Alexandre Charron

26606954

Assignment 2 documentation

The first step in modifying the grammar was to replace the star notation with right recursive productions as well as removing left recursions. Here are my initial modifications:

prog -> classDeclList progBody

classDeclList -> classDecl classDeclList | EPSILON

classDecl -> class id { varDeclList funcDefList } ;

varDeclList -> varDecl varDeclList | EPSILON

funcDefList -> funcDef funcDefList | EPSILON

progBody -> program funcBody ; funcDefList

funcHead -> type id ( fParams )

funcDef -> funcHead funcBody ;

funcBody -> { varDeclList statementList }

statementList -> statement statementList | EPSILON

varDecl -> type id arraySizeList ;

arraySizeList -> arraySize arraySizeList | EPSILON

statement -> if ( expr ) then statBlock else statBlock ;

| for ( type id assignOp expr ; relExpr ; assignStat ) statBlock ;

| get ( variable ) ;

| put ( expr ) ;

| return ( expr ) ;

| assignStat ;

assignStat -> variable assignOp expr

statBlock -> { statementList } | statement | EPSILON

expr -> relExpr | arithExpr

relExpr -> arithExpr relOp arithExpr

arithExpr -> term arithExpr1

arithExpr1 -> addOp term arithExprP | EPSILON

sign -> + | -

term -> factor term1

term1 -> multOp factor termP | EPSILON

factor -> variable

| idnestList id ( aParams )

| ( arithExpr )

| fnum

| inum

| not factor

| sign factor

idnestList -> idnest idnestList | EPSILON

variable -> idnestList id indiceList

indiceList -> indice indiceList | EPSILON

idnest -> id indiceList .

indice -> [ arithExpr ]

arraySize -> [ inum ]

type -> float | id | int

fParams -> type id arraySizeList fParamsTailList | EPSILON

fParamsTailList -> fParamsTail fParamsTailList | EPSILON

aParams -> expr aParamsTailList | EPSILON

aParamsTailList -> aParamsTail aParamsTailList | EPSILON

fParamsTail -> , type id arraySizeList

aParamsTail -> , expr

assignOp -> =

relOp -> < | <= | <> | == | > | >=

addOp -> + | - | or

multOp -> \* | / | and

After hand-writing this initial grammar, I passed it through kfG edit to check if the LL(1) conditions were met. Both conditions were not met.

There are a few ambiguities:

1)

factor -> variable | idnestList id ( aParams ) | ...

variable -> idnestList id indiceList

I modified the grammar like so:

factor -> idnestList id factor1 | ...

factor1 -> indiceList | ( aParams )

But there is still an ambiguity, ID is in the FIRST(idnestList) and FOLLOW(idnestList)so after writing out the possibilities for factor and variable, which were the only non-terminals that used idnestList, I rewrote their productions.

Since the first and follow contain ID, I decided to remove idnestList from factor, and added a dot to factor1.

factor -> id factor1 | ...

factor1 -> indiceList . | ( aParams )

This only allows for one iteration of ‘id indiceList .’ but the grammar allows for any amount, so I added another non-terminal factor2

factor -> id factor1 | ...

factor1 -> indiceList factor2 | ( aParams )

factor2 -> . id factor1 | EPSILON

As for variable, it can be rewritten as so in BNF:

Variable ::= id indiceList [. id indiceList]\*

So I rewrote it as so:

variable -> id indiceList variable1

variable1 -> . id indiceList variable1 | EPSILON

Now idnest and idnestlist are useless so I removed them from the grammar.

2)

There’s an ambiguity here:

expr -> relExpr | arithExpr

relExpr -> arithExpr relOp arithExpr

The FIRST(relExpr) and the FIRST(arithExpr) are the same. I modified the grammar as so:

expr -> arithExpr expr1

expr1 -> relOp arithExpr | EPSILON

3)

The FIRST(varDeclList) and FOLLOW(varDeclList) both contain the same terminals, it is because of classDecl -> class id { varDeclList funcDefList } ;

FIRST(funcDefList) contains type, which is the problem, so I decided to pull it out and place it into a new non-terminal called classBody:

classDecl -> class id { classBody };

classBody -> type id varOrFunc | EPSILON

varOrFunc is a new non-terminal that allows to repeat the content of varDeclList or diverge into funcDefList

varOrFunc -> arraySizeList ; classBody | ( fParams ) funcBody ; funcDefList

4)

There is still a problem with varDeclList, FIRST(varDeclList) has id and FOLLOW(varDeclList) also has id. These are the affected rules:

funcBody -> { varDeclList statementList }

varDeclList -> varDecl varDeclList | EPSILON

varDecl -> type id arraySizeList ;

type -> float | int | id

statementList -> statement statementList | EPSILON

statement -> assignStat | ...

assignStat -> variable assignOp expr

variable -> id indiceList variable1

I highlighted the two occasions of id that are an ambiguity. My initial step for fixing this was to replace varDeclList and statementList by a new non-terminal I call funcBlock:

funcBody -> { funcBlock }

funcBlock -> float id arraySizeList ; funcBlock | int id arraySizeList ; funcBlock

| id varOrStatement | statementList | EPSILON

varOrStatement -> id arraySizeList ; funcBlock | indiceList variable1 assignOp expr ;

statementList

I opened up the repetition in varDeclList by creating a production that starts with float and one that starts with int, and then created a new non-terminal for the case where the rule starts with id. varDecl and varDeclList are now useless so I removed them.

This however didn’t remove the ambiguity because FIRST(statementList) still contains id, which is ambiguous with funcBlock -> id varOrStatement. I decided to open up statementList and recrete a new unambiguous rule for each of its productions.

funcBlock -> float id arraySizeList ; funcBlock | int id arraySizeList ; funcBlock

| id varOrStatement | if ( expr ) then statBlock else statBlock ; statementList | for ( type id assignOp expr ; relExpr ; assignStat ) statBlock ; statementList | get ( variable ) ; statementList | put ( expr ) ; statementList | return ( expr ) | EPSILON

The final grammar is below, the changes have been highlighted:

prog -> classDeclList progBody

classDeclList -> classDecl classDeclList | EPSILON

classDecl -> class id { classBody } ;

classBody -> type id varOrFunc | EPSILON

varOrFunc -> ( fParams ) funcBody ; funcDefList | arraySizeList ; classBody

funcDefList -> funcDef funcDefList | EPSILON

progBody -> program funcBody ; funcDefList

funcHead -> type id ( fParams )

funcDef -> funcHead funcBody ;

funcBody -> { funcBlock }

funcBlock -> float id arraySizeList ; funcBlock

| int id arraySizeList ; funcBlock

| id varOrStatement

| if ( expr ) then statBlock else statBlock ; statementList

| for ( type id assignOp expr ; relExpr ; assignStat ) statBlock ; statementList

| get ( variable ) ; statementList

| put ( expr ) ; statementList

| return ( expr ) ; statementList

| EPSILON

varOrStatement -> id arraySizeList ; funcBlock | indiceList variable1 assignOp expr ; statementList

statementList -> statement statementList | EPSILON

arraySizeList -> arraySize arraySizeList | EPSILON

statement -> if ( expr ) then statBlock else statBlock ;

| for ( type id assignOp expr ; relExpr ; assignStat ) statBlock ;

| get ( variable ) ;

| put ( expr ) ;

| return ( expr ) ;

| assignStat ;

assignStat -> variable assignOp expr

statBlock -> { statementList } | statement | EPSILON

expr -> arithExpr expr1

expr1 -> relOp arithExpr | EPSILON

relExpr -> arithExpr relOp arithExpr

arithExpr -> term arithExpr1

arithExpr1 -> addOp term arithExpr1 | EPSILON

sign -> + | -

term -> factor term1

term1 -> multOp factor term1 | EPSILON

factor -> ( arithExpr ) | fnum | id factor1

| inum | not factor | sign factor

factor1 -> ( aParams ) | indiceList factor2

factor2 -> . id factor1 | EPSILON

variable -> id indiceList variable1

variable1 -> . id indiceList variable1 | EPSILON

indiceList -> indice indiceList | EPSILON

indice -> [ arithExpr ]

arraySize -> [ inum ]

type -> float | id | int

fParams -> type id arraySizeList fParamsTailList | EPSILON

fParamsTailList -> fParamsTail fParamsTailList | EPSILON

aParams -> expr aParamsTailList | EPSILON

aParamsTailList -> aParamsTail aParamsTailList | EPSILON

fParamsTail -> , type id arraySizeList

aParamsTail -> , expr

assignOp -> =

relOp -> < | <= | <> | == | > | >=

addOp -> + | - | or

multOp -> \* | / | and

Here are the first and follow sets, generated using kfG edit and <http://hackingoff.com/compilers/predict-first-follow-set> :

|  |  |  |
| --- | --- | --- |
|  | FIRST | FOLLOW |
| prog | class program | $ |
| classDeclList | class EPSILON | program |
| classDecl | class | class program |
| classBody | float id int EPSILON | } |
| varOrFunc | [ ; ( | } |
| funcDefList | float id int EPSILON | } $ |
| progBody | program | $ |
| funcHead | float id int | { |
| funcDef | float id int | float id int } $ |
| funcBody | { | ; float id int } |
| funcBlock | for if get put return float int id EPSILON | } |
| varOrStatement | [ . = id | } |
| statementList | for if get put return id EPSILON | } |
| arraySizeList | [ EPSILON | ; ) , |
| statement | for if get put return id | for if get put return id else ; } |
| assignStat | id | ; ) |
| statBlock | { for if get put return id EPSILON | else ; |
| expr | ( fnum id inum not + - | ) ; , |
| expr1 | < <= <> == > >= EPSILON | ) ; , |
| relExpr | ( fnum id inum not + - | ; |
| arithExpr | ( fnum id inum not + - | ) ; , < <= <> == > >= ] |
| arithExpr1 | + - or EPSILON | ) ; , < <= <> == > >= ] |
| sign | + - | ( fnum id inum not + - |
| term | ( fnum id inum not + - | ) ; , < <= <> == > >= ] + - or |
| term1 | \* / and EPSILON | ) ; , < <= <> == > >= ] + - or |
| factor | ( fnum id inum not + - | ) ; , < <= <> == > >= ] + - or \* / and |
| factor1 | ( [ EPSILON . | ) ; , < <= <> == > >= ] + - or \* / and |
| factor2 | . EPSILON | ) ; , < <= <> == > >= ] + - or \* / and |
| variable | id | = ) |
| variable1 | . EPSILON | = ) |
| indiceList | [ EPSILON | . = ) ; , < <= <> == > >= ] + - or \* / and |
| indice | [ | . = ) ; , < <= <> == > >= [ ] + - or \* / and |
| arraySize | [ | ) ; , [ |
| type | float id int | id |
| fParams | float id int EPSILON | ) |
| fParamsTailList | , EPSILON | ) |
| aParams | ( fnum id inum not + - EPSILON | ) |
| aParamsTailList | , EPSILON | ) |
| fParamsTail | , | ) , |
| aParamsTail | , | ) , |
| assignOp | = | ( fnum id inum not + - |
| relOp | < <= <> == > >= | ( fnum id inum not + - |
| addOp | + - or | ( fnum id inum not + - |
| multOp | \* / and | ( fnum id inum not + - |

DESIGN / IMPLEMENTATION

I developed a top-down predictable table-driven parser for this part of the assignment. To build the table, I manually filled an excel spreadsheet(provided in my submission) with all the rules. For every terminal in the FIRST set of a non-terminal, I put a rule in the column of the terminal and the row of the non-terminal. If EPSILON is in the FIRST set of a non-terminal, I put a rule in the column of every terminal in the FOLLOW set of the non-terminal and row of the non-terminal.

I then replicated the excel spreadsheet as a 2D array of strings manually, which was very error-prone, but through testing I believe I have corrected all transcription mistakes.

For the error detection, I put the string “pop” in every position where a terminal is in the FOLLOW set of a non-terminal, but EPSILON is not in the FIRST set of that non-terminal. When SkipErrors is called, it finds the appropriate row and column in the table, if it returns an empty string, it scans until it finds a rule. If it returns “pop”, it pops the stack and resumes.

TOOLS

For this assignment, the only tools I used were kfG edit, Microsoft Excel and the website

<http://hackingoff.com/compilers/predict-first-follow-set>