Homework 2: Syntax Analysis
CS 421 Compiler Design and Construction

Asad Tariq & Fahad Shaikh

Habib University

Fall 2022

Due: October 16, 2022

1 CFG for TUPLE (The Ultimate Programming LanguagE)

Program $\rightarrow$ *dt id (ParamList)* {*Stmts*}
ParamList $\rightarrow$ *dt id PList*
PList $\rightarrow$ *, dt id PList* | ϵ
Stmts $\rightarrow$ *Stmts DecStmt* | *Stmts AssignStmt* | *Stmts ForStmt* | *Stmts IfStmt* | *Stmts ReturnStmt* | ϵ
DecStmts $\rightarrow$ *dt id OptionalAssign List*
List $\rightarrow$ *, dt OptionalAssign List* | ϵ
OptionalAssign $\rightarrow$ *= Expr ;* | ϵ
AssignStmt $\rightarrow$ *id = Expr ;*
Expr $\rightarrow$ *T E‘*
E‘ $\rightarrow$ *+ T E‘* | ϵ
T $\rightarrow$ *F T‘*
T‘ $\rightarrow$ ** F T‘* | ϵ
F $\rightarrow$ *(Expr)* | *id*
ForStmt $\rightarrow$ *for (Type id Expr ; Expr relop Expr ; id ++)* {*Stmts*}
Type $\rightarrow$ *dt* | ϵ
IfStmt $\rightarrow$ *if (Expr relop Expr)* {*Stmts*} *OptionalElse*
OptionalElse $\rightarrow$ *else* {*Stmts*} | ϵ
ReturnStmt $\rightarrow$ *return Expr ;*

1. $\boxed{\text{3 points}}$ Given the grammar for TUPLE, transform it into $LL(1)$ with First and Follow sets.

Solution:

2 *LL(1) Grammar*

To convert a grammar into a LL(1) grammar we need to do the following:

1. Eliminate left recursion in all production rules of the grammar.
2. Perform left factoring of the grammar.

We shall start with the elimination of left recursion in the grammar. Therefore the production rules of *Stmts* must be altered.
The resulting grammar is as follows:

$Program \rightarrow dt\ id\ (ParamList)\ \{Stmts\}$
$ParamList \rightarrow dt\ id\ PList$
$PList \rightarrow ,\ dt\ id\ PList \mid \epsilon$
$Stmts \rightarrow \overline{Stmts}$
$\overline{Stmts} \rightarrow DecStmt\ \overline{Stmts} \mid AssignStmt\ \overline{Stmts} \mid ForStmt\ \overline{Stmts} \mid IfStmt\ \overline{Stmts} \mid ReturnStmt\ \overline{Stmts} \mid \epsilon$
$DecStmts \rightarrow dt\ id\ OptionalAssign\ List$
$List \rightarrow ,\ dt\ OptionalAssign\ List \mid \epsilon$
$OptionalAssign \rightarrow = Expr\ ; \mid \epsilon$
$AssignStmt \rightarrow id = Expr\ ;$
$Expr \rightarrow T\ E`$
$E` \rightarrow +\ T\ E` \mid \epsilon$
$T \rightarrow F\ T`$
$T` \rightarrow *\ F\ T` \mid \epsilon$
$F \rightarrow (Expr) \mid id$
$ForStmt \rightarrow for\ (Type\ id\ Expr\ ;\ Expr\ relop\ Expr\ ;\ id\ ++)\ \{Stmts\}$
$Type \rightarrow dt \mid \epsilon$
$IfStmt \rightarrow if\ (Expr\ relop\ Expr)\ \{Stmts\}\ OptionalElse$
$OptionalElse \rightarrow else\ \{Stmts\} \mid \epsilon$
$ReturnStmt \rightarrow return\ Expr\ ;$

There is no need for left factoring as there is no production rule that has a resulting production which starts with the same terminal or non-terminal, i.e there is no production rule of the following format:

$$A \rightarrow \alpha A \mid \alpha B$$

We now find the First set of each production rule of this $LL(1)$ grammar:

- FIRST($Program$) = FIRST(dt) = {dt}
- FIRST($ParamList$) = FIRST(dt) = {dt}
- FIRST($PList$) = FIRST(,) $\cup$ FIRST(ϵ) = {, ϵ}
- FIRST($Stmts$) = FIRST($\overline{Stmts}$) = FIRST($DecStmt$) $\cup$ FIRST($AssignStmt$) $\cup$ FIRST($ForStmt$) $\cup$ FIRST($IfStmt$) $\cup$ FIRST($ReturnStmt$) = FIRST(dt) $\cup$ FIRST(id) $\cup$ FIRST(for) $\cup$ FIRST(if) $\cup$ FIRST($return$) $\cup$ FIRST(ϵ) = {dt, id, for, if, return, ϵ}
- FIRST($\overline{Stmts}$) = FIRST($DecStmt$) $\cup$ FIRST($AssignStmt$) $\cup$ FIRST($ForStmt$) $\cup$ FIRST($IfStmt$) $\cup$ FIRST($ReturnStmt$) = FIRST(dt) $\cup$ FIRST(id) $\cup$ FIRST(for) $\cup$ FIRST(if) $\cup$ FIRST($return$) $\cup$ FIRST(ϵ) = {dt, id, for, if, return, ϵ}
- FIRST($DecStmts$) = FIRST(dt) = {dt}
- FIRST($List$) = FIRST(,) $\cup$ FIRST(ϵ) = {, ϵ}
- FIRST($OptionalAssign$) = FIRST(=) $\cup$ FIRST(ϵ) = {=, ϵ}
- FIRST($AssignStmt$) = FIRST(id) = {id}
- FIRST($Expr$) = FIRST(T) = FIRST(F) = FIRST($($) $\cup$ FIRST(id) = {(, id}
- FIRST(E‘) = FIRST($+$) $\cup$ FIRST(ϵ) = {+, ϵ}
- FIRST(T) = FIRST(F) = FIRST($($) $\cup$ FIRST(id) = {(, id}
- FIRST(T‘) = FIRST($*$) $\cup$ FIRST(ϵ) = {*, ϵ}
- FIRST(F) = FIRST($($) $\cup$ FIRST(id) = {(, id}
- FIRST($ForStmt$) = FIRST(for) = {for}
- FIRST($Type$) = FIRST(dt) $\cup$ FIRST(ϵ) = {dt, ϵ}
- FIRST($IfStmt$) = FIRST(if) = {if}
- FIRST($OptionalElse$) = FIRST($else$) $\cup$ FIRST(ϵ) = {else, ϵ}
- FIRST($ReturnStmt$) = FIRST($return$) = {return}

Next we find the Follow set of each production rule of this $LL(1)$ grammar:

- FOLLOW($Program$) = {\$}
- FOLLOW($ParamList$) = FIRST($)$) = {)}
- FOLLOW($PList$) = FOLLOW($ParamList$) = FIRST($)$) = {)}

- FOLLOW($Stmts$) = FIRST(}) = {}}
- FOLLOW($\overline{Stmts}$) = FOLLOW($Stmts$) = FIRST(})= FIRST(}) = {}}
- FOLLOW($DecStmts$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {dt, id, for, if, return, }}
- FOLLOW($List$) = FOLLOW($DecStmts$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {dt, id, for, if, return, }}
- FOLLOW($OptionalAssign$) = FIRST($LIST$) ∪ FOLLOW($DecStmts$) − {ϵ} = {, dt, id, for, if, return, }}
- FOLLOW($AssignStmt$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {dt, id, for, if, return, }}
- FOLLOW($Expr$) = FIRST(*;*) ∪ FIRST(*)*) ∪ FIRST($relop$) = {;,), relop}
- FOLLOW(E‘) = FOLLOW($Expr$) = {;,), relop}
- FOLLOW(T) = FIRST(E‘) ∪ FOLLOW(E‘) − {ϵ} = {+, ;,), relop}
- FOLLOW(T‘) = FOLLOW(T) = {+, ;,), relop}
- FOLLOW(F) = FIRST(T‘) ∪ FOLLOW(T) − {ϵ} = {*, +, ;,), relop}
- FOLLOW($ForStmt$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {dt, id, for, if, return, }}
- FOLLOW($Type$) = FIRST(id) = {id}
- FOLLOW($IfStmt$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {dt, id, for, if, return, }}
- FOLLOW($OptionalElse$) = FOLLOW($IfStmt$) = {dt, id, for, if, return, }}
- FOLLOW($ReturnStmt$) = FIRST($\overline{Stmts}$) ∪ FOLLOW($Stmts$) − {ϵ} = {, dt, id, for, if, return, }}

The First and Follow set of each of the production rules of TUPLE was found using the rules given in the *Dragon Book* in addition to those, found in of *Compilers: Principles, Techniques and Tools* 2^{nd} *Ed. (pgs. 220-222).*

3 *Panic Mode Recovery*

In our implementation of the Parser, we have ensured that when a parsing error is encountered, it is recorded first and then the current token is incremented to point at the next token in the list of all tokens. Therefore, our parser does not halt when it encounters a parsing errors, but rather records it and moves onto the next token. This can be seen in the parsing trace of the *test03.tpl* file (as shown below), which encounters a total of *3* parsing errors but still completes the parsing of the whole program.

```
matched <dt, int>
matched <id, 1>
matched <(>
matched <dt, float>
matched <id, 2>
matched <,>
matched <dt, int>
matched <id, 3>
Parsing Error!
Parsing Error!
matched <{>
In abc()
matched <return>
matched <id, 2>
matched <+>
matched <id, 3>
matched <;>
Parsing Error!
matched <}>
Exiting abc()
EOF
```

Figure 1: The parsing trace of *test03.tpl*