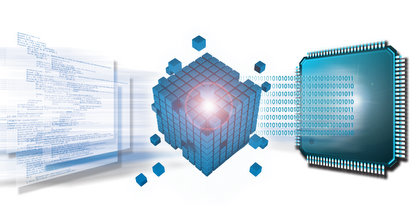
 

Compiladores

2º Semestre



**Relatório do Projecto**

Trabalho realizado por:

* Daniel Coelho Bastos 2009109605
* Tiago Martins Andrade 2010165760

2014/2015

# Índice

[Índice 1](#_Toc421041168)

[Introdução 2](#_Toc421041169)

[Fase 1 – Análise Lexical 3](#_Toc421041170)

[Fase 2 – Análise Sintática 5](#_Toc421041171)

[Fase 3 – Análise Semântica 8](#_Toc421041172)

[Fase 3.1 - Tabela de Símbolos 8](#_Toc421041173)

[Fase 3.2 - Detecção de Erros 9](#_Toc421041174)

[Fase 4 – Geração de Código 10](#_Toc421041175)

[Conclusão 11](#_Toc421041176)

# Introdução

Este projeto consiste no desenvolvimento de um compilador para a linguagem “mili-Pascal,” que é um pequeno subconjunto da linguagem Pascal Standard ISO 7185:1990 com extensões relativas à passagem de parâmetros através da linha de comandos.

O compilador desenvolvido neste trabalho foi implementado em C com recurso às ferramentas lex, yacc e llvm.

Este projeto está dividido nas seguintes fases:

1. Análise Lexical (verifica-se se os tokens introduzidos são válidos)
2. Análise Sintática (verifica-se se os tokens seguem a gramática definida e constrói a árvore de sintaxe abstrata)
3. Análise Semântica (constrói-se a tabela de símbolos e detecta-se a existência de erros semânticos)
4. Geração de código (cria-se um ficheiro em código llvm que implementa as mesmas funcionalidades do programa de entrada)

# Fase 1 – Análise Lexical

O analisador lexical foi implementado em C utilizando a ferramenta lex. Os tokens da linguagem são apresentados de seguida.

**ID**: sequências alfanuméricas começadas por uma letra.

**INTLIT**: sequências de dígitos decimais.

**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.

**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. Por exemplo, “'abc'” e “'texto entre ''aspas'''”.

**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. NOTA: os identificadores requeridos boolean, false, integer, real e true são usados em fases posteriores do projeto, e por isso não são RESERVED.

O ficheiro de especificação do lex, tal como yacc, respeita o seguinte formato:

...definições...

%%

...regras...

%%

...subrotinas...

No caso de o interpretador detectar um caractere inválido é feita uma contagem de linhas e de colunas para o mostrar não só o caractere inválido mas também a linha e a coluna onde se encontra o erro. Também é mantida em memória a linha e coluna do início do token actual. Isto permite no caso de uma expressão não passar a análise lexical mostrar não só o tipo de erro mas também a linha e a coluna onde ocorre.

# Fase 2 – Análise Sintática

O analisador sintático foi implementado através do interpretador lexical descrito anteriormente e com recurso à ferramenta yacc. A seguinte gramática em notação EBNF define a sintaxe utilizada para o desenvolvimento deste compilador da linguagem miliPascal.

**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

Uma vez que a gramática dada é ambígua e é apresentada em notação EBNF, onde […] representa “opcional” e {…} representa “zero ou mais repetições,” esta deverá ser modificada para permitir a análise sintática ascendente com o yacc. Será necessário ter em conta a precedência e as regras de associação dos operadores, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens miliPascal e Pascal.

A seguinte tabela define a ordem de precedência bem como a ordem de associação, isto é se associa à direita ou à esquerda, utilizada neste projecto.

|  |  |
| --- | --- |
| **Operador** | **Associatividade** |
| Assign (:=) | Direita |
| Greater (>), Less (<), Gequal (>=), Lequal (<=), Equals (=), Different (<>) | Esquerda |
| Plus (+), Minus (-), Or (or) | Esquerda |
| Mult (\*), Realdiv (/), Div (div), Mod (mod), And (and) | Esquerda |
| Not (not) | Direita |
| Rbrac ((), Lbrac ()) | Esquerda |
| Then (then), Else (else) | Direita |

Representação da estrutura da Árvore de Sintaxe Abstrata (AST):

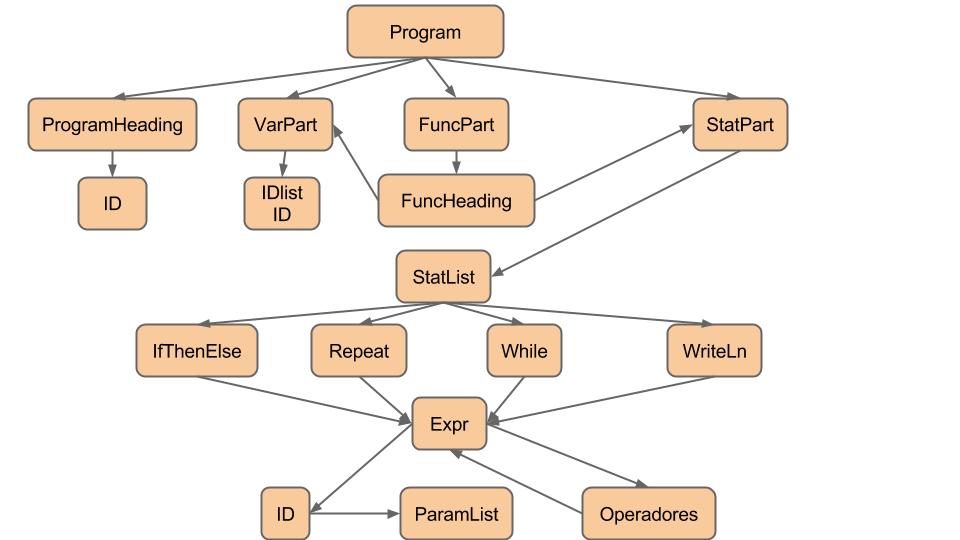


Figura 1 - Árvore de Sintaxe Abstrata

Inicialmente implementámos uma estrutura específica para cada regra e uma abordagem Top-Down da Árvore de Sintaxe Abstrata (AST). Verificou-se que em determinados casos de teste existia um número elevado de Nós desnecessários a serem criados e a corromperem a AST. Decidimos então mudar de abordagem.

Para implementar esta gramática criámos uma estrutura genérica (figura 1). A partir desta estrutura são criados Nós para guardar a informação necessária. Com os Nós preenchidos podemos então construir a Árvore de Sintaxe Abstrata (AST).

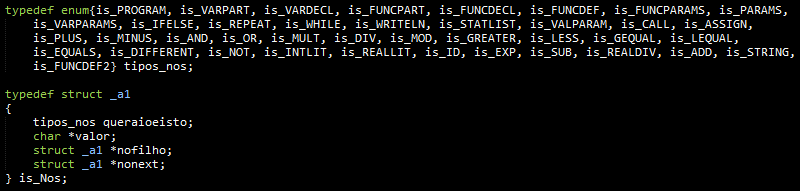


Figura 2 - Estrutura is\_Nos (ficheiro structures.h)

A Árvore de Sintaxe Abstrata ficou então organizada em Nós pais, filhos e irmãos. Assim, cada Nó tem um tipo, um valor, um ponteiro para o seu primeiro filho (\*nofilho) e para o seu primeiro irmão (\*nonext) e assim em diante, formando listas ligadas de nós.

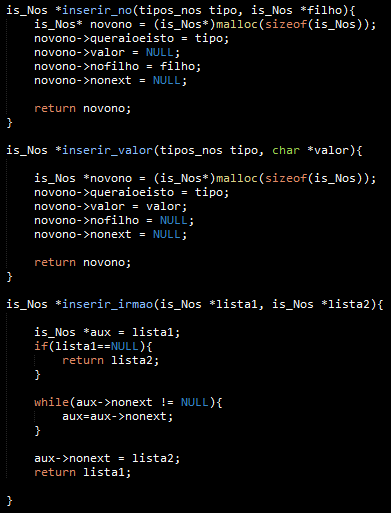


Figura 3 - Funcões de criação da AST (functions.c)

A impressão da árvore (activada com o comando opcional -t) é feita percorrendo a árvore uma única vez através de uma procura em Profundidade Primeiro (depth-first search). Para cada elemento da árvore, a informação relativa a si é impressa, seguida da informação relativa aos seus descendentes. A função *espaçamento* insere a indentação correcta na impressão da AST.

# Fase 3 – Análise Semântica

A análise semântica da linguagem miliPascal foi implementada em linguagem C utilizando as ferramentas lex e yacc e tendo por base o analisador sintático desenvolvido na fase anterior.

## Fase 3.1 - Tabela de Símbolos

Para que seja possível uma análise semântica de um programa foi necessária a construção de uma tabela de símbolos para cada região (programa ou função) do programa de entrada, incluindo uma tabela exterior contendo os identificadores requeridos boolean, integer, real, false e true, o identificador da função pré-definida paramcount (que implementa o acesso ao número de parâmetros passados na linha de comandos) e uma referência ao próprio programa.

Por sua vez, a tabela correspondente ao programa irá conter os identificadores das variáveis e funções declaradas e definidas, respectivamente. Finalmente, as tabelas correspondentes às funções contêm o próprio identificador da função (enquanto valor de retorno) e os identificadores dos respectivos parâmetros formais e variáveis locais.

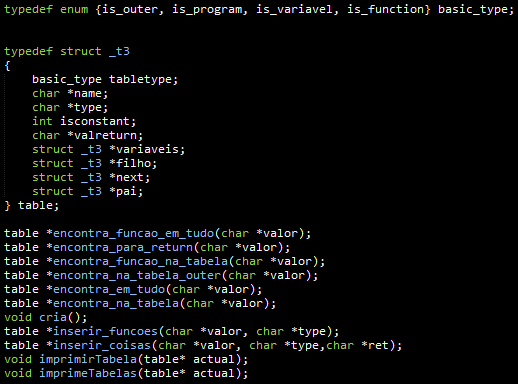


Figura 2 - Estrutura e Funções da Tabela de Símbolos

A tabela de símbolos é constituída por uma lista ligada de variáveis globais e de funções em que cada função tem uma lista ligada das suas variáveis locais bem como dos seus parâmetros.

A estrutura *table* contém o tipo da tabela (tabletype), o seu nome (name), o tipo de valor que retorna (type), o valor que retorna (valreturn), se o valor é constante ou não (isconstant).

A função *cria* cria a tabela outer, o paramcount e a tabela program.

A função *encontra\_na\_tabela\_outer* procura na tabela outer uma dada variável e devolve o Nó respectivo caso a encontre.

As funções *encontra\_funcao\_em\_tudo* e *encontra\_em\_tudo* procuram em todas as tabelas a função ou a variável, respectivamente, e devolvem o Nó caso a encontrem.

As funções *encontra\_funcao\_na\_tabela* e *encontra\_na\_tabela* procuram em determinada tabela (que é dada pela variável global symtab) a função ou a variável, respectivamente, e devolvem o Nó caso a encontrem.

As funções *insere\_funcoes* e *insere\_coisas* inserem uma dada função ou variável na tabela dada pela variável symtab, respectivamente.

As funções *imprimirTabela* e *imprimeTabelas* estão interligadas para poderem imprimir todas as tabelas.

## Fase 3.2 - Detecção de Erros

A detecção de erros semânticos é efectuada em duas fases:

* Detecção de símbolos duplicados
* Detecção de utilização de símbolos não existentes ou incompatibilidade de tipos nas operações

Na primeira fase é feita uma pesquisa linear nas tabelas e verifica-se se nenhuma das variáveis já foi definida no scope em utilização.

Na segunda fase cada statement é verificado para confirmar se há concordância entre os vários tipos a ser usados. Verifica-se também se as funções têm o número correcto de argumentos.

Assim que o primeiro erro é detectado, a mensagem de erro associada é emitida e a variável global *erros* é alterada para 1. Esta variável é usada para impedir que outros erros sejam emitidos ou que o código seja gerado havendo erros, pois o compilador verifica sempre se a variável não está a 1 antes de imprimir texto.

PÓS META! Call às funções alínea G! faltam 38 pontos!

# Fase 4 – Geração de Código

A geração de código resulta na transformação de input recebido num output que implementa as mesmas funcionalidades do programa de entrada representado na linguagem intermédia LLVM.

Os tipos de dados que estamos a usar para as variáveis são:

* i32 para valores inteiros
* double para valores reais
* i1 para valores boolean

A memória para guardar o valor das variáveis é alocada no momento da sua declaração. Quando se pretende guardar valores nessas variáveis é feito um *store* para guardar o valor.

Variáveis Globais

Variáveis Locais

Assign de valores a variáveis

Expressões de soma, subtracção, divisão, multiplicação,

# Conclusão