CSet - Analisador semântico

Daniella Angelos [1]

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

1. Objetivo

Este trabalho visa apresentar o analisador semântico desenvolvido 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 semântica é a última fase da etapa de análise, utiliza a árvore sintática e a tabela de símbolos para checar a consistência semântica do código fonte, de acordo com a definição da linguagem.

3. Gramática

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

```
Outset          -> Function
                   | Declaration
                   | Outset Function

Function        -> Type identifier ( ArgList ) CompoundStmt

ArgList         -> Arglistlist
                   | ε

Arglistlist      -> Arg
                   | ArgList , Arg

Arg             -> Type identifier

Declaration     -> Type IdentList
                   | Type Attr

Type            -> int
                   | float
                   | char
```

```
                  | bool
                  | set << Type >>
                  | pair << Type , Type >>

Identlist      -> Identlistlist
                  | ε

IdentListlist  -> 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 inserção da regra `Outset -> Declaration` teve como objetivo permitir a declaração de variáveis globais.

Uma pequena alteração nas regras de lista de argumentos e *ids* também foi realizado, pois a versão antiga permitia que ambas possuísse um número arbitrário de vírgulas sem que houvesse elementos entre elas.

4. Léxico

Erros léxicos correspondem a caracteres inválidos, em CSet são, por exemplo 'ç', '@' etc, e nomes de identificadores começados por números. Para capturar este erro de identificadores, a seguinte regra foi criada.

```
{D}+(_|{L}|{D})* {return -2;}
```

Se um token não casar com nenhuma regra, será capturado pela regra a seguir.

```
. {return -1;}
```

O retorno de um número negativo ao token corresponde a um erro. Este será, então, mostrado ao usuário com informações de linha, coluna, conteúdo e tipo de erro. Neste caso, a função `tratar_erros` é chamada, ela é responsável por gerar a mensagem correspondente.

5. Sintático

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 '}'.

5.1. Árvore sintática

Para cada uma das variáveis da gramática (com uma ou outra exceção, que puderam utilizar uma definição mais genérica) foi criada uma estrutura que define o tipo do nó que é gerado após a regra ser atingida.

Cada uma dessas estruturas possuem campos referentes à variável e correspondem, em sua maioria, a outros nós na árvore.

5.2. Tabela de símbolos

Para armazenar informações sobre identificadores de variáveis e funções, uma tabela de símbolos é criada e que será de extrema utilidade ao analisador semântico.

Ao encontrar um identificador, o sintático verifica se a entrada já existe na tabela, caso não exista, a mesma é criada.

6. Semântico

Com o auxílio da tabela de símbolos e da árvore sintática, o analisador semãntico encontra inconsistências no arquivo fonte que dizem respeito, principalmente, à tipagem das expressões e à declaração de variáveis.

Os erros que este analisador captura estão listados a seguir:

- Inconsistência de tipos em uma atribuição de variável
- Variáveis usadas, mas não declaradas
- Chamada a uma função não declarada
- Chamada a uma função com o número inválido de argumentos
- Passagem de argumentos a uma função com tipos que divergem dos parâmetros da mesma
- Condição em expressões `if` e `while` com tipo diferente de `bool`

Para as verificações de tipo, foi implementado uma checagem de tipo que analisa a tipagem de cada um dos termos que compõem a expressão.

7. Geração de código

A geração de código intermediária foi iniciada, apesar de não finalizada. Está sendo utilizado o formato TAC.

Chamadas a função geram o código e um esqueleto é produzido para expressões `if` e `while`. Também já cria-se os rótulos para procedimentos.

O código que está sendo gerado é salvo no arquivo cset.tac ao final da execução do compilador.

8. Exemplos

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

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

Erros:

- Arquivo 1:
  - Erro semantico linha 4. Tipos incompativeis na atribuicao.
  - Erro semantivo linha 8. Condicao em while deve ter tipo bool.
  - Erro semantico, linha 17. Tipo de argumento nao esperado.
- Arquivo 2:
  - Erro semantico, linha 6. Chamada a funcao nao definida!
  - Erro semantico linha 7. O tipo do lado esquerdo da atribuicao nao pode ser avaliado.
  - Erro semantico. Identificador 'c' nao declarado.

9. Considerações finais

Para compilar o programa basta rodar `make` no diretório que o contém. Para executar: `./cset <input.file>`

Algumas dificuldades foram enfrentadas que, inclusive, ocuparam muito tempo do desenvolvimento. Uma das maiores foi a realização da checagem de tipo.

Referências

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