

Construção de um Analisador Léxico e de um Analisador Sintático e da Árvore Sintática Abstrata (AST) para um subconjunto da linguagem Ada

Compiladores

Grupo 7

PL1

Maria Luíza Barros Correia Baltar

Sara de Sousa Lima Soares

Este projeto implementa a etapa de análise léxica e de análise sintática de um compilador para um subconjunto da linguagem Ada. O objetivo é analisar programas escritos nessa linguagem e gerar uma Árvore Sintática Abstrata (AST) que é necessária na etapa de análise semântica (não abordada neste primeiro trabalho).

A implementação é feita em C e utiliza Flex para a análise léxica e Bison para a análise sintática, e faz a construção e a impressão da AST.

Primeira parte: Análise Léxica (usando o flex)

O analisador léxico lê o código-fonte caractere por caractere, identifica os tokens (unidades léxicas significativas) e envia-os ao parser.

Cada token representa um elemento sintático da linguagem, como palavras reservadas, identificadores, números, operadores e símbolos de pontuação.

Os tokens são classificados em grupos, conforme sua função:

Palavras-chave: PROCEDURE, BEGIN, END, IF, THEN, ELSE, WHILE, LOOP, IS

Operadores lógicos: AND, OR, XOR, NOT

Operadores aritméticos e relacionais: PLUS, MINUS, TIMES, DIVISION, MOD, REM, POWER, EQUAL_TO, DIFFERENT_THAN, EQUAL_OR_GREATER, etc.

Identificadores e literais: ID, NUM, REAL, TRUE, FALSE

Símbolos de pontuação: SEMICOLON, COLON, LPAREN, RPAREN, etc.

Para associar valores semânticos aos tokens (por exemplo, o valor numérico de um literal), utiliza-se uma união:

typedef		union		{
int	ival;	//	para	números inteiros
float	fval;	//	para	números reais
char	*text;	//	para	identificadores
int	bval;	//	para	valores booleanos (TRUE/FALSE)
}	TokenValue;			

Segunda Parte: Análise Sintática (usando o Bison)

O analisador sintático (parser) recebe do lexer a sequência de tokens e verifica se eles obedecem à gramática da linguagem. Quando uma regra gramatical é reconhecida, o parser cria um nó na AST correspondente, usando funções construtoras definidas em ast.c.

A gramática descreve uma linguagem simples baseada em Ada, com expressões, comandos e procedimentos principais.

Estrutura do programa

proc : PROCEDURE MAIN IS BEGIN stmt_lst END MAIN SEMICOLON
Comandos
<pre> stmt_lst : stmt_lst stmt stmt /* empty */ stmt : matched_stmt unmatched_stmt matched_stmt : IF expr THEN matched_stmt ELSE matched_stmt END IF SEMICOLON WHILE expr LOOP stmt_lst END LOOP SEMICOLON ID ASSIGN expr SEMICOLON unmatched_stmt : IF expr THEN stmt END IF SEMICOLON IF expr THEN matched_stmt ELSE unmatched_stmt END IF SEMICOLON </pre>
Expressões
<pre> expr : NUM REAL ID TRUE FALSE LPAREN expr RPAREN expr EQUAL_TO expr expr DIFFERENT_THAN expr expr GREATER_THAN expr </pre>

```
| expr LESS_THAN expr  
| expr EQUAL_OR_GREATER expr  
| expr EQUAL_OR_LESS expr  
| expr AND expr  
| expr OR expr  
| expr XOR expr | NOT expr  
| expr PLUS expr | expr MINUS expr | expr TIMES expr  
| expr DIVISION expr  
| expr MOD expr  
| expr REM expr  
| expr POWER expr
```

Essa gramática permite a definição de comandos compostos e expressões aninhadas, respeitando a precedência e associatividade dos operadores.

Árvore Sintática Abstrata (AST)

A AST representa a estrutura hierárquica do programa. Cada nó corresponde a uma expressão (Exp) ou a um comando (Stm).

Diferentemente dos comandos, as expressões têm um valor associado. As expressões incluem identificadores, literais e operadores. Cada expressão é identificada por um tipo (ExpType):

```
typedef enum { IDEXP, NUMEXP, FLOATEXP, BOOLEXP, OPEXP, PARENEXP }  
ExpType;
```

Funções construtoras:

```
Exp mk_numexp(int num);  
Exp mk_idexp(char *ident);  
Exp mk_boolexp(int boolean);  
Exp mk_opexp(Exp left, BinOp op, Exp right);
```

Comandos

Os comandos são categorizados por StmType:

```
typedef enum { ASSIGNSTM, COMPOUNDSTM, IFSTM, WHILESTM, PROCEDSTM }  
StmType;
```

Construtores:

```
Stm mk_assign(char *ident, Exp exp);  
Stm mk_if(Exp cond, Stm then_part, Stm else_part);  
Stm mk_while(Exp cond, Stm body);  
Stm mk_proced(Stm stmt);
```

Essa estrutura permite representar naturalmente comandos aninhados, como condicionais e laços de repetição.

A seguir, as ações semânticas no arquivo Bison chamam as funções construtoras da AST.

Por exemplo:

```
stmt : ID ASSIGNMENT expr SEMICOLON { $$ = mk_assign($1, $3); }  
      | IF expr THEN stmt_1st ELSE stmt_1st END IF SEMICOLON { $$ = mk_if($2,  
$4, $5); }  
      | WHILE expr LOOP stmt_1st END LOOP SEMICOLON { $$ = mk_while($2,  
$4); }
```

Cada regra cria o nó apropriado, e o nó raiz do programa é armazenado na variável global root.

Impressão da AST

A árvore pode ser impressa recursivamente por meio das funções print_stm() e print_exp().

Essas funções percorrem a estrutura e reproduzem o código de forma legível.

Por exemplo, o programa:

```
procedure Main is  
begin  
  x := 5;  
  if x > 0 then  
    y := x;  
  else  
    y := 0;  
  end if;  
end Main;
```

Resulta em:

```
PROCEDURE MAIN IS BEGIN  
x = 5;  
IF x > 0 THEN  
y = x;  
ELSE  
y = 0;  
END IF;  
END MAIN;
```

O arquivo main.c gerencia a leitura do arquivo-fonte, chama o parser e imprime a AST:

```
int main(int argc, char** argv) {  
  --argc; ++argv;  
  if (argc != 0) {
```

```

    yyin = fopen(*argv, "r");
    if (!yyin) {
        printf("%s': could not open file\n", *argv);
        return 1;
    }
} // yyin = stdin
if (yyparse() == 0) {
    printf("Abstract Syntax Tree:\n");
    print_stm(root);
}

fclose(yyin);
return 0;
}

```

Considerações finais

Este projeto demonstra as primeiras etapas da construção de um compilador.

- Análise Léxica: reconhecimento dos tokens e literais;
- Análise Sintática: verificação da conformidade gramatical;
- Construção da AST: representação semântica hierárquica;
- Impressão da AST: validação e visualização da estrutura.

Esta base permite a futura extensão do compilador com análise semântica, verificação de tipos e geração de código intermediário ou objeto.