SDT 구현과제

과　　목 : 컴파일러 02
학　　과 : 컴퓨터공학
학　　번 : 2020112011
이　　름 : 정찬형
제 출 일 : 2025.11.23.

1. 개요

본 보고서는 기존 MiniC Parser에 Syntax Directed Translation(SDT) 기능을 적용하여 소스 코드를 파싱하고 추상 구문 트리(AST)를 생성하는 과정을 기술한다. 특히 기존 Parser.c 코드에서 SDT 구현을 위해 변경된 사항을 중점적으로 다루며, perfect.mc 예제 파일을 대상으로 한 실험 결과를 분석한다.

2. 기존 Parser.c 수정 내역 (SDT 통합)

기존의 Parser.c는 구문 분석(Parsing)만을 수행하였으나, SDT를 통해 AST를 생성하도록 다음과 같이 코드를 수정 및 확장하였다.

2.1 자료구조 및 전역 변수 추가

- **Node 구조체 활용**: AST 노드를 표현하기 위한 Node 구조체(토큰 정보, 자식 노드, 형제 노드 포인터 포함)를 활용한다.
- **valueStack 추가**: 파싱 상태를 저장하는 stateStack, symbolStack 외에, 생성된 AST 노드의 포인터를 관리하기 위한 valueStack 배열을 추가하였다. c Node* valueStack[PS_SIZE]; // value stack
- **매핑 배열 추가**: 문법 규칙 번호(Rule Number)와 노드 이름(Token Name)을 매핑하기 위한 ruleName[] 및 nodeName[] 배열을 추가하여, 트리 출력 시 사람이 읽을 수 있는 형태(예: PROGRAM, FUNC_DEF)로 출력되도록 하였다.

2.2 AST 생성 및 출력 로직 통합

기존에 별도 파일(sdt.c)로 분리되어 있던 AST 관련 함수들을 Parser.c 내부로 통합하여 의존성을 단순화하였다. - buildTree(int nodeNumber, int rhsLength): - 리덕션(Reduction)이 발생할 때 호출된다. - valueStack에서 rhsLength만큼의 자식 노드들을 꺼내어(pop), 새로운 부모 노드에 연결하고 해당 부모 노드를 반환한다. - printTree(Node* pt, int indent): - 완성된 AST의 루트 노드를 받아 재귀적으로 트리를 순회하며 들여쓰기를 적용해 출력한다.

2.3 parser() 함수 로직 변경

파싱 루프(while(1)) 내의 리덕션(entry < 0) 및 수락(GOAL_RULE) 처리 부분을 수정하였다.

- **Reduction 동작 수정**: 기존에는 스택 포인터(sp)만 감소시켰으나, 수정된 코드에서는 buildTree를 호출하여 트리를 구성하고 결과를 valueStack에 저장한다. ```c // [수정 전] 단순 스택 포인터 감소 // sp = sp - rightLength[ruleNumber];

  // [수정 후] 트리 생성 및 valueStack 업데이트 Node* ptr = buildTree(ruleName[ruleNumber], rightLength[ruleNumber]); sp = sp -

rightLength[ruleNumber]; valueStack[sp] = ptr; // 생성된 서브 트리의 루트를 스택에 저장 ```

- **Accept 동작 수정**: 파싱이 성공적으로 완료되면(GOAL_RULE), valueStack의 최상위에 있는 루트 노드를 printTree 함수에 전달하여 전체 AST를 출력하도록 코드를 추가하였다.

2.4 기타 호환성 수정

- **Scanner.c 수정**: Windows 환경의 strcpy_s 함수를 Linux 표준인 strncpy로 변경하여 컴파일 오류를 해결하였다.
- **Main Driver 분리**: 파서의 진입점을 명확히 하기 위해 sdt_main.c를 생성하고, Parser.c는 파싱 로직 구현체로만 동작하도록 구조를 정리하였다.

3. 실험 환경

- **OS**: Linux (Arch Linux)
- **Compiler**: GCC
- **Target File**: perfect.mc (완전수 계산 예제)

4. 실행 결과

4.1 빌드 및 실행

```
gcc -o sdt-main.out sdt_main.c Parser.c Scanner.c
./sdt-main.out perfect.mc
```

4.2 파싱 과정 (Reduction Log)

파싱 진행 중 발생한 리덕션 규칙 번호들이 출력되었다. 이는 semantic() 함수가 호출되면서 파싱 경로를 추적할 수 있게 해준다.

```
reduced rule number = 13
reduced rule number = 11
reduced rule number = 9
...
reduced rule number = 1
```

4.3 생성된 AST (Abstract Syntax Tree)

perfect.mc 소스 코드에 대한 구문 트리가 정상적으로 생성되었다. PROGRAM을 루트로 하여 변수 선언, 함수 정의, 제어문 등이 계층적으로 구성됨을 확인하였다.

```
 Nonterminal: PROGRAM
      Nonterminal: DCL
           Nonterminal: DCL_SPEC
                Nonterminal: CONST_TYPE
                Nonterminal: INT_TYPE
           Nonterminal: DCL_ITEM
```

```
                Nonterminal: SIMPLE_VAR
                     Terminal: max
                Terminal: 500
      Nonterminal: FUNC_DEF
           Nonterminal: FUNC_HEAD
                Nonterminal: DCL_SPEC
                     Nonterminal: VOID_TYPE
                Terminal: main
...
```

5. 결론

기존의 단순 파서에 SDT 기능을 성공적으로 통합하였다. Parser.c의 리덕션 단계에 buildTree 로직을 주입함으로써, 소스 코드의 문법적 구조를 반영하는 AST를 동적으로 생성할 수 있게 되었다. 실험 결과 perfect.mc의 구조가 올바르게 트리 형태로 변환됨을 확인하였다.