

Compiler Project #2. Parser Report

2018008277 노주찬

1. Compilation Environment

아래의 환경에서 프로젝트를 컴파일하고 테스트했습니다.

항목	값
OS	Ubuntu 20.04.4 LTS
Architecture	x86_64
C Compiler	gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Lex/Flex	flex 2.6.4
Yacc/Bison	Bison 3.8.1 (http://ftp.gnu.org/gnu/bison/)

apt에서 기본으로 설치되는 bison 대신 더 최신 버전의 bison을 설치해서 사용하였습니다. 또한 bison에서 제공하는 추가적인 기능을 사용하기 위해 POSIX yacc를 emulate 하지 않도록 Makefile 중 y.tab.c 빌드 커맨드의 yacc을 bison으로 수정했습니다.

2. Scanner Modification

AST에 정보를 저장하기 위해 number 토큰이 나타내는 integer 값과 ID 토큰이 나타내는 identifier string 정보가 필요했습니다. 기존 Tiny 구현에서는 mid rule과 전역으로 정의된 savedName을 사용해 identifier string을 저장하고 있었지만, mid rule을 사용하지 않기 위해 scanner 단계에서 number, ID 토큰에 syntax value를 지정하도록 수정했습니다.

56	56	"{"	{return RCURLY;}
57	57	";"	{return SEMI;}
58	58	","	{return COMMA;}
59		- {number}	{return NUM;}
60		- {identifier}	{return ID;}
	59	+ {number}	{yyval.num_value = atoi(yytext); return NUM;}
	60	+ {identifier}	{yyval.id_name = copyString(yytext); return ID;}
61	61	{newline}	{lineno++;}
62	62	{whitespace}	{/* skip whitespace */}
63	63	.	{return ERROR;}

3. Bison

3.1. %union

앞서 언급한 ID, NUM 토큰 이외에도 구현의 편의를 위해서 `TreeNode*` 이외의 다른 타입의 syntax value를 사용할 필요가 있었기 때문에, 아래와 같이 `%union`을 정의하여 syntax value를 사용했습니다.

```
%union {  
    TreeNode *node;  
    char* id_name;  
    ExpType type_spec;  
    int num_value;  
    TokenType op_type;  
}
```

이름	설명
TreeNode* node	아래 기술된 특수한 경우를 제외한 기본 Syntax Value 타입입니다.
char* id_name	ID 토큰인 경우, identifier string을 저장합니다.
ExpType type_spec	Array var_decl을 용이하게 처리하기 위해 type_spec 토큰에 type 정보(int 또는 void)를 저장합니다.
int num_value	NUMBER 토큰인 경우, integer 값을 저장합니다.
TokenType op_type	relop, addop, mulop 등의 토큰에 binary operator를 저장합니다.

3.2. Grammar, Action

대부분의 문법과 Action은 주어진 C-Minus의 문법을 따라가면서 기존 Tiny 구현을 참고하는 식으로 작성했습니다. 예외적으로 기존 Tiny 구현에서 같은 종류의 node (Declaration, Parameter, Statement 등)가 반복되는 list를 저장할 때 매번 sibling을 계속해서 타고 들어가서 저장하는 부분이 비효율적이라고 생각되었습니다. 이 부분을 개선하기 위해 Linked List 형태로 `TreeNode`를 저장하는 `ListNode`를 추가했습니다. `ListNode`에서는 첫번째 child를 list의 첫번째 node로, 두번째 child를 list의 마지막 node로 두는 식으로 구현했습니다.

3.3. Dangling-Else (Closest-If)

Dangling Else 문제를 해결하기 위해서 2_Syntax_Analysis_1_1의 마지막 슬라이드를 참고하여 Statement를 if-else 사이의 statement 위치로 들어갔을 때 뒤에서 등장하는 else와 매칭이 가능한 if statement (unmatched statement)가 포함된 statement와 그렇지 않은 statement로 분리한 다음, unmatched statement가 if와 else 사이의 statement 위치에 등장하지 않도록 문법을 수정했습니다.

4. globals.h (ast.h)

y.tab.h를 include 하기 전에 %union에서 사용되는 타입들의 정의가 이뤄지지 않으면 컴파일 오류가 발생했습니다. 따라서 TreeNode에 관련된 부분을 ast.h라는 별도의 헤더 파일로 분리하여 globals.h가 y.tab.h 앞에서 include 하도록 수정했습니다.

5. Util.c

추가된 Kind에 맞게 출력문을 작성하고, 타입을 출력하는 함수를 추가하여 사용하는 것으로 대부분의 경우를 처리할 수 있었습니다. 단, ListNode에서 일부 예외처리가 필요했는데 실제로 ListNode는 출력되지 않아야 하므로 Indent 처리를 하지 않고, list의 마지막 node를 가리키는 두번째 child는 순회하지 않도록 처리했습니다.

6. Examples and Results

```
1  int main ( void ) {  
2      a ! b; /* unrecognized op '!' */  
3  }  
4  |  
└─> ./cminus_parser test.cm  
  
C-MINUS COMPILATION: test.cm  
Syntax error at line 2: syntax error  
Current token: ERROR: !  
  
Syntax tree:
```

먼저 ERROR 토큰이 잘 처리되는지 확인하기 위해 C-Minus에 정의되지 않은 logical NOT operator이 등장하는 Example로 테스트를 해보았고, 정상적으로 ERROR 토큰이 발견되는 즉시 Syntax error가 발생하는 것을 확인했습니다.

```

/* fun decl test */
void fun (int a, void b, int c[], void d[]) {
/* dangling-else with unmatched if statement in a while statement */
if (a) while (b) if (c[0]) {
/* associativity test */
a = a + b + c[0];
} else
/* cmpnd statement test */
{
int e;
e = a + c[0];
}

/* non-value return test */
return ;
}

/* void-parameter test */
int main (void) {
/* local_decles test */
int a;
int b[4];
void c;
void d[5];

/* if, if-else, addop, mulop, dangling else test */
if (a) if (b[1] * b[2] - b[3] / b[4] + b[5]) a = a + 2; else a = a + 1;

/* var, relop test */
d[1] > 2] = d[3 ≥ 4] = d[5 < 6] = d[7 ≤ 8] = d[9 == 10] = d[11 ≠ 12];

/* function call test */
fun(a, c, b, d);

/* return test */
return 0;
}

```

```

1 C-MINUS COMPILATION: ./test.cm
2
3 Syntax tree:
4 Function Declaration: name = fun, return type = void
5 Parameter: name = a, type = int
6 Parameter: name = b, type = void
7 Parameter: name = c, type = int[]
8 Parameter: name = d, type = void[]
9 Compound Statement:
10 If Statement:
11 Variable: name = a
12 While Statement:
13 Variable: name = b
14 If-Else Statement:
15 Variable: name = c
16 Const: 0
17 Compound Statement:
18 Assign:
19 Variable: name = a
20 Op: +
21 Op: +
22 Variable: name = a
23 Variable: name = b
24 Variable: name = c
25 Const: 0
26 Compound Statement:
27 Variable Declaration: name = e, type = int
28 Assign:
29 Variable: name = e
30 Op: +
31 Variable: name = a
32 Variable: name = c
33 Const: 0
34 Non-value Return Statement
35

```

```

36 Function Declaration: name = main, return type = int
37 Void Parameter
38 Compound Statement:
39 Variable Declaration: name = a, type = int
40 Variable Declaration: name = b, type = int[]
41 Const: 4
42 Variable Declaration: name = c, type = void
43 Variable Declaration: name = d, type = void[]
44 Const: 5
45 If Statement:
46 Variable: name = a
47 If-Else Statement:
48 Op: +
49 Op: -
50 Op: *
51 Variable: name = b
52 Const: 1
53 Variable: name = b
54 Const: 2
55 Op: /
56 Variable: name = b
57 Const: 3
58 Variable: name = b
59 Const: 4
60 Variable: name = b
61 Const: 5
62 Assign:
63 Variable: name = a
64 Op: +
65 Variable: name = a
66 Const: 2
67 Assign:
68 Variable: name = a
69 Op: +
70 Variable: name = a
71 Const: 1
72 Assign:
73 Variable: name = d
74 Op: >
75

```

```

75 Const: 1
76 Const: 2
77 Assign:
78 Variable: name = d
79 Op: ≥
80 Const: 3
81 Const: 4
82 Assign:
83 Variable: name = d
84 Op: <
85 Const: 5
86 Const: 6
87 Assign:
88 Variable: name = d
89 Op: ≤
90 Const: 7
91 Const: 8
92 Assign:
93 Variable: name = d
94 Op: ==
95 Const: 9
96 Const: 10
97 Variable: name = d
98 Op: ≠
99 Const: 11
100 Const: 12
101 Call: function name = fun
102 Variable: name = a
103 Variable: name = c
104 Variable: name = b
105 Variable: name = d
106 Return Statement:
107 Const: 0
108

```

Dangling-else를 포함하여 여러 경우를 포함한 테스트 케이스에서도 의도된 대로 동작하는 것을 확인했습니다.