Bachelor Project in Compiler Construction

Parsing

February 2019

Report from group GROUPNUMBER: 9

Anton Nørgaard (antno16), Bjørn Glue Hansen (bhans09) & Thor Skjold Haagensen (thhaa16)

1 Introduction

In this section we will be discussing the implementation of the tools Flex (Fast lexical analyzer) and Bison, in conjunction with our procedure for creating an abstract syntax tree and finally our methods for printing and verifying the result of our program. At this point we haven't expanded nor limited the language definition of the language.

1.1 Implementation Status

The program functions as intended. The abstract syntax tree, in conjunction with the pretty printer, prints out the tree as it is supposed to look according to our precedence rules. Comment blocks are handled by regular expression states and can not be nested. The grammar is expected to be expanded in the future, e.g. with support for float numbers and negative literals.

2 Parsing and Abstract Syntax Trees

2.1 The Grammar

The grammar of the language demands that there are several things that must be checked when generating parse trees. The parsing mainly focuses on the syntactic correctness, as opposed to semantic correctness of the program. That is to say, we do checks like ensuring that the operators for an expression is one of the valid ones. The definition of functions for how their declaration begin and end with the same identifier is checked during construction of the syntax tree.

During parsing, some of the grammar was found to be ambiguous. There were several shift-reduce conflicts, the most numerous ones being association and precedence of binary operators and handling the dangling else problem. We chose not to rewrite the grammar, but instead state some precedence rules that allows bison to resolve such conflicts.

2.2 Use of the `flex` Tool

Introduction

The core purpose of the lexical analysis phase is to generate a token stream which is passed to bison. we also filter out anything which should be ignored, such as comments and illegal characters. To this end we use the tool Flex.
Flex scans a text document and matches elements with either strings or regular expressions, if an element matches such expression a token represented by an integer value will be returned to bison.

Implementation

Flex allows the program to be in different states according to to some pattern being matched. The states are distinguished by the symbols $< STATE >$ enclosing a state name as shown. In our implementation we use two states, $< INITIAL >$ and $< COMMENT >$. The program starts in the initial state and only enters the comment state when the token "(*" is matched, and returns to the initial state when the first token "*)" is matched. These two tokens define a multi line comment. Because the initial state is reentered on the first "*)" matched, we do not allow for nested multiline comments. Inside the comment state we ignore all other symbols except for newline, which increments the lineCount variable when matched. This variable is used for error messages and debugging information.
To finish comments, we'll quickly touch on single lined comments. A single lined comment in Kitty begins with the symbol #. We match this symbol with the following regular expression: on line 22 $< INITIAL > [\#][\wedge\backslash n][\backslash n]\{lineCount+\}$. Translated to human language this reads, on matching # ignore everything until the newline symbol. We increase lineCount by one as the action. The initial state consists of a combination of keywords, symbols, numbers and identifiers. The keywords and symbols are static. The keywords are placed before the identifier as both will match on the same string and higher placement gives higher priority. Identifiers and integers can vary and are matched with regular expressions. Our regular expression for identifiers is $< INITIAL > [\_a - zAZ][\_a - zA - Z0 - 9]*$. This expression restricts identifiers to start with an underscore or a letter and can then be followed by a combination of underscores and alpha-numerals. Integers has the restriction of not starting with 0 unless it is exactly "0".

2.3 Use of the `bison` Tool

In the bison tool, we handle shift-reduce conflicts in several ways. We specify that all binary operators, such as && , multiplication and addition, are left binding and given priorities identical to the mathematical precedence of the operators. For logical expressions, && binds more tightly than || to make expressions and statements behave like it would be expected of most programs. To handle the dangling-else problem, the "else" keyword binds more tightly than the "then" keyword, in other words an expression such as : if a then if b then s else s2 would be evaluated as: if a then (if b then s else s2). This was more of an arbitrary choice of how to handle conflicts during parsing as there is no universal policy of how to resolve such conflicts.

When it comes to generating the syntax trees, we define a semantic action for each rule. We use the semantic actions to construct each node in the syntax tree.

For instance, the rule

```
var_type:         ID ":" type
                { $$ = make_var_type($1, $3); }
```

defines that in the event we encounter an ID token followed by a colon token and then a type token, we make a variable type node in the syntax tree, storing the semantic values of ID and type.

2.4 Abstract Syntax Trees

To implement the abstract syntax tree and in turn help guide the discovery of syntax errors, we have a global integer called lineCount. We increment the variable as we move further down the parsing of the file. Several of the nodes in the syntax tree consist of a value, a line number for error messages and a kind which helps specifying which rule was applied. The value is a union of various types to have a single struct that support a group of rules.

The root of the tree is the ”body” rule in our CFG, which consists of a declaration list and a statement list:

```
struct _body
{
    array_list *decl_list;
    array_list *stat_list;
};
```

Where each different possible declaration and statement type maintains an enumerate to denote what kind they are and in turn a union that holds the value for that particular kind, in order to define the components of the syntax tree.

2.5 Test

The parsing was tested by checking if certain input produced errors when expected and was otherwise accepted. The structure of the abstract syntax tree was tested by making a pretty printer. The pretty printer should produce the input with added parentheses to show the precedence of expressions, functions, if, if else and while statements.

The results of the tests can be found on the next page.

Below table shows the results of the tests. Op is an abbreviation for binary operators.

#	**Test**	**Expected Result**	**Pass**
	Parsing.sh: Boolean Precedence Tests		
1	Boolean ops are left most associative.	Inner parentheses around first op.	✓
2	&& op has higher precedence than || op.	Inner parentheses around && op.	✓
	Parsing.sh: Comparison Association Tests		
3	Comparison ops are non-associative.	Syntax error message.	✓
	Parsing.sh: Arithmetic Precedence Tests		
4	Ops are left most associative.	Inner parentheses around first op.	✓
5	/ and * have higher precedence.	Inner parentheses around second op.	✓
	Parsing.sh: Identifier Parsing Tests		
6	Accepted Identifiers.	Ouput is the same as input.	✓
7	Identifier starts with a number.	Syntax error message.	✓
	Parsing.sh: Combination Parsing Tests		
8	Precedence for all ops.	Arithmetic ops has highest precedence, then comparison ops and finally boolean ops.	✓
	Parsing.sh: Statement Parsing Tests		
9	Dangling else.	Ouput is the same as input.	✓
	./compiler <*.src files		
10	Use ./compiler on *.src files	Ouput is the same as input with added parentheses.	✓