Once we mark an syntax node as an `invalid` one, the next time, other expressions or assignment statements access this node, they will also report ambiguous and useless errors. Our optimization is reducing these errors by omitting errors caused by `invalid` children nodes.

Here is an example below to illustrate what we do to improve the performance of error handler.

```plaintext
1 program a;
2 var
3   j , z : integer;
4 begin
5   i := 2; { no declaration }
6   j := 2 * i + i;
7   j := -3;
8   z := i * j;
9   z := i mod j;
10 end.
```

Before Optimization, ambiguous errors about type mismatch reported in line 6/8/9, because `invalid` node `i` is accessed in these lines, while as an invalid node it doesn’t have type actually. After our effort, the compiler seems to be more user friendly.

### 3.2 Optimization on Const values

Another optimization we made is based on the multiple const value operations and assignments. To be more specific, the operation between const type pair `[integer, integer]`, `[integer, real]` and `[real, real]` are optimized before the assembly is generated to the output file.

Before we use our compiler to optimize the generated code, we use the following style to move a constant `integer` and a constant `real` into a certain variable respectively. Due to the limited `latex` package support, the following assembly code are shown in `x86masm` style the actually generated assembly is in `AT&T` style to make it runnable on x86-linux.

When there is a constant operation like
Before Optimization

After Optimization

Figure 3: Error Handler Optimisation

The original way is going to generate twice the const assign. But after we check the “const operation” before generate the assembly, the compiler is responsible for calculating the constant value before it generates the code. So instead of move 10 and 15 respectively into registers and calculate them after, the generated code would only be as shown in following assembly, i.e. directly move result 150 into register for future use:

\begin{verbatim}
mov eax, $150
\end{verbatim}

This optimization apply to all kinds of const types and thier inter-operation, i.e. operation between two types. As we described in the section 3.1, the value is cast to upper type, if allowed. Here is a simple example. Assume we have code:

\begin{verbatim}
var a : int;
a := 10 * 15;
\end{verbatim}

The optimized code would be:

\begin{verbatim}
_mov eax, $150
\end{verbatim}

Also, we maintain a map for those already generated real constants to make sure that no same
real constants are generate twice. For instance, if we have some pascal statements refer to a real constant 1.5000 serveral times. There would be only one label for the constant instead of two or more.

3.3 Debug Mode

As a course project, this compiling system is not as perfect as users expect, so when someone doubts there is something wrong in the compiler, they would like to see some debug information to check the result. That is why we conduct two mode in our system: User Mode and Debug Mode.

The debug mode can be triggered when word “-d” is added after the command line. In this mode, the parse tree of the whole program will be printed on the screen, followed by symbol tables of all nested levels (as Figure 4 and 5 shows).

Customers who only use this system as a compiler will not like to see pages of mediate result but just get the executive program, while program developers and maintainers want to look up the debug information as detailed as possible. So the selection between two modes gives them the opportunity to get what they want.

4 Test Results

4.1 Record

- Source Code

```pascal
program a;
type
tank = record
  wheel: integer;
cost: real;
end;
var
tank_ins: tank;
begin
  tank_ins.wheel := 5;
tank_ins.cost := 11235.56;
 writeln(tank_ins.wheel);
end.
```

Figure 4: Print syntax tree
Figure 5: Print symbol table
As is shown in Figure 6, the record type “tank” is successfully declared and the value of the elements can be changed using assignment operation. And the final values of two elements are printed out by writeln syscall.

4.2 Array

- Source Code

```
program a;
var
i:integer;
a:array[5..10] of integer;
begin
  for i:=5 to 10 do
    a[i]:=10 - i;
for i:=5 to 10 do
  writeln(a[i]);
end.
```

- Output

As is shown in Figure 7, an array is successfully declared and its subscript begins at 5. The final values of the array are printed out by writeln syscall.

4.3 Recursive Call

- Source Code

```
program a;
var
i,j:integer;
function gcd(a:integer;b:integer):integer;
begin
  if b=0 then
    gcd:=a
  end;
end.
```
8     else
9     gcd:=gcd (b , a mod b );
10 end;
11 begin
12     j:=gcd (36 , 24);
13     write (j);
14 end.

• Output

Figure 8: Recursive Call Output

As is shown in Figure 8, a function is called recursively to calculate the greatest common divisor of two integers. The result is printed out by writeln syscall ($gcd(24, 36) = 12$).

4.4 Cascade Function

• Source Code

1 program a;
2 var
3   i: integer;
4 function f(x: integer): integer;
5   var j : integer;
6     function g(x: integer): integer;
7       var k : integer;
8     function h(x: integer): integer;
9     begin
10       i := 10;
11       j := 19;
12       k := 0;
13       h := x;
14     end;
15     begin
16       g := h (x + 1);
17       writeln (i);
18     end;
19     begin
20       f := g (x + 1);
21       writeln(j);
22     end;
23 begin
24       i := f (1);
25       writeln (i);
26     end.

• Output
As is shown in Figure 9, three cascaded functions are declared. The variables declared in “upper” function are used in “lower” function. The results are printed out by writeln syscall.

### 4.5 Calculation

- Source Code

```pascal
program a;
var
i, j, z: integer;
a, b, c: real;
ch: char;
begin
  a := 0.2;
b := -0.3;
c := a + b + 0.001;
writeln(c);
ch := 'a';
c := ch * b;
writeln(c);
i := 21;
c := a * i;
writeln(c);
c := a / b;
writeln(c);
if c < 1 then
  writeln(123)
else
  writeln(0);
if c <= 1.0 then
  writeln(321)
else
  writeln(111);
if c = -0.666667 then
  writeln(666)
else
  begin
    writeln(111);
    writeln(999);
  end;
i := 2;
writeln(i);
j := -3;
writeln(j);
z := i * j;
writeln(z);
z := i mod j;
writeln(z);
```

As is shown in Figure 10, several calculations are performed. The operations between `integer-integer` pairs, `real-real` pairs, `char-real` pairs and `integer-real` pairs are all shown. The results are printed out by `writeln` syscall.

4.6 Final Test Case

The test case that Prof. Li offered has been compiled **successfully** by our compiler, except numerical types. Since the source program and output are too long to be showed in this document, they are attached in another fold called “test”.

5 Appendix: Using Instructions

Get into the “src” directory, input the command “make” to get the executive program, then run “make test” to get the running result of the Pascal program in test.pas.

```
    cd src
    make
    make test
```

If users want to see the detail information of the syntax tree and symbol table, they can execute command as below to enter the debug mode.

```
    ./compiler test.pas -d
```