Compilation Techniques

Laboratory 4

Left recursion elimination and common prefix factorization

To implement a Recursive Descent Parser it is necessary for the syntactic rules to follow some rules:

- there must not be left recursion (A ::= A α | β), because when A is implemented as a function, this function would call itself in an infinite recursion. The left recursion elimination can be accomplished with the following formula:

$$\textbf{A ::= A } \alpha_1 \textbf{ | … | A } \alpha_m \textbf{ | } \beta_1 \textbf{ | … | } \beta_n \longrightarrow \begin{array}{l} \textbf{A ::= } \beta_1 \textbf{ A' | … | } \beta_n \textbf{ A'} \\ \textbf{A' ::= } \alpha_1 \textbf{ A' | … | } \alpha_m \textbf{ A' | } \varepsilon \end{array}$$

  Example: „exprOr ::= exprOr OR exprAnd | exprAnd” → „exprOr ::= exprAnd exprOr1” and „exprOr1 ::= OR exprAnd exprOr1 | ε”

- there must not be common prefixes between alternatives (A ::= α β$_1$ | α β$_2$), because if α appears at input, multiple choices can be made. The common prefixes factorization can be made by applying the following formula:

$$\textbf{A ::= } \alpha \beta_1 \textbf{ | … | } \alpha \beta_n \longrightarrow \textbf{A ::= } \alpha \textbf{ (} \beta_1 \textbf{ | … | } \beta_n \textbf{)}$$

The Greek letters mean any expression (rule part). A’ is a new rule which can have any name not yet used. **ε** (epsilon) is the empty rule and it is accomplished by any input sequence, including a void one. The presence of **ε** as a member in an alternative makes that alternative to become optional: **„α | ε” = „α?”**.

Recursive Descent Parser (RDP)

RDP is relatively simple to implement by hand and it offers an increased flexibility. Its implementation is based on the following algorithm.

We consider a list of tokens. The global variable „**Token *crtTk;**” points to the current token (initially the first token in list). The last token has the code **END**.

Each syntactic rule (nonterminal) will be implemented as a separate function. This function will return **true** (1) if starting with the current token (the one pointed by **crtTk**) it is a sequence of tokens which match that rule, or **false** (0) otherwise. The functions which return a boolean value are also named predicates. If a matching sequence of tokens is found, the function will consume (will pass over) all tokens which form that sequence and finally **crtTk** will point to the first token after these. If the function does not find any matching sequence of tokens according to its rule, **crtTk** remains unmodified.

It results that each predicate which implements a nonterminal is responsible for consuming all the tokens which form that nonterminal. If a nonterminal contains other nonterminals, it will call them and if they are matched, each one will consume its own constituent tokens and will return **true**. If a predicate is not matched, the current position in the tokens list will remain unchanged and it will return **false**.

The entire syntactic analysis will be done by calling the start nonterminal. This in turn will call the other nonterminals.

To consume terminals (tokens) the function „**consume(int code)**” is used. If at the current position it is a token with the specified code it consumes it and returns **true**, else the current position remains unchanged and it returns **false**. A possible implementation of the function **consume** is the following one:

```
Token   *consumedTk;

int     consume(int code)
{
if(crtTk->code==code){
    consumedTk=crtTk;
    crtTk=crtTk->next;
    return 1;
    }
return 0;
}
```

consume maintains a pointer to the last consumed token in the variable **consumedTk**. This will be useful for example when it will be needed to consume a constant and after that its value needs to used, that token being already consumed.

By combining the above aspects it results that:

- **the tokens (terminals)** are always consumed by using the function **consume**
- **the nonterminals** are consumed by calling the predicates which implements these nonterminals

It is possible that while testing a rule a partial advance is made inside it without being possible to reach the end. In this case there must be a way to come back to the initial position in the tokens list (the one at the beginning of the predicate), to maintain the condition that if a predicate is not matched, it will not change the current position. This can be easily accomplished by setting in the beginning of the predicate a variable with the initial value of **crtTk**: „**Token *startTk=crtTk**;”. If a comeback is needed, this will be done with „**crtTk=startTk;**”.

In the next sections some suggestions will be given about how different EBNF constructs can be implemented.

<u>SEQUENCE (a b …)</u>

It is implemented by testing one by one each of its components and by returning **true** only if all are matched.

ruleWhile ::= WHILE LPAR expr RPAR stm

```
int     ruleWhile()
{
Token   *startTk=crtTk;
if(consume(WHILE)){
    if(consume(LPAR)){
        if(expr()){
            if(consume(RPAR)){
                if(stm()){
                    return 1;
                    }else tkerr(crtTk,"missing while statement");
                }else tkerr(crtTk,"missing )");
            }else tkerr(crtTk,"invalid expression after (");
        }else tkerr(crtTk,"missing ( after while");
    }
crtTk=startTk;
return 0;
}
```

It can be seen that in order to return **true** all the sequence components must be matched. **WHILE**, **LPAR** and **RPAR** are terminals (tokens), so they are consumed by using the function **consume**. **expr** and **stm** are nonterminals and they are consumed by calling their own predicates.

In the case of the sequences very precise error messages can be generated, because it is known what must follow in sequence. For example after „while” must follow an open parenthesis, else an error is generated. The only ambiguity is the fact that if the expression which is tested by “while” is incorrect, for example „while(-/a)”, **expr** will return **false**, which in this case does not mean that the expression is missing, but that it is incorrect. It would also be possible that it is missing: „while()”, but this case is far less encountered. For this reason it was chosen an error message for the most common case. The complete message would be „missing or invalid expression after (”. In the case of the advanced implementations even this ambiguity can be eliminated by testing if the two parentheses are in consecutive positions.

In this particular implementation, if it is taken into account that **tkerr** ends the entire program (by using the function **exit** or by generating an exception), so it is no comeback in **ruleWhile**, the predicate can be written simplified in the following way:

```
int     ruleWhile()
{
    if(!consume(WHILE))return 0;
    if(!consume(LPAR))tkerr(crtTk,"missing ( after while");
    if(!expr())tkerr(crtTk,"invalid expression after (");
    if(!consume(RPAR))tkerr(crtTk,"missing )");
    if(!stm())tkerr(crtTk,"missing while statement");
    return 1;
}
```

If the **WHILE** token was not consumed, it is no need to come back to **startTk**, because **crtTk** remained unchanged (this is also true for the former version, but **startTk** was used for exemplification).

It can be seen that also in this case, as well as in the former version, no error is generated if **WHILE** is missing, but it is only returned 0. This is because an instruction can still be something else, not only „while” (for example it can be: if, return, for, …). If the absence of „while” would have generated an error, then any other instruction whose test had been after „while” would have been forbidden. Anyway, if the **WHILE** keyword is consumed so its specific sequence has begun, the next components are required as they are described in grammar, so their absence should generate errors.

From the above aspects it results that it is important to analyze when errors must be generated and when only a “return false” is needed in order to signal that the predicate did not match. The rule is the following: **If only one possibility exists and this is not matched an error will be generated. If there are multiple possibilities, the non-matching of one of them will only signal “non-matched” so the others can be also tried. An error will be generated if none of these possibilities was matched.**

ALTERNATIVE (a | b)

It is implemented by testing in turn each component and by returning **true** at the first one matched. If none is matched, in the end it is returned **false**.

factor ::= ID | CT_INT | LPAR expr RPAR

```
int     factor()
{
    Token *startTk=crtTk;
    if(consume(ID)){
        return 1;
        }
    if(consume(CT_INT)){
        return 1;
        }
    if(consume(LPAR)){
        if(expr()){
                if(consume(RPAR)){
                    return 1;
                    }else tkerr(crtTk,"missing )");
                }else tkerr(crtTk,"invalid expression after (");
        crtTk=startTk; // restore crtTk to the entry value
        }
    return 0;
}
```

This rule contains an alternative whose last part is a sequence. The alternative was implemented using a sequence of „if"s, each „if" testing one component of the alternative. If any component is matched, the predicate returns true. If none of the components are matched, in the end it returns false.

If the final sequence is not fully satisfied, it was necessary to restore **crtTk** to the predicate entry position, because it is possible to be consumed only LPAR or LPAR and expr, but not also RPAR. In these cases the initial position must be restored before returning false.

Optionality (a?)

Because in this case **a** can exist or not, it is tried to consume it but without taking into account the result of the trial.

typeName ::= typeBase arrayDecl?

```
int     typeName()
{
    if(!typeBase())return 0;
    arrayDecl();
    return 1;
}
```

It can be seen that if **typeBase** is consumed, it does not count anymore if **arrayDecl** (which is optional) exists or not. If it exists it will be consumed, but in any situation the final result will not be influenced. If further processing depends on the existence of **arrayDecl**, it can be tested with an „if".

The empty alternative (epsilon or ε) can be implemented in the same way, taking into account that **„a | ε" = „a?"**.

Optional repetition (a*)

The **a** testing is included in a loop which last while **a** can still be consumed. In simpler cases **a** can be written directly as the "while" condition (**while(a){…}**), else an infinite loop can be used which exits when **a** can be no more consumed (**while(1){ if(!a())break; }**).

stmCompound: LACC (declVar | stm)* RACC ;

```
int     stmCompound()
{
    if(!consume(LACC))return 0;
    while(1){
        if(declVar()){
            }
        else if(stm()){
            }
        else break;
        }
    if(!consume(RACC))tkerr(crtTk,"missing } or syntax error");
    return 1;
}
```

In this case an entire alternative (**declVar | stm**) may be optionally repeated. The entire alternative was put in an infinite loop, which is exited only when none of its components can be matched. It can be seen that it does not matter how many times (or never) the alternative was matched, because there is no counter for this. A variant to repeat this alternative can be „**while(declVar() || stm()){…}**".

The repetition with at least one occurrence (a+) can be implemented in the same way, taking into account that **„a+” = „a a*”**.

Observations

After the RDP algorithm was presented it can be understood why left recursion is not allowed. For example if a rule such as „**exprOr ::= exprOr OR exprAnd | exprAnd**” is implemented in its original form, this would have a beginning such as „**int exprOr(){ if(exprOr()){ …**”. In this situation, the first call from inside **exprOr** is also **exprOr** and this without consuming any token between the two calls. It results an infinite recursion, without any modification of the current position in the tokens list.

Because the predicates are calling each other, it is necessary to write them in the correct order for the C language, where a symbol must first be declared and then used. For example the predicate **declVar** which is used in **unit** will need to be written before this one. When two predicates have cross references to each other (ex: **stmCompound** and **stm**), when the header of the second (**int stm();**) can be declared before the first in order to be visible from this one.

In this phase the role of the syntactic analyzer is only to analyze the input program in order to check if it is correct from the syntactic point of view. If the program is correct, no error will be generated.

Attention: almost all the compiler future phases will add code inside the syntactic analyzer, so this one must be well understood and implemented as good as possible.

Application 4.1: *Write the predicates for the rules **exprAdd** and **exprPrimary**.*

Homework: Write the entire syntactic analyzer for the AtomC language.