

Compiladores - Projecto

Compilador para Mili-Pascal

31 Maio 2014

Grupo: Miguéis;

João Miguel Borges Subtil
Pedro Miguel Quitério Lourenço

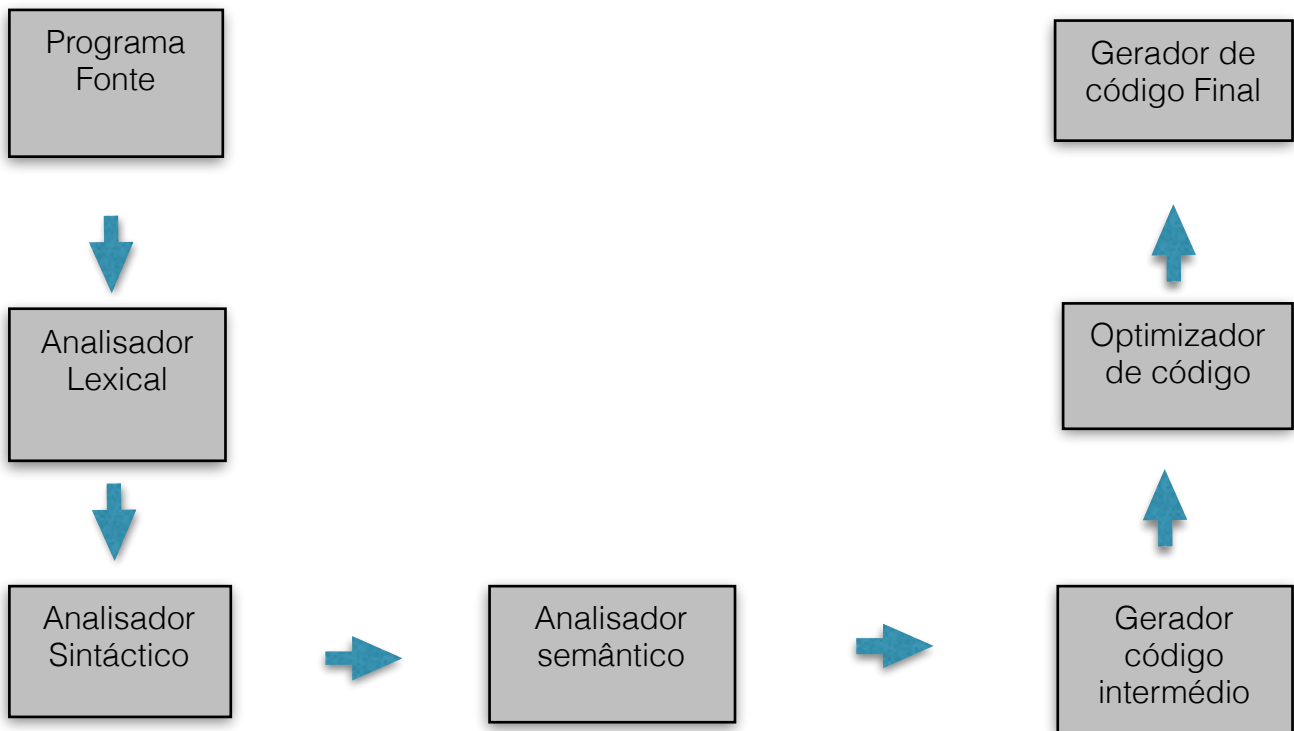
Nº 2012151975
Nº 2012143635

Compiladores - Projecto	1
1 Introdução	3
2 Análise Lexical	4
2.1 Tokens	4
2.2 Comentários	5
2.3 Tratamento de Erros	6
3 Analisador Sintático	6
3.1 Gramática	7
3.1.1 Gramática inicial	7
3.1.2 Gramática na forma BNF	8
3.2 Tratamento de erros sintáticos	10
3.3 Árvore de sintaxe Abstracta	11
3.5 Considerações Finais	12
4 Análise Semântica	12
4.1 Tabelas de Símbolos	13
4.2 Criação de tabelas	13
4.3 Tratamento de erros	14
5 Geracao de codigo	15

1 Introdução

No presente trabalho pretende-se desenvolver um compilador para a linguagem mili-pascal, um pequeno subconjunto da Linguagem Pascal. O projecto foi estruturado em 4 fases, a primeira consistia na análise lexical para qual foi usada a ferramenta *Lex*, a segunda consistia na análise sintática recorrendo á ferramenta *yacc*, a terceira consistia na construção de tabelas e detecção de erros semânticos e, por fim a última fase consistia na geração de código em LLVM.

O seguinte esquema ilustra as fases de compilação.



2 Análise Lexical

A análise lexical consiste em analisar o *input* dado e produzir uma cadeia de símbolos que podem ser manipulados pelo *parser*.

No nosso analisador os comentários são tratados com recurso a uma macro, assim sempre que o *lex* lê “(* ...*)” ou “{ ...}” ou uma mistura entre os dois o analisador ignora o que estiver dentro desse comentário.

Sempre que é detectado um carácter ou uma sequência de caracteres que não constituam um *token* é gerado um erro lexical, sendo impressa uma mensagem de erro contendo o tipo, visto que podem existir vários tipos de erros diferentes e com o número da linha e coluna onde ocorreu.

2.1 Tokens

De seguida apresentamos a lista de *tokens* aceites pela linguagem sendo que não estão representadas as palavras reservadas por questões de comodidade. Foram também definidos novos *tokens* para facilitar depois a construção da AST numa fase posterior.

- ID: Sequências alfanuméricas começadas por uma letra e seguidas por uma combinação de letras e/ou números “ [a-z][a-z0-9]* ”
- Intlit: sequências de dígitos decimais. “ [0-9]+”
- Reallit: Sequências de dígitos decimais interrompidas por um único ponto e opcionalmente seguidas de um expoente, ou sequências de dígitos decimais seguidas de um expoente. O expoente consiste na letra “e”, opcionalmente seguida de um sinal de + ou de - , seguida de uma sequência de dígitos decimais. “ {num} (“.”{num})?(e[-+]?{num})? “
- String: Sequências de caracteres (excluindo mudanças de linha) iniciadas por uma aspa simples (') e terminadas pela primeira ocorrência de uma aspa simples que não seja seguida imediatamente por outra aspa simples. “'''([^\n']|'')*'''”
- Assign: “:=“
- Begin: “begin”
- Colon: “.”
- Comma: “,”
- Do: “do”
- Dot: “.”
- Else: “else”
- End: “end”
- Forward: “forward”
- Function: “function”
- If: “if”
- Lbrac: “(“
- Not: “not”
- Output: “output”
- Paramstr: “paramstr”
- Program: “program”
- Rbrac: “)”

- Repeat: "repeat"
- Semic: " ; "
- Then: "then"
- Until: "until"
- Val: "val"
- Var: "var"
- While: "while"
- Writeln: "writeln"
- Op1: "and" | "or"
- Op2: "<" | ">" | "=" | "<=" | ">="
- Op3: "+" | "-"
- Op4: "*" | "/" | "mod" | "div"
- RESERVED: palavras reservadas e identificadores requeridos do Pascal standard não usados.

Foram criados novos tokens com base na separação dos tokens OP1, OP2 e OP4 para resolver problemas de ambiguidade, as transformações são apresentadas em baixo.

- A op1 foi dividida em AND e OR
- Em op2 "<" passou a NEQ, "<=" passou a LEQ, ">=" passou a GEQ
- Na op4 "mod" e "div" passaram a MOD e DIV respectivamente
- BEGIN passou a BEG

2.2 Comentários

Para identificar correctamente os comentários recorreremos à macro TESTECOMENT, quando é detectado o início de um comentário tudo é ignorado até o mesmo ser terminado correctamente com "*" ou "}", esta macro também trata os casos de comentário multi-linha.

Caso seja detectado um <<EOF>> sem o comentário ter sido terminado correctamente, é exibida uma mensagem de erro com a linha e a coluna em que começa o dito comentário.

2.3 Tratamento de Erros

Tal como já foi referido anteriormente sempre que o analisador detecta um erro (quer seja por um comentário/string não terminado(a)) é impressa uma mensagem de erro que indica a linha e a coluna em que o dito erro ocorreu.

Quando é detectado um carácter ilegal é impressa a mensagem de “ilegal carácter” com a respectiva linha e coluna, o programa continua até chegar ao fim.

Após todo o conteúdo ter sido analisado o programa termina e imprime as mensagens correspondentes aos erros detectados, se existirem.

3 Analisador Sintático

O analisador sintático foi implementado com recurso à ferramenta *yacc*, a ideia é o *lex* reconhecer os tokens e posteriormente enviá-los ao *yacc* que irá verificar se estes pertencem à linguagem.

Sempre que o *lex* identifica um token envia-o ao *yacc*, contudo este token pode ser recebido como *yytext* numa estrutura *union* criada especialmente para facilitar a manipulação e armazenamento de dados , ou pode ser enviado como *token* que também está definido no início do ficheiro *mpaparser.y*.

```
%union{
    char* str;
    int u_line,u_col;
    struct _node * node;
}
```

Esta estrutura contém um campo(char*) para armazenar Id's, Intlit's, Reallits e Strings, contém também um campo para a linha e a coluna (embora não estejam a ser usados) e por fim uma estrutura (node) que depois irá ser utilizada como estrutura genérica para a construção da AST.

Foi necessário também definir algumas variáveis externas como o *yytext*, linha e coluna para partilhar valores entre ambos os programas.

```
extern int line;
extern unsigned long col;
extern char* yytext;
```

3.1 Gramática

A gramática é a maneira formal de especificar a sintaxe de uma linguagem. Para o projecto foi nos fornecida a gramática na forma EBNF, contudo esta teve de ser transformada para BNF para remover as ambiguidades.

3.1.1 Gramática inicial

```
Prog → ProgHeading SEMIC ProgBlock DOT
ProgHeading → PROGRAM ID LBRAC OUTPUT RBRAC
ProgBlock → VarPart FuncPart StatPart
VarPart → [ VAR VarDeclaration SEMIC { VarDeclaration SEMIC } ] VarDeclaration
          → IDList COLON ID
IDList → ID { COMMA ID }
FuncPart → { FuncDeclaration SEMIC }
FuncDeclaration → FuncHeading SEMIC FORWARD
FuncDeclaration → FuncIdent SEMIC FuncBlock
FuncDeclaration → FuncHeading SEMIC FuncBlock
FuncHeading → FUNCTION ID [ FormalParamList ] COLON ID
FuncIdent → FUNCTION ID
```

```

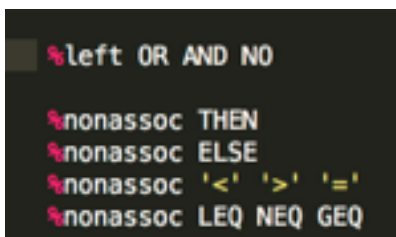
FormalParamList → LBRAC FormalParams { SEMIC FormalParams } RBRAC
FormalParams → [ VAR ] IDList COLON ID
FuncBlock → VarPart StatPart
StatPart → CompStat
CompStat → BEGIN StatList END
StatList → Stat { SEMIC Stat }
Stat → CompStat
Stat → IF Expr THEN Stat [ ELSE Stat ]
Stat → WHILE Expr DO Stat
Stat → REPEAT StatList UNTIL Expr
Stat → VAL LBRAC PARAMSTR LBRAC Expr RBRAC COMMA ID RBRAC
Stat → [ ID ASSIGN Expr ]
Stat → WRITELN [ WritelnPList ]
WritelnPList → LBRAC ( Expr | STRING ) { COMMA ( Expr | STRING ) } RBRAC
Expr → Expr (OP1 | OP2 | OP3 | OP4) Expr
Expr → (OP3 | NOT) Expr
Expr → LBRAC Expr RBRAC
Expr → INTLIT | REALLIT
Expr → ID [ ParamList ]
ParamList → LBRAC Expr {COMMA Expr} RBRAC

```

3.1.2 Gramática na forma BNF

Para remover as ambiguidades tivemos de efectuar diversas alterações, algumas delas foram:

- Criação de estados adicionais para regras que implicam repetição de tokens
- Estabelecimento de prioridades entre operadores
- Definição de regras de associação



A gramática resultante das alterações acima descritas é a seguinte:

```

Program           : ProgHeading ';' ProgBlock '.'
ProgHeading       : PROGRAM ID '(' OUTPUT ')'
ProgBlock         : VarPart FuncPart StatPart
VarPart           : VAR VarDeclaration ';' VarPart2
                  |
VarPart2          : VarDeclaration ';' VarPart2
                  |
VarDeclaration    : IDList ':' ID
IDList            : ID IDList2
IDList2           : ',' ID IDList2
                  |
FuncPart          : FuncDeclaration ';' FuncPart
                  |
FuncPart          : FuncDeclaration ';' FuncPart
                  | FuncIdent ';' FuncBlock
                  | FuncHeading ';' FuncBlock
FuncHeading       : FUNCTION ID ':' ID
                  | FUNCTION ID FormalParamList ':' ID
FuncIdent         : FUNCTION ID
FormalParamList   : '(' FormalParams FormalParamListaux ')'
FormalParamListaux : ';' FormalParams FormalParamListaux
FormalParams      : IDList ':' ID
                  | VAR IDList ':' ID
FuncBlock         : VarPart StatPart
StatPart          : CompStat
CompStat          : BEG StatList END
StatList          : Stat Stat2
Stat2             : ';' Stat Stat2
                  |
Stat              : CompStat
                  | IF Expr THEN Stat
                  | IF Expr THEN Stat ELSE Stat
                  | WHILE Expr DO Stat
                  | REPEAT StatList UNTIL Expr
                  | VAL '(' PARAMSTR '(' Expr ')' ',' ID ')'
                  | optStat2
                  | WRITELN
                  | WRITELN WritelnPList
optStat2          : ID ASSIGN Expr
                  |
WritelnPList      : '(' optWritelnPList WritelnPList2 ')'
optWritelnPList   : Expr
                  | STRING
WritelnPList2     : ',' optWritelnPList WritelnPList2
                  |
Expr              : SimpleExpression
                  | SimpleExpression '=' SimpleExpression
                  | SimpleExpression '>' SimpleExpression

```

	SimpleExpression '<' SimpleExpression
	SimpleExpression LEQ SimpleExpression
	SimpleExpression NEQ SimpleExpression
	SimpleExpression GEQ SimpleExpression
SimpleExpression	: '+' Term
	'-' Term
	Term
	SimpleExpression '+' Term
	SimpleExpression '-' Term
	SimpleExpression OR Term
Term	: Term AND Factor
	Term MOD Factor
	Term DIV Factor
	Term '*' Factor
	Term '/' Factor
	Factor
Factor	: NOT Factor
	'(' Expr ')'
	INTLIT
	REALLIT
	ID ParamList
	ID
ParamList	: '(' Expr repParamList ')'
repParamList	: ',' Expr repParamList

3.2 Tratamento de erros sintáticos

O tratamento de erros sintáticos é feito recorrendo á função *yyerror* que imprime a linha, coluna e o erro detectado, quando um erro é detectado o programa termina.

```
void yyerror (char *s) {
    printf ("Line %d, col %d: %s: %s\n", line, (int)(col)-(int)strlen(yytext), s, yytext);
}
```

3.3 Árvore de sintaxe Abstracta

Para a estrutura da árvore decidimos criar um nó genérico que contém a informação necessária para tornar ao seu acesso mais simples tanto na sua criação como em acessos futuros.

```
typedef struct _node {  
    char *id;  
    struct _node *brother;  
    struct _node *son;  
    char * type;  
    char * value;  
    int line;  
    int col;  
}Node;
```

O campo id é o nome do respectivo nó, brother e son são ponteiros para irmão e filho respectivamente, existe ainda linha e coluna para facilidades de acesso a imprimir erros na fase de análise semântica.

Os campos type e value são utilizados para guardar Id's , Intlit's , Reallit's e Strings, escolhemos criar a estrutura desta maneira para ser mais fácil fazer comparações quando necessário, assim em vez de ler "IntLit(%s)" no id e fazer várias operações sobre essas strings, pode-se utilizar um simples *strcmp()* sobre o campo type ou value.

```

//create new node a return it
Node * make_node(char *name,char *type,char *value,Node *son,Node *brother, int line, int col){
}

//add a node as a brother
void addBrother(Node * temp, Node * brother) {
}

//add a child to a node
void addChild(Node * temp, Node * child) {
    temp->son = child;
}

//print ast in postfix notation
void printAll(Node * node,int level) {
}

//check statlist sons, if its null create one, if it has no brothers return, else create new node with
Node * check_statlist(Node * temp, int line, int col) {
}

//if it has no brothers or it's null return it , else create new node with "temp" as a brother
Node * check_statlist2(Node * temp, int line, int col) {
}

```

As funções anteriores são responsáveis pela criação da árvore de sintaxe abstracta, a função `make_node` cria o nó e devolve-o para ser ligado a outros nós, a função `addBrother` insere um nó como irmão de outro, o `addChild` adiciona um nó como filho, a função `printAll` imprime tudo recursivamente e as duas funções `checkstatlist` verificam como se deve adicionar um novo nó "Statlist".

3.5 Considerações Finais

Nesta meta (meta2) o nosso grupo só conseguiu obter 540 pontos , apenas após alterações na pós-meta 2 conseguimos obter a totalidade dos pontos (600), isto devia-se ao facto de estarmos a perder alguns nós `statlist`.

4 Análise Semântica

Nesta meta a principal preocupação recai sobre as operações a realizar no programa verificando a compatibilidade dos tipos de dados envolvidos, para tal foi necessário implementar Tabelas de símbolos que armazenam e permitem a consulta *á posteriori* das variáveis. Utilizando as tabelas podemos consultar os tipos de dados envolvidos nas diversas operações e detectar situações de incompatibilidade entre tipos nos programas

4.1 Tabelas de Símbolos

Na tabela de símbolos são armazenados as variáveis de vários tipos e a sua localização na estrutura do programa, as tabelas são criadas ao percorrer a Árvore de Sintaxe Abstracta onde são percorridos todos os nós que contêm declarações de métodos e atributos.

```
//dados especificos da tabela
typedef struct _tablestructure{
    Table_structure * next;//proxima tabela
    char *table_name;
    Table_lines * data;//linhas da tabela
}_TableStructure;

//campo generico das tabelas, corresponde a uma linha
typedef struct _tablelines{
    char * name;
    char * type;
    char * flag;
    char * value;
    Table_lines *next; // proxima linha da tabela
}_TableLines;
```

A `_tablestructure` consiste num “header genérico” que contém um ponteiro para a próxima tabela e para um outra estrutura `_tablelines` que representa cada linha da tabela, a `_tablelines` também tem um ponteiro para a próxima linha

4.2 Criação de tabelas

De seguida são apresentadas as funções responsáveis pela criação e manipulação de tabelas, no caso da `create_tables`, esta também insere os dados iniciais da outer table e da function symbol table.

```

//creates first 2 tables and the program table , returns outer table
Table_structure * create_tables(Node * node){
}

//cria estrutura base e devolve ponteiro
Table_lines* generic_lines_table(void){
}

//insert a new line in _tablelines
void insert_data(Table_lines *line,char *name, char *type, char *flag, char *value){
}

//cria 2 tabela generica a symbol table
void create_base_structure_table(Table_structure * temp){
}

//cria nova tabela "mãe" e adiciona no fim, retorna ponteiro para a ultima
Table_structure* create_generic_table(Table_structure *temp, char* nome){
}

// Adds a new line after the last one, Sets new line contents to be equal to the arguments
Table_lines * first_line(char *name, char *type, char *flag, char *value){
}

```

4.3 Tratamento de erros

O tratamento de erros foi alterado na pós meta, sendo que na meta normal nenhum erro foi tratado, já na pós meta foi adicionado o tratamento de erros de símbolos duplicados, símbolos não definidos e type identifier expected.

Embora o tratamento desses erros não tenha atingido a pontuação total , passamos de 205 pontos na meta 3 para 236, a tab.

```

void error_symbolalreadydefined(char * symbol,int line, int col){
}
int error_symbolnotdefined(char * symbol,int line, int col){
}
//Type identifier expected
int error_typeidentifierexpected(int line, int col){
}

```

5 Geracao de codigo

Para a meta 4 não tivemos qualquer pontuação, tivemos uma submissão mas não conseguimos obtivemos