2018년도 1 학기 COMPILER PROJECT

본 프로젝트는 수업시간에 배운 방식을 사용하여 Finite State Automata의 하나인 scanner와 parser를 구현하는 것입니다. Scanner를 구현하는 방법에는 여러 가지가 있겠지만, 본 과제에서는 Finite State Automata의 형태를 갖춘 Scanner를 만드는 것이며, 주어진 Grammar로부터 정확한 Transition table을 만드는 일이 중요합니다. 여러분이 작성한 Finite State Automata 의 제어 부분은 아래에 주어진 문법을 기본으로 하여야 합니다.

Parser는 Scanner가 제공하는 Token을 전달받아 문법과의 일치여부를 확인하고 목적코드를 생성하는 Parser와 Code Generator를 구현하는 일입니다. Parser의 구현을 위한 Syntax는 아래에 <표 1>에 주어진 문법을 사용하게 되는 데, Parsing Table을 기반으로 한 Pushdown Automata 를 구현하기를 권장합니다. 그러나, 불가피한 경우에 한해 Recursive Predictive Parser로 구현할 수 도 있습니다. 이를 위한 Input, Output의 형태와 구현을 위한 처리 조건은 아래와 같습니다.

Interfacing은 자유롭게 구현할 수 있습니다만, 파일에 저장해서 전달하는 방법은 지양하는 것이 좋습니다. 실시간으로 전달되는 방법을 취하는 것이 좋습니다. 또한, 여러분이 만든 Parser의 출력 내용을 Code Generator를 통해 Psuedo Code 로 작성하면 되는데, 만일 Assembly Code를 만들어 실행할 수 있으면 추가 점수가 주어집니다. Psuedo Code 코드 생성을 위한 Instruction set 은 <표 2>에 주어져 있습니다.

▶ 입력

☆ 표 1에 주어진 문법에 맞게 작성된 Program File로 File 이름은 사용자가 입력하기로 한다. 즉 여러분이 만든 Compiler를 구동하기 위해서는

compiler2014 <input_file_name> (예를 들면, compiler2014 testfile)

를 입력하도록 한다.

▶ 출력

☆ 표 2에 주어진 Instruction Set을 이용한 목적 Code File로, File 이름은 입력 File의 이름에 Code 라는 Extension을 붙인다. 예를 들면, 상기한 입력의 File이 입력되면 그 출력 File 이름은 testfile.code 가 된다. 또한 symbold table을 당은 testfile.symbol을 제출하기 바랍니다.

▶ 처리 조건

☆ 구현하는 언어는 Unix 환경에서의 C, C++ 혹은 Java 중 선택하기를 권합니다.

☆ 목적 Code는 표 2 에 주어진 명령어(Instruction)들 만을 사용해야 합니다. 부득이한 경우 새로운 명령어를 사용할 때는 Document에 명확하게 명시해야 합니다.

☆ Function은 "BEGIN function_name" 으로 시작하며 "END function_name" 으로 종료합니다.

☆ Register는 충분히 많다고 가정하나 될 수 있으면 적은 숫자의 Register를 사용하도록 하고 사용된 Register의 개수를 Code의 마지막 줄에 출력하도록 합니다.

☆ 부득이한 사정으로 Grammar를 변경한 경우에는 Documentation 에 무엇을 어떻게 그리고 왜 변경했는지를 명기해야 합니다.

▶ 과제 제출물 < [CP]Project2_team#.zip (ex. [CP]Project2_team3.zip) >

1) Print-out of your Source Programs
2) 출력된 Execution File(Pseudo code)
3) Documentation(이 부분은 전체 점수의 50%를 차지합니다)
4) Programmer 자신이 test한 test data files와 결과 file, symbol table file
5) Due Date : 2018년 6월 28일 23시 59분

표 1 < Grammar >

```
prog    ::=   vtype word "(" words ")" block ;
decls   ::=   decls decl
        | ;
decl    ::=   vtype words ";" ;
words   ::=   words "," word
        |     word ;
vtype   ::=   INT | CHAR
        | ;
block   ::=   "{" decls slist "}"
        | ;
slist   ::=   slist stat
        |     stat ;
stat    ::=   block
        |     IF cond THEN block ELSE block
        |     WHILE cond block
        |     word "=" cond ";"
        |     RETURN cond ";"
        | ;
cond    ::=   expr ">" expr
        |     expr "<" expr ;
expr    ::=   term
        |     term "+" term ;
term    ::=   fact
        |     fact "*" fact ;
fact    ::=   num
        |     word ;
word    ::=   ([a-z] | [A-Z])* ;
num     ::=   [0-9]*
```

표 2 < Extended instruction set>

LD Reg#1, addr(or num)	Load var (or num) into the Reg#1
ST Reg#1, addr	Store value of Reg#1 into var
ADD Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 + Reg#3
MUL Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 x Reg#3
LT Reg#1, Reg#2, Reg#3	1 if (Reg#2 < Reg#3), 0 otherwise , store into Reg#1
CALL function_name	Invoke function name
JUMPF Reg#1 label	Jump to label if Reg#1 contains 0
JUMPT Reg#1 label	Jump to label if Reg#1 contains Non-0
JUMP label	Jump to label without condition
MV Reg#1, Reg#2	Set the value of Reg#1 into Reg#2