CSet - Analisador sintático

Daniella Angelos [1]

[1]Departamento de Ciência da Computação - Universidade de Brasília

1. Objetivo

Este trabalho visa apresentar o analisador sintático desenvolvido com o auxílio da ferramenta *Bison*, para a linguagem CSet proposta anteriormente.

2. Introdução

Um compilador pode ser dividido em duas partes principais: análise e síntese. A parte de análise quebra o programa fonte em pedaços com significado e aplica uma estrutura gramatical a eles, informando possíveis erros que foram encontrados. Esta parte também é responsável por coletar informações sobre o programa fonte e armazená-las em uma estrutura de dados chamada *tabela de símbolos*, que é passada ao longo das fases intermediárias da análise até a parte de síntese.

A análise sintática é uma das primeiras fases de um compilador. O parser recebe tokens passados pelo analisador léxico e determina sua estrutura gramatical de acordo com a gramática da linguagem especificada. Duas estruturas de dados são populadas durante a análise sintática: a tabela de símbolos e a árvore sintática, que futuramente serão utilizadas pelo analisador semântico.

3. Gramática

A gramática da linguagem apresentada anteriormente sofreu algumas alterações que estão destacadas a seguir.

```
Outset          -> Function
                 | Outset Function

Function        -> Type identifier ( ArgList ) CompoundStmt

ArgList         -> Arg
                 | ArgList , Arg

Arg             -> Type identifier

Declaration     -> Type IdentList
                 | Type Attr

Type            -> int
                 | float
                 | char
                 | bool
                 | set << Type >>
```

```
                  | pair << Type , Type >>

IdentList      -> identifier , IdentList
                  | identifier

Stmt           -> WhileStmt
                  | Expr ;
                  | IfStmt
                  | CompoundStmt
                  | Declaration ;
                  | IO ;
                  | ReturnStmt ;

WhileStmt      -> while ( Expr ) Stmt

IfStmt         -> if ( Expr ) CompoundStmt
                  | if ( Expr ) CompoundStmt ElsePart

ElsePart       -> else Stmt

IO             -> print ( str )
                  | print ( identifier )
                  | read ( identifier )

ReturnStmt     -> return Expr
                  | return

CompoundStmt   -> { StmtList }

StmtList       -> StmtList Stmt
                  | ε

Expr           -> Attr
                  | Rvalue
                  | FuncCall

Attr           -> identifier = Expr

Rvalue         -> Rvalue Compare LogicalOr
                  | LogicalOr

Compare        -> ==
                  | <
                  | >
                  | <=
                  | >=
```

```
                  | !=

LogicalOr       -> LogicalAnd || LogicalOr
                  | LogicalAnd

LogicalAnd      -> Pertinence && LogicalAnd
                  | Pertinence

Pertinence      -> Pertinence <?> Cartesian
                  | Cartesian

Cartesian       -> Cartesian <*> Addition
                  | Addition

Addition        -> Addition + Multiplication
                  | Addition - Multiplication
                  | Multiplication

Multiplication -> Multiplication * Factor
                  | Multiplication / Factor
                  | Term

Term            -> ( Expr )
                  | - Term
                  | + Term
                  | $ Term
                  | ! Term
                  | { FactorList }
                  | ( Pair )
                  | Factor

Factor          -> identifier
                  | boolean
                  | number
                  | character

FactorList      -> FactorList , Factor
                  | Factor
                  | ε

Pair            -> Factor , Factor

FuncCall        -> identifier ( IdentList )
```

A única alteração relevante realizada foi na regra do comando *if-then-else*, para eliminar os conflitos que a forma anterior causava. (Outras alteações corresponderam à

eliminação da regra vazia pela mesma razão.)

4. Analisador sintático

Compilando e executando o analisador sintático:

- Léxico:
  ```
  $ make tokenizer
  ```
- Sintático:
  ```
  $ make parser
  $ make compile
  ```
  (Os passos anteriores podem ser realizados de uma só vez com o comando `make`)
- Executando:
  ```
  $ ./cset <input.cset>
  ```

O arquivo de entrada será analisado e serão geradas a tabela de símbolos e a árvore sintática referentes ao programa. Entretanto, estas duas estrutuuras só serão mostradas ao usuário caso nenhum erro seja encontrado.

4.1. Erros

Erros sintáticos estão relacionados à estrutura incorreta do programa. Trata-se de tokens que foram validados pelo analisador léxico mas aparecem em um local não esperado. Por exemplo, lexicamente, podemos escrever

```
int int;
```

Mas não sintaticamente, pois `int` corresponde a uma palavra reservada de tipo primitivo da linguagem, e depois de sua ocorrência é esperado um token de identificador, ou seja, esta linha não irá corresponder a nenhuma regra da gramática da linguagem, ocasionando um erro sintático.

Para que o analisador sintático possa identificar o maior número de erros possíveis, uma regra adicional foi acrescentada a algumas variáveis para que pudessem lidar com erros. Por exemplo:

```
| error '}' { synerrors++; yyerrok; }
```

Faz parte da variável `function` e toda vez que um erro é encontrado, o mesmo é reportado e o analisador ignora qualquer token que for enviado pelo léxico até encontrar `'}'`.

4.2. Exemplos

Este documento foi entregue em conjunto com 4 exemplos de programas em CSet. Dentre eles, 2 contendo códigos lexicamente e sintaticamente corretos, e 2 com erros..

- Corretos: *example(1,2).cset*
- Com erros: *error_example(1,2).cset*

Erros:

- Arquivo 1:
  - Erro sintático, linha 3: o tipo `set` só está definido para um argumento.

  - Erro léxico, linha 3: token de identificador não deve começar com dígitos.
  - Erro léxico, linha 4: aspas sem fechamento.
- Arquivo 2:
  - Erro sintático, linha 11: falta um ponto e vírgula do final do comando.

Por causa da tentativa de fazer o sintático detectar o maior número de erros possíveis, alguns "falsos positivos" podem ser gerados ao analisar um arquivo.

5. Considerações finais

A ferramenta *Bison* auxiliou em grande parte do processo de construção do analisador sintático. Algumas difilcudades foram encontradas ao lidar com os conflitos gerados pela gramática original, mas todos foram resolvidos.

Referências

[1] A. V. Abo, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers - Principles, Techniques and Tools* 2nd ed. 1986

[2] https://www.gnu.org/software/bison/manual/ [1]

[1] Consultado ao longo do desenvolvimento do analisador (2/10 - 8/10/2015).