컴파일러 과제 보고서 _2020039070

생성일	@April 3, 2024 6:47 PM
태그	과제 컴파일러
텍스트	2020039070_전종영

- 이 C 프로그램은 LR 파서를 구현한 것입니다. LR 파서는 컴파일러의 구문 분석 단계에서 사용되며, 문법 규칙을 따라 입력 문자열이 유효한지를 판별합니다.
- 구현된 LR 파서는 주어진 입력(`input[MAX]`)에 대해 파싱 테이블(`action_tbl`, `goto_tbl`)을 사용하여 입력을 분석하고, 문법에 따라 입력이 유효한지 검사합니다.
- 파싱 과정은 Shift, Reduce, Accept, Error 동작을 포함하며, 이러한 동작은 구문 분석을 완료하기 위해 순차적으로 수행됩니다.

구현 세부 사항

주요 변수 및 데이터 구조

- `action_tbl`: 파서의 상태와 현재 입력 토큰에 따른 동작을 지정하는 2차원 배열입니다. 양수 값은 Shift 동작, 음수 값은 Reduce 동작, 999는 Accept 동작을 나타냅니다.
- `goto_tbl`: Non-terminal 기호에 대한 상태값을 지정하는 2차원 배열입니다.
- `lhs`: 각 규칙에 대한 left-hand side를 저장하는 배열입니다.(⇒0번째 인덱스는 더미입니다.)
- `rhs_len`: 각 규칙의 right-hand side 길이를 저장하는 배열입니다.(⇒0번째 인덱스는 더미입니다.)
- `token` 및 `NT`: 각각 terminal과 non-terminal 기호(⇒0번째 인덱스는 더미입니다.)를 나타내는 배열입니다.
- `stack`: 파서의 현재 상태를 추적하고 나타내는데 사용되는 stack입니다.

함수

- `main`: 프로그램의 시작점입니다. 사용자로부터 입력(`input[MAX]`)을 받아 파싱을 시작합니다. `$`가 입력되면 프로그램이 종료됩니다.

- `LR_Parser`: 주어진 입력 문자열에 대한 LR 파싱을 수행하는 함수입니다. 파싱 과정에서 다양한 동작을 수행합니다.
- `get_row`, `IsEmpty`, `IsFull`, `push`, `pop`: stack 작업 및 기타 유틸리티 함수입니다.

파싱 과정

1. **초기화**: 스택이 초기화되고 시작 상태(0)가 스택에 push됩니다.
2. **Shift 동작**: 현재 입력 토큰이 터미널 기호와 일치할 때, 해당 토큰과 상태를 스택에 푸시합니다. 이후 입력 포인터가 다음 토큰으로 이동합니다.
3. **Reduce 동작**: `action_tbl`에서 음수 값(예: -2)이 결정되면, 해당 규칙을 사용하여 reduce를 수행합니다. 이 과정에서 스택에서 여러 개의 요소를 pop하고, 적절한 non-terminal을 스택에 푸시합니다.
4. **Accept 동작**: `action_tbl`에서 999를 만나면, 입력 문자열이 성공적으로 파싱되었음을 의미하고 프로그램이 종료됩니다.
5. **Error 동작**: 유효하지 않은 토큰을 만나거나 파싱 테이블에 따른 동작을 수행할 수 없을 경우 오류 메시지를 출력하고 파싱을 중단합니다.

결론

이 프로그램은 LR 파싱 알고리즘을 구현한 예시로서, 구문 분석의 중요성을 강조합니다. 파서가 입력 문자열을 어떻게 분석하는지를 보여주며, 이 과정은 프로그램의 구문적 정확성을 확보하고, 컴파일러의 설계 및 구현에 있어 필수적인 역할을 합니다. 단순히 이론과 의사 코드의 이해에 그치지 않고, C언어를 통해 직접 LR 파서를 구현함으로써, 파서의 작동 원리를 실질적으로 이해하고 파악하는 데 매우 유익한 경험이 되었습니다.

5가지 테스트 결과 예시

1. 정상 실행: d*d+d$

```
Input: d*d+d$
(1) initial : 0  d*d+d$
(2) shift 5:  0d5  *d+d$
(3) reduce 6: 0F3  *d+d$
(4) reduce 4: 0T2  *d+d$
(5) shift 7:  0T2*7  d+d$
(6) shift 5:  0T2*7d5  +d$
(7) reduce 6: 0T2*7F10  +d$
(8) reduce 3: 0T2  +d$
(9) reduce 2: 0E1  +d$
(10) shift 6:  0E1+6  d$
(11) shift 5:  0E1+6d5  $
(12) reduce 6: 0E1+6F3  $
(13) reduce 4: 0E1+6T9  $
(14) reduce 1: 0E1  $
(15) accept
```

2. 정상 실행: d+d*d$

```
Input: d+d*d$
(1) initial : 0  d+d*d$
(2) shift 5:  0d5  +d*d$
(3) reduce 6: 0F3  +d*d$
(4) reduce 4: 0T2  +d*d$
(5) reduce 2: 0E1  +d*d$
(6) shift 6:  0E1+6  d*d$
(7) shift 5:  0E1+6d5  *d$
(8) reduce 6: 0E1+6F3  *d$
(9) reduce 4: 0E1+6T9  *d$
(10) shift 7:  0E1+6T9*7  d$
(11) shift 5:  0E1+6T9*7d5  $
(12) reduce 6: 0E1+6T9*7F10  $
(13) reduce 3: 0E1+6T9  $
(14) reduce 1: 0E1  $
(15) accept
```

3. 토큰 오류: d-d+d$

```
Input: d-d+d$
(1) initial : 0  d-d+d$
(2) shift 5:  0d5  -d+d$
(3) invalid token (-) error
```

4. 토큰 오류: d/d+d$

```
Input: d/d+d$
(1) initial : 0  d/d+d$
(2) shift 5:  0d5  /d+d$
(3) invalid token (/) error
```

5. 문법 오류: dd+d$

```
Input: dd+d$
(1) initial : 0  dd+d$
(2) shift 5:  0d5  d+d$
(3) error
```