프로그래밍 기초

> [19반] 정민석_7000

AST 파일 형태 분석

다음은 ast.json에서 main이 정의된 부분만을 추출한 결과입니다.


```
{
   "_nodetype": "FileAST",
   "coord": null,
   "ext": [
            {
         "_nodetype": "FuncDef",
         "body": {
            "_nodetype": "Compound",
            "block_items": [
               {
                  "_nodetype": "Return",
                  "coord": "target.c:9:3",
                  "expr": {
                     "_nodetype": "FuncCall",
                     "args": null,
                     "coord": "target.c:9:10",
                     "name": {
                        "_nodetype": "ID",
                        "coord": "target.c:9:10",
                        "name": "main1"
                     }
                  }
               }
            ],
            "coord": "target.c:8:1"
         },
         "coord": "target.c:7:5",
```

```
        "decl": {
          "_nodetype": "Decl",
          "align": [],
          "bitsize": null,
          "coord": "target.c:7:5",
          "funcspec": [],
          "init": null,
          "name": "main",
          "quals": [],
          "storage": [],
          "type": {
            "_nodetype": "FuncDecl",
            "args": null,
            "coord": "target.c:7:5",
            "type": {
              "_nodetype": "TypeDecl",
              "align": null,
              "coord": "target.c:7:5",
              "declname": "main",
              "quals": [],
              "type": {
                "_nodetype": "IdentifierType",
                "coord": "target.c:7:1",
                "names": [
                  "int"
                ]
              }
            }
          }
        },
        "param_decls": null
      }
    ]
}
```


먼저 최상위의 _nodetype은 이 파일이 AST 파일임을 나타냅니다. 이후 ext의 리스트에는 실제 함수들이 정의되는 등의 코드와 관련한 object가 나옵니다.

ext 내의 각 object는 _nodetype을 가지며, 함수 정의와 관련한 object는 FuncDef라는 _nodetype을 가짐을 확인할 수 있습니다. 더하여 이 object는 크게 body와 decl로 나누어볼 수 있습니다. 만약 body 내의 _nodetype이 Compound라면 함수가 어떤 코드를 실행해야하는 지를 알 수 있습니다. 더하여 decl에서는 다음이 성립합니다.

- 함수의 이름: Decl(type)→FuncDecl(type)→TypeDecl(type)→Identifier(names)
- 반환 타입: Decl(type)→FuncDecl(type)→TypeDecl(type)→Identifier(names)
- 파라미터 타입: Decl(type)→ParamList(args)→ParamList(params)→... →TypeDecl(names)
- 파라미터 이름: Decl(type)→ParamList(args)→ParamList(params)→... →TypeDecl(declname)

특히 파라미터의 경우 ParamList(params) 이후에 포인터 여부에 따라 파싱 방법이 달라졌습니다. 이러한 분석을 토대로 AST 분석기를 구현하였습니다.

AST 분석기 구현 방법

먼저 FuncDef 노드를 재귀적으로 찾습니다. 찾았다면 decl object를 추출하여, 위의 분석한 구조를 토대로 함수의 이름, 반환타입, 파라미터 타입, 파라미터 이름을 각각 출력합니다. 특히 파라미터의 경우 ParamList(params) 이후에 TypeDecl을 찾을때까지 재귀적으로 호출됩니다. 그리고 if 문의 경우 body object에서 _nodetype이 If인 것을 재귀적으로 찾아 if 의 개수를 출력합니다.

실행 결과 중 일부를 첨부하였습니다.

```
함수 이름: expression
  반환 타입: int
  파라미터:
  if 개수: 2

함수 이름: type_name
  반환 타입: void
  파라미터:
  if 개수: 0

함수 이름: statement
  반환 타입: void
  파라미터:
  if 개수: 8

함수 이름: program
  반환 타입: void
  파라미터:
  if 개수: 4

함수 이름: main1
  반환 타입: int
  파라미터:
  if 개수: 0

총 함수 개수: 36
```

AST 파일을 통한 원본 소스코드 복원

제가 복원한 함수는 main, error, accept, expect 함수 입니다. decl에서 함수의 이름, 반환값, 파라미터 등의 정보로 뼈대를 구성하고, body를 통하여 함수가 어떤 코드를 실행해야 하는지를 분석하여 복원하였습니다.

추가적으로 복원한 코드와 analyzer.c의 결과를 비교해보았습니다.

```
// target.c:7:5
int main(void) {
    return main1();
}
/*
    함수 이름: main
    반환 타입: int
    파라미터:
    if 개수: 0
*/

// target.c:27:6
void error(void) {
    exit(1);
}
/*
    함수 이름: error
  반환 타입: void
  파라미터:
  if 개수: 0
*/

// target.c:94:5
int peek(char *s) {
    int i = 0;
    while (s[i] == token[i] && s[i] != 0) {
        i = i + 1;
    }
    return s[i] == token[i];
```

```
}
/*
    함수 이름: peek
  반환 타입: int
  파라미터:
    - 파라미터: char * s
  if 개수: 0
*/

// target.c:102:5
int accept(char *s) {
    if (peek(s)) {
        get_token();
        return 1;
    } else {
        return 0;
    }
}
/*
    함수 이름: accept
  반환 타입: int
  파라미터:
    - 파라미터: char * s
  if 개수: 1
*/

// target.c:112:6
void expect(char *s) {
    if (accept(s) == 0) {
        error();
    }
}
/*
    함수 이름: expect
  반환 타입: void
  파라미터:
    - 파라미터: char * s
```

```
  if 개수: 1
*/
```