COMP

Relatório final de Compiladores 14/15

Resumo: Este trabalho tem como objectivo o desenvolvimento de um compilador para a linguagem miliPascal. Para tal, o input é tratado passando por várias etapas, culminando com a geração de código LLVM-IR, "low-level virtual machine". O código em miliPascal é analisado (Lex e Yacc) e guardado representativamente em estruturas, em forma de árvore, para finalmente ser representado de diferentes formas.

• U  C •

Trabalho elaborado por:
Miguel Veloso - 2011152740
Grupo LCMV

Indíce

1 - Introdução

Este trabalho foi produzido no âmbito da cadeira de Compiladores, e tem como objectivo a compilação de código fonte respectivo à linguagem **miliPascal**. Para tal foi necessária a aprendizagem sobre **Lex, Yacc e LLVM-IR.**

O **Lex** é uma ferramenta de análise lexical que tem como objectivo receber e "formatar" o código (ou texto) de input, segundo as regras lexicais definidas retornando tokens necessárias para uma posterior análise sintática (Yacc). Para tal são definidas todas as "palavras" aceites pelo lex, podendo serem emitidos erros caso essas regras não sejam cumpridas. O mesmo, no entanto, não é capaz de fazer uma análise sintática do código a ser compilado.

O **Yacc**, por sua vez, complementa o Lex na medida em que é capaz de definir uma gramática sobre a qual o código fonte tem de estar "formatado". O mesmo pode emitir erros sintáticos, caso a gramática não seja cumprida. O Yacc necessita do Lex para receber as tokens que estam representadas na gramática. Os dois, após compilados e utilizados em conjunto formam uma ferramenta poderosa e eficiente de análise Lexical e Sintática.

As ferramentas anteriormente apresentadas fizeram parte e foram o núcleo central das duas primeiras metas deste projecto, sendo que as outras duas foram mais dedicadas a construção de tabelas, reprodução de código LLVM e tratamento de dados.

LLVM-IR é uma linguagem de baixo nível, semelhante ao assembly, utilizada para a reprodução do código fonte. É uma linguagem muito particular dado que não estamos habituados a utiliza-la e este é o primeiro contacto que temos com a mesma.

2 - LEX

O lex é uma ferramenta de análise lexical, que serve para receber e "formatar" o código (ou texto) de input, segundo as regras lexicais definidas, complementando o Yacc (ver página 5), fornecendo-lhe as "tokens" necessárias.

Este encontra-se subdividido em três partes: definições, regras e métodos. Nas definições encontramos a declaração das "strings" esperadas no código fonte (desde métodos da linguagem a números inteiros, etc.); nas regras dispomos de acções que são desplotadas mediante o que é analisado pelo lex. Essas acções são por exemplo: o calculo da linha e coluna em que nos encontramos do código fonte e o retorno de uma token para que o Yacc possa proceder ao seu trabalho; por último temos os métodos que podemos implementar em caso de necessidade. O lex trabalha com a linguagem C.

Seguidamente apresento as categoria que considerei importantes para o desenvolvimento deste projecto:

```
INTLIT      [0-9]+
REALLIT     [0-9]+(("."[0-9]+)|(("."[0-9]+)*"e"(("+"|"-")*[0-9]+)+))+
STRING      """("""|[^\n'])*"""
NOTSTRING   """("""|[^\n'])*
ID          [a-zA-Z]+([0-9a-zA-Z])*
```

Estas são as categorias que eu considero essenciais para o bom desenvolvimento da análise lexical, já que todas as restantes estão directamente relacionadas com aspectos da linguagem a ser compilada. Estas categorias representam inteiros, strings, doubles e nomes de variáveis que o programador possa utilizar no seu código fonte. Além das mesmas, criei uma categoria chamada NOTSTRING,

que encontra “strings” mal construídas, apresentando posteriormente um erro ao utilizador do compilador.

Além do grupo apresentado em cima, ainda foram definidos outros, que, tal como referi anteriormente, estão directamente relacionadas com a linguagem a ser analisada.

ASSIGN	":="
BEGIN1	"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"
AND	"and"

OR "or"
OP2 "<"|">"|"="|"<>"|"<="|">="
OP3 "+"|"-"
OP4 "*"|"/"|"mod"|"div"

O analisador lexical, alem da interpretação do código fonte e posterior ligação ao analisador semântico, apresenta erros lexicais como comentários não terminados, strings não terminadas e caracteres ilegais. Como referi, todos estes erros pertencem à categoria de erros lexicais.

Por fim, gostaria de acrescentar que o lex após compilado é executado pelo analisador sintático (yacc), passando por um processo em que são requisitadas tokens até que o que o analisador léxico identifique o fim do fluxo ou quando o analisador sintático identifica um erro gramatical.

Exemplo da passagem de uma token para o analisador sintático:

```
{ASSIGN}        {coluna+=strlen(yytext);return ASSIGN;}
```

Esta token está definida anteriormente no bloco de regras do analisador lexical.

3 - Yacc

O Yacc é o analizador sintático, ficando responsável por manter a coerência sintática de um determinado código fonte, isto é "obriga" a que o utilizador, que usa o compilador, escreva da forma predefinida pelo criador do mesmo. No Yacc fica, então, predefinida uma gramática formal, e este ao receber o código fonte e verificar que não existem erros gramaticais, passa os dados para a parte do compilador escrita em C, onde são armazenados e transformados os dados.

O Lex e o Yacc trabalham os dois em conjunto, já que o Lex não consegue fazer uma análise sintática ao input, apenas se foca nas expressões regulares (é limitado a máquinas de estado finito); por sua vez o Yacc necessita de tokens e não consegue ler a partir de um input, daí ser importante a comunicação entre os dois.

Exemplos da gramática formal definidos neste trabalho:

ProgramHeading : PROGRAM ID LBRAC OUTPUT RBRAC

VarPart : VAR VarDecl SEMIC AditionalVarPart

AditionalVarPart : VarDecl SEMIC AditionalVarPart

VarDecl : IdList COLON ID

IdList : ID AditionalIds

AditionalIds : COMMA ID AditionalIds

Statement : CompStat
| IF Expression THEN Statement ELSE Statement
| IF Expression THEN Statement
| WHILE Expression DO Statement
| REPEAT StatList UNTIL Expression
| VAL LBRAC PARAMSTR LBRAC Expression RBRAC COMMA ID RBRAC
| ID ASSIGN Expression
| WRITELN
| WRITELN Writeln

Como podemos ver nos exemplos anteriores, é possível, através das tokens recebidas do Lex, construir uma gramática a ser respeitada pelo código fonte. Associadas à gramática estão as funções de inserção que guardam os dados necessários nas estruturas predefinidas. A gramática funciona como um grafo, sendo que que cada estado pode ser final, recursivo ou levar a ou estado gramatical.

As tokens estão definidas antes da gramática ser construída e seguidamente vamos poder ver alguns exemplos:

%token <id> ID
%token <id> INTLIT
%token <id> REALLIT
%token <id> STRING
%token ASSIGN
%token BEGIN1
%token COLON
%token COMMA
%token DO
%token DOT
%token ELSE

É desta forma que as tokens são definidas no Yacc. Há 7 (sete) tokens que são passadas ao yacc pelo meio de um ponteiro para o endereço de memória onde se encontra uma string, os id's que representam os nomes das variáveis, os intlit que representam inteiros, os reallit que representam doubles, as string, e 3 (três) tipos de operações que incluem várias sub operações.

Por sua vez, estas tokens predefinidas (não todas) têm uma precedência e associatividade. Existem três tipos: %nonassoc, %left, %right. Neste projecto foram definidas algumas regras de precedência e seguidamente estão exemplos das mesmas:

%nonassoc THEN
%nonassoc ELSE
%nonassoc IF
%left LBRAC RBRAC
%left OP4 AND
%left OR OP3
%left OP2
%right ASSIGN
%right NOT

Tal como o Lex, o Yacc encontra-se dividido em três fazes: declaração de varáveis e definições, regras gramaticais e acções (é utilizado também C para a execução das mesmas).

De referir que na primeira parte do yacc, estão definidas variáveis, maioritariamente respectivas a estruturas que posteriormente vão ser visitadas pelo analisador C, onde vão ser armazenadas todas as informações recolhidas. São também recolhidas variáveis externas, respectivas ao Lex, como a coluna e a linha do código fonte (onde nos encontramos no momento da leitura das tokens). É nesta parte que também está representada a função

yyerror, responsável pela emissão de erros de gramaticais, sempre que a gramática predefinida não for respeitada. É por isso que é necessário o acesso às variáveis da linha e coluna emitidas pelo Lex.

Exemplos de variáveis definidas:

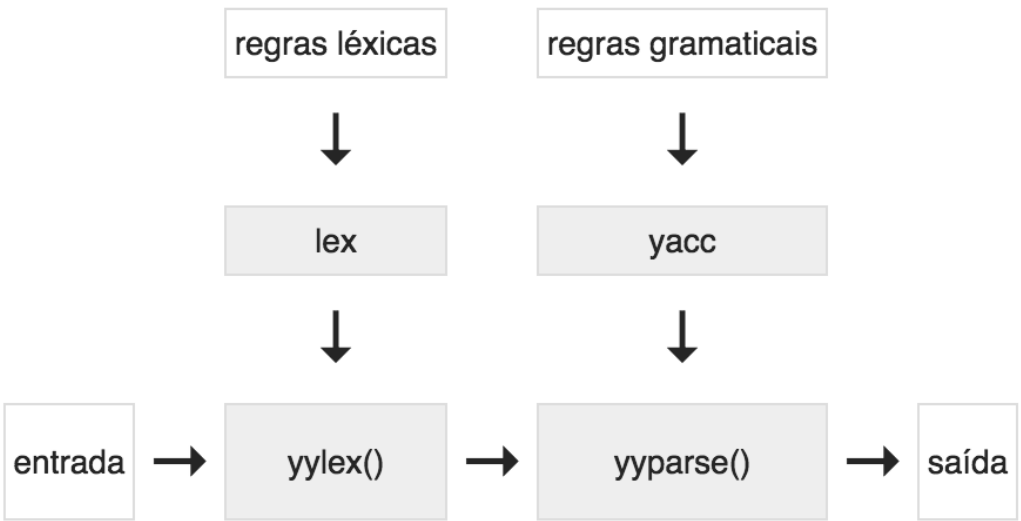
extern int linha;
extern int coluna;
extern char * yytext;

void yyerror(char * s);
int yylex();

start * program_start = NULL;
symbol_table * symbols = NULL;

Na parte final do Yacc é onde são executadas as acções, como por exemplo pedir ao analisador C que analise os error semânticos, ou que construa uma árvore representativa, ou que imprima a tabela de símbolos ou que represente o código fonte em LLVM-IR.

A imagem seguinte representa a interacção Lex - Yacc.



4 - Árvore AST

Esta fase do projecto é uma representação do código fonte após ser pré analisado pelo Lex, analisado pelo Yacc e guardado pelo C. Quero com isto dizer que para a representação da arvore de sintaxe abstrata é necessário o código de input ser aceite pelo Lex e não ter erros lexicais, ser aceite pelo Yacc e a gramática do mesmo estar bem formulada. No entanto pode não estar correta semânticamente (ver página 14).

Para esta fase foi necessária a formulação de estruturas que suportem a informação recebida e que seja de fácil pesquisa. Para tal optei por criar uma estrutura por cada tipo gramatica, isto é uma estrutura para os statements, outra para as expressions, etc.

Estas estruturas foram implementadas aquando da implementação desta fase e irão ser fulcrais para o resto do projecto, como vamos poder constatar daqui em diante.

Exemplos de estruturas implementadas:

```
struct no_stat{
   char * id;
   char * operation_type;
   comp_stat * node_comp_stat;
   expression * node_expr;
   stat * node_stat;
   stat * node_stat2;
   stat_list * node_stat_list;
   writeln_list * node_writeln_list;
   code_position * posicao;
```

```
  /*
    1 - CompStat
    2 - IF
    3 - IF - ELSE
    4 - WHILE
    5 - REPEAT
    6 - VAL PARAMSTER
    7 - ASSIGN
    8 - WRITELN
  */
};

struct no_id_list{
   char * id;
   aditional_id_list * next_id;
   code_position * posicao;
};

struct no_aditional_id_list{
   char * id;
   aditional_id_list * next_id;
   code_position * posicao;
};
```

Com a informação guardada em estruturas deste género, torna-se fácil imprimir a árvore de sintaxe abstrata, tendo apenas que percorrer o "flow" das estruturas até ao estado final, em que todos os ponteiros estão a NULL e não há mais informação disponível.

Para a impressão da informação foram criadas cerca de 30 funções que se chamam entre si tal como as estruturas estudadas pelas mesmas.

5 - Tabela de Símbolos

Nesta fase foram criadas duas novas estruturas que guardam toda a informação relativa às variáveis globais ou representadas nas funções, representadas no código fonte.

As estruturas são:

```
struct no_symbol_table{
   char * type;
   symbol_entry * node_symbol_entry;
   symbol_table * next_symbol_table;
   code_position * posicao;
};

struct no_symbol_entry{
   char * id;
   char * type;
   char * second_type;
   symbol_entry * next_symbol_entry;
   code_position * posicao;
};
```

A primeira guarda ponteiros para uma lista ligada para as variáveis globais ou apresentadas numa dada função, bem como o tipo da função. A segunda estrutura guarda todas as variáveis e os seus tipos e, inclusive, o nome da função em que se encontra.

Para se guardar toda esta informação foram desenvolvidas funções que percorrem novamente as estruturas principais todas, procurando guardar toda a informação referida anteriormente.

Nesta fase foi também desenvolvida a captação de erros semânticos, impedindo, caso seja encontrado algum, de que a "compilação" do código fonte continue e que seja imprimida a tabela simbólica (ver página 14).

Por último e ainda relativo a esta fase, foi necessário o desenvolvimento de uma função auxiliar que passasse todos os nomes de variáveis para "lower case".

6 - Erros Semânticos

A implementação deste ponto foi feita na mesma meta que a anterior e estão interligadas. Ao verificar todo o código para a criação e impressão da tabela de símbolos, são emitidos erros quando as regras e tipos não respeitados.

Os erros semânticos correspondem aos erros de má construção programática, quando as variáveis não correspondem ao tipo que lhes é associado ou quando a função retorna um tipo diferente do seu, etc.

Exemplos de erros implementados:

- Cannot write values of type <type>
- Function identifier expected
- Incompatible type in assigment to <token> (got <type>, expected <type>)
- Incompatible type in <statement> statement (got <type>, expected <type>)
- Symbol <token> already defined
- Symbol <token> not defined

Ao mesmo tempo que são verificados e armazenados os símbolos, todo o código é percorrido, por ordem, à procura de possíveis erros semânticos que possam existir. No caso de algum ser encontrado, é imprimido e o programa termina abruptamente. Desta forma, a análise não continua e por consequencia a tabela de símbolos não é imprimida.

Este processo comporta-se da mesma forma que foi percorrido todo o código para a implementação das árvores de sintaxe abstrata, com a diferença que guarda variáveis nas estruturas representadas no ponto anterior e emite erros em caso de existir um erro semântico.

Neste ponto, não tive a pontuação que esperava obter pois penso que o número da coluna do erro que estava ser impressa estava errada e não consegui implementar alguns erros que eram pedidos no enunciado.

7 - Geração de código LLVM

Esta fase corresponde à ultima meta pedida no âmbito da cadeira de Compiladores. Foi implementada de forma muito semelhante à 3º meta, árvore de sintaxe abstrata, imprimindo código em LLVM-IR, uma linguagem de baixo nível, semelhante ao assembly.

Utilizei como base as funções de pesquisa já implementadas anteriormente e, com o auxilio da libraria Clang para entender o funcionamento da linguagem (conversão .c - .ll), foi possível reproduzir o código fonte em LLVM.

Foi a meta mais custosa de todas dado que LLVM é uma linguagem um pouco diferente ao que programo habitualmente, mas foi possível implementar grande parte exceptuando os statements de IfElse, While e Repeat.

Foi necessário o uso dos dois tipos de estrutura desenvolvidos, tanto a que contem a informação toda relativa ao código, como a que contém apenas as variáveis globais ou correspondentes a cada função. Assim, foi possível uma pesquisa mais rápida pelas variáveis declaradas e tipos que lhes correspondem.

Um exemplo do uso "base de dados" está na função get_type(char * id), que desta forma me dá acesso ao tipo da variável sem que tenha de percorrer toda o grafo novamente.

Para ser possível o acesso aos parâmetros de entrada, tratei em LLVM o código global do miliPascal como se fosse a função *main* com **argc** e **argv**, acedendo aos dados aquando da chamada val(paramstr(i), j).

De resto todas as funções e variáveis do LLVM são tratadas como miliPascal com a devida sintaxe.

8 - Conclusão

Neste trabalho aprendi novos conceitos e a trabalhar com ferramentas poderosas com o Lex e o Yacc, associando-os a uma linguagem com um poder de execução excelente comparado com outras linguagens bastante utilizadas.

Foi me possível implementar e ter noções de como reproduzir um compilador. A combinação dos Lex e o Yacc mostrou ser muito eficiente. Considero que a aprendizagem feita sobre os mesmos representa o ponto fulcral deste trabalho, assim como o contacto com LLVM.

Não tive as notas que gostaria mas termino este projecto com a consciência de dever cumprido, pelo menos no que toca a aprendizagem de conceitos e mecânica dos mesmos. Tive bastantes dificuldades e gostaria de obter um resultado melhor, mas não me foi possível pois alem de estar envolvido em cadeiras de mestrado (o que não é desculpa), fui deixado pelo meu colega de grupo a fazer o trabalho sozinho, e quero já antecipar que a defesa será feita apenas por mim.

Gostaria de acrescentar que esta cadeira foi, a par com outras duas, a minha favorita no curso pois deu-me grandes noções de como funciona um IDE e como é compilado um código de uma linguagem. Penso mesmo até que seria bom, no futuro, estar relacionado com um projecto deste tipo.

9 - Anexos

COMP_Meta1.zip

COMP_Meta2.zip

COMP_PosMeta2.zip

COMP_Meta3.zip

COMP_Meta4.zip