

Compiladores

01/06/2015

Compilador para a linguagem mili-Pascal (mPa)

Mooshak Login: Migueis

Quitério,	PEDRO	pmlour@student.dei.uc.pt	2012143635	LEI
Subtil,	JOÃO	jsubtil@student.dei.uc.pt	2012151975	LEI

Conteúdo

1	Introdução	2
2	Análise Lexical	3
2.1	Tokens	3
2.2	Comentários	6
2.3	Tratamento de Erros	6
2.4	Linhas e colunas	7
3	Análise Sintáctica	8
3.1	Gramática	9
3.1.1	Gramática inicial	9
3.1.2	Gramática na forma BNF	10
3.2	Tratamento de erros sintácticos	13
3.3	Árvore de Sintaxe Abstracta - AST	13
3.4	Considerações Finais - Pós-Meta 2	15
4	Análise semântica	16
4.1	Tabelas de Símbolos	16
4.2	Criação de Tabelas	17
4.3	Tratamento de erros	18
5	Geração de código	19

1 Introdução

No presente trabalho pretende-se desenvolver um compilador para a linguagem mili-Pascal, um pequeno subconjunto da Linguagem Pascal. Devido ao facto de se tratar de um subconjunto de Pascal, qualquer linguagem aceite por mili-Pascal será também aceite pela linguagem Pascal, podendo, no entanto, não se verificar o contrário.

O projecto foi estruturado em 4 fases:

1. Análise lexical, na qual se recorreu à ferramenta **Lex**;
2. Análise sintática, tendo-se recorrido à ferramenta **yacc**;
3. Construção de tabelas de símbolos e detecção de erros semânticos;
4. Geração de código em LLVM;

O seguinte esquema ilustra as fases de compilação:

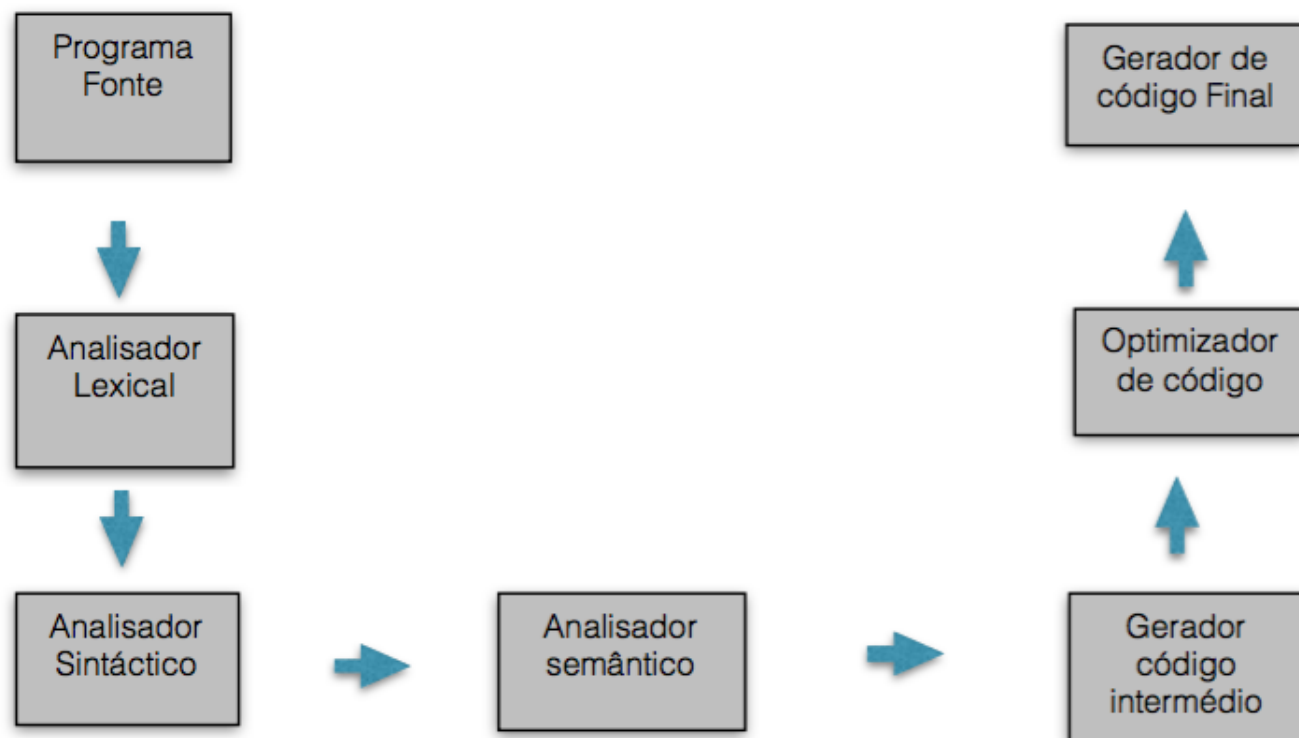


Fig.1 - Fases de Compilação;

2 Análise Lexical

A análise lexical consiste em analisar o *input* introduzido e, a partir do mesmo, produzir uma cadeia de símbolos (*tokens*) que possam 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 desses caracteres.

Sempre que é detectado um caracter ou uma sequência de caracteres que não constituam um token, é gerado um erro lexical, sendo impressa uma mensagem de erro contendo o tipo, linha e coluna de ocorrência (Esta mensagem é diferente, para se distinguir dos restantes tipos de erros que possam existir).

2.1 Tokens

De seguida, apresentamos a lista de tokens aceites pela linguagem. Foram também definidos novos tokens para facilitar 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 (') ou 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”;
- 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 encontram-se 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 `TESTCOMMENT`.

Quando é detectado o início de um comentário tudo é ignorado até ao carácter de terminação ser detectado.

Através desta macro, são também tratados os casos de comentários multi-linha.

Dentro de um comentário, todos os caracteres que não servem para fechar o comentário são tratados da mesma forma (apenas é incrementada a coluna), com excepção do carácter de mudança de linha (que incrementa a contagem de linhas e reinicia a contagem de colunas). Caso seja detectado um «EOF» sem o comentário ter sido terminado correctamente, é exibida uma mensagem de erro com a linha e coluna em que começa o mesmo.

2.3 Tratamento de Erros

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

Quando é detectado um carácter ilegal (um carácter que não corresponda aos tokens esperados) é impressa a mensagem *“illegal character”* com a respectiva linha e coluna, continuando o programa até ao fim.

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

2.4 Linhas e colunas

Apesar de a contagem de linhas e colunas estar funcional para a primeira e segunda metas, a mesma teve que ser alterada para ficar totalmente funcional para a meta 3, onde esses valores são necessários para a correcta apresentação dos erros semânticos.

Deste modo, alterámos o uso de variáveis do tipo *extern* para o uso de uma **macro** interna do **lex** que permitiu a passagem correcta dos valores o programa de detecção de erros.

```
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = line; yylloc.first_column = col;  
                        yylloc.last_column = col+yyleng-1; col += yyleng;
```

Fig.2 - Macro para contagem de linhas e colunas;

3 Análise Sintáctica

O analisador sintáctico foi implementado com recurso à ferramenta **yacc**.

A ideia passa por o **lex** reconhecer os tokens e posteriormente enviá-los ao **yacc** que irá verificar se a sua organização está de acordo com a estrutura sintática da linguagem.

O **lex** executa a função *yylex()* que trata e envia os tokens para o **yacc** que corre a partir da função *yyarse()*, produzindo um conjunto de saída (neste caso, uma **AST**).

Sempre que o **lex** identifica um token, envia-o ao **yacc**. Contudo, este token pode ser recebido como *yytext* numa estrutura union criada especial 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;
}
```

Fig.3 - Estrutura Union;

Esta estrutura contém um campo (*char**) para armazenar *Id's*, *Intlit's*, *Reallit's* e *String's*, um campo para a linha e coluna (embora não estejam a ser usados) e por fim uma estrutura (*_node*) que depois irá ser utilizada 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;
```

Fig.4 - Variáveis Externas;

3.1 Gramática

A gramática é a maneira formal de especificar a sintaxe de uma linguagem. Para o projecto foi fornecida a gramática na forma **EBNF** (*Extended Backus-Naur Form*), tendo esta que ser transformada em **BNF** (*Backus-Naur Form*) para remover as ambiguidades.

3.1.1 Gramática inicial

Prog \rightarrow ProgHeading SEMIC ProgBlock DOT

ProgHeading \rightarrow PROGRAM ID LBRAC OUTPUT RBRAC

ProgBlock \rightarrow VarPart FuncPart StatPart

VarPart \rightarrow [VAR VarDeclaration SEMIC { VarDeclaration SEMIC }]

VarDeclaration \rightarrow IDList COLON ID

IDList \rightarrow ID { COMMA ID }

FuncPart \rightarrow { FuncDeclaration SEMIC }

FuncDeclaration \rightarrow FuncHeading SEMIC FORWARD

FuncDeclaration \rightarrow FuncIdent SEMIC FuncBlock

FuncDeclaration \rightarrow FuncHeading SEMIC FuncBlock

FuncHeading \rightarrow FUNCTION ID [FormalParamList] COLON ID

FuncIdent \rightarrow FUNCTION ID

FormalParamList \rightarrow LBRAC FormalParams { SEMIC FormalParams } RBRAC

FormalParams \rightarrow [VAR] IDList COLON ID

FuncBlock \rightarrow VarPart StatPart

StatPart \rightarrow CompStat

CompStat \rightarrow BEGIN StatList END

StatList \rightarrow Stat { SEMIC Stat }

Stat \rightarrow CompStat

Stat \rightarrow IF Expr THEN Stat [ELSE Stat]

Stat \rightarrow WHILE Expr DO Stat

Stat \rightarrow REPEAT StatList UNTIL Expr

Stat \rightarrow VAL LBRAC PARAMSTR LBRAC Expr RBRAC COMMA ID RBRAC

Stat \rightarrow [ID ASSIGN Expr]

Stat \rightarrow WRITELN [WritelnPList]

WritelnPList \rightarrow LBRAC (Expr | STRING) { COMMA (Expr | STRING) } RBRAC

Expr \rightarrow Expr (OP1 | OP2 | OP3 | OP4) Expr

Expr \rightarrow (OP3 | NOT) Expr

Expr \rightarrow LBRAC Expr RBRAC

Expr \rightarrow INTLIT | REALLIT

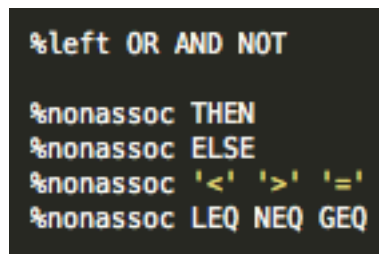
Expr \rightarrow ID [ParamList]

ParamList \rightarrow LBRAC Expr { COMMA Expr } RBRAC

3.1.2 Gramática na forma BNF

Para remover as ambiguidades tivemos de efectuar diversas alterações, tais como:

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



```
%left OR AND NOT
%nonassoc THEN
%nonassoc ELSE
%nonassoc '<' '>' '='
%nonassoc LEQ NEQ GEQ
```

Fig.4 - Precedência de Operadores;

A gramática resultante das operações 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
FuncDeclaration : FuncHeading ';' FORWARD
                | 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

```

Fig.6 - Gramática BNF - parte 1;

```

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
                   |

```

Fig.7 - Gramática BNF - parte 2;

3.2 Tratamento de erros sintácticos

O tratamento de erros sintácticos é feito recorrendo à função `yyerror()` que imprime a linha, coluna e erro detectado.

```
void yyerror (char *s) {  
    printf ("Line %d, col %d: %s: %s\n", line, (int)(col)-(int)strlen(yytext), s, yytext);  
    exit(0);  
}
```

Fig.8 - Função `yyerror`;

3.3 Árvore de Sintaxe Abstracta - AST

Para a estrutura da árvore decidimos criar um nó genérico que contém a informação necessária para tornar o seu acesso mais simples tanto na sua criação como em acessos futuros (nomeadamente, durante a criação das tabelas de símbolos e tratamento de erros).

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

Fig.9 - Estrutura do nó genérico;

O campo *id* é o nome do respectivo nó com o seu valor entre parênteses e *brother* e *son* são ponteiros para o irmão e o filho respectivamente. Existe ainda *line* e *col* para facilitar a impressão de erros na fase de análise semântica.

Os campos *type* e *value* são utilizados para guardar Id's, Intlit's, Reallit's e String's.

Escolhemos criar a estrutura desta forma por ser mais fácil fazer comparações quando necessário. Assim, em vez de ler “Intlit(%)” no id e fazer várias operações sobre essas strings, podemos utilizar directamente um simples `strcmp()` sobre os campos *type* ou *value*.

As funções seguintes são responsáveis pela criação da árvore de sintaxe abstracta:

- *make_node* - cria o nó e devolve-o para ser ligado a outros nós;
- *addBrother* - insere um nó como irmão do outro;
- *addChild* - adiciona um nó como filho;
- *printAll* - é uma função recursiva que percorre todos os nós e os imprime;
- *checkstatlist* - Duas funções que verificam como se deve adicionar um novo nó “**Statlist**”, impedindo a criação de nós supérfluos;

3.4 Considerações Finais - Pós-Meta 2

Nesta meta (meta 2), o nosso grupo apenas conseguiu obter 540 pontos.

No entanto, após umas pequenas alterações na fase de pós-meta chegámos aos 600 pontos. Para isto, tivemos que alterar a criação dos nós **statlist**, visto que estávamos a perder os ponteiros para alguns destes nós.

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 à *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

Nas tabelas de símbolos são armazenadas as variáveis de vários tipos e a sua localização na estrutura do programa.

As tabelas são criada ao percorrer a **AST** onde são encontrados todos os nós que contêm declarações de funções e variáveis.

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

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

Fig.10 - Estruturas das tabelas;

A *TableStructure* consiste num “*header genérico*” que contém um ponteiro para a próxima tabela e um para outra estrutura *TableLines* que representa cada linha da tabela.

A estrutura *TableLines* também tem um ponteiro para o 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){ ...
}
```

Fig.11 - Funções de criação das tabelas;

4.3 Tratamento de erros

O tratamento de erros foi alterado na pós-meta, sendo que na meta normal quase nenhum erro foi tratado.

Na fase de pós-meta foram adicionados os tratamentos de erros para símbolos duplicados, símbolos não definidos e sítios em que se esperava o identificador de tipo de variável.

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

```
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){ ... }
```

Fig.12 - Funções de verificação de erros;

5 Geração de código

Na meta 4, não conseguimos obter qualquer ponto apesar do código produzido e várias tentativas.

Conseguimos ainda assim colocar o programa a imprimir a declaração do *printf* e a declaração das *strings* necessárias para o *writeln*.

Para além da impressão das *strings*, conseguimos ainda imprimir as variáveis dentro dos nós *varDecl*.