Compilador para a linguagem mili-Pascal (mPa)

Relatório de Projecto

Mariana Gomes Leal 2012138251 mleal@student.dei.uc.pt
Rosa Manuela Rodrigues de Faria 2005128014 rfaria@student.dei.uc.pt
Login Mooshak: rfaria

Departamento de Engenharia Informática

2 de Junho de 2015

Conteúdo

Lista de Tabelas

1. Introdução

Este projecto tem como objectivo o desenvolvimento para uma linguagem derivada do Pascal, o miliPascal. Esta linguagem terá regras específicas, regras essas que permitem que tudo o que for válido em miliPascal o é também em Pascal.

O enunciado deste projecto determina que apenas serão considerados como tipos de variáveis os inteiros, reais e booleanos (e literais correspondentes), podendo haver strings apenas numa instrução de writeln. As instruções de atribuição e controlo suportadas serão if-else, while e repeat. Serão ainda suportadas operações aritméticas básicas (soma, subtracção, multiplicação, divisão inteira e real, resto da divisão), comparadores simples (igual, diferente, maior, menor, maior ou igual, menor ou igual) e operadores booleanos (and, or e not).

Poderão ser declaradas funções, mas não será possível fazer declarações de funções dentro de outras funções. Estas funções poderão retornar inteiros, reais ou booleanos, e poderão ter variáveis passadas por valor ou referência.

Este compilador foi desenvolvido em quatro etapas, descritas nos capítulos seguintes:

- Análise Lexical, para detecção dos tokens inseridos no input;
- Análise Sintáctica, para detecção de produções da gramática em causa, fornecida no enunciado na forma EBNF, e construção da respectiva árvore de sintaxe abstracta;
- Análise Semântica, onde é verificada a estrutura do programa e validada a sua semântica, construindo ainda a tabela de símbolos;
- Geração de código intermédio, em que é gerado o código intermédio em llvm.

2. Análise Lexical

Para a análise lexical foi usado a ferramenta flex, em que foram definidos os tokens a utilizar em todo o compilador.

Dado que a linguagem miliPascal não é sensível às maiúsculas, todas os símbolos que representam letras foram escritos na forma [sS], sendo s o símbolo em minúsculas e o S o símbolo em maiúsculas.

O output corresponde ao nome do token em maiúsculas tal como dados no enunciado.

São pré-definidas as seguintes expressões:

- lowercase: [a-z] (não usada)
- uppercase: [A-Z] (não usada)
- number: [0-9]
- real: number+\.number+
- expoente: [eE][+-]?number+
- letter: [a-zA-Z]

2.1 Expressões regulares

A tabela seguinte contém as expressões regulares utilizadas, tendo em conta que nesta tabela apenas são escritas as expressões com minúsculas para facilidade de compreensão.

Token	**Expressão regular**
ASSIGN	:=
COLON	:
COMMA	,
DOT	.
SEMIC	;
LBRAC	(
RBRAC	)
OP1	and | or
OP2	< | > | = | <> | <= | >=
OP3	+ | -
OP4	* | / | mod | div
BEGIN	begin

DO	do
ELSE	else
END	end
FORWARD	forward
FUNCTION	function
IF	if
NOT	not
OUTPUT	output
PARAMSTR	paramstr
PROGRAM	program
REPEAT	repeat
THEN	then
UNTIL	until
VAL	val
VAR	var
WHILE	while
WRITELN	writeln

Tabela 2.1: Tokens e suas expressões regulares

2.1.1 Palavras Reservadas

As palavras reservadas incluídas foram retiradas do standard da linguagem Pascal, sendo elas as seguintes:

- abs
- arctan
- array
- case
- char
- chr
- const
- cos
- dispose
- to
- downto
- goto
- eof
- eoln
- exp
- file
- for
- get
- in
- input
- label
- ln
- maxint
- new
- nil
- of
- odd
- ord
- packed
- pack
- page
- pred
- procedure

- put
- read
- readln
- record
- set
- reset
- rewrite
- round
- sin
- sqrt
- sqr
- succ
- text
- trunc
- type
- unpack
- with
- write

2.1.2 Tratamento de comentários

Para o tratamento de comentários foi considerado um estado MULTI_ LINE_ COMMENT_ S, de modo a determinar se o comentário nunca era fechado (fim do documento) e as linhas e colunas necessárias para o output do erro.

Temos assim as seguintes expressões regulares:

- Para dar início ao comentário e passar para esse estado: "(* "{ "
- Para detectar fim de ficheiro dentro do comentário: <MULTI_ LINE_ COMMENT_ S>«EOF»
- Para detectar fim do comentário: <MULTI_ LINE_ COMMENT_ S>"*)- "} "
- Para avançar o número de linhas e colunas: <MULTI_ LINE_ COMMENT_ S>. <MULTI_ LINE_ COMMENT_ S>
  n

O conteúdo dos comentários é irrelevante para o resto do programa, logo é ignorado.

2.1.3 Tratamento de espaços e literais

Caso ocorram espaços ou tabulações é apenas incrementado o número de colunas, e no caso de haver um caracter de newline o número de colunas retorna a 1 e é incrementado o número de linhas.

Para o tratamento de strings é usada a expressão regular "'"([^\n\']|""")*"'", que prevê o uso de duas ou mais aspas seguidas sem ser terminada a string, sendo neste caso impresso o token STRING seguido da mesma. Uma string não pode conter \n, ou seja, apenas pode ser de uma linha, permitindo isto que seja usada uma expressão regular e não sejam necessários estados como para o tratamento dos comentários. A detecção do erro de uma string não terminada, ou porque há \n ou por End of File, é dada pela expressão "'"([^\n\']|""")*"'", caso em que é impressa uma mensagem de erro.

Nesta linguagem um ID é composto por letras e números, tendo obrigatoriamente de começar por uma letra. É assim dado pela expressão regular {letter}({letter}|{number})*. Já o número inteiro INTLIT é dado por pelo menos um algarismo, e só algarismos, correspondendo portanto à expressão {number}+ . Um REALLIT pode, por sua vez, ser dado por um real com ou sem expoente ou por um inteiro com expoente, sendo por isso representado por ({real}{expoente}?)|({number}+{expoente}).

2.2 Erros

Em caso de o input não pertencer a nenhuma das expressões regulares acima descritas, este irá para a última expressão criada, que representa qualquer caracter, e que imprime a mensagem de erro "illegal character"com o caracter e linha e coluna correspondentes (valores esses que foram incrementados em todas as detecções de tokens).

Pode também ser detectado erro de comentário não terminado, como mencionado acima, caso em que será impresso o erro "unterminated comment"com linha e coluna correspondentes ao início do comentário.

Um último erro refere-se a strings não terminadas, tendo esse a mensagem "unterminated string", com linha e coluna também correspondentes ao início da string.

3. Análise Sintática

Para a construção da árvore abstrata de sintaxe (análise sintáctica) foi utilizada a ferramenta yacc, com a qual foram especificadas as regras de produção da gramática e acções decorrentes.

3.1 Alterações à análise lexical

Nesta fase foi necessária a reestruturação de algumas partes do ficheiro flex da análise lexical. Dado que o que era pretendido era, não que imprimisse o token, mas sim que devolvesse o token e o seu valor, foi substituído o comando de impressão por um comando que copia o texto do input para o token e o devolve.

Houve ainda que ter em atenção o facto de os erros sintáticos deverem aparecer com a linha e coluna do início do token e não do fim, pelo que foram criadas variáveis que guardavam esse início (prev_ line e prev_ col), variáveis essas que foram declaradas como externas no yacc para por ele serem acedidas, e que eram actualizadas aquando do retorno de um token.

Finalmente, foi necessário separar os tokens OP1, OP2, OP3 e OP4 pelas operações individuais que representam, nomeadamente:

Token	**Expressão regular**
AND	and
OR	or
SMALLER	<
GREATER	>
EQUAL	=
NEQUAL	<>
SMALLEQ	<=
GREATEQ	>=
PLUS	+
MINUS	-
TIMES	*
DIVISION	/
MOD	mod
DIV	div

Tabela 3.1: Tokens de operações

3.2 Estruturas de dados

Para representar os nós da árvore foi criada uma estrutura node, que contém um char* para o valor e outro para o tipo, bem como uma lista dos filhos. Esta lista é outra estrutura que contém um array de filhos alocado dinamicamente e um inteiro com o tamanho desse array. Esta estrutura permite que em fases posteriores se possa aceder directamente por exemplo ao último elemento, já que será necessária a verificação deste em específico nalguns tipos de nós.

3.3 Tipos de nós da AST

Todos os tipos de nós designados no enunciado são da estrutura anteriormente referida. Seguem-se os nós, com respectiva descrição.

3.3.1 Programa

Program

Este nó representa a raiz do programa, tendo como filhos o seu nome, e as partes de declaração de variáveis, declaração de funções e lista de statements.

VarPart

Este nó representa a parte de declaração de variáveis, sendo composto por uma lista de declarações de variáveis, lista essa que pode ser vazia.

FuncPart

Identicamente ao nó anterior, este representa a declaração de funções, sendo também composto por uma lista de declarações de funções (de 3 tipos diferentes) que pode ser vazia.

3.3.2 Declaração de variáveis

VarDecl

Em cada um destes nós, que representa uma declaração de variáveis, está uma lista de ids, em que o último deverá ser um identificador de tipo (verificado na análise semântica), e os restantes os identificadores pretendidos para as variáveis a declarar.

3.3.3 Declaração de funções

FuncDecl

Um nó do tipo FuncDecl representa uma declaração de um protótipo de uma função, contendo portanto o seu nome, uma lista de parâmetros, e um identificador de tipo.

FuncDef

Este nó serve para representar uma declaração completa de uma função, ou seja, com o seu nome, lista de parâmetros, identificador do tipo de retorno, parte de declaração de variáveis e lista de statements, idêntica a um programa. No caso da linguagem miliPascal não são permitidas as declarações de funções dentro de funções, pelo que este nó não terá outra FuncPart.

FuncDef2

Para a continuação da declaração de uma função cujo protótipo foi declarado com FuncDecl existe este nó FuncDef2, que conterá o identificador da função e respectivas partes de declarações de variáveis e de statements.

FuncParams

A lista de parâmetros de uma função será dada no nó FuncParams, que conterá parâmetros passados por valor (Param) ou por variável (VarParam). Esta lista pode ser vazia, sendo porém necessário que exista o nó, mesmo sem filhos.

Params e VarParams

Cada parâmetro será representado por um nó Param ou VarParam (tendo em conta se é ou não passado por valor), nó esse que terá como último filho o identificador de tipo, e restantes filhos identificadores das variáveis a serem usadas dentro da função, tendo de existir pelo menos um filho além do identificador de tipo.

3.3.4 Statements

StatList

Este nó representa uma lista de statements. Caso o nó apenas tenha um filho quando este nó é retornado à regra anterior este será substituído pelo seu filho. Se não tiver qualquer filho será mantido o nó StatList sem filhos.

Assign

A instrução de atribuição é representada por um nó do tipo Assign, cujo primeiro filho será um id correspondente ao identificador de variável a que queremos atribuir um valor, e o segundo filho uma expressão de que resultará o valor pretendido.

IfElse

Um nó IfElse representa uma instrução condicional If-Then-Else, em que o primeiro filho é a condição de verificação do If, e os outros dois a "StatPart"de cada um dos outros componentes da instrução (Then e Else). Caso a instrução seja apenas do tipo If-Then é criado o mesmo tipo de nó, sendo o terceiro filho um nó do tipo StatList vazio.

Repeat

O ciclo Repeat é dado por um nó com o mesmo nome, em que o primeiro filho será a lista de statements e o segundo a condição a verificar para sair do ciclo.

While

Identicamente ao nó anterior, o nó While representa um ciclo While, em que o primeiro filho é a expressão que representa a condição a verificar para se manter no ciclo e o segundo filho é a lista de statements.

ValParam

Este nó representa uma instrução do tipo val(paramstr(a),b), sendo o seu primeiro uma expressão (sendo o "a"do exemplo) e o segundo um identificador de variável("b").

WriteLn

Por fim, a instrução de escrita é representada num nó WriteLn, cujos filhos serão as expressões correspondentes a parâmetros chamados na instrução (poderá não haver nenhum filho).

3.3.5 Operações

Nesta sub-secção são discutidos os nós que representam operadores. É aqui preciso ter em conta a distinção entre expressões simples, termos e factores. Esta explicação é feita na próxima secção, sobre as regras de produção.

Minus e Plus

Estes nós representam o sinal da expressão que os segue, tendo apenas um filho, que será um termo.

Not

O nó representativo da operação de negação terá também apenas um filho, correspondente ao factor que irá negar.

Call

A chamada de funções é representada pelo nó Call, cujo primeiro filho será o identificador da função a chamar, e os restantes filhos os parâmetros da chamada.

Add, Sub e Or

Neste caso estão representadas as operações de adição, subtração e de disjunção, tendo 2 filhos, cujo primeiro será uma expressão simples e o segundo um termo.

Mul, RealDiv, And, Mod e Div

Estes operadores aplicam-se a termos, tendo cada um dois filhos desse tipo.

Lt, Gt, Leq, Geq, Eq e Neq

Estes nós representam operações de comparação, nomeadamente menor que, maior que, menor ou igual, maior ou igual, igual e diferente. Cada um destes tem duas expressões simples como filhos.

3.3.6 Terminais

Os nós dos tipos Id, IntLit, RealLit e String representam símbolos terminais, e têm portanto no seu valor os tokens devolvidos pelo flex, não tendo quaisquer filhos.

3.4 Regras de produção

A tabela que se segue contém as regras de produção incluídas no yacc, bem como as acções correspondentes.

Símbolo	**Regra**	**Acção AST**
Prog	ProgHeading SEMIC VarPart FuncPart StatPart DOT	Cria nó "Program"
ProgHeading	PROGRAM TermID LBRAC OUTPUT RBRAC	Devolve o nome do programa (TermID)
VarPart	[VAR VarDeclaration SEMIC VarDecList]	Cria nó "VarPart"
VarDecList	[VarDecList VarDeclaration SEMIC]	Adiciona nova declaração à lista a guardar em VarPart
VarDeclaration	TermID IDList COLON TermID	Cria nó "VarDecl"
IDList	[IDList COMMA TermID]	Adiciona ID a uma lista de IDs
FuncPart	FuncPartList	Cria nó "FuncPart"
FuncPartList	[FuncPartList FuncDeclaration SEMIC]	Adiciona declaração de função a "FuncPart"
FuncDeclaration	FUNCTION TermID FormalParamList COLON TermID SEMIC FORWARD	Cria nó "FuncDecl"
FuncDeclaration	FUNCTION TermID FormalParamList COLON TermID SEMIC VarPart StatPart	Cria nó "FuncDef"
FuncDeclaration	FuncIdent SEMIC VarPart StatPart	Cria nó "FuncDef2"
FormalParamList	[LBRAC FormalParams FormalParamListList RBRAC]	Cria nó "FuncParams"

FormalParamListList	FormalParamListList SEMIC FormalParams	Adiciona parâmetro a lista
FormalParams	VAR TermID IDList COLON TermID	Cria nó "VarParam"
FormalParams	TermID IDList COLON TermID	Cria nó "Param"
FuncIdent	FUNCTION TermID	Devolve nome da função
StatPart	CompStat	Devolve bloco de statements.
CompStat	BEGINA StatList END	Devolve lista de statements do bloco em questão
StatList	Stat StatListList	Cria nó "StatList", verificando se é supérfluo.
StatListList	[StatListList SEMIC Stat]	Adiciona statement à lista.
Stat	CompStat	Devolve lista de statements
Stat	TermID ASSIGN Expr	Cria nó "Assign"
Stat	WHILE Expr DO Stat	Cria nó "While"
Stat	REPEAT StatList UNTIL Expr	Cria nó "Repeat"
Stat	IF Expr THEN Stat [ELSE Stat]	Cria nó "IfElse"
Stat	VAL LBRAC PARAMSTR LBRAC Expr RBRAC COMMA TermID RBRAC	Cria nó "ValParam"
Stat	WRITELN [WritelnPList]	Cria nó "WriteLn"
WritelnPList	LBRAC StringOpt WriteListList RBRAC	Cria a lista de parâmetros de "WriteLn"
StringOpt	Expr	Devolve expressão como parâmetro de "Writeln"
StringOpt	String	Devolve string como parâmetro de "Writeln"
WriteListList	WriteListList COMMA StringOpt	Adiciona elemento a lista de parâmetros de "WriteLn"
Expr	Simple	Devolve Simple Expression
	Simple SMALLER Simple	Cria nó "Lt"
	Simple GREATER Simple	Cria nó "Gt"
	Simple SMALLEQ Simple	Cria nó "Leq"
	Simple GREATEQ Simple	Cria nó "Geq"
	Simple EQUAL Simple	Cria nó "Eq"
	Simple NEQUAL Simple	Cria nó "Neq"
Simple	Simple MINUS Term	Cria nó "Sub"
	Simple PLUS Term	Cria nó "Add"
	Simple OR Term	Cria nó "Or"
	PLUS Term	Cria nó "Plus"

	MINUS Term	Cria nó "Minus"
	Term	Devolve termo
Term	Term TIMES Term	Cria nó "Mul"
	Term DIVISION Term	Cria nó "RealDiv"
	Term AND Term	Cria nó "And"
	Term MOD Term	Cria nó "Mod"
	Term DIV Term	Cria nó "Div"
	Factor	Devolve factor
Factor	NOT Factor	Cria nó "Not"
	INTLIT	Cria nó "IntLit"
	REALLIT	Cria nó "RealLit"
	TermID	Devolve ID
	TermID ParamList	Cria nó "Call"
	LBRAC Expr RBRAC	Devolve expressão
ParamList	LBRAC Expr ParamListList RBRAC	Cria lista de parâmetros da chamada de função
ParamListList	ParamListList COMMA Expr	Adiciona expressão à lista de parâmetros
TermID	ID	Cria nó "Id"

Tabela 3.2: Regras de produção e acções

3.4.1 Expressões e prioridades

Para definir correctamente as prioridades das expressões estas foram divididas em quatro níveis, nomeadamente a expressão, a expressão simples, o termo e o factor.

As operações com menor prioridade (a serem reduzidas mais tarde) são os comparadores lógicos, sendo as expressões o resultado destas operações.

Seguidamente temos os operadores de adição, subtracção (incluindo os operadores unários que definem o sinal de um número) e disjunção, sendo o seu resultado uma expressão simples.

As operações com mais prioridade, exceptuando os terminais, são analisadas nas regras de produção dos termos. Assim, um termo é o resultado das operações de multiplicação, divisão e conjunção.

Um factor será um terminal, uma chamada de função ou uma expressão entre parêntesis (que terá assim mais prioridade, estando já analisada quando se chega à regra de produção do factor). Um factor pode ser ainda uma negação de um factor.

O facto da prioridade ser à esquerda é definido com as regras % left no início do ficheiro yacc.

4. Análise Semântica

De forma a realizar a análise semântica foram construídas funções recursivas que, ao percorrer a árvore sintática, verificavam a validade semântica de cada nó, retornando um erro ou a ausência dele à função chamadora, ou retornando o tipo resultante da expressão, no caso em que o objectivo é avaliar se a expressão é válida e qual o seu resultado (função evalExpr).

Foi ainda construída a tabela de símbolos, com a estrutura pedida no enunciado, à qual eram adicionadas variáveis no respectivo scope à medida que se percorria a árvore. Existia assim um outer scope, com as variáveis pré-definidas, um scope do programa e um scope por cada função declarada.

4.1 Estruturas de dados

Na elaboração desta fase tornaram-se necessárias algumas alterações às estruturas de dados anteriores. Foram ainda criadas novas estruturas, não só para a construção da tabela de símbolos, mas também para armazenar informação sobre cada token (nomeadamente a linha e coluna) de forma a poder identificar as linhas e colunas dos erros correctamente.

Os tipos dos dados (type, function, integer, real, boolean, string) foram usados como enum para facilitar a leitura do programa.

Seguem-se as descrições das várias estruturas.

4.1.1 Representação de tokens - opcol

A estrutura criada para representar os tokens passou a conter os campos char* op, para guardar o token, os inteiros "col"e "line"(para guardar a coluna e a linha) e o char* nome, que guarda o tipo de token.

Esta alteração repercutiu-se no ficheiro do flex, que passou a devolver esta estrutura em vez de utilizar apenas o yylval. A alteração também se fez sentir na construção da AST, como se verá a seguir.

4.1.2 Árvore de sintaxe abstracta - node

Para armazenar a linha e coluna de cada nó foram acrescentados dois campos inteiros à estrutura node, "linha"e "coluna". Para obter estes valores foram feitas alterações no yacc, de modo a passar esses valores correctamente para as funções de construção de nós a partir dos tokens detectados.

A estrutura da lista ligada de filhos do nó manteve-se inalterada.

4.1.3 Tabela de símbolos - scope

Cada scope é descrito por cinco atributos. Este terá uma lista de variáveis (descritas na sub-secção seguinte), um inteiro com o nível (0 para outer scope, 1 para program scope e 2 para as funções) e ponteiros para o próximo da lista

(apenas usado nos scopes de funções) e para o pai. O último atributo é um ponteiro para o próximo da fila, que será usado na impressão, descrita mais à frente.

4.1.4 Símbolo da tabela - variavel

Existem 6 campos para descrever cada variável. Os quatro primeiros são os que aparecerão impressos na tabela e que estão definidos no enunciado: nome, tipo, flag e valor. Os restantes dois atributos são um ponteiro para o seguinte na lista de variáveis, e um ponteiro para o scope, caso a variável seja um identificador de função.

No caso das funções há ainda que notar que foi usada a flag para determinar se a função já tinha sido completamente definida. Assim, a flag será "0"se foi declarado apenas o protótipo da função, "1"se a função foi declarada por protótipo e posteriormente definida, e "2"se foi completamente definida através de um FuncDef.

4.2 Impressão da tabela de símbolos

Para imprimir a tabela de símbolos a árvore de tabelas é percorrida em pós-ordem, sendo que sempre que é impressa uma variável que identifica uma função a respectiva função é colocada numa fila FIFO para ser impressa, sendo portanto necessário o já mencionado ponteiro para o próximo da fila.

4.3 Erros considerados

Erro	**Quando ocorre**	**Exemplo**

Cannot write values of type	Quando se chama um símbolo do tipo type numa instrução writeln	program ex(output);
		begin
		writeln(integer);
		end.

Function identifier expected	Numa chamada de função o identificador usado existe mas não é de uma função	program ex(output);
		var x,y:integer;
		begin
		y:=x(1,2);
		end.

Wrong number of arguments	Numa chamada a uma função o número de argumentos é diferente do número de parâmetros necessário	program ex(output);
		function a(b,c:boolean): integer;
		forward;
		begin
		writeln(a(true));
		end.

Incompatible type for argument	Numa chamada de função o argumento numa dada posição é do tipo errado	program ex(output);
		var x:integer;
		function a (b:integer):integer;
		forward;
		begin
		x:=a(true);
		end.

Incompatible type in assignment	Numa atribuição o tipo resultante da expressão do lado direito não corresponde ao tipo da variável do lado esquerdo	program ex(output);
		var x: integer;
		y:boolean;
		begin
		x:=y;
		end.

Incompatible type in statement	Num statement do tipo if, while ou repeat a expressão que é avaliada como condição não é booleana.	program ex(output);
		begin
		if (2+3) then ;
		end.

Symbol already defined	Quando se tenta declarar uma variável ou função num scope em que já existe um símbolo com o mesmo nome. Inclui o caso em que está declarado o protótipo da função e se tenta definir por completo uma função com FuncDef.	program ex(output);
		var x,x:integer;
		begin
		end.

Symbol not defined	Quando se tenta aceder a um símbolo que não é encontrado na tabela de símbolo em nenhum dos scopes em que é chamado (função, programa ou outer). Inclui o caso em que se tenta definir uma função através de um FuncDef2 e esta não foi anteriormente declarada.	program ex(output);
		begin
		x:=2;
		end.

Type identifier expected	Ocorre numa declaração de variáveis ou de função se o id correspondente ao tipo da variável não é um símbolo do tipo "type".	program ex(output);
		var integer:real;
		x:integer;
		begin
		end.

Variable identifier expected	Este erro aparece caso o segundo filho de uma instrução valparam não seja uma variável ou se é chamada uma função com um argumento não variável que o devia ser por estar declarado como varparam. Apenas aparece se o símbolo for encontrado e não for uma variável, ou seja, se tiver flag "constant"ou se for função (quando não está no scope dessa função)	program ex(output);
		begin
		val(paramstr(2),3);
		end.

Operator cannot be applied to type(s)	Em caso de qualquer dos operadores descritos no capítulo anterior serem aplicados a tipos incompatíveis, tanto com o operador como entre eles (caso seja mais que um). A compatibilidade de tipos encontra-se descrita no documento ISO do Pascal standard.	program ex(output);
		a:integer;
		begin
		a:=2*true;
		end.

Tabela 4.1: Erros da análise semântica

4.3.1 Erros não considerados

Além dos erros acima descritos, pode ainda ocorrer um erro, cuja descrição não foi feita no enunciado, mas que seria ainda assim um erro semântico: a declaração de um protótipo de uma função sem que esta seja posteriormente definida por completo. Dado que não havia indicação do que fazer nesse caso não foi implementada a detecção desse erro, sendo porém simples detectá-la, bastando no final percorrer a tabela de símbolos verificando se existiam funções declaradas com flag "0"(ainda não definidas).

4.4 Limpeza da memória

Na elaboração desta meta foi acrescentada a limpeza da memória alocada no fim de todo o processo, processo verificado com a ferramenta valgrind. A limpeza é feita com funções e formas de percorrer as estruturas idênticas às de percorrer a árvore de sintaxe abstracta e a tabela de símbolos. No final é também é também usado o comando yylex_ destroy(), para que o yacc liberte toda a memória que alocou (que constitui grande parte da memória alocada).

5. Geração de código

A meta da geração de código consistiu em produzir um conjunto de instruções correspondentes à representação intermédia LLVM IR para cada nó da nossa AST. O modo como esta é percorrida é exactamente igual ao realizado durante a análise semântica, mantendo a mesma lógica recursiva.

5.1 Estruturas de Dados

Para a elaboração desta fase optámos não por criar novas estruturas de dados para cada variável mas sim por editar a struct já existente *_variavel*, ao qual foi acrescentado um campo para a respectiva representação da variável no código llvm a ser gerado (por exemplo, @a ou %a).

5.2 Variáveis, Funções e Strings Universais

Independentemente do programa cujo código queremos gerar, existe um conjunto de variáveis,funções e strings que são sempre geradas. Assim sendo, o cabeçalho de qualquer ficheiro .ll gerado será deste tipo:

```
declare i32 @printf(i8*, ...)
declare i32 @atoi(i8*)

@str.integer_str = private unnamed_addr constant
                           [3 x i8] c"\%d\00"
@str.double_str = private unnamed_addr constant
                           [6 x i8] c"\%.12E\00"
@str.false_str = private unnamed_addr constant
                           [6 x i8] c"FALSE\00"
@str.true_str = private unnamed_addr constant
                           [5 x i8] c"TRUE\00"
@str.par_str = private unnamed_addr constant
                           [2 x i8] c"\0A\00"

@.argc = common global i32 0
@.argv = common global i8** null
@.true = common global i1 1
@.false = common global i1 0
```

Todas as declarações anteriores são úteis em diferentes fases da geração do código intermédio, nomeadamente na impressão do output quando é executado

o ficheiro .ll e na leitura de inputs do stdin. Para a impressão recorremos à função printf e aos formatos explicitados nas strings - %d para inteiros, %.12E para reais e 'TRUE' ou 'FALSE' para os booleanos. As variáveis @.argc e @.argv serão onde guardaremos o número de parâmetros lidos do stdin e os próprios parâmetros, respectivamente. Por fim, são definidas as variáveis @.true e @.false para auxiliar em eventuais casos de assignmente directo de booleanos (por exemplo a:=true);

Para além destas declarações são ainda definidas mais duas funções que também estão sempre presentes. São elas a função paramcount, cujo objectivo é retornar o número de parâmetros inseridos no stdin, e a função main, que corresponde ao *program* dos ficheiros Pascal. É nesta função que é feito o store de valores nas variáveis @.argc e @.argv, como expresso de seguida:

```
define i32 @..paramcount()
{
%.1 = load i32* @.argc
%.2 = sub i32 %.1, 1
ret i32 \%.2
}

define i32 @main(i32 \%-argc, i8** \%-argv)
{
%.ret = alloca i32
%.3 = alloca i32
%.4 = alloca i8**
store i32 %-argc, i32* %.3
store i8** %-argv, i8*** %.4
%.5 = load i32* %.3
store i32 %.5, i32* @.argc
%.6 = load i8*** %.4
store i8** %.6, i8*** @.argv

; StatList start
; StatList end

ret i32 0
}
```

5.3 Atribuição do Nome de Variáveis

Para além do standard de declarar variáveis globais com o símbolo @ e variáveis locais com o símbolo %, tivemos de criar os nossos próprios standard para atribuição do nome de variáveis para ter a certeza de que não existiriam conflitos de qualquer tipo.

No geral, todas as variáveis e funções são representadas com pelo menos um '.' no início, que sabemos ser inválido em Pascal. As **variáveis globais** declaradas no VarPart do *program* são precedidas ainda pelo símbolo _, para lidar com o facto de ser possível criar uma variável global com nomes já escolhidos por nós antes, como 'argv' ou 'printf'.

Por causa do problema semelhante que ocorre para as **funções** , optámos por representar as funções precedidas também de um '.' (excepção do paramcount universal, que é definido com dois '.'). Torna-se assim possível declarar funções com os nomes 'printf' e 'paramcount', por exemplo. De notar que no caso de redefinição da função 'paramcount' é a nova que deve ser utilizada, tornando-se a original obsoleta.

Para as **variáveis locais** não optámos por nenhuma alteração, uma vez que localmente não há risco de existirem variáveis com nomes iguais. Ao longo de toda a geração de código fazemos um uso extensivo de **variáveis temporárias**. O método *getNextTempVariable* é responsável por retornar a próxima variável temporária a usar (por exemplo, %.12), pronta a ser usada. Sempre que uma é retornada é incrementado o número da variável. Recomeçar o contador sempre que entrássemos na geração de código de uma nova função teria sido um melhoramento interessante que, infelizmente, não chegámos a implementar.

5.4 Declaração de Funções e Retorno de Resultados

Como mencionado anteriormente, não deve ser válida a declaração de uma função sem a sua posterior definição. Assim sendo, as declarações das funções em código intermédio apenas são feitas quando é encontrado um nó com definição de função (nomeadamente FuncDef e FuncDef2). Assim, uma vez encontrado esse nó, a declaração da função e respectivas variáveis é feita a partir da tabela de símbolos. A primeira parte da declaração de uma função é semelhante à da função main utilizada para representar o *program*. Uma vez que tem de ser possível modificar os parâmetros da função dentro dela, os mesmos devem ser armazenados na pilha. O exemplo de declaração seguinte:

```
function funcao( a : integer ; d: boolean):real;
```

produz o seguinte código:

```
 define double @.funcao(i32 %a, i1 %d)
{
%.ret = alloca double
%.3 = alloca i32
%.4 = alloca i1
store i32 %a, i32* %.3
store i1 %d, i1* %.4
```

A variável %.ret é comum a todas as funções, e é nela que se guarda o valor final associado à função, também através de um *store*.

5.4.1 Passagem de Parâmetros por Valor e Referência

Nas chamadas de funções, dado que os parâmetros podem ser passados de duas formas, foram usadas duas estratégias. No caso dos parâmetros passados por valor é colocado directamente na chamada o nome da variável temporária que contém o valor avaliado através da avaliação das expressões e o respectivo tipo (convertendo para double se for esse o tipo do parâmetro alvo. Quando os parâmetros são passados por referência a variável é procurada na tabela, sendo enviado o nome correspondente ao seu endereço de memória.

5.5 Acesso a Parâmetros do stdin

Já referimos que os parâmetros passados por linhas de comandos são armazenados na variável global @.argv e o seu número total em @.argc. Em Pascal, a leitura desses parâmetros é feita aquando da utilização do operador *val(paramstr(n),x)*, onde n corresponde ao n-ésimo argumento e x à variável onde o guardar. Uma vez que em argv[0] é guardado o nome do executável, podemos fazer o mapeamento directo entre n e o índice do array argv. À partida só estamos aceitar inputs do tipo inteiro, por isso é necessário chamar a função @atoi para o parâmetro lido, para guardarmos realmente um inteiro.

Exemplificando, o seguinte código Pascal:

```
begin
    c := paramcount;
    val(paramstr(2),b);
end.
```

levaria à próxima produção:

```
 ; StatList start

; Assign start
%.7 = call i32@..paramcount()
store i32 %.7, i32* @_c
; Assign end

; ValParam start
%.9 = add i32 0, 2
%.14 = load i8*** @.argv
%.11 = getelementptr inbounds i8** %.14, i32 %.9
%.12 = load i8** %.11
%.13 = call i32 @atoi(i8* %.12)
store i32 %.13, i32* @_b
; ValParam end

; StatList end
```

5.6 Statements If-else, While e Repeat

Para gerar devidamente o código intermédio para este tipo de statements tivemos de recorrer a um sistema de labels semelhante ao das variáveis temporárias. Assim sendo, desenvolvemos a função 'getNextLabelName' que nos devolve a próxima label a ser utilizada. Os exemplos para estes casos são auto-explicativos, por isso aqui fica o código intermédio gerado para o seguinte código Pascal:

```
begin
    if (a) then
        writeln(b)
    else
        writeln(a);

    while ( a ) do
        begin
            a := b;
         end;

    repeat
        c := c - 1;

    until ( c = 0 );
end.
```

Para o statement If-else:

```
; IfElse start
%.7 = load i1* @_a
br i1 %.7, label %.label1, label %.label2

.label1:
; WriteLn start
%.8 = load i1* @_b
(...)
; WriteLn end
br label %.label3

.label2:
; WriteLn start
(...)
; WriteLn end

br label %.label3
.label3:
; IfElse end
```

Para o statement while:

```
; While start
br label %.label10

.label10:
%.15 = load i1* @_a
br i1 %.15, label %.label11, label %.label12

.label11:
; Assign start
%.16 = load i1* @_b
store i1 %.16, i1* @_a
; Assign end
br label %.label10

.label12:
; While end
```

Para o statement Repeat-until:

```
 ; Repeat start
br label %.label13

.label13:
; Assign start
%.19 = load i32* @_c
%.20 = add i32 0, 1
%.21 = sub i32 %.19, %.20
store i32 %.21, i32* @_c
; Assign end

%.23 = load i32* @_c
%.24 = add i32 0, 0
%.25 = sitofp i32 %.23 to double
%.26 = sitofp i32 %.24 to double
%.27 = fcmp oeq double %.25, %.26
br i1 %.27, label %.label14, label %.label13

.label14:
; Repeat end
```

5.7 Impressão para o stdout

Já foi referido no início que são desde logo declaradas as strings que permitem imprimir os valores numéricos e booleanos, bem como a própria função que o permite fazer, o @printf. Sendo uma função como as outras, o call funciona da mesma maneira. Se quisermos imprimir as variáveis a, b e c, onde *a:=10*, *b:=5.5* e *c:=true* teríamos o seguinte código:

```
; WriteLn start
%.13 = load i32* @_a
call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
([3 x i8]* @str.integer_str, i32 0, i32 0), i32 %.13)
; WriteLn end

; WriteLn start
%.15 = load double* @_b
call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
([6 x i8]* @str.double_str, i32 0, i32 0), double %.15)
; WriteLn end

; WriteLn start
%.17 = load i1* @_c
%.19 = icmp eq i1 %.17, 1
br i1 %.19, label %.label1, label %.label2

.label1:
call i32 (i8*, ...)* @printf(i8* getelementptr
           ([5 x i8]* @str.true_str, i32 0, i32 0))
br label %.label3

.label2:
call i32 (i8*, ...)* @printf(i8* getelementptr
           ([6 x i8]* @str.false_str, i32 0, i32 0))
br label %.label3

.label3:
; WriteLn end
```

Feito isto, restava ainda resolver a impressão de strings. Essa situação não era tão trivial uma vez que as mesmas devem estar declaradas fora das funções, tal como estavam as específicas para os outros tipos de dados a imprimir. Uma vez que não é a relevante a ordem pela qual surgem as instruções de impressão e as declarações das strings, optámos por criar uma lista ligada de Strings que era percorrida no fim de gerar o resto do código, ficando a lista de declarações de strings no fim do ficheiro .ll. Por exemplo, o seguinte código em Pascal:

```
program gcd2(output);
begin
    writeln('Compiladores ''2015''');
    writeln('Mooshak');
end.
```

produziria o seguinte output:

```
define i32 @main(i32 %-argc, i8** %-argv)
{
(...)
; StatList start

call i32 (i8*, ...)* @printf(i8* getelementptr
           ([20 x i8]* @.str1, i32 0, i32 0))

call i32 (i8*, ...)* @printf(i8* getelementptr
            ([8 x i8]* @.str2, i32 0, i32 0))

; StatList end

ret i32 0
}

@.str1 = private unnamed_addr constant
             [20 x i8] c"Compiladores '2015'\00"
@.str2 = private unnamed_addr constant
             [8 x i8] c"Mooshak\00"
```

6. Conclusão

Num período aproximado de três meses fomos capazes de desenvolver um compilador totalmente a partir do zero para a linguagem mili-Pascal que, para além de ter sido entregue atempadamente, cumpria também todos os requisitos que foram apresentados ao longo das metas. Sendo a primeira vez que lidamos com o campo de compiladores estreámo-nos nas ferramentas do flex e do yacc e enriquecemos ainda mais as nossas bases de LLVM e de geração de código.

No entanto, o nível de aprofundamento do nosso conhecimento ao longo do projecto estendeu-se também aos nossos próprios métodos de projectar,programar e trabalhar em equipa. Não só desenvolvemos um compilador funcional como também fomos capazes de aplicar boas práticas de engenharia de software que nos permitem não só apresentar um código 'limpo' e legível mas também proceder a eventuais alterações que sejam necessárias sem que tal acarrete danos maiores para o projecto inteiro.

Olhando para trás agora, as decisões que tomámos na altura seriam muito semelhantes às que tomaríamos após todo este processo. Se tivéssemos de mudar alguma coisa seria a limpeza da memória que teve de ser adicionada *a posteriori* e mesmo assim não ficou concluída. Também no processo da geração do código intermédio tivemos problemas de última hora relacionados com a alocação de memória na pilha, que nos obrigou a uma reestruturação de alguns métodos. No fim tudo ficou resolvido, mas teria sido mais fácil se tivéssemos pensado nessas mesmas consequências um pouco mais cedo. Para concluir, podemos dizer que ficámos muito satisfeitas com o nosso trabalho, não só porque realizamos tudo aquilo a que nos auto-propusemos mas também porque terminámos a saber que estamos agora aptas a fazer um compilador de mili-pascal e, eventualmente, compiladores de linguagens mais avançadas.